CapablePtrs: Securely Compiling Partial Programs using the Pointers-as-Capabilities Principle (Technical Report)

Akram El-Korashy[1], Stelios Tsampas[2], Marco Patrignani[3],
Dominique Devriese[4], Deepak Garg[1] and Frank Piessens[2]
[1] MPI-SWS, Germany;
email {elkorashy, dg}@mpi-sws.org
[2] imec-Distrinet, KU Leuven, Belgium;
email name.surname@cs.kuleuven.be
[3] CISPA, Germany;
email marco.patrignani@cispa.saarland
[4] Vrije Universiteit Brussel, Belgium;
email dominique.devriese@vub.be

Contents

1 The target language (CHERIExp) 10
1.1 Values, expressions, and commands . . . 10
1.2 Target setup, and initial and terminal states . . . 16
1.3 Memory Reachability . . . 19
1.4 Summary of target language features . . . 70

2 A source language (ImpMod) with pointers and modules 71
2.1 Program and module representation, and well-formedness . . . 71
2.2 Values, expressions, and commands . . . 71
2.3 Program state . . . 73
2.4 Initial, terminal and execution states . . . 76
2.5 Memory Reachability . . . 87

3 Compiling pointers as capabilities (ImpMod to CHERIExp) 95
3.1 Whole-program compiler correctness . . . 96
3.2 Compositionality: linking-and-convergence-preserving homomorphism . . . 140

4 A sound trace semantics for CHERIExp 150
4.1 Soundness . . . 159

5 A complete trace semantics for ImpMod 166
5.1 Completeness using back-translation . . . 171

6 Security guarantee about the compiler: full abstraction 173
6.1 Lifting compiler forward and backward simulation to trace semantics . . . 176
6.2 Strong and weak similarity . . . 180
6.3 Stack similarity (successor-preserving isomorphism) . . . 181
6.4 Trace-Indexed Cross-Language (TrICL) simulation relation . . . 245

7 Corollaries for free **250**
7.1 Completeness of the trace semantics of **CHERIExp** . . . 250
7.2 Soundness of the trace semantics of **ImpMod** . . . 250

8 Note on non-commutative linking **250**

9 Example output of the source-to-source transformation **251**

List of Definitions and Lemmas

1 Definition (Unforged code/data capability) . . . 10
2 Definition (Valid code/data capability) . . . 10
3 Definition (Subset relation and disjoint capabilities) . . . 10
1 Lemma (The subset and disjointness relations are offset oblivious) . . . 10
4 Definition (Comparing a capability to a set of addresses) . . . 10
5 Definition (Membership of a capability's address in a set of addresses) . . . 10
6 Definition (Equal-bounds capabilities) . . . 10
2 Lemma (Reduction does not change call frame sizes, imports map or code memory) 13
3 Lemma (A reduction is enabled only on a valid program counter) . . . 14
7 Definition (Code region of an imports map) . . . 14
4 Lemma (Expression evaluation cannot forge code capabilities) . . . 14
8 Definition (Disjoint object capabilities) . . . 16
9 Definition (Valid Linking) . . . 16
10 Definition (Initial state) . . . 16
11 Definition (Initial state function) . . . 18
12 Definition (Main module) . . . 18
1 Claim (The function $\mathsf{initial\_state}$ and the judgment $\vdash_i$ are compatible) . . . 18
13 Definition (Terminal state) . . . 18
14 Definition (Addition of an offset ω to the data memory) . . . 18
15 Definition (Addition of an offset ω to the imp map) . . . 18
16 Definition (Addition of an offset ω to a program t) . . . 18
17 Definition (Linkability, loadability, and convergence of execution in the target language) 19
18 Definition (Target contextual equivalence) . . . 19
19 Definition (Valid execution state) . . . 19
5 Lemma (Initial states are valid execution states) . . . 19
20 Definition (Accessible addresses) . . . 19
21 Definition (k-accessible addresses) . . . 19
22 Definition (Reachable addresses) . . . 19
6 Lemma (Reachability is not affected by offsets, only bounds) . . . 19
7 Lemma ($\mathsf{access}_{\mathcal{M}_d}$ is expansive) . . . 19
8 Lemma ($\mathsf{access}_{n,\mathcal{M}_d}$ is expansive) . . . 20
9 Lemma (Fixed points lead to convergence of $\mathsf{access}_{k,\mathcal{M}_d}$) . . . 20
10 Lemma (In an empty memory, only the starting addresses are reachable) . . . 20
11 Lemma (k-accessibility either adds a new memory address or a fixed point has been reached) . . . 21
12 Lemma (k-accessibility set contains at least k mapped addresses) . . . 21
13 Lemma ($|\mathcal{M}_d|$-accessibility suffices) . . . 22
14 Lemma (Invariance to non-δ-capability values) . . . 23
15 Lemma (Overwriting a non-δ-capability value does not shrink the accessibility set) . 23
16 Lemma (Additivity of $\mathsf{access}_{\mathcal{M}_d}$) . . . 24
17 Lemma (Additivity of $\mathsf{access}_{k,\mathcal{M}_d}$) . . . 25
18 Lemma (Additivity of $\mathsf{reachable\_addresses}$ in the first argument) . . . 25

19 Lemma (Additivity of reachable_addresses in the first argument using addr) 26
20 Lemma (Invariance to capability's location so long as it is reachable) 26
21 Lemma (Invariance to unreachable memory updates) 30
22 Lemma (Updating k-inaccessible locations does not affect the k-accessibility set) . . 32
23 Lemma (Updating a location does not affect its own k-accessibility) 33
24 Lemma (Updating a location does not affect its own reachability) 33
25 Lemma (Completeness of reachable_addresses) 33
26 Lemma (Expression evaluation cannot forge data capabilities) 35
23 Definition (Derivable capability) 36
27 Lemma (Upward closure of derivability) 36
28 Lemma (Reachability traverses all derivable capabilities) 36
29 Lemma (Preservation of reachability equivalence under safe memory updates) 37
24 Definition (Shrunk access: Access set without using the capability at location a) . . 39
25 Definition (Shrunk k-th access: K-th access set without using the capability at location a) 40
30 Lemma (Additivity of χ_k) 40
31 Lemma (χ_k is upper-bounded by k-accessibility) 40
32 Lemma (One capability is potentially lost from accessible addresses as a result of a non-capability update) 40
33 Lemma (χ_k captures k-accessibility after potential deletion of a capability) 40
34 Lemma (Reachability is captured by union over χ_k after potential deletion of a capability) 40
35 Lemma (Accessible addresses shrink by non-δ-capability updates) 40
36 Lemma (k-accessible addresses shrink by non-δ-capability updates) 41
37 Lemma (Reachability shrinks by non-δ-capability updates) 41
38 Lemma (Safe memory updates only shrink reachability) 42
39 Lemma (Safe allocation adds only allocated addresses to k-accessibility) 43
40 Lemma (Safe allocation adds only allocated addresses to reachability) 44
41 Lemma (Safe allocation causes reduction of k-accessibility to χ_k and addition of exactly the allocated addresses) 44
42 Lemma (Effect of assigning a derivable capability) 45
43 Lemma (Assigning a derivable capability does not enlarge reachability) 45
26 Definition (Sub-capability-closed predicate) 45
27 Definition ($\mathbb{Z}$-trivial predicate) 45
28 Definition (Offset-oblivious predicate) 45
29 Definition (Allocation-compatible predicate) 45
30 Definition (State-universal predicate) 45
44 Lemma (Predicates that are guaranteed to hold on the result of expression evaluation) 46
45 Lemma (Preservation of state universality of predicates) 46
31 Definition (Code capabilities have an imports origin) 50
46 Lemma (κ_has_origin$_{imp}$ is sub-capability closed) 50
47 Lemma (κ_has_origin$_{imp}$ is $\mathbb{Z}$-trivial) 50
48 Lemma (κ_has_origin$_{imp}$ is offset oblivious) 50
49 Lemma (κ_has_origin$_{imp}$ is allocation compatible) 50
50 Lemma (κ_has_origin$_{imp}$ is initial-state-universal) 51
51 Lemma (κ_has_origin$_{imp}$ is universal for subsequent states) 51
1 Corollary (There is at least one module that is executing at any time) 52
52 Lemma (Preservation of $\vdash_{exec}$ by reduction) 52
2 Corollary (Preservation of $\vdash_{exec}$ by $\rightarrow^*$) 68
3 Corollary (Data and stack capabilities always hold a data-capability value) 68
53 Lemma (Preservation of $\vdash_{exec}$ by $\succ_{\approx}$) 69

54 Lemma (At the initial state, the program counter capability pcc and the data capability ddc are prescribed by some capability object) . . . 69
2 Claim (At the initial state, the data and stack capabilities are disjoint) . . . 69
3 Claim (Uniqueness of the initial state (Existence of at most one initial state for a given *TargetSetup*)) . . . 69
55 Lemma (Preservation of the bounds of stack capabilities) . . . 69
32 Definition (Valid linking) . . . 71
33 Definition (Set of function definitions of a list of modules) . . . 73
34 Definition (Function ID to function definition map) . . . 73
35 Definition (Module variables map) . . . 73
36 Definition (Valid execution state of a program) . . . 76
37 Definition (Initial state) . . . 76
38 Definition (Initial state function) . . . 76
39 Definition (Main module) . . . 76
4 Claim (The function initial_state and the judgment $\vdash_i$ are compatible) . . . 76
40 Definition (Terminal state) . . . 76
41 Definition (Layout places $\overline{m_1}$ before $\mathbb{C}$) . . . 76
42 Definition (Layout-ordered linking) . . . 78
43 Definition (Linkability, loadability, and convergence of execution in the source language) 78
44 Definition (Addition of an offset ω to the data segment's bounds) . . . 78
45 Definition (Source contextual equivalence) . . . 78
56 Lemma (Preservation of $\vdash_{exec}$) . . . 78
4 Corollary (Preservation of $\vdash_{exec}$ by the reflexive transitive closure) . . . 87
46 Definition (Static Addresses) . . . 87
47 Definition (Memory accessibility) . . . 88
48 Definition (Memory k-accessibility) . . . 88
49 Definition (Reachable Addresses) . . . 88
57 Lemma (Reachable addresses are static addresses or are memory-stored) . . . 88
58 Lemma (access is expansive) . . . 88
59 Lemma (access_n is expansive) . . . 88
60 Lemma (Fixed points lead to convergence of access_k) . . . 88
61 Lemma (In an empty memory, only the starting addresses are reachable) . . . 88
62 Lemma (k-accessibility either adds a new memory address or a fixed point has been reached) . . . 88
63 Lemma (k-accessibility set contains at least k mapped addresses) . . . 89
64 Lemma ($|Mem|$-accessibility suffices) . . . 89
65 Lemma (Safe allocation adds only allocated addresses to k-accessibility) . . . 89
66 Lemma (Safe allocation adds only allocated addresses to reachability) . . . 89
67 Lemma (Safe allocation causes reduction of k-accessibility to χ_k and addition of exactly the allocated addresses) . . . 89
68 Lemma (Invariance to unreachable memory updates) . . . 89
69 Lemma (Updating k-inaccessible locations does not affect the k-accessibility set) . . . 90
70 Lemma (Updating a location does not affect its own k-accessibility) . . . 90
71 Lemma (Updating a location does not affect its own reachability) . . . 90
72 Lemma (χ_k is upper-bounded by k-accessibility) . . . 90
73 Lemma (One capability is potentially lost from accessible addresses as a result of a non-capability update) . . . 90
74 Lemma (χ_k captures k-accessibility after potential deletion of a capability) . . . 90
75 Lemma (Reachability is captured by union over χ_k after potential deletion of a capability) . . . 90
50 Definition (Derivable capability) . . . 90
76 Lemma (Reachability traverses all derivable capabilities) . . . 91

77 Lemma (Additivity of access) . . . 91
78 Lemma (Additivity of access_k) . . . 91
79 Lemma (Effect of assigning a derivable capability) . . . 91
80 Lemma (Assigning a derivable capability does not enlarge reachability) . . . 91
81 Lemma (Completeness of reachable_addresses) . . . 91
51 Definition (Data segment capability of a module) . . . 93
52 Definition (Stack capability of a module) . . . 93
53 Definition (Capabilities of a module) . . . 94
54 Definition (Static capabilities) . . . 94
82 Lemma (Static addresses are precisely those of static capabilities) . . . 94
55 Definition (Access to capabilities) . . . 94
83 Lemma (Accessed addresses are precisely the addresses of accessed capabilities) . . . 94
56 Definition (k-access to capabilities) . . . 94
84 Lemma (k-accessed addresses are precisely the addresses of k-accessed capabilities) . 94
57 Definition (Reachable capabilities) . . . 94
85 Lemma (Reachable addresses are precisely the addresses of the reachable capabilities) 94
58 Definition (Expression Translation) . . . 95
59 Definition (Command Translation) . . . 95
86 Lemma (Code and data segment capabilities are precise with respect to the code and data memory initializations) . . . 95
60 Definition (Source-target value relatedness) . . . 96
87 Lemma (Expression translation forward simulation - case $\mathsf{addr}(vid)$) . . . 96
88 Lemma (Expression translation forward simulation) . . . 99
89 Lemma (Expression translation backward simulation - case $\mathsf{addr}(vid)$) . . . 102
90 Lemma (Expression translation backward simulation) . . . 103
91 Lemma (Memory bounds are preserved by compilation) . . . 108
92 Lemma (No additional code/data is added by the compiler) . . . 109
93 Lemma (Code memory is the translation of the commands arranged according to K_{mod} and K_{fun}) . . . 109
61 Definition (Related program counters) . . . 109
62 Definition (Related stacks) . . . 109
63 Definition (Related local stack usage) . . . 109
64 Definition (Cross-language compiled-program state similarity) . . . 110
94 Lemma (Cross-language equi-k-accessibility and memory equality is preserved by deleting assignments and safe allocation) . . . 110
95 Lemma (Cross-language equi-reachability and memory equality is preserved by deleting assignments, safe allocation, and assigning derivable capabilities) . . . 111
96 Lemma (Compiled-program state similarity implies equi-reachability) . . . 112
97 Lemma (Compiler forward simulation) . . . 112
98 Lemma (Compiler backward simulation) . . . 127
99 Lemma (Compiler forward simulation, multiple steps) . . . 134
1 Theorem (Compiler backward simulation, multiple steps (Compiler correctness)) . . 135
100 Lemma (Source and compiled initial states are cross-language related) . . . 135
65 Definition (Target empty context) . . . 138
101 Lemma (Target empty context is universally linkable) . . . 138
66 Definition (Target whole-program convergence compatible with partial convergence) 138
67 Definition (Source empty context) . . . 138
102 Lemma (Source empty context is universally linkable and universally order-preserving) 138
68 Definition (Source whole-program convergence compatible with partial convergence) 138
103 Lemma (Cross-language relatedness implies equi-terminality) . . . 138
104 Lemma (Existence of an initial state is preserved and reflected by $[\![\cdot]\!]$) . . . 140
105 Lemma (Convergence is preserved and reflected by $[\![\cdot]\!]$) . . . 144

106 Lemma (Compilation preserves linkability and convergence, i.e., $[\![\cdot]\!]$ is a linking-preserving homomorphism and more) . . . 146
107 Lemma (Compiler is a linking-preserving homomorphism) . . . 149
69 Definition (Alternatingly-communicating finite traces) . . . 150
5 Claim (Extending an alternating prefix to keep it alternating) . . . 150
70 Definition (Reflexive transitive closure of trace actions) . . . 152
71 Definition (Non-silent trace steps) . . . 152
6 Claim (A non-silent trace is not the empty string) . . . 153
7 Claim ($\rightharpoonup\!\!\!\!\rightharpoonup$ eliminates τ actions) . . . 153
8 Claim ($\rightharpoonup\!\!\!\!\rightharpoonup$ is supported by $\rightharpoonup$) . . . 153
9 Claim ($\rightharpoonup\!\!\!\!\rightharpoonup$ decomposes) . . . 153
10 Claim (Non-silent part of $\rightharpoonup^*$ is supported by $\rightharpoonup\!\!\!\!\rightharpoonup$) . . . 153
72 Definition (A prefix of an execution trace is possible for a component) . . . 154
73 Definition (Trace equivalence) . . . 154
11 Claim (Termination markers appear only at the end of an execution trace) . . . 154
12 Claim (Prefix-closure of trace set membership) . . . 154
13 Claim (A state that is reachable by $\rightarrow$ reduction or by $\succ_{\approx}$ is also reachable by $\rightharpoonup$) . 154
14 Claim (A non-$\bot$ state that is reachable by $\rightharpoonup$ is also reachable by $\rightarrow$ reduction) . . . 154
15 Claim (Silent trace steps correspond to $\rightarrow$ steps) . . . 154
16 Claim (Non-stuck trace steps correspond to $\rightarrow$ execution steps) . . . 154
17 Claim (The set of shared addresses ς does not change by silent trace steps) . . . 155
5 Corollary (Reachability by $\rightarrow^*$ implies reachability by $\rightharpoonup^*$) . . . 155
6 Corollary (Reachability by $\rightharpoonup^*$ implies reachability by $\rightarrow^*$ when the state is non-$\bot$) 155
108 Lemma (Non-communication actions do not change context/compiled component's ownership of pcc) . . . 155
7 Corollary (Non-communication actions do not change ownership of pcc (star-closure)) 157
109 Lemma (Traces consist of alternating input/output actions) . . . 157
74 Definition (Alternating Strong-Weak Similarity (ASWS)) . . . 159
110 Lemma (Initial states are ASWS-related) . . . 159
111 Lemma (Two peripheral terminal states are ASWS-related to only a mixed state that is also terminal) . . . 160
75 Definition (View change of a trace step) . . . 160
1 Fact (View change is an involution) . . . 160
18 Claim (Existence of a view change of a trace step) . . . 160
112 Lemma (ASWS satisfies the alternating simulation condition) . . . 161
113 Lemma (ASWS satisfies the alternating simulation condition – whole trace) . . . 162
114 Lemma (Soundness of trace equivalence with respect to contextual equivalence) . . 163
76 Definition (Reflexive transitive closure of trace actions) . . . 166
77 Definition (Non-silent trace steps) . . . 166
19 Claim (A non-silent trace is not the empty string) . . . 166
20 Claim ($\rightharpoonup\!\!\!\!\rightharpoonup$ eliminates τ actions) . . . 166
21 Claim ($\rightharpoonup\!\!\!\!\rightharpoonup$ is supported by $\rightharpoonup$) . . . 168
22 Claim ($\rightharpoonup\!\!\!\!\rightharpoonup$ decomposes) . . . 168
23 Claim (Non-silent part of $\rightharpoonup^*$ is supported by $\rightharpoonup\!\!\!\!\rightharpoonup$) . . . 168
78 Definition (A prefix of an execution trace is possible for a component) . . . 169
79 Definition (Trace equivalence) . . . 169
24 Claim (Termination markers appear only at the end of an execution trace) . . . 169
25 Claim (Prefix-closure of trace set membership) . . . 169
26 Claim (A state that is reachable by $\rightarrow$ reduction or by $\succ_{\approx}$ is also reachable by $\rightharpoonup$) . 169
27 Claim (A non-$\bot$ state that is reachable by $\rightharpoonup$ is also reachable by $\rightarrow$ reduction) . . . 169
28 Claim (Silent trace steps correspond to $\rightarrow$ steps) . . . 170
29 Claim (Non-stuck trace steps correspond to $\rightarrow$ execution steps) . . . 170

30 Claim (The set of shared addresses ς does not change by silent trace steps) 170
8 Corollary (Reachability by $\rightarrow^*$ implies reachability by $\rightharpoonup^*$) 170
9 Corollary (Reachability by $\rightharpoonup^*$ implies reachability by $\rightarrow^*$ when the state is non-$\perp$) 170
115 Lemma (Non-communication actions do not change context/compiled component's ownership of pc) 170
10 Corollary (Non-communication actions do not change ownership of pc (star-closure)) 171
116 Lemma (Traces consist of alternating input/output actions) 171
117 Lemma (Completeness of trace equivalence with respect to contextual equivalence) . 171
80 Definition (Distinguishing snippet for equi-flow trace actions) 171
118 Lemma (Value cross-relatedness on integers is compatible with **ImpMod** subtraction) 172
119 Lemma (If two target values are unequal, then distinguishArgs produces code that terminates on exactly one of them) 172
81 Definition (Compiler full abstraction) 173
2 Theorem ($[\![\cdot]\!]$ is fully abstract) 173
120 Lemma ($[\![\cdot]\!]$ reflects contextual equivalence) 174
121 Lemma ($[\![\cdot]\!]$ preserves contextual equivalence) 175
122 Lemma (Compilation preserves trace equivalence) 175
123 Lemma (Forward simulation of call attempt) 176
124 Lemma (Forward simulation of call attempt) 176
125 Lemma (Compiler forward simulation lifted to a trace step) 176
126 Lemma (Compiler backward simulation lifted to a trace step) 177
127 Lemma (Compiler forward simulation lifted to many trace steps) 178
128 Lemma (Compiler backward simulation lifted to many trace steps) 179
129 Lemma (Compiler forward simulation lifted to compressed trace steps) 179
130 Lemma (Compiler backward simulation lifted to compressed trace steps) 180
131 Lemma (No trace is removed by compilation) 180
82 Definition (Component-controlled memory region) 180
31 Claim (Controlled-region equality implies reachability equality) 181
83 Definition (Similarity of stack capabilities) 181
32 Claim (Similarity of mstc is an equivalence relation) 181
84 Definition (Strong stack-similarity) 182
85 Definition (Weak stack-similarity) 182
132 Lemma (A strictly-monotone function is injective) 182
86 Definition (Trace-state similarity) 182
133 Lemma (Strong stack-similarity is an equivalence relation) 183
33 Claim (Weak stack-similarity is an equivalence relation) 184
34 Claim (State similarity is an equivalence relation) 184
134 Lemma (Similarity of stack capabilities compatible with uniform substitution) 184
135 Lemma (Initial states of the program of interest are strongly related) 184
136 Lemma (Initial states of the context are weakly related) 184
137 Lemma (Terminal states are strongly-related to only terminal states) 185
138 Lemma (Equality of expression evaluation between strongly-similar states) 185
139 Lemma (The empty stack is in a singleton equivalence class of strong stack-similarity) 186
140 Lemma (Adequacy of strong stack-similarity (syncing border-crossing return to non-$\overline{c}$ call-site)) 186
141 Lemma (Weak stack-similarity is preserved by a unilateral silent return) 187
142 Lemma (Weak stack-similarity is preserved by a unilateral silent call) 187
143 Lemma (Weakening of strong stack-similarity) 188
144 Lemma (Strong stack-similarity is preserved by a bilateral call (from same $\overline{c}$-call-site)) 188
145 Lemma (Strong stack-similarity is weakened by a bilateral return to a non-$\overline{c}$-call-site) 190
146 Lemma (Strong stack-similarity is preserved by a bilateral return to a $\overline{c}$-call-site) . . 190
147 Lemma (Strengthening of weak stack-similarity by a bilateral call from non-$\overline{c}$ call-sites) 191

148 Lemma (A silent action on strongly-similar states satisfies lock-step simulation) . . . 193
11 Corollary (Star silent actions on strongly-similar states satisfy simulation) 200
149 Lemma (Strong state-similarity determines non-silent output actions and is weakened by them) 200
150 Lemma (Option simulation: preservation of stack similarity by a silent action) . . . 203
151 Lemma (Option simulation: preservation of mstc similarity by a silent action) 204
152 Lemma (Option simulation: preservation of weak similarity by a silent action) 206
153 Lemma (Matching input actions retrieve back strong state-similarity) 207
87 Definition (Per-subject state-universal predicate) 208
154 Lemma (Predicates that are guaranteed to hold on the result of expression evaluation under the execution of a specific subject) 209
155 Lemma (Preservation of per-subject state universality of predicates) 209
88 Definition (Four-origin policy) 209
35 Claim (Border state invariant to silent state invariant - $\bar{c}$ executing) 210
36 Claim (Border state invariant to silent state invariant - t_{ctx} executing) 210
156 Lemma (Possible origins of capability values at border states) 210
157 Lemma (Preservation of the silent-state invariant) 214
158 Lemma (Preservation of the border-state invariant $\vdash_{border}$) 225
89 Definition (Main module of the emulating context) 228
90 Definition (Context module IDs of a trace) 228
91 Definition (Context function IDs of a trace) 228
92 Definition (Number of arguments of a function inferred from either the trace α_1 or the trace α_2) 228
93 Definition (Memory of a trace label) 228
94 Definition (Allocation status of a trace label) 229
95 Definition (Shared addresses throughout a trace prefix α) 229
96 Definition (Context addresses collected from a trace) 229
97 Definition (Data segment that the context shares (collected from a trace)) 229
98 Definition (A trace compatible with a program's data segment) 229
99 Definition (A trace satisfies monotonic sharing) 229
100 Definition (A trace satisfies no-deallocation) 229
101 Definition (Syntactically-sane trace) 229
102 Definition (Global variables of the module `mainModule`) 229
103 Definition (The function `readAndIncrementTraceIdx`) 230
104 Definition (The functions `saveArgs`) 230
105 Definition (Functions of the module `mainModule`) 230
106 Definition (Constructing dereferences from path) 231
107 Definition (Constructing path to target address) 231
108 Definition (Construct address back-translation for addresses reachable from a capability argument) 231
109 Definition (Construct address back-translation map from a call-/return to- context label) 231
110 Definition (Diverging block of code) 232
111 Definition (Converging block of code) 232
112 Definition (If-then-else in **ImpMod**) 232
113 Definition (Switch-block for integers in **ImpMod**) 232
114 Definition (Upcoming commands at an execution state) 232
159 Lemma (If-then-else construction is correct) 232
160 Lemma (Switch construction is correct) 233
161 Lemma (A converge block leads to a terminal state) 233
162 Lemma (A diverge block does not lead to a terminal state) 233
163 Lemma (Effect of calling `readAndIncrementTraceIdx`) 233

115 Definition (Independent set of assignments) 237
164 Lemma (Effect of calling `saveArgs`) 237
116 Definition (Logged memory correct) 240
117 Definition (Arguments saved correctly) 241
118 Definition (Allocation pointers saved) 241
37 Claim (There is a source function that does allocations according to allocation_pointers_saved) 241
119 Definition (Emulate call or return or exit command of i-th output action) 241
120 Definition (Emulate i-th output action) 242
121 Definition (Responses for suffix) 242
165 Lemma (Adequacy of emulate_responses_for_suffix) 242
122 Definition (Emulating function) 242
123 Definition (Emulating module) 243
124 Definition (Emulating modules) 243
125 Definition (The emulating context) 243
166 Lemma (The emulating context is linkable and loadable) 243
126 Definition (Emulate invariants) 243
167 Lemma (Initial state of emulate satisfies emulate_invariants) 244
168 Lemma (Adequacy of emulate_invariants) 244
169 Lemma (Preservation of emulate_invariants) 245
127 Definition (Trace-Indexed Cross-Language (TrICL) simulation relation) 245
170 Lemma (TrICL satisfies the alternating simulation condition) 245
171 Lemma (Initial states are TrICL-related) 247
172 Lemma (TrICL-related states are co-terminal) 247
173 Lemma (No trace is added by compilation) 248
12 Corollary (Completeness of target trace equivalence for contextual equivalence of compiled components) 250
13 Corollary (Soundness of source traces) 250

1 The target language (CHERIExp)

Our target language models a platform that supports memory and object capabilities, and is strongly inspired by the CHERI system [1, 2], a MIPS-based capability-machine architecture. CHERI offers fine-grained memory capabilities through hardware support, and it offers object capabilities through a combination of hardware support, kernel support and a user-space library (libcheri).

Accordingly, we model in this section a low-level target language, which we call **CHERIExp**. This language includes abstractions that mimic the interfaces offered by libcheri as well as CHERI's capabilities. Our model of capabilities draws heavily from a prior model of a capability machine [3].

1.1 Values, expressions, and commands

Values in **CHERIExp** are denoted by $\mathcal{V} = \mathbb{Z} \cup \mathit{Cap}$ and range over integers $\mathbb{Z}$ and memory capabilities $\mathit{Cap} = \{\kappa, \delta\} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$. Memory capabilities are code or data capabilities, denoted by κ and δ respectively, where the κ-labeled elements describe a range of the code memory $\mathcal{M}_c$ and an offset within this range, and the δ-labeled elements describe the same for the data memory $\mathcal{M}_d$. We separate capabilities from integers to model unforgeability of capabilities, which is a key design feature in CHERI [1, 2]. Formal arguments of how this unforgeability is guaranteed by the CHERI architecture are beyond the scope of this paper, but can be found in [3].

Definition 1 (Unforged code/data capability)**.**
We use the judgment $\vDash_x (y, s, e, \mathit{off})$ to mean that $y = x$ and that $(y, s, e, \mathit{off}) \in \{y\} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$ which means that (y, s, e, off) is an unforged capability value of type x.

Definition 2 (Valid code/data capability)**.**
We use the judgment $\vdash_x (y, s, e, \mathit{off})$ to mean that $\vDash_x (y, s, e, \mathit{off})$ and that $s + \mathit{off} \in [s, e)$ which is the condition necessary for valid access using this capability.

Validity of a code/data capability $(\sigma, s, e, \mathit{off})$ ensures that it is of the intended capability type x, and that its offset lies within the legal range that it prescribes.

Definition 3 (Subset relation and disjoint capabilities)**.**
We use the judgment $(x, s_1, e_1, \_) \subseteq (x, s_2, e_2, \_)$ to mean $[s_1, e_1) \subseteq [s_2, e_2)$ and similarly $(x, s_1, e_1, \_) \cap (x, s_2, e_2, \_) = \emptyset$ to mean that $[s_1, e_1) \cap [s_2, e_2) = \emptyset$.

Lemma 1 (The subset and disjointness relations are offset oblivious)**.**

$$\begin{aligned}
&\forall x, \sigma_1, e_1, \sigma_2, e_2, \mathit{off}_1, \mathit{off}_2\, \mathit{off}'_1, \mathit{off}'_2.\\
&((x, \sigma_1, e_1, \mathit{off}_1) \subseteq (x, \sigma_2, e_2, \mathit{off}_2) \implies (x, \sigma_1, e_1, \mathit{off}'_1) \subseteq (x, \sigma_2, e_2, \mathit{off}'_2)) \wedge\\
&((x, \sigma_1, e_1, \mathit{off}_1) \cap (x, \sigma_2, e_2, \mathit{off}_2) = \emptyset \implies (x, \sigma_1, e_1, \mathit{off}'_1) \cap (x, \sigma_2, e_2, \mathit{off}'_2) = \emptyset)
\end{aligned}$$

Proof.
Immediate by Definition 3. □

Definition 4 (Comparing a capability to a set of addresses)**.**
We overload the notation $\subseteq$ to represent a relation over $\mathit{Cap} \times 2^{\mathbb{Z}}$ between a capability and a set of integers where $(\_, s, e, \_) \subseteq X$ means that the interval $[s, e)$ of integers is a subset of X ($[s, e) \subseteq X$).

Definition 5 (Membership of a capability's address in a set of addresses)**.**
We similarly use the set membership notation $\in$ to mean with $(\_, s, e, \mathit{off}) \in X$ that the address $s + \mathit{off}$ is a member in the set X of natural numbers (i.e., $s + \mathit{off} \in X$).

Definition 6 (Equal-bounds capabilities)**.**
We use the judgment $(x, \sigma_1, e_1, \_) \doteq (x, \sigma_2, e_2, \_) \stackrel{\mathsf{def}}{=} \sigma_1 = \sigma_2 \wedge e_1 = e_2$ to mean that the bounds of two capabilities are the same (i.e., the two capabilities give authority over the same range of memory addresses). Notice that $a \doteq b$ is equivalent to $a \subseteq b \wedge b \subseteq a$ for any two capabilities a and b.

And we define the function `inc`: $\mathit{Cap} \times \mathbb{Z} \to \mathit{Cap}$ as $\mathtt{inc}((x, s, e, \mathit{off}), z) \stackrel{def}{=} (x, s, e, \mathit{off} + z)$ which increments the offset of a capability by z.

Memory notation

Code and data memories ($\mathcal{M}_c : \mathbb{N} \xrightarrow{\text{fin}} \mathit{Cmd}$ and $\mathcal{M}_d : \mathbb{Z} \xrightarrow{\text{fin}} \mathcal{V}$) are finite maps from addresses –that are natural numbers– to commands and values respectively. Memory values have been described above. Below we describe expressions and commands. But we first fix some notation regarding code and data memories:

- We refer to the type $\mathbb{N} \xrightarrow{\text{fin}} \mathit{Cmd}$ as $\mathit{CodeMemory}$ and to the type $\mathbb{Z} \xrightarrow{\text{fin}} \mathcal{V}$ as $\mathit{DataMemory}$.

- The operator $\uplus$ is used to refer to the disjoint union of sets or functions. For functions f and g with $\mathtt{dom}(f) \cap \mathtt{dom}(g) = \emptyset$, the function $(f \uplus g)$ has domain $\mathtt{dom}(f) \cup \mathtt{dom}(g)$ and is defined as $(f \uplus g)(x) \stackrel{\text{def}}{=} f(x)$ if $x \in \mathtt{dom}(f)$, and $g(x)$ otherwise. We use the notation $\mathcal{M}_c = \biguplus_i \mathcal{M}_{c_i}$ to mean the linking of several code memories $\mathcal{M}_{c_i}$ with disjoint mapped addresses into one code memory $\mathcal{M}_c$, and similarly for other constructs that are maps or functions.

Commands in CHERIExp

Figure 1 shows the semantics of **CHERIExp** commands. The semantics is given by the reduction relation $\rightarrow \;\subseteq \mathit{TargetState} \times \mathit{TargetState}$. The reduction relation is additionally parameterized by $\nabla \in \mathbb{Z}$ which prescribes the total amount of memory available for dynamic allocation. We omit it from the symbol $\rightarrow_\nabla$, and always write just $\rightarrow$ for convenience. Every statement that mentions the reduction relation $\rightarrow$ should be understood to be in the scope of one outermost universal quantification over ∇ unless otherwise is explicitly mentioned. The type $\mathit{TargetState}$ is defined in the section below. An auxiliary relation $\succ_\approx$ is used to describe the behavior of the Cinvoke command in the case when there is enough stack space. This is useful for re-factoring and proof purposes. Commands Cmd in **CHERIExp** are the following:

- Assign $\mathcal{E}_L$ $\mathcal{E}_R$ which evaluates the expression $\mathcal{E}_R$ to a value $v \in \mathcal{V}$, evaluates the expression $\mathcal{E}_L$ to a data capability value $c \in \{\delta\} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$, and stores in the data memory $\mathcal{M}_d$ the value v at the address indicated by c (the address $(s + o)$ for $c = (\delta, s, e, o)$).

- Alloc $\mathcal{E}_L$ $\mathcal{E}_{size}$ which allocates new memory and stores a data capability giving authority over the newly-allocated memory. The parameter ∇ is the first unavailable address indicating the limit of memory usage. Alloc fails (i.e., execution gets stuck) if this limit is reached.

- JumpIfZero $\mathcal{E}_{cond}$ $\mathcal{E}_{off}$ is a conditional jump which evaluates the expression $\mathcal{E}_{cond}$ to a value $v \in \mathbb{N}$, and if $v \neq 0$, then it evaluates the expression $\mathcal{E}_{off}$ to an offset that is added to pcc. Otherwise ($v = 0$), nothing is done.

- Cinvoke mid fid $\overline{e}$ [1] , which is used to invoke an object capability. Our target platform is configured (in the imp component of the initial machine state, see below) with a fixed number of object capabilities identified by module identifiers $\mathit{mid} \in \mathit{ModID}$, and each object capability supports invocation of a fixed number of functions specified by function identifiers $\mathit{fid} \in \mathit{FunID}$. Each secure call to a function fid gets access via stc to a new data stack frame of size $\phi(\mathit{fid})$ for local use. Argument values are also written by the Cinvoke command in this region. This latter design choice is a simpler alternative to modeling a register file.

- CReturn, which is used to return from a call that has been performed using Cinvoke. The rules cinvoke and creturn in fig. 1 specify the exact operations performed to push and pop the necessary capabilities to/from the trusted stack.

[1] We use the notation $\overline{x}$ to denote that x has a list type. And we also use the same notation for types (i.e., as a type constructor). For instance, we write $\overline{\mathbb{N}}$ to denote the type of lists of natural numbers.

Figure 1: Evaluation of commands *Cmd* in **CHERIExp**. The reduction relation is parameterized by ∇. We omit it from the symbol $\rightarrow$ for convenience.

(assign)

$$\frac{\begin{array}{c}\vdash_\kappa \mathsf{pcc} \qquad \mathsf{pcc}' = \mathtt{inc}(\mathsf{pcc}, 1) \\ \mathcal{M}_c(\mathsf{pcc}) = \mathsf{Assign}\ \mathcal{E}_L\ \mathcal{E}_R \qquad \mathcal{E}_R, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow v \\ \mathcal{E}_L, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow c \qquad \vdash_\delta c \qquad \vDash_\delta v \implies (v \cap \mathsf{stc} = \emptyset \vee c \subseteq \mathsf{stc}) \qquad \mathcal{M}_d' = \mathcal{M}_d[c \mapsto v]\end{array}}{\langle \mathcal{M}_c, \mathcal{M}_d, \mathit{stk}, \mathit{imp}, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle \rightarrow \langle \mathcal{M}_c, \mathcal{M}_d', \mathit{stk}, \mathit{imp}, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}', \mathsf{mstc}, \mathsf{nalloc}\rangle}$$

(allocate)

$$\frac{\begin{array}{c}\vdash_\kappa \mathsf{pcc} \qquad \mathsf{pcc}' = \mathtt{inc}(\mathsf{pcc}, 1) \\ \mathcal{M}_c(\mathsf{pcc}) = \mathsf{Alloc}\ \mathcal{E}_L\ \mathcal{E}_{size} \qquad \mathcal{E}_{size}, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow v \qquad \mathcal{E}_L, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow c \\ v \in \mathbb{Z}^+ \qquad \vdash_\delta c \qquad \mathcal{M}_d' = \mathcal{M}_d[c \mapsto (\delta, \mathsf{nalloc} - v, \mathsf{nalloc}, 0), i \mapsto 0\ \forall i \in [\mathsf{nalloc} - v, \mathsf{nalloc})] \\ \mathsf{nalloc}' = \mathsf{nalloc} - v \qquad \mathsf{nalloc}' > \nabla\end{array}}{\langle \mathcal{M}_c, \mathcal{M}_d, \mathit{stk}, \mathit{imp}, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle \rightarrow \langle \mathcal{M}_c, \mathcal{M}_d', \mathit{stk}, \mathit{imp}, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}', \mathsf{mstc}, \mathsf{nalloc}'\rangle}$$

(jump0)

$$\frac{\begin{array}{c}\vdash_\kappa \mathsf{pcc} \qquad \mathcal{M}_c(\mathsf{pcc}) = \mathsf{JumpIfZero}\ \mathcal{E}_{cond}\ \mathcal{E}_{\mathit{off}} \qquad \mathcal{E}_{cond}, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow v \qquad v = 0 \\ \mathcal{E}_{\mathit{off}}, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow \mathit{off} \qquad \mathit{off} \in \mathbb{Z} \qquad \mathsf{pcc}' = \mathtt{inc}(\mathsf{pcc}, \mathit{off})\end{array}}{\langle \mathcal{M}_c, \mathcal{M}_d, \mathit{stk}, \mathit{imp}, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle \rightarrow \langle \mathcal{M}_c, \mathcal{M}_d, \mathit{stk}, \mathit{imp}, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}', \mathsf{mstc}, \mathsf{nalloc}\rangle}$$

(jump1)

$$\frac{\begin{array}{c}\vdash_\kappa \mathsf{pcc} \qquad \mathcal{M}_c(\mathsf{pcc}) = \mathsf{JumpIfZero}\ \mathcal{E}_{cond}\ \mathcal{E}_{\mathit{off}} \\ \mathcal{E}_{cond}, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow v \qquad v \neq 0 \qquad \mathsf{pcc}' = \mathtt{inc}(\mathsf{pcc}, 1)\end{array}}{\langle \mathcal{M}_c, \mathcal{M}_d, \mathit{stk}, \mathit{imp}, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle \rightarrow \langle \mathcal{M}_c, \mathcal{M}_d, \mathit{stk}, \mathit{imp}, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}', \mathsf{mstc}, \mathsf{nalloc}\rangle}$$

(cinvoke-aux)

$$\frac{\begin{array}{c}\vdash_\kappa \mathsf{pcc} \qquad \mathcal{M}_c(\mathsf{pcc}) = \mathsf{Cinvoke}\ \mathit{mid}\ \mathit{fid}\ \overline{e} \qquad \mathit{stk}' = \mathtt{push}(\mathit{stk}, (\mathsf{ddc}, \mathsf{pcc}, \mathit{mid}, \mathit{fid})) \\ \phi(\mathit{mid}, \mathit{fid}) = (\mathit{nArgs}, \mathit{nLocal}) \qquad (\delta, s, e, \mathit{off}) = \mathsf{mstc}(\mathit{mid}) \qquad \mathit{off}' = \mathit{off} + \mathit{nArgs} + \mathit{nLocal} \\ \mathsf{stc}' = (\delta, s, e, \mathit{off}') \\ \overline{e}(i), \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow v_i\ \forall i \in [0, \mathit{nArgs}) \qquad \forall i \in [0, \mathit{nArgs}).\ \vDash_\delta v_i \implies v_i \cap \mathsf{stc} = \emptyset \\ \mathcal{M}_d' = \mathcal{M}_d[s + \mathit{off} + i \mapsto v_i\ \forall i \in [0, \mathit{nArgs})][s + \mathit{off} + \mathit{nArgs} + i \mapsto 0\ \forall i \in [0, \mathit{nLocal})] \\ \mathsf{mstc}' = \mathsf{mstc}[\mathit{mid} \mapsto \mathsf{stc}'] \qquad (c, d, \mathit{offs}) = \mathit{imp}(\mathit{mid}) \qquad \mathsf{ddc}' = d \qquad \mathsf{pcc}' = \mathtt{inc}(c, \mathit{offs}(\mathit{fid}))\end{array}}{\langle \mathcal{M}_c, \mathcal{M}_d, \mathit{stk}, \mathit{imp}, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle \succ_\approx \langle \mathcal{M}_c, \mathcal{M}_d', \mathit{stk}', \mathit{imp}, \phi, \mathsf{ddc}', \mathsf{stc}', \mathsf{pcc}', \mathsf{mstc}', \mathsf{nalloc}\rangle}$$

(cinvoke)

$$\frac{\begin{array}{c}\langle \mathcal{M}_c, \mathcal{M}_d, \mathit{stk}, \mathit{imp}, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle \succ_\approx \langle \mathcal{M}_c, \mathcal{M}_d', \mathit{stk}', \mathit{imp}, \phi, \mathsf{ddc}', \mathsf{stc}', \mathsf{pcc}', \mathsf{mstc}', \mathsf{nalloc}\rangle \\ \vdash_\delta \mathsf{stc}'\end{array}}{\langle \mathcal{M}_c, \mathcal{M}_d, \mathit{stk}, \mathit{imp}, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle \rightarrow \langle \mathcal{M}_c, \mathcal{M}_d', \mathit{stk}', \mathit{imp}, \phi, \mathsf{ddc}', \mathsf{stc}', \mathsf{pcc}', \mathsf{mstc}', \mathsf{nalloc}\rangle}$$

(creturn)

$$\frac{\begin{array}{c}\vdash_\kappa \mathsf{pcc} \qquad \mathcal{M}_c(\mathsf{pcc}) = \mathsf{Creturn} \qquad \mathit{stk}', (\mathsf{ddc}', \mathsf{pcc}', \mathit{mid}, \mathit{fid}) = \mathtt{pop}(\mathit{stk}) \\ \phi(\mathit{mid}, \mathit{fid}) = (\mathit{nArgs}, \mathit{nLocal}) \qquad (\delta, s, e, \mathit{off}) = \mathsf{mstc}(\mathit{mid}) \qquad \mathit{off}' = \mathit{off} - \mathit{nArgs} - \mathit{nLocal} \\ \mathsf{mstc}' = \mathsf{mstc}[\mathit{mid} \mapsto (\delta, s, e, \mathit{off}')] \qquad \exists \mathit{mid}'.\ \mathsf{pcc}' \doteq \mathit{imp}(\mathit{mid}').\mathsf{pcc}\ \wedge\ \mathsf{stc}' = \mathsf{mstc}(\mathit{mid}')\end{array}}{\langle \mathcal{M}_c, \mathcal{M}_d, \mathit{stk}, \mathit{imp}, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle \rightarrow \langle \mathcal{M}_c, \mathcal{M}_d, \mathit{stk}', \mathit{imp}, \phi, \mathsf{ddc}', \mathsf{stc}', \mathsf{pcc}', \mathsf{mstc}', \mathsf{nalloc}\rangle}$$

(cexit)

$$\frac{\vdash_\kappa \mathsf{pcc} \qquad \mathcal{M}_c(\mathsf{pcc}) = \mathsf{Exit}}{\langle \mathcal{M}_c, \mathcal{M}_d, \mathit{stk}, \mathit{imp}, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle \rightarrow \langle \mathcal{M}_c, \mathcal{M}_d, \mathit{stk}, \mathit{imp}, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle}$$

CHERIExp program state

A state $\langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc} \rangle$ of a program in **CHERIExp** consists of:

- code and data memories, $\mathcal{M}_c$ and $\mathcal{M}_d$ as defined earlier (We define $\mathcal{M}_d((\delta, s, e, o)) \stackrel{\mathsf{def}}{=} \mathcal{M}_d(s + o)$, and similarly for update expressions and for $\mathcal{M}_c$ with κ-labeled values. We also (ab)use the set membership notation $(\_, s, \_, \mathit{off}) \in X$ for $X \subseteq \mathbb{N}$ to mean $s + \mathit{off} \in X$. We use it to say that the capability points to an address within a certain range of addresses, say $\mathsf{pcc} \in \mathsf{dom}(\mathcal{M}_c)$.),
- a trusted call stack $stk : \overline{Cap \times Cap \times ModID \times FunID}$, which is a list of 4-tuples; each tuple consists of two capabilities, a module ID, and a function ID. The trusted call stack stores the history of the values of $\mathsf{ddc}, \mathsf{pcc}$ at the call locations. It also stores the identifier of the function (and module) that is being called. The storing of the function identifier allows us to build into the target language an assumption that it implements safe management of the data part of the stack frames.
- a map of imports $imp : ModID \to CapObj$ that for each module identifier, keeps an object capability ($CapObj = (\{\kappa\} \times \mathbb{N} \times \mathbb{N} \times \mathbb{Z}) \times (\{\delta\} \times \mathbb{N} \times \mathbb{N} \times \mathbb{Z}) \times (FunID \to \mathbb{N})$). An object capability consists of
  - a code capability that grants access to the module's code region in $\mathcal{M}_c$,
  - a data capability that grants access to the module's data region in $\mathcal{M}_d$,
  - and an offsets map, that for each function identifier in the module, specifies the offset within the module's code memory at which the function's code starts (i.e., this map of offsets describes the legitimate entry points to the module).
- a map of call frame sizes $\phi : (ModID \times FunID) \to (\mathbb{N} \times \mathbb{N})$ that for each function (given by the module identifier and the function identifier) gives the number of arguments and the number of local variables that this function allocates.
- three capability registers/variables:
  - $\mathsf{ddc} : \{\delta\} \times \mathbb{N} \times \mathbb{N} \times \mathbb{Z}$, the data capability (which specifies the region in the data memory $\mathcal{M}_d$ that is private to the active module),
  - $\mathsf{stc} : \{\delta\} \times \mathbb{N} \times \mathbb{N} \times \mathbb{Z}$, the stack-data capability (which specifies the region in the data memory $\mathcal{M}_d$ that corresponds to the current activation record),
  - and $\mathsf{pcc} : \{\kappa\} \times \mathbb{N} \times \mathbb{N} \times \mathbb{Z}$, the program counter capability (which specifies the region in the code memory $\mathcal{M}_c$ in which the currently-executing module is defined),
- a map $\mathsf{mstc} : ModID \to Cap$ that for each module identifier keeps the most recent value of its stack capability. This value is managed by the trusted $\mathsf{Cinvoke}$ and $\mathsf{Creturn}$ commands. The map records the most recent update to the stc capability. Updates to mstc made done by only the two commands $\mathsf{Cinvoke}$ and $\mathsf{Creturn}$.
- a marker $\mathsf{nalloc} : \mathbb{Z}$ that holds the first non-allocated address in $\mathcal{M}_d$ in the direction of growth of the heap (i.e., the dynamically-allocated segment of $\mathcal{M}_d$).

The type of **CHERIExp** program states is denoted by $TargetState = CodeMemory \times DataMemory \times \overline{(Cap \times Cap \times Cap)} \times (ModID \to CapObj) \times ((ModID \times FunID) \to (\mathbb{N} \times \mathbb{N})) \times (\{\delta\} \times \mathbb{N} \times \mathbb{N} \times \mathbb{Z}) \times (\{\delta\} \times \mathbb{N} \times \mathbb{N} \times \mathbb{Z}) \times (\{\kappa\} \times \mathbb{N} \times \mathbb{N} \times \mathbb{Z}) \times (ModID \to Cap) \times \mathbb{Z}$.

It is worth noting that the map of imports imp, and the code memory $\mathcal{M}_c$ are fixed at load time, and their contents are not modified by any instruction.

Lemma 2 (Reduction does not change call frame sizes, imports map or code memory)**.**

$$\forall s, s'.\ s \to^* s' \implies (s.\phi = s'.\phi \wedge s.imp = s'.imp \wedge s.\mathcal{M}_c = s'.\mathcal{M}_c)$$

Proof. By induction on the reduction steps and inspecting the rules of Figure 1. □

Lemma 3 (A reduction is enabled only on a valid program counter)**.**

$$\forall s.\ (\exists s'.\ s \to s' \vee s \succ_{\approx} s') \implies \vdash_{\kappa} s.\mathsf{pcc}$$

Proof. By inversion of $s \to s'$ (resp. $s \succ_{\approx} s'$). □

Definition 7 (Code region of an imports map)**.**

$$\mathsf{code\_region} : (\mathit{ModID} \to \mathit{CapObj}) \to 2^{\mathbb{Z}}$$

$$\mathsf{code\_region}(\mathit{imp}) \stackrel{\mathsf{def}}{=} \bigcup_{\mathit{mid} \in \mathsf{dom}(\mathit{imp})} [\mathit{imp}(\mathit{mid}).\mathsf{pcc}.\sigma, \mathit{imp}(\mathit{mid}).\mathsf{pcc}.e)$$

The syntax of the language enables the use of capabilities that are expressible in terms of two distinguished names, "`ddc`", and "`stc`" denoting *data capability*, and *stack capability* respectively. Notice that the program counter capability register is not addressable. Instead, the jump instruction can only increment the offset of the capability value in that register. Effectively, there is no way for code capabilities to live in memory. This is proved in Lemma 52.

Expressions in **CHERIExp** are denoted by the grammar

$$\begin{aligned}
\mathcal{E} ::= &\\
&\mathbb{Z}\\
&\mid \texttt{ddc}\\
&\mid \texttt{stc}\\
&\mid \texttt{inc}(\mathcal{E}, \mathcal{E})\\
&\mid \texttt{deref}(\mathcal{E})\\
&\mid \texttt{lim}(\mathcal{E}, \mathcal{E}, \mathcal{E})\\
&\mid \texttt{capType}(\mathcal{E})\\
&\mid \texttt{capStart}(\mathcal{E})\\
&\mid \texttt{capEnd}(\mathcal{E})\\
&\mid \texttt{capOff}(\mathcal{E})\\
&\mid \mathcal{E} \oplus \mathcal{E}
\end{aligned}$$

where $\oplus ::= + \mid - \mid *$, and $\mathbb{Z}$ is the set of integers. The forms `ddc` and `stc` are the distinguished names for the corresponding capabilities. An expression $\texttt{inc}(\mathcal{E}, \mathbb{Z})$ increments the offset of a capability value. An expression $\texttt{deref}(\mathcal{E})$ evaluates to the value at the memory address pointed to by a capability only if it is a valid capability according to Definition 2. The expression $\texttt{lim}(\mathcal{E}, \mathcal{E}, \mathcal{E})$ evaluates to a shrunk copy of the capability given by its first argument. The second and third arguments determine the new range of memory prescribed by the shrunk copy. The expressions $\texttt{capType}(\mathcal{E})$, $\texttt{capStart}(\mathcal{E})$, $\texttt{capEnd}(\mathcal{E})$, and $\texttt{capOff}(\mathcal{E})$ select the corresponding fields of the capability value given by evaluating their argument expression. The evaluation of expressions $\mathcal{E}$ to values $\mathcal{V}$ is given by rules of the form $\mathcal{E}, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow \mathcal{V}$ listed in fig. 2.

Lemma 4 (Expression evaluation cannot forge code capabilities)**.**

$$\begin{aligned}
&\forall a, s, \mathcal{E}.\\
&s.\mathsf{ddc} \notin \{\kappa\} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \ \wedge\ s.\mathsf{stc} \notin \{\kappa\} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}\\
&\wedge\ (s.\mathcal{M}_d(a) \neq (\kappa, \sigma_a, e_a, _))\\
&\wedge\ \mathcal{E}, s.\mathcal{M}_d, s.\mathsf{ddc}, s.\mathsf{stc}, s.\mathsf{pcc} \Downarrow v\\
&\implies\\
&v \neq (\kappa, _, _, _)
\end{aligned}$$

Figure 2: Evaluation of expressions $\mathcal{E}$ in **CHERIExp**

(evalconst)
$$\frac{n \in \mathbb{Z}}{n, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow n}$$

(evalddc)
$$\frac{}{\mathtt{ddc}, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow \mathsf{ddc}}$$

(evalstc)
$$\frac{}{\mathtt{stc}, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow \mathsf{stc}}$$

(evalCapType)
$$\frac{\begin{array}{cc}\mathcal{E}, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow v & v \in \mathbb{Z} \implies v' = 0 \\ v \in \{\kappa\} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \implies v' = 1 & v \in \{\delta\} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \implies v' = 2\end{array}}{\mathtt{capType}(\mathcal{E}), \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow v'}$$

(evalCapStart)
$$\frac{\mathcal{E}, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow v \qquad v = (x, s, e, \mathit{off}) \in \mathit{Cap} \qquad v' = s}{\mathtt{capStart}(\mathcal{E}), \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow v'}$$

(evalCapEnd)
$$\frac{\mathcal{E}, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow v \qquad v = (x, s, e, \mathit{off}) \in \mathit{Cap} \qquad v' = e}{\mathtt{capEnd}(\mathcal{E}), \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow v'}$$

(evalCapOff)
$$\frac{\mathcal{E}, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow v \qquad v = (x, s, e, \mathit{off}) \in \mathit{Cap} \qquad v' = \mathit{off}}{\mathtt{capOff}(\mathcal{E}), \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow v'}$$

(evalBinOp)
$$\frac{\mathcal{E}_1, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow v_1 \qquad v_1 \in \mathbb{Z} \qquad \mathcal{E}_2, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow v_2 \qquad v_2 \in \mathbb{Z} \qquad v' = v_1[\oplus]v_2}{\mathcal{E}_1 \oplus \mathcal{E}_2, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow v'}$$

(evalIncCap)
$$\frac{\begin{array}{c}\mathcal{E}, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow v \qquad \mathcal{E}_z, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow v_z \qquad v = (x, s, e, \mathit{off}) \in \mathit{Cap} \\ v_z \in \mathbb{Z} \qquad v' = (x, s, e, \mathit{off} + v_z)\end{array}}{\mathtt{inc}(\mathcal{E}, \mathcal{E}_z), \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow v'}$$

(evalDeref)
$$\frac{\mathcal{E}, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow v \qquad v = (x, s, e, \mathit{off}) \in \mathit{Cap} \qquad \vdash_\delta v \qquad v' = \mathcal{M}_d(s + \mathit{off})}{\mathtt{deref}(\mathcal{E}), \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow v'}$$

(evalLim)
$$\frac{\begin{array}{c}\mathcal{E}, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow v \qquad \mathcal{E}_s, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow s' \qquad \mathcal{E}_e, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow e' \\ s' \in \mathbb{Z} \qquad e' \in \mathbb{Z} \qquad v = (x, s, e, \_) \in \mathit{Cap} \qquad [s', e') \subseteq [s, e) \qquad v' = (x, s', e', 0)\end{array}}{\mathtt{lim}(\mathcal{E}, \mathcal{E}_s, \mathcal{E}_e), \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow v'}$$

Proof.
Easy by induction on the evaluation $\mathcal{E}, s.\mathcal{M}_d, s.\mathsf{ddc}, s.\mathsf{stc}, s.\mathsf{pcc} \Downarrow v$. □

1.2 Target setup, and initial and terminal states

Having defined the program state, we now define a target setup

$$\mathit{TargetSetup} \stackrel{\mathsf{def}}{=} \mathit{CodeMemory} \times \mathit{DataMemory} \times (\mathit{ModID} \to \mathit{CapObj}) \times (\mathit{ModID} \to \mathit{Cap}) \times ((\mathit{ModID} \times \mathit{FunID}) \to (\mathbb{N} \times \mathbb{N}))$$

as a tuple of code memory, data memory, imports map, stack capabilities map, and call-frame-sizes map.

Definition 8 (Disjoint object capabilities)**.**
For $c, c' \in \mathit{CapObj}$, $c \cap c' = \emptyset \stackrel{\mathsf{def}}{=} c.1 \cap c'.1 = \emptyset \wedge c.2 \cap c'.2 = \emptyset$ where disjointness of capabilities is as in Definition 3.

We hence define the linking $\ltimes : \mathit{TargetSetup} \to \mathit{TargetSetup} \to \mathtt{Option}(\mathit{TargetSetup})$ of two target setups t_1 and $t_2 \in \mathit{TargetSetup}$ as follows:

Definition 9 (Valid Linking)**.** *Valid linking of $t_1, t_2 \in \mathit{TargetSetup}$ is the component-wise disjoint union of code memories $t_1.\mathcal{M}_c, t_2.\mathcal{M}_c$, data memories $t_1.\mathcal{M}_d, t_2.\mathcal{M}_d$, imports maps $t_1.\mathit{imp}, t_2.\mathit{imp}$, and call-frame-sizes maps $t_1.\phi, t_2.\phi$ under the well-formedness conditions given by the rule valid-linking in Figure 3.*

Design choices for linking

The disjointness conditions on the address ranges and on the capability ranges in rule valid-linking are not surprising. But notice the non-commutativity of the valid linking operator $\ltimes$. The linking operator is designed to be aware of the context. All the context (i.e., untrusted) modules should be put on the left-hand side of $\ltimes$. The right-hand side operand should include all and only the trusted modules (if any). In case only untrusted modules are being linked, the order does not really matter.

There are **two noteworthy design choices** here that cause the linking operator $\ltimes$ to be non-commutative. They are expressed by the two conditions $\mathtt{max}(\mathsf{dom}(\mathcal{M}_{c1})) < \mathtt{min}(\mathsf{dom}(\mathcal{M}_{c2}))$ and $\mathtt{min}(\mathsf{dom}(\mathcal{M}_{d1})) > \mathtt{max}(\mathsf{dom}(\mathcal{M}_{d2}))$ of the rule valid-linking. The **first of these conditions is a necessary security measure**, while the second condition is required only as an artifact of our security proof techniques. The first condition ensures that the code memory segment of the context is always placed before the code memory segment of the trusted/compiled program. This ensures hiding (away from the context) information about the size of the code segment of the trusted program. The second condition ensures a reverse order on the data segments of the context and the program. This is a restriction that is required only as a result of our proof technique. In particular, we want to avoid reasoning about the scenario where the data layout of the program is shifted by a fixed amount of memory. The reason is that this places an unnecessary restriction on the way we have to construct a distinguishing context for two programs that we know are distinguishable.

An **initial** state of a **CHERIExp** program is one where the trusted stack is empty, the free memory marker captures the correct amount of dynamically-allocated memory (i.e., zero memory consumption), and the main function is about to start execution (the local stack of the main module contains the corresponding frame). We refer to a state s that is **initial** for setup t as $t \vdash_i s$.

Definition 10 (Initial state)**.** *A state s is initial for a target setup t (written $t \vdash_i s$ iff the preconditions described by rule initial-state in Figure 3 hold.*

Figure 3: Valid linking of two *TargetSetup*'s – Initial state of a *TargetSetup* – Execution state invariant

(valid-program)

$$\frac{\begin{array}{c} t = (\mathcal{M}_c, \mathcal{M}_d, imp, \mathsf{mstc}_t, \phi) \qquad modIDs = \mathsf{dom}(imp) = \mathsf{dom}(\mathsf{mstc}_t) \\ \forall mid \in modIDs.\ \vDash_\kappa imp(mid).\mathsf{pcc} \ \wedge\ \vDash_\delta imp(mid).\mathsf{ddc} \ \wedge\ \vDash_\delta \mathsf{mstc}(mid) \\ \mathsf{dom}(\mathcal{M}_c) = \biguplus_{mid \in modIDs} [imp(mid).\mathsf{pcc}.\sigma, imp(mid).\mathsf{pcc}.e) \\ \mathsf{dom}(\mathcal{M}_d) = \biguplus_{mid \in modIDs} [imp(mid).\mathsf{ddc}.\sigma, imp(mid).\mathsf{ddc}.e) \uplus [\mathsf{mstc}_t(mid).\sigma, \mathsf{mstc}_t(mid).e) \\ \mathsf{dom}(\mathcal{M}_d) \cap (-\infty, 0) = \emptyset \\ funIDs = [fid \ \mid\ fid \in \mathsf{dom}(imp(mid).offs) \ \wedge\ mid \in modIDs] \\ \mathsf{all\_distinct}(funIDs) \qquad \mathsf{dom}(\phi) = \{(mid, fid) \mid fid \in \mathsf{dom}(imp(mid).offs) \ \wedge\ mid \in modIDs\} \end{array}}{\vdash_{valid} t}$$

(valid-linking)

$$\frac{\begin{array}{c} \forall i \in \{1,2\}.\ t_i = (\mathcal{M}_{ci}, \mathcal{M}_{di}, imp_i, \mathsf{mstc}_i, \phi_i) \ \wedge\ \vdash_{valid} t_i \\ t = (\mathcal{M}_{c1} \uplus \mathcal{M}_{c2}, \mathcal{M}_{d1} \uplus \mathcal{M}_{d2}, imp_1 \uplus imp_2, \mathsf{mstc}_1 \uplus \mathsf{mstc}_2, \phi_1 \uplus \phi_2) \\ \mathtt{min}(\mathsf{dom}(\mathcal{M}_{d1})) > \mathtt{max}(\mathsf{dom}(\mathcal{M}_{d2})) \qquad \vdash_{valid} t \end{array}}{t_1 \ltimes t_2 = \lfloor t \rfloor}$$

(initial-state)

$$\frac{\begin{array}{c} \vdash_{valid} t \\ t = (\mathcal{M}_c, \mathcal{M}_{dt}, imp, \mathsf{mstc}_t, \phi) \qquad s = \langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc} \rangle \\ stk = \mathtt{nil} \qquad \mathcal{M}_d = \{a \mapsto 0 \mid a \in \mathsf{dom}(\mathcal{M}_{dt})\} \\ imp(mainMod) = (p, d, offs) \qquad \mathtt{main} \in \mathsf{dom}(offs) \\ \mathsf{pcc} = (\kappa, p.\sigma, p.e, offs(\mathtt{main})) \qquad \mathsf{ddc} = d \qquad \phi(mainMod, \mathtt{main}) = (nArgs, nLocal) \\ \mathsf{stc} = \mathsf{mstc}(mainMod) = (\delta, \mathsf{mstc}_t(mainMod).\sigma, \mathsf{mstc}_t(mainMod).e, nArgs + nLocal) \\ \forall mid \in modIDs \setminus \{mainMod\}.\ \mathsf{mstc}(mid) = (\delta, \mathsf{mstc}_t(mid).\sigma, \mathsf{mstc}_t(mid).e, 0) \\ \mathsf{nalloc} = -1 \end{array}}{t \vdash_i s}$$

(exec-state)

$$\frac{\begin{array}{c} t = (\mathcal{M}_c, \mathcal{M}_{dt}, imp, \mathsf{mstc}_t, \phi) \\ \vdash_{valid} t \qquad s = \langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc} \rangle \\ \vDash_\kappa \mathsf{pcc} \qquad \vDash_\delta \mathsf{ddc} \qquad \vDash_\delta \mathsf{stc} \qquad \mathsf{nalloc} < 0 \\ modIDs = \mathsf{dom}(imp) = \mathsf{dom}(\mathsf{mstc}) = \mathsf{dom}(\mathsf{mstc}_t) \qquad \forall mid \in modIDs.\ \vdash_\delta \mathsf{mstc}(mid) \\ \forall mid \in modIDs.\ \mathsf{mstc}(mid).off = \sum_{(\_,\_,mid,fid) \in stk} \phi(mid, fid).nArgs + \phi(mid, fid).nLocal + \\ (\mathtt{main} \in \mathsf{dom}(imp(mid).offs)\ ?\ \phi(mid, \mathtt{main}).nArgs + \phi(mid, \mathtt{main}).nLocal \ :\ 0) \\ \exists mid \in modIDs.\ \mathsf{pcc} \doteq imp(mid).\mathsf{pcc} \ \wedge\ \mathsf{ddc} \doteq imp(mid).\mathsf{ddc} \ \wedge\ \mathsf{stc} \doteq \mathsf{mstc}(mid) \\ \forall (dc, cc, \_, \_) \in \mathtt{elems}(stk).\ \vDash_\delta dc \ \wedge\ \vDash_\kappa cc \ \wedge \\ \exists mid \in modIDs.\ cc \doteq imp(mid).\mathsf{pcc} \ \wedge\ dc \doteq imp(mid).\mathsf{ddc} \\ \forall mid \in modIDs.\ \mathsf{mstc}(mid) \doteq \mathsf{mstc}_t(mid) \\ \mathsf{dom}(\mathcal{M}_d) = \bigcup_{mid \in modIDs} [imp(mid).\mathsf{ddc}.\sigma, imp(mid).\mathsf{ddc}.e) \cup [\mathsf{mstc}(mid).\sigma, \mathsf{mstc}(mid).e) \cup [\mathsf{nalloc}, -1) \\ \mathsf{reachable\_addresses}(\bigcup_{mid \in modIDs} \{imp(mid).\mathsf{ddc}, \mathsf{mstc}(mid)\}, \mathcal{M}_d) \subseteq \mathsf{dom}(\mathcal{M}_d) \\ \forall mid, a.\ a \in \mathsf{reachable\_addresses}(\{\mathsf{mstc}(mid), imp(mid).\mathsf{ddc}\}, \mathcal{M}_d) \implies \\ a \notin \bigcup_{mid' \in modIDs \setminus \{mid\}} [\mathsf{mstc}(mid').\sigma, \mathsf{mstc}(mid').e) \\ \forall a, mid \in modIDs.\ \mathcal{M}_d(a) = (\delta, \sigma, e, \_) \ \wedge\ [\sigma, e) \subseteq \mathsf{mstc}(mid) \implies a \in [\mathsf{mstc}(mid).\sigma, \mathsf{mstc}(mid).e) \\ \forall a.\ \mathcal{M}_d(a) \neq (\kappa, \sigma, e, \_) \qquad \forall a.\ \mathcal{M}_d(a) = (\delta, \sigma, e, \_) \implies [\sigma, e) \subseteq \mathsf{dom}(\mathcal{M}_d) \\ stk \neq \mathtt{nil} \implies \mathsf{pcc} \doteq imp(\mathtt{top}(stk).mid).\mathsf{pcc} \\ \forall i \in [1, \mathtt{length}(stk) - 1].\ stk(i).\mathsf{pcc} \doteq imp(stk(i-1).mid).\mathsf{pcc} \end{array}}{t \vdash_{exec} s}$$

Definition 11 (Initial state function).

$$
\begin{array}{l}
\mathsf{initial\_state}(t, mainMod) \stackrel{\mathsf{def}}{=} \\
\langle \\
t.\mathcal{M}_c, \\
\{a \mapsto 0 \mid a \in \mathsf{dom}(t.\mathcal{M}_d)\}, \\
\mathtt{nil}, \\
t.imp, \\
t.\phi, \\
t.imp(mainMod).\mathsf{ddc}, \\
(\delta, t.\mathsf{mstc}(mainMod).\sigma, t.\mathsf{mstc}(mainMod).e, t.\phi(mainMod, \mathtt{main}).nArgs + t.\phi(mainMod, \mathtt{main}).nLocal), \\
t.imp(mainMod).\mathsf{pcc}, \\
\{mid \mapsto (\delta, t.\mathsf{mstc}(mid).\sigma, t.\mathsf{mstc}(mid).e, 0) \mid mid \in \mathsf{dom}(t.\mathsf{mstc}) \setminus \{mainMod\}\ \} \uplus \\
\quad \{mainMod \mapsto (\delta, t.\mathsf{mstc}(mainMod).\sigma, t.\mathsf{mstc}(mainMod).e, t.\phi(mainMod, \mathtt{main}).nArgs + t.\phi(mainMod, \mathtt{main}).nLocal)\}, \\
-1 \\
\rangle
\end{array}
$$

Definition 12 (Main module).

$$\mathsf{main\_module}(t) = mid \iff \mathtt{main} \in \mathsf{dom}(t.imp(mid).offs)$$

Claim 1 (The function $\mathsf{initial\_state}$ and the judgment $\vdash_i$ are compatible).

$$
\begin{array}{l}
\forall t, s, mainMod. \\
\mathsf{initial\_state}(t, mainMod) = s \wedge \\
\vdash_{valid} t \wedge \\
\mathtt{main} \in \mathsf{dom}(t.imp(mainMod).offs) \\
\implies \\
t \vdash_i s
\end{array}
$$

Proof. Follows easily after unfolding the assumptions usnig Definition 11, and inversion of the goal using rule initial-state. ☐

Definition 13 (Terminal state).
A program state $s = \langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle$ *is* ***terminal****, written* $\vdash_t s$ *iff* $\mathcal{M}_c(\mathsf{pcc}) = \mathsf{Exit}$.

Definition 14 (Addition of an offset ω to the data memory).

$$\mathcal{M}_d + \omega \stackrel{\mathsf{def}}{=} \{a + \omega \mapsto \mathcal{M}_d(a) \mid a \in \mathsf{dom}(\mathcal{M}_d)\}$$

Definition 15 (Addition of an offset ω to the imp map).

$$imp + \omega \stackrel{\mathsf{def}}{=} \{mid \mapsto (\mathsf{pcc}, (\delta, \mathsf{ddc}.\sigma + \omega, \mathsf{ddc}.e + \omega, \mathsf{ddc}.off), offs) \mid (mid \mapsto (\mathsf{pcc}, \mathsf{ddc}, offs)) \in imp\}$$

Definition 16 (Addition of an offset ω to a program t).

$$t + \omega \stackrel{\mathsf{def}}{=} (t.\mathcal{M}_c, t.\mathcal{M}_d + \omega, t.imp + \omega, t.\mathsf{mstc}, t.\phi)$$

Given two target setups $t_1, t_2 \in TargetSetup$, we write $t_1[t_2] \Downarrow$ (convergence) to mean that $t_1 \ltimes t_2$ is defined, that there is at least one valid initial state, and that for all possible initial states, there is a reduction to a terminal state.

Definition 17 (Linkability, loadability, and convergence of execution in the target language)**.**

$$\begin{aligned}\nabla \vdash \mathbb{C}[t_1] \Downarrow \;\stackrel{\mathsf{def}}{=}\;& \exists t'.\ \mathbb{C} \ltimes t_1 = \lfloor t' \rfloor \;\wedge \\ & \exists s_t.\ \mathsf{initial\_state}(t', \mathsf{main\_module}(t')) \rightarrow^*_{\nabla} s_t \;\wedge\; \vdash_t s_t\end{aligned}$$

Definition 18 (Target contextual equivalence)**.**

$$t_1 \simeq_{\nabla} t_2 \stackrel{\mathsf{def}}{=} \forall \mathbb{C}.\ \nabla \vdash \mathbb{C}[t_1] \Downarrow \iff \nabla \vdash \mathbb{C}[t_2] \Downarrow$$

Definition 19 (Valid execution state)**.** *A state s is a valid execution state for a target setup t (written $t \vdash_{exec} s$) iff the preconditions described by rule exec-state in Figure 3 hold.*

Lemma 5 (Initial states are valid execution states)**.** $\forall t, s.\ t \vdash_i s \implies t \vdash_{exec} s$

We skip the details here. By inversion of our goal using exec-state, all subgoals follow easily from preconditions of the rule initial-state.

1.3 Memory Reachability

Definition 20 (Accessible addresses)**.**

$$\begin{aligned}&\mathsf{access}_{\mathcal{M}_d} : 2^{\mathbb{Z}} \rightarrow 2^{\mathbb{Z}} \\ &\mathsf{access}_{\mathcal{M}_d}\ A \stackrel{\mathsf{def}}{=} A \cup \bigcup_{a \in A,\ \mathcal{M}_d(a) = (\delta, s, e, \_)} [s, e)\end{aligned}$$

Definition 21 (k-accessible addresses)**.**

$$\begin{aligned}&\mathsf{access}_{0,\mathcal{M}_d}\ A \stackrel{\mathsf{def}}{=} A \\ &\mathsf{access}_{k+1,\mathcal{M}_d} \stackrel{\mathsf{def}}{=} \mathsf{access}_{\mathcal{M}_d}(\mathsf{access}_{k,\mathcal{M}_d} A)\end{aligned}$$

Definition 22 (Reachable addresses)**.**

$$\begin{aligned}&\mathsf{reachable\_addresses} : (2^{\{\delta\} \times \mathbb{N} \times \mathbb{N} \times \mathbb{Z}} \times \mathit{DataMemory}) \rightarrow 2^{\mathbb{Z}} \\ &\mathsf{reachable\_addresses}(C, \mathcal{M}_d) \stackrel{\mathsf{def}}{=} \bigcup_{k \in [0, |\mathcal{M}_d|]} \mathsf{access}_{k,\mathcal{M}_d}\Big(\bigcup_{c \in C} [c.s, c.e)\Big) \\ &\mathsf{reachable\_addresses\_closure} : (2^{\mathbb{Z}} \times \mathit{DataMemory}) \rightarrow 2^{\mathbb{Z}} \\ &\mathsf{reachable\_addresses\_closure}(A, \mathcal{M}_d) \stackrel{\mathsf{def}}{=} \bigcup_{k \in [0, |\mathcal{M}_d|]} \mathsf{access}_{k,\mathcal{M}_d} A\end{aligned}$$

Lemma 6 (Reachability is not affected by offsets, only bounds)**.**

$$\forall c, \mathcal{M}_d, c'.\ c \doteq c' \implies \mathsf{reachable\_addresses}(\{c\}, \mathcal{M}_d) = \mathsf{reachable\_addresses}(\{c'\}, \mathcal{M}_d)$$

Proof. Immediate by Definitions 6 and 22. □

Lemma 7 ($\mathsf{access}_{\mathcal{M}_d}$ is expansive)**.**

$$\forall A, \mathcal{M}_d.\ \mathsf{access}_{\mathcal{M}_d} A \supseteq A$$

Proof. Immediate by Definition 20 and the reflexivity of $\supseteq$. □

Lemma 8 ($\mathsf{access}_{n,\mathcal{M}_d}$ is expansive)**.**

$$\forall n, A, \mathcal{M}_d.\ \mathsf{access}_{n,\mathcal{M}_d} A \supseteq A$$

Proof. We prove it by induction on n.

- **Base case** $n = 0$**:**

  Immediate by Definition 21; $\mathsf{access}_{0,\mathcal{M}_d} A = A \supseteq A$.

- **Inductive case:**

  Assuming for an arbitrary k that $\forall A.\ \mathsf{access}_{k,\mathcal{M}_d} A \supseteq A$, we show for an arbitrary B that $\mathsf{access}_{k+1,\mathcal{M}_d} B \supseteq B$.

  By Definition 21, our goal becomes $\mathsf{access}_{\mathcal{M}_d}(\mathsf{access}_{k,\mathcal{M}_d} B) \supseteq B$.

  But by assumption (the induction hypothesis), we have by universal instantiation that $\mathsf{access}_{k,\mathcal{M}_d} B \supseteq B$.

  And by Lemma 7, we have $\mathsf{access}_{\mathcal{M}_d}(\mathsf{access}_{k,\mathcal{M}_d}(B)) \supseteq \mathsf{access}_{k,\mathcal{M}_d}(B)$.

  So, by transitivity of $\supseteq$, we have our goal.

□

Lemma 9 (Fixed points lead to convergence of $\mathsf{access}_{k,\mathcal{M}_d}$)**.**

$$\forall k, \mathcal{M}_d, A.\ k > 0 \implies (\mathsf{access}_{k,\mathcal{M}_d} A = A \implies \mathsf{access}_{k+1,\mathcal{M}_d} A = A)$$

Proof.

- We fix arbitrary $k, A, \mathcal{M}_d$ and assume both antecedents.
- By the assumptions and Definition 21, we have (*):
  $A = \mathsf{access}_{\mathcal{M}_d}(\mathsf{access}_{k-1,\mathcal{M}_d} A)$.
- Then by expansiveness of $\mathsf{access}_{\mathcal{M}_d}$ (Lemma 7), we obtain:
  $A = \mathsf{access}_{\mathcal{M}_d}(\mathsf{access}_{k-1,\mathcal{M}_d} A) \supseteq \mathsf{access}_{k-1,\mathcal{M}_d} A$.
- We also have by expansiveness of $\mathsf{access}_{k-1,\mathcal{M}_d}$ (Lemma 8) that:
  $A = \mathsf{access}_{\mathcal{M}_d}(\mathsf{access}_{k-1,\mathcal{M}_d} A) \supseteq \mathsf{access}_{k-1,\mathcal{M}_d} A \supseteq A$.
- Thus, we conclude:
  $\mathsf{access}_{k-1,\mathcal{M}_d} A = A$.
- We substitute this equality in (*) to get (**):
  $\mathsf{access}_{\mathcal{M}_d} A = A$.
- Our goal is to show the consequent of the lemma statement: $\mathsf{access}_{k+1,\mathcal{M}_d} A = A$.
- By Definition 21, our goal becomes $\mathsf{access}_{\mathcal{M}_d}(\mathsf{access}_{k,\mathcal{M}_d} A) = A$.
- And by the assumption $\mathsf{access}_{k,\mathcal{M}_d} A = A$, our goal becomes $\mathsf{access}_{\mathcal{M}_d} A = A$.
- But this goal is exactly statement (**) that we already obtained above.

□

Lemma 10 (In an empty memory, only the starting addresses are reachable)**.**

$$\forall C, \mathcal{M}_d.$$
$$(\forall v.\ v \in \mathsf{range}(\mathcal{M}_d) \implies v \neq (\delta, \_, \_, \_)) \implies \mathsf{reachable\_addresses}(C, \mathcal{M}_d) = \bigcup_{c \in C} [c.\sigma, c.e)$$

Proof. Immediate by Definitions 20 to 22. $\square$

Lemma 11 (k-accessibility either adds a new memory address or a fixed point has been reached)**.**

$$\forall k, A, \mathcal{M}_d.\ k > 0 \implies$$
$$\mathsf{access}_{k+1,\mathcal{M}_d} A \supsetneq \mathsf{access}_{k,\mathcal{M}_d} A \implies$$
$$(\exists a.\ a \in \mathsf{dom}(\mathcal{M}_d)\ \wedge\ a \in \mathsf{access}_{k,\mathcal{M}_d} A \setminus \mathsf{access}_{k-1,\mathcal{M}_d} A)$$

Proof. We fix arbitrary k, A and $\mathcal{M}_d$, and we assume both antecedents.

- By Definitions 20 and 21, we have from the assumption that:
  $\mathsf{access}_{k,\mathcal{M}_d} A \cup \bigcup_{a \in \mathsf{access}_{k,\mathcal{M}_d} A,\ \mathcal{M}_d(a)=(\delta,s,e,\_)} [s,e) \supset \mathsf{access}_{k,\mathcal{M}_d} A$
- So the set $\bigcup_{a \in \mathsf{access}_{k,\mathcal{M}_d} A,\ \mathcal{M}_d(a)=(\delta,s,e,\_)} [s,e) \neq \emptyset$, and in particular:
  (*) $\exists a, a'.\ a \in \mathsf{access}_{k,\mathcal{M}_d} A \wedge \mathcal{M}_d(a) = (\delta, s, e, \_) \wedge a' \in [s,e) \wedge a' \notin \mathsf{access}_{k,\mathcal{M}_d} A$.
- Suppose for the sake of contradiction that $a \in \mathsf{access}_{k-1,\mathcal{M}_d} A$.
  - By Definitions 20 and 21, we know that
    (**) $\mathsf{access}_{k,\mathcal{M}_d} A = \mathsf{access}_{k-1,\mathcal{M}_d} A \cup \bigcup_{a \in \mathsf{access}_{k-1,\mathcal{M}_d} A,\ \mathcal{M}_d(a)=(\delta,s,e,\_)} [s,e)$
  - From (*), we know that our obtained a satisfies $\mathcal{M}_d(a) = (\delta, s, e, \_)$ and that our a' satisfies $a' \in [s,e)$.
  - Thus, we conclude that $a' \in \bigcup_{a \in \mathsf{access}_{k-1,\mathcal{M}_d} A,\ \mathcal{M}_d(a)=(\delta,s,e,\_)} [s,e)$.
  - Thus by (**), $a' \in \mathsf{access}_{k,\mathcal{M}_d} A$. But this contradicts conjunct $a' \notin \mathsf{access}_{k,\mathcal{M}_d} A$ of (*).
- Thus, necessarily $a \notin \mathsf{access}_{k-1,\mathcal{M}_d} A$.
- Thus, the obtained a from (*) satisfies our goal:
  $a \in \mathsf{dom}(\mathcal{M}_d) \wedge a \in \mathsf{access}_{k,\mathcal{M}_d} A \setminus \mathsf{access}_{k-1,\mathcal{M}_d} A$.

$\square$

Lemma 12 (k-accessibility set contains at least k mapped addresses)**.**

$$\forall k, A, \mathcal{M}_d.$$
$$\mathsf{access}_{k+1,\mathcal{M}_d} A \supsetneq \mathsf{access}_{k,\mathcal{M}_d} A \implies$$
$$|\{a \mid a \in \mathsf{access}_{k,\mathcal{M}_d} A \wedge a \in \mathsf{dom}(\mathcal{M}_d)\}| > k$$

Proof. We fix arbitrary A and $\mathcal{M}_d$.
We prove it by induction on k.

- **Base case ($k = 0$):**
  Our goal is: $|\{a \mid a \in \mathsf{access}_{0,\mathcal{M}_d} A \wedge a \in \mathsf{dom}(\mathcal{M}_d)\}| > 0$.
  We have by assuming the antecedent that $\mathsf{access}_{1,\mathcal{M}_d} A \supsetneq \mathsf{access}_{0,\mathcal{M}_d} A$.
  By Definitions 20 and 21, this simplifies to $A \cup \bigcup_{a \in A,\ \mathcal{M}_d(a)=(\delta,s,e,\_)} [s,e) \supsetneq A$.
  Thus, $\exists a, a'.\ a \in A \wedge \mathcal{M}_d(a) = (\delta, s, e, \_) \wedge a' \in [s,e)$.
  Thus, the set $\{a \mid a \in A \wedge a \in \mathsf{dom}(\mathcal{M}_d)\} \neq \emptyset$.
  By Definition 21, we substitute A by $\mathsf{access}_{0,\mathcal{M}_d} A$ to get our goal:
  $\{a \mid a \in \mathsf{access}_{0,\mathcal{M}_d} A \wedge a \in \mathsf{dom}(\mathcal{M}_d)\} \neq \emptyset$, i.e.,
  $|\{a \mid a \in \mathsf{access}_{0,\mathcal{M}_d} A \wedge a \in \mathsf{dom}(\mathcal{M}_d)\}| > 0$

- **Inductive case ($k > 0$):**

  Here, we have by the inductive hypothesis:
  (*) $\mathsf{access}_{k,\mathcal{M}_d} A \supsetneq \mathsf{access}_{k-1,\mathcal{M}_d} A \implies |\{a \mid a \in \mathsf{access}_{k-1,\mathcal{M}_d} A \wedge a \in \mathsf{dom}(\mathcal{M}_d)\}| > k-1$

  We have by assuming the antecedent that $\mathsf{access}_{k+1,\mathcal{M}_d} A \supsetneq \mathsf{access}_{k,\mathcal{M}_d} A$.

  Thus by Lemma 11, we have that:
  (**) $\exists a^*.\ a^* \in \mathsf{dom}(\mathcal{M}_d) \ \wedge \ a^* \in \mathsf{access}_{k,\mathcal{M}_d} A \setminus \mathsf{access}_{k-1,\mathcal{M}_d} A$.

  The latter gives us by the definition of $\supsetneq$ that $\mathsf{access}_{k,\mathcal{M}_d} A \supsetneq \mathsf{access}_{k-1,\mathcal{M}_d} A$.

  Thus, by instantiating the induction hypothesis (*), we get:
  (***) $|\{a \mid a \in \mathsf{access}_{k-1,\mathcal{M}_d} A \wedge a \in \mathsf{dom}(\mathcal{M}_d)\}| > k-1$.

  We rewrite it as: (***) $|\{a \mid a \in \mathsf{access}_{k-1,\mathcal{M}_d} A \wedge a \in \mathsf{dom}(\mathcal{M}_d)\}| \geq k$

  But by (**), we already also obtained a^* with:
  $a^* \in \mathsf{dom}(\mathcal{M}_d) \wedge a^* \in \mathsf{access}_{k,\mathcal{M}_d} A \setminus \mathsf{access}_{k-1,\mathcal{M}_d} A$.

  Thus, we can conclude that:
  $\{a \mid a \in \mathsf{access}_{k,\mathcal{M}_d} A \wedge a \in \mathsf{dom}(\mathcal{M}_d)\} \supseteq \{a \mid a \in \mathsf{access}_{k-1,\mathcal{M}_d} A \wedge a \in \mathsf{dom}(\mathcal{M}_d)\} \uplus \{a^*\}$. (The left operand of $\uplus$ is contained by expansiveness (Lemma 8)).

  Thus, we have:
  $|\{a \mid a \in \mathsf{access}_{k,\mathcal{M}_d} A \wedge a \in \mathsf{dom}(\mathcal{M}_d)\}| \geq |\{a \mid a \in \mathsf{access}_{k-1,\mathcal{M}_d} A \wedge a \in \mathsf{dom}(\mathcal{M}_d)\}| + |\{a^*\}|$.

  Thus, by (***) and simplification:
  $|\{a \mid a \in \mathsf{access}_{k,\mathcal{M}_d} A \wedge a \in \mathsf{dom}(\mathcal{M}_d)\}| \geq k+1 > k$

$\square$

Lemma 13 ($|\mathcal{M}_d|$-accessibility suffices)**.**

$$\forall A, \mathcal{M}_d, k.\ k \geq 0 \implies \mathsf{access}_{|\mathcal{M}_d|+k,\mathcal{M}_d} A = \mathsf{access}_{|\mathcal{M}_d|,\mathcal{M}_d} A$$

Proof. We fix arbitrary A and $\mathcal{M}_d$, and prove it by induction on k.

- **Base case ($k = 0$):**

  Holds by reflexivity.

- **Inductive case ($k > 0$):**

  We assume $\mathsf{access}_{|\mathcal{M}_d|+k-1,\mathcal{M}_d} A = \mathsf{access}_{|\mathcal{M}_d|,\mathcal{M}_d} A$

  Suppose for the sake of contradiction that $\mathsf{access}_{|\mathcal{M}_d|+k,\mathcal{M}_d} A \supsetneq \mathsf{access}_{|\mathcal{M}_d|+k-1,\mathcal{M}_d} A$.

  Then, we know by Lemma 12 that necessarily
  $|\{a \mid a \in \mathsf{access}_{|\mathcal{M}_d|+k-1,\mathcal{M}_d} A \wedge a \in \mathsf{dom}(\mathcal{M}_d)\}| > |\mathcal{M}_d| + k - 1$.

  But $k > 0$. Thus, $k - 1 \geq 0$.

  So, our statement says
  $|\{a \mid a \in \mathsf{access}_{|\mathcal{M}_d|+k-1,\mathcal{M}_d} A \wedge a \in \mathsf{dom}(\mathcal{M}_d)\}| > |\mathcal{M}_d|$.

  But this is immediately a contradiction because
  $|\{a \mid a \in \mathsf{dom}(\mathcal{M}_d)\}| = |\mathcal{M}_d|$, and
  $\{a \mid a \in \mathsf{dom}(\mathcal{M}_d)\} \supseteq \{a \mid a \in \mathsf{access}_{|\mathcal{M}_d|+k-1,\mathcal{M}_d} A \wedge a \in \mathsf{dom}(\mathcal{M}_d)\}$.

  Thus, necessarily by our contradictory assumption and Lemma 8:
  $\mathsf{access}_{|\mathcal{M}_d|+k,\mathcal{M}_d} A = \mathsf{access}_{|\mathcal{M}_d|+k-1,\mathcal{M}_d} A$.

  So, by substitution from our inductive hypothesis, we get our goal:
  $\mathsf{access}_{|\mathcal{M}_d|+k,\mathcal{M}_d} A = \mathsf{access}_{|\mathcal{M}_d|,\mathcal{M}_d} A$

$\square$

Lemma 14 (Invariance to non-δ-capability values)**.**

$$\forall C, \mathcal{M}_d, a, v.$$
$$v \neq (\delta, \_, \_, \_) \ \wedge \ \mathcal{M}_d(a) = v$$
$$\implies \mathsf{reachable\_addresses}(C, \mathcal{M}_d) = \mathsf{reachable\_addresses}(C, \mathcal{M}_d[a \mapsto 0])$$

Proof.

- We fix arbitrary $C, \mathcal{M}_d, a$, and v. We assume the antecedents $v \neq (\delta, \_, \_, \_) \ \wedge \ \mathcal{M}_d(a) = v$.
- Our goal is $\mathsf{reachable\_addresses}(C, \mathcal{M}_d) = \mathsf{reachable\_addresses}(C, \mathcal{M}_d[a \mapsto 0])$.
- By Definition 22, it suffices to show that: $\forall n.\ \mathsf{access}_{n,\mathcal{M}_d} A = \mathsf{access}_{n,\mathcal{M}_d[a\mapsto 0]} A$.
- We prove it by induction on n.
  - **Base case ($n = 0$):** By Definition 21, $\mathsf{access}_{0,\mathcal{M}_d} A = \mathsf{access}_{0,\mathcal{M}_d[a\mapsto 0]} A = A$.
  - **Inductive case ($n > 0$):** By the induction hypothesis, we have: $\mathsf{access}_{n-1,\mathcal{M}_d} A = \mathsf{access}_{n-1,\mathcal{M}_d[a\mapsto 0]} A = s_{ind}$.
    By unfolding Definition 21, our goal becomes (after substitution): $\mathsf{access}_{\mathcal{M}_d} s_{ind} = \mathsf{access}_{\mathcal{M}_d[a\mapsto 0]} s_{ind}$.
    By Definition 20, our goal is:
    $s_{ind} \cup \bigcup_{a' \in s_{ind},\ \mathcal{M}_d(a') = (\delta, s, e, \_)} [s, e] = s_{ind} \cup \bigcup_{a' \in s_{ind},\ \mathcal{M}_d[a\mapsto 0](a') = (\delta, s, e, \_)} [s, e]$
    Thus, it suffices to show that:
    $\forall a', s, e.\ \in s_{ind}.\ \mathcal{M}_d(a') = (\delta, s, e, \_) \iff \mathcal{M}_d[a \mapsto 0](a') = (\delta, s, e, \_)$.
    We prove it for an arbitrary $a', s, , e$ by distinguishing the following cases:
    - **Case $a' \neq a$:** In this case, by the definition (stability) of the function update operator, we have: $\mathcal{M}_d(a') = \mathcal{M}_d[a \mapsto 0](a')$, which implies our goal: $\mathcal{M}_d(a') = (\delta, s, e, \_) \iff \mathcal{M}_d[a \mapsto 0](a') = (\delta, s, e, \_)$.
    - **Case $a' = a$:** "$\implies$": In this case, suppose $\mathcal{M}_d(a) = (\delta, s, e, \_)$. Then, we get a contradiction to our assumption that $v \neq (\delta, \_, \_, \_)$. So, any goal is provable.
      "$\impliedby$": In this case, suppose $\mathcal{M}_d[a \mapsto 0](a) = (\delta, s, e, \_)$. This is immediately a contradiction by the disjointness of $\mathbb{Z}$ and $\{\delta\} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$. So, any goal is provable.

□

Lemma 15 (Overwriting a non-δ-capability value does not shrink the accessibility set)**.**

$$\forall k, \mathcal{M}_d, A, a, v.\ \mathcal{M}_d(a) \neq (\delta, \_, \_, \_) \implies \mathsf{access}_{k,\mathcal{M}_d} A \subseteq \mathsf{access}_{k,\mathcal{M}_d[a\mapsto v]} A$$

Proof. We fix arbitrary $\mathcal{M}_d$, A, and v, and assume the antecedent. We prove it by induction on k.

- **Base case ($k = 0$):**

  In this case, our goal is to show that $\mathsf{access}_{0,\mathcal{M}_d} A \subseteq \mathsf{access}_{0,\mathcal{M}_d[a\mapsto v]} A$. By Definition 21, we have: $\mathsf{access}_{0,\mathcal{M}_d} A = \mathsf{access}_{0,\mathcal{M}_d[a\mapsto v]} A = A$ which satisfies our goal.

- **Inductive case ($k > 0$):**
  Here, the I.H. gives us $\mathsf{access}_{k-1,\mathcal{M}_d} A \subseteq \mathsf{access}_{k-1,\mathcal{M}_d[a\mapsto v]} A$.
  We pick an arbitrary $a' \in \mathsf{access}_{k,\mathcal{M}_d} A$.
  By Definitions 20 and 21, we distinguish two cases:
  - **Case $a' \in \mathsf{access}_{k-1,\mathcal{M}_d} A$:**
    In this case, by the I.H., we know $a' \in \mathsf{access}_{k-1,\mathcal{M}_d[a\mapsto v]} A$.
    So by expansiveness (Lemma 8), we have our goal.
  - **Case $a' \in \bigcup_{a'' \in \mathsf{access}_{k-1,\mathcal{M}_d} A, \mathcal{M}_d(a'')=(\delta,s,e,\_)} [s,e)$:**
    In this case, we obtain a'' where $a'' \in \mathsf{access}_{k-1,\mathcal{M}_d} A \wedge \mathcal{M}_d(a'') = (\delta, s, e, \_) \wedge a' \in [s, e)$.
    We now distinguish two cases for a'':
    * **Case $a'' = a$:**
      This case is impossible because by assumption we know $\mathcal{M}_d(a) \neq (\delta, \_, \_, \_)$.
    * **Case $a'' \neq a$:**
      In this case, we know that $\mathcal{M}_d[a \mapsto v](a'') = \mathcal{M}_d(a'') = (\delta, s, e, \_)$.
      Thus, we have that
      $a' \in \bigcup_{a'' \in \mathsf{access}_{k-1,\mathcal{M}_d} A, \mathcal{M}_d[a\mapsto v](a'')=(\delta,s,e,\_)} [s,e)$.
      But by the I.H., this gives us:
      $a' \in \bigcup_{a'' \in \mathsf{access}_{k-1,\mathcal{M}_d[a\mapsto v]} A, \mathcal{M}_d[a\mapsto v](a'')=(\delta,s,e,\_)} [s,e)$.
      By Definition 21 of $a' \in \mathsf{access}_{k,\mathcal{M}_d[a\mapsto v]} A$, our goal is satisfied.

□

Lemma 16 (Additivity of $\mathsf{access}_{\mathcal{M}_d}$)**.**

$$\forall A_1, A_2, \mathcal{M}_d.\ \mathsf{access}_{\mathcal{M}_d}(A_1 \cup A_2) = \mathsf{access}_{\mathcal{M}_d} A_1 \cup \mathsf{access}_{\mathcal{M}_d} A_2$$

Proof.

- By Definition 20, our goal becomes:
  $A_1 \cup A_2 \cup \bigcup_{a \in A_1 \cup A_2,\ \mathcal{M}_d(a)=(\delta,s,e,\_)} [s,e) = A_1 \cup \bigcup_{a \in A_1,\ \mathcal{M}_d(a)=(\delta,s,e,\_)} [s,e) \cup A_2 \cup \bigcup_{a \in A_2,\ \mathcal{M}_d(a)=(\delta,s,e,\_)} [s,e)$
- Then, it suffices to show that:
  $\bigcup_{a \in A_1 \cup A_2,\ \mathcal{M}_d(a)=(\delta,s,e,\_)} [s,e) = \bigcup_{a \in A_1,\ \mathcal{M}_d(a)=(\delta,s,e,\_)} [s,e) \cup \bigcup_{a \in A_2,\ \mathcal{M}_d(a)=(\delta,s,e,\_)} [s,e)$
- The above goal can be shown as follows:
  - Pick an arbitrary $a' \in \bigcup_{a \in A_1 \cup A_2,\ \mathcal{M}_d(a)=(\delta,s,e,\_)} [s,e)$.
  - Notice that by the definition of $\cup$, this is equivalent to:
    $\exists a.\ a \in A_1 \cup A_2 \wedge \mathcal{M}_d(a) = (\delta, s, e, \_) \wedge a' \in [s, e)$
  - By the definition of $a \in A_1 \cup A_2$, this is equivalent to:
    $\exists a.\ (a \in A_1 \vee a \in A_2) \wedge \mathcal{M}_d(a) = (\delta, s, e, \_) \wedge a' \in [s, e)$
  - By distributivity, this is equivalent to:
    $\exists a.\ (a \in A_1 \wedge \mathcal{M}_d(a) = (\delta, s, e, \_) \wedge a' \in [s, e)) \vee (a \in A_2 \wedge \mathcal{M}_d(a) = (\delta, s, e, \_) \wedge a' \in [s, e))$
  - By folding back the definition of $\bigcup$, this is equivalent to:
    $a' \in \bigcup_{a \in A_1,\ \mathcal{M}_d(a)=(\delta,s,e,\_)} [s,e) \cup \bigcup_{a \in A_2,\ \mathcal{M}_d(a)=(\delta,s,e,\_)} [s,e)$

This concludes the proof of our sufficient goal.

□

Lemma 17 (Additivity of $\mathsf{access}_{k,\mathcal{M}_d}$)**.**

$$\forall k, A_1, A_2, \mathcal{M}_d.\ \mathsf{access}_{k,\mathcal{M}_d} A_1 \cup A_2 = \mathsf{access}_{k,\mathcal{M}_d} A_1 \cup \mathsf{access}_{k,\mathcal{M}_d} A_2$$

Proof. We fix arbitrary A_1, A_2, and $\mathcal{M}_d$, and prove it by induction on k.

- **Base case** ($k = 0$)**:**

  Our goal is to show that $\mathsf{access}_{0,\mathcal{M}_d} A_1 \cup A_2 = \mathsf{access}_{0,\mathcal{M}_d} A_1 \cup \mathsf{access}_{0,\mathcal{M}_d} A_2$.

  By unfolding Definition 21, it becomes $A_1 \cup A_2 = A_1 \cup A_2$ which holds by reflexivity.

- **Inductive case** ($k > 0$)**:**

  By the induction hypothesis, we have:

  $\mathsf{access}_{k-1,\mathcal{M}_d} A_1 \cup A_2 = \mathsf{access}_{k-1,\mathcal{M}_d} A_1 \cup \mathsf{access}_{k-1,\mathcal{M}_d} A_2$.

  By Definition 21, our goal is to show that:
  $\mathsf{access}_{\mathcal{M}_d}(\mathsf{access}_{k-1,\mathcal{M}_d} A_1 \cup A_2) = \mathsf{access}_{\mathcal{M}_d}(\mathsf{access}_{k-1,\mathcal{M}_d} A_1) \cup \mathsf{access}_{\mathcal{M}_d}(\mathsf{access}_{k-1,\mathcal{M}_d} A_2)$

  By substitution using the induction hypothesis, our goal becomes:
  $\mathsf{access}_{\mathcal{M}_d}(\mathsf{access}_{k-1,\mathcal{M}_d} A_1 \cup \mathsf{access}_{k-1,\mathcal{M}_d} A_2) = \mathsf{access}_{\mathcal{M}_d}(\mathsf{access}_{k-1,\mathcal{M}_d} A_1) \cup \mathsf{access}_{\mathcal{M}_d}(\mathsf{access}_{k-1,\mathcal{M}_d} A_2)$

  This goal can be directly satisfied by Lemma 16.

□

Lemma 18 (Additivity of $\mathsf{reachable\_addresses}$ in the first argument)**.**

$\forall C_1, C_2, \mathcal{M}_d.$
$\mathsf{reachable\_addresses}(C_1 \cup C_2, \mathcal{M}_d) = \mathsf{reachable\_addresses}(C_1, \mathcal{M}_d) \cup \mathsf{reachable\_addresses}(C_2, \mathcal{M}_d)$

Proof.

- We fix arbitrary C_1, C_2, and $\mathcal{M}_d$.

- By Definition 22, our goal becomes
  $$\bigcup_{n \in [0,|\mathcal{M}_d|]} \mathsf{access}_{n,\mathcal{M}_d}(\mathsf{addr}(C_1 \cup C_2)) = \bigcup_{n \in [0,|\mathcal{M}_d|]} \mathsf{access}_{n,\mathcal{M}_d}(\mathsf{addr}(C_1)) \cup \bigcup_{n \in [0,|\mathcal{M}_d|]} \mathsf{access}_{n,\mathcal{M}_d}(\mathsf{addr}(C_2))$$

  where $\mathsf{addr}(C) \overset{\mathsf{def}}{=} \bigcup_{c \in C} [c.s, c.e)$.

- ***Claim*** ***(*addr *is additive):*** $\mathsf{addr}(C_1 \cup C_2) = \mathsf{addr}(C_1) \cup \mathsf{addr}(C_2)$.

- It suffices for our goal to show that:
  $\forall n.\ \mathsf{access}_{n,\mathcal{M}_d}(\mathsf{addr}(C_1 \cup C_2)) = \mathsf{access}_{n,\mathcal{M}_d}(\mathsf{addr}(C_1)) \cup \mathsf{access}_{n,\mathcal{M}_d}(\mathsf{addr}(C_2))$.

- By the claimed additivity of addr, it suffices to show that:
  $\forall n.\ \mathsf{access}_{n,\mathcal{M}_d}(\mathsf{addr}(C_1) \cup \mathsf{addr}(C_2)) = \mathsf{access}_{n,\mathcal{M}_d}(\mathsf{addr}(C_1)) \cup \mathsf{access}_{n,\mathcal{M}_d}(\mathsf{addr}(C_2))$.

- The latter directly follows by Lemma 17.

□

Lemma 19 (Additivity of $\mathsf{reachable\_addresses}$ in the first argument using addr)**.**

$$\forall C, C_1, C_2, \mathcal{M}_d.$$
$$\mathsf{addr}(C) = \mathsf{addr}(C_1) \cup \mathsf{addr}(C_2)$$
$$\implies$$
$$\mathsf{reachable\_addresses}(C, \mathcal{M}_d) = \mathsf{reachable\_addresses}(C_1, \mathcal{M}_d) \cup \mathsf{reachable\_addresses}(C_2, \mathcal{M}_d)$$

Proof. Similar to the proof of Lemma 18. □

Lemma 20 (Invariance to capability's location so long as it is reachable)**.**

$$\forall C, \mathcal{M}_d, a, c.$$
$$\mathcal{M}_d(a) \neq (\delta, \_, \_, \_) \wedge c = (\delta, \_, \_, \_) \wedge$$
$$a \in \mathsf{reachable\_addresses}(C, \mathcal{M}_d)$$
$$\implies \mathsf{reachable\_addresses}(C \cup \{c\}, \mathcal{M}_d) = \mathsf{reachable\_addresses}(C, \mathcal{M}_d[a \mapsto c])$$

Proof.

- We fix arbitrary a, c, C, and $\mathcal{M}_d$, and assume the antecedent:
  $\mathcal{M}_d(a) \neq (\delta, \_, \_, \_) \;\wedge\; c = (\delta, \_, \_, \_) \;\wedge\; a \in \mathsf{reachable\_addresses}(C, \mathcal{M}_d)$.
- We let $A = \mathsf{addr}(C)$ where $\mathsf{addr}(C) \stackrel{\mathsf{def}}{=} \bigcup\limits_{c \in C} [c.s, c.e)$.
- From the antecedent, and by Definition 22 and the definition of $\cup$, we thus have:
  (*) $\exists k_a.\ a \in \mathsf{access}_{k_a, \mathcal{M}_d} A$
- By Lemma 18, our goal can be rewritten as:
  $\mathsf{reachable\_addresses}(C, \mathcal{M}_d) \cup \mathsf{reachable\_addresses}(\{c\}, \mathcal{M}_d)$
  $= \mathsf{reachable\_addresses}(C, \mathcal{M}_d[a \mapsto c])$.
- By Definition 22, it is equivalent to show that:
  $\forall b.\ b \in \Big(\bigcup\limits_{n \in [0, |\mathcal{M}_d|]} \mathsf{access}_{n, \mathcal{M}_d} A \cup \bigcup\limits_{n \in [0, |\mathcal{M}_d|]} \mathsf{access}_{n, \mathcal{M}_d}(\mathsf{addr}(\{c\}))\ \Big)$
  $\iff b \in \bigcup\limits_{n \in [0, |\mathcal{M}_d[a \mapsto c]|]} \mathsf{access}_{n, \mathcal{M}_d[a \mapsto c]} A$

  We have two proof obligations:

  – **Goal "$\implies$":**
    Here, we assume for an arbitrary b that:
    $b \in \Big(\bigcup\limits_{n \in [0, |\mathcal{M}_d|]} \mathsf{access}_{n, \mathcal{M}_d} A \cup \bigcup\limits_{n \in [0, |\mathcal{M}_d|]} \mathsf{access}_{n, \mathcal{M}_d}(\mathsf{addr}(\{c\}))\ \Big)$.
    Our goal is to show that:
    $b \in \bigcup\limits_{n \in [0, |\mathcal{M}_d[a \mapsto c]|]} \mathsf{access}_{n, \mathcal{M}_d[a \mapsto c]} A$
    We consider the two possible cases from our assumption:

    1. **Case** $b \in \bigcup\limits_{n \in [0, |\mathcal{M}_d|]} \mathsf{access}_{n, \mathcal{M}_d} A$**:**
       By the definition of $\cup$, we have:
       (**) $\exists k_b.\ k_b \in [0, |\mathcal{M}_d|] \wedge b \in \mathsf{access}_{k_b, \mathcal{M}_d} A$.
       Under our lemma's antecedents, we show the following:
       $$\boxed{\forall k, b'.\ b' \in \mathsf{access}_{k, \mathcal{M}_d} A \implies b' \in \mathsf{access}_{k, \mathcal{M}_d[a \mapsto c]} A}$$

* **Case $b' = a$:**
  In this case, our goal already follows by Lemma 23 which states that an update to a location (in this case, a) does not affect its own accessibility.
* **Case $b' \neq a$:**
  Here, we prove our statement by induction on k:

  (a) **Base case $k = 0$:**
  We assume $b' \in \mathsf{access}_{0,\mathcal{M}_d} A$, i.e., by Definition 21, that $b' \in A$.
  Our goal is to show that $b' \in \mathsf{access}_{0,\mathcal{M}_d[a\mapsto c]} A$, which by Definition 21 is $b' \in A$.

  (b) **Inductive case $k > 0$:**
  By the induction hypothesis, we have:
  $\forall b'.\ b' \in \mathsf{access}_{k-1,\mathcal{M}_d} A \implies b' \in \mathsf{access}_{k-1,\mathcal{M}_d[a\mapsto c]} A$.
  We assume $b' \in \mathsf{access}_{k,\mathcal{M}_d} A$, and our goal is to show that $b' \in \mathsf{access}_{k,\mathcal{M}_d[a\mapsto c]} A$.
  By unfolding Definition 21, we distinguish two cases:

  i. **Case $b' \in \mathsf{access}_{k-1,\mathcal{M}_d} A$:**
  In this case, by instantiating the induction hypothesis, we conclude:
  $b' \in \mathsf{access}_{k-1,\mathcal{M}_d[a\mapsto c]} A$.
  By Definition 21, and expansiveness (Lemma 7) of $\mathsf{access}_{\mathcal{M}_d[a\mapsto c]}$, we obtain our goal: $b' \in \mathsf{access}_{k,\mathcal{M}_d[a\mapsto c]} A$.

  ii. **Case $b' \in \bigcup_{a' \in \mathsf{access}_{k-1,\mathcal{M}_d} A,\ \mathcal{M}_d(a')=(\delta,s,e,\_)} [s,e)$:**
  By the definition of $\cup$, we have:
  $\exists a'.\ a' \in \mathsf{access}_{k-1,\mathcal{M}_d} A \wedge \mathcal{M}_d(a') = (\delta, s, e, \_) \wedge b' \in [s,e)$.
  By the induction hypothesis, we have: $a' \in \mathsf{access}_{k-1,\mathcal{M}_d[a\mapsto c]} A$.
  So, we distinguish two cases:

  A. **Case $a' \neq a$:**
  Here, by the definition/stability of the function update operator, we have that:
  $\mathcal{M}_d[a \mapsto c](a') = \mathcal{M}_d(a') = (\delta, s, e, \_)$.
  So our goal is satisfied by seeing that we have the judgment:
  $\exists a'.\ a' \in \mathsf{access}_{k-1,\mathcal{M}_d[a\mapsto c]} A \wedge \mathcal{M}_d[a \mapsto c](a') = (\delta, s, e, \_) \wedge b' \in [s,e)$.
  So, by folding back the definition of $\cup$ and Definition 21 of $\mathsf{access}_{k,\mathcal{M}_d[a\mapsto c]} A$, we see that indeed $b' \in \mathsf{access}_{k,\mathcal{M}_d[a\mapsto c]} A$.

  B. **Case $a' = a$:**
  Here, conjunct $\mathcal{M}_d(a') = (\delta, s, e, \_)$ contradicts our antecedent $\mathcal{M}_d(a) \neq (\delta, \_, \_, \_)$.
  So any goal is provable.

Having shown our boxed statement, we now instantiate it with b and k_b from (**) to obtain:
$b \in \mathsf{access}_{k_b,\mathcal{M}_d[a\mapsto c]} A$.
Thus, by $k_b \in [0, |\mathcal{M}_d|]$ of (**), and the definition of $\cup$, we have our goal:
$b \in \bigcup_{n\in[0,|\mathcal{M}_d[a\mapsto c]|]} \mathsf{access}_{n,\mathcal{M}_d[a\mapsto c]} A$ by noticing that $|\mathcal{M}_d[a \mapsto c]| \geq |\mathcal{M}_d|$.

2. **Case $b \in \bigcup_{n\in[0,|\mathcal{M}_d|]} \mathsf{access}_{n,\mathcal{M}_d}(\mathsf{addr}(\{c\}))$:**
By the definition of $\cup$, we have:
(**2) $\exists k_b.\ k_b \in [0, |\mathcal{M}_d|] \wedge b \in \mathsf{access}_{k_b,\mathcal{M}_d} \mathsf{addr}(\{c\})$.
From (*), we know k_a.
Under our lemma's antecedents, we show the following:

$$\boxed{\forall k'_a, k'_b, b'.\ b' \in \mathsf{access}_{k'_b,\mathcal{M}_d} \mathsf{addr}(\{c\}) \wedge a \in \mathsf{access}_{k'_a,\mathcal{M}_d} A \implies b' \in \mathsf{access}_{k'_a+k'_b+1,\mathcal{M}_d[a\mapsto c]} A}$$

We consider two cases:

* **Case $b' = a$:**
  In this case, by Lemma 23, we know $b' \in \mathsf{access}_{k'_a,\mathcal{M}_d[a\mapsto c]} A$. Thus by Lemma 8, we know $b' \in \mathsf{access}_{k'_a+k'_b+1,\mathcal{M}_d[a\mapsto c]} A$.

* **Case $b' \neq a$:**
  In this case, we prove it by induction on k'_a.

(a) **Base case $k'_a = 0$:**
In this case, we know by the antecedent and unfolding Definition 21 that $a \in A$.
We prove our goal by induction on k'_b.

i. **Base case ($k'_b = 0$):**
In this case, we know by the antecedent and Definition 21 that $b' \in \mathsf{addr}(\{c\})$, and our goal is to show that:
$b' \in A \cup \bigcup_{a^* \in A,\ \mathcal{M}_d[a \mapsto c](a^*) = (\delta, s, e, \_)} [s, e)$.
We show the goal by choosing $a^* := a$. We notice that a satisfies $a \in A$ by our former base case.
And given our lemma's antecedent $c = (\delta, s, e, \_)$, all that remains to be shown is that $b' \in [s, e)$.
But that follows directly from the definition of $\mathsf{addr}(\{c\})$ instantiated with the singleton set $\{c\}$.
So, our goal is satisfied by the definition of $\cup$ by satisfying membership in the right-hand-side set.

ii. **Inductive case ($k'_b > 0$):**
By the induction hypothesis, we have:
$\forall b'.\ b' \in \mathsf{access}_{k'_b - 1, \mathcal{M}_d} \mathsf{addr}(\{c\}) \wedge a \in A \implies b' \in \mathsf{access}_{k'_b, \mathcal{M}_d[a \mapsto c]} A$
By assumption, we have $a \in A$ and $b' \in \mathsf{access}_{k'_b, \mathcal{M}_d} \mathsf{addr}(\{c\})$, and our goal is to show that $b' \in \mathsf{access}_{k'_b + 1, \mathcal{M}_d[a \mapsto c]} A$.
From the assumption $b' \in \mathsf{access}_{k'_b, \mathcal{M}_d} \mathsf{addr}(\{c\})$, we know by Definition 21 that there are two possible cases:

- **Case $b' \in \mathsf{access}_{k'_b - 1, \mathcal{M}_d} \mathsf{addr}(\{c\})$:**
  In this case, we instantiate the induction hypothesis and obtain:
  $b' \in \mathsf{access}_{k'_b, \mathcal{M}_d[a \mapsto c]} A$.
  Thus, our goal is satisifed by expansiveness (Lemma 8).
- **Case $b' \in \bigcup_{a^* \in \mathsf{access}_{k'_b - 1, \mathcal{M}_d} \mathsf{addr}(\{c\}),\ \mathcal{M}_d(a^*) = (\delta, s, e, \_)} [s, e)$:**
  In this case we obtain a^* by the definition of $\bigcup$, and we distinguish the following two cases:
  - **Case $a^* = a$:**
  This case is impossible because the $\bigcup$-condition $\mathcal{M}_d(a^*) = (\delta, \_, \_, \_)$ contradicts our lemma's assumed antecedent.
  - **Case $a^* \neq a$:**
  In this case, we conclude from $a^* \in \mathsf{access}_{k'_b - 1, \mathcal{M}_d} \mathsf{addr}(\{c\})$ and the induction hypothesis that $a^* \in \mathsf{access}_{k'_b, \mathcal{M}_d[a \mapsto c]} A$.
  Thus, given that $b' \in [s, e)$ where $\mathcal{M}_d[a \mapsto c](a^*) = (\delta, s, e, \_)$, we conclude by folding Definition 21 that $b' \in \mathsf{access}_{k'_b + 1, \mathcal{M}_d[a \mapsto c]} A$ by membership in the right operand of $\cup$ in Definition 21.
  This last conclusion is our goal.

(b) **Inductive case $k'_a > 0$:**
By the induction hypothesis, we have (IHka):
$\forall k'_b, b'.\ b' \in \mathsf{access}_{k'_b, \mathcal{M}_d} \mathsf{addr}(\{c\}) \wedge a \in \mathsf{access}_{k'_a - 1, \mathcal{M}_d} A \implies b' \in \mathsf{access}_{k'_a + k'_b, \mathcal{M}_d[a \mapsto c]} A$
And, our goal is to show that:
$\forall k'_b, b'.\ b' \in \mathsf{access}_{k'_b, \mathcal{M}_d} \mathsf{addr}(\{c\}) \wedge a \in \mathsf{access}_{k'_a, \mathcal{M}_d} A \implies b' \in \mathsf{access}_{k'_a + k'_b + 1, \mathcal{M}_d[a \mapsto c]} A$
Again, we prove our goal by induction on k'_b.

i. **Base case ($k'_b = 0$):**
In this case, we know $b' \in \mathsf{addr}(\{c\})$, and $a \in \mathsf{access}_{k'_a, \mathcal{M}_d} A$, and our goal is to show that $b' \in \mathsf{access}_{k'_a + 1, \mathcal{M}_d[a \mapsto c]} A$.

By Lemma 15, we know that $a \in \mathsf{access}_{k'_a,\mathcal{M}_d[a\mapsto c]} A$.
For our goal, it suffices to show that:
$b' \in \bigcup_{a^* \in \mathsf{access}_{k'_a,\mathcal{M}_d[a\mapsto c]} A, \mathcal{M}_d[a\mapsto c](a^*)=(\delta,s,e,\_)} [s,e)$.
We pick $a^* := a$, so we know from just above that $a \in \mathsf{access}_{k'_a,\mathcal{M}_d[a\mapsto c]} A$ holds, and then it suffices to show that $b' \in [s,e)$ where $c = (\delta, s, e, \_)$.
The latter follows by our assumption $b' \in \mathsf{addr}(\{c\})$ by unfolding our definition of addr given in the beginning.

ii. **Inductive case ($k'_b > 0$):**
In this case, we know by the I.H. that (IHkb):
$\forall b'.\ b' \in \mathsf{access}_{k'_b-1,\mathcal{M}_d} \mathsf{addr}(\{c\}) \wedge a \in \mathsf{access}_{k'_a,\mathcal{M}_d} A \implies b' \in \mathsf{access}_{k'_a+k'_b,\mathcal{M}_d[a\mapsto c]} A$
We assume the antecedents of our goal for arbitrary b':
$b' \in \mathsf{access}_{k'_b,\mathcal{M}_d} \mathsf{addr}(\{c\}) \wedge a \in \mathsf{access}_{k'_a,\mathcal{M}_d} A$.
By Definition 21, we distinguish the following three cases:

- **Case $a \in \mathsf{access}_{k'_a-1,\mathcal{M}_d} A$:**
In this case, we obtain by (IHka) that $b' \in \mathsf{access}_{k'_a+k'_b,\mathcal{M}_d[a\mapsto c]} A$.
By Lemma 8, we have our goal.
- **Case $b' \in \mathsf{access}_{k'_b-1,\mathcal{M}_d} \mathsf{addr}(\{c\})$:**
In this case, we obtain by (IHkb) that $b' \in \mathsf{access}_{k'_a+k'_b,\mathcal{M}_d[a\mapsto c]} A$.
By Lemma 8, we have our goal.
- **Case $a \notin \mathsf{access}_{k'_a-1,\mathcal{M}_d} A \wedge b' \notin \mathsf{access}_{k'_b-1,\mathcal{M}_d} \mathsf{addr}(\{c\})$:**
Equivalently (from the unfolding of Definition 21 in both of our antecedents), we know in this case that:
$a \in \bigcup_{a^* \in \mathsf{access}_{k'_a-1,\mathcal{M}_d} A, \mathcal{M}_d(a^*)=(\delta,s_a,e_a,\_)} [s_a, e_a) \wedge b' \in \bigcup_{b^* \in \mathsf{access}_{k'_b-1,\mathcal{M}_d} \mathsf{addr}(\{c\}), \mathcal{M}_d(b^*)=(\delta,s_b,e_b,\_)} [s_b, e_b)$
From the right conjunct, we obtain b^* satisfying
$b^* \in \mathsf{access}_{k'_b-1,\mathcal{M}_d} \mathsf{addr}(\{c\}) \wedge \mathcal{M}_d(b^*) = (\delta, s_b, e_b, \_) \wedge b' \in [s_b, e_b)$. So, by (IHkb), we know that $b^* \in \mathsf{access}_{k'_a+k'_b,\mathcal{M}_d[a\mapsto c]} A$.
By Definitions 20 and 21 and the definition of $\cup$, it suffices for our goal ($b' \in \mathsf{access}_{k'_a+k'_b+1,\mathcal{M}_d[a\mapsto c]} A$) to show that
$b' \in \bigcup_{b^* \in \mathsf{access}_{k'_a+k'_b,\mathcal{M}_d[a\mapsto c]} A,\ \mathcal{M}_d[a\mapsto c](b^*)=(\delta,s_b,e_b,\_)} [s_b, e_b)$.
We satisfy the latter by picking the b^* we obtained above noticing that it satisfies $b^* \in \mathsf{access}_{k'_a+k'_b,\mathcal{M}_d[a\mapsto c]} A$ by Lemma 15, and that it satisfies $\mathcal{M}_d[a \mapsto c](b^*) = (\delta, s_b, e_b, \_)$ because $b^* \neq a$ must hold (otherwise, we contradict our antecedent $\mathcal{M}_d(a) \neq (\delta, \_, \_, \_)$).
This concludes our case.

This concludes the proof of our boxed statement; we instantiate it by (**2) and (*) to obtain (**2*):
$b \in \mathsf{access}_{k_a+k_b+1,\mathcal{M}_d[a\mapsto c]} A$.
Recall that our goal is to show that $\exists n.\ n \in [0, |\mathcal{M}_d[a \mapsto c]|] \wedge b \in \mathsf{access}_{n,\mathcal{M}_d[a\mapsto c]} A$.
We distinguish two cases for $k_a + k_b + 1$:

* **Case $k_a + k_b + 1 \leq |\mathcal{M}_d[a \mapsto c]|$:**
In this case, our goal follows directly from (**2*).
* **Case $k_a + k_b + 1 > |\mathcal{M}_d[a \mapsto c]|$:**
In this case, we know by Lemma 13 that:
$\mathsf{access}_{k_a+k_b+1,\mathcal{M}_d[a\mapsto c]} A = \mathsf{access}_{|\mathcal{M}_d[a\mapsto c]|,\mathcal{M}_d[a\mapsto c]} A$.
So, we pick $n := |\mathcal{M}_d[a \mapsto c]|$ satisfying our goal.

This concludes **Goal "$\implies$"**.

– **Goal "$\impliedby$":**

Here, we assume for an arbitrary b that:
$b \in \bigcup_{n\in[0,|\mathcal{M}_d[a\mapsto c]|]} \mathsf{access}_{n,\mathcal{M}_d[a\mapsto c]} A$
Our goal is to show that:
$b \in (\bigcup_{n\in[0,|\mathcal{M}_d|]} \mathsf{access}_{n,\mathcal{M}_d} A \cup \bigcup_{n\in[0,|\mathcal{M}_d|]} \mathsf{access}_{n,\mathcal{M}_d} (\mathsf{addr}(\{c\})))$.
By assumption, we know (#):
$\exists n.\ n \in [0, |\mathcal{M}_d[a \mapsto c]|] \wedge b \in \mathsf{access}_{n,\mathcal{M}_d[a\mapsto c]} A$.
We prove the general statement:

$$\boxed{\forall n, b'.\ b' \in \mathsf{access}_{n,\mathcal{M}_d[a\mapsto c]} A \implies b' \in \mathsf{access}_{n,\mathcal{M}_d} A \vee b' \in \mathsf{access}_{n,\mathcal{M}_d}(\mathsf{addr}(\{c\}))}$$

We prove our goal by induction on n.

* **Base case ($n = 0$):**
  In this case, we know $b' \in \mathsf{access}_{0,\mathcal{M}_d[a\mapsto c]} A = A = \mathsf{access}_{0,\mathcal{M}_d} A$.
  So, our goal is satisfied by satisfying the left disjunct.
* **Inductive case ($n > 0$):**
  The induction hypothesis gives us:
  $\forall b'.\ b' \in \mathsf{access}_{n-1,\mathcal{M}_d[a\mapsto c]} A \implies b' \in \mathsf{access}_{n-1,\mathcal{M}_d} A \vee b' \in \mathsf{access}_{n-1,\mathcal{M}_d}(\mathsf{addr}(\{c\}))$
  By assumption and Definitions 20 and 21, we distinguish two cases:

  · **Case $b' \in \mathsf{access}_{n-1,\mathcal{M}_d[a\mapsto c]} A$:**
  In this case, we have by the induction hypothesis that:
  $b' \in \mathsf{access}_{n-1,\mathcal{M}_d} A \vee b' \in \mathsf{access}_{n-1,\mathcal{M}_d}(\mathsf{addr}(\{c\}))$.
  So, in either case (left disjunct or right disjunct holds), we have our goal by unfolding Definition 21 in our goal and applying Lemma 7.

  · **Case $\exists b''.\ b'' \in \mathsf{access}_{n-1,\mathcal{M}_d[a\mapsto c]} A \wedge \mathcal{M}_d[a \mapsto c](b'') = (\delta, s, e, \_) \wedge b' \in [s, e)$:**
  By the induction hypothesis, we know:
  $b'' \in \mathsf{access}_{n-1,\mathcal{M}_d} A \vee b'' \in \mathsf{access}_{n-1,\mathcal{M}_d}(\mathsf{addr}(\{c\}))$.
  We distinguish two cases:

  - Case $b'' \neq a$:
  In this case, we know that $\mathcal{M}_d[a \mapsto c](b'') = \mathcal{M}_d(b'') = (\delta, s, e, \_)$.
  So, by Definition 21, we can conclude:
  $b' \in \mathsf{access}_{n,\mathcal{M}_d} A$ in case $b'' \in \mathsf{access}_{n-1,\mathcal{M}_d} A$, and
  $b' \in \mathsf{access}_{n,\mathcal{M}_d}(\mathsf{addr}(\{c\}))$ in case $b'' \in \mathsf{access}_{n-1,\mathcal{M}_d}(\mathsf{addr}(\{c\}))$.
  - Case $b'' = a$:
  In this case, we know that $c = \mathcal{M}_d[a \mapsto c](b'') = (\delta, s, e, \_) \wedge b' \in [s, e)$.
  So, in particular, we know $b' \in \mathsf{addr}(\{c\})$.
  So, by Definition 21, we know $b' \in \mathsf{access}_{0,\mathcal{M}_d}(\mathsf{addr}(\{c\}))$.
  So, by $n > 0$, and by expansiveness (Lemmas 7 and 8), we conclude:
  $b' \in \mathsf{access}_{n,\mathcal{M}_d}(\mathsf{addr}(\{c\}))$, which satisfies the right disjunct of our goal.

This concludes the proof of our boxed statement.
Instantiating it with (#) gives us by Lemma 13 an n satisfying our goal.
This concludes **Goal**“ $\Longleftarrow$ ”, which concludes the proof of Lemma 20.

□

Lemma 21 (Invariance to unreachable memory updates)**.**

$$\forall C, \mathcal{M}_d, a, v.\ a \notin \mathsf{reachable\_addresses}(C, \mathcal{M}_d) \implies$$
$$\mathsf{reachable\_addresses}(C, \mathcal{M}_d) = \mathsf{reachable\_addresses}(C, \mathcal{M}_d[a \mapsto v])$$

Proof.

- We fix arbitrary $C, \mathcal{M}_d, a$, and v. We assume the antecedent.
- By unfolding Definition 22, and the definition of $\cup$, our antecedent can be re-written as (‡): $\forall n \in [0, |\mathcal{M}_d|].\ a \notin \mathsf{access}_{n,\mathcal{M}_d} \mathsf{addr}(C)$,

  where $\mathsf{addr}(C) \stackrel{\mathsf{def}}{=} \bigcup_{c \in C} [c.s, c.e)$.
- Thus, by Lemma 22, we conclude that (‡‡): $\forall n \in [0, |\mathcal{M}_d|].\ \mathsf{access}_{n,\mathcal{M}_d} \mathsf{addr}(C) = \mathsf{access}_{n,\mathcal{M}_d[a \mapsto v]} \mathsf{addr}(C)$.
- Thus, by identities of $\cup$, we have that (*): $\bigcup_{n \in [0, |\mathcal{M}_d|]} \mathsf{access}_{n,\mathcal{M}_d} \mathsf{addr}(C) = \bigcup_{n \in [0, |\mathcal{M}_d|]} \mathsf{access}_{n,\mathcal{M}_d[a \mapsto v]} \mathsf{addr}(C)$
- (Intuition) By looking at the right-hand side, we notice that the set union could be missing one extra step for the expression to satisfy $\mathsf{reachable\_addresses}(C, \mathcal{M}_d[a \mapsto v])$. The intuition is $|\mathcal{M}_d[a \mapsto v]| \in [|\mathcal{M}_d|, |\mathcal{M}_d| + 1]$.

  In particular, we distinguish the two possible cases:

  - **Case $a \in \mathsf{dom}(\mathcal{M}_d)$:**
    In this case, $|\mathcal{M}_d| = |\mathcal{M}_d[a \mapsto v]|$.
    So statement (*) directly satisfies our goal by folding using Definition 22 and the definition of addr.
  - **Case $a \notin \mathsf{dom}(\mathcal{M}_d)$:**
    In this case, $|\mathcal{M}_d[a \mapsto v]| = |\mathcal{M}_d| + 1$. So, we assume for the sake of contradiction that: (\$) $\mathsf{access}_{|\mathcal{M}_d|+1,\mathcal{M}_d[a \mapsto v]}(\mathsf{addr}(C)) \supsetneq \mathsf{access}_{|\mathcal{M}_d|,\mathcal{M}_d[a \mapsto v]}(\mathsf{addr}(C))$.
    (Notice that by Lemma 8, necessarily $\mathsf{access}_{|\mathcal{M}_d|+1,\mathcal{M}_d[a \mapsto v]}(\mathsf{addr}(C)) \supseteq \mathsf{access}_{|\mathcal{M}_d|,\mathcal{M}_d[a \mapsto v]}(\mathsf{addr}(C))$).
    * In this case, we know by unfolding Definitions 20 and 21 that (‡ ‡ ‡): $\exists a', a''.\ a' \notin \mathsf{access}_{|\mathcal{M}_d|,\mathcal{M}_d[a \mapsto v]}(\mathsf{addr}(C)) \wedge$ $a'' \in \mathsf{access}_{|\mathcal{M}_d|,\mathcal{M}_d[a \mapsto v]}(\mathsf{addr}(C)) \wedge \mathcal{M}_d[a \mapsto v](a'') = (\delta, s, e, \_) \wedge a' \in [s, e)$.
    * We distinguish two cases for a'':
      - **Case $a'' = a$:**
        In this case, we know by (‡ ‡ ‡) that $a \in \mathsf{access}_{|\mathcal{M}_d|,\mathcal{M}_d[a \mapsto v]}(\mathsf{addr}(C))$.
        But by (‡‡), this means that $a \in \mathsf{access}_{|\mathcal{M}_d|,\mathcal{M}_d}(\mathsf{addr}(C))$.
        But this contradicts (‡). So, any goal is provable.
      - **Case $a'' \neq a$:**
        Again, we know by (‡ ‡ ‡) that $a'' \in \mathsf{access}_{|\mathcal{M}_d|,\mathcal{M}_d[a \mapsto v]}(\mathsf{addr}(C))$.
        And again by (‡‡), this means that $a'' \in \mathsf{access}_{|\mathcal{M}_d|,\mathcal{M}_d}(\mathsf{addr}(C))$.
        And by conjunct $\mathcal{M}_d[a \mapsto v](a'') = \_$ of (‡ ‡ ‡) together with our case condition $a'' \neq a$, we know that $a'' \in \mathsf{dom}(\mathcal{M}_d)$.
        Thus, we have by (‡ ‡ ‡) that the following expression holds:
        $a'' \in \mathsf{access}_{|\mathcal{M}_d|,\mathcal{M}_d}(\mathsf{addr}(C)) \wedge \mathcal{M}_d(a'') = (\delta, s, e, \_) \wedge a' \in [s, e)$.
        This gives us by folding Definition 21 that:
        $a' \in \mathsf{access}_{|\mathcal{M}_d|+1,\mathcal{M}_d}(\mathsf{addr}(C))$.
        But we know from (‡ ‡ ‡) that $a' \notin \mathsf{access}_{|\mathcal{M}_d|,\mathcal{M}_d[a \mapsto v]}(\mathsf{addr}(C))$, which by (‡‡) gives us $a' \notin \mathsf{access}_{|\mathcal{M}_d|,\mathcal{M}_d}(\mathsf{addr}(C))$.
        This means that $a' \in \mathsf{access}_{|\mathcal{M}_d|+1,\mathcal{M}_d}(\mathsf{addr}(C)) \setminus \mathsf{access}_{|\mathcal{M}_d|,\mathcal{M}_d}(\mathsf{addr}(C))$, i.e., $\mathsf{access}_{|\mathcal{M}_d|+1,\mathcal{M}_d}(\mathsf{addr}(C)) \supsetneq \mathsf{access}_{|\mathcal{M}_d|,\mathcal{M}_d}(\mathsf{addr}(C))$

By Lemma 12, we, hence, conclude:
($$) $|\{a^* \mid a^* \in \mathsf{access}_{|\mathcal{M}_d|,\mathcal{M}_d}(\mathsf{addr}(C)) \wedge a^* \in \mathsf{dom}(\mathcal{M}_d)\}| > |\mathcal{M}_d|$.
But, $\{a^* \mid a^* \in \mathsf{access}_{|\mathcal{M}_d|,\mathcal{M}_d}(\mathsf{addr}(C)) \wedge a^* \in \mathsf{dom}(\mathcal{M}_d)\} \subseteq \mathsf{dom}(\mathcal{M}_d)$.
Thus, $|\{a^* \mid a^* \in \mathsf{access}_{|\mathcal{M}_d|,\mathcal{M}_d}(\mathsf{addr}(C)) \wedge a^* \in \mathsf{dom}(\mathcal{M}_d)\}| \leq |\mathcal{M}_d|$.
This contradicts ($$). So, any goal is provable.

$\square$

Lemma 22 (Updating k-inaccessible locations does not affect the k-accessibility set)**.**

$$\forall a, k, \mathcal{M}_d, A, v.\ a \notin \mathsf{access}_{k,\mathcal{M}_d} A \implies \mathsf{access}_{k,\mathcal{M}_d} A = \mathsf{access}_{k,\mathcal{M}_d[a\mapsto v]} A$$

Proof. We prove it by induction on k.

- **Base case ($k = 0$):**

  Fix arbitrary a, A, v, and $\mathcal{M}_d$.

  By Definition 21, we have that $\mathsf{access}_{0,\_} A = A = \mathsf{access}_{0,\mathcal{M}_d} A = \mathsf{access}_{0,\mathcal{M}_d[a\mapsto v]} A$.

- **Inductive case ($k > 0$):**

  The induction hypothesis gives us (*):
  $\forall a, \mathcal{M}_d, A, v.\ a \notin \mathsf{access}_{k-1,\mathcal{M}_d} A \implies \mathsf{access}_{k-1,\mathcal{M}_d} A = \mathsf{access}_{k-1,\mathcal{M}_d[a\mapsto v]} A$

  We fix arbitrary $a, \mathcal{M}_d, A$, and v, and we assume $a \notin \mathsf{access}_{k,\mathcal{M}_d} A$.

  Now, by Definitions 20 and 21, we have:
  $\mathsf{access}_{k,\mathcal{M}_d} A = \mathsf{access}_{k-1,\mathcal{M}_d} A \cup \bigcup_{a' \in \mathsf{access}_{k-1,\mathcal{M}_d} A,\ \mathcal{M}_d(a') = (\delta, s, e, \_)} [s, e)$

  Thus, by our assumption together with the definition of $\cup$, we conclude:
  (**1) $a \notin \mathsf{access}_{k-1,\mathcal{M}_d} A$, and
  (**2) $a \notin \bigcup_{a' \in \mathsf{access}_{k-1,\mathcal{M}_d} A,\ \mathcal{M}_d(a') = (\delta, s, e, \_)} [s, e)$

  By (**1) and (*), we have (***):
  $\mathsf{access}_{k-1,\mathcal{M}_d} A = \mathsf{access}_{k-1,\mathcal{M}_d[a\mapsto v]} A$.

  Now, in order to show our goal ($\mathsf{access}_{k,\mathcal{M}_d} A = \mathsf{access}_{k,\mathcal{M}_d[a\mapsto v]} A$), it suffices by Definitions 20 and 21 to show that both:
  (g1) $\mathsf{access}_{k-1,\mathcal{M}_d} A = \mathsf{access}_{k-1,\mathcal{M}_d[a\mapsto v]} A$, and
  (g2) $\bigcup_{a' \in \mathsf{access}_{k-1,\mathcal{M}_d} A,\ \mathcal{M}_d(a') = (\delta, s, e, \_)} [s, e) = \bigcup_{a' \in \mathsf{access}_{k-1,\mathcal{M}_d[a\mapsto v]} A,\ \mathcal{M}_d[a\mapsto v](a') = (\delta, s, e, \_)} [s, e)$.

  We already have (g1) by (***).

  By substitution using (***), our goal (g2) becomes:
  (g2) $\bigcup_{a' \in \mathsf{access}_{k-1,\mathcal{M}_d[a\mapsto v]} A,\ \mathcal{M}_d(a') = (\delta, s, e, \_)} [s, e) = \bigcup_{a' \in \mathsf{access}_{k-1,\mathcal{M}_d[a\mapsto v]} A,\ \mathcal{M}_d[a\mapsto v](a') = (\delta, s, e, \_)} [s, e)$.

  So it suffices to show that $\forall a' \in \mathsf{access}_{k-1,\mathcal{M}_d[a\mapsto v]} A.\ \mathcal{M}_d(a') = \mathcal{M}_d[a \mapsto v](a')$.

  - **Case $a' \neq a$:**

    By the definition of function update, we have our goal:
    $\mathcal{M}_d(a') = \mathcal{M}_d[a \mapsto v](a')$

  - **Case $a' = a$:**

    Impossible because by substituting using (***) in (**1),
    we get a contradiction to $a' \in \mathsf{access}_{k-1,\mathcal{M}_d[a\mapsto v]} A$.

  This concludes the inductive case, which concludes the proof of Lemma 22.

$\square$

Lemma 23 (Updating a location does not affect its own k-accessibility)**.**

$$\forall a, A, k_a, \mathcal{M}_d, v.\ a \in \mathsf{access}_{k_a,\mathcal{M}_d} A \implies a \in \mathsf{access}_{k_a,\mathcal{M}_d[a\mapsto v]} A$$

Proof. We fix arbitrary $a, A, k_a, \mathcal{M}_d$, and v. We assume the antecedent $a \in \mathsf{access}_{k_a,\mathcal{M}_d} A$.

Our goal is to show that $a \in \mathsf{access}_{k_a,\mathcal{M}_d[a\mapsto v]} A$.

Assume for the sake of contradiction the contrary of our goal: $(a \notin \mathsf{access}_{k_a,\mathcal{M}_d[a\mapsto v]} A)$. Then:

- By Lemma 22, we conclude that:
  $\mathsf{access}_{k_a,\mathcal{M}_d[a\mapsto v]} A = \mathsf{access}_{k_a,\mathcal{M}_d[a\mapsto v][a\mapsto \mathcal{M}_d(a)]} A$, which simplifies to:
  $\mathsf{access}_{k_a,\mathcal{M}_d[a\mapsto v]} A = \mathsf{access}_{k_a,\mathcal{M}_d} A$.
- Substituting using this equality into our latest assumption, we get:
  $a \notin \mathsf{access}_{k_a,\mathcal{M}_d} A$.
- This contradicts our antecedent, so our latest assumption must be false.

This concludes the proof of Lemma 23.

□

Lemma 24 (Updating a location does not affect its own reachability)**.**

$$\forall C, a, v, \mathcal{M}_d.\ a \in \mathsf{reachable\_addresses}(C, \mathcal{M}_d) \implies a \in \mathsf{reachable\_addresses}(C, \mathcal{M}_d[a \mapsto v])$$

Proof.

- We fix arbitrary $C, a, v, \mathcal{M}_d$, and assume the antecedent.
- By assumption and unfolding Definition 22, we have $a \in \bigcup_{k\in[0,|\mathcal{M}_d|]} \mathsf{access}_{k,\mathcal{M}_d} \mathsf{addr}(C)$,

  where $\mathsf{addr}(C) \stackrel{\mathsf{def}}{=} \bigcup_{c\in C} [c.s, c.e)$.
- Thus, by the definition of $\cup$, we have (*): $\exists k_a \in [0, |\mathcal{M}_d|].\ a \in \mathsf{access}_{k_a,\mathcal{M}_d} \mathsf{addr}(C)$.
- And then by Lemma 23, we conclude that (**): $a \in \mathsf{access}_{k_a,\mathcal{M}_d[a\mapsto v]} \mathsf{addr}(C)$.
- And by the definition of the function update operator, we notice that
  $k_a \in [0, |\mathcal{M}_d|] \implies k_a \in [0, |\mathcal{M}_d[a \mapsto v]|]$ which gives us $k_a \in [0, |\mathcal{M}_d[a \mapsto v]|]$ by (*).
- Thus, by definition of $\cup$, we have from (**) that: $a \in \bigcup_{k\in[0,|\mathcal{M}_d[a\mapsto v]|]} \mathsf{access}_{k,\mathcal{M}_d[a\mapsto v]} \mathsf{addr}(C)$.
- Thus, by folding using Definition 22, we get our goal: $a \in \mathsf{reachable\_addresses}(C, \mathcal{M}_d[a \mapsto v])$.

□

Lemma 25 (Completeness of $\mathsf{reachable\_addresses}$)**.**

$$\begin{aligned}
&\forall \mathcal{E}, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}.\\
&\mathsf{ddc} = (\delta, _, _, _) \ \wedge\ \mathsf{stc} = (\delta, _, _, _) \ \wedge\\
&\mathcal{E}, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow (\delta, s, e, \mathit{off}) \implies [s, e) \subseteq \mathsf{reachable_addresses}(\{\mathsf{stc}, \mathsf{ddc}\}, \mathcal{M}_d)
\end{aligned}$$

Proof. We prove it by induction on the evaluation $\mathcal{E}, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow (\delta, s, e, \mathit{off})$ of the expression $\mathcal{E}$:

- **Case evalconst**,

- **Case evalCapType**,
- **Case evalCapStart**,
- **Case evalCapEnd**,
- **Case evalCapOff**, and
- **Case evalBinOp:**

  These are all vacuous cases because of disjointness of the integer values and the data capability values.

- **Case evalddc**, and
- **Case evalstc:**

  These two cases are similar. We show the proof for evalddc.

  Let $\mathsf{ddc} = (\delta, s, e, \mathit{off})$.
  By evalddc, our goal is to show that $[s, e) \subseteq \mathsf{reachable\_addresses}(\{\mathsf{stc}, \mathsf{ddc}\}, \mathcal{M}_d)$.

  By Definition 22, our goal is to show that:
  $\forall a.\ a \in [\mathsf{ddc}.s, \mathsf{ddc}.e) \implies \exists k.\ k \in [0, |\mathcal{M}_d|] \wedge a \in \mathsf{access}_{k,\mathcal{M}_d} \bigcup_{c \in \{\mathsf{stc},\mathsf{ddc}\}} [c.s, c.e)$.

  We pick $k := 0$, and by Definition 21, our goal is satisfied.

- **Case evalIncCap:**

  Here, the goal follows directly from the inductive hypothesis.

  We obtain the preconditions $\mathcal{E}, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow v$ and $v = (x, s, e, \mathit{off})$ with the inductive hypothesis being $x = \delta \implies [s, e) \subseteq \mathsf{reachable\_addresses}(\{\mathsf{stc}, \mathsf{ddc}\}, \mathcal{M}_d)$. But this is exactly our goal because $v'.s = v.s$ and $v'.e = v.e$.

- **Case evalDeref:**

  We obtain the preconditions $\mathcal{E}, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow v$, $v = (x, s, e, \mathit{off})$ and $\vdash_\delta v$,

  together with the inductive hypothesis that $[s, e) \subseteq \mathsf{reachable\_addresses}(\{\mathsf{stc}, \mathsf{ddc}\}, \mathcal{M}_d)$.

  And our goal is to show that:

  $\mathcal{M}_d(s + \mathit{off}) = (\delta, s', e', \_) \implies [s', e') \subseteq \mathsf{reachable\_addresses}(\{\mathsf{stc}, \mathsf{ddc}\}, \mathcal{M}_d)$.

  Re-writing our goal by Definition 22, it is required to show that:

  $\mathcal{M}_d(s + \mathit{off}) = (\delta, s', e', \_) \implies$
  $\forall a \in [s', e').\ \exists k.\ k \in [0, |\mathcal{M}_d|] \wedge a \in \mathsf{access}_{k,\mathcal{M}_d} \bigcup_{c \in \{\mathsf{ddc},\mathsf{stc}\}} [c.s, c.e)$.

  We observe that $s + \mathit{off} \in \mathsf{reachable\_addresses}(\{\mathsf{stc}, \mathsf{ddc}\}, \mathcal{M}_d)$ by the induction hypothesis and $\vdash_\delta v$.

  Hence, by Definition 22, we have:

  $\exists k.\ k \in [0, |\mathcal{M}_d|] \wedge s + \mathit{off} \in \mathsf{access}_{k,\mathcal{M}_d} \bigcup_{c \in \{\mathsf{ddc},\mathsf{stc}\}} [c.s, c.e)$

  Hence, by Definitions 20 and 21 of $\mathsf{access}_{k+1,\mathcal{M}_d} \bigcup_{c \in \{\mathsf{ddc},\mathsf{stc}\}} [c.s, c.e)$,

  and by assuming $\mathcal{M}_d(s + \mathit{off}) = (\delta, s', e', \_)$ (the antecedent of our goal),
  we conclude that $[s', e') \subseteq \mathsf{access}_{k+1,\mathcal{M}_d} \bigcup_{c \in \{\mathsf{ddc},\mathsf{stc}\}} [c.s, c.e)$.

  Thus, we can re-write this conclusion as:
  $\exists k.\ k \in [0, |\mathcal{M}_d| + 1] \wedge [s', e') \subseteq \mathsf{access}_{k,\mathcal{M}_d} \bigcup_{c \in \{\mathsf{ddc},\mathsf{stc}\}} [c.s, c.e)$.

But by Lemma 13 about sufficiency of $|\mathcal{M}_d|$-accessibility, our conclusion is equivalent to:
$\exists k.\ k \in [0, |\mathcal{M}_d|] \wedge [s', e') \subseteq \mathsf{access}_{k,\mathcal{M}_d} \bigcup_{c \in \{\mathsf{ddc},\mathsf{stc}\}} [c.s, c.e)$,

which satisfies our goal.

- **Case evalLim:**

  Here, we obtain the preconditions $\mathcal{E}', \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow v'$, $v' = (x, \sigma', e', \_) \in \mathit{Cap}$, with the inductive hypothesis that $x = \delta \implies [\sigma', e') \subseteq \mathsf{reachable\_addresses}(\{\mathsf{stc}, \mathsf{ddc}\}, \mathcal{M}_d)$.

  We also obtain the preconditions $[\sigma, e) \subseteq [\sigma', e')$, and $\mathcal{E}, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow (x, \sigma, e, \_)$ and our goal is to show given $x = \delta$ that $[\sigma, e) \subseteq \mathsf{reachable\_addresses}(\{\mathsf{stc}, \mathsf{ddc}\}, \mathcal{M}_d)$.

  So, our goal follows immediately by transitivity of $\subseteq$.

∎

Lemma 26 (Expression evaluation cannot forge data capabilities)**.**

$$
\begin{aligned}
&\forall s, \mathcal{E}, \sigma, e.\\
&\models_\delta s.\mathsf{ddc} \wedge\\
&\models_\delta s.\mathsf{stc} \wedge\\
&\mathcal{E}, s.\mathcal{M}_d, s.\mathsf{ddc}, s.\mathsf{stc}, s.\mathsf{pcc} \Downarrow (\delta, \sigma, e, \_)\\
&\implies\\
&((\delta, \sigma, e, \_) \subseteq s.\mathsf{ddc} \vee\\
&(\delta, \sigma, e, \_) \subseteq s.\mathsf{stc} \vee\\
&\exists a.\ (\delta, \sigma, e, \_) \subseteq s.\mathcal{M}_d(a) \ \wedge\ a \in \mathsf{reachable\_addresses}(\{s.\mathsf{ddc}, s.\mathsf{stc}\}, s.\mathcal{M}_d))
\end{aligned}
$$

Proof.

- We assume the antecedents
- And we prove our goal by induction on the evaluation of expression $\mathcal{E}$:

  1. **Case evalconst,**
  2. **Case evalBinOp,**
  3. **Case evalCapType,**
  4. **Case evalCapStart,**
  5. **Case evalCapEnd,**
  6. **Case evalCapOff:**
     In all of these cases, we notice that $\mathcal{E}, \_, \_, \_, \_ \Downarrow z$ with $z \in \mathbb{Z}$.
     This contradicts our assumed antecedent $\mathcal{E}, s.\mathcal{M}_d, s.\mathsf{ddc}, s.\mathsf{stc}, s.\mathsf{pcc} \Downarrow (\delta, \sigma, e, \_)$ because $(\delta, \_, \_, \_) \notin \mathbb{Z}$. So these cases are impossible.
  7. **Case evalddc:**
     In this case, we choose the leftmost disjunct, so our goal becomes $s.\mathsf{ddc} \subseteq s.\mathsf{ddc}$ which by the reflexivity of $\subseteq$ (Definition 3) is immediate.
  8. **Case evalstc:**
     In this case, we choose the middle disjunct, so our goal becomes $s.\mathsf{stc} \subseteq s.\mathsf{stc}$ which by the reflexivity of $\subseteq$ (Definition 3) is immediate.

9. **Case evalDeref:**
   Here, we obtain the preconditions:
   $\mathcal{E}, s.\mathcal{M}_d, s.\mathsf{ddc}, s.\mathsf{stc}, s.\mathsf{pcc} \Downarrow (\delta, \sigma, e, \mathit{off})$, $\vdash_\delta v$, and $v' = s.\mathcal{M}_d(\sigma + \mathit{off})$.
   By instantiating Lemma 25 using the preconditions $\mathcal{E}, s.\mathcal{M}_d, s.\mathsf{ddc}, s.\mathsf{stc}, s.\mathsf{pcc} \Downarrow (\delta, \sigma, e, \mathit{off})$, $\vdash_\delta v$, and our lemma assumptions, we conclude (*):
   $\sigma + \mathit{off} \in \mathsf{reachable\_addresses}(\{s.\mathsf{ddc}, s.\mathsf{stc}\}, s.\mathcal{M}_d)$

   Now, we choose the rightmost disjunct of our goal.
   We thus have two subgoals to prove.
   The left subgoal (after the choice of $a = \sigma + \mathit{off}$) is immediate by the preconditions obtained above.
   The right conjunct is exactly (*) that we proved above.

10. **Case evalIncCap:**
    Here, by Lemma 1 about the obliviousness of $\subseteq$ to the capability offset, our goal is immediate from the induction hypothesis.

11. **Case evalLim:**
    Here, our goal follows by the transitivity of $\subseteq$ from the induction hypothesis, and assumptions.

This concludes the proof of Lemma 26. □

Definition 23 (Derivable capability). *A capability $c^* = (x, s, e, \_)$ is derivable from a set of capabilities $C : 2^{Cap}$ on memory $\mathcal{M}_d$, written $C, \mathcal{M}_d \vDash c^*$ iff $\forall a \in [s, e).\ a \in \mathsf{reachable\_addresses}(C, \mathcal{M}_d)$.*

Lemma 27 (Upward closure of derivability)**.**

$$\forall c, C, C', \mathcal{M}_d.\ C, \mathcal{M}_d \vDash c \wedge C \subseteq C' \implies C', \mathcal{M}_d \vDash c$$

Proof.

- Take C'' such that $C' = C \cup C''$.
- By Definition 23, our goal is to show that:
  $\forall a \in [c.\sigma, c.e).\ a \in \mathsf{reachable\_addresses}(C \cup C'', \mathcal{M}_d)$
- By additivity (Lemma 18), it is equivalent to show that:
  $\forall a \in [c.\sigma, c.e).\ a \in \mathsf{reachable\_addresses}(C, \mathcal{M}_d) \cup \mathsf{reachable\_addresses}(C'', \mathcal{M}_d)$
- The assumption $C, \mathcal{M}_d \vDash c$ gives us:
  $\forall a \in [c.\sigma, c.e).\ a \in \mathsf{reachable\_addresses}(C, \mathcal{M}_d)$ (by Definition 23) which suffices for our goal.

□

Lemma 28 (Reachability traverses all derivable capabilities)**.**

$$\forall C, \mathcal{M}_d, c.\quad C, \mathcal{M}_d \vDash c \implies \mathsf{reachable\_addresses}(C, \mathcal{M}_d) \supseteq \mathsf{reachable\_addresses}(\{c\}, \mathcal{M}_d)$$

Proof.

- We fix arbitrary $C, \mathcal{M}_d$, and c, and assume the antecedent $C, \mathcal{M}_d \vDash c$.
- By Definition 23, we thus have:
  $\forall a \in [c.s, c.e).\ a \in \mathsf{reachable\_addresses}(C, \mathcal{M}_d)$.

- By Definition 22, we thus have (*):
  $\forall a \in [c.s, c.e).\ \exists k.\ k \in [0, |\mathcal{M}_d|] \wedge a \in \mathsf{access}_{k,\mathcal{M}_d} \bigcup_{c' \in C} [c'.s, c'.e)$.

- Our goal is to show that:
  $\mathsf{reachable\_addresses}(\{c\}, \mathcal{M}_d) \subseteq \mathsf{reachable\_addresses}(C, \mathcal{M}_d)$.

- By Definition 22, and the definition of $\subseteq$, our goal becomes:
  $\forall a.\ (\exists k.\ k \in [0, |\mathcal{M}_d|] \wedge a \in \mathsf{access}_{k,\mathcal{M}_d}[c.s, c.e)) \implies$
  $(\exists k.\ k \in [0, |\mathcal{M}_d|] \wedge a \in \mathsf{access}_{k,\mathcal{M}_d} \bigcup_{c' \in C} [c'.s, c'.e))$

- We fix an arbitrary a, assume the antecedent $k \in [0, |\mathcal{M}_d|] \wedge a \in \mathsf{access}_{k,\mathcal{M}_d}[c.s, c.e)$, and revert back a and $a \in \mathsf{access}_{k,\mathcal{M}_d}[c.s, c.e)$ to the goal.

- We prove our statement by induction on k.

  - **Base case ($k = 0$):**
    We fix an arbitrary a.
    In this case, by Definition 21, we have from our antecedent that:
    $a \in [c.s, c.e)$.
    In this case, by universal instantiation of (*), we get:
    $\exists k.\ k \in [0, |\mathcal{M}_d|] \wedge a \in \mathsf{access}_{k,\mathcal{M}_d} \bigcup_{c' \in C} [c'.s, c'.e)$, which is our goal.

  - **Inductive case ($k > 0$):**
    Here, by the induction hypothesis, we have:
    $\forall a.\ a \in \mathsf{access}_{k-1,\mathcal{M}_d}[c.s, c.e) \implies \exists k'.\ k' \in [0, |\mathcal{M}_d|] \wedge a \in \mathsf{access}_{k',\mathcal{M}_d} \bigcup_{c' \in C} [c'.s, c'.e)$
    We fix an arbitrary a, and we assume the antecedent:
    $a \in \mathsf{access}_{k,\mathcal{M}_d}[c.s, c.e)$
    We distinguish two cases by Definitions 20 and 21:

    * **Case $a \in \mathsf{access}_{k-1,\mathcal{M}_d}[c.s, c.e)$:**
      In this case, the induction hypothesis gives us that:
      $\exists k'.\ k' \in [0, |\mathcal{M}_d|] \wedge a \in \mathsf{access}_{k',\mathcal{M}_d} \bigcup_{c' \in C} [c'.s, c'.e)$, which is our goal.

    * **Case $a' \in \mathsf{access}_{k-1,\mathcal{M}_d}[c.s, c.e) \wedge \mathcal{M}_d(a') = (\delta, s, e, \_) \wedge a \in [s, e)$:**
      In this case, the induction hypothesis gives us that:
      $\exists k'.\ k' \in [0, |\mathcal{M}_d|] \wedge a' \in \mathsf{access}_{k',\mathcal{M}_d} \bigcup_{c' \in C} [c'.s, c'.e)$
      Thus, by Definition 21 of $a \in \mathsf{access}_{k'+1,\mathcal{M}_d} \bigcup_{c' \in C} [c'.s, c'.e)$, and by the case conditions $\mathcal{M}_d(a') = (\delta, s, e, \_) \wedge a \in [s, e)$, we obtain:
      $\exists k''.\ k'' \in [1, |\mathcal{M}_d| + 1] \wedge a \in \mathsf{access}_{k'',\mathcal{M}_d} \bigcup_{c' \in C} [c'.s, c'.e)$.
      By Lemma 13, we know we have:
      $\exists k''.\ k'' \in [1, |\mathcal{M}_d|] \wedge a \in \mathsf{access}_{k'',\mathcal{M}_d} \bigcup_{c' \in C} [c'.s, c'.e)$, which suffices for our goal.

$\square$

Lemma 29 (Preservation of reachability equivalence under safe memory updates)**.**

$$\begin{aligned}
&\forall C, \mathcal{M}_{d1}, \mathcal{M}_{d2}, r_1, r_2, \hat{a}, v.\\
&r_1 = \mathsf{reachable_addresses}(C, \mathcal{M}_{d1}) \wedge r_2 = \mathsf{reachable_addresses}(C, \mathcal{M}_{d2}) \ \wedge\\
&r_1 = r_2 \ \wedge\ \mathcal{M}_{d1}|_{r_1} = \mathcal{M}_{d2}|_{r_2} \ \wedge\ (C, \mathcal{M}_{d1} \vDash v \vee v \notin \{\delta\} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z})\\
&\implies \mathsf{reachable_addresses}(C, \mathcal{M}_{d1}[\hat{a} \mapsto v]) = \mathsf{reachable_addresses}(C, \mathcal{M}_{d2}[\hat{a} \mapsto v])
\end{aligned}$$

Proof.

- We fix arbitrary $C, \mathcal{M}_{d1}, \mathcal{M}_{d2}, r_1, r_2, \hat{a}, v$.
- We assume the antecedents $r_1 = \mathsf{reachable\_addresses}(C, \mathcal{M}_{d1})$, $r_2 = \mathsf{reachable\_addresses}(C, \mathcal{M}_{d2})$, $r_1 = r_2$, $\mathcal{M}_{d1}|_{r_1} = \mathcal{M}_{d2}|_{r_2}$, and $(C, \mathcal{M}_{d1} \vDash v \vee v \notin \{\delta\} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z})$, which by $r_1 = r_2$ and by Definition 23 gives us also that $(C, \mathcal{M}_{d2} \vDash v \vee v \notin \{\delta\} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z})$.

  We now distinguish two cases:

- **Case** $\hat{a} \in r_1$**:**

  In this case, we know from the assumptions $r_1 = r_2$ and $\mathcal{M}_{d1}|_{r_1} = \mathcal{M}_{d2}|_{r_2}$ that $\mathcal{M}_{d1}(\hat{a}) = \mathcal{M}_{d2}(\hat{a})$.

  We distinguish four different cases:

  - **Case** $\mathcal{M}_{d1}(\hat{a}) \neq (\delta, \_, \_, \_) \wedge v \neq (\delta, \_, \_, \_)$**:**
    - In this case, we know by Lemma 14 about irrelevance of non-δ-capability values that $r_1 = \mathsf{reachable\_addresses}(C, \mathcal{M}_{d1}[\hat{a} \mapsto 0]) = \mathsf{reachable\_addresses}(C, \mathcal{M}_{d1}[\hat{a} \mapsto v])$. And because $\mathcal{M}_{d2}(\hat{a}) = \mathcal{M}_{d1}(\hat{a}) \neq (\delta, \_, \_, \_)$, we analogously then have by Lemma 14 that $r_2 = \mathsf{reachable\_addresses}(C, \mathcal{M}_{d1}[\hat{a} \mapsto 0]) = \mathsf{reachable\_addresses}(C, \mathcal{M}_{d2}[\hat{a} \mapsto v])$.
    - So by substitution in the assumption $r_1 = r_2$, we get our goal $\mathsf{reachable\_addresses}(C, \mathcal{M}_{d1}[\hat{a} \mapsto v]) = \mathsf{reachable\_addresses}(C, \mathcal{M}_{d2}[\hat{a} \mapsto v])$.
  - **Case** $\mathcal{M}_{d1}(\hat{a}) \neq (\delta, \_, \_, \_) \wedge v = (\delta, s, e, \_)$**:**
    - By Lemma 20 about invariance to the location of v, we have: $\mathsf{reachable\_addresses}(C, \mathcal{M}_{d1}[\hat{a} \mapsto v]) = \mathsf{reachable\_addresses}(C \cup \{v\}, \mathcal{M}_{d1})$.
    - So, by Lemma 18 about "additivity in the first argument", we get: $\mathsf{reachable\_addresses}(C, \mathcal{M}_{d1}[\hat{a} \mapsto v]) =$ $\mathsf{reachable\_addresses}(C, \mathcal{M}_{d1}) \cup \mathsf{reachable\_addresses}(\{v\}, \mathcal{M}_{d1})$
    - By the assumption $C, \mathcal{M}_{d1} \vDash v \vee v \notin \{\delta\} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$, we have in this case that $C, \mathcal{M}_{d1} \vDash v$, resp. $C, \mathcal{M}_{d2} \vDash v$.
    - So, by Lemma 28, we have that: $\mathsf{reachable\_addresses}(\{v\}, \mathcal{M}_{d1}) \subseteq \mathsf{reachable\_addresses}(C, \mathcal{M}_{d1})$.
    - Thus, we obtain: $\mathsf{reachable\_addresses}(C, \mathcal{M}_{d1}[\hat{a} \mapsto v]) = \mathsf{reachable\_addresses}(C, \mathcal{M}_{d1}) = r_1$.
    - By an argument analogous to the above, we have that: $\mathsf{reachable\_addresses}(C, \mathcal{M}_{d2}[\hat{a} \mapsto v]) = \mathsf{reachable\_addresses}(C, \mathcal{M}_{d2}) = r_2$.
    - So by substitution in the assumption $r_1 = r_2$, we get our goal $\mathsf{reachable\_addresses}(C, \mathcal{M}_{d1}[\hat{a} \mapsto v]) = \mathsf{reachable\_addresses}(C, \mathcal{M}_{d2}[\hat{a} \mapsto v])$.
  - **Case** $\mathcal{M}_{d1}(\hat{a}) = (\delta, s_a, e_a, \_) \wedge v = (\delta, s, e, \_)$**:**

    In this case, we break down the memory update operation into two memory updates, namely, the update $\lambda x.\ x[\hat{a} \mapsto 0]$ followed by $\lambda x.\ x[\hat{a} \mapsto v]$.
    - So, we notice that $\mathcal{M}_{d1}[\hat{a} \mapsto v] = \mathcal{M}_{d1}[\hat{a} \mapsto 0][\hat{a} \mapsto v]$.
    - Thus, by Lemma 20 about invariance to a capability's location, we get: $\mathsf{reachable\_addresses}(C, \mathcal{M}_{d1}[\hat{a} \mapsto v]) = \mathsf{reachable\_addresses}(C \cup \{v\}, \mathcal{M}_{d1}[\hat{a} \mapsto 0])$.
    - Thus, by additivity (Lemma 18), we get: $\mathsf{reachable\_addresses}(C, \mathcal{M}_{d1}[\hat{a} \mapsto v]) =$ $\mathsf{reachable\_addresses}(C, \mathcal{M}_{d1}[\hat{a} \mapsto 0]) \cup \mathsf{reachable\_addresses}(\{v\}, \mathcal{M}_{d1}[\hat{a} \mapsto 0])$.
    - Now recall that by assumption we know $C, \mathcal{M}_{d1} \vDash v$, so we can use Lemma 28 to get: $(\ddagger\ddagger 1)$ $\mathsf{reachable\_addresses}(C, \mathcal{M}_{d1}[\hat{a} \mapsto v]) = \mathsf{reachable\_addresses}(C, \mathcal{M}_{d1}[\hat{a} \mapsto 0])$
    - By a similar argument, we also have for $\mathcal{M}_{d2}$ that: $(\ddagger\ddagger 2)$ $\mathsf{reachable\_addresses}(C, \mathcal{M}_{d2}[\hat{a} \mapsto v]) = \mathsf{reachable\_addresses}(C, \mathcal{M}_{d2}[\hat{a} \mapsto 0])$

* Next we work out the right-hand side of the $\mathcal{M}_{d1}$ equality to reach the right-hand side of the $\mathcal{M}_{d2}$ equality, thus satisfying our goal.
* First, we notice that by $\hat{a} \in r_1$, and by Lemma 24, we have that: $\hat{a} \in \mathsf{reachable\_addresses}(C, \mathcal{M}_{d1}[\hat{a} \mapsto 0])$.
* Thus, we can now use Lemma 20 with the instantiation $\mathcal{M}_d := \mathcal{M}_{d1}[\hat{a} \mapsto 0], c := \mathcal{M}_{d1}(\hat{a})$ to get:
$\mathsf{reachable\_addresses}(C \cup \{\mathcal{M}_{d1}(\hat{a})\}, \mathcal{M}_{d1}[\hat{a} \mapsto 0]) = \mathsf{reachable\_addresses}(C, \mathcal{M}_{d1}) = r_1$.
* So, by additivity (Lemma 18), we conclude that:
$\mathsf{reachable\_addresses}(C, \mathcal{M}_{d1}[\hat{a} \mapsto 0]) \subseteq r_1$
* Thus, we pick an arbitrary $a' \notin r_1$, and we know that:
it also satisfies $a' \notin \mathsf{reachable\_addresses}(C, \mathcal{M}_{d1}[\hat{a} \mapsto 0])$.
Thus, we know by Lemma 21 about invariance to unreachable memory updates that:
$\mathsf{reachable\_addresses}(C, \mathcal{M}_{d1}[\hat{a} \mapsto 0]) = \mathsf{reachable\_addresses}(C, \mathcal{M}_{d1}[\hat{a} \mapsto 0][a' \mapsto \mathcal{M}_{d2}(a')])$.
* Now by applying Lemma 21 inductively on the list of successive updates to $\mathcal{M}_{d1}$ at addresses from $\{a' \mid a' \in \mathsf{dom}(\mathcal{M}_{d1}) \cup \mathsf{dom}(\mathcal{M}_{d2}) \setminus r_1\}$, and by the assumption $\mathcal{M}_{d1}|_{r_1} = \mathcal{M}_{d2}|_{r_1}$, we get the desired transformation:
$\mathsf{reachable\_addresses}(C, \mathcal{M}_{d1}[\hat{a} \mapsto 0]) = \mathsf{reachable\_addresses}(C, \mathcal{M}_{d2}[\hat{a} \mapsto 0])$.
* By substituting the above equality in ($\ddagger\ddagger 1$), we get our goal by ($\ddagger\ddagger 2$):
$\mathsf{reachable\_addresses}(C, \mathcal{M}_{d1}[\hat{a} \mapsto v]) = \mathsf{reachable\_addresses}(C, \mathcal{M}_{d2}[\hat{a} \mapsto v])$.

– **Case $\mathcal{M}_{d1}(\hat{a}) = (\delta, s_a, e_a, \_) \wedge v \neq (\delta, \_, \_, \_)$:**
This case is very similar to the case above (unsurprisingly strictly shorter).
* First, we notice that by $\hat{a} \in r_1$, and by Lemma 24, we have that: $\hat{a} \in \mathsf{reachable\_addresses}(C, \mathcal{M}_{d1}[\hat{a} \mapsto v])$.
* Thus, we can now use Lemma 20 with the instantiation $\mathcal{M}_d := \mathcal{M}_{d1}[\hat{a} \mapsto v], c := \mathcal{M}_{d1}(\hat{a})$ to get:
$\mathsf{reachable\_addresses}(C \cup \{\mathcal{M}_{d1}(\hat{a})\}, \mathcal{M}_{d1}[\hat{a} \mapsto v]) = \mathsf{reachable\_addresses}(C, \mathcal{M}_{d1}) = r_1$.
* So, by additivity (Lemma 18), we conclude that:
$\mathsf{reachable\_addresses}(C, \mathcal{M}_{d1}[\hat{a} \mapsto v]) \subseteq r_1$
* Thus, we pick an arbitrary $a' \notin r_1$, and we know that:
it also satisfies $a' \notin \mathsf{reachable\_addresses}(C, \mathcal{M}_{d1}[\hat{a} \mapsto v])$.
Thus, we know by Lemma 21 about invariance to unreachable memory updates that:
$\mathsf{reachable\_addresses}(C, \mathcal{M}_{d1}[\hat{a} \mapsto v]) = \mathsf{reachable\_addresses}(C, \mathcal{M}_{d1}[\hat{a} \mapsto v][a' \mapsto \mathcal{M}_{d2}(a')])$.
* Now by applying Lemma 21 inductively on the list of successive updates to $\mathcal{M}_{d1}$ at addresses from $\{a' \mid a' \in \mathsf{dom}(\mathcal{M}_{d1}) \cup \mathsf{dom}(\mathcal{M}_{d2}) \setminus r_1\}$, and by the assumption $\mathcal{M}_{d1}|_{r_1} = \mathcal{M}_{d2}|_{r_1}$, we get our goal:
$\mathsf{reachable\_addresses}(C, \mathcal{M}_{d1}[\hat{a} \mapsto v]) = \mathsf{reachable\_addresses}(C, \mathcal{M}_{d2}[\hat{a} \mapsto v])$.

• **Case $\hat{a} \notin r_1$:**

By assumption $r_1 = r_2$, we also have that $\hat{a} \notin r_2$.

Thus, by Lemma 21, we have that
$\mathsf{reachable\_addresses}(C, \mathcal{M}_{d1}[\hat{a} \mapsto v]) = r_1$, and $\mathsf{reachable\_addresses}(C, \mathcal{M}_{d2}[\hat{a} \mapsto v]) = r_2$.

By substitution these two claims in the assumption $r_1 = r_2$, our goal
$\mathsf{reachable\_addresses}(C, \mathcal{M}_{d1}[\hat{a} \mapsto v]) = \mathsf{reachable\_addresses}(C, \mathcal{M}_{d2}[\hat{a} \mapsto v])$ follows.

□

Definition 24 (Shrunk access: Access set without using the capability at location a)**.**

$$\chi(A, \mathcal{M}_d, a) \overset{\mathsf{def}}{=} A \cup \{a^* \mid a^* \in [\sigma, e) \;\wedge\; \mathcal{M}_d(a') = (\delta, \sigma, e, \_) \;\wedge\; a' \in A \setminus \{a\}\}$$

Definition 25 (Shrunk k-th access: K-th access set without using the capability at location a)**.**

$$\begin{aligned}\chi_0(A, \mathcal{M}_d, a) &\stackrel{\mathsf{def}}{=} \chi(A, \mathcal{M}_d, a)\\ \chi_k(A, \mathcal{M}_d, a) &\stackrel{\mathsf{def}}{=} \chi(\chi_{k-1}(A, \mathcal{M}_d, a), \mathcal{M}_d, a)\end{aligned}$$

Lemma 30 (Additivity of χ_k)**.**

$$\forall k, A_1, A_2, \mathcal{M}_d, a.\ \chi_k(A_1 \cup A_2, \mathcal{M}_d, a) = \chi_k(A_1, \mathcal{M}_d, a) \cup \chi_k(A_2, \mathcal{M}_d, a)$$

Proof. By induction on k. Similar to Lemma 17. □

Lemma 31 (χ_k is upper-bounded by k-accessibility)**.**

$$\forall k, \mathcal{M}_d, A, a.\ \chi_k(A, \mathcal{M}_d, a) \subseteq \mathsf{access}_{k,\mathcal{M}_d} A$$

Proof. Immediate by Definitions 21 and 25. □

Lemma 32 (One capability is potentially lost from accessible addresses as a result of a non-capability update)**.**

$$\forall A, a, \mathcal{M}_d, v.\ v \neq (\delta, \_, \_, \_) \implies \mathsf{access}_{\mathcal{M}_d[a \mapsto v]} A = \chi(A, \mathcal{M}_d, a)$$

Proof.
Follows from Definitions 20 and 24 by observing that $\mathcal{M}_d[a \mapsto v](a) \neq (\delta, \_, \_, \_)$ and that $\mathcal{M}_d[a \mapsto v](a') = \mathcal{M}_d(a')$ for $a' \neq a$. □

Lemma 33 (χ_k captures k-accessibility after potential deletion of a capability)**.**

$$\forall A, a, \mathcal{M}_d, v.\ v \neq (\delta, \_, \_, \_) \implies \mathsf{access}_{k,\mathcal{M}_d[a \mapsto v]} A = \chi_k(A, \mathcal{M}_d, a)$$

Proof.
Follows by induction on k from Definitions 21 and 25 using Lemma 32. □

Lemma 34 (Reachability is captured by union over χ_k after potential deletion of a capability)**.**

$$\begin{aligned}&\forall C, \mathcal{M}_d, a, v.\ v \neq (\delta, \_, \_, \_) \implies\\ &\mathsf{reachable\_addresses}(C, \mathcal{M}_d[a \mapsto v]) = \bigcup_k (\chi_k(\bigcup_{c \in C} [c.\sigma, c.e), \mathcal{M}_d, a))\end{aligned}$$

Proof.
Immediate by Definition 22 and lemma 33. □

Lemma 35 (Accessible addresses shrink by non-δ-capability updates)**.**

$$\forall A, a, \mathcal{M}_d, v.\ v \neq (\delta, \_, \_, \_) \implies \mathsf{access}_{\mathcal{M}_d[a \mapsto v]} A \subseteq \mathsf{access}_{\mathcal{M}_d} A$$

Proof.
Immediate by Definition 20 and Lemma 32. Here is an alternative proof:

- By Definition 20, our goal is to show that:
$A \cup \bigcup_{a' \in A, \mathcal{M}_d[a \mapsto v](a') = (\delta, s, e, \_)} [s, e) \subseteq A \cup \bigcup_{a' \in A, \mathcal{M}_d(a') = (\delta, s, e, \_)} [s, e)$
- Thus, it suffices to show that:
$\bigcup_{a' \in A, \mathcal{M}_d[a \mapsto v](a') = (\delta, s, e, \_)} [s, e) \subseteq \bigcup_{a' \in A, \mathcal{M}_d(a') = (\delta, s, e, \_)} [s, e)$
- We consider an arbitrary $a' \in A$, and distinguish the following two cases:

– **Case** $a' = a$**:**

In this case, the condition $\mathcal{M}_d[a \mapsto v](a) = (\delta, \_, \_, \_)$ is not satisfied. So the set $[s, e)$ is $\emptyset$. So, we have $\emptyset \subseteq [s, e)$ for any s, e with $\mathcal{M}_d(a) = (\delta, s, e, \_)$

– **Case** $a' \neq a$**:**

In this case, $\subseteq$ follows by equality: $\mathcal{M}_d[a \mapsto v](a') = \mathcal{M}_d(a') = (\delta, s, e, \_)$.

This suffices by set identities (preservation of $\subseteq$ by $\cup$) to show our goal.

□

Lemma 36 (k-accessible addresses shrink by non-δ-capability updates)**.**

$$\forall k, A, a, \mathcal{M}_d, v.\ v \neq (\delta, \_, \_, \_) \implies \mathsf{access}_{k,\mathcal{M}_d[a\mapsto v]} A \subseteq \mathsf{access}_{k,\mathcal{M}_d} A$$

Proof.
We prove it by induction on k:

- **Base case (**$k = 0$**):**

  Trivial by $A \subseteq A$.

- **Inductive case (**$k > 0$**):**

  By the inductive hypothesis, we know $\mathsf{access}_{k-1,\mathcal{M}_d[a\mapsto v]} A \subseteq \mathsf{access}_{k-1,\mathcal{M}_d} A$.

  By Definition 21, our goal is to show that:
  $\mathsf{access}_{\mathcal{M}_d[a\mapsto v]}(\mathsf{access}_{k-1,\mathcal{M}_d[a\mapsto v]} A) \subseteq \mathsf{access}_{\mathcal{M}_d}(\mathsf{access}_{k-1,\mathcal{M}_d} A)$

  We rewrite the inductive hypothesis as: $\exists B.\ \mathsf{access}_{k-1,\mathcal{M}_d} A = B \cup \mathsf{access}_{k-1,\mathcal{M}_d[a\mapsto v]} A$.

  Thus, by substitution, our goal becomes:
  $\mathsf{access}_{\mathcal{M}_d[a\mapsto v]}(\mathsf{access}_{k-1,\mathcal{M}_d[a\mapsto v]} A) \subseteq \mathsf{access}_{\mathcal{M}_d}(B \cup \mathsf{access}_{k-1,\mathcal{M}_d[a\mapsto v]} A)$

  By additivity of $\mathsf{access}_{\mathcal{M}_d}$ (Lemma 16), it is equivalent to show:
  $\mathsf{access}_{\mathcal{M}_d[a\mapsto v]}(\mathsf{access}_{k-1,\mathcal{M}_d[a\mapsto v]} A) \subseteq \mathsf{access}_{\mathcal{M}_d}(B) \cup \mathsf{access}_{\mathcal{M}_d}(\mathsf{access}_{k-1,\mathcal{M}_d[a\mapsto v]} A)$

  By transitivity of $\subseteq$ (set identities), it suffices to show that:
  $\mathsf{access}_{\mathcal{M}_d[a\mapsto v]}(\mathsf{access}_{k-1,\mathcal{M}_d[a\mapsto v]} A) \subseteq \mathsf{access}_{\mathcal{M}_d}(\mathsf{access}_{k-1,\mathcal{M}_d[a\mapsto v]} A)$

  The latter follows immediately by Lemma 35, which proves our goal.

□

Lemma 37 (Reachability shrinks by non-δ-capability updates)**.**

$$\forall C, \mathcal{M}_d, a, v.\ v \neq (\delta, \_, \_, \_) \implies$$
$$\mathsf{reachable\_addresses}(C, \mathcal{M}_d[a \mapsto v]) \subseteq \mathsf{reachable\_addresses}(C, \mathcal{M}_d)$$

Proof.

- By Definition 22, it is equivalent to show that:
  $\bigcup_{k\in[0,|\mathcal{M}_d[a\mapsto v]|]} \mathsf{access}_{k,\mathcal{M}_d[a\mapsto v]}(\bigcup_{c\in C}[c.s, c.e)) \subseteq \bigcup_{k\in[0,|\mathcal{M}_d|]} \mathsf{access}_{k,\mathcal{M}_d}(\bigcup_{c\in C}[c.s, c.e))$

- By preservation of $\subseteq$ under $\cup$ (set identities), it suffices to show that:
  $\forall k \in [0, |\mathcal{M}_d[a \mapsto v]|].\ \mathsf{access}_{k,\mathcal{M}_d[a\mapsto v]}(\bigcup_{c\in C}[c.s, c.e)) \subseteq \mathsf{access}_{k,\mathcal{M}_d}(\bigcup_{c\in C}[c.s, c.e))$

- But for an arbitrary k, the assertion $\mathsf{access}_{k,\mathcal{M}_d[a\mapsto v]}(\bigcup_{c\in C}[c.s, c.e)) \subseteq \mathsf{access}_{k,\mathcal{M}_d}(\bigcup_{c\in C}[c.s, c.e))$ follows immediately by Lemma 36. This concludes the proof.

$\square$

Lemma 38 (Safe memory updates only shrink reachability)**.**

$$\begin{aligned}
&\forall C, \mathcal{M}_d, \hat{a}, v. \\
&\hat{a} \in \mathsf{reachable_addresses}(C, \mathcal{M}_d) \ \wedge \\
&(C, \mathcal{M}_d \vDash v \vee v \notin \{\delta\} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}) \\
&\implies \mathsf{reachable_addresses}(C, \mathcal{M}_d[\hat{a} \mapsto v]) \subseteq \mathsf{reachable_addresses}(C, \mathcal{M}_d)
\end{aligned}$$

Proof. Similarly to the proof of Lemma 29, we distinguish the following four cases:

- **Case** $v \neq (\delta, \_, \_, \_) \wedge \mathcal{M}_d(\hat{a}) \neq (\delta, \_, \_, \_)$**, and**
- **Case** $v \neq (\delta, \_, \_, \_) \wedge \mathcal{M}_d(\hat{a}) = (\delta, \sigma, e, \_)$**:**

  In these two cases, our goal follows immediately by Lemma 37.

- **Case** $C, \mathcal{M}_d \vDash v \wedge \mathcal{M}_d(\hat{a}) \neq (\delta, \_, \_, \_)$**:**

  By Definition 23, we know $v = (\delta, \sigma_v, e_v, \_)$.

  Thus, by Lemma 20, we know that:
  $\mathsf{reachable\_addresses}(C \cup \{v\}, \mathcal{M}_d) = \mathsf{reachable\_addresses}(C, \mathcal{M}_d[\hat{a} \mapsto v])$

  Thus, by additivity – Lemma 18, we have (*):
  $\mathsf{reachable\_addresses}(C, \mathcal{M}_d) \cup \mathsf{reachable\_addresses}(\{v\}, \mathcal{M}_d) = \mathsf{reachable\_addresses}(C, \mathcal{M}_d[\hat{a} \mapsto v])$

  But by Lemma 28, we know:
  $\mathsf{reachable\_addresses}(\{v\}, \mathcal{M}_d) \subseteq \mathsf{reachable\_addresses}(C, \mathcal{M}_d)$.

  Thus, we can rewrite (*) as:
  $\mathsf{reachable\_addresses}(C, \mathcal{M}_d) = \mathsf{reachable\_addresses}(C, \mathcal{M}_d[\hat{a} \mapsto v])$ which suffices for our goal.

- **Case** $C, \mathcal{M}_d \vDash v \wedge \mathcal{M}_d(\hat{a}) = (\delta, \sigma, e, \_)$**:**

  By Definition 23, we know $v = (\delta, \sigma_v, e_v, \_)$.

  Thus, by Lemma 20, we know that:
  $\mathsf{reachable\_addresses}(C \cup \{v\}, \mathcal{M}_d[\hat{a} \mapsto 0]) = \mathsf{reachable\_addresses}(C, \mathcal{M}_d[\hat{a} \mapsto v])$

  Thus, by additivity – Lemma 18, we have (**):

  $\mathsf{reachable\_addresses}(C, \mathcal{M}_d[\hat{a} \mapsto 0]) \cup \mathsf{reachable\_addresses}(\{v\}, \mathcal{M}_d[\hat{a} \mapsto 0]) =$
  $\mathsf{reachable\_addresses}(C, \mathcal{M}_d[\hat{a} \mapsto v])$

  We consider an arbitrary address $a \in \mathsf{reachable\_addresses}(C, \mathcal{M}_d[\hat{a} \mapsto v])$. We distinguish the two possible cases that arise from (**):

  - **Case** $a \in \mathsf{reachable\_addresses}(C, \mathcal{M}_d[\hat{a} \mapsto 0])$**:**
    In this case, we know by Lemma 37, and the definition of $\subseteq$ that $a \in \mathsf{reachable\_addresses}(C, \mathcal{M}_d)$, which by definition of $\subseteq$ gives us our goal.
  - **Case** $a \in \mathsf{reachable\_addresses}(\{v\}, \mathcal{M}_d[\hat{a} \mapsto 0])$**:**
    Analogously, here, we know by Lemma 37, and the definition of $\subseteq$ that:
    $a \in \mathsf{reachable\_addresses}(\{v\}, \mathcal{M}_d)$.
    But by Lemma 28, and the definition of $\subseteq$, we know that $a \in \mathsf{reachable\_addresses}(C, \mathcal{M}_d)$, which by the definition of $\subseteq$ gives our goal.

$\square$

Lemma 39 (Safe allocation adds only allocated addresses to k-accessibility)**.**

$$\begin{aligned}&\forall A, \mathcal{M}_d, \hat{a}, a_a, \sigma, e, k.\\ &\forall a \in [\sigma, e).\ \mathcal{M}_d[\hat{a} \mapsto (\delta, \sigma, e, \_)](a) = v \implies v \notin \{\delta\} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \wedge\\ &a_a \in \mathsf{access}_{k, \mathcal{M}_d[\hat{a} \mapsto (\delta, \sigma, e, \_)]} A\\ &\implies a_a \in \mathsf{access}_{k, \mathcal{M}_d} A \vee a_a \in [\sigma, e)\end{aligned}$$

Proof.

- We fix arbitrary $A, \mathcal{M}_d, \hat{a}, \sigma, e$, and we assume the antecedents.
- We prove our goal by induction on k.
  - **Base case** ($k = 0$)**:**
    We fix arbitrary a_a.
    By Definition 21, we unfold $a_a \in \mathsf{access}_{0, \mathcal{M}_d[\hat{a} \mapsto (\delta, \sigma, e, \_)]} A$ to get $a_a \in A$.
    By Definition 21, we thus conclude $a_a \in \mathsf{access}_{0, \mathcal{M}_d} A$ satisfying our goal (the left disjunct).
  - **Inductive case** ($k > 0$)**:**
    By the inductive hypothesis, we have:
    $\forall a.\ a \in \mathsf{access}_{k-1, \mathcal{M}_d[\hat{a} \mapsto (\delta, \sigma, e, \_)]} A \implies a \in \mathsf{access}_{k-1, \mathcal{M}_d} A \vee a \in [\sigma, e)$.
    We fix arbitrary a_a.
    By Definition 21, we unfold $a_a \in \mathsf{access}_{k, \mathcal{M}_d[\hat{a} \mapsto (\delta, \sigma, e, \_)]}$ to get:
    $a_a \in \mathsf{access}_{\mathcal{M}_d[\hat{a} \mapsto (\delta, \sigma, e, \_)]} (\mathsf{access}_{k-1, \mathcal{M}_d[\hat{a} \mapsto (\delta, \sigma, e, \_)]} A)$.
    By Definition 20, we distinguish two cases:
    * **Case** $a_a \in \mathsf{access}_{k-1, \mathcal{M}_d[\hat{a} \mapsto (\delta, \sigma, e, \_)]} A$**:**
      By the inductive hypothesis, we thus have:
      $a_a \in \mathsf{access}_{k-1, \mathcal{M}_d} A \vee a_a \in [\sigma, e)$.
      Two cases are possible:
      · **Case** $a_a \in \mathsf{access}_{k-1, \mathcal{M}_d} A$**:**
        By Lemma 8, we immediately obtain our goal (the left disjunct).
      · **Case** $a_a \in [\sigma, e)$**:**
        This is immediately the right disjunct of our goal.
    * **Case** $\exists a^*, \sigma^*, e^*.\ a_a \in [\sigma^*, e^*) \wedge \mathcal{M}_d[\hat{a} \mapsto (\delta, \sigma, e, \_)](a^*) = (\delta, \sigma^*, e^*, \_)$
      $\wedge\, a^* \in \mathsf{access}_{k-1, \mathcal{M}_d[\hat{a} \mapsto (\delta, \sigma, e, \_)]} A$**:**
      By instantiating the inductive hypothesis with $a^* \in \mathsf{access}_{k-1, \mathcal{M}_d[\hat{a} \mapsto (\delta, \sigma, e, \_)]} A$, we obtain: $a^* \in \mathsf{access}_{k-1, \mathcal{M}_d} A \vee a^* \in [\sigma, e)$.
      So, we consider the two possible cases:
      · **Case** $a^* \in [\sigma, e)$**:**
        In this case, we instantiate this assumed antecedent of our lemma:
        $\forall a \in [\sigma, e).\ \mathcal{M}_d[\hat{a} \mapsto (\delta, \sigma, e, \_)](a) = v \implies v \notin \{\delta\} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$ and get a contradiction to the conjunct $\mathcal{M}_d[\hat{a} \mapsto (\delta, \sigma, e, \_)](a^*) = (\delta, \sigma^*, e^*, \_)$.
        So, this case is impossible.
      · **Case** $a^* \in \mathsf{access}_{k-1, \mathcal{M}_d} A$**:**
        Here, we further distinguish two cases:
        <u>**Case** $a^* = \hat{a}$**:**</u>
        In this case, $[\sigma^*, e^*) = [\sigma, e)$. Thus, by substitution, we immediately obtain $a_a \in [\sigma, e)$ which satisfies our goal (the right disjunct).
        <u>**Case** $a^* \neq \hat{a}$**:**</u>
        In this case, we know $a^* \in \mathsf{dom}(\mathcal{M}_d)$ and $\mathcal{M}_d(a^*) = (\delta, \sigma^*, e^*, \_)$.

And already we know $a_a \in [\sigma^*, e^*)$ and $a^* \in \mathsf{access}_{k-1,\mathcal{M}_d} A$.
So, by Definitions 20 and 21, we have:
$a_a \in \mathsf{access}_{k,\mathcal{M}_d} A$ which satisfies our goal (the left disjunct).

This concludes the two cases arising from the instantiated inductive hypothesis.

This concludes the two cases arising from Definition 20, and thus concludes the inductive case of our lemma.

- This concludes the proof of Lemma 39.

□

Lemma 40 (Safe allocation adds only allocated addresses to reachability)**.**

$$
\begin{aligned}
&\forall C, \mathcal{M}_d, \hat{a}, a_a, \sigma, e.\\
&\forall a \in [\sigma, e).\ \mathcal{M}_d[\hat{a} \mapsto (\delta, \sigma, e, \_)](a) = v \implies v \notin \{\delta\} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \land\\
&a_a \in \mathsf{reachable\_addresses}(C, \mathcal{M}_d[\hat{a} \mapsto (\delta, \sigma, e, \_)])\\
&\implies a_a \in \mathsf{reachable\_addresses}(C, \mathcal{M}_d) \lor a_a \in [\sigma, e)
\end{aligned}
$$

Proof.

- We fix arbitrary $C, \mathcal{M}_d, \hat{a}, a_a, \sigma$ and e, and assume the antecedents.
- From the antecedent $a_a \in \mathsf{reachable\_addresses}(C, \mathcal{M}_d[\hat{a} \mapsto (\delta, \sigma, e, \_)])$ and by Definition 22, we have:
$\exists k.\ k \in [0, |\mathcal{M}_d[\hat{a} \mapsto \_]|\ ] \land a_a \in \mathsf{access}_{k,\mathcal{M}_d[\hat{a}\mapsto(\delta,\sigma,e,\_)]}(\bigcup_{c \in C} [c.\sigma, c.e))$
- Thus, by Lemma 39, we have:
$a_a \in \mathsf{access}_{k,\mathcal{M}_d}(\bigcup_{c \in C} [c.\sigma, c.e)) \lor a_a \in [\sigma, e)$
- We distinguish the following two cases:
  - **Case** $a_a \in \mathsf{access}_{k,\mathcal{M}_d}(\bigcup_{c \in C} [c.\sigma, c.e))$**:**
    In this case, we would like to show the left disjunct of our goal.
    By Definition 22, we would like to show that:
    $\exists k.\ k \in [0, |\mathcal{M}_d|\ ] \land a_a \in \mathsf{access}_{k,\mathcal{M}_d}(\bigcup_{c \in C} [c.\sigma, c.e))$
    Since we know our obtained k from above satisfies $k \geq |\mathcal{M}_d|$, then Lemma 13 suffices for the above re-statement of our goal.
  - **Case** $a_a \in [\sigma, e)$**:**
    Here, immediately our goal holds (its right disjunct).

□

Lemma 41 (Safe allocation causes reduction of k-accessibility to χ_k and addition of exactly the allocated addresses)**.**

$$
\begin{aligned}
&\forall A, \mathcal{M}_d, \hat{a}, a_a, \sigma, e, k.\\
&\forall a \in [\sigma, e).\ \mathcal{M}_d[\hat{a} \mapsto (\delta, \sigma, e, \_)](a) = v \implies v \notin \{\delta\} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \land\\
&\hat{a} \in \mathsf{access}_{k,\mathcal{M}_d} A\\
&\implies\\
&\mathsf{access}_{k,\mathcal{M}_d[\hat{a}\mapsto(\delta,\sigma,e,\_)]} A = \chi_k(A, \mathcal{M}_d, \hat{a}) \cup [\sigma, e)
\end{aligned}
$$

Proof. The proof should follow by induction on k, and should be similar to the proof of Lemma 39. ☐

Lemma 42 (Effect of assigning a derivable capability)**.**

$$\begin{aligned}
&\forall C, \mathcal{M}_d, a, c. \\
&C, \mathcal{M}_d \vDash c \;\wedge\; a \in \mathsf{reachable_addresses}(C, \mathcal{M}_d) \\
&\implies \\
&\mathsf{reachable_addresses}(C, \mathcal{M}_d[a \mapsto c]) = \\
&\bigcup_k \chi_k\Big(\bigcup_{c' \in C} [c'.\sigma, c'.e) \cup [c.\sigma, c.e), \mathcal{M}_d, a\Big)
\end{aligned}$$

Proof. Follows from Lemmas 17, 18, 20, 30 and 34. ☐

Lemma 43 (Assigning a derivable capability does not enlarge reachability)**.**

$$\begin{aligned}
&\forall C, \mathcal{M}_d, a, c. \\
&C, \mathcal{M}_d \vDash c \;\wedge\; a \in \mathsf{reachable_addresses}(C, \mathcal{M}_d) \\
&\implies \\
&\mathsf{reachable_addresses}(C, \mathcal{M}_d[a \mapsto c]) \subseteq \mathsf{reachable_addresses}(C, \mathcal{M}_d)
\end{aligned}$$

Proof. After substitution using Lemma 42, we apply Lemma 30 to get two subgoals that are provable using Lemma 31 and Lemma 28 respectively. ☐

Definition 26 (Sub-capability-closed predicate)**.** *For a predicate $P : \mathcal{V} \to \mathbb{B}$, sub-capability closure is defined as follows:*

$$\mathsf{subcap\_closed}(P) \stackrel{\mathsf{def}}{=} \forall x, \sigma, e, \mathit{off}, \sigma', e'.\ P(x, \sigma, e, \mathit{off}) \wedge [\sigma', e') \subseteq [\sigma, e) \implies P(x, \sigma', e', \mathit{off})$$

Definition 27 ($\mathbb{Z}$-trivial predicate)**.** *For a predicate $P : \mathcal{V} \to \mathbb{B}$, $\mathbb{Z}$-triviality is defined as follows:*

$$\mathsf{z\_trivial}(P) \stackrel{\mathsf{def}}{=} \forall z \in \mathbb{Z}.\ P\ z$$

Definition 28 (Offset-oblivious predicate)**.** *For a predicate $P : \mathcal{V} \to \mathbb{B}$, offset obliviousness is defined as follows:*

$$\mathsf{offset\_oblivious}(P) \;\stackrel{\mathsf{def}}{=}\; \forall x, \sigma, e, \mathit{off}, \mathit{off}'.\ P(x, \sigma, e, \mathit{off}) \implies P(x, \sigma, e, \mathit{off}')$$

Definition 29 (Allocation-compatible predicate)**.** *For a predicate $P : \mathcal{V} \to \mathbb{B}$, and an allocation bound ∇, allocation compatibility is defined as follows:*

$$\mathsf{allocation\_compatible}(P, \nabla) \;\stackrel{\mathsf{def}}{=}\; \forall \sigma, e.\ [\sigma, e) \subseteq (\nabla, -1] \implies P(\delta, \sigma, e, 0)$$

Definition 30 (State-universal predicate)**.** *A predicate $P : \mathcal{V} \to \mathbb{B}$ holds universally for all values of a program state s when:*

$$\begin{aligned}
\mathsf{state_universal}(P, s) \;\stackrel{\mathsf{def}}{=}\;& \forall a.\ P(s.\mathcal{M}_d(a)) \;\wedge \\
& P(s.\mathsf{ddc}) \;\wedge\; P(s.\mathsf{stc}) \;\wedge\; P(s.\mathsf{pcc}) \;\wedge \\
& \forall \mathit{mid}.\ P(s.\mathit{imp}(\mathit{mid}).\mathsf{pcc}) \;\wedge\; P(s.\mathit{imp}(\mathit{mid}).\mathsf{dcc}) \;\wedge\; P(s.\mathsf{mstc}(\mathit{mid})) \;\wedge \\
& \forall (\mathit{cc}, \mathit{dc}, _, _) \in s.\mathit{stk}.\ P(\mathit{cc}) \;\wedge\; P(\mathit{dc})
\end{aligned}$$

Lemma 44 (Predicates that are guaranteed to hold on the result of expression evaluation)**.**

$$
\begin{aligned}
&\forall \mathcal{E}, s, v. \\
&\mathcal{E}, s.\mathcal{M}_d, s.\mathsf{ddc}, s.\mathsf{stc}, s.\mathsf{pcc} \Downarrow v \ \wedge \\
&\mathsf{state\_universal}(P, s) \ \wedge \\
&\mathsf{offset\_oblivious}(P) \ \wedge \\
&\mathsf{z\_trivial}(P) \ \wedge \\
&\mathsf{subcap\_closed}(P) \\
&\implies \\
&P(v)
\end{aligned}
$$

Proof.
We assume the antecedents, and prove it by induction on expression evaluation.

1. **Case evalconst**,
2. **Case evalBinOp**,
3. **Case evalCapType**,
4. **Case evalCapStart**,
5. **Case evalCapEnd**, and
6. **Case evalCapOff:**

   All of these subgoals follow immediately by assumption $\mathsf{z\_trivial}(P)$ (unfolding Definition 27).
7. **Case evalIncCap:**

   Follows from the induction hypothesis, and by assumption $\mathsf{offset\_oblivious}(P)$ (unfolding Definition 28).
8. **Case evalDeref:**

   Follows from the assumption $\mathsf{state\_universal}(P, s)$ (unfolding Definition 30).
9. **Case evalLim:**

   Follows from the induction hypothesis, and by assumption $\mathsf{subcap\_closed}(P)$ (unfolding Definition 26).
10. **Case evalddc**, and
11. **Case evalstc:**

    Follow from assumption $\mathsf{state\_universal}(P, s)$ (unfolding Definition 30).

□

Lemma 45 (Preservation of state universality of predicates)**.**

$$\begin{aligned}
&\forall P, s, s'.\\
&s.\mathsf{nalloc} < 0 \;\wedge\\
&\mathsf{state_universal}(P, s) \;\wedge\\
&\mathsf{allocation_compatible}(P, s'.\mathsf{nalloc} - 1) \;\wedge\\
&\mathsf{offset_oblivious}(P) \;\wedge\\
&\mathsf{z_trivial}(P) \;\wedge\\
&\mathsf{subcap_closed}(P) \;\wedge\\
&s \to^* s'\\
&\;\Longrightarrow\\
&\mathsf{state_universal}(P, s') \;\wedge\; s'.\mathsf{nalloc} < 0
\end{aligned}$$

We prove $\mathsf{state\_universal}(P, s')$ by induction on $s \to^* s'$:

- **Base case:**

  Immediate by assumption.

- **Inductive case:**

  Here, we have s'' with $\mathsf{state\_universal}(P, s'')$, $s''.\mathsf{nalloc} < 0$, and $s'' \to s'$. Our goal $\mathsf{state\_universal}(P, s')$ consists of the following subgoals (by unfolding Definition 30):

  1. $\forall a.\ P(s'.\mathcal{M}_d(a))$
  2. $P(s'.\mathsf{ddc})$
  3. $P(s'.\mathsf{stc})$
  4. $P(s'.\mathsf{pcc})$
  5. $\forall mid.\ P(s'.imp(mid).\mathsf{pcc}) \;\wedge\; P(s'.imp(mid).\mathsf{dcc}) \;\wedge\; P(s'.\mathsf{mstc}(mid))$
  6. $\forall (cc, dc, \_, \_) \in s'.stk.\ P(cc) \;\wedge\; P(dc)$

  For each of the possible cases of $s'' \to s'$, we prove all of these subgoals:

  1. **Case assign:**

     Subgoals 2, 3, 5, and 6 are immediate after substitution by the induction hypothesis $\mathsf{state\_universal}(P, s'')$.

     For subgoal 4, we apply the assumption $\mathsf{offset\_oblivious}(P)$ (unfolding Definition 28), so our generated subgoal is immediate by the induction hypothesis $\mathsf{state\_universal}(P, s'')$.

     For subgoal 1, we have $s'.\mathcal{M}_d = s''.\mathcal{M}_d[c \mapsto v]$ with $\mathcal{E}_r, s''.\mathcal{M}_d, s''.\mathsf{ddc}, s''.\mathsf{stc}, s''.\mathsf{pcc} \Downarrow v$, and we distinguish two cases for an arbitrary $a \in \mathsf{dom}(s'.\mathcal{M}_d)$:

     – **Case** $a = c.\sigma + c.off$**:**
       Here, our goal $P(s'.\mathcal{M}_d(a))$ follows by Lemma 44.

     – **Case** $a \neq c.\sigma + c.off$**:**
       Here, our goal $P(s'.\mathcal{M}_d(a))$ follows by the induction hypothesis $\mathsf{state\_universal}(P, s'')$ (unfolding Definition 30).

  2. **Case allocate:**

     Subgoals 2, 3, 5, and 6 are immediate after substitution by the induction hypothesis $\mathsf{state\_universal}(P, s'')$.

For subgoal 4, we apply the assumption $\mathsf{offset\_oblivious}(P)$ (unfolding Definition 28), so our generated subgoal is immediate by the induction hypothesis $\mathsf{state\_universal}(P, s'')$.

For subgoal 1, we have:
$s'.\mathcal{M}_d = s''.\mathcal{M}_d[c \mapsto (\delta, s'.\mathsf{nalloc}, s''.\mathsf{nalloc}, 0)][i \mapsto 0 \mid i \in [s'.\mathsf{nalloc}, s''.\mathsf{nalloc})]$, and we distinguish three cases for an arbitrary $a \in \mathsf{dom}(s'.\mathcal{M}_d)$:

- **Case** $a = c.\sigma + c.\mathit{off}$**:**
  Here, our goal $P(s'.\mathcal{M}_d(a))$ follows by applying assumption $\mathsf{allocation\_compatible}(P, s'.\mathsf{nalloc} - 1)$ (unfolding Definition 29) to get the following subgoal:
  $[s'.\mathsf{nalloc}, s''.\mathsf{nalloc}) \subseteq (s'.\mathsf{nalloc} - 1, -1]$
  for which it suffices to show that:
  $s'.\mathsf{nalloc} - 1 < s'.\mathsf{nalloc}$
  (immediate), and
  $s''.\mathsf{nalloc} \leq -1$
  which is immediate by the induction hypothesis $s''.\mathsf{nalloc} < 0$.
- **Case** $a \in [s'.\mathsf{nalloc}, s''.\mathsf{nalloc})$**:**
  Here, our goal $P(s'.\mathcal{M}_d(a))$ follows by assumption $\mathsf{z\_trivial}(P)$ (unfolding Definition 27).
- **Case** $a \notin [s'.\mathsf{nalloc}, s''.\mathsf{nalloc}) \land a \neq c.\sigma + c.\mathit{off}$**:**
  Here, our goal follows by the induction hypothesis $\mathsf{state\_universal}(P, s'')$ (unfolding Definition 30).

3. **Case jump0:**
   Subgoals 1, 2, 3, 5, and 6 follow immediately after substitution by the induction hypothesis $\mathsf{state\_universal}(P, s'')$.

   Subgoal 4 follows by Lemma 44.
4. **Case jump1:**
   Subgoals 1, 2, 3, 5, and 6 follow immediately after substitution by the induction hypothesis $\mathsf{state\_universal}(P, s'')$.

   Subgoal 4 follows after applying assumption $\mathsf{offset\_oblivious}(P)$ (unfolding Definition 28) from the induction hypothesis $\mathsf{state\_universal}(P, s'')$.
5. **Case cinvoke:**
   For subgoal 1, and by inversion of cinvoke-aux, we distinguish the following three cases for an arbitrary $a \in \mathsf{dom}(s'.\mathcal{M}_d)$:

   - **Case** $a \in [s + \mathit{off}, s + \mathit{off} + \mathit{nArgs})$**:**
     Here, our goal follows by applying Lemma 44 (The generated subgoals are available by the preconditions of rule cinvoke-aux).
   - **Case** $a \in [s + \mathit{off} + \mathit{nArgs}, s + \mathit{off} + \mathit{nArgs} + \mathit{nLocal})$**:**
     Here, our goal follows from the assumption $\mathsf{z\_trivial}(P)$ (unfolding Definition 27).
   - **Case** $a \notin [s + \mathit{off}, s + \mathit{off} + \mathit{nArgs} + \mathit{nLocal})$**:**
     Here, our goal follows from the induction hypothesis $\mathsf{state\_universal}(P, s'')$ (unfolding Definition 30).

   Subgoal 2 follows by applying the induction hypothesis $\mathsf{state\_universal}(P, s'')$ (unfolding Definition 30 and applying conjunct
   $\forall \mathit{mid}.\ P(s''.\mathit{imp}(\mathit{mid}).\mathsf{pcc}) \ \land \ P(s''.\mathit{imp}(\mathit{mid}).\mathsf{dcc}) \ \land \ P(s''.\mathsf{mstc}(\mathit{mid})))$.

The generated subgoals are immediate by the preconditions of cinvoke-aux defining $s'.\mathsf{ddc}$.

Subgoal 3 follows by applying the induction hypothesis $\mathsf{state\_universal}(P, s'')$ (unfolding Definition 30 and applying conjunct
$\forall mid.\ P(s''.imp(mid).\mathsf{pcc}) \ \wedge\ P(s''.imp(mid).\mathsf{dcc}) \ \wedge\ P(s''.\mathsf{mstc}(mid)))$.
The generated subgoals are immediate by applying assumption $\mathsf{offset\_oblivious}(P)$ and the preconditions of cinvoke-aux defining $s'.\mathsf{stc}$.

Subgoal 4 follows by applying the induction hypothesis $\mathsf{state\_universal}(P, s'')$ (unfolding Definition 30 and applying conjunct
$\forall mid.\ P(s''.imp(mid).\mathsf{pcc}) \ \wedge\ P(s''.imp(mid).\mathsf{dcc}) \ \wedge\ P(s''.\mathsf{mstc}(mid)))$.
The generated subgoals are immediate by applying assumption $\mathsf{offset\_oblivious}(P)$ and the preconditions of cinvoke-aux defining $s'.\mathsf{pcc}$.

For subgoal 5, the first two conjuncts follow by applying the induction hypothesis $\mathsf{state\_universal}(P, s'')$ (unfolding Definition 30 and applying conjunct
$\forall mid.\ P(s''.imp(mid).\mathsf{pcc}) \ \wedge\ P(s''.imp(mid).\mathsf{dcc}) \ \wedge\ P(s''.\mathsf{mstc}(mid)))$. The generated subgoals are immediate by substitution.
For the third conjunct, we distinguish two cases:

- **Case** $mid = mid_{cinvoke}$**:**
  Here, the proof is the same as the proof of subgoal 3 above, after noticing the precondition $s'.\mathsf{mstc}(mid) = \mathsf{s}'.\mathsf{stc}$ of cinvoke-aux, and cinvoke.
- **Case** $mid \neq mid_{cinvoke}$**:**
  Here, again the goal follows by applying the induction hypothesis $\mathsf{state\_universal}(P, s'')$.

For subgoal 6, we distinguish the following cases:

- **Case** $(cc, dc, \_, \_) = \mathtt{top}(s'.stk)$**:**
  Here, the goal follows by applying the induction hypothesis $\mathsf{state\_universal}(P, s'')$ (the conjuncts about $s''.\mathsf{pcc}$ and $s''.\mathsf{ddc}$).
- **Case** $(cc, dc, \_, \_) \neq \mathtt{top}(s'.stk)$**:**
  Here, the goal follows by applying the induction hypothesis $\mathsf{state\_universal}(P, s'')$ (the conjunct about $s''.stk$).

6. **Case creturn:**
   Subgoal 1 follows immediately after substitution from the induction hypothesis $\mathsf{state\_universal}(P, s'')$.

   Subgoal 2 follows by applying the induction hypothesis $\mathsf{state\_universal}(P, s'')$ (the conjunct about $s''.stk$).

   Subgoal 3 follows by applying the induction hypothesis $\mathsf{state\_universal}(P, s'')$ (the conjunct about $s''.\mathsf{mstc}$).

   Subgoal 4 follows by applying the induction hypothesis $\mathsf{state\_universal}(P, s'')$ (the conjunct about $s''.stk$).

   Subgoal 5 follows by applying assumption $\mathsf{offset\_oblivious}(P)$ followed by applying the induction hypothesis $\mathsf{state\_universal}(P, s'')$ (the conjunct about $s''.\mathsf{mstc}$).

   Subgoal 6 follows from the corresponding conjunct of the induction hypothesis $\mathsf{state\_universal}(P, s'')$ after noticing that $\mathtt{elems}(s'.stk) \subset \mathtt{elems}(s''.stk)$.

7. **Case cexit:**
All subgoals are immediate after substitution by the induction hypothesis $\mathsf{state\_universal}(P, s'')$.

Definition 31 (Code capabilities have an imports origin)**.**

$$\kappa\_\mathsf{has\_origin}_{imp}(v) \stackrel{\mathsf{def}}{=} \vDash_\kappa v \implies \exists mid \in \mathsf{dom}(imp).\ v \subseteq imp(mid).\mathsf{pcc}$$

Lemma 46 ($\kappa\_\mathsf{has\_origin}_{imp}$ is sub-capability closed)**.**

$$\forall imp.\ \mathsf{subcap\_closed}(\kappa\_\mathsf{has\_origin}_{imp})$$

Proof.
By unfolding Definition 26 of sub-capability closure, we assume for arbitrary $imp, x, \sigma, e, \mathit{off}, \sigma', e'$ that $\kappa\_\mathsf{has\_origin}_{imp}(x, \sigma, e, \mathit{off})$, and that $[\sigma', e') \subseteq [\sigma, e)$.

Our goal is: $\kappa\_\mathsf{has\_origin}_{imp}(x, \sigma', e', \mathit{off})$.

By unfolding Definition 31, our goal is:
$\vDash_\kappa (x, \sigma', e', \mathit{off}) \implies \exists mid \in \mathsf{dom}(imp).\ (x, \sigma', e', \mathit{off}) \subseteq imp(mid).\mathsf{pcc}$

Two cases arise (after unfolding Definition 1):

- **Case $x = \kappa$:**
  Here, after unfolding Definition 3, our goal holds by applying the transitivity of $\subseteq$ on intervals. The generated subgoals follow from the assumptions (after unfolding Definitions 3 and 31 in the assumption).
- **Case $x \neq \kappa$:**
  Here, our goal holds vacuously.

□

Lemma 47 ($\kappa\_\mathsf{has\_origin}_{imp}$ is $\mathbb{Z}$-trivial)**.**

$$\forall imp.\ \mathsf{z\_trivial}(\kappa\_\mathsf{has\_origin}_{imp})$$

Proof.
Our goal, by unfolding Definitions 27 and 31, then Definition 1 holds vacuously. □

Lemma 48 ($\kappa\_\mathsf{has\_origin}_{imp}$ is offset oblivious)**.**

$$\forall imp.\ \mathsf{offset\_oblivious}(\kappa\_\mathsf{has\_origin}_{imp})$$

Proof.
Our goal, after unfolding Definitions 28 and 31 follows by applying Lemma 1 about the offset obliviousness of $\subseteq$. □

Lemma 49 ($\kappa\_\mathsf{has\_origin}_{imp}$ is allocation compatible)**.**

$$\forall \nabla, imp.\ \mathsf{allocation\_compatible}(\kappa\_\mathsf{has\_origin}_{imp}, \nabla)$$

Proof.
By unfolding Definition 29 of allocation-compatibility, it suffices to show for arbitrary imp that $\kappa\_\mathsf{has\_origin}_{imp}((\delta, \_, \_, \_))$.

This latter goal is vacuously true after we unfold Definition 31 then Definition 1. □

Lemma 50 (κ_has_origin$_{imp}$ is initial-state-universal)**.**

$$\forall t, s.\ t \vdash_i s \implies \mathsf{state\_universal}(\kappa\_\mathsf{has\_origin}_{s.imp}, s)$$

Proof.
We assume $t \vdash_i s$ for arbitrary t and s.
By Definition 30, we have the following subgoals:

- $\forall a.\ \kappa\_\mathsf{has\_origin}_{s.imp}(s.\mathcal{M}_d(a))$

  By unfolding Definitions 1 and 31 and inverting the assumption using initial-state, this subgoal is vacuously true.

- $\kappa\_\mathsf{has\_origin}_{s.imp}(s.\mathsf{ddc})$

  By unfolding Definitions 1 and 31 and inverting the assumption using initial-state then exec-state (obtaining $\vDash_\delta s.\mathsf{ddc}$), this subgoal is vacuously true.

- $\kappa\_\mathsf{has\_origin}_{s.imp}(s.\mathsf{stc})$

  By unfolding Definitions 1 and 31 and inverting the assumption using initial-state then exec-state (obtaining $\vDash_\delta s.\mathsf{stc}$), this subgoal is vacuously true.

- $\kappa\_\mathsf{has\_origin}_{s.imp}(s.\mathsf{pcc})$

  By unfolding Definitions 1 and 31 and inverting the assumption using initial-state, our goal is satisfied by choosing $mid = mainMod$.

- $\forall mid'.\ \kappa\_\mathsf{has\_origin}_{s.imp}(s.imp(mid').\mathsf{pcc})$

  By unfolding Definitions 1 and 31, this subgoal holds by the reflexivity of $\subseteq$ (choosing $mid = mid'$).

- $\forall mid'.\ \kappa\_\mathsf{has\_origin}_{s.imp}(s.imp(mid').\mathsf{ddc})$

  By unfolding Definitions 1 and 31, and inverting the assumption using initial-state then exec-state (obtaining $\vDash_\delta s.imp(mid').\mathsf{ddc}$), this subgoal is vacuously true.

- $\forall mid'.\ \kappa\_\mathsf{has\_origin}_{s.imp}(s.\mathsf{mstc}(mid'))$

  By unfolding Definitions 1 and 31, and inverting the assumption using initial-state then exec-state (obtaining $\vDash_\delta s.\mathsf{mstc}(mid')$), this subgoal is vacuously true.

- $\forall (cc, dc, \_, \_) \in s.stk.\ \kappa\_\mathsf{has\_origin}_{s.imp}(cc)\ \wedge\ \kappa\_\mathsf{has\_origin}_{s.imp}(dc)$

  By unfolding Definitions 1 and 31 and inverting the assumption using initial-state, this subgoal is vacuously true.

This concludes the proof of Lemma 50. ☐

Lemma 51 (κ_has_origin$_{imp}$ is universal for subsequent states)**.**

$$\forall t, s, s'.\ t \vdash_i s \wedge s \to^* s' \implies \mathsf{state\_universal}(\kappa\_\mathsf{has\_origin}_{s.imp}, s')$$

Proof.
By Lemma 50, we know (*):
$\mathsf{state\_universal}(\kappa\_\mathsf{has\_origin}_{s.imp}, s)$

We apply Lemma 45 to our goal to get the following subgoals:

- $s.\mathsf{nalloc} < 0$

  Immediate by inversion of assumption $t \vdash_i s$ using rule initial-state.

- $\mathsf{state\_universal}(\kappa\_\mathsf{has\_origin}_{s.imp}, s)$

  Immediate by (*).

- $\forall\nabla.\ \mathsf{allocation\_compatible}(\kappa\_\mathsf{has\_origin}_{s.imp}, \nabla)$

  Immediate by Lemma 49.

- $\mathsf{offset\_oblivious}(\kappa\_\mathsf{has\_origin}_{s.imp})$

  Immediate by Lemma 48.

- $\mathsf{z\_trivial}(\kappa\_\mathsf{has\_origin}_{s.imp})$

  Immediate by Lemma 47.

- $\mathsf{subcap\_closed}(\kappa\_\mathsf{has\_origin}_{s.imp})$

  Immediate by Lemma 46.

- $s \to^* s'$

  Immediate by assumption.

This concludes the proof of Lemma 51. □

Corollary 1 (There is at least one module that is executing at any time)**.**

$$\forall t : \mathit{TargetSetup}, s, s' : \mathit{TargetState}.\ t \vdash_i s \wedge s \to^* s' \implies \exists c \in \mathsf{range}(s'.imp).\ s'.\mathsf{pcc} \subseteq c.1$$

Proof.
Follows by applying Lemma 51 after unfolding Definition 30 and Definition 31. □

Lemma 52 (Preservation of $\vdash_{exec}$ by reduction)**.**

$$\forall t, s, s'.\ t \vdash_{exec} s \wedge s \to s' \implies t \vdash_{exec} s'$$

Proof. We assume the antecedent $t \vdash_{exec} s \wedge s \to s'$ for arbitrary t, s, s'.

By inversion using rules exec-state and valid-program, we obtain the following assumptions:

t definition

$t = (\mathcal{M}_c, \_, imp, \mathsf{mstc}_t, \phi)$

s definition

$s = \langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc} \rangle$

pcc type

$\vDash_\kappa \mathsf{pcc}$

ddc type

$\vDash_\delta \mathsf{ddc}$

stc type

$\vDash_\delta \mathsf{stc}$

nalloc is negative

$\mathsf{nalloc} < 0$

Domains are $modIDs$

$modIDs = \mathsf{dom}(imp) = \mathsf{dom}(\mathsf{mstc}) = \mathsf{dom}(\mathsf{mstc}_t)$

Static memory is non-negative

$\left(\bigcup_{mid \in modIDs} [imp(mid).\mathsf{ddc}.\sigma, imp(mid).\mathsf{ddc}.e) \cup [\mathsf{mstc}(mid).\sigma, \mathsf{mstc}(mid).e)\right) \cap (-\infty, 0) = \emptyset$

Types of imp and mstc

$\forall mid \in modIDs.\ \vDash_\kappa imp(mid).\mathsf{pcc}\ \wedge\ \vDash_\delta imp(mid).\mathsf{ddc}\ \wedge\ \vDash_\delta \mathsf{mstc}(mid)$

mstc capabilities are in-bounds

$\forall mid \in modIDs.\ \vdash_\delta \mathsf{mstc}(mid)$

mstc offsets correspond to the sizes of frames of the called functions

$$\forall mid \in modIDs.\ \mathsf{mstc}(mid).off = \sum_{(\_,\_,mid,fid)\in stk} \phi(mid,fid).nArgs + \phi(mid,fid).nLocal + (\mathtt{main} \in \mathsf{dom}(imp(mid).offs)\ ?\ \phi(mid,\mathtt{main}).nArgs + \phi(mid,\mathtt{main}).nLocal\ :\ 0)$$

Capability registers describe a module

$\exists mid \in modIDs.\ \mathsf{pcc}\ \doteq\ imp(mid).\mathsf{pcc}\ \wedge\ \mathsf{ddc}\ \doteq\ imp(mid).\mathsf{ddc}\ \wedge\ \mathsf{stc}\ \doteq\ \mathsf{mstc}(mid)$

stk frames describe a module

$\forall(dc, cc, \_, \_) \in \mathtt{elems}(stk).$
$\vDash_\delta dc\ \wedge\ \vDash_\kappa cc\ \wedge\ \exists mid \in modIDs.\ cc\ \doteq\ imp(mid).\mathsf{pcc}\ \wedge\ dc\ \doteq\ imp(mid).\mathsf{ddc}$

Capabilities describe parts of the memory domains

$\forall mid \in modIDs.\ imp(mid).\mathsf{pcc} \subseteq \mathsf{dom}(\mathcal{M}_c)\ \wedge\ imp(mid).\mathsf{ddc} \subseteq \mathsf{dom}(\mathcal{M}_d)$

Stack region is pre-allocated statically

$\forall mid \in modIDs.\ \mathsf{mstc}(mid) \doteq \mathsf{mstc}_t(mid)$

Data memory is addressable at static locations and newly-allocated ones

$$\mathsf{dom}(\mathcal{M}_d) = \bigcup_{mid\in modIDs} [imp(mid).\mathsf{ddc}.\sigma, imp(mid).\mathsf{ddc}.e) \cup [\mathsf{mstc}(mid).\sigma, \mathsf{mstc}(mid).e) \cup [\mathsf{nalloc}, -1)$$

Reachable addresses are addressable

$$\mathsf{reachable\_addresses}(\bigcup_{mid\in modIDs} \{imp(mid).\mathsf{ddc}, \mathsf{mstc}(mid)\}, \mathcal{M}_d) \subseteq \mathsf{dom}(\mathcal{M}_d)$$

A module does not have access to any other module's stack

$$\forall mid, a.\ a \in \mathsf{reachable\_addresses}(\{\mathsf{mstc}(mid), imp(mid).\mathsf{ddc}\}, \mathcal{M}_d) \implies a \notin \bigcup_{mid'\in modIDs\setminus\{mid\}} [\mathsf{mstc}(mid').\sigma, \mathsf{mstc}(mid').e)$$

Stack capabilities do not leak outside the stack

$\forall a, mid \in modIDs.\ \mathcal{M}_d(a) = (\delta, \sigma, e, \_)\ \wedge\ [\sigma, e) \subseteq \mathsf{mstc}(mid)\ \implies\ a \in [\mathsf{mstc}(mid).\sigma, \mathsf{mstc}(mid).e)$

Stack regions and data segments are disjoint

$\forall sc \in \mathsf{range}(\mathsf{mstc}), c \in \mathsf{range}(imp).\ sc \cap c.2 = \emptyset$

No code capability lives in memory

$\forall a.\ \mathcal{M}_d(a) \neq (\kappa, \sigma, e, \_)$

Data capabilities in memory describe addressable locations

$\forall a.\ \mathcal{M}_d(a) = (\delta, \sigma, e, \_)\ \implies\ [\sigma, e) \subseteq \mathsf{dom}(\mathcal{M}_d)$

Top of the stack mentions currently-executing module

$stk \neq \mathtt{nil}\ \implies\ \mathsf{pcc} \doteq imp(\mathtt{top}(stk).mid).\mathsf{pcc}$

Each stack frame describes the module-identity of the pcc of in the next frame

$\forall i \in [1, \mathtt{length}(stk) - 1].\ stk(i).\mathsf{pcc} \doteq imp(stk(i-1).mid).\mathsf{pcc}$

Our goal consists of similar subgoals about s'. For brevity, we use for the subgoals the same names that were used for the assumptions above.

Subgoals t **definition**, s' **definition** are immediate.

Subgoals **Domains are** modIDs, **Types of** imp **and** mstc, **Stack region is pre-allocated statically**, **Stack regions and data segments are disjoint**, and **Static memory is non-negative** follow from their corresponding assumptions by applying Lemmas 2 and 55 obtaining subgoals that are immediate by the assumption $s \to s'$.

By case distinction on the assumption $s \to s'$, we get the following cases. We prove our remaining subgoals separately for each of them:

1. **Case assign:**

   We obtain the following preconditions:

   (S-PCC-IN-BOUNDS):
   $\vdash_\kappa s.\mathsf{pcc}$

   (S'-PCC):
   $s'.\mathsf{pcc} = \mathtt{inc}(s.\mathsf{pcc}, 1)$

   (S-INSTR):
   $s.\mathcal{M}_c(s.\mathsf{pcc}) = \mathsf{Assign}\ \mathcal{E}_L\ \mathcal{E}_R$

   (ER-EVAL-V):
   $\mathcal{E}_R, s.\mathcal{M}_d, s.\mathsf{ddc}, s.\mathsf{stc}, s.\mathsf{pcc} \Downarrow v$

   (EL-EVAL-C):
   $\mathcal{E}_L, s.\mathcal{M}_d, s.\mathsf{ddc}, s.\mathsf{stc}, s.\mathsf{pcc} \Downarrow c$

   (C-IN-BOUNDS):
   $\vdash_\delta c$

   (STC-PROHIBITION):
   $\vDash_\delta v \implies (v \cap s.\mathsf{stc} = \emptyset \vee c \subseteq s.\mathsf{stc})$

   (S'-MEM):
   $s'.\mathcal{M}_d = s.\mathcal{M}_d[c \mapsto v]$

   (S'-DDC):
   $s'.\mathsf{ddc} = s.\mathsf{ddc}$

   (S'-STC):
   $s'.\mathsf{stc} = s.\mathsf{stc}$

   (S'-NALLOC):
   $s'.\mathsf{nalloc} = s.\mathsf{nalloc}$

   (S'-STK):
   $s'.\mathit{stk} = s.\mathit{stk}$

   (S'-MSTC):
   $s'.\mathsf{mstc} = s.\mathsf{mstc}$

   Subgoal $s'.\mathsf{pcc}$ **type** follows from the corresponding assumption after unfolding using (S'-PCC) and the definition of $\mathtt{inc}$.

   Subgoal $s'.\mathsf{ddc}$ **type** is immediate from the corresponding assumption after substitution using (S'-DDC).

   Subgoal $s'.\mathsf{stc}$ **type** is immediate from the corresponding assumption after substitution using (S'-STC).

   Subgal $s'.\mathsf{nalloc}$ **is negative** is immediate from the corresponding assumption after substitution using (S'-NALLOC).

   Subgoal mstc **capabilities are in-bounds** is immediate from the corresponding assumption after substitution using (S'-MSTC).

   Subgoal mstc **offsets correspond to the sizes of frames of the called functions** is immediate from the corresponding assumption after substitution using (S'-MSTC).

Subgoal **Capability registers describe a module** follows easily from the corresponding assumption after substitution using (S'-PCC), (S'-DDC), and (S'-STC) by the definition of `inc` and by instantiating Lemma 2.

Subgoal $s'.stk$ **frames describe a module** follows easily from the corresponding assumption after substitution using (S'-STK) and instantiation of Lemma 2.

Subgoal **Capabilities describe parts of the memory domains** follows easily from the corresponding assumption after substitution using (S'-MEM) and noticing that $\mathsf{dom}(s'.\mathcal{M}_d) \supseteq \mathsf{dom}(s.\mathcal{M}_d)$ and instantiation of Lemma 2.

For subgoal **Data memory is addressable at static locations and newly-allocated ones**, we have to prove:

$\mathsf{dom}(s'.\mathcal{M}_d) =$
$\bigcup_{mid \in modIDs} [s'.imp(mid).\mathsf{ddc}.\sigma, s'.imp(mid).\mathsf{ddc}.e) \cup [s'.\mathsf{mstc}(mid).\sigma, s'.\mathsf{mstc}(mid).e) \cup [s'.\mathsf{nalloc}, -1)$

By applying transitivity, it suffices to prove the following subgoals:

- $\bigcup_{mid \in modIDs} [s'.imp(mid).\mathsf{ddc}.\sigma, s'.imp(mid).\mathsf{ddc}.e) \cup [s'.\mathsf{mstc}(mid).\sigma, s'.\mathsf{mstc}(mid).e) \cup [s'.\mathsf{nalloc}, -1) =$
  $\bigcup_{mid \in modIDs} [s.imp(mid).\mathsf{ddc}.\sigma, s.imp(mid).\mathsf{ddc}.e) \cup [s.\mathsf{mstc}(mid).\sigma, s.\mathsf{mstc}(mid).e) \cup [s.\mathsf{nalloc}, -1)$
  This follows easily by substitution using (S'-NALLOC), and by using the instantiated Lemmas 2 and 55.
- $\mathsf{dom}(s.\mathcal{M}_d) = \mathsf{dom}(s'.\mathcal{M}_d)$
  - We pick an arbitrary $a \in \mathsf{dom}(s.\mathcal{M}_d)$, and we show that $a \in \mathsf{dom}(s'.\mathcal{M}_d)$. This is immediate by (S'-MEM).
  - We pick an arbitrary $a \in \mathsf{dom}(s'.\mathcal{M}_d)$, and we show that $a \in \mathsf{dom}(s.\mathcal{M}_d)$. We distinguish the following two cases:
    * **Case** $a = c.\sigma + c.\mathit{off}$**:**
      Here, by applying the definition of $\subseteq$ instantiated with assumption **Reachable addresses are addressable**, it suffices to instead show that:
      $c.\sigma + c.\mathit{off} \in \mathsf{reachable\_addresses}(\bigcup_{mid \in modIDs} \{s.imp(mid).\mathsf{ddc}, s.\mathsf{mstc}(mid)\}, s.\mathcal{M}_d)$
      By applying Lemma 18, it suffices by easy set identities to show that:
      $\exists mid \in modIDs.\ c.\sigma + c.\mathit{off} \in \mathsf{reachable\_addresses}(\{s.imp(mid).\mathsf{ddc}, s.\mathsf{mstc}(mid)\}, s.\mathcal{M}_d)$
      We then apply Lemma 19 obtaining the following subgoal (after applying some set identities):
      $\exists mid \in modIDs, C.$
      $\mathsf{addr}(C) \cup \mathsf{addr}(\{s.\mathsf{ddc}, s.\mathsf{stc}\}) = \mathsf{addr}(\{s.imp(mid).\mathsf{ddc}, s.\mathsf{mstc}(mid)\}) \wedge$
      $c.\sigma + c.\mathit{off} \in \mathsf{reachable\_addresses}(\{s.\mathsf{ddc}, s.\mathsf{stc}\}, s.\mathcal{M}_d)$
      We choose the mid given by assumption **Capability registers describe a module**.
      And choose $C := \{$
      $(\delta, s.imp(mid).\mathsf{ddc}.\sigma, s.\mathsf{ddc}.\sigma, \_),$
      $(\delta, s.\mathsf{ddc}.e, s.imp(mid).\mathsf{ddc}.e, \_),$
      $(\delta, s.\mathsf{mstc}(mid).\sigma, s.\mathsf{stc}.\sigma, \_),$
      $(\delta, s.\mathsf{stc}.e, s.\mathsf{mstc}(mid).e, \_)$
      $\}$
      The first conjunct is thus immediate by assumption **Capability registers describe a module** after unfolding the definition of addr in the goal and the Definition 3 of $\subseteq$ in the assumption.

      For the second conjunct, we apply Lemma 25, and some set identities obtaining the following subgoals:

· $\mathcal{E}, s.\mathcal{M}_d, s.\mathsf{ddc}, s.\mathsf{stc}, s.\mathsf{pcc} \Downarrow (\delta, c.\sigma, c.e, c.\mathit{off})$
Immediate by (EL-EVAL-C) and (C-IN-BOUNDS), after unfolding Definition 2.

· $c.\sigma + c.\mathit{off} \in [c.\sigma, c.e)$
Immediate by (C-IN-BOUNDS), after unfolding Definition 2.

· $s.\mathsf{pcc} = (\kappa, \_, \_, \_)$
Immediate by assumption pcc **type**.

· $s.\mathsf{ddc} = (\delta, \_, \_, \_)$
Immediate by assumption ddc **type**.

· $s.\mathsf{stc} = (\delta, \_, \_, \_)$
Immediate by assumption stc **type**.

* **Case** $a \neq c.\sigma + c.\mathit{off}$**:**
Here, by (S'-MEM), our goal is immediate.

- $\mathsf{dom}(s.\mathcal{M}_d) = \bigcup_{mid \in modIDs} [s.imp(mid).\mathsf{ddc}.\sigma, s.imp(mid).\mathsf{ddc}.e) \cup [s.\mathsf{mstc}(mid).\sigma, s.\mathsf{mstc}(mid).e) \cup [s.\mathsf{nalloc}, -1)$
This is immediate by the assumption **Data memory is addressable at static locations and newly-allocated ones**.

For subgoal **Reachable addresses are addressable**, we have to prove that:
$\mathsf{reachable\_addresses}(\bigcup_{mid \in modIDs} \{s'.imp(mid).\mathsf{ddc}, s'.\mathsf{mstc}(mid)\}, s'.\mathcal{M}_d) \subseteq \mathsf{dom}(s'.\mathcal{M}_d)$

By applying the corresponding assumption, we are left with the following two subgoals:

- $\mathsf{dom}(s.\mathcal{M}_d) = \mathsf{dom}(s'.\mathcal{M}_d)$
Proved above.

- $\mathsf{reachable\_addresses}(\bigcup_{mid \in modIDs} \{s.imp(mid).\mathsf{ddc}, s.\mathsf{mstc}(mid)\}, s.\mathcal{M}_d) =$
$\mathsf{reachable\_addresses}(\bigcup_{mid \in modIDs} \{s'.imp(mid).\mathsf{ddc}, s'.\mathsf{mstc}(mid)\}, s'.\mathcal{M}_d)$
By substitution using $s'.\mathsf{mstc} = s.\mathsf{mstc}$ and $s'.imp = s.imp$, it suffices to show that:
$\mathsf{reachable\_addresses}(\bigcup_{mid \in modIDs} \{s.imp(mid).\mathsf{ddc}, s.\mathsf{mstc}(mid)\}, s.\mathcal{M}_d) =$
$\mathsf{reachable\_addresses}(\bigcup_{mid \in modIDs} \{s.imp(mid).\mathsf{ddc}, s.\mathsf{mstc}(mid)\}, s'.\mathcal{M}_d)$
Here, we apply Lemma 38.
The generated subgoals are easy by (S'-MEM), (EL-EVAL-C) and by Lemma 25 using (ER-EVAL-V), and (C-IN-BOUNDS), unfolding Definition 23.

For subgoal **No code capability lives in memory**, we pick an arbitrary a where $a \in \mathsf{dom}(s'.\mathcal{M}_d)$.
Using (S'-MEM), we distinguish the following two cases:

- **Case** $a \neq c.\sigma + c.\mathit{off}$**:**
Here, our goal follows from assumption **No code capability lives in memory**.

- **Case** $a = c.\sigma + c.\mathit{off}$**:**
Here, our goal follows by applying Lemma 4 obtaining subgoals that are immediate by assumption ddc **type**, assumption stc **type**, assumption **No code capability lives in memory**, and by (ER-EVAL-V).

For subgoal **Data capabilities in memory describe addressable locations**,
we pick an arbitrary a where $a \in \mathsf{dom}(s'.\mathcal{M}_d)$.

Assume $s'.\mathcal{M}_d(a) = (\delta, \sigma, e, \_)$.

Our goal is: $[\sigma, e) \subseteq \mathsf{dom}(s'.\mathcal{M}_d)$.

Using (S'-MEM), we distinguish the following two cases:

- **Case** $a \neq c.\sigma + c.\mathit{off}$**:**
  Here, our goal follows from assumption **Data capabilities in memory describe addressable locations**.
- **Case** $a = c.\sigma + c.\mathit{off}$**:**
  Here, instantiate Lemma 25 using (ER-EVAL-V) and using assumptions pcc **type**, ddc **type**, and stc **type** obtaining:
  $v = (\delta, \sigma, e, \_) \implies [\sigma, e) \subseteq \mathsf{reachable\_addresses}(\{\mathsf{stc}, \mathsf{ddc}\}, \mathcal{M}_d)$
  Instantiating this using our assumption above, we obtain:
  $[\sigma, e) \subseteq \mathsf{reachable\_addresses}(\{\mathsf{stc}, \mathsf{ddc}\}, \mathcal{M}_d)$
  By transitivity of $\subseteq$ and using assumption **Reachable addresses are addressable**, we know:
  $[\sigma, e) \subseteq \mathsf{dom}(\mathcal{M}_d)$
  which is our goal.

For subgoal **A module does not have access to any other module's stack**, we have to prove:
$\forall \mathit{mid}, a.\ a \in \mathsf{reachable\_addresses}(\{s'.\mathsf{mstc}(\mathit{mid}), \mathit{imp}(\mathit{mid}).\mathsf{ddc}\}, s'.\mathcal{M}_d) \implies$
$a \notin \bigcup_{\mathit{mid}' \in \mathit{modIDs} \setminus \{\mathit{mid}\}} [s'.\mathsf{mstc}(\mathit{mid}').\sigma, s'.\mathsf{mstc}(\mathit{mid}').e)$

Fix arbitrary mid, a.

Assume $a \in \mathsf{reachable\_addresses}(\{s.\mathsf{mstc}(\mathit{mid}), \mathit{imp}(\mathit{mid}).\mathsf{ddc}\}, s'.\mathcal{M}_d)$ (applied (S'-MSTC))

Our goal is: $a \notin \bigcup_{\mathit{mid}' \in \mathit{modIDs} \setminus \{\mathit{mid}\}} [s.\mathsf{mstc}(\mathit{mid}').\sigma, s.\mathsf{mstc}(\mathit{mid}').e)$ (applied (S'-MSTC))

By instantiating Lemma 38, we know that:
$a \in \mathsf{reachable\_addresses}(\{s.\mathsf{mstc}(\mathit{mid}), \mathit{imp}(\mathit{mid}).\mathsf{ddc}\}, s.\mathcal{M}_d)$

which we use to instantiate the corresponding assumption (**A module does not have access to any other module's stack**) immediately obtaining our goal.

For subgoal **Stack capabilities do not leak outside the stack**, we have to prove:
$\forall a, \mathit{mid} \in \mathit{modIDs}.\ s'.\mathcal{M}_d(a) = (\delta, \sigma, e, \_) \ \wedge\ [\sigma, e) \subseteq s'.\mathsf{mstc}(\mathit{mid}) \implies a \in [s'.\mathsf{mstc}(\mathit{mid}).\sigma, s'.\mathsf{mstc}(\mathit{mid}).e)$

Pick arbitrary a, mid where $a \in \mathsf{dom}(s'.\mathcal{M}_d)$ and $\mathit{mid} \in \mathit{modIDs}$.

Assume $s'.\mathcal{M}_d(a) = (\delta, \sigma, e, \_)$,

and assume $[\sigma, e) \subseteq s'.\mathsf{mstc}(\mathit{mid})$.

Our goal is: $a \in [s'.\mathsf{mstc}(\mathit{mid}).\sigma, s'.\mathsf{mstc}(\mathit{mid}).e)$.

By (S'-MSTC), it suffices to prove:
$a \in [s.\mathsf{mstc}(\mathit{mid}).\sigma, s.\mathsf{mstc}(\mathit{mid}).e)$

Using (S'-MEM), distinguish the following cases:

- **Case** $a = c.\sigma + c.\mathit{off}$**:**
  By instantiating (STC-PROHIBITION) using the first assumption, we know (*):
  $v \cap s.\mathsf{stc} \neq \emptyset \implies c \subseteq s.\mathsf{stc}$
  We claim: $[\sigma, e) \subseteq s'.\mathsf{mstc}(\mathit{mid}) \implies s.\mathsf{stc} \doteq \mathsf{mstc}(\mathit{mid})$

– Using assumption **Capability registers describe a module**, obtain mid^* with: $s.\mathsf{stc} \doteq \mathsf{mstc}(mid^*)$

– Thus, our claim becomes: $\forall mid.\ [\sigma, e) \subseteq s'.\mathsf{mstc}(mid) \implies mid = mid^*$

– By Lemma 25, we know $[\sigma, e) \subseteq \mathsf{reachable\_addresses}(\{\mathsf{mstc}(mid^*), imp(mid^*).\mathsf{ddc}\}, s.\mathcal{M}_d)$
Thus, by instantiating assumption **A module does not have access to any other module's stack**, we know:
$[\sigma, e) \cap \bigcup_{mid' \in modIDs \setminus \{mid\}} [s.\mathsf{mstc}(mid').\sigma, s.\mathsf{mstc}(mid').e) = \emptyset$
Together with assumption $[\sigma, e) \subseteq s.\mathsf{mstc}(mid)$,
we conclude using set identities that $mid = mid^*$.

But then we know $[\sigma, e) \subseteq s.\mathsf{stc}$.
Thus, we instantiate (*), obtaining:
$c \subseteq s.\mathsf{stc}$
But by (C-IN-BOUNDS), we know:
$c.\sigma + c.\mathit{off} \in s.\mathsf{stc}$
Thus, by easy substitutions using our case condition, and using the claim above about mid, we obtain:
$a \in s.\mathsf{mstc}(mid)$
which is our goal.

- **Case** $a \neq c.\sigma + c.\mathit{off}$**:**
Here, by (S'-MEM), know $s.\mathcal{M}_d(a) = s'.\mathcal{M}_d(a)$.
By instantiating the corresponding assumption about $s.\mathcal{M}_d$, we know:
$s'.\mathcal{M}_d(a) = (\delta, \sigma, e, \_) \ \wedge\ \exists mid \in modIDs.\ [\sigma, e) \subseteq s.\mathsf{mstc}(mid) \implies$
$a \in [s.\mathsf{mstc}(mid).\sigma, s.\mathsf{mstc}(mid).e)$
By instantiation using the assumptions above, we immediately have our goal.

Subgoal **Top of the stack mentions currently-executing module** is immediate by substitution using (S'-STK) and (S'-PCC).

Subgoal **Each stack frame describes the module-identity of the pcc of in the next frame** is immediate by substitution using (S'-STK) and (S'-PCC).

This concludes the proof of **case assign**.

2. **Case allocate:**

We obtain the following preconditions:

(S-PCC-IN-BOUNDS):
$\vdash_\kappa s.\mathsf{pcc}$

(S'-PCC):
$s'.\mathsf{pcc} = \mathtt{inc}(s.\mathsf{pcc}, 1)$

(S-INSTR):
$s.\mathcal{M}_c(s.\mathsf{pcc}) = \mathsf{Alloc}\ \mathcal{E}_L\ \mathcal{E}_R$

(ESIZE-EVAL-V):
$\mathcal{E}_{size}, s.\mathcal{M}_d, s.\mathsf{ddc}, s.\mathsf{stc}, s.\mathsf{pcc} \Downarrow v$

(EL-EVAL-C):
$\mathcal{E}_L, s.\mathcal{M}_d, s.\mathsf{ddc}, s.\mathsf{stc}, s.\mathsf{pcc} \Downarrow c$

(C-IN-BOUNDS):
$\vdash_\delta c$

(V-POSITIVE):
$v \in \mathbb{Z}^+$

(S'-MEM):
$s'.\mathcal{M}_d = s.\mathcal{M}_d[c \mapsto (\delta, \mathsf{nalloc} - v, \mathsf{nalloc}, 0), i \mapsto 0 \ \forall i \in [\mathsf{nalloc} - v, \mathsf{nalloc})]$

(S'-DDC):
$s'.\mathsf{ddc} = s.\mathsf{ddc}$

(S'-STC):
$s'.\mathsf{stc} = s.\mathsf{stc}$

(S'-NALLOC):
$s'.\mathsf{nalloc} = s.\mathsf{nalloc} - v$

(S'-NALLOC-INF):
$s'.\mathsf{nalloc} > \nabla$

(S'-STK):
$s'.stk = s.stk$

(S'-MSTC):
$s'.\mathsf{mstc} = s.\mathsf{mstc}$

Subgoal $s'.\mathsf{pcc}$ **type** follows from the corresponding assumption after unfolding using (S'-PCC) and the definition of `inc`.

Subgoal $s'.\mathsf{ddc}$ **type** is immediate from the corresponding assumption after substitution using (S'-DDC).

Subgoal $s'.\mathsf{stc}$ **type** is immediate from the corresponding assumption after substitution using (S'-STC).

Subgal $s'.\mathsf{nalloc}$ **is negative** is immediate from the corresponding assumption after substitution using (S'-NALLOC) and noting (V-POSITIVE).

Subgoal mstc **capabilities are in-bounds** is immediate from the corresponding assumption after substitution using (S'-MSTC).

Subgoal mstc **offsets correspond to the sizes of frames of the called functions** is immediate from the corresponding assumption after substitution using (S'-MSTC).

Subgoal **Capability registers describe a module** follows easily from the corresponding assumption after substitution using (S'-PCC), (S'-DDC), and (S'-STC) by the definition of `inc` and by instantiating Lemma 2.

Subgoal $s'.stk$ **frames describe a module** follows easily from the corresponding assumption after substitution using (S'-STK) and instantiation of Lemma 2.

Subgoal **Capabilities describe parts of the memory domains** follows easily from the corresponding assumption after substitution using (S'-MEM) and noticing that $\mathsf{dom}(s'.\mathcal{M}_d) \supseteq \mathsf{dom}(s.\mathcal{M}_d)$ and instantiation of Lemma 2.

For subgoal **Data memory is addressable at static locations and newly-allocated ones**, we have to prove:

$$\mathsf{dom}(s'.\mathcal{M}_d) = \bigcup_{mid \in modIDs} [s'.imp(mid).\mathsf{ddc}.\sigma, s'.imp(mid).\mathsf{ddc}.e) \cup [s'.\mathsf{mstc}(mid).\sigma, s'.\mathsf{mstc}(mid).e) \cup [s'.\mathsf{nalloc}, -1)$$

Using (S'-MEM) and properties about the map update operator, we know that (*):
$\mathsf{dom}(s'.\mathcal{M}_d) = \mathsf{dom}(s.\mathcal{M}_d[c \mapsto \_]) \cup [s'.\mathsf{nalloc}, s.\mathsf{nalloc})$

Thus, from (*) and (S'-NALLOC) and (V-POSITIVE) and by set identities, it suffices for our goal to show:

$\mathsf{dom}(s.\mathcal{M}_d[c \mapsto \_]) =$
$\bigcup_{mid \in modIDs} [s'.imp(mid).\mathsf{ddc}.\sigma, s'.imp(mid).\mathsf{ddc}.e) \cup [s'.\mathsf{mstc}(mid).\sigma, s'.\mathsf{mstc}(mid).e) \cup [s.\mathsf{nalloc}, -1)$

This is now exactly the same as the corresponding goal in **case assign**. We omit the proof here.

For subgoal **Reachable addresses are addressable**, we have to prove that:

$\mathsf{reachable\_addresses}(\bigcup_{mid \in modIDs} \{s'.imp(mid).\mathsf{ddc}, s'.\mathsf{mstc}(mid)\}, s'.\mathcal{M}_d) \subseteq \mathsf{dom}(s'.\mathcal{M}_d)$

It suffices to show that:

$\mathsf{reachable\_addresses}(\bigcup_{mid \in modIDs} \{s.imp(mid).\mathsf{ddc}, s.\mathsf{mstc}(mid)\}, s'.\mathcal{M}_d) \subseteq \mathsf{dom}(s'.\mathcal{M}_d)$

By instantiating Lemma 40 using $\mathcal{M}_d := s.\mathcal{M}_d[i \mapsto 0 \mid i \in [s.\mathsf{nalloc} - v, s.\mathsf{nalloc})]$, and $\hat{a} := c.\sigma + c.\mathit{off}$ from (S'-MEM), we know (*):

$\mathsf{reachable\_addresses}(\bigcup_{mid \in modIDs} \{s.imp(mid).\mathsf{ddc}, s.\mathsf{mstc}(mid)\}, s'.\mathcal{M}_d) =$
$\mathsf{reachable\_addresses}(\bigcup_{mid \in modIDs} \{s.imp(mid).\mathsf{ddc}, s.\mathsf{mstc}(mid)\}, s.\mathcal{M}_d) \cup [s'.\mathsf{nalloc}, s.\mathsf{nalloc})$

And by assumption **Reachable addresses are addressable**, we know (**):

$\mathsf{reachable\_addresses}(\bigcup_{mid \in modIDs} \{s.imp(mid).\mathsf{ddc}, s.\mathsf{mstc}(mid)\}, s.\mathcal{M}_d) \subseteq \mathsf{dom}(s.\mathcal{M}_d)$

From (**) and (*) using set identities, we have:

$\mathsf{reachable\_addresses}(\bigcup_{mid \in modIDs} \{s.imp(mid).\mathsf{ddc}, s.\mathsf{mstc}(mid)\}, s'.\mathcal{M}_d) \subseteq$
$\mathsf{dom}(s.\mathcal{M}_d) \cup [s'.\mathsf{nalloc}, s.\mathsf{nalloc})$

Thus, it suffices for our goal by substitution to show that:

$\mathsf{dom}(s'.\mathcal{M}_d) = \mathsf{dom}(s.\mathcal{M}_d) \cup [s'.\mathsf{nalloc}, s.\mathsf{nalloc})$

For this, it suffices to show that:
$\mathsf{dom}(s.\mathcal{M}_d[c \mapsto \_]) = \mathsf{dom}(s.\mathcal{M}_d)$

That has been proved for the previous subgoal. We avoid repetition.

For subgoal **No code capability lives in memory**, we pick an arbitrary a where $a \in \mathsf{dom}(s'.\mathcal{M}_d)$.

Our goal is: $s'.\mathcal{M}_d(a) \neq (\kappa, \_, \_, \_)$.

Using (S'-MEM), we distinguish the following three cases:

- **Case** $a = c.\sigma + c.\mathit{off}$**:**
  Immediate by (S'-MEM).
- **Case** $a \in [s'.\mathsf{nalloc}, s.\mathsf{nalloc})$**:**
  Immediate by (S'-MEM).
- **Case** $a \notin \{c.\sigma + c.\mathit{off}\} \cup [s'.\mathsf{nalloc}, s.\mathsf{nalloc})$**:**
  Immediate by assumption **No code capability lives in memory**.

For subgoal **Data capabilities in memory describe addressable locations**,

we pick an arbitrary a where $a \in \mathsf{dom}(s'.\mathcal{M}_d)$.

Assume $s'.\mathcal{M}_d(a) = (\delta, \sigma, e, \_)$.

Our goal is: $[\sigma, e) \subseteq \mathsf{dom}(s'.\mathcal{M}_d)$.

Using (S'-MEM), we distinguish the following three cases:

- **Case** $a = c.\sigma + c.\mathit{off}$**:**
  Here, our goal follows by the map update operator in (S'-MEM).
- **Case** $a \in [s'.\mathsf{nalloc}, s.\mathsf{nalloc})$**:**
  Here, our goal is true after deriving a contradiction to assumption $s'.\mathcal{M}_d(a) = (\delta, \_, \_, \_)$.
- **Case** $a \notin \{c.\sigma + c.\mathit{off}\} \cup [s'.\mathsf{nalloc}, s.\mathsf{nalloc})$**:**
  Here, our goal follows by instantiating assumption **Data capabilities in memory describe addressable locations**.

For subgoal **A module does not have access to any other module's stack**, we have to prove:

$\forall \mathit{mid}, a.\ a \in \mathsf{reachable\_addresses}(\{s'.\mathsf{mstc}(\mathit{mid}), \mathit{imp}(\mathit{mid}).\mathsf{ddc}\}, s'.\mathcal{M}_d) \implies$
$a \notin \bigcup_{\mathit{mid}' \in \mathit{modIDs} \setminus \{\mathit{mid}\}} [s'.\mathsf{mstc}(\mathit{mid}').\sigma, s'.\mathsf{mstc}(\mathit{mid}').e)$

Fix arbitrary mid, a.

Assume $a \in \mathsf{reachable\_addresses}(\{s.\mathsf{mstc}(\mathit{mid}), \mathit{imp}(\mathit{mid}).\mathsf{ddc}\}, s'.\mathcal{M}_d)$ (applied (S'-MSTC))

Our goal is: $a \notin \bigcup_{\mathit{mid}' \in \mathit{modIDs} \setminus \{\mathit{mid}\}} [s.\mathsf{mstc}(\mathit{mid}').\sigma, s.\mathsf{mstc}(\mathit{mid}').e)$ (applied (S'-MSTC))

By instantiating Lemma 40 using $\mathcal{M}_d := s.\mathcal{M}_d[i \mapsto 0 \mid i \in [s.\mathsf{nalloc} - v, s.\mathsf{nalloc})]$, and $\hat{a} := c.\sigma + c.\mathit{off}$ from (S'-MEM), we know (*):

$\mathsf{reachable\_addresses}(\bigcup_{\mathit{mid} \in \mathit{modIDs}} \{s.\mathit{imp}(\mathit{mid}).\mathsf{ddc}, s.\mathsf{mstc}(\mathit{mid})\}, s'.\mathcal{M}_d) =$
$\mathsf{reachable\_addresses}(\bigcup_{\mathit{mid} \in \mathit{modIDs}} \{s.\mathit{imp}(\mathit{mid}).\mathsf{ddc}, s.\mathsf{mstc}(\mathit{mid})\}, s.\mathcal{M}_d) \cup [s'.\mathsf{nalloc}, s.\mathsf{nalloc})$

Thus, distinguish two cases:

- **Case** $a \in \mathsf{reachable\_addresses}(\bigcup_{\mathit{mid} \in \mathit{modIDs}} \{s.\mathit{imp}(\mathit{mid}).\mathsf{ddc}, s.\mathsf{mstc}(\mathit{mid})\}, s.\mathcal{M}_d)$**:**
  Here, instantiate the corresponding assumption, **A module does not have access to any other module's stack**, obtaining our goal.
- **Case** $a \in [s'.\mathsf{nalloc}, s.\mathsf{nalloc})$**:**
  Here, our goal follows from both assumptions **Static memory is non-negative** and nalloc **is negative**.

For subgoal **Stack capabilities do not leak outside the stack**, we have to prove:

$\forall a, \mathit{mid} \in \mathit{modIDs}.\ s'.\mathcal{M}_d(a) = (\delta, \sigma, e, \_) \ \wedge\ [\sigma, e) \subseteq s'.\mathsf{mstc}(\mathit{mid}) \implies a \in [s'.\mathsf{mstc}(\mathit{mid}).\sigma, s'.\mathsf{mstc}(\mathit{mid}).e)$

Pick arbitrary a, mid where $a \in \mathsf{dom}(s'.\mathcal{M}_d)$ and $\mathit{mid} \in \mathit{modIDs}$.

Assume $s'.\mathcal{M}_d(a) = (\delta, \sigma, e, \_)$,

and assume $[\sigma, e) \subseteq s'.\mathsf{mstc}(\mathit{mid})$.

Our goal is: $a \in [s'.\mathsf{mstc}(\mathit{mid}).\sigma, s'.\mathsf{mstc}(\mathit{mid}).e)$.

By (S'-MSTC), it suffices to prove:
$a \in [s.\mathsf{mstc}(mid).\sigma, s.\mathsf{mstc}(mid).e)$

Using (S'-MEM), distinguish the following cases:

- **Case** $a = c.\sigma + c.\mathit{off}$**:**
  Here, our goal is provable after deriving a contradiction to assumption $[s'.\mathsf{nalloc}, s.\mathsf{nalloc}) \subseteq s'.\mathsf{mstc}(mid)$ from assumptions **Static memory is non-negative** and nalloc **is negative**.
- **Case** $a \in [s'.\mathsf{nalloc}, s.\mathsf{nalloc})$**:**
  Here, our goal is provable after deriving a contradiction to assumption $s'.\mathcal{M}_d(a) = (\delta, \_, \_, \_)$ using (S'-MEM).
- **Case** $a \notin [s'.\mathsf{nalloc}, s.\mathsf{nalloc}) \cup \{c.\sigma + c.\mathit{off}\}$**:**
  Follows from the corresponding assumption, **Stack capabilities do not leak outside the stack** using (S'-MEM).

Subgoal **Top of the stack mentions currently-executing module** is immediate by substitution using (S'-STK) and (S'-PCC).

Subgoal **Each stack frame describes the module-identity of the pcc of in the next frame** is immediate by substitution using (S'-STK) and (S'-PCC).

This concludes the proof of **case allocate**.

3. **Case jump0:**

   We obtain the following preconditions:

   (S-PCC-IN-BOUNDS):
   $\vdash_\kappa s.\mathsf{pcc}$

   (S-INSTR):
   $s.\mathcal{M}_c(s.\mathsf{pcc}) = \mathsf{JumpIfZero}\ \mathcal{E}_{cond}\ \mathcal{E}_{\mathit{off}}$

   (ECOND-EVAL-V):
   $\mathcal{E}_{cond}, s.\mathcal{M}_d, s.\mathsf{ddc}, s.\mathsf{stc}, s.\mathsf{pcc} \Downarrow v$

   (V-ZERO):
   $v = 0$

   (EOFF-EVAL-OFF):
   $\mathcal{E}_{\mathit{off}}, s.\mathcal{M}_d, s.\mathsf{ddc}, s.\mathsf{stc}, s.\mathsf{pcc} \Downarrow \mathit{off}$

   (OFF-INTEGER):
   $\mathit{off} \in \mathbb{Z}$

   (S'-PCC):
   $s'.\mathsf{pcc} = \mathtt{inc}(s.\mathsf{pcc}, \mathit{off})$

   (S'-MEM):
   $s'.\mathcal{M}_d = s.\mathcal{M}_d$

   (S'-DDC):
   $s'.\mathsf{ddc} = s.\mathsf{ddc}$

   (S'-STC):
   $s'.\mathsf{stc} = s.\mathsf{stc}$

   (S'-NALLOC):
   $s'.\mathsf{nalloc} = s.\mathsf{nalloc}$

   (S'-STK):
   $s'.stk = s.stk$

(S'-MSTC):
$s'.\mathsf{mstc} = s.\mathsf{mstc}$

Subgoal $s'.\mathsf{pcc}$ **type** follows from the corresponding assumption after unfolding using (S'-PCC) and the definition of `inc`.

Subgoal **Capability registers describe a module** follows easily from the corresponding assumption after substitution using (S'-PCC), (S'-DDC), and (S'-STC) by the definition of `inc` and by instantiating Lemma 2.

All other subgoals are immediate by the corresponding assumptions after substitution from the preconditions.

4. **Case jump1:**

   We obtain the following preconditions:

   (S-PCC-IN-BOUNDS):
   $\vdash_\kappa s.\mathsf{pcc}$

   (S-INSTR):
   $s.\mathcal{M}_c(s.\mathsf{pcc}) = \mathsf{JumpIfZero}\ \mathcal{E}_{cond}\ \mathcal{E}_{off}$

   (ECOND-EVAL-V):
   $\mathcal{E}_{cond}, s.\mathcal{M}_d, s.\mathsf{ddc}, s.\mathsf{stc}, s.\mathsf{pcc} \Downarrow v$

   (V-NON-ZERO):
   $v \neq 0$

   (S'-PCC):
   $s'.\mathsf{pcc} = \mathtt{inc}(s.\mathsf{pcc}, 1)$

   (S'-MEM):
   $s'.\mathcal{M}_d = s.\mathcal{M}_d$

   (S'-DDC):
   $s'.\mathsf{ddc} = s.\mathsf{ddc}$

   (S'-STC):
   $s'.\mathsf{stc} = s.\mathsf{stc}$

   (S'-NALLOC):
   $s'.\mathsf{nalloc} = s.\mathsf{nalloc}$

   (S'-STK):
   $s'.stk = s.stk$

   (S'-MSTC):
   $s'.\mathsf{mstc} = s.\mathsf{mstc}$

   Subgoal $s'.\mathsf{pcc}$ **type** follows from the corresponding assumption after unfolding using (S'-PCC) and the definition of `inc`.

   Subgoal **Capability registers describe a module** follows easily from the corresponding assumption after substitution using (S'-PCC), (S'-DDC), and (S'-STC) by the definition of `inc` and by instantiating Lemma 2.

   All other subgoals are immediate by the corresponding assumptions after substitution from the preconditions.

5. **Case cinvoke:**

   We obtain the following preconditions (after inversion using cinvoke-aux):

   (S-PCC-IN-BOUNDS):
   $\vdash_\kappa s.\mathsf{pcc}$

   (S-INSTR):
   $s.\mathcal{M}_c(s.\mathsf{pcc}) = \mathsf{Cinvoke}\ \mathit{mid}_{call}\ \mathit{fid}_{call}\ \overline{e}$

   (S'-STK):
   $s'.\mathit{stk} = \mathtt{push}(s.\mathit{stk}, (s.\mathsf{ddc}, s.\mathsf{pcc}, \mathit{mid}_{call}, \mathit{fid}_{call}))$

   (PHI-MID-FID):
   $\phi(\mathit{mid}_{call}, \mathit{fid}_{call}) = (\mathit{nArgs}, \mathit{nLocal})$

   (MSTC-MID):
   $s.\mathsf{mstc}(\mathit{mid}_{call}) = (\delta, \sigma, e, \mathit{off})$

   (S'-STC):
   $s'.\mathsf{stc} = (\delta, \sigma, e, \mathit{off} + \mathit{nArgs} + \mathit{nLocal})$

   (Es-EVAL):
   $\forall i \in [0, \mathit{nArgs}).\ \overline{e}(i), s.\mathcal{M}_d, s.\mathsf{ddc}, s.\mathsf{stc}, s.\mathsf{pcc} \Downarrow v_i$

   (NO-STC-LEAK):
   $\forall i \in [0, \mathit{nArgs}).\ \vDash_\delta v_i \implies v_i \cap s.\mathsf{stc} = \emptyset$

   (S'-MEM):
   $s'.\mathcal{M}_d = s.\mathcal{M}_d[\sigma + \mathit{off} + i \mapsto v_i\ \forall i \in [0, \mathit{nArgs})][\sigma + \mathit{off} + \mathit{nArgs} + i \mapsto 0\ \forall i \in [0, \mathit{nLocal})]$

   (S'-MSTC):
   $\mathsf{mstc}' = \mathsf{mstc}[\mathit{mid}_{call} \mapsto \mathsf{stc}']$

   (IMP-MID):
   $(c, d, \mathit{offs}) = \mathit{imp}(\mathit{mid}_{call})$

   (S'-DDC):
   $s'.\mathsf{ddc} = d$

   (S'-PCC):
   $s'.\mathsf{pcc} = \mathtt{inc}(c, \mathit{offs}(\mathit{fid}))$

   (S'-STC-IN-BOUNDS):
   $\vdash_\delta s'.\mathsf{stc}$

   Subgoal $s'.\mathsf{pcc}$ **type** follows from assumption **Types of** imp **and** mstc instantiated with mid_{call} after substitution from (IMP-MID) in (S'-PCC) and unfolding the definition of $\mathtt{inc}$.

   Subgoal $s'.\mathsf{ddc}$ **type** follows from assumption **Types of** imp **and** mstc instantiated with mid_{call} after substitution from (IMP-MID) in (S'-DDC).

   Subgoal $s'.\mathsf{stc}$ **type** is immediate from the corresponding assumption and (S'-STC).

   Subgal $s'.\mathsf{nalloc}$ **is negative** is immediate from the corresponding assumption after substitution using (S'-NALLOC).

   Subgoal mstc **capabilities are in-bounds** follows from (S'-MSTC) and (S'-STC-IN-BOUNDS).

   Subgoal mstc **offsets correspond to the sizes of frames of the called functions** follows by easy arithmetic after substitution using (S'-MSTC), (S'-STC), and (S'-STK).

Subgoal **Capability registers describe a module** follows easily from (S'-PCC), (S'-DDC), and (S'-STC) after substitution using (MSTC-MID) and (IMP-MID).

For subgoal $s'.stk$ **frames describe a module**, we distinguish two cases for arbitrary dc, cc with $(dc, cc, \_, \_) \in \texttt{elems}(s.stk)$:

- **Case** $\texttt{top}(s'.stk) = (dc, cc, \_, \_)$**:**
  Here, our goal follows from assumptions pcc **type**, ddc **type**, and **Capability registers describe a module** after unfolding (S'-STK).
- **Case** $\texttt{top}(s'.stk) \neq (dc, cc, \_, \_)$**:**
  Here, our goal follows from the corresponding assumption, stk **frames describe a module**.

Subgoal **Capabilities describe parts of the memory domains** follows easily from the corresponding assumption after substitution using (S'-MEM) and noticing that $\mathsf{dom}(s'.\mathcal{M}_d) \supseteq \mathsf{dom}(s.\mathcal{M}_d)$ and instantiation of Lemma 2.

For subgoal **Data memory is addressable at static locations and newly-allocated ones**, we have to prove:

$$\mathsf{dom}(s'.\mathcal{M}_d) = \bigcup_{mid \in modIDs} [s'.imp(mid).\mathsf{ddc}.\sigma, s'.imp(mid).\mathsf{ddc}.e) \cup [s'.\mathsf{mstc}(mid).\sigma, s'.\mathsf{mstc}(mid).e) \cup [s'.\mathsf{nalloc}, -1)$$

Notice by Lemma 2 and by substitution using (S'-MSTC), (S'-STC), and (S'-NALLOC) that it suffices to prove:

$$\mathsf{dom}(s'.\mathcal{M}_d) = \bigcup_{mid \in modIDs} [s.imp(mid).\mathsf{ddc}.\sigma, s.imp(mid).\mathsf{ddc}.e) \cup [s.\mathsf{mstc}(mid).\sigma, s.\mathsf{mstc}(mid).e) \cup [s.\mathsf{nalloc}, -1)$$

Thus, by substitution using assumption **Data memory is addressable at static locations and newly-allocated ones**, it suffices to prove:

$$\mathsf{dom}(s'.\mathcal{M}_d) = \mathsf{dom}(s.\mathcal{M}_d)$$

Thus, it suffices by (S'-MEM) to prove $[\sigma + \mathit{off}, \sigma + \mathit{off}') \subseteq \mathsf{dom}(s.\mathcal{M}_d)$.

By substitution again using assumption **Data memory is addressable at static locations and newly-allocated ones**, it suffices to prove:

$$[\sigma + \mathit{off}, \sigma + \mathit{off}') \subseteq [s.\mathsf{mstc}(mid_{call}).\sigma, s.\mathsf{mstc}(mid_{call}).e).$$

This follows from (S'-STC-IN-BOUNDS) and from assumption mstc **capabilities are in-bounds**.

For subgoal **Reachable addresses are addressable**, we have to prove that:

$$\mathsf{reachable\_addresses}(\bigcup_{mid \in modIDs} \{s'.imp(mid).\mathsf{ddc}, s'.\mathsf{mstc}(mid)\}, s'.\mathcal{M}_d) \subseteq \mathsf{dom}(s'.\mathcal{M}_d)$$

By Lemmas 6 and 18 instantiated using (S'-MSTC), it suffices to show that:

$$\mathsf{reachable\_addresses}(\bigcup_{mid \in modIDs} \{s.imp(mid).\mathsf{ddc}, s.\mathsf{mstc}(mid)\}, s'.\mathcal{M}_d) \subseteq \mathsf{dom}(s'.\mathcal{M}_d)$$

This follows similarly as in case assign.

Subgoal **No code capability lives in memory** follows similarly as in case assign.

Subgoal **Data capabilities in memory describe addressable locations** follows similarly as in case assign.

Subgoal **A module does not have access to any other module's stack** is similar to the same subgoal of case assign.

For subgoal **Stack capabilities do not leak outside the stack**, we have to prove:
$\forall a, mid \in modIDs.\ s'.\mathcal{M}_d(a) = (\delta, \sigma, e, \_) \ \wedge\ [\sigma, e) \subseteq s'.\mathsf{mstc}(mid) \implies a \in [s'.\mathsf{mstc}(mid).\sigma, s'.\mathsf{mstc}(mid).e)$

Pick arbitrary a, mid where $a \in \mathsf{dom}(s'.\mathcal{M}_d)$ and $mid \in modIDs$.

Assume $s'.\mathcal{M}_d(a) = (\delta, \sigma, e, \_)$,

and assume $[\sigma, e) \subseteq s'.\mathsf{mstc}(mid)$.

Our goal is: $a \in [s'.\mathsf{mstc}(mid).\sigma, s'.\mathsf{mstc}(mid).e)$.

By (S'-MSTC) and (S'-STC), it suffice to prove:
$a \in [s.\mathsf{mstc}(mid).\sigma, s.\mathsf{mstc}(mid).e)$

Using (S'-MEM), distinguish the following cases:

- **Case** $a \in [\sigma + \mathit{off}, \sigma + \mathit{off} + nArgs)$**:**
  This is similar, after instantiating (NO-STC-LEAK) to the corresponding sub-case of **case assign**.
- **Case** $a \in [\sigma + \mathit{off} + nArgs, \sigma + \mathit{off} + nArgs + nLocal)$**:**
  Here, by contradiction from (S'-MEM) to assumption $s'.\mathcal{M}_d(a) = (\delta, \sigma, e, \_)$, our goal follows vacuously.
- **Case** $a \notin [\sigma + \mathit{off}, \sigma + \mathit{off} + nArgs + nLocal)$**:**
  Here, have $s'.\mathcal{M}_d(a) = s.\mathcal{M}_d(a)$ by (S'-MEM).
  Thus, goal follows by instantiating the corresponding assumption **Stack capabilities do not leak outside the stack**.

Subgoal **Top of the stack mentions currently-executing module** follows immediately from the preconditions (S'-STK), (S'-PCC), and (IMP-MID).

Subgoal **Each stack frame describes the module-identity of the pcc of in the next frame** follows in one case from assumption **Top of the stack mentions currently-executing module** after noticing the precondition (S'-STK), and in the other cases from the corresponding assumption.

This concludes the proof of **case cinvoke**.

6. **Case creturn:**

   We obtain the following preconditions:

   (S-PCC-IN-BOUNDS):
   $\vdash_\kappa s.\mathsf{pcc}$

   (S-INSTR):
   $s.\mathcal{M}_c(s.\mathsf{pcc}) = \mathsf{Creturn}$

   (S'-STK-DDC-PCC):
   $stk', (\mathsf{ddc}', \mathsf{pcc}', mid, \mathit{fid}) = \mathsf{pop}(stk)$

   (PHI-MID-FID):
   $\phi(mid, \mathit{fid}) = (nArgs, nLocal)$

(MSTC-MID):
$(\delta, s, e, \mathit{off}) = \mathsf{mstc}(\mathit{mid})$

(OFF'):
$\mathit{off}' = \mathit{off} - \mathit{nArgs} - \mathit{nLocal}$

(S'-MSTC-MID):
$\mathsf{mstc}' = \mathsf{mstc}[\mathit{mid} \mapsto (\delta, s, e, \mathit{off}')]$

(S'-STC):
$\exists \mathit{mid}'.\ \mathsf{pcc}' \doteq \mathit{imp}(\mathit{mid}').\mathsf{pcc}\ \wedge\ \mathsf{stc}' = \mathsf{mstc}(\mathit{mid}')$

(S'-MEM):
$s'.\mathcal{M}_d = s.\mathcal{M}_d$

(S'-NALLOC):
$s'.\mathsf{nalloc} = s.\mathsf{nalloc}$

Subgoal $s'.\mathsf{pcc}$ **type** follows from assumption stk **frames describe a module** after substitution using (S'-STK-DDC-PCC).

Subgoal $s'.\mathsf{ddc}$ **type** follows from assumption stk **frames describe a module** after substitution using (S'-STK-DDC-PCC).

Subgoal $s'.\mathsf{stc}$ **type** follows from assumption **Types of** imp **and** mstc after substitution using (S'-STC).

Subgoal $s'.\mathsf{nalloc}$ **is negative** is immediate from the corresponding assumption after substitution using (S'-NALLOC).

For subgoal mstc **capabilities are in-bounds**, we fix an arbitrary mid' such that $\mathit{mid}' \in \mathit{modIDs}$.

Our goal (after unfolding Definition 2, applying arithmetic, and removing the already proven conjunct, $\vDash_\delta s'.\mathsf{mstc}(\mathit{mid}')$) is:

$s'.\mathsf{mstc}(\mathit{mid}').\mathit{off} \in [0, s'.\mathsf{mstc}(\mathit{mid}').e - s'.\mathsf{mstc}(\mathit{mid}').\sigma)$

Distinguish two cases:

- **Case** $\mathit{mid}' = \mathit{mid}$**:**
  Here, our goal follows by arithmetic after substitutions using (S'-STK-DDC-PCC), (PHI-MID-FID), (OFF'), (S'-MSTC-MID), and assumption mstc **offsets correspond to the sizes of frames of the called functions**.
- **Case** $\mathit{mid}' \neq \mathit{mid}$**:**
  Here, goal follows from the corresponding assumption mstc **capabilities are in-bounds**.

Subgoal mstc **offsets correspond to the sizes of frames of the called functions** follows by arithmetic after substitutions using (S'-STK-DDC-PCC), (S'-MSTC-MID), (OFF)', and (PHI-MID-FID).

Subgoal **Capability registers describe a module** follows from assumptions stk **frames describe a module**, and (S'-STC) after substitution using (S'-STK-DDC-PCC).

Subgoal *stk* **frames describe a module** follows by instantiating the corresponding assumption after noticing from (S'-STK-DDC-PCC) that $\texttt{elems}(s'.stk) \subset \texttt{elems}(s.stk)$.

Subgoal **Capabilities describe parts of the memory domain** follows by substitution using (S'-MEM) and Lemma 2 from the corresponding assumption.

Subgoal **Data memory is addressable at static locations and newly-allocated ones** follows from the corresponding assumption after substitution using (S'-MEM) and (S'-NALLOC).

Subgoal **Reachable addresses are addressable** follows from the corresponding assumption after substitution using (S'-MEM).

Subgoal **A module does not have access to any other module's stack** follows from the corresponding assumption after substitution using (S'-MEM).

Subgoal **Stack capabilities do not leak outside the stack** follows from the corresponding assumption after substitution using (S'-MEM).

Subgoal **No code capability lives in memory** follows from the corresponding assumption after substitution using (S'-MEM).

Subgoal **Data capabilities in memory describe addressable locations** follows from the corresponding assumption after substitution using (S'-MEM).

Subgoal **Top of the stack mentions currently-executing module** follows from assumption **Each stack frame describes the module-identity of the pcc of in the next frame** by noticing the precondition (S'-STK-DDC-PCC).

Subgoal **Each stack frame describes the module-identity of the pcc of in the next frame** follows immediately from the corresponding assumption after noticing the precondition (S'-STK).

This concludes the proof of **case creturn**.

7. **Case cexit:**

   All goals are immediate by substitution. Notice that $s' = s$.

This concludes the proof of Lemma 52. □

Corollary 2 (Preservation of $\vdash_{exec}$ by $\to^*$)**.**
$\forall t, s, s'.\ t \vdash_{exec} s \wedge s \to^* s' \implies t \vdash_{exec} s'$

Proof. Easy by Lemma 52. □

Corollary 3 (Data and stack capabilities always hold a data-capability value)**.**

$$\forall t : TargetSetup, s, s' : TargetState.\ t \vdash_{exec} s \wedge s \to^* s' \implies$$

$$(s'.\mathsf{ddc} \in \{\delta\} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \wedge s'.\mathsf{stc} \in \{\delta\} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z})$$

Proof.
Follows by Lemma 52. □

Lemma 53 (Preservation of $\vdash_{exec}$ by $\succ_{\approx}$)**.**
$\forall t, s, s'.\ t \vdash_{exec} s \wedge s \succ_{\approx} s' \implies t \vdash_{exec} s'$

Proof. After inversion of the assumptions using rules cinvoke-aux and exec-state, the proof proceeds similarly to case cinvoke in the proof of Lemma 52. We avoid repetition. □

Lemma 54 (At the initial state, the program counter capability pcc and the data capability ddc are prescribed by some capability object)**.**
$\forall t, s.\ t \vdash_i s \implies \exists (cc, dc, \_) \in \mathtt{range}(s.imp).\ \mathsf{pcc} \subseteq cc \wedge \mathsf{ddc} \subseteq dc$

Proof. Immediate by inversion using rules initial-state then exec-state. □

Claim 2 (At the initial state, the data and stack capabilities are disjoint)**.**
$\forall t, s.\ t \vdash_i s \implies s.\mathsf{stc} \cap s.\mathsf{ddc} = \emptyset$

Proof. Immediate by rules initial-state and exec-state. □

Claim 3 (Uniqueness of the initial state (Existence of at most one initial state for a given *TargetSetup*))**.**

$$\begin{aligned}
&\forall t : TargetSetup, funIDs.\\
&funIDs = [fid \mid fid \in \mathsf{dom}(offs) \wedge (_, _, offs) \in \mathsf{range}(t.imp)] \wedge\\
&\mathsf{all_distinct}(funIDs) \wedge\\
&\exists s, s'.\ t \vdash_i s \ \wedge\ t \vdash_i s'\\
&\implies s = s'
\end{aligned}$$

Proof. Follows from rules initial-state and exec-state. □

Lemma 55 (Preservation of the bounds of stack capabilities)**.**
$\forall s.\ s \to s' \implies (\forall mid, \sigma, e.\ s.\mathsf{mstc}(mid) = (\delta, \sigma, e, \_) \implies s'.\mathsf{mstc}(mid) = (\delta, \sigma, e, \_))$

Proof. We fix an arbitrary state s, assume the antecedent $s \to s'$ and consider all the possible cases for $s \to s'$:

1. **Case assign,**
2. **Case allocate,**
3. **Case jump1, and**
4. **Case jump0:**

   In all of these cases, we notice that $s.\mathsf{mstc} = s'.\mathsf{mstc}$, and so our goal follows by definition of equality on maps.

5. **Case cinvoke:**

   Here, we obtain the necessary precondition $s \succ_{\approx} s'$, from which by rule cinvoke-aux, we obtain the following necessary preconditions for some fixed mid:

   - $s.\mathsf{mstc}(mid) = (\delta, \sigma, e, off)$
   - $\mathsf{stc}' = (\delta, \sigma, e, off')$
   - $s'.\mathsf{mstc} = s.\mathsf{mstc}[mid \mapsto \mathsf{stc}']$

   Thus, we can show our goal for an arbitrary $mid' \in \mathsf{dom}(s.\mathsf{mstc})$ by case distinction on mid':

   - Case $mid' = mid$:
     In this case, our goal follows from $\mathsf{stc}'.\sigma = s.\mathsf{mstc}(mid).\sigma$ and $\mathsf{stc}'.e = s.\mathsf{mstc}(mid).e$.

- Case $mid' \neq mid$:

  In this case, the value in the s'.mstc map was not updated, so our goal follows from $s'.\mathsf{mstc}(mid') = s.\mathsf{mstc}(mid')$.

6. **Case creturn:**

   This case is similar to cinvoke.

□

1.4 Summary of target language features

Our model, **CHERIExp**, aims to model the essential security features provided by the CHERI hardware architecture and its runtime library, libcheri. In particular, call invocations between mutually distrustful components is a core feature of CHERI, which can be used to attain compartmentalized execution [3]. Passing parameters of function calls while ensuring non-retention of access to the stack frame of the callee after the call has returned is also a core feature of CHERI that we model in our language using the stack capability, and a restriction on storing the stack capability in memory (note that the rule assign categorically prohibits storing the stack capability in memory). In the actual CHERI architecture, these restrictions can be implemented using what is called the "permissions field" on capabilities. Here, we abstract a bit by modeling specific uses of this field rather than the field itself. Formal arguments showing that the permissions field can actually be used to attain our abstractions already exist in prior work [3, 4].

One limitation (to attacker strength) in our **CHERIExp** model is that the default data capability (ddc), and the stack capability (stc) are managed by the trusted call (cinvoke) and return (creturn) instructions, but there is no way to assign them directly. While in the actual CHERI architecture, only system-reserved registers are protected from arbitrary load operations [2], we still claim that our additional reservation on the root data and stack capability registers does not significantly weaken the attacker model. In particular, rather than being able to change the view of the memory by changing the values of ddc and stc, an attacker code that gets access to unlawful data-capabilities can still use them to load data from the unlawful memory region and store it in the region referenced by the current fixed ddc and stc. This way, it (the malicious code) can effectively change the view of the memory by copying the actual data rather than by directly installing the stolen data capabilities into the ddc or stc registers.

This built-in trust though (in how ddc and stc are managed) admittedly weakens the attacker model a bit because it enables for honest code the defense mechanism of checking the integrity of the data capabilities before executing sensitive code. So, subverting control flow attacks are allowed, but they are constrained in the sense that data capability registers are not arbitrarily loadable.

2 A source language (ImpMod) with pointers and modules

The source language of our transformation is a simple **imp**erative language **ImpMod** that features **mod**ules and functions with conditional goto statements. By design, **ImpMod** features protection of module-private variables.

2.1 Program and module representation, and well-formedness

A program in **ImpMod** consists of a list of modules. Each module consists of a list of function definitions, and a list of module-private variables. We skip the syntax of module and function definitions, and we directly represent them as structures (tuples of lists) that are output by the parser. We refer to the set of module identifiers as ModID, function identifiers as FunID, variable identifiers as VarID, and commands as Cmd. We give the syntax for commands and expressions later. We define the set of functions as $\mathit{FunDef} = \mathit{ModID} \times \mathit{FunID} \times \overline{\mathit{VarID}} \times \overline{\mathit{VarID}} \times \overline{\mathit{Cmd}}$ where a function specifies argument names `args`, local variable names `localIDs`, and a body (list of `commands`). Modules $\mathit{Mod} = \mathit{ModID} \times \overline{\mathit{VarID}} \times \overline{\mathit{FunDef}}$ where a module specifies a list of module-private variable names, and a list of function definitions. Programs $\mathit{Prog} = \overline{\mathit{Mod}}$ are lists of modules subject to the following **well-formedness conditions** (formally stated in fig. 4):

1. Module identifiers are unique across the program.
2. Function identifiers are unique across the program.
3. Programs are closed (i.e., the set of all function identifiers existing in a program contains all the function identifiers that are called by any command in the program).
4. The last command of every function is a Return.

We refer to the operation of linking two lists of modules $\overline{\mathit{mods_1}}$ and $\overline{\mathit{mods_2}}$ into one well-formed program P as $P = \overline{\mathit{mods_1}} \ltimes \overline{\mathit{mods_2}}$ where $\ltimes$ reorders and concatenates the two lists of modules only if they form a well-formed program P, and is not defined otherwise.

Definition 32 (Valid linking)**.** *Two programs (lists of modules) can be linked if there exists $\overline{m}$ where judgment $\overline{m_1} \ltimes \overline{m_2} = \lfloor \overline{m} \rfloor$ holds according to rule Valid-linking-src in Figure 7. If that is the case, then we sometimes write $\overline{m_1}[\overline{m_2}]$ for such $\overline{m}$.*

2.2 Values, expressions, and commands

Expressions $\mathcal{E} ::= \mathsf{addr}(\mathit{VarID}) \mid \mathsf{deref}(\mathcal{E}) \mid \mathcal{E} \oplus \mathcal{E} \mid \mathbb{Z} \mid \mathit{VarID} \mid \mathcal{E}[\mathcal{E}] \mid \mathsf{addr}(\mathcal{E}[\mathcal{E}])$ $\mid \mathsf{start}(\mathcal{E}) \mid \mathsf{end}(\mathcal{E}) \mid \mathsf{offset}(\mathcal{E}) \mid \mathsf{limRange}(\mathcal{E}, \mathcal{E}, \mathcal{E}) \mid \mathsf{capType}(\mathcal{E})$ in **ImpMod** manipulate integer values and a bounds-checked version of C pointers. Expressions allow reading and storing addresses of variables and they allow basic pointer arithmetic (addition) and by definition of the evaluation semantics, they allow only safe dereferencing. Evaluation of an expression that performs an unsafe memory dereference gets stuck. Values $\mathcal{V} = \mathbb{Z} \uplus (\{\delta, \kappa\} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z})$ are integers, or fat pointer values (i.e., values that represent the bounds and offset of a memory entity). The labels δ and κ on fat pointers indicate that the permissions available on the memory entity (the pointee) are **data** or **code** permissions respectively. The availability of code permissions still does not allow the source language semantics to execute this code; only code that is part of the program definition is executable (see Jump-zero, Jump-non-zero and Call). The ability to distinguish code pointers from data pointers though is important for defensive programming (and hence, for enhancing the expressiveness of the source programs as compared to the target ones, which is needed for proving that the translation between the two languages is fully abstract). Evaluation of expressions is given by the rules of the form $\mathcal{E}, \Sigma, \Delta, \beta, \mathit{MVar}, \mathit{Fd}, \mathit{Mem}, \Phi, \mathit{pc} \Downarrow \mathcal{V}$.

The syntax of commands is given by the grammar
$\mathit{Cmd} ::= \mathsf{Assign}\ \mathcal{E}_l\ \mathcal{E}_r \mid \mathsf{Alloc}\ \mathcal{E}_l\ \mathcal{E}_{size} \mid \mathsf{Call}\ \mathit{FunID}\ \overline{\mathcal{E}} \mid \mathsf{Return} \mid \mathsf{JumpIfZero}\ \mathcal{E}_c\ \mathcal{E}_{\mathit{off}} \mid \mathsf{Exit}$.

Figure 4: Well-formed programs of **ImpMod**

(Whole program)

$$\frac{\begin{array}{c}\texttt{wfp}(P) \\ \forall cmd.\ (cmd = \mathsf{Call}\ \mathit{fid}\ \_ \wedge \exists mod, fd.\ mod \in \overline{mods} \wedge fd \in \texttt{funDefs}(mod) \wedge \\ cmd \in \texttt{commands}(fd)) \implies \exists mod', fd'.\ mod' \in \overline{mods} \wedge fd' \in \texttt{funDefs}(mod') \wedge \mathit{fid} = \texttt{funID}(fd')\end{array}}{\texttt{whole}(P)}$$

(Well-formed program)

$$\frac{\begin{array}{c}P = \overline{mods} \qquad \forall mod \in \overline{mods}.\ MVar(mid) \cap \{\texttt{localIDs}(fd) \cup \texttt{args}(fd) \mid fd \in \texttt{funDefs}(mod)\} = \emptyset \\ \forall mod, mod' \in \overline{mods}. \\ \texttt{moduleID}(mod) = \texttt{moduleID}(mod') \implies mod = mod' \\ \forall mod, fd, mod', fd'.\ (mod, mod' \in \overline{mods} \wedge fd \in \texttt{funDefs}(mod) \wedge fd' \in \texttt{funDefs}(mod') \wedge \\ \texttt{funID}(fd) = \texttt{funID}(fd')) \implies (fd = fd' \wedge mod = mod')\end{array}}{\texttt{wfp}(P)}$$

(Well-formed program and parameters)

$$\frac{\begin{array}{c}\texttt{wfp}(\overline{mods}) \\ modIDs = \{modID \mid (modID, \_, \_) \in \overline{mods}\} \\ \forall mid, mid' \in modIDs.\ mid \neq mid' \implies \\ \Delta(mid) \cap \Delta(mid') = \emptyset \wedge K_{mod}(mid) \cap K_{mod}(mid') = \emptyset \wedge \Sigma(mid) \cap \Sigma(mid') = \emptyset \\ \bigcup \Delta(mid) \cap \bigcup \Sigma(mid) = \emptyset \\ (\bigcup \Delta(mid) \cup \bigcup \Sigma(mid)) \cap (-\infty, 0) = \emptyset \\ \mathsf{dom}(K_{mod}) = \mathsf{dom}(MVar) = \mathsf{dom}(\Sigma) = \mathsf{dom}(\Delta) = modIDs \\ Fd = \mathsf{fd\_map}(\overline{mods}) \qquad\qquad MVar = \mathsf{mvar}(\overline{mods}) \\ \mathsf{dom}(\beta) = \{(vid, \mathit{fid}, mid) \mid mid \in modIDs\ \wedge \\ (vid \in MVar(mid) \wedge \mathit{fid} = \bot \vee \mathit{fid} \in \mathsf{dom}(Fd) \wedge vid \in \texttt{localIDs}(Fd(\mathit{fid})) \cup \texttt{args}(Fd(\mathit{fid})))\} \\ \forall mid, \mathit{fid}, vid.\ vid \in \texttt{args}(Fd(\mathit{fid})) \wedge \beta(vid, \mathit{fid}, mid) = (s, e) \implies |s - e| = 1 \\ \forall \mathit{fid} \in \mathsf{dom}(Fd).\ \texttt{frameSize}(Fd(\mathit{fid})) \geq 0 \\ \forall mid \in modIDs, \mathit{fid} \in \mathsf{dom}(Fd).\ \biguplus_{vid \in \texttt{localIDs}(Fd(\mathit{fid})) \cup \texttt{args}(Fd(\mathit{fid}))} \beta(vid, \mathit{fid}, mid) = [-\texttt{frameSize}(Fd(\mathit{fid})), 0) \\ \forall mid \in modIDs.\ \biguplus_{vid \in MVar(mid)} \beta(vid, \bot, mid) = [0, \Delta(mid).2 - \Delta(mid).1) \\ \forall mid \in modIDs, \mathit{fid} \in \mathsf{dom}(Fd).\ |K_{fun}(\mathit{fid})| = |\texttt{commands}(Fd(\mathit{fid}))| \\ \forall mid \in modIDs.\ \biguplus_{\mathit{fid}\ \in\ \{\mathit{fid}\ \mid\ \texttt{moduleID}(Fd(\mathit{fid}))\ =\ mid\}} K_{fun}(\mathit{fid}) = [0, |K_{mod}(mid)|)\end{array}}{\texttt{wfp\_params}(\overline{mods}, \Delta, \Sigma, \beta, K_{mod}, K_{fun})}$$

2.3 Program state

A program state $\langle \mathit{Mem}, \mathit{stk}, \mathit{pc}, \Phi, \mathit{nalloc} \rangle$ whose type is denoted by $\mathit{SourceState}$ consists of:

- a **data memory** $\mathit{Mem} : \mathbb{Z} \xrightarrow{\text{fin}} \mathcal{V}$ which is a map from addresses $\mathbb{Z}$ to values $\mathcal{V}$.
- a **call stack** $\mathit{stk} : \overline{\mathit{FunID} \times \mathbb{N}}$ which is a **list of program counters** that record the function calls history (see pc below),
- $\Phi : \mathit{ModID} \to \mathbb{Z}$ which maintains **for every module a pointer to its top-most stack frame**,
- a program counter $\mathit{pc} : \mathit{FunID} \times \mathbb{N}$ modeling the **index of the executing command** within the list of commands of the current function. We define $\texttt{inc}((\mathit{funId}, n)) \stackrel{\text{def}}{=} (\mathit{funId}, n + 1)$.
- and an allocation status $\mathit{nalloc} : \mathbb{Z}$ which simply represents the **first** (in descending order) **free memory address** (i.e., the first address that was never allocated before).

A program evaluation context $\Sigma; \Delta; \beta; \mathit{MVar}; \mathit{Fd}$ consists of:

- $\Sigma : \mathit{ModID} \to \mathbb{Z}^2$ which maintains **for every module the start and end addresses of its stack region**. Recall that each module in **ImpMod** has its own stack which stores the local variables when this module is callee. Notice that return pointers on the other hand are stored on the trusted stack stk rather than on a module's own stack. The latter only stores arguments and local variables,
- $\Delta : \mathit{ModID} \to \mathbb{Z}^2$ which maps each **module to a range of addresses representing the data segment** in which the static data of the module lives. Offsets from β are added to the first component of the range that is output by this map in order to compute the location in memory of module-global variables.
- $\beta : (\mathit{VarID} \times (\mathit{FunID} \uplus \bot) \times \mathit{ModID}) \to \mathbb{Z}^2$ which maps each **variable identifier to bounds** that represent the offsets within the data segment or the stack frame to which the (module-global or function-local) variable is mapped,
- an immutable map $\mathit{MVar} : \mathit{ModID} \to \overline{\mathit{VarID}}$ of module IDs to module-private variable identifiers,
- and an immutable map $\mathit{Fd} : \mathit{FunID} \to \mathit{FunDef}$ of function identifiers to function definitions.

The following are useful representations of a program:

Definition 33 (Set of function definitions of a list of modules)**.**

$$\mathsf{fun\_defs}(\overline{\mathit{mods}}) \stackrel{\text{def}}{=} \left\{ \mathit{mdef} \;\middle|\; \mathit{mdef} \in \mathit{mdefs} \wedge (\_, \_, \mathit{mdefs}) \in \overline{\mathit{mods}} \right\}$$

Definition 34 (Function ID to function definition map)**.**

$$\mathsf{fd\_map}(\overline{\mathit{mods}}) \stackrel{\text{def}}{=} \left\{ \mathit{fid} \mapsto \mathit{fdef} \;\middle|\; \mathit{fdef} \in \mathsf{fun\_defs}(\overline{\mathit{mods}}) \wedge \mathit{fdef} = (\_, \mathit{fid}, \_, \_, \_) \right\}$$

Definition 35 (Module variables map)**.**

$$\mathsf{mvar}(\overline{\mathit{mods}}) \stackrel{\text{def}}{=} \left\{ \mathit{mid} \mapsto \overline{\mathit{vids}} \;\middle|\; (\mathit{mid}, \overline{\mathit{vids}}, \_) \in \overline{\mathit{mods}} \right\}$$

The semantics of expressions and commands are given in fig. 5 and fig. 6.

Figure 5: Evaluation of expressions $\mathcal{E}$ in **ImpMod**

(Evaluate-expr-const)

$$\frac{z \in \mathbb{Z}}{z, \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow z}$$

(Evaluate-expr-cast-to-integer-start)

$$\frac{e, \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow (\_, z, \_, \_)}{\mathsf{start}(e), \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow z}$$

(Evaluate-expr-cast-to-integer-end)

$$\frac{e, \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow (\_, \_, z, \_)}{\mathsf{end}(e), \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow z}$$

(Evaluate-expr-cast-to-integer-offset)

$$\frac{e, \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow (\_, \_, \_, z)}{\mathsf{offset}(e), \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow z}$$

(Evaluate-expr-cap-type)

$$\frac{\begin{array}{c} e, \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow x \\ x \in \mathbb{Z} \implies v = 0 \quad x \in \{\kappa\} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \implies v = 1 \quad x \in \{\delta\} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \implies v = 2 \end{array}}{\mathsf{capType}(e), \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow v}$$

(Evaluate-expr-binop)

$$\frac{\begin{array}{c} e_1, \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow z_1 \quad z_1 \in \mathbb{Z} \\ e_2, \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow z_2 \quad z_2 \in \mathbb{Z} \\ z_r = z_1 [\oplus] z_2 \end{array}}{e_1 \oplus e_2, \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow z_r}$$

(Evaluate-expr-addr-local)

$$\frac{\begin{array}{c} (fid, \_) = pc \quad vid \in \mathtt{localIDs}(Fd(fid)) \cup \mathtt{args}(Fd(fid)) \quad mid = \mathtt{moduleID}(Fd(fid)) \\ \beta(vid, fid, mid) = [s, e) \quad \phi = \Sigma(mid).1 + \Phi(mid) \end{array}}{\mathsf{addr}(vid), \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow (\delta, \phi + s, \phi + e, 0)}$$

(Evaluate-expr-addr-module)

$$\frac{\begin{array}{c} (fid, \_) = pc \quad vid \notin \mathtt{localIDs}(Fd(fid)) \cup \mathtt{args}(Fd(fid)) \quad mid = \mathtt{moduleID}(Fd(fid)) \\ vid \in MVar(mid) \quad \beta(vid, \bot, mid) = [s, e) \end{array}}{\mathsf{addr}(vid), \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow (\delta, \Delta(mid).1 + s, \Delta(mid).1 + e, 0)}$$

(Evaluate-expr-var)

$$\frac{\begin{array}{c} vid \in VarID \quad \mathsf{addr}(vid), \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow (\delta, s, e, \mathit{off}) \\ s \leq s + \mathit{off} < e \quad Mem(s + \mathit{off}) = v \end{array}}{vid, \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow v}$$

(Evaluate-expr-addr-arr)

$$\frac{\begin{array}{c} \mathsf{addr}(e_{arr}), MVar, \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow (\delta, s, e, \mathit{off}) \\ e_{idx}, \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow \mathit{off}' \\ \mathit{off}' \in \mathbb{Z} \end{array}}{\mathsf{addr}(e_{arr}[e_{idx}]), \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow (\delta, s, e, \mathit{off} + \mathit{off}')}$$

(Evaluate-expr-arr)

$$\frac{\begin{array}{c} \mathsf{addr}(e_{arr}[e_{idx}]), \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow (\delta, s, e, \mathit{off}) \\ s \leq s + \mathit{off} < e \quad Mem(s + \mathit{off}) = v \end{array}}{e_{arr}[e_{idx}], \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow v}$$

(Evaluate-expr-deref)

$$\frac{\begin{array}{c} e, \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow (\delta, s, e, \mathit{off}) \\ s \leq s + \mathit{off} < e \quad Mem(s + \mathit{off}) = v \end{array}}{\mathsf{deref}(e), \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow v}$$

(Evaluate-expr-limrange)

$$\frac{\begin{array}{c} e, \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow (x, s, e, \_) \\ e_s, \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow s' \quad s' \in \mathbb{Z} \\ e_e, \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow e' \quad e' \in \mathbb{Z} \\ [s', e') \subseteq [s, e) \end{array}}{\mathsf{limRange}(e, e_s, e_e), \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow (x, s', e', 0)}$$

Figure 6: Evaluation of commands Cmd in **ImpMod**

(Assign-to-var-or-arr)

$$\frac{\begin{array}{c}(\mathit{fid}, n) = \mathit{pc} \qquad \texttt{commands}(\mathit{Fd}(\mathit{fid}))(n) = \mathsf{Assign}\ e_l\ e_r \\ e_l, \Sigma, \Delta, \beta, \mathit{MVar}, \mathit{Fd}, \mathit{Mem}, \Phi, \mathit{pc} \Downarrow (\delta, s, e, \mathit{off}) \qquad e_r, \Sigma, \Delta, \beta, \mathit{MVar}, \mathit{Fd}, \mathit{Mem}, \Phi, \mathit{pc} \Downarrow v \\ \mathit{modID} = \texttt{moduleID}(\mathit{Fd}(\mathit{fid})) \qquad \forall s', e'.\ v = (\delta, s', e', \_) \implies ([s', e') \cap \Sigma(\mathit{modID}) = \emptyset \lor [s, e) \subseteq \Sigma(\mathit{modID})) \\ s \leq s + \mathit{off} < e \qquad \mathit{Mem}' = \mathit{Mem}[s + \mathit{off} \mapsto v]\end{array}}{\Sigma; \Delta; \beta; \mathit{MVar}; \mathit{Fd} \vdash \langle \mathit{Mem}, \mathit{stk}, \mathit{pc}, \Phi, \mathit{nalloc}\rangle \to \langle \mathit{Mem}', \mathit{stk}, \texttt{inc}(\mathit{pc}), \Phi, \mathit{nalloc}\rangle}$$

(Allocate)

$$\frac{\begin{array}{c}(\mathit{fid}, n) = \mathit{pc} \qquad \texttt{commands}(\mathit{Fd}(\mathit{fid}))(n) = \mathsf{Alloc}\ e_l\ e_{size} \qquad e_l, \Sigma, \Delta, \beta, \mathit{MVar}, \mathit{Fd}, \mathit{Mem}, \Phi, \mathit{pc} \Downarrow (\delta, s, e, \mathit{off}) \\ e_{size}, \Sigma, \Delta, \beta, \mathit{MVar}, \mathit{Fd}, \mathit{Mem}, \Phi, \mathit{pc} \Downarrow v \qquad s \leq s + \mathit{off} < e \qquad v \in \mathbb{Z}^+ \qquad \mathit{nalloc} - v > \nabla \\ \mathit{nalloc}' = \mathit{nalloc} - v \qquad \mathit{Mem}' = \mathit{Mem}[s + \mathit{off} \mapsto (\delta, \mathit{nalloc}', \mathit{nalloc}, 0)][a \mapsto 0 \mid a \in [\mathit{nalloc}', \mathit{nalloc})]\end{array}}{\Sigma; \Delta; \beta; \mathit{MVar}; \mathit{Fd} \vdash \langle \mathit{Mem}, \mathit{stk}, \mathit{pc}, \Phi, \mathit{nalloc}\rangle \to \langle \mathit{Mem}', \mathit{stk}, \texttt{inc}(\mathit{pc}), \Phi, \mathit{nalloc}'\rangle}$$

(Call)

$$\frac{\begin{array}{c}(\mathit{fid}, n) = \mathit{pc} \qquad \texttt{commands}(\mathit{Fd}(\mathit{fid}))(n) = \mathsf{Call}\ \mathit{fid}_{call}\ \overline{e} \\ \mathit{modID} = \texttt{moduleID}(\mathit{Fd}(\mathit{fid}_{call})) \qquad \mathit{argNames} = \texttt{args}(\mathit{Fd}(\mathit{fid}_{call})) \qquad \mathit{localIDs} = \texttt{localIDs}(\mathit{Fd}(\mathit{fid}_{call})) \\ \mathit{nArgs} = \texttt{length}(\mathit{argNames}) = \texttt{length}(\overline{e}) \qquad \mathit{nLocal} = \texttt{length}(\mathit{localIDs}) \qquad \mathit{frameSize} = \texttt{frameSize}(\mathit{Fd}(\mathit{fid}_{call})) \\ \mathit{curFrameSize} = \texttt{frameSize}(\mathit{Fd}(\mathit{fid})) \qquad \mathit{curModID} = \texttt{moduleID}(\mathit{Fd}(\mathit{fid})) \\ \Sigma(\mathit{modID}).1 + \Phi(\mathit{modID}) + \mathit{frameSize} < \Sigma(\mathit{modID}).2 \qquad \Phi' = \Phi[\mathit{modID} \mapsto \Phi(\mathit{modID}) + \mathit{frameSize}] \\ \phi' = \Sigma(\mathit{modID}).1 + \Phi'(\mathit{modID}) \\ \overline{e}(i), \Sigma, \Delta, \beta, \mathit{MVar}, \mathit{Fd}, \mathit{Mem}, \Phi, \mathit{pc} \Downarrow v_i\ \forall i \in [0, \mathit{nArgs}) \\ \forall i \in [0, \mathit{nArgs}), s', e'.\ v_i = (\delta, s', e', \_) \implies [s', e') \cap \Sigma(\mathit{curModID}) = \emptyset \\ \mathit{stk}' = \texttt{push}(\mathit{stk}, \mathit{pc}) \qquad \mathit{pc}' = (\mathit{fid}_{call}, 0) \\ \mathit{Mem}' = \mathit{Mem}[\phi' + s_i \mapsto v_i \mid s_i \in \beta(\mathit{argNames}(i), \mathit{fid}_{call}, \mathit{modID}) \land i \in [0, \mathit{nArgs})] \\ [\phi' + s_i \mapsto 0 \mid s_i \in \beta(\mathit{localIDs}(i), \mathit{fid}_{call}, \mathit{modID}) \land i \in [0, \mathit{nLocal})]\end{array}}{\Sigma; \Delta; \beta; \mathit{MVar}; \mathit{Fd} \vdash \langle \mathit{Mem}, \mathit{stk}, \mathit{pc}, \Phi, \mathit{nalloc}\rangle \to \langle \mathit{Mem}', \mathit{stk}', \mathit{pc}', \Phi', \mathit{nalloc}\rangle}$$

(Return)

$$\frac{\begin{array}{c}(\mathit{fid}, n) = \mathit{pc} \qquad \texttt{commands}(\mathit{Fd}(\mathit{fid}))(n) = \mathsf{Return} \qquad (\mathit{pc}', \mathit{stk}') = \texttt{pop}(\mathit{stk}) \qquad \mathit{pc}' = (\mathit{fid}', \_) \\ \mathit{curFrameSize} = \texttt{frameSize}(\mathit{Fd}(\mathit{fid})) \qquad \mathit{curModID} = \texttt{moduleID}(\mathit{Fd}(\mathit{fid})) \\ \Phi' = \Phi[\mathit{curModID} \mapsto \Phi(\mathit{curModID}) - \mathit{curFrameSize}]\end{array}}{\Sigma; \Delta; \beta; \mathit{MVar}; \mathit{Fd} \vdash \langle \mathit{Mem}, \mathit{stk}, \mathit{pc}, \Phi, \mathit{nalloc}\rangle \to \langle \mathit{Mem}, \mathit{stk}', \texttt{inc}(\mathit{pc}'), \Phi', \mathit{nalloc}\rangle}$$

(Jump-non-zero)

$$\frac{(\mathit{fid}, n) = \mathit{pc} \qquad \texttt{commands}(\mathit{Fd}(\mathit{fid}))(n) = \mathsf{JumpIfZero}\ e_c\ e_{\mathit{off}} \qquad e_c, \Sigma, \Delta, \beta, \mathit{MVar}, \mathit{Fd}, \mathit{Mem}, \Phi, \mathit{pc} \Downarrow v \qquad v \neq 0}{\Sigma; \Delta; \beta; \mathit{MVar}; \mathit{Fd} \vdash \langle \mathit{Mem}, \mathit{stk}, \mathit{pc}, \Phi, \mathit{nalloc}\rangle \to \langle \mathit{Mem}, \mathit{stk}, \texttt{inc}(\mathit{pc}), \Phi, \mathit{nalloc}\rangle}$$

(Jump-zero)

$$\frac{\begin{array}{c}(\mathit{fid}, n) = \mathit{pc} \qquad \texttt{commands}(\mathit{Fd}(\mathit{fid}))(n) = \mathsf{JumpIfZero}\ e_c\ e_{\mathit{off}} \qquad e_c, \Sigma, \Delta, \beta, \mathit{MVar}, \mathit{Fd}, \mathit{Mem}, \Phi, \mathit{pc} \Downarrow v \\ e_{\mathit{off}}, \Sigma, \Delta, \beta, \mathit{MVar}, \mathit{Fd}, \mathit{Mem}, \Phi, \mathit{pc} \Downarrow \mathit{off} \qquad \mathit{off} \in \mathbb{Z} \qquad v = 0\end{array}}{\Sigma; \Delta; \beta; \mathit{MVar}; \mathit{Fd} \vdash \langle \mathit{Mem}, \mathit{stk}, \mathit{pc}, \Phi, \mathit{nalloc}\rangle \to \langle \mathit{Mem}, \mathit{stk}, (\mathit{fid}, n + \mathit{off}), \Phi, \mathit{nalloc}\rangle}$$

(Exit)

$$\frac{(\mathit{fid}, n) = \mathit{pc} \qquad \texttt{commands}(\mathit{Fd}(\mathit{fid}))(n) = \mathsf{Exit}}{\Sigma; \Delta; \beta; \mathit{MVar}; \mathit{Fd} \vdash \langle \mathit{Mem}, \mathit{stk}, \mathit{pc}, \Phi, \mathit{nalloc}\rangle \to \langle \mathit{Mem}, \mathit{stk}, \mathit{pc}, \Phi, \mathit{nalloc}\rangle}$$

2.4 Initial, terminal and execution states

Definition 36 (Valid execution state of a program).
A state $\langle Mem, stk, pc, \Phi, nalloc\rangle$ *is a* ***valid execution state*** *of a program* $\overline{mods}$ *if it satisfies the judgment* $\overline{mods}; \Sigma; \Delta; \beta; MVar; Fd \vdash_{exec} \langle Mem, stk, pc, \Phi, nalloc\rangle$ *defined in rule Exec-state-src of Figure 7.*

Definition 37 (Initial state).
An ***initial state*** *of a program* $\overline{mods}$ *is any state* $\langle Mem, stk, pc, \Phi, nalloc\rangle$ *satisfying* $\overline{mods}; \Sigma; \Delta; \beta; MVar; Fd \vdash_{i} \langle Mem, stk, pc, \Phi, nalloc\rangle$ *which is defined in rule Initial-state-src in Figure 7.*

Definition 38 (Initial state function).

$$\mathsf{initial\_state}(\overline{m}, \Delta, \Sigma, mainModID) \stackrel{\mathsf{def}}{=}$$
$$\langle$$
$$\{a \mapsto 0 \mid a \in \bigcup_{m \in \overline{m}} \Delta(m.mid) \cup \Sigma(m.mid)\},$$
$$\mathtt{nil},$$
$$(\mathtt{main}, 0),$$
$$\{mainModID \mapsto \mathtt{frameSize}(\overline{m}(mainModID).\overline{fds}(\mathtt{main}))\} \cup \bigcup_{mid \in \{m.mid \mid m \in \overline{m}\} \setminus \{mainModID\}} \{mid \mapsto 0\},$$
$$-1$$
$$\rangle$$

Definition 39 (Main module).

$$\mathsf{main\_module}(\overline{m}) = mid \iff \exists m, fd.\ m \in \overline{m} \ \wedge\ fd \in m.\overline{fds} \ \wedge\ \mathtt{main} = \mathtt{funID}(fd) \ \wedge\ \mathtt{moduleID}(m) = mid$$

Claim 4 (The function $\mathsf{initial\_state}$ and the judgment $\vdash_i$ are compatible).

$$\begin{aligned}
&\forall K_{mod}; K_{fun}; \overline{m}, \Sigma, \Delta, \beta\\
&\mathsf{main_module}(\overline{m}) = mainModuleID\ \wedge\\
&\mathtt{wfp_params}(\overline{m}, \Delta, \Sigma, \beta, K_{mod}, K_{fun})\ \wedge\\
&\mathsf{initial_state}(\overline{m}, \Delta, \Sigma, mainModuleID) = s_i\\
&\implies\\
&\exists MVar, Fd.\ K_{mod}; K_{fun}; \overline{m}; \Sigma; \Delta; \beta; MVar; Fd \vdash_i s_i
\end{aligned}$$

Definition 40 (Terminal state).
A ***terminal state*** *of a program* $\overline{mods}$ *is any state* $\langle Mem, stk, pc, \Phi, nalloc\rangle$ *satisfying* $\mathsf{fd\_map}(\overline{mods}) \vdash_t \langle Mem, stk, pc, \Phi, nalloc\rangle$ *which is defined in rule Terminal-state-src-exit in Figure 7.*

We now define convergence of a program $\overline{m_1}$ running with a context $\mathbb{C}$ as successful linking, successful loading, and reachability of a terminal state from every loadable initial state.

Definition 41 (Layout places $\overline{m_1}$ before $\mathbb{C}$).

$$\begin{aligned}
&\overline{m_1} \rhd_{L_1, L_2} \mathbb{C} \stackrel{\mathsf{def}}{=}\\
&\max_{mod \in \overline{m_1}} \{L_1(\mathtt{moduleID}(mod)).2\} \cup \{L_2(\mathtt{moduleID}(mod)).2\} <\\
&\min_{mod \in \mathbb{C}} \{L_1(\mathtt{moduleID}(mod)).1\} \cup \{L_2(\mathtt{moduleID}(mod)).1\}
\end{aligned}$$

Figure 7: Valid execution and initial states in **ImpMod**

(Valid-linking-src)

$$\frac{\overline{m} = \overline{m_1} \uplus \overline{m_2} \qquad \mathtt{wfp}(\overline{m})}{\overline{m_1} \ltimes \overline{m_2} = \lfloor \overline{m} \rfloor}$$

(Equal-interfaces-src)

$$\frac{\begin{array}{c} \mathit{modIDs} = \{\mathit{mid} \mid (\mathit{mid}, \_, \_) \in \overline{m_1}\} = \{\mathit{mid} \mid (\mathit{mid}, \_, \_) \in \overline{m_2}\} \\ \mathit{fDefs_1} = \{\mathit{fdef} \mid \mathit{fdef} \in \mathit{fdefs} \ \wedge \ (\_, \_, \mathit{fdefs}) \in \overline{m_1}\} \\ \mathit{fDefs_2} = \{\mathit{fdef} \mid \mathit{fdef} \in \mathit{fdefs} \ \wedge \ (\_, \_, \mathit{fdefs}) \in \overline{m_2}\} \\ \mathit{fSigs_1} = \{(\mathit{mid}, \mathit{fid}, \mathit{nArgs}) \mid \mathit{fd} \in \mathit{fDefs_1} \ \wedge \ \mathit{mid} = \mathtt{moduleID}(\mathit{fd}) \ \wedge \ \mathit{fid} = \mathtt{funID}(\mathit{fd}) \ \wedge \ \mathit{nArgs} = |\mathtt{args}(\mathit{fd})|\} \\ \mathit{fSigs_2} = \{(\mathit{mid}, \mathit{fid}, \mathit{nArgs}) \mid \mathit{fd} \in \mathit{fDefs_2} \ \wedge \ \mathit{mid} = \mathtt{moduleID}(\mathit{fd}) \ \wedge \ \mathit{fid} = \mathtt{funID}(\mathit{fd}) \ \wedge \ \mathit{nArgs} = |\mathtt{args}(\mathit{fd})|\} \\ \mathit{fSigs_1} = \mathit{fSigs_2} \end{array}}{\overline{m_1} \frown \overline{m_2}}$$

(Exec-state-src)

$$\frac{\begin{array}{c} \mathtt{wfp\_params}(\overline{\mathit{mods}}, \Delta, \Sigma, \beta, K_{mod}, K_{fun}) \\ \mathit{modIDs} = \{\mathit{modID} \mid (\mathit{modID}, \_, \_) \in \overline{\mathit{mods}}\} \\ \mathsf{dom}(K_{mod}) = \mathsf{dom}(\mathit{MVar}) = \mathsf{dom}(\Sigma) = \mathsf{dom}(\Delta) = \mathit{modIDs} \\ \mathit{Fd} = \mathsf{fd\_map}(\overline{\mathit{mods}}) \qquad \mathit{MVar} = \mathsf{mvar}(\overline{\mathit{mods}}) \\ \mathit{pc} = (\mathit{funID}, \_) \ \wedge \ \mathit{funID} \in \mathsf{dom}(\mathit{Fd}) \\ \forall (\mathit{fid}, \_) \in \mathtt{elems}(\mathit{stk}).\ \mathit{fid} \in \mathsf{dom}(\mathit{Fd}) \\ \mathsf{static\_addresses}(\Sigma, \Delta, \mathit{modIDs}) \subseteq \mathsf{dom}(\mathit{Mem}) \\ \nabla < -1 \implies (\mathit{nalloc} > \nabla \ \wedge \\ \forall a \in \mathsf{dom}(\mathit{Mem}).\ a > \nabla \ \wedge \\ \forall a, s, e, v.\ v \in \mathsf{range}(\mathit{Mem}) \ \wedge \ v = (\delta, s, e, \_) \ \wedge \ a \in [s, e) \implies a > \nabla) \\ \forall \mathit{mid} \in \mathit{modIDs}.\ \Phi(\mathit{mid}) = \\ \sum\limits_{\mathit{fid} \ \in \ \{\mathit{fid} \ \mid \ \mathtt{moduleID}(\mathit{Fd}(\mathit{fid})) \ = \ \mathit{mid}\}} \mathtt{frameSize}(\mathit{Fd}(\mathit{fid})) \times (\mathtt{countIn}((\mathit{fid}, \_), \mathit{stk}) + (\mathit{pc} = (\mathit{fid}, \_) \ ? \ 1 \ : \ 0)) \\ \forall \mathit{mid} \in \mathit{modIDs}.\ \Sigma(\mathit{mid}).1 + \Phi(\mathit{mid}) \leq \Sigma(\mathit{mid}).2 \\ \mathit{stk} = \mathtt{nil} \implies \mathit{pc}.\mathit{fid} = \mathtt{main} \\ \mathit{stk} \neq \mathtt{nil} \implies \mathit{stk}(0).\mathit{fid} = \mathtt{main} \\ \forall \mathit{mid}, a, \sigma, e.\ \mathit{Mem}(a) = (\delta, \sigma, e, \_) \ \wedge \ [\sigma, e) \cap \Sigma(\mathit{mid}) \neq \emptyset \implies a \in \Sigma(\mathit{mid}) \\ \mathit{nalloc} < 0 \end{array}}{K_{mod}; K_{fun}; \overline{\mathit{mods}}; \Sigma; \Delta; \beta; \mathit{MVar}; \mathit{Fd} \vdash_{exec} \langle \mathit{Mem}, \mathit{stk}, \mathit{pc}, \Phi, \mathit{nalloc} \rangle}$$

(Initial-state-src)

$$\frac{\begin{array}{c} K_{mod}; K_{fun}; \overline{\mathit{mods}}; \Sigma; \Delta; \beta; \mathit{MVar}; \mathit{Fd} \vdash_{exec} \langle \mathit{Mem}, \mathit{stk}, \mathit{pc}, \Phi, \mathit{nalloc} \rangle \\ \mathit{nalloc} = -1 \qquad \mathit{stk} = \mathtt{nil} \qquad \mathit{Mem} = \{a \mapsto 0 \mid a \in \bigcup\limits_{\mathit{mid} \in \mathsf{dom}(\Delta)} \Delta(\mathit{mid}) \cup \Sigma(\mathit{mid})\} \\ \mathit{pc} = (\mathtt{main}, 0) \\ \Phi = \{\mathtt{moduleID}(\mathit{Fd}(\mathtt{main})) \mapsto \mathtt{frameSize}(\mathit{Fd}(\mathtt{main})\} \cup \bigcup\limits_{\mathit{mid} \in \mathsf{dom}(\Delta) \setminus \{\mathtt{moduleID}(\mathit{Fd}(\mathtt{main}))\}} \{\mathit{mid} \mapsto 0\} \end{array}}{K_{mod}; K_{fun}; \overline{\mathit{mods}}; \Sigma; \Delta; \beta; \mathit{MVar}; \mathit{Fd} \vdash_{i} \langle \mathit{Mem}, \mathit{stk}, \mathit{pc}, \Phi, \mathit{nalloc} \rangle}$$

(Terminal-state-src-exit)

$$\frac{\mathit{pc} = (\mathit{fid}, n) \qquad \mathtt{commands}(\mathit{Fd}(\mathit{fid}))(n) = \mathsf{Exit}}{\mathit{Fd} \vdash_t \langle \mathit{Mem}, \mathit{stk}, \mathit{pc}, \Phi, \mathit{nalloc} \rangle}$$

Definition 42 (Layout-ordered linking)**.**

$$\mathbb{C}[\overline{m_1}]_{\Delta,\Sigma} = \overline{m} \iff \mathbb{C} \ltimes \overline{m_1} = \lfloor \overline{m} \rfloor \ \wedge\ \overline{m_1} \triangleright_{\Delta,\Sigma} \mathbb{C}$$

Definition 43 (Linkability, loadability, and convergence of execution in the source language)**.**

$$\begin{aligned}
&\Sigma, \Delta, \beta, \nabla \vdash \mathbb{C}[\overline{m_1}] \Downarrow \stackrel{\mathsf{def}}{=} \\
&\qquad \exists \overline{m}.\ \mathbb{C}[\overline{m_1}]_{\Delta,\Sigma} = \overline{m}\ \wedge \\
&\qquad \exists s_t.\ \Sigma; \Delta; \beta; \mathsf{mvar}(\overline{m}); \mathsf{fd_map}(\overline{m}) \vdash \mathsf{initial_state}(\overline{m}, \Delta, \Sigma, \mathsf{main_module}(\overline{m})) \to^*_{\nabla} s_t\ \wedge \\
&\qquad \mathsf{fd_map}(\overline{m}) \vdash_t s_t
\end{aligned}$$

where $\to^*$ *is the reflexive transitive closure of the evaluation relation defined in fig. 6.*

Definition 44 (Addition of an offset ω to the data segment's bounds)**.**

$$\Delta + \omega \stackrel{\mathsf{def}}{=} \{mid \mapsto \Delta(mid) + \omega \mid mid \in \mathsf{dom}(\Delta)\}$$

where $\Delta(mid) + \omega$ *is the addition of a constant to an interval which is given by* $[a, b) + c \stackrel{\mathsf{def}}{=} [a + c, b + c)$.

Two programs $\overline{m_1}$ and $\overline{m_2}$ that have the same per-module data-segment size $\tilde{\Delta}$ and that have respectively data segment layouts β_1 and β_2 are said to be contextually equivalent in the execution environment $\tilde{\Sigma}, \nabla$ denoted
$\tilde{\Delta}, \beta_1, \overline{m_1} \simeq_{\tilde{\Sigma},\nabla} \tilde{\Delta}, \beta_2, \overline{m_2}$ when they are equi-linkable, equi-loadable, and equi-convergent in all contexts $\mathbb{C}$ with an arbitrary data segment size Δ, data segment layout β, stack sizes Σ.

Definition 45 (Source contextual equivalence)**.**

$$\begin{aligned}
&\tilde{\Delta}, \beta_1, \overline{m_1} \simeq_{\tilde{\Sigma},\omega,\nabla} \tilde{\Delta}, \beta_2, \overline{m_2} \stackrel{\mathsf{def}}{=} \\
&\forall \Delta, \beta, \Sigma, \mathbb{C}. \\
&\mathtt{wfp}(\mathbb{C}) \implies \\
&(\Sigma \uplus \tilde{\Sigma}, (\Delta \uplus \tilde{\Delta}) + \omega, \beta \uplus \beta_1, \nabla \ \vdash\ \mathbb{C}[\overline{m_1}] \Downarrow \iff \\
&\Sigma \uplus \tilde{\Sigma}, (\Delta \uplus \tilde{\Delta}) + \omega, \beta \uplus \beta_2, \nabla \ \vdash\ \mathbb{C}[\overline{m_2}] \Downarrow)
\end{aligned}$$

Lemma 56 (Preservation of $\vdash_{exec}$)**.**

$$\begin{aligned}
&K_{mod}; K_{fun}; \overline{mods}; \Sigma; \Delta; \beta; MVar; Fd \vdash_{exec} \langle Mem, stk, pc, \Phi, nalloc\rangle\ \wedge \\
&\Sigma; \Delta; \beta; MVar; Fd \vdash \langle Mem, stk, pc, \Phi, nalloc\rangle \to \langle Mem', stk', pc', \Phi', nalloc'\rangle \\
&\implies \\
&\overline{mods}; \Sigma; \Delta; \beta; MVar; Fd \vdash_{exec} \langle Mem', stk', pc', \Phi', nalloc'\rangle
\end{aligned}$$

Proof. By inversion using rules Exec-state-src and Well-formed program and parameters, we obtain the following assumptions:

Well formed program and parameters $\mathtt{wfp\_params}(\overline{mods}, \Delta, \Sigma, \beta, K_{mod}, K_{fun})$

Module IDs $modIDs = \{modID \mid (modID, \_, \_) \in \overline{mods}\}$

Equal domains $\mathsf{dom}(K_{mod}) = \mathsf{dom}(MVar) = \mathsf{dom}(\Sigma) = \mathsf{dom}(\Delta) = modIDs$

Function definitions
$funDefs = \{modFunDef \mid modFunDef \in modFunDefs \wedge (\_, \_, modFunDefs) \in \overline{mods}\}$

Fd $Fd = \{funID \mapsto funDef \mid funDef \in funDefs \wedge funDef = (\_, funID, \_, \_, \_)\}$

MVar $MVar = \{modID \mapsto \overline{varIDs} \mid (modID, \overline{varIDs}, \_) \in \overline{mods}\}$

pc points to an existing function $pc = (funID, \_) \ \wedge \ funID \in \mathsf{dom}(Fd)$

All pc's on stack point to existing functions

$$\forall (fid, \_) \in \texttt{elems}(stk).\ fid \in \mathsf{dom}(Fd)$$

dom(β)

$$\mathsf{dom}(\beta) = \{(vid, fid, mid) \mid mid \in modIDs \wedge \\ (vid \in MVar(mid) \wedge fid = \bot \vee fid \in \mathsf{dom}(Fd) \wedge vid \in \texttt{localIDs}(Fd(fid)) \cup \texttt{args}(Fd(fid)))\}$$

Arguments are non-arrays

$$\forall mid, fid, vid.\ vid \in \texttt{args}(Fd(fid)) \wedge \beta(vid, fid, mid) = (s, e) \implies |s - e| = 1$$

Static addresses are mapped addresses

$$\mathsf{static\_addresses}(\Sigma, \Delta, modIDs) \subseteq \mathsf{dom}(Mem)$$

No address exists that is out-of-memory

$$\nabla < 0 \implies (nalloc > \nabla \ \wedge \\ \forall a \in \mathsf{dom}(Mem).\ a > \nabla \ \wedge \\ \forall a, s, e, v.\ v \in \mathsf{range}(Mem) \ \wedge \ v = (\delta, s, e, \_) \ \wedge \ a \in [s, e) \implies a > \nabla)$$

No stack overflow

$$\forall mid \in modIDs.\ \Sigma(mid).1 + \Phi(mid) \leq \Sigma(mid).2$$

Frame sizes are non-negative

$$\forall fid \in \mathsf{dom}(Fd).\ \texttt{frameSize}(Fd(fid)) \geq 0$$

Stack pointers are the sum of all frame sizes on stack

$$\forall mid \in modIDs.\ \Phi(mid) = \\ \sum_{fid\ \in\ \{fid\ \mid\ \texttt{moduleID}(Fd(fid))\ =\ mid\}} \texttt{frameSize}(Fd(fid)) \times (\texttt{countIn}((fid, \_), stk) + (pc = (fid, \_)\ ?\ 1\ :\ 0))$$

Variables occupy exactly the frame

$$\forall mid \in modIDs, fid \in \mathsf{dom}(Fd).\ \biguplus_{vid \in \texttt{localIDs}(Fd(fid)) \cup \texttt{args}(Fd(fid))} \beta(vid, fid, mid) = [-\texttt{frameSize}(Fd(fid)), 0)$$

Static variables occupy exactly the data segment

$$\forall mid \in modIDs.\ \biguplus_{vid \in MVar(mid)} \beta(vid, \bot, mid) = [0, \Delta(mid).2 - \Delta(mid).1)$$

One address per command

$$\forall mid \in modIDs, fid \in \mathsf{dom}(Fd).\ |K_{fun}(fid)| = |\texttt{commands}(Fd(fid))|$$

Module's code is a contiguous concatenation of its functions

$$\forall mid \in modIDs.\ \biguplus_{fid\ \in\ \{fid\ \mid\ \texttt{moduleID}(Fd(fid))\ =\ mid\}} K_{fun}(fid) = [0, |K_{mod}(mid)|)$$

Data segments are disjoint and code segments are disjoint

$$\forall mid, mid' \in modIDs.\ mid \neq mid' \implies \Delta(mid) \cap \Delta(mid') = \emptyset \wedge K_{mod}(mid) \cap K_{mod}(mid') = \emptyset$$

If no function has been called, then `main` is executing

$$stk = \texttt{nil} \implies pc.fid = \texttt{main}$$

The first function to start executing was `main`

$$stk \neq \texttt{nil} \implies stk(0).fid = \texttt{main}$$

Stack addresses (capabilities) only live on the stack

$$\forall mid, a, \sigma, e.\ Mem(a) = (\delta, \sigma, e, \_) \ \wedge \ [\sigma, e) \cap \Sigma(mid) \neq \emptyset \implies a \in \Sigma(mid)$$

Dynamically-allocated addresses are negative
$nalloc < 0$

Our goal is $\overline{mods}; \Sigma; \Delta; \beta; MVar; Fd \vdash_{exec} \langle Mem', stk', pc', \Phi', nalloc' \rangle$. We prove it using rule Exec-state-src. We use the names that we gave to the assumptions above to also describe the subgoals about the state $\langle Mem', stk', pc', \Phi', nalloc' \rangle$.

The following subgoals are immediate:

- **Well formed program and parameters** (This is a predicate of only the program text $\overline{mods}$, and the static parameters $\Delta, \Sigma, \beta, K_{mod}, K_{fun}$.)
- **Module IDs**,
- **Equal domains**,
- **Function definitions**,
- **Fd**, and
- **MVar**

It remains to prove the following subgoals:

- **pc points to an existing function**,
- **All pc's on stack point to existing functions**,
- **Static addresses are mapped addresses**,
- **No address exists that is out-of-memory**,
- **Stack pointers are the sum of all frame sizes on stack**,
- **No stack overflow**,
- **Stack addresses (capabilities) only live on the stack**, and
- **Dynamically-allocated addresses are negative**.

We prove them by case distinction over the reduction relation $\rightarrow$.

Case Assign-to-var-or-arr:

The goal "**pc points to an existing function**" is immediate from the corresponding assumption.

The goal "**All pc's on stack point to existing functions**" is immediate from the corresponding assumption by substitution.

In this case, the goal "**Static addresses are mapped addresses**" about Mem' holds by transitivity of $\subseteq$ after noticing that $\mathsf{dom}(Mem) \subseteq \mathsf{dom}(Mem')$.

The goals "**No stack overflow**" and "**Stack pointers are the sum of all frame sizes on stack**" follow by substitution using $\Phi' = \Phi$ and $stk = stk'$.

The goal "**No address exists that is out-of-memory**" has three conjuncts:
Conjunct $nalloc' > \nabla$ holds by substitution using the precondition $nalloc' = nalloc$.

The second and third conjuncts follow from the corresponding assumption "**No address exists that is out-of-memory**" relying on Lemmas 57 and 81. (A detailed proof would be similar to the one in the next case. We skip it here for brevity.)

The goals "**If no function has been called, then `main` is executing**" and "**The first function to start executing was `main`**" are immediate from the corresponding assumptions after substitution using $stk = stk'$ and $pc.fid = pc'.fid$.

To prove the goal "**Stack addresses (capabilities) only live on the stack**", we obtain the precondition $s'.Mem = s.Mem[\sigma + off \mapsto v]$.

Then, we fix an arbitrary memory address a, and an arbitrary module ID mid. We prove our goal for the following two cases:

- **Case $a = \sigma + off$:**
  Here, we obtain the following preconditions of rule Assign-to-var-or-arr:
  (PRECOND-ASSN):
  $modID = \mathtt{moduleID}(Fd(s.pc.fid))$, and
  $v = (\delta, \sigma', e', \_) \implies ([\sigma', e') \cap \Sigma(modID) = \emptyset \vee [\sigma, e) \subseteq \Sigma(modID))$
  Assuming (STK-CAP-ASSM):
  $v = (\delta, \sigma', e', \_) \wedge [\sigma', e') \cap \Sigma(mid) \neq \emptyset$,
  our goal is $\sigma + off \in \Sigma(mid)$.
  We distinguish the following two cases:
  - **Case $mid \neq modID$:**
    Here, we obtain a contradiction to the assumption $[\sigma', e') \cap \Sigma(mid) \neq \emptyset$. Here is how we show $[\sigma', e') \cap \Sigma(mid) = \emptyset$.
    * **First**, we show $[\sigma', e') \subseteq \mathsf{reachable\_addresses}(\Sigma, \Delta, \{modID\}, s.Mem)$.
    * To prove this, we apply Lemma 81 choosing $modIDs = \{modID\}$ to obtain the following subgoals:
      · $e_r, \Sigma, \Delta, \beta, MVar, Fd, s.Mem, s.\Phi, s.pc \Downarrow (\delta, \sigma', e', \_)$
      This is immediate by the precondition of Assign-to-var-or-arr together with the assumption (STK-CAP-ASSM).
      · $\_ \vdash_{exec} s$
      This is immediate by our lemma's assumption.
      · $\mathtt{moduleID}(Fd(s.pc.fid)) \in \{modID\}$
      This is immediate by (PRECOND-ASSN).
    * **Second**, we show that $\mathsf{reachable\_addresses}(\Sigma, \Delta, \{modID\}, s.Mem) \cap \Sigma(mid) = \emptyset$
      By unfolding Definitions 48 and 49, our goal is:
      $(\Delta(modID) \cup \Sigma(modID) \cup \mathsf{access}_{|s.Mem|}(\Delta(modID) \cup \Sigma(modID), s.Mem)) \cap \Sigma(mid) = \emptyset$
      It suffices by easy set identities to show individually:
      · $\Delta(modID) \cap \Sigma(mid) = \emptyset$
      Immediate by **Well formed programs and parameters**.
      · $\Sigma(modID) \cap \Sigma(mid) = \emptyset$
      Immediate by **Well formed programs and parameters**.
      · $\mathsf{access}_{|s.Mem|}(\Delta(modID) \cup \Sigma(modID), s.Mem)) \cap \Sigma(mid) = \emptyset$
      We prove it by induction on k with $0 \leq k \leq |s.Mem|$.
      Base case:
      $\mathsf{access}_0(\Delta(modID) \cup \Sigma(modID), s.Mem)) \cap \Sigma(mid) = \emptyset$
      By Definition 48, it suffices to prove $(\Delta(modID) \cup \Sigma(modID)) \cap \Sigma(mid) = \emptyset$.
      This is the same as the previous cases.

Inductive case:
The induction hypothesis is:
$\mathsf{access}_k(\Delta(\mathit{modID}) \cup \Sigma(\mathit{modID}), s.\mathit{Mem})) \cap \Sigma(\mathit{mid}) = \emptyset$.
And for convenience let:
$A = \mathsf{access}_k(\Delta(\mathit{modID}) \cup \Sigma(\mathit{modID}), s.\mathit{Mem}))$
Our goal is:
$\mathsf{access}_{k+1}(\Delta(\mathit{modID}) \cup \Sigma(\mathit{modID}), s.\mathit{Mem})) \cap \Sigma(\mathit{mid}) = \emptyset$
By Definitions 47 and 48 and after simplification using the induction hypothesis, it suffices for the remaining subgoal to prove:
$\forall a' \in A.\ s.\mathit{Mem}(a') = (\delta, \sigma', e', \_) \implies [\sigma', e') \cap \Sigma(\mathit{mid}) = \emptyset$
We prove it by contradiction. Assume the contrary, i.e., assume for an arbitrary address $a' \in A$ that $s.\mathit{Mem}(a') = (\delta, \sigma', e', \_) \wedge [\sigma', e') \cap \Sigma(\mathit{mid}) \neq \emptyset$
Now by assumption **"Stack addresses (capabilities) only live on the stack"**, we have (*):
$a' \in \Sigma(\mathit{mid})$
But we know $a' \in A$, and by the induction hypothesis, we know $A \cap \Sigma(\mathit{mid}) = \emptyset$.
Thus, we know that $a' \notin \Sigma(\mathit{mid})$ (contradiction to (*)).
This **concludes our inductive proof** that
$\mathsf{access}_{|s.\mathit{Mem}|}(\Delta(\mathit{modID}) \cup \Sigma(\mathit{modID}), s.\mathit{Mem})) \cap \Sigma(\mathit{mid}) = \emptyset$.

This concludes the proof of **Second** which concludes the proof of **Case** $\mathit{mid} \neq \mathit{modID}$.

- **Case** $\mathit{mid} = \mathit{modID}$**:**
  By instantiating (PRECOND-ASSN) using the assumptions above, we obtain the following two cases:
  * **Case** $[\sigma', e') \cap \Sigma(\mathit{modID}) = \emptyset$**:**
    Here, we obtain a contradiction to our assumptions. So, any goal is provable.
  * **Case** $[\sigma, e) \subseteq \Sigma(\mathit{modID})$**:**
    Here, our goal is immediate by compatibility of $\in$ and $\subseteq$ because of the precondition $\sigma + \mathit{off} \in [\sigma, e)$ together with our case condition.

- **Case** $a \neq \sigma + \mathit{off}$**:**
  Here, our goal is immediate by the corresponding assumption.

The goal "**Dynamically-allocated addresses are negative**" is immediate by substitution using $s'.\mathit{nalloc} = s.\mathit{nalloc}$.

Case Allocate:

The goal "**pc points to an existing function**" is immediate from the corresponding assumption.

The goal "**All pc's on stack point to existing functions**" is immediate from the corresponding assumption by substitution.

In this case, the goal "**Static addresses are mapped addresses**" about Mem' holds by transitivity of $\subseteq$ after noticing that $\mathsf{dom}(\mathit{Mem}) \subseteq \mathsf{dom}(\mathit{Mem}')$.

Next, we prove the goal " **No address exists that is out-of-memory**".

In this case, we obtain the preconditions $\mathit{nalloc} - v > \nabla$ and $\mathit{nalloc}' = \mathit{nalloc} - v$ which by substitution in one another prove the first conjunct of the consequent of statement **No address exists that is out-of-memory**.

The second conjunct of **No address exists that is out-of-memory** is proved by fixing an arbitrary $a \in \mathsf{dom}(\mathit{Mem}')$ and distinguishing the cases that arise by the precondition
$\mathit{Mem}' = \mathit{Mem}[s + \mathit{off} \mapsto (\delta, \mathit{nalloc}', \mathit{nalloc}, 0)][a \mapsto 0 \mid a \in [\mathit{nalloc}', \mathit{nalloc})]$:

- **Case** $a \notin \{s + \mathit{off}\} \cup [\mathit{nalloc}', \mathit{nalloc})$**:**
  Follows by the corresponding assumption, i.e., "**No address exists that is out-of-memory**".
- **Case** $a = s + \mathit{off}$**:**
  In this case, we know by Lemma 81 that:
  $a \in \mathsf{reachable\_addresses}(\Sigma, \Delta, \mathit{modIDs}, \mathit{Mem})$.
  Thus, by Lemma 57, we know:
  $a \in \mathsf{static\_addresses}(\Sigma, \Delta, \mathit{modIDs}) \lor \exists s, e.\ (\delta, s, e, \_) \in \mathsf{range}(\mathit{Mem}) \land a \in [s, e)$
  Thus, we consider each case:
  - **Case** $a \in \mathsf{static\_addresses}(\Sigma, \Delta, \mathit{modIDs})$**:**
    Here, by transitivity of $\subseteq$ from assumption **"Static addresses are mapped addresses"**, we have:
    $a \in \mathsf{dom}(\mathit{Mem})$.
    So, our conclusion $a > \nabla$ follows by assumption "**No address exists that is out of memory**".
  - **Case** $\exists s, e.\ (\delta, s, e, \_) \in \mathsf{range}(\mathit{Mem}) \land a \in [s, e)$**:**
    Here, our conclusion $a > \nabla$ follows by assumption "**No address exists that is out of memory**".
- **Case** $a \in [\mathit{nalloc}', \mathit{nalloc})$**:**
  In this case, $a \geq \mathit{nalloc}'$ and $\mathit{nalloc}' > \nabla$ (which is a precondition of Allocate) give us our conclusion $a > \nabla$.

The third conjunct of the goal "**No address exists that is out-of-memory**" is proved by fixing arbitrary s, e with $(\delta, s, e, \_) \in \mathsf{range}(\mathit{Mem}') \land a \in [s, e)$ and proving that $a > \nabla$.

We distinguish the following cases based on the definition of Mem' (similar to the cases above for $a \in \mathsf{dom}(\mathit{Mem}')$):

- **Case** $(\delta, s, e, \_) \in \mathsf{range}(\mathit{Mem})$**:**
  Here, our goal follows by the third conjunct of the corresponding assumption, i.e., "**No address exists that is out-of-memory**".
- **Case** $(\delta, s, e, \_) = \mathit{Mem}'(s + \mathit{off})$**:**
  Here, the goal follows by the conclusion $\mathit{nalloc}' > \nabla$ that we already argued.
- **Case** $(\delta, s, e, \_) = \mathit{Mem}'(a') \land a' \in [\mathit{nalloc}', \mathit{nalloc})$**:**
  This is an impossible case because $\mathit{Mem}(a') = 0$ in this case by the definition of Mem'.

This concludes the proof of the goal "**No address exists that is out-of-memory**".

The goals "**No stack overflow**" and "**Stack pointers are the sum of all frame sizes on stack**" follow by substitution using $\Phi' = \Phi$ and $\mathit{stk} = \mathit{stk}'$.

The goals "**If no function has been called, then `main` is executing**" and "**The first function to start executing was `main`**" are proved exactly as in the previous case.

We prove the goal "**Stack addresses (capabilities) only live on the stack**" by fixing an arbitrary address a where $a \in \mathsf{dom}(\mathit{Mem}')$ and distinguishing the cases that arise by the precondition
$\mathit{Mem}' = \mathit{Mem}[\sigma + \mathit{off} \mapsto (\delta, \mathit{nalloc}', \mathit{nalloc}, 0)][a \mapsto 0 \mid a \in [\mathit{nalloc}', \mathit{nalloc})]$:

- **Case** $a \notin \{\sigma + \mathit{off}\} \cup [s'.\mathit{nalloc}, s.\mathit{nalloc})$**:**
  Here, our goal is immediate by the corresponding assumption.

- **Case** $a = \sigma + \mathit{off}$**:**
  Here, we know $s'.\mathit{Mem}(a) = (\delta, s'.\mathit{nalloc}, s.\mathit{nalloc}, 0)$.
  So, we prove our goal vacuously by proving that:
  $[s'.\mathit{nalloc}, s.\mathit{nalloc}) \cap \Sigma(\mathit{mid}) = \emptyset$.
  By inversion of rule Well-formed program and parameters in assumption **Well formed programs and parameters**, and by applying the obtained precondition:
  $\Sigma(\mathit{mid}) \cap (-\infty, 0) = \emptyset$
  to our goal, we obtain the following subgoal:
  $[s'.\mathit{nalloc}, s.\mathit{nalloc}) \subseteq (-\infty, 0)$
  This is immediate by assumption "**Dynamically-allocated addresses are negative**".
- **Case** $a \in [s'.\mathit{nalloc}, s.\mathit{nalloc})$**:**
  Here, our goal is vacuously true.

Case Call:

The goal "**pc points to an existing function**" follows from the precondition $\mathit{modID} = \mathtt{moduleID}(\mathit{Fd}(\mathit{fid}_{call}))$.

The goal "**All pc's on stack point to existing functions**" follows from both the corresponding assumption and from the assumption **pc points to an existing function**.

In this case, the goal "**Static addresses are mapped addresses**" about Mem' holds by transitivity of $\subseteq$ after noticing that $\mathsf{dom}(\mathit{Mem}) \subseteq \mathsf{dom}(\mathit{Mem}')$.

The goal "**No address exists that is out-of-memory**" has three conjuncts:
Conjunct $\mathit{nalloc}' > \nabla$ holds by substitution using the precondition $\mathit{nalloc}' = \mathit{nalloc}$.

The second and third conjuncts follow from the corresponding assumption "**No address exists that is out-of-memory**" relying on Lemmas 57 and 81. (A detailed proof would be similar to the one in **case Allocate**. We skip it here for brevity.)

Next, we prove the goal "**No stack overflow**", namely:
$\forall \mathit{mid} \in \mathit{modIDs}.\ \Sigma(\mathit{mid}).1 + \Phi'(\mathit{mid}) \leq \Sigma(\mathit{mid}).2$.

We obtain from Call the preconditions:

- $\Sigma(\mathit{modID}).1 + \Phi(\mathit{modID}) + \mathit{frameSize} \leq \Sigma(\mathit{modID}).2$
- $\Phi' = \Phi[\mathit{modID} \mapsto \Phi(\mathit{modID}) + \mathit{frameSize}]$

These are sufficient to immediately prove our goal after case distinction on $\mathit{mid} = \mathit{modID}$.

Next, we prove the goal "**Stack pointers are the sum of all frame sizes on stack**".

Our goal is:
$$\forall \mathit{mid} \in \mathit{modIDs}.\ \Phi'(\mathit{mid}) = \sum_{\mathit{fid}\ \in\ \{\mathit{fid}\ \mid\ \mathtt{moduleID}(\mathit{Fd}(\mathit{fid}))\ =\ \mathit{mid}\}} \mathtt{frameSize}(\mathit{Fd}(\mathit{fid})) \times (\mathtt{countIn}((\mathit{fid}, \_), \mathit{stk}') + (\mathit{pc}' = (\mathit{fid}, \_)\ ?\ 1\ :\ 0))$$
We distinguish three cases:

- **Case** $\mathit{mid} = \mathtt{moduleID}(\mathit{Fd}(\mathit{fid}_{call}))$**:**
  In this case, we further distinguish two cases:
  - **Case** $\mathit{pc}.\mathit{fid} = \mathit{fid}_{call}$, and

- **Case** $pc.fid \neq fid_{call}$**:**
  In both of these cases, we notice that the right-hand-side factor in the right side of the equality increases by one for the term corresponding to fid_{call}.
  Thus, by the precondition $\Phi'(mid) = \Phi(mid) + \texttt{frameSize}(Fd(fid_{call}))$, we can satisfy the equality.

- **Case** $mid \neq \texttt{moduleID}(Fd(fid_{call})) \wedge mid = \texttt{moduleID}(Fd(pc.fid))$**:**

  In this case, we notice that all the terms of the right side of the equality remain the same. And in particular the term for $pc.fid$ remains the same because its right-hand-side factor remains the same because:
  $(pc' = (pc.fid)?1 : 0) - (pc = (pc.fid)?1 : 0) = -1$, and
  $\texttt{countIn}((pc.fid, \_), stk') - \texttt{countIn}((pc.fid, \_), stk) = 1$

  Thus, by substituting using the precondition $\Phi'(mid) = \Phi(mid)$ in the left side of our goal equality, our goal holds by assumption.

- **Case** $mid \neq \texttt{moduleID}(Fd(fid_{call})) \wedge mid \neq \texttt{moduleID}(Fd(pc.fid))$**:**

  In this case, our goal holds directly by the assumption.

This concludes the proof of the goal "**Stack pointers are the sum of all frame sizes on stack**".

The goal "**If no function has been called, then** $\texttt{main}$ **is executing**" is vacuously true by noticing that $stk' \neq \texttt{nil}$.

To prove the goal "**The first function to start executing was** $\texttt{main}$", i.e.,
$stk' \neq \texttt{nil} \implies stk'(0).fid = \texttt{main}$, we distinguish the following two cases:

- **Case** $stk = \texttt{nil}$**:**

  Here, by assumption "**If no function has been called, then** $\texttt{main}$ **is executing**", we know $pc.fid = \texttt{main}$. Thus, by the precondition $stk' = \texttt{push}(stk, pc)$, we have our goal.

- **Case** $stk \neq \texttt{nil}$**:**

  Here, observe that $stk(0) = stk'(0)$, so our goal is immediate by the corresponding assumption about stk.

We prove the goal "**Stack addresses (capabilities) only live on the stack**" by fixing an arbitrary address a where $a \in \mathsf{dom}(Mem')$ and distinguishing the cases that arise by the precondition:
$s'.Mem = s.Mem[\phi' + s_i \mapsto v_i \mid \beta(argNames(i), fid_{call}, modID) = [s_i, \_) \wedge i \in [0, nArgs)]$
$[\phi' + s_i \mapsto 0 \mid \beta(localIDs(i), fid_{call}, modID) = [s_i, \_) \wedge i \in [0, nLocal)]$

- **Case** $\exists i \in [0, nArgs).\ a \in \phi' + \beta(argNames(i), fid_{call}, modID)$**:**

  Here, we obtain the following precondition of rule Call:
  (PRECOND-CALL):
  $curModID = \texttt{moduleID}(Fd(fid))$, and
  $\forall i \in [0, nArgs), s', e'.\ \ v_i = (\delta, s', e', \_) \implies [s', e') \cap \Sigma(curModID) = \emptyset$

  We now obtain from our case condition $i \in [0, nArgs)$, and we instantiate (PRECOND-CALL).

  The proof then proceeds exactly as in case Assign-to-var-or-arr replacing (PRECOND-ASSIGN) with (PRECOND-CALL). We avoid repetition.

- **Case** $\exists i \in [0, nLocal).\ a \in \phi' + \beta(localIDs(i), fid_{call}, modID)$**:**

  Here, our goal holds vacuously.

- **Case** $\nexists i.\ i \in [0, nArgs)\ \wedge\ a \in \phi' + \beta(argNames(i), fid_{call}, modID)\ \vee$ $i \in [0, nLocal)\ \wedge\ a \in \phi' + \beta(localIDs(i), fid_{call}, modID)$**:**
  Here, our goal is immediate by the corresponding assumption.

The goal "**Dynamically-allocated addresses are negative**" is immediate by substitution using $s'.nalloc = s.nalloc$.

Case Return:

The goal "**pc points to an existing function**" follows from the assumption **All pc's on stack point to existing functions**.

The goal "**All pc's on stack point to existing functions**" follows from the corresponding assumption.

In this case, the goal "**Static addresses are mapped addresses**" about Mem' holds by substitution using $Mem' = Mem$.

The goal "**No address exists that is out-of-memory**" holds by substitution using the preconditions $nalloc' = nalloc$ and $Mem' = Mem$.

Next, we prove the goal "**Stack pointers are the sum of all frame sizes on stack**".

Our goal is:

$$\forall mid \in modIDs.\ \Phi'(mid) = \sum_{fid\ \in\ \{fid\ |\ \texttt{moduleID}(Fd(fid))\ =\ mid\}} \texttt{frameSize}(Fd(fid)) \times (\texttt{countIn}((fid, \_), stk') + (pc' = (fid, \_)\ ?\ 1\ :\ 0))$$

We distinguish three cases:

- **Case** $mid = \texttt{moduleID}(Fd(pc.fid))$**:**
  In this case, we further distinguish two cases:
  - **Case** $pc.fid = pc'.fid$, and
  - **Case** $pc.fid \neq pc'.fid$**:**
    In both of these cases, we notice that the right-hand-side factor in the right side of the equality decreases by one for the term corresponding to $pc.fid$.
    Thus, by the precondition $\Phi'(mid) = \Phi(mid) - \texttt{frameSize}(Fd(pc.fid))$, we can satisfy the equality.
- **Case** $mid \neq \texttt{moduleID}(Fd(pc.fid)) \wedge mid = \texttt{moduleID}(Fd(pc'.fid))$**:**
  In this case, we notice that all the terms of the right side of the equality remain the same. And in particular the term for $pc'.fid$ remains the same because its right-hand-side factor remains the same because:
  $(pc' = (pc'.fid)?1 : 0) - (pc = (pc'.fid)?1 : 0) = 1$, and
  $\texttt{countIn}((pc'.fid, \_), stk') - \texttt{countIn}((pc'.fid, \_), stk) = -1$
  Thus, by substituting using the precondition $\Phi'(mid) = \Phi(mid)$ in the left side of our goal equality, our goal holds by assumption.
- **Case** $mid \neq \texttt{moduleID}(Fd(pc.fid)) \wedge mid \neq \texttt{moduleID}(Fd(pc'.fid))$**:**
  In this case, our goal holds directly by the assumption.

This concludes the proof of the goal "**Stack pointers are the sum of all frame sizes on stack**".

Next, we prove the goal "**No stack overflow**", namely:
$\forall mid \in modIDs.\ \Sigma(mid).1 + \Phi'(mid) \leq \Sigma(mid).2$.

Here, by case distinction on $mid = \texttt{moduleID}(Fd(pc.fid))$, our goal follows immediately by transitivity of $\leq$ after obtaining the precondition
$\Phi'(mid) = \Phi(mid) - frameSize$
in one case, and immediately by assumption in the other case.

(The assumption "**Frame sizes are non-negative**" was used here.)

The goal "**If no function has been called, then** `main` **is executing**" follows from assumption "**The first function to start executing was** `main`" about stk.

The goal "**The first function to start executing was** `main`" follows from the corresponding assumption about stk.

The goal "**Stack addresses (capabilities) only live on the stack**" is immediate after substitution using $s'.Mem = s.Mem$.

The goal "**Dynamically-allocated addresses are negative**" is immediate by substitution using $s'.nalloc = s.nalloc$.

Case Jump-zero:
All remaining goals hold by substitution (using $\Phi' = \Phi$, $stk = stk'$, $nalloc' = nalloc$, $Mem' = Mem$, and $pc'.1 = pc.1$)

Case Jump-non-zero:
All remaining goals hold by substitution (using $\Phi' = \Phi$, $stk = stk'$, $nalloc' = nalloc$, $Mem' = Mem$, and $pc'.1 = pc.1$)

Case Exit:
Here, all goals hold by substitution (using $\Phi' = \Phi$, $stk = stk'$, $nalloc' = nalloc$, $Mem' = Mem$, and $pc' = pc$).

This concludes the proof of Lemma 56. □

Corollary 4 (Preservation of $\vdash_{exec}$ by the reflexive transitive closure)**.**

$$\forall \overline{mods}, s, s'.\ \overline{mods} \vdash_{exec} s \ \wedge\ s \rightarrow^* s' \quad \Longrightarrow \quad \overline{mods} \vdash_{exec} s'$$

Proof. Trivial by Lemma 56. □

2.5 Memory Reachability

Given a memory context $\Sigma; \Delta; \beta; MVar; Fd$ and a **ImpMod** program state $\langle Mem, stk, pc, \Phi, nalloc \rangle$, we would like to characterize the set $A \subseteq \mathbb{Z}$ of reachable memory addresses which informally captures all the addresses that an expression in the given state can evaluate to. In other words, the set A of reachable addresses should satisfy the condition that whenever an expression e evaluates to an address in the given state (i.e., $e, \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow a$ where $a = (\delta, st, end, \_)$), then $[st, end) \subseteq A$.

More formally, Lemma 81 captures the previous intuition.

Definition 46 (Static Addresses)**.**

$$\begin{aligned}
&\mathsf{static_addresses}(\Sigma, \Delta, modIDs) \stackrel{\mathsf{def}}{=} \\
&\{a \mid a \in \Delta(mid) \wedge mid \in modIDs\} \\
&\uplus \\
&\{a \mid a \in \Sigma(mid) \wedge mid \in modIDs\}
\end{aligned}$$

Definition 47 (Memory accessibility).

$$\begin{aligned}&\mathsf{access}(A, Mem) \stackrel{\mathsf{def}}{=} \\ &A \ \cup \ \{a \mid a \in [s, e) \wedge Mem(a') = (\delta, s, e, \_) \wedge a' \in A\}\end{aligned}$$

Definition 48 (Memory k-accessibility).

$$\mathsf{access}_0(A, \_) = A$$

$$\mathsf{access}_{k+1}(A, Mem) \stackrel{\mathsf{def}}{=} \mathsf{access}(\mathsf{access}_k(A, Mem), Mem)$$

Definition 49 (Reachable Addresses).

$$\begin{aligned}&\mathsf{reachable\_addresses}(\Sigma, \Delta, modIDs, Mem) \stackrel{\mathsf{def}}{=} \\ &\mathsf{static\_addresses}(\Sigma, \Delta, modIDs) \\ &\cup \ \mathsf{access}_{|Mem|}(\mathsf{static\_addresses}(\Sigma, \Delta, modIDs), Mem)\end{aligned}$$

Lemma 57 (Reachable addresses are static addresses or are memory-stored).

$$\begin{aligned}&\forall a, \Sigma, \Delta, modIDs, Mem. \\ &a \in \mathsf{reachable\_addresses}(\Sigma, \Delta, modIDs, Mem) \implies \\ &a \in \mathsf{static\_addresses}(\Sigma, \Delta, modIDs) \vee \exists s, e. \ (\delta, s, e, \_) \in \mathsf{range}(Mem) \wedge a \in [s, e)\end{aligned}$$

Proof. By Definitions 46 to 49. ☐

Lemma 58 (access is expansive).

$$\forall A, Mem. \ \mathsf{access}(A, Mem) \supseteq A$$

Proof. Similar to Lemma 7 ☐

Lemma 59 (access_n is expansive).

$$\forall n, A, Mem. \ \mathsf{access}_n(A, Mem) \supseteq A$$

Proof. Similar to Lemma 8 ☐

Lemma 60 (Fixed points lead to convergence of access_k).

$$\begin{aligned}&\forall k, Mem, A. \ k > 0 \\ &\implies (\mathsf{access}_k(A, Mem) = A \implies \mathsf{access}_{k+1}(A, Mem) = A)\end{aligned}$$

Proof. Similar to Lemma 9 ☐

Lemma 61 (In an empty memory, only the starting addresses are reachable).

$$\begin{aligned}&\forall \Sigma, \Delta, modIDs, Mem. \\ &(\forall v. \ v \in \mathsf{range}(Mem) \implies v \neq (\delta, \_, \_, \_)) \\ &\implies \mathsf{reachable\_addresses}(\Sigma, \Delta, modIDs, Mem) = \mathsf{static\_addresses}(\Sigma, \Delta, modIDs)\end{aligned}$$

Proof. Similar to Lemma 10. Immediate by Definitions 47 to 49. ☐

Lemma 62 (k-accessibility either adds a new memory address or a fixed point has been reached).

$$\begin{aligned}&\forall k, A, Mem. \ k > 0 \implies \\ &\mathsf{access}_k(A, Mem) \supsetneq \mathsf{access}_{k+1}(A, Mem) \implies \\ &\exists a. \ a \in \mathsf{dom}(Mem) \ \wedge \ a \in \mathsf{access}_k(A, Mem) \setminus \mathsf{access}_{k-1}(A, Mem)\end{aligned}$$

Proof. Similar to Lemma 11 ☐

Lemma 63 (k-accessibility set contains at least k mapped addresses)**.**

$$\begin{aligned}&\forall k, A, Mem.\\&\mathsf{access}_{k+1}(A, Mem) \supsetneq \mathsf{access}_k(A, Mem) \implies\\&|\{a \mid a \in \mathsf{access}_k(A, Mem) \wedge a \in \mathsf{dom}(Mem)\}| > k\end{aligned}$$

Proof. Similar to Lemma 12 ☐

Lemma 64 ($|Mem|$-accessibility suffices)**.**

$$\begin{aligned}&\forall A, Mem, k.\ k \geq 0 \implies\\&\mathsf{access}_{|Mem|+k}(A, Mem) = \mathsf{access}_{|Mem|}(A, Mem)\end{aligned}$$

Proof. Similar to lemma 13 ☐

Lemma 65 (Safe allocation adds only allocated addresses to k-accessibility)**.**

$$\begin{aligned}&\forall A, Mem, \hat{a}, a_a, \sigma, e, k.\\&\forall a \in [\sigma, e).\ Mem[\hat{a} \mapsto (\delta, \sigma, e, \_)](a) = v \implies v \notin \{\delta\} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \wedge\\&a_a \in \mathsf{access}_k(A, Mem[\hat{a} \mapsto (\delta, \sigma, e, \_)])\\&\implies a_a \in \mathsf{access}_k(A, Mem) \vee a_a \in [\sigma, e)\end{aligned}$$

Proof. Similar to Lemma 39. ☐

Lemma 66 (Safe allocation adds only allocated addresses to reachability)**.**

$$\begin{aligned}&\forall \Sigma, \Delta, modIDs, Mem, \hat{a}, a_a, \sigma, e.\\&\forall a \in [\sigma, e).\ Mem[\hat{a} \mapsto (\delta, \sigma, e, \_)](a) = v \implies v \notin \{\delta\} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \wedge\\&a_a \in \mathsf{reachable\_addresses}(\Sigma, \Delta, modIDs, Mem[\hat{a} \mapsto (\delta, \sigma, e, \_)])\\&\implies a_a \in \mathsf{reachable\_addresses}(\Sigma, \Delta, modIDs, Mem) \vee a_a \in [\sigma, e)\end{aligned}$$

Proof. Similar to Lemma 40. ☐

Lemma 67 (Safe allocation causes reduction of k-accessibility to χ_k and addition of exactly the allocated addresses)**.**

$$\begin{aligned}&\forall A, Mem, \hat{a}, a_a, \sigma, e, k.\\&\forall a \in [\sigma, e).\ Mem[\hat{a} \mapsto (\delta, \sigma, e, \_)](a) = v \implies v \notin \{\delta\} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \wedge\\&\hat{a} \in \mathsf{access}_k(A, Mem)\\&\implies\\&\mathsf{access}_k(A, Mem[\hat{a} \mapsto (\delta, \sigma, e, \_)]) = \chi_k(A, Mem, \hat{a}) \cup [\sigma, e)\end{aligned}$$

Proof. Similar to Lemma 41. Should follow by induction on k, and should be similar to the proof of Lemma 65. ☐

Lemma 68 (Invariance to unreachable memory updates)**.**

$$\begin{aligned}&\forall \Sigma, \Delta, modIDs, Mem, a, v.\ a \notin \mathsf{reachable\_addresses}(\Sigma, \Delta, modIDs, Mem) \implies\\&\mathsf{reachable\_addresses}(\Sigma, \Delta, modIDs, Mem) = \mathsf{reachable\_addresses}(\Sigma, \Delta, modIDs, Mem[a \mapsto v])\end{aligned}$$

Proof.
Similar to Lemma 21 using Lemmas 59, 63 and 69. ☐

Lemma 69 (Updating k-inaccessible locations does not affect the k-accessibility set).

$$\forall a, k, Mem, A, v.\ a \notin \mathsf{access}_k(A, Mem) \implies \mathsf{access}_k(A, Mem) = \mathsf{access}_k(A, Mem[a \mapsto v])$$

Proof.
Similar to Lemma 22 using Definitions 47 and 48. □

Lemma 70 (Updating a location does not affect its own k-accessibility).

$$\forall a, A, k_a, Mem, v.\ a \in \mathsf{access}_{k_a}(A, Mem) \implies a \in \mathsf{access}_{k_a}(A, Mem[a \mapsto v])$$

Proof.
Similar to Lemma 23 using Lemma 69. □

Lemma 71 (Updating a location does not affect its own reachability).

$$\begin{aligned}&\forall \Sigma, \Delta, modIDs, a, v, Mem.\\ &a \in \mathsf{reachable\_addresses}(\Sigma, \Delta, modIDs, Mem) \implies\\ &a \in \mathsf{reachable\_addresses}(\Sigma, \Delta, modIDs, Mem[a \mapsto v])\end{aligned}$$

Proof.
Similar to Lemma 24 using Lemma 70 and definition 49. □

Lemma 72 (χ_k is upper-bounded by k-accessibility).

$$\forall k, Mem, A, a.\ \chi_k(A, Mem, a) \subseteq \mathsf{access}_k(A, Mem)$$

Proof. Immediate by Definitions 25 and 48. □

Lemma 73 (One capability is potentially lost from accessible addresses as a result of a non-capability update).

$$\forall A, a, Mem, v.\ v \neq (\delta, \_, \_, \_) \implies \mathsf{access}(A, Mem[a \mapsto v]) = \chi(A, Mem, a)$$

Proof.
Similar to Lemma 32. Follows from Definitions 24 and 47 by observing that $Mem[a \mapsto v](a) \neq (\delta, \_, \_, \_)$ and that $Mem[a \mapsto v](a') = Mem(a')$ for $a' \neq a$. □

Lemma 74 (χ_k captures k-accessibility after potential deletion of a capability).

$$\forall A, a, Mem, v.\ v \neq (\delta, \_, \_, \_) \implies \mathsf{access}_k(A, Mem[a \mapsto v]) = \chi_k(A, Mem, a)$$

Proof.
Similar to Lemma 33. Follows by induction on k from Definitions 25 and 48 using Lemma 73. □

Lemma 75 (Reachability is captured by union over χ_k after potential deletion of a capability).

$$\forall \Sigma, \Delta, modIDs, Mem, a, v.\ v \neq (\delta, \_, \_, \_) \implies$$
$$\mathsf{reachable\_addresses}(\Sigma, \Delta, modIDs, Mem[a \mapsto v]) = \bigcup_k (\chi_k(\mathsf{static\_addresses}(\Sigma, \Delta, modIDs, Mem), Mem, a))$$

Proof.
Similar to Lemma 34. Immediate by Definition 49 and lemma 74. □

Definition 50 (Derivable capability). *A capability $c^* = (x, \sigma, e, \_)$ is derivable from reachability parameters $\Sigma, \Delta, modIDs$ on memory Mem, written $\Sigma, \Delta, modIDs \vDash c^*$ iff* $\forall a \in [\sigma, e).\ a \in \mathsf{reachable\_addresses}(\Sigma, \Delta, modIDs, Mem)$.

Lemma 76 (Reachability traverses all derivable capabilities)**.**

$$
\begin{aligned}
&\forall \Sigma, \Delta, \mathit{modIDs}, \mathit{Mem}, c.\\
&\Sigma, \Delta, \mathit{modIDs}, \mathit{Mem} \vDash c \implies\\
&\mathsf{reachable\_addresses}(\Sigma, \Delta, \mathit{modIDs}, \mathit{Mem}) \supseteq [c.\sigma, c.e) \cup \ \mathsf{access}_{|\mathit{Mem}|}([c.\sigma, c.e), \mathit{Mem})
\end{aligned}
$$

Proof. Similar to Lemma 28. □

Lemma 77 (Additivity of access)**.**

$$\forall A_1, A_2, \mathcal{M}_d.\ \mathsf{access}(A_1 \cup A_2, \mathit{Mem}) = \mathsf{access}(A_1, \mathit{Mem}) \cup \mathsf{access}(A_2, \mathit{Mem})$$

Proof. Similar to Lemma 16. □

Lemma 78 (Additivity of access_k)**.**

$$\forall k, A_1, A_2, \mathcal{M}_d.\ \mathsf{access}_k(A_1 \cup A_2, \mathit{Mem}) = \mathsf{access}_k(A_1, \mathit{Mem}) \cup \mathsf{access}_k(A_2, \mathit{Mem})$$

Proof. Similar to Lemma 17. Follows by induction on k using Lemma 77. □

Lemma 79 (Effect of assigning a derivable capability)**.**

$$
\begin{aligned}
&\forall \Sigma, \Delta, \mathit{modIDs}, \mathit{Mem}, a, c.\\
&\Sigma, \Delta, \mathit{modIDs}, \mathit{Mem} \vDash c \ \wedge \ a \in \mathsf{reachable\_addresses}(\Sigma, \Delta, \mathit{modIDs}, \mathit{Mem})\\
&\implies\\
&\mathsf{reachable\_addresses}(\Sigma, \Delta, \mathit{modIDs}, \mathit{Mem}[a \mapsto c]) =\\
&\bigcup_k \chi_k(\mathsf{static\_addresses}(\Sigma, \Delta, \mathit{modIDs}, \mathit{Mem}) \cup [c.\sigma, c.e), \mathit{Mem}, a)
\end{aligned}
$$

Proof. Follows from Lemmas 30, 75 and 78. □

Lemma 80 (Assigning a derivable capability does not enlarge reachability)**.**

$$
\begin{aligned}
&\forall \Sigma, \Delta, \mathit{modIDs}, \mathit{Mem}, a, c.\\
&\Sigma, \Delta, \mathit{modIDs}, \mathit{Mem} \vDash c \ \wedge \ a \in \mathsf{reachable\_addresses}(\Sigma, \Delta, \mathit{modIDs}, \mathit{Mem})\\
&\implies\\
&\mathsf{reachable\_addresses}(\Sigma, \Delta, \mathit{modIDs}, \mathit{Mem}[a \mapsto c]) \subseteq \mathsf{reachable\_addresses}(\Sigma, \Delta, \mathit{modIDs}, \mathit{Mem})
\end{aligned}
$$

Proof. After substitution using Lemma 79, we apply Lemma 30 to get two subgoals that are provable using Lemma 72 and Lemma 76 respectively. □

Lemma 81 (Completeness of reachable_addresses)**.**

$$
\begin{aligned}
&\forall \mathit{st}, \mathit{end}, e, \Sigma, \Delta, \beta, \mathit{MVar}, \mathit{Fd}, \mathit{Mem}, \Phi, \mathit{pc}, \mathit{modIDs}.\\
&e, \Sigma, \Delta, \beta, \mathit{MVar}, \mathit{Fd}, \mathit{Mem}, \Phi, \mathit{pc} \Downarrow (\delta, \mathit{st}, \mathit{end}, \_) \ \wedge\\
&\exists \overline{\mathit{mods}}, \mathit{nalloc}, \mathit{stk}.\ \overline{\mathit{mods}}; \Sigma; \Delta; \beta; \mathit{MVar}; \mathit{Fd} \vdash_{\mathit{exec}} \langle \mathit{Mem}, \mathit{stk}, \mathit{pc}, \Phi, \mathit{nalloc} \rangle \ \wedge\\
&\texttt{moduleID}(\mathit{Fd}(\mathit{pc}.\mathit{fid})) \in \mathit{modIDs}\\
&\implies\\
&[\mathit{st}, \mathit{end}) \subseteq \mathsf{reachable\_addresses}(\Sigma, \Delta, \mathit{modIDs}, \mathit{Mem})
\end{aligned}
$$

Proof.

- We fix arbitrary $st, end, e, \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc$, and assume the antecedent $e, \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow (\delta, st, end, \_)$.
- We prove the consequent by induction on the evaluation of e.
  - **Case Evaluate-expr-const:**
  - **Case Evaluate-expr-cast-to-integer-start:**
  - **Case Evaluate-expr-cast-to-integer-end:**
  - **Case Evaluate-expr-cast-to-integer-offset:**
  - **Case Evaluate-expr-cap-type:**
  - **Case Evaluate-expr-binop:**

    All of these cases are vacuous because in all, the antecedent does not hold because $e, \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow z$ with $z \notin \{\delta\} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$.
  - **Case Evaluate-expr-addr-local**:

    In this case, we obtain the preconditions:
    $(fid, \_) = pc$, $vid \in \texttt{localIDs}(Fd(fid)) \cup \texttt{args}(Fd(fid))$, $mid = \texttt{moduleID}(Fd(fid))$, $\beta(vid, fid, mid) = (s, e)$ and $\phi = \Sigma(mid).1 + \Phi(mid)$.
    Our goal is to show that:
    $[\phi + s, \phi + e) \subseteq \mathsf{reachable\_addresses}(\Sigma, \Delta, modIDs, Mem)$.
    We instead show the following goal:
    $[\phi - frameSize, \phi) \subseteq \mathsf{reachable\_addresses}(\Sigma, \Delta, modIDs, Mem)$
    where $frameSize = \texttt{frameSize}(Fd(fid))$.
    The latter follows immediately by Definitions 46 and 49.
    And it suffices for our goal by transitivity of $\subseteq$ assuming:
    $[\phi + s, \phi + e) \subseteq [\phi - frameSize, \phi)$.
    This latter assumption follows by interval arithmetic identities from:
    $[s, e) \subseteq [-frameSize, 0)$.
    This last statement follows from:
    $$\biguplus_{vid \in \texttt{localIDs}(Fd(fid)) \cup \texttt{args}(Fd(fid))} \beta(vid, fid, mid) = [-frameSize, 0)$$
    which in turn can be obtained from the assumption
    $\overline{mods}; \Sigma; \Delta; \beta; MVar; Fd \vdash_{exec} \langle Mem, stk, pc, \Phi, nalloc \rangle$
    of our lemma by inversion using rule Exec-state-src then inversion using rule Well-formed program and parameters.
    This concludes case Evaluate-expr-addr-local.
  - **Case Evaluate-expr-addr-module**:

    This case is similar to the previous one, but not identical.
    We obtain the preconditions:
    $(fid, \_) = pc$, $vid \notin \texttt{localIDs}(Fd(fid)) \cup \texttt{args}(Fd(fid))$,
    $mid = \texttt{moduleID}(Fd(fid))$, $\beta(vid, \perp, mid) = (s, e)$, and $vid \in MVar(mid)$.
    Our goal is to show that:
    $[\Delta(mid).1 + s, \Delta(mid).1 + e) \subseteq \mathsf{reachable\_addresses}(\Sigma, \Delta, modIDs, Mem)$.
    We instead show the following goal:
    $[\Delta(mid).1, \Delta(mid).2) \subseteq \mathsf{reachable\_addresses}(\Sigma, \Delta, modIDs, Mem)$.
    The latter follows immediately by Definitions 46 and 49.
    And it suffices for our goal by transitivity of $\subseteq$ assuming:
    $[\Delta(mid).1 + s, \Delta(mid).1 + e) \subseteq [\Delta(mid).1, \Delta(mid).2)$.
    This last statement follows from:
    $$\biguplus_{vid \in MVar(mid)} \beta(vid, \perp, mid) = \Delta(mid)$$

which in turn can be obtained from the assumption
$\overline{mods}; \Sigma; \Delta; \beta; MVar; Fd \vdash_{exec} \langle Mem, stk, pc, \Phi, nalloc\rangle$
of our lemma by inversion using rule Exec-state-src then inversion using rule Well-formed program and parameters.
This concludes case Evaluate-expr-addr-module.

- **Case Evaluate-expr-var:**
  We obtain the preconditions $\mathsf{addr}(vid), \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow a'$
  and $Mem(a') = v$.
  We distinguish the following two cases:
  * **Case $v \notin \{\delta\} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$:**
    This case is vacuous.
  * **Case $v \in \{\delta\} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$:**
    Here, we know $v = (\delta, st, end, \_)$ and our goal is to show that:
    $[st, end) \subseteq \mathsf{reachable\_addresses}(\Sigma, \Delta, modIDs, Mem)$.
    We first show that $a' \in \mathsf{reachable\_addresses}(\Sigma, \Delta, modIDs, Mem)$ by distinguishing the following two cases:
    · **Case $vid \in \mathtt{localIDs}(Fd(fid)) \cup \mathtt{args}(Fd(fid))$:**
      This case is then identical to case Evaluate-expr-addr-local of our current lemma.
    · **Case $id \notin \mathtt{localIDs}(Fd(fid)) \cup \mathtt{args}(Fd(fid))$:**
      This case is then identical to case Evaluate-expr-addr-module of our current lemma.

    Now, having proved that $a' \in \mathsf{reachable\_addresses}(\Sigma, \Delta, modIDs, Mem)$,
    we distinguish by unfolding Definition 49 the following cases:
    · **Case $a' \in \mathsf{static\_addresses}(\Sigma, \Delta, modIDs)$:**
      In this case, we know by $a' \in \mathsf{dom}(Mem)$ which was obtained above that $|Mem| \geq 1$ and thus by unfolding Definitions 47 to 49 of our goal, we're done.
    · **Case $a' \in \mathsf{access}_{|Mem|}(\mathsf{static\_addresses}(\Sigma, \Delta, modIDs), Mem)$:**
      Here, by unfolding Definitions 47 and 48, we know that:
      $[st, end) \subseteq \mathsf{access}_{|Mem|+1}(\mathsf{static\_addresses}(\Sigma, \Delta, modIDs), Mem)$
      But then by Lemma 64, we conclude:
      $[st, end) \subseteq \mathsf{access}_{|Mem|}(\mathsf{static\_addresses}(\Sigma, \Delta, modIDs), Mem)$.
      The last statement by Definition 49 gives us our goal.
- **Case Evaluate-expr-addr-arr:**
  Immediate by the induction hypothesis.
- **Case Evaluate-expr-arr:**
  Similar to case Evaluate-expr-var.
- **Case Evaluate-expr-deref:**
  Similar to cases Evaluate-expr-var and Evaluate-expr-arr.
- **Case Evaluate-expr-limrange:**
  Immediate by the induction hypothesis and transitivity of $\subseteq$.

This concludes the proof of Lemma 81. □

Definition 51 (Data segment capability of a module).

$$\mathsf{data\_segment\_capability}(\Delta, modID) \stackrel{\mathsf{def}}{=} (\delta, \Delta(modID).1, \Delta(modID).2, 0)$$

Definition 52 (Stack capability of a module).

$$\mathsf{stack\_capability}(\Sigma, modID) \stackrel{\mathsf{def}}{=} (\delta, \Sigma(modID).1, \Sigma(modID).2, 0)$$

Definition 53 (Capabilities of a module)**.**

$$\begin{aligned}&\mathsf{module\_caps}(\Delta, \Sigma, \mathit{modID}) \stackrel{\mathsf{def}}{=} \\ &\{\mathsf{data\_segment\_capability}(\Delta, \mathit{modID}), \mathsf{stack\_capability}(\Sigma, \mathit{modID})\}\end{aligned}$$

Definition 54 (Static capabilities)**.**

$$\begin{aligned}&\mathsf{static\_capabilities}(\Sigma, \Delta, \mathit{modIDs}) \stackrel{\mathsf{def}}{=} \\ &\bigcup_{\mathit{modID} \in \mathit{modIDs}} \mathsf{module\_caps}(\Delta, \Sigma, \mathit{modID})\end{aligned}$$

Lemma 82 (Static addresses are precisely those of static capabilities)**.**

$$\mathsf{static\_addresses}(\Sigma, \Delta, \mathit{modIDs}) = \mathsf{addr}(\mathsf{static\_capabilities}(\Sigma, \Delta, \mathit{modIDs}))$$

Proof. Immediate by unfolding addr, Definition 54, Definition 53, Definition 52, Definition 51, and Definition 46. □

Definition 55 (Access to capabilities)**.**

$$\begin{aligned}&\mathsf{access\_cap}(C, \mathit{Mem}) \stackrel{\mathsf{def}}{=} \\ &C \cup \{(\delta, \sigma, e, 0) \mid \mathit{Mem}(a') = (\delta, \sigma, e, \_) \ \wedge \ a' \in \mathsf{addr}(C)\}\end{aligned}$$

Lemma 83 (Accessed addresses are precisely the addresses of accessed capabilities)**.**

$$\mathsf{access}(\mathsf{addr}(C), \mathit{Mem}) = \mathsf{addr}(\mathsf{access\_cap}(C, \mathit{Mem}))$$

Proof. Straightforward by unfolding addr, Definition 55, and Definition 47. □

Definition 56 (k-access to capabilities)**.**

$$\begin{aligned}&\mathsf{access\_cap}_0(C, \mathit{Mem}) \stackrel{\mathsf{def}}{=} C \\ &\mathsf{access\_cap}_{k+1}(C, \mathit{Mem}) \stackrel{\mathsf{def}}{=} \mathsf{access\_cap}(\mathsf{access\_cap}_k(C, \mathit{Mem}), \mathit{Mem})\end{aligned}$$

Lemma 84 (k-accessed addresses are precisely the addresses of k-accessed capabilities)**.**

$$\mathsf{access}_k(\mathsf{addr}(C), \mathit{Mem}) = \mathsf{addr}(\mathsf{access\_cap}_k(C, \mathit{Mem}))$$

Proof. Straightforward by induction on k; the base case is immediate then we apply Lemma 83 in the inductive case, after unfolding the goal using addr, Definition 56, and Definition 48. □

Definition 57 (Reachable capabilities)**.**

$$\begin{aligned}&\mathsf{reachable\_caps}(\Sigma, \Delta, \mathit{modIDs}, \mathit{Mem}) \stackrel{\mathsf{def}}{=} \\ &\mathsf{static\_capabilities}(\Sigma, \Delta, \mathit{modIDs}) \\ &\cup \mathsf{access\_cap}_{|\mathit{Mem}|}(\mathsf{static\_capabilities}(\Sigma, \Delta, \mathit{modIDs}), \mathit{Mem})\end{aligned}$$

Lemma 85 (Reachable addresses are precisely the addresses of the reachable capabilities)**.**

$$\begin{aligned}&\mathsf{reachable\_addresses}(\Sigma, \Delta, \mathit{modIDs}, \mathit{Mem}) = \\ &\mathsf{addr}(\mathsf{reachable\_caps}(\Sigma, \Delta, \mathit{modIDs}, \mathit{Mem}))\end{aligned}$$

Proof. By unfolding Definition 57 and Definition 49, and by applying the linearity of addr, our goal follows from Lemma 84 and Lemma 82. □

3 Compiling pointers as capabilities (ImpMod to CHERIExp)

Definition 58 (Expression Translation).

- $\wr z \int_{\_} \stackrel{\text{def}}{=} z$ *for* $z \in \mathbb{Z}$
- $\wr \mathsf{addr}(vid) \int_{\_,mid,\beta} \stackrel{\text{def}}{=} \texttt{lim}(\texttt{ddc}, \texttt{capStart}(\texttt{ddc}) + s, \texttt{capStart}(\texttt{ddc}) + e)$ *with* $\beta(vid, \bot, mid) = (s, e)$
- $\wr \mathsf{addr}(vid) \int_{fid,mid,\beta} \stackrel{\text{def}}{=} \texttt{lim}(\texttt{stc}, \texttt{capStart}(\texttt{stc}) + \texttt{capOff}(\texttt{stc}) + s, \texttt{capStart}(\texttt{stc}) + \texttt{capOff}(\texttt{stc}) + e)$ *with* $fid \neq \bot$, $\beta(vid, fid, mid) = (s, e)$
- $\wr vid \int_{fid,mid,\beta} \stackrel{\text{def}}{=} \texttt{deref}(\ \wr \mathsf{addr}(vid) \int_{fid,mid,\beta})$
- $\wr e_1 \oplus e_2 \int_{fid,mid,\beta} \stackrel{\text{def}}{=} \ \wr e_1 \int_{fid,mid,\beta} \oplus \wr e_2 \int_{fid,mid,\beta}$
- $\wr \mathsf{deref}(e) \int_{fid,mid,\beta} \stackrel{\text{def}}{=} \texttt{deref}(\wr e \int_{fid,mid,\beta})$
- $\wr \mathsf{addr}(e_{arr}[e_{off}]) \int_{fid,mid,\beta} \stackrel{\text{def}}{=} \texttt{inc}(\wr \mathsf{addr}(e_{arr}) \int_{fid,mid,\beta}, \wr e_{off} \int_{fid,mid,\beta})$
- $\wr e_{arr}[e_{off}] \int_{fid,mid,\beta} \stackrel{\text{def}}{=} \texttt{deref}(\ \wr \mathsf{addr}(e_{arr}[e_{off}]) \int_{fid,mid,\beta})$
- $\wr \mathsf{start}(e) \int_{fid,mid,\beta} \stackrel{\text{def}}{=} \texttt{capStart}(\wr e \int_{fid,mid,\beta})$
- $\wr \mathsf{end}(e) \int_{fid,mid,\beta} \stackrel{\text{def}}{=} \texttt{capEnd}(\wr e \int_{fid,mid,\beta})$
- $\wr \mathsf{offset}(e) \int_{fid,mid,\beta} \stackrel{\text{def}}{=} \texttt{capOff}(\wr e \int_{fid,mid,\beta})$
- $\wr \mathsf{capType}(e) \int_{fid,mid,\beta} \stackrel{\text{def}}{=} \texttt{capType}(\wr e \int_{fid,mid,\beta})$
- $\wr \mathsf{limRange}(e, e_s, e_e) \int_{fid,mid,\beta} \stackrel{\text{def}}{=} \texttt{lim}(\wr e \int_{fid,mid,\beta}, \wr e_s \int_{fid,mid,\beta}, \wr e_e \int_{fid,mid,\beta})$

We also define expression translation for a list of expressions as $\wr \overline{e} \int_{fid,mid,\beta} \stackrel{\text{def}}{=} \wr e_0 \int_{fid,mid,\beta} ... \wr e_{n-1} \int_{fid,mid,\beta}$ where $\overline{e} \equiv e_0 ... e_{n-1}$.

Definition 59 (Command Translation).

- $(\!| \mathsf{Assign}\ e_l\ e_r |\!)_{\_,\_,fid,mid,\beta} \stackrel{\text{def}}{=} \mathsf{Assign}\ \wr e_l \int_{fid,mid,\beta}\ \ \wr e_r \int_{fid,mid,\beta}$
- $(\!| \mathsf{Alloc}\ e_l\ e_{size} |\!)_{\_,\_,fid,mid,\beta} \stackrel{\text{def}}{=} \mathsf{Alloc}\ \wr e_l \int_{fid,mid,\beta}\ \ \wr e_{size} \int_{fid,mid,\beta}$
- $(\!| \mathsf{Call}\ fid_{call}\ \overline{e} |\!)_{Fd,\_,fid,mid,\beta} \stackrel{\text{def}}{=} \mathsf{Cinvoke}\ \texttt{moduleID}(Fd(fid_{call}))\ \ fid_{call}\ \ \wr \overline{e} \int_{fid,mid,\beta}$
- $(\!| \mathsf{Return} |\!)_{\_,\_,\_,\_,\_} \stackrel{\text{def}}{=} \mathsf{Creturn}$
- $(\!| \mathsf{JumpIfZero}\ e_c\ e_{off} |\!)_{\_,K_{fun},fid,mid,\beta} \stackrel{\text{def}}{=} \mathsf{JumpIfZero}\ \wr e_c \int_{fid,mid,\beta}\ \ \wr e_{off} \int_{fid,mid,\beta}$
- $(\!| \mathsf{Exit} |\!)_{\_,\_,\_,\_,\_} \stackrel{\text{def}}{=} \mathsf{Exit}$

Lemma 86 (Code and data segment capabilities are precise with respect to the code and data memory initializations)**.**

$$\begin{array}{l}
\forall \mathcal{M}_c, \mathcal{M}_d, imp.\ \langle \mathcal{M}_c, \mathcal{M}_d, imp, _, _ \rangle \in \mathsf{range}([\![\cdot]\!]) \\
\implies \\
\forall a.\ a \in \mathsf{dom}(\mathcal{M}_c) \iff \exists c \in \mathsf{range}(imp).\ c.1 = (\kappa, s, e, _) \wedge a \in [s, e) \wedge \\
\forall a.\ a \in \mathsf{dom}(\mathcal{M}_d) \iff \exists c \in \mathsf{range}(imp).\ c.2 = (\delta, s, e, _) \wedge a \in [s, e)
\end{array}$$

Proof. Follows from rules Module-list-translation, Module-translation and Function-translation. ☐

Figure 8: Compilation of functions, modules and module lists

(Function-translation)

$$\frac{\begin{array}{c} i_{start} = K_{mod}(mid).1 + K_{fun}(fid).1 \\ \mathcal{M}_c \;=\; \biguplus_{i \in [0, |\overline{cmd}|)} i_{start} + i \mapsto (\!|\overline{cmd}(i)|\!)_{Fd, K_{fun}, fid, mid, \beta} \end{array}}{\llbracket (mid, fid, \overline{args}, \; \overline{localvars}, \; \overline{cmd}) \rrbracket_{Fd, K_{mod}, K_{fun}, \beta} \;=\; \mathcal{M}_c}$$

(Module-translation)

$$\frac{\begin{array}{c} \mathcal{M}_c = \biguplus_{j \in [0, |\overline{fundef}|)} \llbracket \overline{fundef}(j) \rrbracket_{Fd, K_{mod}, K_{fun}, \beta} \\ \mathcal{M}_d = \{i \mapsto 0 \mid i \in \Delta(mid)\} \\ offs = \{fid \mapsto K_{fun}(fid).1 \mid fid \in \mathsf{dom}(Fd)\} \\ imp = \{mid \mapsto ((\kappa, K_{mod}(mid).1, K_{mod}(mid).2, 0), (\delta, \Delta(mid).1, \Delta(mid).2, 0), offs)\} \\ \mathsf{mstc} = \{mid \mapsto (\delta, \Sigma(mid).1, \Sigma(mid).2, 0)\} \\ \phi = \{(mid, fid) \mapsto (\mathtt{length}(\mathtt{args}(Fd(fid))), \mathtt{length}(\mathtt{localIDs}(Fd(fid)))) \mid fid \in \mathsf{dom}(Fd)\} \end{array}}{\llbracket (mid, \overline{privvars}, \overline{fundef}) \rrbracket_{Fd, \Delta, \Sigma, \beta, K_{mod}, K_{fun}} = (\mathcal{M}_c, \mathcal{M}_d, imp, \mathsf{mstc}, \phi)}$$

(Module-list-translation)

$$\frac{\begin{array}{c} \mathtt{wfp\_params}(\overline{m}, \Delta, \Sigma, \beta, K_{mod}, K_{fun}) \\ funDefs = \{modFunDef \mid modFunDef \in modFunDefs \wedge (\_, \_, modFunDefs) \in \overline{m}\} \wedge \\ Fd = \{funID \mapsto funDef \mid funDef \in funDefs \wedge funDef = (\_, funID, \_, \_, \_)\} \wedge \\ (\mathcal{M}_c, \mathcal{M}_d, imp, \mathsf{mstc}, \phi) = \biguplus_{j \in [0, |\overline{m}|)} \llbracket \overline{m}(j) \rrbracket_{Fd, \Delta, \Sigma, \beta, K_{mod}, K_{fun}} \end{array}}{\llbracket \overline{m} \rrbracket_{\Delta, \Sigma, \beta, K_{mod}, K_{fun}} = (\mathcal{M}_c, \mathcal{M}_d, imp, \mathsf{mstc}, \phi)}$$

3.1 Whole-program compiler correctness

Compiler correctness is given by Theorem 1 (backward simulation).

Definition 60 (Source-target value relatedness)**.**
Value relatedness $\cong \;\subseteq\; \mathcal{V} \times \mathcal{V}$ *is syntactic equality:*

$$\forall v.\; v \cong v$$

Lemma 87 (Expression translation forward simulation - case $\mathsf{addr}(vid)$)**.**

$$\begin{array}{l} \forall \overline{mods}, \Sigma, \Delta, \beta, MVar, Fd, Mem, stk, pc, \Phi, nalloc, mid, fid, vid, \mathcal{M}_d, \mathsf{stc}, \mathsf{ddc}. \\ pc = (fid, \_) \;\wedge\; \Delta(mid) = (\mathsf{ddc}.\sigma, \mathsf{ddc}.e) \;\wedge \\ \Sigma(mid) = (\mathsf{stc}.\sigma, \mathsf{stc}.e) \;\wedge\; \Phi(mid) = \mathsf{stc}.off \;\wedge \\ \_;\_;\overline{mods}; \Sigma; \Delta; \beta; MVar; Fd \vdash_{exec} \langle Mem, stk, pc, \Phi, nalloc \rangle \;\wedge \\ \exists v.\; \mathsf{addr}(vid), \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow v \\ \;\Longrightarrow\; \exists v.\; \wr\, \mathsf{addr}(vid) \,\int_{fid, mid, \beta}, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \_ \Downarrow v \wedge v \cong v \end{array}$$

Proof.

- From the assumption $\exists v.\; \mathsf{addr}(vid), \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow v$, there are two cases for inversion:

  Case Evaluate-expr-addr-local:
  Here, we have by inversion:

  1. $vid \in \mathtt{localIDs}(Fd(fid)) \cup \mathtt{args}(Fd(fid))$

2. $mid = \texttt{moduleID}(Fd(fid))$
3. $\beta(vid, fid, mid) = [s, e)$
4. $\phi = \Sigma(mid).1 + \Phi(mid)$
5. $v = (\delta, \phi + s, \phi + e, 0)$

Thus, by value relatedness, we would like to show that:
$\wr \mathsf{addr}(vid) \int_{fid,mid,\beta}, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \_ \Downarrow (\delta, \phi + s, \phi + e, 0)$.
We know that $\bot \notin \mathsf{dom}(Fd)$. Thus, we conclude $fid \neq \bot$, which by Definition 58 gives us:
$\wr \mathsf{addr}(vid) \int_{fid,mid,\beta} = \texttt{lim}(\texttt{stc}, \texttt{capStart}(\texttt{stc}) + \texttt{capOff}(\texttt{stc}) + s, \texttt{capStart}(\texttt{stc}) + \texttt{capOff}(\texttt{stc}) + e)$.
By substitution, our goal becomes:
$\texttt{lim}(\texttt{stc}, \texttt{capStart}(\texttt{stc}) + \texttt{capOff}(\texttt{stc}) + s, \texttt{capStart}(\texttt{stc}) + \texttt{capOff}(\texttt{stc}) + e), \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \_ \Downarrow$ $(\delta, \phi + s, \phi + e, 0)$.
By applying evalLim, (and evalstc to some of the subgoals), we obtain three subgoals:

- $\texttt{capStart}(\texttt{stc}) + \texttt{capOff}(\texttt{stc}) + s, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \_ \Downarrow \phi + s$
- $\texttt{capStart}(\texttt{stc}) + \texttt{capOff}(\texttt{stc}) + e, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \_ \Downarrow \phi + e$
- $[\phi + s, \phi + e) \subseteq [\mathsf{stc}.\sigma, \mathsf{stc}.e)$

These subgoals become (by further applying evalBinOp, evalCapStart, evalCapOff and evalstc and substitution using the definition of ϕ, s and e given above):

- $\mathsf{stc}.\sigma + \mathsf{stc}.\mathit{off} = \Sigma(mid).1 + \Phi(mid)$
- $[\Sigma(mid).1 + \Phi(mid) + \beta(vid, fid, mid).1, \Sigma(mid).1 + \Phi(mid) + \beta(vid, fid, mid).2)$ $\subseteq [\mathsf{stc}.\sigma, \mathsf{stc}.e)$

The first subgoal holds immediately by reflexivity after substitution from the assumptions of our lemma.
The second subgoal after substitution becomes:
$[\Sigma(mid).1 + \Phi(mid) + \beta(vid, fid, mid).1, \Sigma(mid).1 + \Phi(mid) + \beta(vid, fid, mid).2)$
$\subseteq [\Sigma(mid).1, \Sigma(mid).2)$.

In order to prove this goal, we invert the assumption
$\_; \_; \overline{mods}; \Sigma; \Delta; \beta; MVar; Fd \vdash_{exec} \langle Mem, stk, pc, \Phi, nalloc\rangle$
using rule Exec-state-src then we invert $\texttt{wfp\_params}(\overline{mods}, \Delta, \Sigma, \beta, K_{mod}, K_{fun})$ using rule Well-formed program and parameters to obtain the following:

No stack overflow
$\forall mid \in modIDs.\ \Sigma(mid).1 + \Phi(mid) \leq \Sigma(mid).2$

Frame sizes are non-negative
$\forall fid \in \mathsf{dom}(Fd).\ \texttt{frameSize}(Fd(fid)) \geq 0$

Stack pointers are the sum of all frame sizes on stack
$\forall mid \in modIDs.\ \Phi(mid) =$
$$\sum_{fid\ \in\ \{fid\ \mid\ \texttt{moduleID}(Fd(fid))\ =\ mid\}} \texttt{frameSize}(Fd(fid)) \times (\texttt{countIn}((fid, \_), stk) + (pc = (fid, \_)\ ?\ 1\ :\ 0))$$

Variables occupy exactly the frame
$\forall mid \in modIDs, fid \in \mathsf{dom}(Fd).$
$$\biguplus_{vid \in \texttt{localIDs}(Fd(fid)) \cup \texttt{args}(Fd(fid))} \beta(vid, fid, mid) = [-\texttt{frameSize}(Fd(fid)), 0)$$

Now, by substituting the assumption:
$pc = (fid, \_)$
of our lemma into statement:
Stack pointers are the sum of all frame sizes on stack
instantiated with the assumption (obtained above by inversion):
$mid = \texttt{moduleID}(Fd(fid))$,

together with the constraint:
Frame sizes are non-negative,
we can conclude that:
$\Phi(mid) \geq \texttt{frameSize}(Fd(fid))$.
The latter statement, together with:
Variables occupy exactly the frame
suffice to show that:
$\Sigma(mid).1 + \Phi(mid) + \beta(vid, fid, mid).1 \geq \Sigma(mid).1$.

Thus, it remains to show that:
$\Sigma(mid).1 + \Phi(mid) + \beta(vid, fid, mid).2 \leq \Sigma(mid).2$.
We already know:
$\Sigma(mid).1 + \Phi(mid) \leq \Sigma(mid).2$
by **"No stack overflow"**.
And we know:
$\beta(vid, fid, mid).2 < 0$
by **"Variables occupy exactly the frame"**.
So, we immediately have the desired inequality by arithmetic identities.
This proves the second subgoal, and concludes case Evaluate-expr-addr-local.

Case Evaluate-expr-addr-module:
Here, we have by inversion:

1. $vid \notin \texttt{localIDs}(Fd(fid)) \cup \texttt{args}(Fd(fid))$
2. $mid = \texttt{moduleID}(Fd(fid))$
3. $vid \in MVar(mid)$
4. $\beta(vid, \perp, mid) = [s, e)$
5. $v = (\delta, \Delta(mid).1 + s, \Delta(mid).1 + e, 0)$

Thus, by value relatedness, we would like to show that:
$\wr \mathsf{addr}(vid) \int_{fid,mid,\beta}, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \_ \Downarrow (\delta, \Delta(mid).1 + s, \Delta(mid).1 + e, 0)$.
Here, by the precondition $\beta(vid, \perp, mid) = (s, e)$, we know by Definition 58 that:
$\wr \mathsf{addr}(vid) \int_{fid,mid,\beta} = \texttt{lim}(\mathsf{ddc}, \texttt{capStart}(\mathsf{ddc}) + s, \texttt{capStart}(\mathsf{ddc}) + e)$

Thus, substituting this into our goal, our goal becomes:
$\texttt{lim}(\mathsf{ddc}, \texttt{capStart}(\mathsf{ddc}) + s, \texttt{capStart}(\mathsf{ddc}) + e), \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \_ \Downarrow$
$(\delta, \Delta(mid).1 + s, \Delta(mid).1 + e, 0)$

By applying evalLim, we obtain three subgoals:

- $\texttt{capStart}(\mathsf{ddc}) + s, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \_ \Downarrow \Delta(mid).1 + s$
- $\texttt{capStart}(\mathsf{ddc}) + e, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \_ \Downarrow \Delta(mid).1 + e$
- $[\Delta(mid).1 + s, \Delta(mid).1 + e) \subseteq [\mathsf{ddc}.\sigma, \mathsf{ddc}.e)$

For each of the first two subgoals, we apply evalBinOp and evalCapStart to end up with the following subgoal instead:
$\mathsf{ddc}.\sigma = \Delta(mid).1$
which is immediate by our lemma's assumptions.

For the third subgoal, by substitution from the assumptions, we obtain the following subgoal instead:
$[\Delta(mid).1 + s, \Delta(mid).1 + e) \subseteq [\Delta(mid).1, \Delta(mid).2)$

To prove this subgoal, we invert the assumption:
$\overline{mods}; \Sigma; \Delta; \beta; MVar; Fd \vdash_{exec} \langle Mem, stk, pc, \Phi, nalloc \rangle$
using rule Exec-state-src then by inversion using rule Well-formed program and parameters, we obtain:

Static variables occupy exactly the data segment

$$\forall mid \in modIDs. \biguplus_{vid \in MVar(mid)} \beta(vid, \perp, mid) = [0, \Delta(mid).2 - \Delta(mid).1)$$

from which we conclude:
$[s, e) \subseteq [0, \Delta(mid).2 - \Delta(mid).1)$.
In this last statement, by adding $\Delta(mid).1$ to both components of the intervals on each side, we immediately obtain our goal.
This concludes case Evaluate-expr-addr-module.

This concludes the proof of Lemma 87. □

Lemma 88 (Expression translation forward simulation)**.**

$$\begin{aligned}
&\forall \overline{mods}, \Sigma, \Delta, \beta, MVar, Fd, Mem, stk, pc, \Phi, nalloc, mid, fid, vid, \mathcal{M}_d, \mathsf{stc}, \mathsf{ddc}.\\
&pc = (fid, _) \ \wedge\ \Delta(mid) = (\mathsf{ddc}.\sigma, \mathsf{ddc}.e) \ \wedge\\
&\Sigma(mid) = (\mathsf{stc}.\sigma, \mathsf{stc}.e) \ \wedge\ \Phi(mid) = \mathsf{stc}.off \ \wedge\\
&imp(mid).\mathsf{ddc} \doteq \mathsf{ddc} \ \wedge\ \mathsf{mstc}(mid) \doteq \mathsf{stc} \wedge\\
&_; _; \overline{mods}; \Sigma; \Delta; \beta; MVar; Fd \vdash_{exec} \langle Mem, stk, pc, \Phi, nalloc \rangle \ \wedge\\
&\texttt{moduleID}(Fd(fid)) \in modIDs \wedge\\
&A_s = \mathsf{reachable_addresses}(\Sigma, \Delta, modIDs, Mem) \wedge\\
&A_t = \mathsf{reachable_addresses}(\bigcup_{mid \in modIDs} \{imp(mid).\mathsf{ddc}, \mathsf{mstc}(mid)\}, \mathcal{M}_d) \wedge\\
&A_s = A_t \ \wedge\ Mem|_{A_s} = \mathcal{M}_d|_{A_t} \ \wedge\\
&\exists v.\ e, \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow v\\
&\implies \exists v.\ \wr\, e \,\int_{fid,mid,\beta}, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, _ \Downarrow v \wedge v \cong v
\end{aligned}$$

Proof.
Our goal by Definition 60 is:

$$\wr e \int_{fid,mid,\beta}, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \_ \Downarrow v$$

We assume the antecedents and prove it by induction on the evaluation of the source expression e.

Case Evaluate-expr-const:
By substitution in Definition 58, our goal becomes:
$z, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \_ \Downarrow z$
which is immediate by evalconst.

Case Evaluate-expr-cast-to-integer-start:
By substitution in Definition 58, our goal becomes:
$\texttt{capStart}(\wr\, e' \int_{fid,mid,\beta}), \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \_ \Downarrow z$
where we have the induction hypothesis:
$\wr e' \int_{fid,mid,\beta}, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \_ \Downarrow (\_, z, \_, \_)$.
Thus, by applying rule evalCapStart to the goal, we can immediately show our goal using the induction hypothesis on e'.

Case Evaluate-expr-cast-to-integer-end:

By substitution in Definition 58, our goal becomes:
$\texttt{capEnd}(\wr e' \int_{\mathit{fid},\mathit{mid},\beta}), \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \_ \Downarrow z$

where we have the induction hypothesis:
$\wr e' \int_{\mathit{fid},\mathit{mid},\beta}, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \_ \Downarrow (\_, \_, z, \_)$.

Thus, by applying rule evalCapEnd to the goal, we can immediately show our goal using the induction hypothesis on e'.

Case Evaluate-expr-cast-to-integer-offset:

By substitution in Definition 58, our goal becomes:
$\texttt{capOff}(\wr e' \int_{\mathit{fid},\mathit{mid},\beta}), \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \_ \Downarrow z$

where we have the induction hypothesis:
$\wr e' \int_{\mathit{fid},\mathit{mid},\beta}, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \_ \Downarrow (\_, \_, \_, z)$.

Thus, by applying rule evalCapOff to the goal, we can immediately show our goal using the induction hypothesis on e'.

Case Evaluate-expr-cap-type:

By substitution in Definition 58, our goal becomes:
$\texttt{capType}(\wr e' \int_{\mathit{fid},\mathit{mid},\beta}), \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \_ \Downarrow v$

where we have the induction hypothesis:
$\wr e' \int_{\mathit{fid},\mathit{mid},\beta}, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \_ \Downarrow (x, \_, \_, \_)$.

and the assumptions:
$x = \kappa \implies v = 0$ and $x = \delta \implies v = 1$.

Thus, by applying rule evalCapType to the goal, we can immediately show our goal using the induction hypothesis on e' and the assumptions on x and v.

Case Evaluate-expr-binop:

By substitution in Definition 58, our goal becomes:
$\wr e_1 \int_{\mathit{fid},\mathit{mid},\beta} \oplus \wr e_2 \int_{\mathit{fid},\mathit{mid},\beta}, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \_ \Downarrow z$

where we have the induction hypotheses:
$\wr e_1 \int_{\mathit{fid},\mathit{mid},\beta}, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \_ \Downarrow z_1$ and

$\wr e_2 \int_{\mathit{fid},\mathit{mid},\beta}, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \_ \Downarrow z_2$

and the assumption:
$z = z_1[\oplus]z_2$.

Thus, by applying rule evalBinOp to the goal, we can immediately show our generated subgoals using the induction hypotheses on e_1 and e_2 and the assumption on z, z_1 and z_2.

Case Evaluate-expr-addr-local and

Case Evaluate-expr-addr-module:

These two cases are proved by Lemma 87.

Case Evaluate-expr-var:

By substitution in Definition 58, our goal becomes:
$\texttt{deref}(\wr \mathsf{addr}(\mathit{vid}) \int_{\mathit{fid},\mathit{mid},\beta}), \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \_ \Downarrow v$

And we have the assumptions:

- $\mathsf{addr}(\mathit{vid}), \Sigma, \Delta, \beta, \mathit{MVar}, \mathit{Fd}, \mathit{Mem}, \Phi, \mathit{pc} \Downarrow (\delta, s, e, \mathit{off})$
- $s \leq s + \mathit{off} < e$
- $\mathit{Mem}(s + \mathit{off}) = v$

By Lemma 87, we have that:
$\wr \mathsf{addr}(vid) \int_{fid,mid,\beta}, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \_ \Downarrow (\delta, s, e, \mathit{off})$

Thus, from the assumption $s + \mathit{off} < e$, it follows by substitution that $s \leq s + \mathit{off} < e$.

Applying rule evalDeref to our goal, we get the following goals:

1. $s \leq s + \mathit{off} < e$ which is immediate.
2. $\mathcal{M}_d(s + \mathit{off}) = v$

For the latter goal, we notice first from assumptions:
$\mathsf{addr}(vid), \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow (\delta, s, e, \mathit{off})$ and
$s \leq s + \mathit{off} < e$
and by Lemma 81 that:
$s + \mathit{off} \in \mathsf{reachable\_addresses}(\Sigma, \Delta, modIDs, Mem) = A_s$

And, also by the definition of the value $(\delta, s, e, \mathit{off})$, we have using Lemma 25 that:
$s + \mathit{off} \in \mathsf{reachable\_addresses}(\{\mathsf{ddc}, \mathsf{stc}\}, \mathcal{M}_d)$.

Applying Lemma 18, and using the assumptions $imp(mid).\mathsf{ddc} \doteq \mathsf{ddc}$, and $\mathsf{mstc}(mid) \doteq \mathsf{stc}$, we hence conclude that:
$s + \mathit{off} \in A_t$. (Here, we used Lemmas 6 and 18, and a little hand-waving to prove that the offsets of both ddc and stc do not affect the function reachable_addresses.)

So, by assumptions
$A_s = A_t$ and $Mem|_{A_s} = \mathcal{M}_d|_{A_t}$,
we conclude:
$\mathcal{M}_d(s + \mathit{off}) = Mem(s + \mathit{off})$

This last statement together with assumption $Mem(s + \mathit{off}) = v$ immediately prove our remaining goal.

Case Evaluate-expr-addr-arr:
By substitution in Definition 58, our goal becomes:
$\mathtt{inc}(\wr \mathsf{addr}(e_{arr}) \int_{fid,mid,\beta}, \wr e_{\mathit{off}} \int_{fid,mid,\beta}), \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \_ \Downarrow (\delta, s, e, \mathit{off} + \mathit{off}')$

with the induction hypotheses and assumptions:

- $\mathit{off}' \in \mathbb{Z}$
- $\wr \mathsf{addr}(e_{arr}) \int_{fid,mid,\beta}, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \_ \Downarrow (\delta, s, e, \mathit{off})$
- $\wr e_{\mathit{off}} \int_{fid,mid,\beta}, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \_ \Downarrow \mathit{off}'$

Thus, by applying rule evalIncCap to our goal, we get five subgoals which are immediately satisfiable by our induction hypotheses and assumptions.

Case Evaluate-expr-arr:
By substitution in Definition 58, our goal becomes:
$\mathtt{deref}(\wr \mathsf{addr}(e_{arr}[e_{\mathit{off}}]) \int_{fid,mid,\beta}), \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \_ \Downarrow v$

with the assumptions:

- $\mathsf{addr}(e_{arr}[e_{\mathit{off}}]), \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow (\delta, s, e, \mathit{off})$
- $s \leq s + \mathit{off} < e$
- $Mem(s + \mathit{off}) = v$

From the first assumption using an argument exactly the same as case Evaluate-expr-addr-arr, we conclude that:
$\wr \mathsf{addr}(e_{arr}[e_{\mathit{off}}]) \int_{fid,mid,\beta}, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \_ \Downarrow (\delta, s, e, \mathit{off})$

Thus, by applying rule evalDeref to our goal, we obtain three subgoals.
Two of them are immediate by our conclusions so far (after unfolding $\vdash_\delta (\delta, s, e, \mathit{off})$ using Definition 2).
The subgoal $\mathcal{M}_d(s + \mathit{off}) = v$ is proved by using the assumptions:
$A_s = A_t$ and $\mathit{Mem}|_{A_s} = \mathcal{M}_d|_{A_t}$,
and Lemma 81 as in case Evaluate-expr-var.

Case Evaluate-expr-deref:

By substitution in Definition 58, our goal becomes:
$\texttt{deref}(\wr e \int_{\mathit{fid},\mathit{mid},\beta}), \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \_ \Downarrow v$

with the induction hypothesis and assumptions:

- $\wr e \int_{\mathit{fid},\mathit{mid},\beta}, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \_ \Downarrow (\delta, s, e, \mathit{off})$
- $s \leq s + \mathit{off} < e$
- $\mathit{Mem}(s + \mathit{off}) = v$

Thus, by applying rule evalDeref to our goal, we obtain three subgoals.
Two of them are immediate by our conclusions so far (after unfolding $\vdash_\delta (\delta, s, e, \mathit{off})$ using Definition 2).
The subgoal $\mathcal{M}_d(s + \mathit{off}) = v$ is proved by using the assumptions:
$A_s = A_t$ and $\mathit{Mem}|_{A_s} = \mathcal{M}_d|_{A_t}$,
and Lemmas 18, 25 and 81 as in case Evaluate-expr-var.

Case Evaluate-expr-limrange:

By substitution in Definition 58, our goal becomes:
$\texttt{lim}(\wr e \int_{\mathit{fid},\mathit{mid},\beta}, \wr e_s \int_{\mathit{fid},\mathit{mid},\beta}, \wr e_e \int_{\mathit{fid},\mathit{mid},\beta}), \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \_ \Downarrow (x, s', e', \mathit{off})$

with the induction hypotheses and assumptions:

- $\wr e \int_{\mathit{fid},\mathit{mid},\beta}, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \_ \Downarrow (x, s, e, \mathit{off})$
- $\wr e_s \int_{\mathit{fid},\mathit{mid},\beta}, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \_ \Downarrow s'$
- $\wr e_e \int_{\mathit{fid},\mathit{mid},\beta}, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \_ \Downarrow e'$
- $[s', e') \subseteq [s, e)$

Thus, by applying rule evalLim to our goal, we obtain four subgoals which are immediate by our four assumptions/hypotheses above.

This concludes the proof of Lemma 88. □

Lemma 89 (Expression translation backward simulation - case $\mathsf{addr}(\mathit{vid})$)**.**

$$
\begin{aligned}
&\forall \overline{\mathit{mods}}, \Sigma, \Delta, \beta, \mathit{MVar}, \mathit{Fd}, \mathit{Mem}, \mathit{stk}, \mathit{pc}, \Phi, \mathit{nalloc}, \mathit{mid}, \mathit{fid}, \mathit{vid}, \mathcal{M}_d, \mathsf{stc}, \mathsf{ddc}.\\
&\mathit{pc} = (\mathit{fid}, \_) \ \wedge\ \Delta(\mathit{mid}) = (\mathsf{ddc}.\sigma, \mathsf{ddc}.e) \ \wedge\\
&\Sigma(\mathit{mid}) = (\mathsf{stc}.\sigma, \mathsf{stc}.e) \ \wedge\ \Phi(\mathit{mid}) = \mathsf{stc}.\mathit{off} \ \wedge\\
&\_;\_;\overline{\mathit{mods}}; \Sigma; \Delta; \beta; \mathit{MVar}; \mathit{Fd} \vdash_{\mathit{exec}} \langle \mathit{Mem}, \mathit{stk}, \mathit{pc}, \Phi, \mathit{nalloc}\rangle \ \wedge\\
&\_ \vdash_{\mathit{exec}} \langle \mathcal{M}_c, \mathcal{M}_d, \mathit{stk}, \mathit{imp}, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle \wedge\\
&\exists v.\ \wr \mathsf{addr}(\mathit{vid}) \int_{\mathit{fid},\mathit{mid},\beta}, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \_ \Downarrow v\\
&\quad \Longrightarrow\ \exists v.\ \mathsf{addr}(\mathit{vid}), \Sigma, \Delta, \beta, \mathit{MVar}, \mathit{Fd}, \mathit{Mem}, \Phi, \mathit{pc} \Downarrow v \wedge v \cong v
\end{aligned}
$$

Proof.
We assume the antecedents, and by Definition 58, we consider the following two cases:

- **Case** $\beta(vid, fid, mid) = (s, e)$**:**

  In this case, we know, by Definition 58 and by assumption that:
  $\texttt{lim}(\texttt{stc}, \texttt{capStart}(\texttt{stc}) + \texttt{capOff}(\texttt{stc}) + s, \texttt{capStart}(\texttt{stc}) + \texttt{capOff}(\texttt{stc}) + e), \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \_ \Downarrow v$

  Thus, by rule evalLim, we have (ANTECS-evalLim):
  $\texttt{stc}, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow v'$,
  $\texttt{capStart}(\texttt{stc}) + \texttt{capOff}(\texttt{stc}) + s, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow s'$,
  $\texttt{capStart}(\texttt{stc}) + \texttt{capOff}(\texttt{stc}) + e, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow e'$,
  $s' \in \mathbb{Z}$,
  $e' \in \mathbb{Z}$,
  $v' = (x, s, e, \_) \in Cap$,
  $[s', e') \subseteq [s, e)$, and
  $v = (x, s', e', 0)$

  Thus, by applying rules evalCapStart, evalCapOff, and evalstc to the first three statements of (ANTECS-evalLim), we conclude by substitution from the assumption that:
  $v = (\delta, \Sigma(mid).1 + \Phi(mid) + s, \Sigma(mid).1 + \Phi(mid) + e, 0)$

  Thus, our goal is to show that:
  $\texttt{addr}(vid), \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow (\delta, \Sigma(mid).1 + \Phi(mid) + s, \Sigma(mid).1 + \Phi(mid) + e, 0)$

  By rule Evaluate-expr-addr-local, it suffices to show that:
  $vid \in \texttt{localIDs}(Fd(fid)) \cup \texttt{args}(Fd(fid))$

  This follows from the case condition $\beta(vid, fid, mid) = (s, e)$ together with assumption $\_; \_; \overline{mods}; \Sigma; \Delta; \beta; MVar; Fd \vdash_{exec} \langle Mem, stk, pc, \Phi, nalloc\rangle$ after inversion using rule Exec-state-src then rule Well-formed program and parameters and Well-formed program.

- **Case** $\beta(vid, \bot, mid) = (s, e)$**:**

  In this case, we know, by Definition 58 and by assumption that:
  $\texttt{lim}(\texttt{ddc}, \texttt{capStart}(\texttt{ddc}) + s, \texttt{capStart}(\texttt{ddc}) + e), \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \_ \Downarrow v$

  Thus, by rule evalLim, we have (ANTECS-evalLim-2):
  $\texttt{stc}, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow v'$,
  $\texttt{capStart}(\texttt{ddc}) + s, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow s'$,
  $\texttt{capStart}(\texttt{ddc}) + e, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow e'$,
  $s' \in \mathbb{Z}$,
  $e' \in \mathbb{Z}$,
  $v' = (x, s, e, \_) \in Cap$,
  $[s', e') \subseteq [s, e)$, and
  $v = (x, s', e', 0)$

  Thus, by applying rules evalCapStart, and evalddc to the first three statements of (ANTECS-evalLim-2), we conclude by substitution from the assumption that:
  $v = (\delta, \Delta(mid).1 + s, \Delta(mid).1 + e, 0)$

  Thus, our goal is to show that:
  $\texttt{addr}(vid), \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow (\delta, \Delta(mid).1 + s, \Delta(mid).1 + e, 0)$

  By rule Evaluate-expr-addr-module, it suffices to show that:
  $vid \notin \texttt{localIDs}(Fd(fid)) \cup \texttt{args}(Fd(fid))$

  This follows from the side condition $\beta(vid, \bot, mid) = (s, e)$ together with assumption $\_; \_; \overline{mods}; \Sigma; \Delta; \beta; MVar; Fd \vdash_{exec} \langle Mem, stk, pc, \Phi, nalloc\rangle$ after inversion using rule Exec-state-src then rule Well-formed program and parameters and Well-formed program.

This concludes the proof of Lemma 89. □

Lemma 90 (Expression translation backward simulation)**.**

$$\begin{array}{l}
\forall \overline{mods}, \Sigma, \Delta, \beta, MVar, Fd, Mem, stk, pc, \Phi, nalloc, mid, fid, vid, \mathcal{M}_d, \mathsf{stc}, \mathsf{ddc}.\\
pc = (fid, _) \ \wedge \ \Delta(mid) = (\mathsf{ddc}.\sigma, \mathsf{ddc}.e) \ \wedge\\
\Sigma(mid) = (\mathsf{stc}.\sigma, \mathsf{stc}.e) \ \wedge \ \Phi(mid) = \mathsf{stc}.\mathit{off} \ \wedge\\
imp(mid).\mathsf{ddc} \doteq \mathsf{ddc} \ \wedge \ \mathsf{mstc}(mid) \doteq \mathsf{stc} \wedge\\
_; _; \overline{mods}; \Sigma; \Delta; \beta; MVar; Fd \vdash_{exec} \langle Mem, stk, pc, \Phi, nalloc\rangle \ \wedge\\
_ \vdash_{exec} \langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle \wedge\\
\texttt{moduleID}(Fd(fid)) \in modIDs \wedge\\
A_s = \mathsf{reachable_addresses}(\Sigma, \Delta, modIDs, Mem) \wedge\\
A_t = \mathsf{reachable_addresses}(\bigcup_{mid \in modIDs} \{imp(mid).\mathsf{ddc}, \mathsf{mstc}(mid)\}, \mathcal{M}_d) \wedge\\
A_s = A_t \ \wedge \ Mem|_{A_s} = \mathcal{M}_d|_{A_t} \ \wedge\\
\exists v. \ \wr e \int_{fid,mid,\beta}, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, _ \Downarrow v\\
\quad \implies \exists v. \ e, \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow v \wedge v \cong v
\end{array}$$

Proof.
We assume the antecedents and prove our goal by induction on the expression evaluation $\wr e \int_{fid,mid,\beta}, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \_ \Downarrow v$.

Case evalconst:

Here, $\wr e \int_{fid,mid,\beta} = z$.

By Definition 58, we thus know $e = z$.

Thus, by rule Evaluate-expr-const, we have our goal.

Case evalddc:

Here, $\wr e \int_{fid,mid,\beta} = \texttt{ddc}$.

By Definition 58, we thus know this case is impossible.

Case evalstc:

Here, $\wr e \int_{fid,mid,\beta} = \texttt{stc}$.

By Definition 58, we thus know this case is impossible.

Case evalCapType:

Here, $\wr e \int_{fid,mid,\beta} = \texttt{capType}(\mathcal{E}')$,
with $\mathcal{E}', \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \_ \Downarrow v'$,
and by Definition 58, we know:
$\exists e'. \ e = \mathsf{capType}(e') \ \wedge \ \mathcal{E}' = \wr e' \int_{fid,mid,\beta}$.

Thus, by the induction hypothesis, we know (IH):

$e', \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow v'$.

Now, we consider the following cases:

- **Case** $v' \in \mathbb{Z}$**:**
  In this case, our goal is: $e, \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow 0$.
  But this is immediate by (IH), and rule Evaluate-expr-cap-type.
- **Case** $v' \in \{\kappa\} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$**:**
  In this case, our goal is: $e, \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow 1$.
  But this is immediate by (IH), and rule Evaluate-expr-cap-type.

- **Case** $v' \in \{\delta\} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$**:**
  In this case, our goal is: $e, \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow 2$.
  But this is immediate by (IH), and rule Evaluate-expr-cap-type.

Case evalCapStart:

Here, $\wr e \int_{fid,mid,\beta} = \texttt{capStart}(\mathcal{E}')$,

with $\mathcal{E}', \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \_ \Downarrow v'$,
and by Definition 58, we know:
$\exists e'.\ e = \mathsf{start}(e') \ \wedge \ \mathcal{E}' = \wr e' \int_{fid,mid,\beta}$.

Thus, by the induction hypothesis, we know (IH):

$e', \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow v'$.

Our goal is thus immediate by (IH) and rule Evaluate-expr-cast-to-integer-start.

Case evalCapEnd:

Here, $\wr e \int_{fid,mid,\beta} = \texttt{capEnd}(\mathcal{E}')$,

with $\mathcal{E}', \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \_ \Downarrow v'$,
and by Definition 58, we know:
$\exists e'.\ e = \mathsf{end}(e') \ \wedge \ \mathcal{E}' = \wr e' \int_{fid,mid,\beta}$.

Thus, by the induction hypothesis, we know (IH):

$e', \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow v'$.

Our goal is thus immediate by (IH) and rule Evaluate-expr-cast-to-integer-end.

Case evalCapOff:

Here, $\wr e \int_{fid,mid,\beta} = \texttt{capOff}(\mathcal{E}')$,

with $\mathcal{E}', \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \_ \Downarrow v'$,
and by Definition 58, we know:
$\exists e'.\ e = \mathsf{offset}(e') \ \wedge \ \mathcal{E}' = \wr e' \int_{fid,mid,\beta}$.

Thus, by the induction hypothesis, we know (IH):

$e', \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow v'$.

Our goal is thus immediate by (IH) and rule Evaluate-expr-cast-to-integer-offset.

Case evalBinOp:

By rule evalBinOp and by Definition 58, we know $e = e_1 \oplus e_2$, so we know:
$\wr e \int_{fid,mid,\beta} = \wr e_1 \oplus e_2 \int_{fid,mid,\beta} = \wr e_1 \int_{fid,mid,\beta} \oplus \wr e_2 \int_{fid,mid,\beta}$,
$\wr e_1 \int_{fid,mid,\beta}, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \_ \Downarrow v_1$, and
$\wr e_2 \int_{fid,mid,\beta}, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \_ \Downarrow v_2$.

Thus, by the induction hypothesis, we know (IH1):
$e_1, \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow v_1$,
and (IH2):
$e_2, \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow v_2$

Thus, our goal is immediate by (IH1), (IH2), and rule Evaluate-expr-binop.

Case evalIncCap:

Here, by Definition 58, we know:
$\wr e \int_{fid,mid,\beta} = \wr \mathsf{addr}(e_{arr}[e_{off}]) \int_{fid,mid,\beta} = \texttt{inc}(\wr e_{arr} \int_{fid,mid,\beta}, \wr e_{off} \int_{fid,mid,\beta})$

And by rule evalIncCap, we know:

$\wr e_{arr} \int_{fid,mid,\beta}, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow v \in Cap$, and
$\wr e_{off} \int_{fid,mid,\beta}, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow v_z \in \mathbb{Z}$

By the induction hypothesis, we thus know (IH1):
$e_{arr}, \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow v$,
and (IH2):
$e_{off}, \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow v_z$

Our goal is to show that:
$\mathsf{addr}(e_{arr}[e_{off}]), \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow (\delta, v.\sigma, v.e, v.off + v_z)$,
This is immediate by rule Evaluate-expr-addr-arr.

Case evalDeref:

By rule evalDeref, we know (DEREF-ASSMS):
$\mathcal{E}', \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow v'$, $\vdash_\delta v'$, and $v = \mathcal{M}_d(v'.\sigma + v'.off)$
Our goal is to show that:
$e, \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow v$
By Definition 58, we distinguish the following cases:

- **Case $e = \mathsf{deref}(e')$:**
  Here, by Definition 58, we also know:
  $\wr e' \int_{fid,mid,\beta} = \mathcal{E}'$
  Thus, together, with the assumption above, we have by the induction hypothesis that:
  $e', \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow v'$
  By rule Evaluate-expr-deref, we thus have the following two subgoals:
  - $v'.\sigma \le v'.\sigma + v'.off < v'.e$
    This is immediate by (DEREF-ASSMS)'s $\vdash_\delta v'$ (unfolding Definition 2).
  - $Mem(v'.\sigma + v'.off) = v$
    Here, by (DEREF-ASSMS)'s $v = \mathcal{M}_d(v'.\sigma + v'.off)$, and $\vdash_\delta v'$, and the antecedents, it suffices to show that:
    $v'.\sigma + v'.off \in A_s$.
    This is immediate by Lemma 81.
- **Case $e = vid$:**
  By inverting our goal using rule Evaluate-expr-var, we obtain the following subgoals:
  - $\mathsf{addr}(vid), \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow (\delta, s, e, off)$
    By Definition 58, we know:
    $\mathcal{E}' = \wr \mathsf{addr}(vid) \int_{fid,mid,\beta}$
    Thus, by Lemma 89, we know (ADDR-EVAL):
    $\mathsf{addr}(vid), \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow v'$
    which together with the knowledge of $\vdash_\delta v'$ (DEREF-ASSMS) immediately satisfy our subgoal.
  - $v'.\sigma \le v'.\sigma + v'.off < v'.e$
    Immediate by $\vdash_\delta v'$ (unfolding Definition 2).
  - $Mem(v'.\sigma + v'.off) = v$
    Here, by (DEREF-ASSMS)'s $v = \mathcal{M}_d(v'.\sigma + v'.off)$, and $\vdash_\delta v'$, and the antecedents, it suffices to show that:
    $v'.\sigma + v'.off \in A_s$.
    This is immediate by Lemma 81.
- **Case $e = e_{arr}[e_{off}]$:**
  Here, by Definition 58, we have:
  $\mathcal{E}' = \wr \mathsf{addr}(e_{arr}[e_{off}]) \int_{fid,mid,\beta} = \mathtt{inc}(\ \wr\ e_{arr} \int_{fid,mid,\beta}\ ,\ \wr\ e_{off} \int_{fid,mid,\beta}\ )$
  Thus, by (DEREF-ASSMS), and inversion using rule evalIncCap, we obtain (INC-ASSMS):
  $\wr e_{arr} \int_{fid,mid,\beta}\ , \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \_ \Downarrow (\delta, \sigma_a, e_a, off_a)$,
  $\wr e_{off} \int_{fid,mid,\beta}\ , \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \_ \Downarrow v_z \in \mathbb{Z}$, and
  $v'.off = off_a + v_z$

By the induction hypothesis (instantiated with (INC-ASSMS)), we thus have (IH-E-ARR):
$e_{arr}, \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow (\delta, \sigma_a, e_a, \mathit{off}_a)$,
and (IH-E-OFF):
$e_{\mathit{off}}, \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow v_z \in \mathbb{Z}$,
By inverting our goal using rule Evaluate-expr-arr, we obtain the following subgoals:

- $\mathsf{addr}(e_{arr}[e_{idx}]), \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow v'$
  By inversion using rule Evaluate-expr-addr-arr, we obtain the following subgoals:
  * $e_{arr}, MVar, \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow (\delta, \sigma_a, e_a, \mathit{off}_a)$
    Immediate by (IH-E-ARR).
  * $e_{\mathit{off}}, \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow v_z$, and
  * $v_z \in \mathbb{Z}$
    Immediate by (IH-E-OFF).
- $v'.\sigma \leq v'.\sigma + v'.\mathit{off} < v'.e$
  Immediate by $\vdash_\delta v'$ of (DEREF-ASSMS).
- $Mem(v'.\sigma + v'.\mathit{off}) = v$
  Here, by (DEREF-ASSMS)'s $v = \mathcal{M}_d(v'.\sigma + v'.\mathit{off})$, and $\vdash_\delta v'$, and the antecedents, it suffices to show that:
  $v'.\sigma + v'.\mathit{off} \in A_s$.
  This is immediate by Lemma 81.

Case evalLim:

Here, $\wr e \int_{\mathit{fid},mid,\beta} = \mathtt{lim}(\mathcal{E}, \mathcal{E}_s, \mathcal{E}_e)$

By rule evalLim, we know (LIM-ASSMS):

$\mathcal{E}, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow v$,
$\mathcal{E}_s, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow s'$,
$\mathcal{E}_e, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow e'$,
$s' \in \mathbb{Z}$,
$e' \in \mathbb{Z}$,
$v = (x, s, e, \_) \in Cap$,
$[s', e') \subseteq [s, e)$, and
$v' = (x, s', e', 0)$

By Definition 58, we distinguish the following cases:

- **Case** $e = \mathsf{limRange}(e_{cap}, e_s, e_e)$**:**
  Here, $\mathcal{E} = \wr e_{cap} \int_{\mathit{fid},mid,\beta}$, $\mathcal{E}_s = \wr e_s \int_{\mathit{fid},mid,\beta}$, and $\mathcal{E}_e = \wr e_e \int_{\mathit{fid},mid,\beta}$
  We thus get the following induction hypotheses (IH-limRange):
  $e_{cap}, \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow v$,
  $e_s, \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow s'$, and
  $e_e, \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow e'$
  By inverting our goal using rule Evaluate-expr-limrange, we get the following subgoals instead:

  $e_{cap}, \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow (x, s, e, \_)$,
  $e_s, \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow s'$,
  $s' \in \mathbb{Z}$,
  $e_e, \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow e'$,
  $e' \in \mathbb{Z}$,
  $[s', e') \subseteq [s, e)$, and
  $v' = (x, s', e', 0)$
  which are all immediate by (IH-limRange) and (LIM-ASSMS).

- **Case** $e = \mathsf{addr}(\mathit{vid}) \;\wedge\; \beta(\mathit{vid}, \bot, \mathit{mid}) = (\mathit{st}, \mathit{end})$**:**
  Here, $\mathcal{E} = \mathtt{ddc}$, $\mathcal{E}_s = \mathtt{capStart}(\mathtt{ddc}) + \mathit{st}$, and $\mathcal{E}_e = \mathtt{capStart}(\mathtt{ddc}) + \mathit{end}$
  Thus, by (LIM-ASSMS), inversion using rules evalddc and evalCapStart, and by our lemma assumptions, we conclude:
  $v = (x, s, e, \_) = (\delta, \Delta(\mathit{mid}).1, \Delta(\mathit{mid}).2, \_)$,
  $s' = \Delta(\mathit{mid}).1 + \mathit{st}$, and
  $e' = \Delta(\mathit{mid}).1 + \mathit{end}$
  Thus, $v' = (\delta, \Delta(\mathit{mid}).1 + \mathit{st}, \Delta(\mathit{mid}).1 + \mathit{end}, 0)$
  Thus, by inverting our goal using rule Evaluate-expr-addr-module, only the following subgoals are not immediate:
  - $\mathit{vid} \notin \mathtt{localIDs}(\mathit{Fd}(\mathit{fid})) \cup \mathtt{args}(\mathit{Fd}(\mathit{fid}))$, and
  - $\mathit{vid} \in \mathit{MVar}(\mathit{mid})$
    They both follow by assumption
    $\_;\ \_;\ \overline{\mathit{mods}}; \Sigma; \Delta; \beta; \mathit{MVar}; \mathit{Fd} \vdash_{\mathit{exec}} \langle \mathit{Mem}, \mathit{stk}, \mathit{pc}, \Phi, \mathit{nalloc}\rangle$ after inversion using rule Exec-state-src then rule Well-formed program and parameters and Well-formed program.
- **Case** $e = \mathsf{addr}(\mathit{vid}) \;\wedge\; \beta(\mathit{vid}, \mathit{fid}, \mathit{mid}) = (\mathit{st}, \mathit{end})$**:**
  Here, $\mathcal{E} = \mathtt{stc}$, $\mathcal{E}_s = \mathtt{capStart}(\mathtt{stc}) + \mathtt{capOff}(\mathtt{stc}) + \mathit{st}$, and
  $\mathcal{E}_e = \mathtt{capStart}(\mathtt{stc}) + \mathtt{capOff}(\mathtt{stc}) + \mathit{end}$.
  Thus, by (LIM-ASSMS), inversion using rules evalstc, evalCapStart, and evalCapOff, and by our lemma assumptions, we conclude:
  $v = (x, s, e, \_) = (\delta, \Sigma(\mathit{mid}).1, \Sigma(\mathit{mid}).2, \Phi(\mathit{mid}))$,
  $s' = \Sigma(\mathit{mid}).1 + \Phi(\mathit{mid}) + \mathit{st}$, and
  $e' = \Delta(\mathit{mid}).1 + \Phi(\mathit{mid}) + \mathit{end}$
  Thus, $v' = (\delta, \Sigma(\mathit{mid}).1 + \Phi(\mathit{mid}) + \mathit{st}, \Sigma(\mathit{mid}).1 + \Phi(\mathit{mid}) + \mathit{end}, 0)$
  Thus, by inverting our goal using rule Evaluate-expr-addr-local, only the following subgoal is not immediate: $\mathit{vid} \in \mathtt{localIDs}(\mathit{Fd}(\mathit{fid})) \cup \mathtt{args}(\mathit{Fd}(\mathit{fid}))$
  This follows by assumption
  $\_;\ \_;\ \overline{\mathit{mods}}; \Sigma; \Delta; \beta; \mathit{MVar}; \mathit{Fd} \vdash_{\mathit{exec}} \langle \mathit{Mem}, \mathit{stk}, \mathit{pc}, \Phi, \mathit{nalloc}\rangle$ after inversion using rule Exec-state-src then rule Well-formed program and parameters and Well-formed program.

This concludes case evalLim.

This concludes the proof of Lemma 90. □

Lemma 91 (Memory bounds are preserved by compilation)**.**

$$\forall \overline{\mathit{mods}}, \mathit{mid}, \mathit{fid}, \Delta, \Sigma, \beta, K_{\mathit{mod}}, K_{\mathit{fun}}, \mathcal{M}_c, \mathit{imp}, \mathsf{mstc}, \phi.$$
$$[\![\overline{\mathit{mods}}]\!]_{\Delta,\Sigma,\beta,K_{\mathit{mod}},K_{\mathit{fun}}} = \langle \mathcal{M}_c, \mathcal{M}_d, \mathit{imp}, \mathsf{mstc}, \phi\rangle \;\wedge$$
$$\mathit{funDefs} = \{\mathit{modFunDef} \mid \mathit{modFunDef} \in \mathit{modFunDefs} \wedge (\_, \_, \mathit{modFunDefs}) \in \overline{\mathit{mods}}\} \;\wedge$$
$$\mathit{Fd} = \{\mathit{funID} \mapsto \mathit{funDef} \mid \mathit{funDef} \in \mathit{funDefs} \wedge \mathit{funDef} = (\_, \mathit{funID}, \_, \_, \_)\} \;\wedge$$
$$(\mathit{mid}, \_, \_) \in \overline{\mathit{mods}}$$
$$\implies$$
$$\forall a \in \Delta(\mathit{mid}).\ \mathcal{M}_d(a) = 0 \;\wedge$$
$$\mathit{offs} = \{\mathit{funId} \mapsto K_{\mathit{fun}}(\mathit{fid}).1 \mid \mathit{funId} \in \mathsf{dom}(\mathit{Fd})\} \;\wedge$$
$$\mathit{imp}(\mathit{mid}) = ((\kappa, K_{\mathit{mod}}(\mathit{mid}).1, K_{\mathit{mod}}(\mathit{mid}).2, 0), (\delta, \Delta(\mathit{mid}).1, \Delta(\mathit{mid}).2, 0), \mathit{offs}) \;\wedge$$
$$\mathsf{mstc}(\mathit{mid}) = (\delta, \Sigma(\mathit{mid}).1, \Sigma(\mathit{mid}).2, 0) \;\wedge$$
$$\forall \mathit{fid}.\ \mathit{mid} = \mathtt{moduleID}(\mathit{Fd}(\mathit{fid})) \implies \phi(\mathit{mid}, \mathit{fid}) = (\mathtt{length}(\mathtt{args}(\mathit{Fd}(\mathit{fid}))), \mathtt{length}(\mathtt{localIDs}(\mathit{Fd}(\mathit{fid}))))$$

Proof. Immediate from the assumptions after inversion using rules Module-list-translation and Module-translation. ☐

Lemma 92 (No additional code/data is added by the compiler)**.**

$$\begin{aligned}
&\forall \overline{mods}, \Delta, \Sigma, \beta, K_{mod}, K_{fun}, \mathcal{M}_c, imp, \mathsf{mstc}, \phi. \\
&[\![\overline{mods}]\!]_{\Delta,\Sigma,\beta,K_{mod},K_{fun}} = \langle \mathcal{M}_c, \mathcal{M}_d, imp, \mathsf{mstc}, \phi\rangle \wedge \\
&funDefs = \{modFunDef \mid modFunDef \in modFunDefs \wedge (_, _, modFunDefs) \in \overline{mods}\} \wedge \\
&Fd = \{funID \mapsto funDef \mid funDef \in funDefs \wedge funDef = (_, funID, _, _, _)\} \\
&\implies \\
&(mid \in \mathsf{dom}(imp) \implies \exists fid.\ mid = \mathtt{moduleID}(Fd(fid)) \wedge \\
&mid \in \mathsf{dom}(\mathsf{mstc}) \implies \exists fid.\ mid = \mathtt{moduleID}(Fd(fid)) \wedge \\
&(mid, fid) \in \mathsf{dom}(\phi) \implies mid = \mathtt{moduleID}(Fd(fid)) \wedge \\
&a \in \mathsf{dom}(\mathcal{M}_d) \implies \exists fid.\ mid = \mathtt{moduleID}(Fd(fid)) \ \wedge \ a \in \Delta(mid) \wedge \\
&a \in \mathsf{dom}(\mathcal{M}_c) \implies \exists mid, fid, n.\ mid = \mathtt{moduleID}(Fd(fid)) \ \wedge \ n \in [0, |\mathtt{commands}(Fd(fid))|) \ \wedge \\
&\qquad a = K_{mod}(mid).1 + K_{fun}(fid).1 + n)
\end{aligned}$$

Proof. Immediate from the assumptions after inversion using rules Module-list-translation and Module-translation. ☐

Lemma 93 (Code memory is the translation of the commands arranged according to K_{mod} and K_{fun})**.**

$$\begin{aligned}
&\forall \overline{mods}, \Delta, \Sigma, \beta, K_{mod}, K_{fun}, mid, fid, n, \mathcal{M}_c. \\
&[\![\overline{mods}]\!]_{\Delta,\Sigma,\beta,K_{mod},K_{fun}} = \langle \mathcal{M}_c, _, _, _\rangle \wedge \\
&funDefs = \{modFunDef \mid modFunDef \in modFunDefs \wedge (_, _, modFunDefs) \in \overline{mods}\} \wedge \\
&Fd = \{funID \mapsto funDef \mid funDef \in funDefs \wedge funDef = (_, funID, _, _, _)\} \wedge \\
&mid = \mathtt{moduleID}(Fd(fid)) \ \wedge \ n \in [0, |\mathtt{commands}(Fd(fid))|) \\
&\implies \\
&\mathcal{M}_c(K_{mod}(mid).1 + K_{fun}(fid).1 + n) = (\!|\mathtt{commands}(Fd(fid))(n)|\!)_{Fd,K_{fun},fid,mid,\beta}
\end{aligned}$$

Proof. Immediate from the assumptions after inversion using rules Module-list-translation, Module-translation, and Function-translation. ☐

Definition 61 (Related program counters)**.**

$$\begin{aligned}
&K_{mod}; K_{fun}; \Sigma; \Delta; \beta; MVar; Fd; pc \cong \mathsf{pcc} \ \stackrel{\mathsf{def}}{=} \\
&K_{mod}(\mathtt{moduleID}(Fd(pc.fid))).1 + K_{fun}(pc.fid).1 + pc.n = \mathsf{pcc}.\sigma + \mathsf{pcc}.\mathit{off} \ \wedge \\
&K_{mod}(\mathtt{moduleID}(Fd(pc.fid))) = [\mathsf{pcc}.\sigma, \mathsf{pcc}.e)
\end{aligned}$$

Definition 62 (Related stacks)**.**

$$\begin{aligned}
&K_{mod}; K_{fun}; \Sigma; \Delta; \beta; MVar; Fd; stk_s \cong stk_t \ \stackrel{\mathsf{def}}{=} \\
&\mathtt{length}(stk_s) = \mathtt{length}(stk_t) \wedge \\
&\forall i \in \mathsf{dom}(stk_s) K_{mod}; K_{fun}; \Sigma; \Delta; \beta; MVar; Fd; stk_s(i) \cong stk_t(j).\mathsf{pcc}
\end{aligned}$$

Definition 63 (Related local stack usage)**.**
The usage of local stacks is related between a candidate pair of source and target states when 1. the stack usage $\Phi(mid)$ *in the source state is equal to that given by the capability offset* $\mathsf{mstc}(mid).\mathit{off}$

of the stack capability of the target state, and 2. for all functions fid, fid is callable (i.e., there is enough stack space to call it according to Φ) in the source state iff it is callable in the target state (according to mstc). Additionally, the number of arguments specified in the source interface by the function definitions map Fd matches the number of arguments given by the implementation of the target functions specified by the map ϕ of call frame sizes.

$$
\begin{aligned}
& K_{mod}; K_{fun}; \Sigma; \Delta; \beta; MVar; Fd; \Phi \cong \mathsf{mstc}, \phi \\
& \stackrel{\mathsf{def}}{=} \\
& \forall mid \in \mathsf{dom}(\Phi).\ \Phi(mid) = \mathsf{mstc}(mid).\mathit{off} \\
& \wedge \\
& \forall fid \in \mathsf{dom}(Fd), mid.\ \mathtt{moduleID}(Fd(fid)) = mid \implies \\
& (\mathtt{frameSize}(Fd(fid)) + \Sigma(mid).1 + \Phi(mid) < \Sigma(mid).2 \iff \\
& \phi(mid, fid).1 + \phi(mid, fid).2 + \mathsf{mstc}(mid).\sigma + \mathsf{mstc}(mid).\mathit{off} < \mathsf{mstc}(mid).e) \\
& \wedge \\
& \forall fid \in \mathsf{dom}(Fd), mid.\ \mathtt{moduleID}(Fd(fid)) = mid \implies \\
& \mathtt{length}(\mathtt{args}(Fd(fid))) = \phi(mid, fid).1 \\
& \wedge \\
& \forall (mid, fid) \in \mathsf{dom}(\phi).\ fid \ \in \mathsf{dom}(Fd) \ \wedge \ mid = \mathtt{moduleID}(Fd(fid))
\end{aligned}
$$

Definition 64 (Cross-language compiled-program state similarity)**.**

$$
\begin{aligned}
& K_{mod}; K_{fun}; \Sigma; \Delta; \beta; MVar; Fd; \langle Mem, stk, pc, \Phi, nalloc\rangle \cong_{modIDs} \langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle \\
& \stackrel{\mathsf{def}}{=} \\
& nalloc = \mathsf{nalloc} \ \wedge \\
& A_s = \mathsf{reachable\_addresses}(\Sigma, \Delta, modIDs, Mem) \ \wedge \\
& A_t = \mathsf{reachable\_addresses}(\bigcup_{mid \in modIDs} \{imp(mid).\mathsf{ddc}, \mathsf{mstc}(mid).\mathsf{stc}\}, \mathcal{M}_d) \ \wedge \\
& A_s = A_t \ \wedge \ Mem|_{A_s} = \mathcal{M}_d|_{A_t} \ \wedge \\
& \Delta(\mathtt{moduleID}(Fd(pc.fid))) = [\mathsf{ddc}.\sigma, \mathsf{ddc}.e) \ \wedge \\
& \Sigma(\mathtt{moduleID}(Fd(pc.fid))) = [\mathsf{stc}.\sigma, \mathsf{stc}.e) \ \wedge \\
& \Phi(\mathtt{moduleID}(Fd(pc.fid))) = \mathsf{stc}.\mathit{off} \ \wedge \\
& K_{mod}; K_{fun}; \Sigma; \Delta; \beta; MVar; Fd; pc \cong \mathsf{pcc} \ \wedge \\
& K_{mod}; K_{fun}; \Sigma; \Delta; \beta; MVar; Fd; stk \cong stk \ \wedge \\
& K_{mod}; K_{fun}; \Sigma; \Delta; \beta; MVar; Fd; \Phi \cong \mathsf{mstc}, \phi \\
& \vee \\
& (pc = \bot \ \wedge \ \mathcal{M}_c(\mathsf{pcc}) = \bot)
\end{aligned}
$$

Lemma 94 (Cross-language equi-k-accessibility and memory equality is preserved by deleting assignments and safe allocation).

$$\forall \mathcal{A}, a, v, Mem, \mathcal{M}_d.$$
$$\forall k, \exists \mathcal{A}'.\ \mathcal{A}' = \mathsf{access}_k(\mathcal{A}, Mem) = \mathsf{access}_{k,\mathcal{M}_d}\mathcal{A}\ \wedge$$
$$Mem|_{\mathcal{A}'} = \mathcal{M}_d|_{\mathcal{A}'}\ \wedge$$
$$(v \neq (\delta, \_, \_, \_)\ \vee$$
$$(v = (\delta, \sigma, e, \_)\ \wedge\ \forall a^* \in [\sigma, e).\ \mathcal{M}_d[a \mapsto v](a^*) \neq (\delta, \_, \_, \_)\ \wedge\ Mem[a \mapsto v](a^*) \neq (\delta, \_, \_, \_)))$$
$$\implies$$
$$(\forall k, \exists \mathcal{A}'.\ \mathcal{A}' = \mathsf{access}_k(\mathcal{A}, Mem[a \mapsto v]) = \mathsf{access}_{k,\mathcal{M}_d[a \mapsto v]}\mathcal{A}\ \wedge$$
$$Mem[a \mapsto v]|_{\mathcal{A}'} = \mathcal{M}_d[a \mapsto v]|_{\mathcal{A}'})$$

Proof.
We fix arbitrary $\mathcal{A}, a, v, Mem, \mathcal{M}_d$ and consider the following two cases from the disjunctive assumption:

- **Case** $v \neq (\delta, \_, \_, \_)$**:**

  In this case, by Lemma 33, we know $\mathsf{access}_{k,\mathcal{M}_d[a \mapsto v]}\mathcal{A} = \chi_k(\mathcal{A}, \mathcal{M}_d, a)$.

  Also, by Lemma 74, we know $\mathsf{access}_k(\mathcal{A}, Mem[a \mapsto v]) = \chi_k(\mathcal{A}, Mem, a)$.

  Then, our first subgoal becomes:
  $\forall k.\ \chi_k(\mathcal{A}, Mem, a) = \chi_k(\mathcal{A}, \mathcal{M}_d, a)$.

  This can be shown by an easy induction on k with the help of Lemmas 31 and 72, Definitions 24 and 25 and the assumptions:
  $\forall k, \exists \mathcal{A}'.\ \mathcal{A}' = \mathsf{access}_k(\mathcal{A}, Mem) = \mathsf{access}_{k,\mathcal{M}_d}\mathcal{A}\ \wedge\ Mem|_{\mathcal{A}'} = \mathcal{M}_d|_{\mathcal{A}'}$

  Our next subgoal $\forall k.\ \exists \mathcal{A}'.\ Mem[a \mapsto v]|_{\mathcal{A}'} = \mathcal{M}_d[a \mapsto v]|_{\mathcal{A}'}$ (now with $\mathcal{A}' = \chi_k(\mathcal{A}, Mem, a) = \chi_k(\mathcal{A}, \mathcal{M}_d, a)$) follows again immediately from Lemmas 31 and 72, and the assumptions.

- **Case** $v = (\delta, \sigma, e, \_) \wedge \forall a^* \in [\sigma, e).\ \mathcal{M}_d[a \mapsto v](a^*) \neq (\delta, \_, \_, \_) \wedge Mem[a \mapsto v](a^*) \neq (\delta, \_, \_, \_))$**:**

  Here, we distinguish two cases:

  - **Case** $a \in \mathsf{access}_k(\mathcal{A}, Mem) = \mathsf{access}_{k,\mathcal{M}_d}\mathcal{A}$**:**

    In this case, our goals follow by Lemmas 41 and 67 together with Lemmas 31 and 72 and the assumptions.

  - **Case** $a \notin \mathsf{access}_k(\mathcal{A}, Mem) = \mathsf{access}_{k,\mathcal{M}_d}\mathcal{A}$**:**

    In this case, our goals follow immediately from the assumptions after applying Lemmas 22 and 69.

□

Lemma 95 (Cross-language equi-reachability and memory equality is preserved by deleting assignments, safe allocation, and assigning derivable capabilities).

$\forall a, v, \Sigma, \Delta, modIDs, Mem, C, \mathcal{M}_d.$

$\mathcal{A} = \mathsf{static\_addresses}(\Sigma, \Delta, modIDs) = \bigcup_{c \in C} [c.\sigma, c.e) \ \wedge$

$\exists \mathcal{A}_r.\ \mathcal{A}_r = \mathsf{reachable\_addresses}(\Sigma, \Delta, modIDs, Mem) = \mathsf{reachable\_addresses}(C, \mathcal{M}_d) \ \wedge$

$Mem|_{\mathcal{A}_r} = \mathcal{M}_d|_{\mathcal{A}_r} \ \wedge$

$a \in \mathcal{A}_r \ \wedge$

$(v \neq (\delta, \_, \_, \_) \ \vee$

$(v = (\delta, \sigma, e, \_) \ \wedge \ \forall a^* \in [\sigma, e).\ \mathcal{M}_d[a \mapsto v](a^*) \neq (\delta, \_, \_, \_) \ \wedge \ Mem[a \mapsto v](a^*) \neq (\delta, \_, \_, \_)) \ \vee$

$(v = (\delta, \sigma, e, \_) \ \wedge \ \Sigma, \Delta, modIDs, Mem \vDash v \ \wedge \ C, \mathcal{M}_d \vDash v))$

$\implies$

$\exists \mathcal{A}_r'.\ \mathcal{A}_r' = \mathsf{reachable\_addresses}(\Sigma, \Delta, modIDs, Mem[a \mapsto v]) = \mathsf{reachable\_addresses}(C, \mathcal{M}_d[a \mapsto v])$

$Mem[a \mapsto v]|_{\mathcal{A}_r'} = \mathcal{M}_d[a \mapsto v]|_{\mathcal{A}_r'}$

Proof.
Here, we can use Lemma 13, and by an easy argument using assumptions $Mem|_{\mathcal{A}_r} = \mathcal{M}_d|_{\mathcal{A}_r}$ and $\mathcal{A}_r = \mathsf{reachable\_addresses}(\Sigma, \Delta, modIDs, Mem) = \mathsf{reachable\_addresses}(C, \mathcal{M}_d)$, we obtain the antecedent of Lemma 94, which proves two cases of our goal (again after applying Lemma 13 to pick a finite k).

The remaining case of our goal is proved by applying Lemmas 42 and 79 which give the first subgoal, and then applying Lemmas 43 and 80 to get the second subgoal from the assumptions. ☐

Lemma 96 (Compiled-program state similarity implies equi-reachability).

$\forall K_{mod}; K_{fun}; \Sigma; \Delta; \beta; MVar; Fd; \langle Mem, stk, pc, \Phi, nalloc \rangle, \langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc} \rangle.$

$K_{mod}, K_{fun}, \Sigma; \Delta; \beta; MVar; Fd; \langle Mem, stk, pc, \Phi, nalloc \rangle \cong_{modIDs} \langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc} \rangle$

$\implies$

$\mathsf{reachable\_addresses}(\Sigma, \Delta, modIDs, Mem) = \mathsf{reachable\_addresses}(\bigcup_{mid \in modIDs} \{imp(mid).\mathsf{ddc}, \mathsf{mstc}(mid).\mathsf{stc}\}, \mathcal{M}_d)$

Proof.
Immediate by Definition 64. ☐

Lemma 97 (Compiler forward simulation).

$\forall K_{mod}, K_{fun}, \Sigma; \Delta; \beta; MVar; Fd, \langle Mem, stk, pc, \Phi, nalloc \rangle, \overline{mods_1},$
$\mathcal{M}_c, \mathcal{M}_d, imp, \mathsf{mstc}, \phi.$

$[\![\overline{mods_1}]\!]_{\Delta, \Sigma, \beta, K_{mod}, K_{fun}} = t \ \wedge$

$K_{mod}; K_{fun}; \overline{mods_1}; \Sigma; \Delta; \beta; MVar; Fd \vdash_{exec} \langle Mem, stk, pc, \Phi, nalloc \rangle \ \wedge$

$t \vdash_{exec} \langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc} \rangle \ \wedge$

$modIDs = \{modID \mid (modID, \_, \_) \in \overline{mods_1}\} \ \wedge$

$K_{mod}; K_{fun}; \Sigma; \Delta; \beta; MVar; Fd; \langle Mem, stk, pc, \Phi, nalloc \rangle \cong_{modIDs} \langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc} \rangle \ \wedge$

$\Sigma; \Delta; \beta; MVar; Fd \vdash \langle Mem, stk, pc, \Phi, nalloc \rangle \to \langle Mem', stk', pc', \Phi', nalloc' \rangle$

$\implies$

$\langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc} \rangle \to \langle \mathcal{M}_c, \mathcal{M}_d', stk', imp, \phi, \mathsf{ddc}', \mathsf{stc}', \mathsf{pcc}', \mathsf{mstc}', \mathsf{nalloc}' \rangle \ \wedge$

$K_{mod}; K_{fun}; \Sigma; \Delta; \beta; MVar; Fd; \langle Mem', stk', pc', \Phi', nalloc' \rangle \cong_{modIDs} \langle \mathcal{M}_c, \mathcal{M}_d', stk', imp, \phi, \mathsf{ddc}', \mathsf{stc}', \mathsf{pcc}', \mathsf{mstc}', \mathsf{nalloc}' \rangle$

Proof.
We assume the antecedents, and we unfold assumption
$K_{mod}; K_{fun}; \Sigma; \Delta; \beta; MVar; Fd; \langle Mem, stk, pc, \Phi, nalloc\rangle \cong_{modIDs} \langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle$
using Definition 64 to obtain:

Equal allocation

$nalloc = \mathsf{nalloc}$

Equal reachable memories

$A_s = \mathsf{reachable\_addresses}(\Sigma, \Delta, modIDs, Mem)\ \wedge$
$A_t = \mathsf{reachable\_addresses}(\bigcup_{mid \in modIDs} \{imp(mid).\mathsf{ddc}, \mathsf{mstc}(mid).\mathsf{stc}\}, \mathcal{M}_d)\ \wedge$
$A_s = A_t\ \wedge\ Mem|_{A_s} = \mathcal{M}_d|_{A_t}$

Equal data segments

$\Delta(\mathtt{moduleID}(Fd(pc.fid))) = (\mathsf{ddc}.\sigma, \mathsf{ddc}.e)$

Equal stack regions

$\Sigma(\mathtt{moduleID}(Fd(pc.fid))) = (\mathsf{stc}.\sigma, \mathsf{stc}.e)$

Equal stack pointers

$\Phi(\mathtt{moduleID}(Fd(pc.fid))) = \mathsf{stc}.\mathit{off}$

Related program counters

$K_{mod}; K_{fun}; \Sigma; \Delta; \beta; MVar; Fd; pc \cong \mathsf{pcc}$

Related trusted stacks

$K_{mod}; K_{fun}; \Sigma; \Delta; \beta; MVar; Fd; stk \cong stk$

Related local stack usage

$K_{mod}; K_{fun}; \Sigma; \Delta; \beta; MVar; Fd; \Phi \cong \mathsf{mstc}, \phi$

Static addresses are the same as module's capabilities

We let $C = \bigcup_{mid \in modIDs} \{imp(mid).\mathsf{ddc}, \mathsf{mstc}(mid).\mathsf{stc}\}$.

Then, using assumption $[\![mods_1]\!]_{\Delta,\Sigma,\beta,K_{mod},K_{fun}} = \langle \mathcal{M}_{c1}, \mathcal{M}_{d1}, imp_1, \mathsf{mstc}_1, \phi_1\rangle$ and by Lemmas 91 and 92, we have: $\mathsf{static\_addresses}(\Sigma, \Delta, modIDs) = \bigcup_{c \in C} [c.\sigma, c.e)$

Then, we prove our goal by case distinction on the source reduction
$(\Sigma; \Delta; \beta; MVar; Fd \vdash \langle Mem, stk, pc, \Phi, nalloc\rangle \to \langle Mem', stk', pc', \Phi', nalloc'\rangle)$.

Case Assign-to-var-or-arr:

In this case, by inversion, we have the following assumptions:

1. $(fid, n) = pc$
2. $\mathtt{commands}(Fd(fid))(n) = \mathsf{Assign}\ e_l\ e_r$
3. $frameSize = \mathtt{frameSize}(Fd(fid))$
4. $e_l, \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow (\delta, s, e, \mathit{off})$
5. $e_r, \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow v$
6. $modID = \mathtt{moduleID}(Fd(fid))$
7. $\phi = \Sigma(modID).1 + \Phi(modID)$
8. $\forall s', e'.\ v = (\delta, s', e', \_) \implies ([s', e') \cap \Sigma(modID) = \emptyset \vee [s, e) \subseteq \Sigma(modID))$
9. $s \leq s + \mathit{off} < e$
10. $Mem' = Mem[s + \mathit{off} \mapsto v]$

11. $pc' = \mathtt{inc}(pc)$

And we would like to prove the first subgoal:

$\langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle \to \langle \mathcal{M}_c, \mathcal{M}'_d, stk', imp, \phi, \mathsf{ddc}', \mathsf{stc}', \mathsf{pcc}', \mathsf{nalloc}'\rangle$
By inversion using rule assign, we obtain the following subgoals:

(a) $\vdash_\kappa \mathsf{pcc}$

By unfolding Definition 2, the condition on the capability type follows from assumption $t \vdash_{exec} \langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle$ by inversion using exec-state.

It remains to show the condition on the bounds:

$\mathsf{pcc}.\sigma \leq \mathsf{pcc}.\sigma + \mathsf{pcc}.\mathit{off} < \mathsf{pcc}.e$

By substitution using assumption **Related program counters** after unfolding Definition 61, our goal is:

$\mathsf{pcc}.\sigma \leq \ K_{mod}(\mathtt{moduleID}(Fd(pc.\mathit{fid}))).1 + K_{fun}(pc.\mathit{fid}).1 + pc.n \ < \mathsf{pcc}.e$

By assumption **Related program counters** after unfolding Definition 61,
we know uniquely the values of $\mathsf{pcc}.\sigma$ and $\mathsf{pcc}.e$:

$[\mathsf{pcc}.\sigma, \mathsf{pcc}.e) = K_{mod}(\mathtt{moduleID}(Fd(pc.\mathit{fid})))$

Thus, by substitution and a simple rewriting into interval notation, our goal becomes:

$K_{mod}(\mathtt{moduleID}(Fd(pc.\mathit{fid}))).1 + K_{fun}(pc.\mathit{fid}).1 + pc.n \in K_{mod}(\mathtt{moduleID}(Fd(pc.\mathit{fid})))$

This goal can now be proved by substitution and interval arithmetic:

first by obtaining the condition on $K_{fun}(pc.\mathit{fid})$ and $K_{mod}(\mathtt{moduleID}(Fd(pc.\mathit{fid})))$ from Exec-state-src,

then by noticing that $pc.n \in |\mathtt{commands}(Fd(\mathit{fid}))|$ which we have from assumption (2.) obtained above.

The argument above proves $\vdash_\kappa \mathsf{pcc}$.

(b) $\mathcal{M}_c(\mathsf{pcc}) = \mathsf{Assign}\ \mathcal{E}_L\ \mathcal{E}_R$

This follows immediately by Lemma 93 and definition 59
after replacing $\mathsf{pcc}.\sigma + \mathsf{pcc}.\mathit{off}$ as in the previous goal.

By unrolling Definition 59, we immediately get the following substitutions which we use in the coming goals:

$\mathcal{E}_R = \wr e_r \int_{pc.\mathit{fid},\mathtt{moduleID}(Fd(\mathit{fid})),\beta}$
and $\mathcal{E}_L = \wr e_l \int_{pc.\mathit{fid},\mathtt{moduleID}(Fd(\mathit{fid})),\beta}$.

By assumption **Equal reachable memories**, we can apply Lemma 88 for the next two goals (we have all the assumptions).

(c) $\mathcal{E}_R, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow v$ and

(d) $\mathcal{E}_L, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow c$

are proved by Lemma 88.

(e) $\vdash_\delta c$

This follows by Lemma 88, then by assumptions (4.) and (9.).

(f) $\models_\delta v \implies (v \cap \mathsf{stc} = \emptyset \vee c \subsetneq \mathsf{stc})$

After substitution using the assumption [**Equal stack regions**]:

$\Sigma(\mathtt{moduleID}(Fd(pc.\mathit{fid}))) = (\mathsf{stc}.\sigma, \mathsf{stc}.e)$,

this goal is immediately satisfied by using assumption (8.).

(g) $\mathsf{pcc}' = \mathtt{inc}(\mathsf{pcc}, 1)$, and

(h) $\mathcal{M}'_d = \mathcal{M}_d[c \mapsto v]$

These are inevitable by noticing that only rule assign applies after having proved the precondition

$\mathcal{M}_c(\mathsf{pcc}) = \mathsf{Assign}\ \mathcal{E}_L\ \mathcal{E}_R$.

We also have to prove:
$K_{mod}; K_{fun}; \Sigma; \Delta; \beta; MVar; Fd; \langle Mem', stk', pc', \Phi', nalloc' \rangle \cong_{modIDs}$
$\langle \mathcal{M}_c, \mathcal{M}_d', stk', imp, \phi, \mathsf{ddc}', \mathsf{stc}', \mathsf{pcc}', \mathsf{nalloc}' \rangle$.
By unfolding Definition 64, we obtain the following subgoals:

(i) $nalloc' = \mathsf{nalloc}'$
Immediate by assumption after substitution using the preconditions $nalloc' = nalloc$ and $\mathsf{nalloc}' = \mathsf{nalloc}$.

(j) $A_s' = \mathsf{reachable\_addresses}(\Sigma, \Delta, modIDs, Mem')\ \wedge$
$A_t' = \mathsf{reachable\_addresses}(\bigcup_{mid \in modIDs} \{imp'(mid).\mathsf{ddc}, \mathsf{mstc}'(mid)\}, \mathcal{M}_d')\ \wedge$
$A_s' = A_t'\ \wedge\ Mem'|_{A_s'} = \mathcal{M}_d'|_{A_t'}$
First, we obtain the following statement (*):
$imp(\mathtt{moduleID}(Fd(pc.fid))).\mathsf{ddc} \doteq \mathsf{ddc}$ and $\mathsf{mstc}(\mathtt{moduleID}(Fd(pc.fid))) \doteq \mathsf{stc})$
which follows from rule exec-state together with Lemmas 91 and 93.
Then, we distinguish two cases:

- **Case $v \neq (\delta, \_, \_, \_)$:**
  In this case, we apply Lemma 95 to obtain the following subgoals:
  - $c = (\delta, s, e, off)$, and
  - $v = v$
    These two follow from the successful application of Lemma 88 in the proof of subgoals (c) and (d) above.
  - The remaining subgoals follow immediately from the assumptions **Equal reachable memories** and **Static addresses are the same as module's capabilities**.
- **Case $v = (\delta, \sigma, e, \_)$:**
  In this case, by Lemmas 18, 25 and 81 (using assumption $\mathtt{moduleID}(Fd(pc.fid)) \in modIDs$ for Lemma 81 and statement (*) for Lemmas 18 and 25), we know:
  $[\sigma, e) \subseteq A_s = A_t$
  which by folding Definitions 23 and 50, gives us (**):
  $\Sigma, \Delta, modIDs, Mem \vDash v$, and
  $\bigcup_{mid \in modIDs} \{imp(mid).\mathsf{ddc}, \mathsf{mstc}(mid)\}, \mathcal{M}_d \vDash v$
  Now, we apply Lemma 95 to obtain the following subgoals:
  - $c = (\delta, s, e, off)$, and
  - $v = v$
    These two follow from the successful application of Lemma 88 in the proof of subgoals (c) and (d) above.
  - The remaining subgoals follow immediately from (**) and the assumptions **Equal reachable memories** and **Static addresses are the same as module's capabilities**.

(k) $\Delta(\mathtt{moduleID}(Fd(pc'.fid))) = (\mathsf{ddc}'.\sigma, \mathsf{ddc}'.e)$
Immediate by assumptions after rewriting using $\mathsf{ddc}' = \mathsf{ddc}$ and $pc'.fid = pc.fid$.

(l) $\Sigma(\mathtt{moduleID}(Fd(pc'.fid))) = (\mathsf{stc}'.\sigma, \mathsf{stc}'.e)$
Immediate by assumptions after rewriting using $\mathsf{stc}' = \mathsf{stc}$ and $pc'.fid = pc.fid$.

(m) $\Phi(\texttt{moduleID}(Fd(pc'.fid))) = \mathsf{stc}'.off$

Immediate by assumptions after rewriting using $\mathsf{stc}' = \mathsf{stc}$ and $pc'.fid = pc.fid$.

(n) $K_{mod}(\texttt{moduleID}(Fd(pc'.fid))).1 + K_{fun}(pc'.fid).1 + pc'.n = \mathsf{pcc}'.\sigma + \mathsf{pcc}'.off\ \wedge$
$K_{mod}(\texttt{moduleID}(Fd(pc'.fid))) = [\mathsf{pcc}'.\sigma, \mathsf{pcc}'.e)]$

This is immediate after substitution using the assumptions on pcc and pc and after having proved
$\mathsf{pcc}' = \texttt{inc}(\mathsf{pcc}, 1)$.

(o) $K_{mod}; K_{fun}; \Sigma; \Delta; \beta; MVar; Fd; stk' \cong stk'$

Immediate by assumption after rewriting using $stk' = stk$ and $stk' = stk$.

(p) $K_{mod}; K_{fun}; \Sigma; \Delta; \beta; MVar; Fd; \Phi' \cong \mathsf{mstc}', \phi$

Immediate by assumption after rewriting using $\Phi' = \Phi$ and $\mathsf{mstc}' = \mathsf{mstc}$.

Case Allocate:

In this case, by inversion, we have the following assumptions:

1. $(fid, n) = pc$
2. $\texttt{commands}(Fd(fid))(n) = \mathsf{Alloc}\ e_l\ e_{size}$
3. $e_l, \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow (\delta, s, e, off)$
4. $e_{size}, \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow v$
5. $s \leq s + off < e$
6. $v \in \mathbb{Z}^+$
7. $nalloc - v > \nabla$
8. $nalloc' = nalloc - v$
9. $Mem' = Mem[s + off \mapsto (\delta, nalloc', nalloc, 0)][a \mapsto 0 \mid a \in [nalloc', nalloc)]$

And we would like to prove the first subgoal:

$\langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle \to \langle \mathcal{M}_c, \mathcal{M}_d', stk', imp, \phi, \mathsf{ddc}', \mathsf{stc}', \mathsf{pcc}', \mathsf{nalloc}'\rangle$
By inversion using rule allocate, we obtain the following subgoals:

(a) $\vdash_\kappa \mathsf{pcc}$

Same as in the previous case.

(b) $\mathsf{pcc}' = \texttt{inc}(\mathsf{pcc}, 1)$

Same as in the previous case.

(c) $\mathcal{M}_c(\mathsf{pcc}) = \mathsf{Alloc}\ \mathcal{E}_L\ \mathcal{E}_{size}$

This follows immediately by Lemma 93 and definition 59
after replacing $\mathsf{pcc}.\sigma + \mathsf{pcc}.off$.

By unrolling Definition 59, we immediately get the following substitutions which we use in the coming goals:
$\mathcal{E}_{size} = \wr e_{size} \int_{pc.fid, \texttt{moduleID}(Fd(fid)), \beta}$
and $\mathcal{E}_L = \wr e_l \int_{pc.fid, \texttt{moduleID}(Fd(fid)), \beta}$.

By assumption **Equal reachable memories**, we can apply Lemma 88 for the next two goals (we have all the assumptions).

(d) $\mathcal{E}_{size}, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow v$ and

(e) $\mathcal{E}_L, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow c$

are proved by Lemma 88.

(f) $v \in \mathbb{Z}^+$

This follows by Lemma 88, then by assumption (6.).

(g) $\vdash_\delta c$

This follows by Lemma 88, then by assumptions (3.) and (5.).

(h) $\mathcal{M}'_d = \mathcal{M}_d[c \mapsto (\delta, \mathsf{nalloc} - v, \mathsf{nalloc}, 0), i \mapsto 0\ \forall i \in [\mathsf{nalloc} - v, \mathsf{nalloc})]$

Same as in the previous case (i.e., inevitable after proving that only rule allocate applies).

(i) $\mathsf{nalloc}' = \mathsf{nalloc} - v$

(j) $\mathsf{nalloc}' > \nabla$

The definition of nalloc' is inevitable by rule allocate.

The check follows from Lemma 88 and the corresponding check of precondition (7.).

We also have to prove:

$K_{mod}; K_{fun}; \Sigma; \Delta; \beta; MVar; Fd; \langle Mem', stk', pc', \Phi', nalloc' \rangle \cong_{modIDs}$
$\langle \mathcal{M}_c, \mathcal{M}'_d, stk', imp, \phi, \mathsf{ddc}', \mathsf{stc}', \mathsf{pcc}', \mathsf{nalloc}' \rangle$.

By unfolding Definition 64, we obtain the following subgoals:

(k) $nalloc' = \mathsf{nalloc}'$

This follows from Lemma 88 together with the assumption $nalloc = \mathsf{nalloc}$.

(l) $A'_s = \mathsf{reachable\_addresses}(\Sigma, \Delta, modIDs, Mem')\ \wedge$
$A'_t = \mathsf{reachable\_addresses}(\bigcup_{mid \in modIDs} \{imp'(mid).\mathsf{ddc}, \mathsf{mstc}'(mid)\}, \mathcal{M}'_d)\ \wedge$
$A'_s = A'_t\ \wedge\ Mem'|_{A'_s} = \mathcal{M}'_d|_{A'_t}$

First, we claim that (*):

$\mathsf{reachable\_addresses}(\bigcup_{mid \in modIDs} \{imp(mid).\mathsf{ddc}, \mathsf{mstc}(mid), \mathcal{M}_d[i \mapsto 0 \mid i \in [\mathsf{nalloc} - v, \mathsf{nalloc})]) = A_t$

We prove (*) by applying Lemma 21, so we must prove:

$[\mathsf{nalloc} - v, \mathsf{nalloc}') \cap A_t = \emptyset$

This can be proved by using Lemma 18, to obtain subgoals that are provable using both

(**1) $\forall(\_, dc, \_) \in \mathtt{range}(imp),\ \ a \in \mathsf{reachable\_addresses}(\{dc\}, \mathcal{M}_d) \implies a \geq \mathsf{nalloc}$, and

(**2) $\forall a, st.\ st \in \mathsf{range}(\mathsf{mstc}) \wedge a \in \mathsf{reachable\_addresses}(\{st\}, \mathcal{M}_d) \implies a \geq \mathsf{nalloc}$

We obtain (**1) and (**2) by inverting assumption

$t \vdash_{exec} \langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc} \rangle$

using rule exec-state.

Thus, having (*), we can now apply Lemma 95 to our goal which immediately proves it.

(m) $\Delta(\mathtt{moduleID}(Fd(pc'.fid))) = (\mathsf{ddc}'.\sigma, \mathsf{ddc}'.e)$

Immediate by assumptions after rewriting using $\mathsf{ddc}' = \mathsf{ddc}$ and $pc'.fid = pc.fid$.

(n) $\Sigma(\mathtt{moduleID}(Fd(pc'.fid))) = (\mathsf{stc}'.\sigma, \mathsf{stc}'.e)$

Immediate by assumptions after rewriting using $\mathsf{stc}' = \mathsf{stc}$ and $pc'.fid = pc.fid$.

(o) $\Phi(\mathtt{moduleID}(Fd(pc'.fid))) = \mathsf{stc}'.off$

Immediate by assumptions after rewriting using $\mathsf{stc}' = \mathsf{stc}$ and $pc'.fid = pc.fid$.

(p) $K_{mod}(\mathtt{moduleID}(Fd(pc'.fid))).1 + K_{fun}(pc'.fid).1 + pc'.n = \mathsf{pcc}'.\sigma + \mathsf{pcc}'.off\ \wedge$
$K_{mod}(\mathtt{moduleID}(Fd(pc'.fid))) = [\mathsf{pcc}'.\sigma, \mathsf{pcc}'.e)]$

This is immediate after substitution using the assumptions on pcc and pc and after having proved

$\mathsf{pcc}' = \mathtt{inc}(\mathsf{pcc}, 1)$.

(q) $K_{mod}; K_{fun}; \Sigma; \Delta; \beta; MVar; Fd; stk' \cong stk'$
Immediate by assumption after rewriting using $stk' = stk$ and $stk' = stk$.

(r) $K_{mod}; K_{fun}; \Sigma; \Delta; \beta; MVar; Fd; \Phi' \cong \mathsf{mstc}', \phi$
Immediate by assumption after rewriting using $\Phi' = \Phi$ and $\mathsf{mstc}' = \mathsf{mstc}$.

Case Call:

In this case, by inversion, we have the following assumptions:

1. $(fid, n) = pc$
2. $\mathtt{commands}(Fd(fid))(n) = \mathsf{Call}\ fid_{call}\ \overline{e}$
3. $modID = \mathtt{moduleID}(Fd(fid_{call}))$
4. $argNames = \mathtt{args}(Fd(fid_{call}))$
5. $localIDs = \mathtt{localIDs}(Fd(fid_{call}))$
6. $nArgs = \mathtt{length}(argNames) = \mathtt{length}(\overline{e})$
7. $nLocal = \mathtt{length}(localIDs)$
8. $frameSize = \mathtt{frameSize}(Fd(fid_{call}))$
9. $curFrameSize = \mathtt{frameSize}(Fd(fid))$
10. $curModID = \mathtt{moduleID}(Fd(fid))$
11. $\Sigma(modID).1 + \Phi(modID) + frameSize < \Sigma(modID).2$
12. $\Phi' = \Phi[modID \mapsto \Phi(modID) + frameSize]$
13. $\phi = \Sigma(curModID).1 + \Phi(curModID)$
14. $\phi' = \Sigma(modID).1 + \Phi'(modID)$
15. $\overline{e}(i), \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow v_i \quad \forall i \in [0, nArgs)$
16. $\forall i \in [0, nArgs), s', e'.\ \ v_i = (s', e', \_) \implies [s', e') \cap \Sigma(modID) = \emptyset$
17. $stk' = \mathtt{push}(stk, pc)$
18. $pc' = (fid_{call}, 0)$
19. $Mem' = Mem[\phi' + s_i \mapsto v_i \mid \beta(argNames(i)) = [s_i, \_) \wedge i \in [0, nArgs)]$
$[\phi' + s_i \mapsto 0 \mid \beta(localIDs(i)) = [s_i, \_) \wedge i \in [0, nLocal)]$

And we would like to prove the first subgoal:

$\langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle \to \langle \mathcal{M}_c, \mathcal{M}_d', stk', imp, \phi, \mathsf{ddc}', \mathsf{stc}', \mathsf{pcc}', \mathsf{nalloc}'\rangle$

By inversion using rule cinvoke then cinvoke-aux, we obtain the following subgoals:

(a) $\vdash_\kappa \mathsf{pcc}$
Same as in the previous cases.

(b) $\mathcal{M}_c(\mathsf{pcc}) = \mathsf{Cinvoke}\ modID\ fid_{call}\ \overline{e}$
This follows immediately by Lemma 93 and definition 59
after replacing $\mathsf{pcc}.\sigma + \mathsf{pcc}.\mathit{off}$.
By unrolling Definition 59, we immediately get the following substitutions which we use in the coming goals:
(EXPR-TRANS):
$\overline{e} = \wr\overline{e}\int_{pc.fid, \mathtt{moduleID}(Fd(fid)), \beta}$
and
$modID = \mathtt{moduleID}(Fd(fid_{call}))$.

By assumption **Equal reachable memories**, we can apply Lemma 88 for the next goal (we have all the assumptions).

(c) $\overline{e}(i),\ \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow v_i\ \forall i \in [0, nArgs)$

- First, we need to prove that (*) $nArgs = nArgs$.
  This follows from assumption **Related local stack usage** after unfolding Definition 63 and obtaining conjunct
  $\forall fid \in \mathsf{dom}(Fd), mid.\ \mathtt{moduleID}(Fd(fid)) = mid \implies \mathtt{length}(\mathtt{args}(Fd(fid))) = \phi(mid, fid).1$
  which we instantiate using fid_{call} from assumption (2.) and the substitution (EXPR-TRANS) from the previous subgoal's proof.
- Then, for an arbitrary $i \in [0, nArgs)$, we apply Lemma 88 to the i-th goal (namely, $\overline{e}(i),\ \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow v_i$) obtaining subgoals that are immediate by assumptions (including crucially assumption (15.) and the substitutions (EXPR-TRANS) from the previous subgoal's proof).

(d) $\phi(modID, fid_{call}) = (nArgs, nLocal)$

Here, we just need to prove that $\phi(modID, fid_{call})$ is defined and that $\phi(modID, fid_{call}).1 = nArgs$. This argument was given in the previous subgoal's proof.

(e) $(\delta, \sigma, e, \mathit{off}) = \mathsf{mstc}(modID)$

That the entry $modID$ exists in the domain of mstc follows by inversion of the antecedent using rule exec-state from the fact that $\phi(modID, fid_{call})$ is defined which is proven in previous subgoals.

(f) $\forall i \in [0, nArgs).\ \vDash_\delta v_i \implies v_i \cap \mathsf{stc} = \emptyset$

Here, we need to prove that $nArgs = nArgs$. This fact is proven in previous subgoals. Then, after substituting using that equality, the stated goal follows by assumption (16.) and subgoal (c) after substituting using assumption **Equal stack regions**.

(g) $(c, d, \mathit{offs}) = imp(modID)$

That the entry $modID$ exists in the domain of imp follows by Lemma 91 and by assumption $\mathtt{moduleID}(Fd(pc'.fid)) \in modIDs$.

(h) $\mathit{off}' = \mathit{off} + nArgs + nLocal$,

(i) $\mathsf{stc}' = (\delta, s, e, \mathit{off}')$,

(j) $stk' = \mathsf{push}(stk, (\mathsf{ddc}, \mathsf{pcc}, modID, fid_{call}))$,

(k) $\mathcal{M}'_d = \mathcal{M}_d[s + \mathit{off} + i \mapsto v_i\ \forall i \in [0, nArgs)][s + \mathit{off} + nArgs + i \mapsto 0\ \forall i \in [0, nLocal)]$,

(l) $\mathsf{mstc}' = \mathsf{mstc}[modID \mapsto \mathsf{stc}']$,

(m) $\mathsf{ddc}' = d$, and

(n) $\mathsf{pcc}' = \mathsf{inc}(c, \mathit{offs}(fid_{call}))$

Nothing to prove. (Immediate by cinvoke-aux after knowing that only rule cinvoke possibly applies).

(o) $\vdash_\delta \mathsf{stc}'$

By Definition 2, we have to prove that:
$\mathsf{mstc}(modID).\sigma + \mathit{off} + nArgs + nLocal \in [\mathsf{mstc}(modID).\sigma, \mathsf{mstc}(modID).e)$.
By unfolding assumption **Related local stack usage** using Definition 63, we obtain (*):

$$
\begin{aligned}
&\forall fid \in \mathsf{dom}(Fd), mid.\ \mathtt{moduleID}(Fd(fid)) = mid \implies \\
&(\mathtt{frameSize}(Fd(fid)) + \Sigma(mid).1 + \Phi(mid) < \Sigma(mid).2 \iff \\
&\phi(mid, fid).1 + \phi(mid, fid).2 + \mathsf{mstc}(mid).\sigma + \mathsf{mstc}(mid).\mathit{off} < \mathsf{mstc}(mid).e)
\end{aligned}
$$

which we instantiate using fid_{call} and assumptions (3.) and (11.) respectively to immediately obtain our goal (after simple interval arithmetic).

We also have to prove:
$K_{mod}; K_{fun}; \Sigma; \Delta; \beta; MVar; Fd; \langle Mem', stk', pc', \Phi', nalloc' \rangle \cong_{modIDs}$
$\langle \mathcal{M}_c, \mathcal{M}'_d, stk', imp, \phi, \mathsf{ddc}', \mathsf{stc}', \mathsf{pcc}', \mathsf{nalloc}' \rangle$.
By unfolding Definition 64, we obtain the following subgoals:

(p) $nalloc' = \mathsf{nalloc}'$
Immediate from the assumption **Equal allocation** after substitution.

(q) $A'_s = \mathsf{reachable\_addresses}(\Sigma, \Delta, modIDs, Mem') \ \wedge$
$A'_t = \mathsf{reachable\_addresses}(\bigcup_{mid \in modIDs} \{imp'(mid).\mathsf{ddc}, \mathsf{mstc}'(mid)\}, \mathcal{M}'_d) \ \wedge$
$A'_s = A'_t \ \wedge \ Mem'|_{A'_s} = \mathcal{M}'_d|_{A'_t}$
This is similar to the corresponding subgoal (i.e., (j)) of case Assign-to-var-or-arr.
We sketch the differences:

- First, we prove that $\phi(modID, fid_{call}) = (nArgs, nLocal)$ (i.e., we prove that $nLocal = nLocal$)
  After unfolding the definitions of $argNames$ and $localIDs$, we can apply Lemma 91 to our goal to obtain subgoals that are provable using:
  assumption (6.), and
  $[\![mods_1]\!]_{\Delta,\Sigma,\beta,K_{mod},K_{fun}} = t$.
- We then prove our goal by induction on $nArgs + nLocal$.
- In the k-th induction step, we distinguish two cases:
  - Case $k \in [0, nArgs)$:
    Here, we know from subgoal (c) about v_i that we can apply Lemma 95 obtaining subgoals that are provable similarly to subgoal (j) of case Assign-to-var-or-arr.
  - Case $k \in [nArgs, nArgs + nLocal)$
    Here, we know from subgoal (k) that we can apply Lemma 95 obtaining subgoals that are provable similarly to subgoal (j) of case Assign-to-var-or-arr.

(r) $\Delta(\mathtt{moduleID}(Fd(pc'.fid))) = (\mathsf{ddc}'.\sigma, \mathsf{ddc}'.e)$
This is immediate by Lemma 91.

(s) $\Sigma(\mathtt{moduleID}(Fd(pc'.fid))) = (\mathsf{stc}'.\sigma, \mathsf{stc}'.e)$
This is also immediate by Lemma 91.

(t) $\Phi(\mathtt{moduleID}(Fd(pc'.fid))) = \mathsf{stc}'.off$
This is provable using assumption **Related local stack usage**.

(u) $K_{mod}(\mathtt{moduleID}(Fd(pc'.fid))).1 + K_{fun}(pc'.fid).1 + pc'.n = \mathsf{pcc}'.\sigma + \mathsf{pcc}'.off \ \wedge$
$K_{mod}(\mathtt{moduleID}(Fd(pc'.fid))) = [\mathsf{pcc}'.\sigma, \mathsf{pcc}'.e)$
Immediate by the already-established subgoals ((n) and (g)), and Lemma 91.

(v) $K_{mod}; K_{fun}; \Sigma; \Delta; \beta; MVar; Fd; stk' \cong stk'$
By unfolding Definition 62, our goal follows easily from assumption **Related program counters**.

(w) $K_{mod}; K_{fun}; \Sigma; \Delta; \beta; MVar; Fd; \Phi' \cong \mathsf{mstc}', \phi$

By Definition 63, our goal is:

$$\forall mid \in \mathsf{dom}(\Phi').\ \Phi'(mid) = \mathsf{mstc}'(mid).\mathit{off}$$
$$\wedge$$
$$\forall fid \in \mathsf{dom}(Fd), mid.\ \mathtt{moduleID}(Fd(fid)) = mid \implies$$
$$(\mathtt{frameSize}(Fd(fid)) + \Sigma(mid).1 + \Phi'(mid) < \Sigma(mid).2 \iff$$
$$\phi(mid, fid).1 + \phi(mid, fid).2 + \mathsf{mstc}'(mid).\sigma + \mathsf{mstc}'(mid).\mathit{off} < \mathsf{mstc}'(mid).e)$$
$$\wedge$$
$$\forall fid \in \mathsf{dom}(Fd), mid.\ \mathtt{moduleID}(Fd(fid)) = mid \implies$$
$$\mathtt{length}(\mathtt{args}(Fd(fid))) = \phi(mid, fid).1$$
$$\wedge$$
$$\forall (mid, fid) \in \mathsf{dom}(\phi).\ fid\ \in \mathsf{dom}(Fd)\ \wedge\ mid = \mathtt{moduleID}(Fd(fid))$$

- The first conjunct is immediate by assumption **Related local stack usage** (after unfolding Definition 63) together with assumption (12.) and subgoals (l), (i) and (h).
- For the second conjunct, we fix arbitrary fid and mid, then we distinguish two cases:
  - **Case** $mid = \mathtt{moduleID}(Fd(fid_{call}))$**:**
    Here, the " $\implies$ " direction of our goal follows from subgoal (o) after substitution using subgoal (l).
    And the " $\impliedby$ " direction follows from assumptions (11.) and (12.).
  - **Case** $mid \neq \mathtt{moduleID}(Fd(fid_{call}))$**:**
    Here, our goal is immediate by assumption **Related local stack usage** after substitution using $\mathsf{mstc}'(mid) = \mathsf{mstc}(mid)$ of subgoal (l), and $\Phi'(mid) = \Phi(mid)$ of assumption (12.).
- The remaining subgoals are immediate by assumption **Related local stack usage**.

Case Return:

In this case, by inversion, we have the following assumptions:

1. $(fid, n) = pc$
2. $\mathtt{commands}(Fd(fid))(n) = \mathsf{Return}$
3. $(pc', stk') = \mathtt{pop}(stk)$
4. $pc' = (fid', \_)$
5. $curFrameSize = \mathtt{frameSize}(Fd(fid))$
6. $curModID = \mathtt{moduleID}(Fd(fid))$
7. $\Phi' = \Phi[curModID \mapsto \Phi(curModID) - curFrameSize]$

And we would like to prove the first subgoal:

$\langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle \to \langle \mathcal{M}_c, \mathcal{M}_d', stk', imp, \phi, \mathsf{ddc}', \mathsf{stc}', \mathsf{pcc}', \mathsf{nalloc}'\rangle$

By inversion using rule creturn, we obtain the following subgoals:

(a) $\vdash_\kappa \mathsf{pcc}$

Same as in the previous cases.

(b) $\mathcal{M}_c(\mathsf{pcc}) = \mathsf{Creturn}$

This follows immediately by Lemma 93 and definition 59
after replacing $\mathsf{pcc}.\sigma + \mathsf{pcc}.\mathit{off}$.

(c) $stk', (\mathsf{ddc}', \mathsf{pcc}', mid, fid) = \mathtt{pop}(stk)$

The fact that $\mathtt{pop}(stk)$ is defined can be proved by showing that:

$stk \neq \mathtt{nil}$

Assume for the sake of contradiction that (STK-NIL):

$stk = \mathtt{nil}$

Thus, $\mathtt{length}(stk) = 0$.

Thus, by assumption **Related trusted stacks** (unfolding Definition 62), we obtain f with $f(-1) = -1$ and $f(\mathtt{length}(stk)) = 0$.

Since we know by assumption 3 that $\mathtt{length}(stk) > 0$, we instantiate the " $\Longleftarrow$ " direction of conjunct "+1 preservation" of assumption **Related trusted stacks** (unfolding Definition 62), obtaining a contradiction.

Thus, assumption (STK-NIL) must be false which is our goal.

(d) $\phi(mid, fid) = (nArgs, nLocal)$

Using assumption **Execution in compile code**, and from Lemma 91, we know that $\phi(\mathtt{moduleID}(Fd(pc.fid)), pc.fid)$ exists.

Furthermore, by the definition of $\mathtt{frameSize}$, we can conclude that (##):

$nArgs + nLocal = curFrameSize$ (from assumption (5.))

(e) $(\delta, s, e, off) = \mathsf{mstc}(mid)$

Again, from Lemma 91, we know that $\mathsf{mstc}(mid)$ exists.

(f) $off' = off - nArgs - nLocal$,

(g) $\mathsf{mstc}' = \mathsf{mstc}[mid \mapsto (\delta, s, e, off')]$

Nothing to prove.

(h) $\exists mid'.\ \mathsf{pcc}' \doteq imp(mid').\mathsf{pcc} \ \wedge\ \mathsf{stc}' = \mathsf{mstc}(mid')$

For the first conjunct, it suffices by rule exec-state to prove:

$t \vdash_{exec} \langle \mathcal{M}_c, \mathcal{M}_d', stk', imp, \phi, \mathsf{ddc}', \mathsf{stc}', \mathsf{pcc}', \mathsf{mstc}', \mathsf{nalloc}' \rangle$.

The latter follows from the assumption $t \vdash_{exec} \langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc} \rangle$ by Lemma 52.

For second conjunct, all we need is to prove that $mid' \in \mathsf{dom}(\mathsf{mstc})$.

This follows from the precondition $\mathsf{dom}(imp) = \mathsf{dom}(\mathsf{mstc})$ of also rule exec-state.

We also have to prove:

$K_{mod}; K_{fun}; \Sigma; \Delta; \beta; MVar; Fd; \langle Mem', stk', pc', \Phi', nalloc' \rangle \cong_{modIDs}$
$\langle \mathcal{M}_c, \mathcal{M}_d', stk', imp, \phi, \mathsf{ddc}', \mathsf{stc}', \mathsf{pcc}', \mathsf{nalloc}' \rangle$.

By unfolding Definition 64, we obtain the following subgoals:

(i) $nalloc' = \mathsf{nalloc}'$

This is immediate by assumption **Equal allocation** after substitution.

(j) $A_s' = \mathsf{reachable\_addresses}(\Sigma, \Delta, modIDs, Mem') \ \wedge$
$A_t' = \mathsf{reachable\_addresses}(\bigcup_{mid \in modIDs} \{imp'(mid).\mathsf{ddc}, \mathsf{mstc}'(mid)\}, \mathcal{M}_d') \ \wedge$
$A_s' = A_t' \ \wedge\ Mem'|_{A_s'} = \mathcal{M}_d'|_{A_t'}$

This is immediate (after substitution) by assumption **Equal reachable memories**.

(k) $\Delta(\mathtt{moduleID}(Fd(pc'.fid))) = (\mathsf{ddc}'.\sigma, \mathsf{ddc}'.e)$

By assumption **Related trusted stacks** (unfolding Definition 62), we know that:
$K_{mod}(\texttt{moduleID}(Fd(pc'.fid))) = [\mathsf{pcc}'.\sigma, \mathsf{pcc}'.e)$

Thus, immediately, by exec-state, and the disjointness constraints of valid-linking, we know that:
$imp(\texttt{moduleID}(Fd(pc'.fid))).\mathsf{ddc} \doteq \mathsf{ddc}'$

This (after unfolding Definition 6) suffices for our goal by Lemma 91.

(l) $\Sigma(\texttt{moduleID}(Fd(pc'.fid))) = (\mathsf{stc}'.\sigma, \mathsf{stc}'.e)$

Again, by assumption **Related trusted stacks** (unfolding Definition 62), we know that:
$K_{mod}(\texttt{moduleID}(Fd(pc'.fid))) = [\mathsf{pcc}'.\sigma, \mathsf{pcc}'.e)$

Thus, immediately, by exec-state, and the disjointness constraints of valid-linking, we know that:
$\mathsf{mstc}(\texttt{moduleID}(Fd(pc'.fid))) \doteq \mathsf{stc}'$

This (after unfolding Definition 6) suffices for our goal by Lemma 91.

(m) $\Phi(\texttt{moduleID}(Fd(pc'.fid))) = \mathsf{stc}'.off$

This follows from the assumption **Related local stack usage**.

(n) $K_{mod}(\texttt{moduleID}(Fd(pc'.fid))).1 + K_{fun}(pc'.fid).1 + pc'.n = \mathsf{pcc}'.\sigma + \mathsf{pcc}'.off \ \wedge$
$K_{mod}(\texttt{moduleID}(Fd(pc'.fid))) = [\mathsf{pcc}'.\sigma, \mathsf{pcc}'.e)]$

This follows from assumption **Related trusted stacks** (unfolding Definition 62). here is how:

Using assumption 3 and subgoal (c), together with folding Definition 61, it suffices to show that:
$K_{mod}; K_{fun}; \Sigma; \Delta; \beta; MVar; Fd; stk(\texttt{length}(stk) - 1) \cong stk(\texttt{length}(stk) - 1).\mathsf{pcc}$

The latter is immediate by unfolding assumption **Related trusted stacks** using Definition 62.

(o) $K_{mod}; K_{fun}; \Sigma; \Delta; \beta; MVar; Fd; stk' \cong stk'$

Follows easily from assumption **Related trusted stacks** (unfolding Definition 62), assumption 3, and subgoal (c).

(p) $K_{mod}; K_{fun}; \Sigma; \Delta; \beta; MVar; Fd; \Phi' \cong_{modIDs} \mathsf{mstc}', \phi$

By Definition 63, our goal is:

$$
\begin{array}{l}
\forall mid \in \mathsf{dom}(\Phi').\ \Phi'(mid) = \mathsf{mstc}'(mid).off \\
\wedge \\
\forall fid \in \mathsf{dom}(Fd), mid.\ \texttt{moduleID}(Fd(fid)) = mid \implies \\
(\texttt{frameSize}(Fd(fid)) + \Sigma(mid).1 + \Phi'(mid) < \Sigma(mid).2 \iff \\
\phi(mid, fid).1 + \phi(mid, fid).2 + \mathsf{mstc}'(mid).\sigma + \mathsf{mstc}'(mid).off < \mathsf{mstc}'(mid).e) \\
\wedge \\
\forall fid \in \mathsf{dom}(Fd), mid.\ \texttt{moduleID}(Fd(fid)) = mid \implies \\
\texttt{length}(\texttt{args}(Fd(fid))) = \phi(mid, fid).1 \\
\wedge \\
\forall (mid, fid) \in \mathsf{dom}(\phi).\ fid \ \in \mathsf{dom}(Fd) \ \wedge \ mid = \texttt{moduleID}(Fd(fid))
\end{array}
$$

- For the first conjunct, we fix an arbitrary mid and distinguish the following two cases:
  - **Case** $mid = \texttt{moduleID}(Fd(pc.fid))$**:**
    Here, after substitution using assumptions (5.), and (7.), and subgoals (e) and (h), our goal follows from assumption **Related local stack usage**.

- **Case** $mid \neq \texttt{moduleID}(Fd(pc.fid))$**:**
  Here, our goal follows after substitution using assumption (7.) and subgoal (h) from assumption **Related local stack usage**.

- For the second conjunct, we fix arbitrary fid and mid and again distinguish the following two cases:
  - **Case** $mid = \texttt{moduleID}(Fd(pc.fid))$**:**
    Here, both the " $\Longrightarrow$ " and " $\Longleftarrow$ " directions follow by substitution using Lemma 91.
  - **Case** $mid \neq \texttt{moduleID}(Fd(pc.fid))$**:**
    Here, our goal follows after substitution using assumption (7.) and subgoal (h) from assumption **Related local stack usage**.
- The remaining conjuncts are immediate by assumption **Related local stack usage**.

Case Jump-non-zero:

In this case, by inversion, we have the following assumptions:

1. $(fid, n) = pc$
2. $\texttt{commands}(Fd(fid))(n) = \mathsf{JumpIfZero}\ e_c\ e_{off}$
3. $e_c, \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow v$
4. $v \neq 0$
5. $pc' = \texttt{inc}(pc)$

And we would like to prove the first subgoal:
$\langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle \to \langle \mathcal{M}_c, \mathcal{M}'_d, stk', imp, \phi, \mathsf{ddc}', \mathsf{stc}', \mathsf{pcc}', \mathsf{nalloc}'\rangle$
By inversion using rule jump1, we obtain the following subgoals:

(a) $\vdash_\kappa \mathsf{pcc}$
Same as in the previous cases.

(b) $\mathcal{M}_c(\mathsf{pcc}) = \mathsf{JumpIfZero}\ \mathcal{E}_{cond}\ \mathcal{E}_{off}$
This follows immediately by Lemma 93 and definition 59
after replacing $\mathsf{pcc}.\sigma + \mathsf{pcc}.off$.
By Definition 59, we have the following substitution which we use in the coming goals:
$\mathcal{E}_{cond} = \wr e_c \int_{fid,mid,\beta}$

(c) $\mathcal{E}_{cond}, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow v$, and

(d) $v \neq 0$
After the substitution, and by assumption **Equal reachable memories**, we can apply Lemma 88 for these two subgoals (we have all the assumptions).
From assumption $v \neq 0$, we thus conclude $v \neq 0$.

(e) $\mathsf{pcc}' = \texttt{inc}(\mathsf{pcc}, 1)$
Immediate by rule jump1.

We also have to prove:
$K_{mod}; K_{fun}; \Sigma; \Delta; \beta; MVar; Fd; \langle Mem', stk', pc', \Phi', nalloc'\rangle \cong_{modIDs}$
$\langle \mathcal{M}_c, \mathcal{M}'_d, stk', imp, \phi, \mathsf{ddc}', \mathsf{stc}', \mathsf{pcc}', \mathsf{nalloc}'\rangle$.
By unfolding Definition 64, we obtain the following subgoals:

(f) $\mathit{nalloc}' = \mathsf{nalloc}'$
Immediate by assumption after substitution using the preconditions $\mathit{nalloc}' = \mathit{nalloc}$ and $\mathsf{nalloc}' = \mathsf{nalloc}$ (of rule jump1).

(g) $A_s' = \mathsf{reachable\_addresses}(\Sigma, \Delta, \mathit{modIDs}, \mathit{Mem}') \ \wedge$
$A_t' = \mathsf{reachable\_addresses}(\bigcup_{\mathit{mid} \in \mathit{modIDs}} \{\mathit{imp}'(\mathit{mid}).\mathsf{ddc}, \mathsf{mstc}'(\mathit{mid})\}, \mathcal{M}_d') \ \wedge$
$A_s' = A_t' \ \wedge \ \mathit{Mem}'|_{A_s'} = \mathcal{M}_d'|_{A_t'}$
Immediate by assumptions after rewriting using $\mathcal{M}_d' = \mathcal{M}_d$ and $\mathit{Mem}' = \mathit{Mem}$.

(h) $\Delta(\mathtt{moduleID}(\mathit{Fd}(\mathit{pc}'.\mathit{fid}))) = (\mathsf{ddc}'.\sigma, \mathsf{ddc}'.e)$
Immediate by assumptions after rewriting using $\mathsf{ddc}' = \mathsf{ddc}$ and $\mathit{pc}'.\mathit{fid} = \mathit{pc}.\mathit{fid}$.

(i) $\Sigma(\mathtt{moduleID}(\mathit{Fd}(\mathit{pc}'.\mathit{fid}))) = (\mathsf{stc}'.\sigma, \mathsf{stc}'.e)$
Immediate by assumptions after rewriting using $\mathsf{stc}' = \mathsf{stc}$ and $\mathit{pc}'.\mathit{fid} = \mathit{pc}.\mathit{fid}$.

(j) $\Phi(\mathtt{moduleID}(\mathit{Fd}(\mathit{pc}'.\mathit{fid}))) = \mathsf{stc}'.\mathit{off}$
Immediate by assumptions after rewriting using $\mathsf{stc}' = \mathsf{stc}$ and $\mathit{pc}'.\mathit{fid} = \mathit{pc}.\mathit{fid}$.

(k) $K_{\mathit{mod}}(\mathtt{moduleID}(\mathit{Fd}(\mathit{pc}'.\mathit{fid}))).1 + K_{\mathit{fun}}(\mathit{pc}'.\mathit{fid}).1 + \mathit{pc}'.n = \mathsf{pcc}'.\sigma + \mathsf{pcc}'.\mathit{off} \ \wedge$
$K_{\mathit{mod}}(\mathtt{moduleID}(\mathit{Fd}(\mathit{pc}'.\mathit{fid}))) = [\mathsf{pcc}'.\sigma, \mathsf{pcc}'.e)]$
This is immediate after substitution using the assumptions on pcc and pc and after having proved
$\mathsf{pcc}' = \mathtt{inc}(\mathsf{pcc}, 1)$.

(l) $K_{\mathit{mod}}; K_{\mathit{fun}}; \Sigma; \Delta; \beta; \mathit{MVar}; \mathit{Fd}; \mathit{stk}' \cong \mathit{stk}'$
Immediate by assumption after rewriting using $\mathit{stk}' = \mathit{stk}$ and $\mathit{stk}' = \mathit{stk}$.

(m) $K_{\mathit{mod}}; K_{\mathit{fun}}; \Sigma; \Delta; \beta; \mathit{MVar}; \mathit{Fd}; \Phi' \cong \mathsf{mstc}', \phi$
Immediate by assumption after rewriting using $\Phi' = \Phi$ and $\mathsf{mstc}' = \mathsf{mstc}$.

Case Jump-zero:
In this case, by inversion, we have the following assumptions:

1. $(\mathit{fid}, n) = \mathit{pc}$
2. $\mathtt{commands}(\mathit{Fd}(\mathit{fid}))(n) = \mathsf{JumpIfZero}\ e_c\ e_{\mathit{off}}$
3. $e_c, \Sigma, \Delta, \beta, \mathit{MVar}, \mathit{Fd}, \mathit{Mem}, \Phi, \mathit{pc} \Downarrow v$
4. $e_{\mathit{off}}, \Sigma, \Delta, \beta, \mathit{MVar}, \mathit{Fd}, \mathit{Mem}, \Phi, \mathit{pc} \Downarrow \mathit{off}$
5. $v = 0$
6. $\mathit{off} \in \mathbb{Z}$
7. $\mathit{pc}' = (\mathit{fid}, n + \mathit{off})$

And we would like to prove the first subgoal:
$\langle \mathcal{M}_c, \mathcal{M}_d, \mathit{stk}, \mathit{imp}, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc} \rangle \to \langle \mathcal{M}_c, \mathcal{M}_d', \mathit{stk}', \mathit{imp}, \phi, \mathsf{ddc}', \mathsf{stc}', \mathsf{pcc}', \mathsf{nalloc}' \rangle$
By inversion using rule jump0, we obtain the following subgoals:

(a) $\vdash_\kappa \mathsf{pcc}$
Same as in the previous cases.

(b) $\mathcal{M}_c(\mathsf{pcc}) = \mathsf{JumpIfZero}\ \mathcal{E}_{\mathit{cond}}\ \mathcal{E}_{\mathit{off}}$
This follows immediately by Lemma 93 and definition 59
after replacing $\mathsf{pcc}.\sigma + \mathsf{pcc}.\mathit{off}$.
By Definition 59, we have the following substitutions which we use in the coming goals:
$\mathcal{E}_{\mathit{cond}} = \wr e_c \int_{\mathit{fid},\mathit{mid},\beta}$, and
$\mathcal{E}_{\mathit{off}} = \wr e_{\mathit{off}} \int_{\mathit{fid},\mathit{mid},\beta}$

(c) $\mathcal{E}_{cond}, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow v$,

(d) $\mathcal{E}_{off}, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow off$, and

(e) $v = 0$

After the substitution, and by assumption **Equal reachable memories**, we can apply Lemma 88 for each of these subgoals (we have all the assumptions).

From assumption $v = 0$, we thus conclude $v = 0$.

From assumption $off \in \mathbb{Z}$, we conclude $off \in Z$.

(f) $\mathsf{pcc}' = \mathtt{inc}(\mathsf{pcc}, off)$

Immediate by rule jump0.

We also have to prove:
$K_{mod}; K_{fun}; \Sigma; \Delta; \beta; MVar; Fd; \langle Mem', stk', pc', \Phi', nalloc'\rangle \cong_{modIDs}$
$\langle \mathcal{M}_c, \mathcal{M}'_d, stk', imp, \phi, \mathsf{ddc}', \mathsf{stc}', \mathsf{pcc}', \mathsf{nalloc}'\rangle$.

By unfolding Definition 64, we obtain the following subgoals:

(g) $nalloc' = \mathsf{nalloc}'$

Immediate by assumption after substitution using the preconditions $nalloc' = nalloc$ and $\mathsf{nalloc}' = \mathsf{nalloc}$ (of rule jump0).

(h) $A'_s = \mathsf{reachable\_addresses}(\Sigma, \Delta, modIDs, Mem') \ \wedge$
$A'_t = \mathsf{reachable\_addresses}(\bigcup_{mid \in modIDs} \{imp'(mid).\mathsf{ddc}, \mathsf{mstc}'(mid)\}, \mathcal{M}'_d) \ \wedge$
$A'_s = A'_t \ \wedge \ Mem'|_{A'_s} = \mathcal{M}'_d|_{A'_t}$

Immediate by assumptions after rewriting using $\mathcal{M}'_d = \mathcal{M}_d$ and $Mem' = Mem$.

(i) $\Delta(\mathtt{moduleID}(Fd(pc'.fid))) = (\mathsf{ddc}'.\sigma, \mathsf{ddc}'.e)$

Immediate by assumptions after rewriting using $\mathsf{ddc}' = \mathsf{ddc}$ and $pc'.fid = pc.fid$.

(j) $\Sigma(\mathtt{moduleID}(Fd(pc'.fid))) = (\mathsf{stc}'.\sigma, \mathsf{stc}'.e)$

Immediate by assumptions after rewriting using $\mathsf{stc}' = \mathsf{stc}$ and $pc'.fid = pc.fid$.

(k) $\Phi(\mathtt{moduleID}(Fd(pc'.fid))) = \mathsf{stc}'.off$

Immediate by assumptions after rewriting using $\mathsf{stc}' = \mathsf{stc}$ and $pc'.fid = pc.fid$.

(l) $K_{mod}(\mathtt{moduleID}(Fd(pc'.fid))).1 + K_{fun}(pc'.fid).1 + pc'.n = \mathsf{pcc}'.\sigma + \mathsf{pcc}'.off \ \wedge$
$K_{mod}(\mathtt{moduleID}(Fd(pc'.fid))) = [\mathsf{pcc}'.\sigma, \mathsf{pcc}'.e)]$

This is immediate after substitution using the assumptions on pcc and pc and after having proved
$\mathsf{pcc}' = \mathtt{inc}(\mathsf{pcc}, off)$.

(m) $K_{mod}; K_{fun}; \Sigma; \Delta; \beta; MVar; Fd; stk' \cong stk'$

Immediate by assumption after rewriting using $stk' = stk$ and $stk' = stk$.

(n) $K_{mod}; K_{fun}; \Sigma; \Delta; \beta; MVar; Fd; \Phi' \cong \mathsf{mstc}', \phi$

Immediate by assumption after rewriting using $\Phi' = \Phi$ and $\mathsf{mstc}' = \mathsf{mstc}$.

Case Exit:

In this case, by inversion, we have the following assumptions:

1. $(fid, n) = pc$
2. $\mathtt{commands}(Fd(fid))(n) = \mathsf{Exit}$

And we would like to prove the first subgoal:

$\langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle \to \langle \mathcal{M}_c, \mathcal{M}'_d, stk', imp, \phi, \mathsf{ddc}', \mathsf{stc}', \mathsf{pcc}', \mathsf{nalloc}'\rangle$

By inversion using rule cexit, we obtain the following subgoals:

(a) $\vdash_\kappa \mathsf{pcc}$

Same as in the previous cases.

(b) $\mathcal{M}_c(\mathsf{pcc}) = \mathsf{Exit}$

This follows immediately by Lemma 93 and definition 59 after replacing $\mathsf{pcc}.\sigma + \mathsf{pcc}.\mathit{off}$.

(All the remaining subgoals are immediate from the assumptions after substitution.)

This concludes the proof of Lemma 97. □

Lemma 98 (Compiler backward simulation)**.**

$\forall K_{mod}, K_{fun}, \Sigma; \Delta; \beta; MVar; Fd, \langle Mem, stk, pc, \Phi, nalloc\rangle, \overline{mods_1}, t$
$\mathcal{M}_c, \mathcal{M}_d, imp, \mathsf{mstc}, \phi.$
$[\![\overline{mods_1}]\!]_{\Delta,\Sigma,\beta,K_{mod},K_{fun}} = t \wedge$
$K_{mod}; K_{fun}; \overline{mods_1}; \Sigma; \Delta; \beta; MVar; Fd \vdash_{exec} \langle Mem, stk, pc, \Phi, nalloc\rangle \wedge$
$t \vdash_{exec} \langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle \wedge$
$modIDs = \{modID \mid (modID, \_, \_) \in \overline{mods_1}\} \wedge$
$K_{mod}; K_{fun}; \Sigma; \Delta; \beta; MVar; Fd; \langle Mem, stk, pc, \Phi, nalloc\rangle \cong_{modIDs} \langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle \wedge$
$\langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle \to \langle \mathcal{M}_c, \mathcal{M}'_d, stk', imp, \phi, \mathsf{ddc}', \mathsf{stc}', \mathsf{pcc}', \mathsf{nalloc}'\rangle$
$\implies$
$\Sigma; \Delta; \beta; MVar; Fd \vdash \langle Mem, stk, pc, \Phi, nalloc\rangle \to \langle Mem', stk', pc', \Phi', nalloc'\rangle \wedge$
$K_{mod}; K_{fun}; \Sigma; \Delta; \beta; MVar; Fd; \langle Mem', stk', pc', \Phi', nalloc'\rangle \cong_{modIDs} \langle \mathcal{M}_c, \mathcal{M}'_d, stk', imp, \phi, \mathsf{ddc}', \mathsf{stc}', \mathsf{pcc}', \mathsf{nalloc}'\rangle$

Proof.

- We assume the antecedents, and we assume for the sake of contradiction that (ASSM-NO-SRC-STEP):
  $\nexists Mem', stk', pc', nalloc'.\ \Sigma; \Delta; \beta; MVar; Fd \vdash \langle Mem, stk, pc, \Phi, nalloc\rangle \to \langle Mem', stk', pc', \Phi', nalloc'\rangle$

- Using assumptions
  $[\![\overline{mods_1}]\!]_{\Delta,\Sigma,\beta,K_{mod},K_{fun}} = t$,
  $modIDs = \{modID \mid (modID, \_, \_) \in \overline{mods_1}\}$, and
  $K_{mod}; K_{fun}; \Sigma; \Delta; \beta; MVar; Fd; \langle Mem, stk, pc, \Phi, nalloc\rangle \cong_{modIDs}$
  $\langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle$,
  we know by Lemma 93, and Definitions 61 and 64 that (CURR-COM-COMPILED):
  $\mathcal{M}_c(\mathsf{pcc}) = (\!|\mathtt{commands}(Fd(pc.fid))(pc.n)|\!)_{Fd, K_{fun}, pc.fid, \mathtt{moduleID}(pc.fid), \beta}$

- We consider the following possible cases of the assumption (TRG-STEPS):
  $\langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle \to \langle \mathcal{M}_c, \mathcal{M}'_d, stk', imp, \phi, \mathsf{ddc}', \mathsf{stc}', \mathsf{pcc}', \mathsf{nalloc}'\rangle$,
  and derive a contradiction to (ASSM-NO-SRC-STEP) for each case.

  Case assign:

  In this case, by inversion, we have the following assumptions:

1. $\vdash_\kappa \mathsf{pcc}$
2. $\mathsf{pcc}' = \mathtt{inc}(\mathsf{pcc}, 1)$
3. $\mathcal{M}_c(\mathsf{pcc}) = \mathsf{Assign}\ \mathcal{E}_L\ \mathcal{E}_R$
4. $\mathcal{E}_R, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow v$
5. $\mathcal{E}_L, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow c$
6. $\vdash_\delta c$
7. $\vDash_\delta v \implies (v \cap \mathsf{stc} = \emptyset \lor c \subseteq \mathsf{stc})$
8. $\mathcal{M}'_d = \mathcal{M}_d[c \mapsto v]$

By unfolding assumption (CURR-COM-COMPILED) using Definition 59, we conclude:

$\mathtt{commands}(Fd(pc.fid))(pc.n) = \mathsf{Assign}\ e_l\ e_r$
with $\mathcal{E}_L = \wr e_l \int_{pc.fid, \mathtt{moduleID}(pc.fid), \beta}$, and
$\mathcal{E}_R = \wr e_r \int_{pc.fid, \mathtt{moduleID}(pc.fid), \beta}$
To contradict (ASSM-NO-SRC-STEP), we have the following subgoals using rule Assign-to-var-or-arr:

- $(fid, n) = pc$, and
- $\mathtt{commands}(Fd(fid))(n) = \mathsf{Assign}\ e_l\ e_r$
  Proved above.
- $frameSize = \mathtt{frameSize}(Fd(fid))$
  Nothing to prove.
- $e_l, \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow (\delta, s, e, off)$, and
- $e_r, \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow v$
  Follow from Lemma 90, and we obtain $v = v$ and $(\delta, s, e, off) = c$.
- $modID = \mathtt{moduleID}(Fd(fid))$
  Existence of $Fd(fid)$ is immediate by assumption.
- $\phi = \Sigma(modID).1 + \Phi(modID)$
  Nothing to prove.
- $\forall s', e'.\ v = (\delta, s', e', \_) \implies ([s', e') \cap \Sigma(modID) = \emptyset \lor [s, e) \subseteq \Sigma(modID))$
  Follows from assumption (7), after substitution using assumption
  $K_{mod}; K_{fun}; \Sigma; \Delta; \beta; MVar; Fd; \langle Mem, stk, pc, \Phi, nalloc\rangle \cong_{modIDs}$
  $\langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle$ (unfolding Definition 64).
- $s \leq s + off < e$
  Immediate by assumption (6) after substitution using $(\delta, s, e, off) = c$ (obtained above).
- $Mem' = Mem[s + off \mapsto v]$
  Nothing to prove.

Case allocate:

In this case, by inversion, we have the following assumptions:

1. $\vdash_\kappa \mathsf{pcc}$
2. $\mathsf{pcc}' = \mathtt{inc}(\mathsf{pcc}, 1)$
3. $\mathcal{M}_c(\mathsf{pcc}) = \mathsf{Alloc}\ \mathcal{E}_L\ \mathcal{E}_{size}$
4. $\mathcal{E}_{size}, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow v$
5. $\mathcal{E}_L, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow c$
6. $v \in \mathbb{Z}^+$
7. $\vdash_\delta c$
8. $\mathcal{M}'_d = \mathcal{M}_d[c \mapsto (\delta, \mathsf{nalloc} - v, \mathsf{nalloc}, 0), i \mapsto 0\ \forall i \in [\mathsf{nalloc} - v, \mathsf{nalloc})]$
9. $\mathsf{nalloc}' = \mathsf{nalloc} - v$

10. $\mathsf{nalloc}' > \nabla$

By unfolding assumption (CURR-COM-COMPILED) using Definition 59, we conclude:

$\texttt{commands}(Fd(pc.fid))(pc.n) = \mathsf{Alloc}\ e_l\ e_{size}$
with $\mathcal{E}_L = \wr e_l \int_{pc.fid,\texttt{moduleID}(pc.fid),\beta}$, and
$\mathcal{E}_{size} = \wr e_{size} \int_{pc.fid,\texttt{moduleID}(pc.fid),\beta}$
To contradict (ASSM-NO-SRC-STEP), we have the following subgoals using rule Allocate:

- $(fid, n) = pc$, and
- $\texttt{commands}(Fd(fid))(n) = \mathsf{Alloc}\ e_l\ e_{size}$
  Proved above.
- $e_l, \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow (\delta, s, e, off)$, and
- $e_{size}, \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow v$
  Follow from Lemma 90, and we obtain $v = v$ and $(\delta, s, e, off) = c$.
- $s \leq s + off < e$
  Immediate (after substitution) by assumption (7) (unfolding Definition 2).
- $v \in \mathbb{Z}^+$
  Immediate by assumption (6) after substitution using $v = v$.
- $nalloc - v > \nabla$
  Immediate by assumption $\mathsf{nalloc}' > \nabla$ after substitution.
- $nalloc' = nalloc - v$, and
- $Mem' = Mem[s + off \mapsto (\delta, nalloc', nalloc, 0)][a \mapsto 0 \mid a \in [nalloc', nalloc)]$
  Nothing to prove.

Case jump0:

In this case, by inversion, we have the following assumptions:

1. $\vdash_\kappa \mathsf{pcc}$
2. $\mathcal{M}_c(\mathsf{pcc}) = \mathsf{JumpIfZero}\ \mathcal{E}_{cond}\ \mathcal{E}_{off}$
3. $\mathcal{E}_{cond}, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow v$
4. $v = 0$
5. $\mathcal{E}_{off}, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow off$
6. $off \in \mathbb{Z}$
7. $\mathsf{pcc}' = \texttt{inc}(\mathsf{pcc}, off)$

By unfolding assumption (CURR-COM-COMPILED) using Definition 59, we conclude:

$\texttt{commands}(Fd(pc.fid))(pc.n) = \mathsf{JumpIfZero}\ e_c\ n_{dest}$
with $\mathcal{E}_{cond} = \wr e_c \int_{pc.fid,\texttt{moduleID}(pc.fid),\beta}$, and $\mathcal{E}_{off} = \wr e_{off} \int_{pc.fid,\texttt{moduleID}(pc.fid),\beta}$
To contradict (ASSM-NO-SRC-STEP), we have the following subgoals using rule Jump-zero:

- $(fid, n) = pc$, and
- $\texttt{commands}(Fd(fid))(n) = \mathsf{JumpIfZero}\ e_c\ e_{off}$
  Proved above.
- $e_c, \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow v$, and
- $v = 0$
  Follow from Lemma 90 by assumptions (3.) and (4.).

Case jump1:

In this case, by inversion, we have the following assumptions:

1. $\vdash_\kappa \mathsf{pcc}$
2. $\mathcal{M}_c(\mathsf{pcc}) = \mathsf{JumpIfZero}\ \mathcal{E}_{cond}\ \mathcal{E}_{off}$
3. $\mathcal{E}_{cond}, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow v$
4. $v \neq 0$
5. $\mathsf{pcc}' = \mathtt{inc}(\mathsf{pcc}, 1)$

By unfolding assumption (CURR-COM-COMPILED) using Definition 59, we conclude:

$\mathtt{commands}(Fd(pc.fid))(pc.n) = \mathsf{JumpIfZero}\ e_c\ n_{dest}$
with $\mathcal{E}_{cond} = \wr e_c \int_{pc.fid,\mathtt{moduleID}(pc.fid),\beta}$, and $\mathcal{E}_{off} = \wr e_{off} \int_{pc.fid,\mathtt{moduleID}(pc.fid),\beta}$
To contradict (ASSM-NO-SRC-STEP), we have the following subgoals using rule Jump-non-zero:

- $(fid, n) = pc$, and
- $\mathtt{commands}(Fd(fid))(n) = \mathsf{JumpIfZero}\ e_c\ n_{dest}$
  Proved above.
- $e_c, \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow v$, and
- $v \neq 0$
  Follow from Lemma 90 by assumptions (3.) and (4.).

Case cinvoke:
In this case, by inversion, we have the following assumptions:

1. $\vdash_\kappa \mathsf{pcc}$
2. $\mathcal{M}_c(\mathsf{pcc}) = \mathsf{Cinvoke}\ mid\ fid\ \overline{e}$
3. $stk' = \mathsf{push}(stk, (\mathsf{ddc}, \mathsf{pcc}, mid, fid))$
4. $\phi(mid, fid) = (nArgs, nLocal)$
5. $(\delta, s, e, off) = \mathsf{mstc}(mid)$
6. $off' = off + nArgs + nLocal$
7. $\mathsf{stc}' = (\delta, s, e, off')$
8. $\overline{e}(i), \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow v_i\ \forall i \in [0, nArgs)$
9. $\forall i \in [0, nArgs).\ \vDash_\delta v_i \implies v_i \cap \mathsf{stc} = \emptyset$
10. $\mathcal{M}_d' = \mathcal{M}_d[s + off + i \mapsto v_i\ \forall i \in [0, nArgs)][s + off + nArgs + i \mapsto 0\ \forall i \in [0, nLocal)]$
11. $\mathsf{mstc}' = \mathsf{mstc}[mid \mapsto \mathsf{stc}']$
12. $(c, d, offs) = imp(mid)$
13. $\mathsf{ddc}' = d$
14. $\mathsf{pcc}' = \mathtt{inc}(c, offs(fid))$
15. $\vdash_\delta \mathsf{stc}'$

By unfolding assumption (CURR-COM-COMPILED) using Definition 59, we conclude:

$\mathtt{commands}(Fd(pc.fid))(pc.n) = \mathsf{Call}\ fid_{call}\ \overline{e}$
with $mid = \mathtt{moduleID}(Fd(fid_{call}))$,
$fid = fid_{call}$, and
$\overline{e} = \wr \overline{e} \int_{pc.fid,\mathtt{moduleID}(pc.fid),\beta}$
To contradict (ASSM-NO-SRC-STEP), we have the following subgoals using rule Call:

- $(fid, n) = pc$, and
- $\mathtt{commands}(Fd(fid))(n) = \mathsf{Call}\ fid_{call}\ \overline{e}$
  Proved above.
- $modID = \mathtt{moduleID}(Fd(fid_{call}))$,
- $argNames = \mathtt{args}(Fd(fid_{call}))$,

- $\mathit{localIDs} = \texttt{localIDs}(\mathit{Fd}(\mathit{fid}_{call}))$,
- $\mathit{nArgs} = \texttt{length}(\mathit{argNames}) = \texttt{length}(\overline{e})$,
- $\mathit{nLocal} = \texttt{length}(\mathit{localIDs})$,
- $\mathit{frameSize} = \texttt{frameSize}(\mathit{Fd}(\mathit{fid}_{call}))$,
- $\mathit{curFrameSize} = \texttt{frameSize}(\mathit{Fd}(\mathit{fid}))$, and
- $\mathit{curModID} = \texttt{moduleID}(\mathit{Fd}(\mathit{fid}))$
  Nothing to prove.
- $\Sigma(\mathit{modID}).1 + \Phi(\mathit{modID}) + \mathit{frameSize} < \Sigma(\mathit{modID}).2$
  By unfolding assumption
  $K_{mod}; K_{fun}; \Sigma; \Delta; \beta; \mathit{MVar}; \mathit{Fd}; \langle \mathit{Mem}, \mathit{stk}, \mathit{pc}, \Phi, \mathit{nalloc}\rangle \cong_{\mathit{modIDs}}$
  $\langle \mathcal{M}_c, \mathcal{M}_d, \mathit{stk}, \mathit{imp}, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle$ using Definition 64 then Definition 63, we obtain (*):
  $\forall \mathit{fid} \in \mathsf{dom}(\mathit{Fd}), \mathit{mid}.\ \texttt{moduleID}(\mathit{Fd}(\mathit{fid})) = \mathit{mid} \implies$
  $(\texttt{frameSize}(\mathit{Fd}(\mathit{fid})) + \Sigma(\mathit{mid}).1 + \Phi(\mathit{mid}) < \Sigma(\mathit{mid}).2 \iff$
  $\phi(\mathit{mid}, \mathit{fid}).1 + \phi(\mathit{mid}, \mathit{fid}).2 + \mathsf{mstc}(\mathit{mid}).\sigma + \mathsf{mstc}(\mathit{mid}).\mathit{off} < \mathsf{mstc}(\mathit{mid}).e)$
  We apply (*) to our goal, then it suffices to show (after substitution using $\mathit{fid} = \mathit{fid}_{call}$ and $\mathit{mid} = \texttt{moduleID}(\mathit{Fd}(\mathit{fid}_{call}))$) that:
  $\phi(\mathit{mid}, \mathit{fid}).1 + \phi(\mathit{mid}, \mathit{fid}).2 + \mathsf{mstc}(\mathit{mid}).\sigma + \mathsf{mstc}(\mathit{mid}).\mathit{off} < \mathsf{mstc}(\mathit{mid}).e$
  This is immediate by assumptions (4.), (5.), (6.), (7.), and (15.).
- $\Phi' = \Phi[\mathit{modID} \mapsto \Phi(\mathit{modID}) + \mathit{frameSize}]$, and
- $\phi' = \Sigma(\mathit{modID}).1 + \Phi'(\mathit{modID})$
  Nothing to prove.
- $\overline{e}(i), \Sigma, \Delta, \beta, \mathit{MVar}, \mathit{Fd}, \mathit{Mem}, \Phi, \mathit{pc} \Downarrow v_i \quad \forall i \in [0, \mathit{nArgs})$
  Follows from Lemma 90 after noticing that:
  $\phi(\mathit{modID}, \mathit{fid}_{call}).1 = \texttt{length}(\texttt{args}(\mathit{Fd}(\mathit{fid}_{call})))$
  (from unfolding assumption
  $K_{mod}; K_{fun}; \Sigma; \Delta; \beta; \mathit{MVar}; \mathit{Fd}; \langle \mathit{Mem}, \mathit{stk}, \mathit{pc}, \Phi, \mathit{nalloc}\rangle \cong_{\mathit{modIDs}}$
  $\langle \mathcal{M}_c, \mathcal{M}_d, \mathit{stk}, \mathit{imp}, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle$ using Definition 64 then Definition 63)
- $\forall i \in [0, \mathit{nArgs}), s', e'.\ \ v_i = (s', e', \_) \implies [s', e') \cap \Sigma(\mathit{curModID}) = \emptyset$
  Follows from Lemma 90 and assumptions (9.) and "$\Sigma(\mathit{curModID}) = [\mathsf{stc}.\sigma, \mathsf{stc}.e)$" which is obtained by unfolding the assumptions using Definition 64.
- $\mathit{stk}' = \texttt{push}(\mathit{stk}, \mathit{pc})$,
- $\mathit{pc}' = (\mathit{fid}_{call}, 0)$, and
- $\mathit{Mem}' = \mathit{Mem}[\phi' + s_i \mapsto v_i \mid \beta(\mathit{argNames}(i)) = [s_i, \_) \wedge i \in [0, \mathit{nArgs})]$
  $[\phi' + s_i \mapsto 0 \mid \beta(\mathit{localIDs}(i)) = [s_i, \_) \wedge i \in [0, \mathit{nLocal})]$
  Nothing to prove.

Case creturn:

In this case, by inversion, we have the following assumptions:

1. $\vdash_\kappa \mathsf{pcc}$
2. $\mathcal{M}_c(\mathsf{pcc}) = \mathsf{Creturn}$
3. $\mathit{stk}', (\mathsf{ddc}', \mathsf{pcc}', \mathit{mid}, \mathit{fid}) = \mathsf{pop}(\mathit{stk})$
4. $\phi(\mathit{mid}, \mathit{fid}) = (\mathit{nArgs}, \mathit{nLocal})$
5. $(\delta, s, e, \mathit{off}) = \mathsf{mstc}(\mathit{mid})$
6. $\mathit{off}' = \mathit{off} - \mathit{nArgs} - \mathit{nLocal}$
7. $\mathsf{mstc}' = \mathsf{mstc}[\mathit{mid} \mapsto (\delta, s, e, \mathit{off}')]$
8. $\exists \mathit{mid}'.\ \mathit{imp}(\mathit{mid}').\mathsf{pcc} \doteq \mathsf{pcc}' \ \wedge\ \mathsf{stc}' = \mathsf{mstc}(\mathit{mid}')$

By unfolding assumption (CURR-COM-COMPILED) using Definition 59, we conclude:

$\texttt{commands}(\mathit{Fd}(\mathit{pc}.\mathit{fid}))(\mathit{pc}.n) = \mathsf{Return}$

To contradict (ASSM-NO-SRC-STEP), we have the following subgoals using rule Return:

– $(\mathit{fid}, n) = \mathit{pc}$, and
– $\texttt{commands}(\mathit{Fd}(\mathit{fid}))(n) = \mathsf{Return}$
Proved above.
– $(\mathit{pc}', \mathit{stk}') = \texttt{pop}(\mathit{stk})$
Here, we need to show that $\mathit{stk} \neq \texttt{nil}$.
This follows easily by assumptions unfolding Definition 64 then Definition 62, and substituting using assumption (3).
– $\mathit{pc}' = (\mathit{fid}', \_)$,
– $\mathit{curFrameSize} = \texttt{frameSize}(\mathit{Fd}(\mathit{fid}))$,
– $\mathit{curModID} = \texttt{moduleID}(\mathit{Fd}(\mathit{fid}))$, and
– $\Phi' = \Phi[\mathit{curModID} \mapsto \Phi(\mathit{curModID}) - \mathit{curFrameSize}]$
The fact that $\Phi(\mathit{curModID})$ exists follows from Lemma 91, and assumption (6.) after rewriting "$\mathit{curModID} \in \mathsf{dom}(\Phi)$" using the preconditions of rule Exec-state-src applied to our lemma's assumptions.

Case cexit:

In this case, by inversion, we have the following assumptions:

1. $\vdash_\kappa \mathsf{pcc}$
2. $\mathcal{M}_c(\mathsf{pcc}) = \mathsf{Exit}$

By unfolding assumption (CURR-COM-COMPILED) using Definition 59, we conclude:

$\texttt{commands}(\mathit{Fd}(\mathit{pc}.\mathit{fid}))(\mathit{pc}.n) = \mathsf{Exit}$

To contradict (ASSM-NO-SRC-STEP), we have the following subgoals using rule Exit:

– $(\mathit{fid}, n) = \mathit{pc}$, and
– $\texttt{commands}(\mathit{Fd}(\mathit{fid}))(n) = \mathsf{Exit}$
Proved above.

- By having considered all the possible cases for $\langle \mathcal{M}_c, \mathcal{M}_d, \mathit{stk}, \mathit{imp}, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle \to \langle \mathcal{M}_c, \mathcal{M}_d', \mathit{stk}', \mathit{imp}, \phi, \mathsf{ddc}', \mathsf{stc}', \mathsf{pcc}', \mathsf{nalloc}'\rangle$, and having derived a contradiction to (ASSM-NO-SRC-STEP) for each case, we proved the first subgoal:
(SUBGOAL-SRC-STEP-PROVED):
$\Sigma; \Delta; \beta; \mathit{MVar}; \mathit{Fd} \vdash \langle \mathit{Mem}, \mathit{stk}, \mathit{pc}, \Phi, \mathit{nalloc}\rangle \to \langle \mathit{Mem}', \mathit{stk}', \mathit{pc}', \Phi', \mathit{nalloc}'\rangle$

- Now we are required to prove: $K_{mod}; K_{fun}; \Sigma; \Delta; \beta; \mathit{MVar}; \mathit{Fd}; \langle \mathit{Mem}', \mathit{stk}', \mathit{pc}', \Phi', \mathit{nalloc}'\rangle \cong_{\mathit{modIDs}} \langle \mathcal{M}_c, \mathcal{M}_d', \mathit{stk}', \mathit{imp}, \phi, \mathsf{ddc}', \mathsf{stc}', \mathsf{pcc}', \mathsf{nalloc}'\rangle$

- For this, we apply Lemma 97 obtaining the following subgoals:

1. $[\![\overline{\mathit{mods}_1}]\!]_{\Delta,\Sigma,\beta,K_{mod},K_{fun}} = \langle \mathcal{M}_{c1}, \mathcal{M}_{d1}, \mathit{imp}_1, \mathsf{mstc}_1, \phi_1\rangle$
Immediate by the corresponding assumption of our lemma.
2. $K_{mod}; K_{fun}; \overline{\mathit{mods}_2} \ltimes \overline{\mathit{mods}_1}; \Sigma; \Delta; \beta; \mathit{MVar}; \mathit{Fd} \vdash_{exec} \langle \mathit{Mem}, \mathit{stk}, \mathit{pc}, \Phi, \mathit{nalloc}\rangle$
Immediate by the corresponding assumption of our lemma.
3. $t = \langle \mathcal{M}_{c2}, \mathcal{M}_{d2}, \mathit{imp}_2, \mathsf{mstc}_2, \phi_2\rangle \ltimes \langle \mathcal{M}_{c1}, \mathcal{M}_{d1}, \mathit{imp}_1, \mathsf{mstc}_1, \phi_1\rangle$
Immediate by the corresponding assumption of our lemma.
4. $t \vdash_{exec} \langle \mathcal{M}_c, \mathcal{M}_d, \mathit{stk}, \mathit{imp}, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle$
Immediate by the corresponding assumption of our lemma.
5. $\mathit{modIDs} = \{\mathit{modID} \mid (\mathit{modID}, \_, \_) \in \overline{\mathit{mods}_1}\}$
Immediate by the corresponding assumption of our lemma.

6. $\texttt{moduleID}(Fd(pc.fid)) \in modIDs$
   Immediate by the corresponding assumption of our lemma.
7. $K_{mod}; K_{fun}; \Sigma; \Delta; \beta; MVar; Fd; \langle Mem, stk, pc, \Phi, nalloc\rangle \cong_{modIDs}$
   $\langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle$
   Immediate by the corresponding assumption of our lemma.
8. $\Sigma; \Delta; \beta; MVar; Fd \vdash \langle Mem, stk, pc, \Phi, nalloc\rangle \to \langle Mem', stk', pc', \Phi', nalloc'\rangle$
   Immediate by the previously proven subgoal (SUBGOAL-SRC-STEP-PROVED).
9. $\texttt{moduleID}(Fd(pc'.fid)) \in modIDs$
   Here, we prove it by case analysis on (SUBGOAL-SRC-STEP-PROVED):

   Case Assign-to-var-or-arr:

   Case Allocate:

   Case Jump-non-zero:

   Case Jump-zero:

   Case Exit:

   In these five cases, we observe that $pc'.fid = pc.fid$.
   Thus, our goal (by substitution) becomes:
   $\texttt{moduleID}(Fd(pc.fid)) \in modIDs$
   But this is immediate by assumption.

   Case Call:

   Here, we obtain the following preconditions:
   $\texttt{commands}(Fd(fid))(n) = \mathsf{Call}\ fid_{call}\ \overline{e}$, and
   $pc' = (fid_{call}, 0)$
   By (CURR-COM-COMPILED), and the first precondition obtained above, we know:
   $\mathcal{M}_c(\mathsf{pcc}) = \mathsf{Cinvoke}\ \texttt{moduleID}(Fd(fid_{call}))\ fid_{call}\ \_$
   From assumption (TRG-STEPS), and by inversion using rules cinvoke then cinvoke-aux, we know:
   (PCC'-BOUNDS):
   $\mathsf{pcc}' \doteq imp(\texttt{moduleID}(Fd(fid_{call}))).1$
   Our goal (by substitution from the second precondition) becomes:
   $\texttt{moduleID}(Fd(fid_{call})) \in modIDs$
   which is immediate by assumptions.

   Case Return:

   Here, we deduce the following from the preconditions:
   $pc' = stk(\texttt{length}(stk) - 1)$
   Thus, our goal (by substitution) becomes:
   $\texttt{moduleID}(Fd(stk(\texttt{length}(stk) - 1)).fid) \in modIDs$
   By unfolding our lemma assumption using Definition 64 then Definition 62, we know that it suffices for our goal to prove:
   $\exists mid \in modIDs.\ K_{mod}(mid) = [stk(\texttt{length}(stk) - 1).\mathsf{pcc}.\sigma, stk(\texttt{length}(stk) - 1).\mathsf{pcc}.e)$

   – By inversion of our lemma assumption using rule creturn, we know (PCC'-IS-STK-TOP-ASSM):
     $stk(\texttt{length}(stk) - 1).\mathsf{pcc} = \mathsf{pcc}'$, and
     (PCC'-IS-SOME-MODULE-CODE):
     $\exists mid'.\ imp(mid').\mathsf{pcc} = \mathsf{pcc}'$
   – We obtain mid' from (PCC'-IS-SOME-MODULE-CODE).
   – But then by Lemmas 91 and 92, and valid-linking, we know:
     $mid' \in modIDs \ \wedge\ imp(mid').\mathsf{pcc} = (\kappa, K_{mod}(mid').1, K_{mod}(mid').2, 0)$
   – By simple rewriting, we know:
     $mid' \in modIDs \ \wedge\ K_{mod}(mid') = [imp(mid').\mathsf{pcc}.\sigma, imp(mid').\mathsf{pcc}.e)$

– Now by substitution using (PCC'-IS-SOME-MODULE-CODE) then (PCC'-IS-STK-TOP-ASSM), we obtain:
$mid' \in modIDs \ \wedge$
$K_{mod}(mid') = [stk(\texttt{length}(stk) - 1).\mathsf{pcc}.\sigma, stk(\texttt{length}(stk) - 1).\mathsf{pcc}.e)$

– This satisfies our goal by choosing mid'.

This concludes our case analysis on (SUBGOAL-SRC-STEP-PROVED) proving subgoal $\texttt{moduleID}(Fd(pc'.fid)) \in modIDs$.

- This concludes the proof of Lemma 98.

□

Lemma 99 (Compiler forward simulation, multiple steps)**.**

$\forall K_{mod}, K_{fun}, \Sigma; \Delta; \beta; MVar; Fd, \langle Mem, stk, pc, \Phi, nalloc\rangle, \overline{mods_1},$
$t, \langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle.$
$[\![mods_1]\!]_{\Delta,\Sigma,\beta,K_{mod},K_{fun}} = t \ \wedge$
$K_{mod}; K_{fun}; \overline{mods_1}; \Sigma; \Delta; \beta; MVar; Fd \vdash_{exec} \langle Mem, stk, pc, \Phi, nalloc\rangle \ \wedge$
$t \vdash_{exec} \langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle \ \wedge$
$modIDs = \{modID \mid (modID, \_, \_) \in \overline{mods_1}\} \ \wedge$
$K_{mod}; K_{fun}; \Sigma; \Delta; \beta; MVar; Fd; \langle Mem, stk, pc, \Phi, nalloc\rangle \cong_{modIDs} \langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle \ \wedge$
$\Sigma; \Delta; \beta; MVar; Fd \vdash \langle Mem, stk, pc, \Phi, nalloc\rangle \to^* \langle Mem', stk', pc', \Phi', nalloc'\rangle$
$\implies$
$\langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle \to^* \langle \mathcal{M}_c, \mathcal{M}_d', stk', imp, \phi, \mathsf{ddc}', \mathsf{stc}', \mathsf{pcc}', \mathsf{nalloc}'\rangle \ \wedge$
$K_{mod}; K_{fun}; \Sigma; \Delta; \beta; MVar; Fd; \langle Mem', stk', pc', \Phi', nalloc'\rangle \cong_{modIDs} \langle \mathcal{M}_c, \mathcal{M}_d', stk', imp, \phi, \mathsf{ddc}', \mathsf{stc}', \mathsf{pcc}', \mathsf{nalloc}'\rangle$

Proof.
We assume the antecedents, and we prove it by induction on the relation $\to^*$.

- **Base case (reflexivity):**

  Here, our goal is immediate by the lemma assumptions.

- **Inductive case (transitivity):**

  Here, we obtain s_s'' such that (ASSM1):
  $\Sigma; \Delta; \beta; MVar; Fd \vdash \langle Mem, stk, pc, \Phi, nalloc\rangle \to^* s_s''$, and
  $\Sigma; \Delta; \beta; MVar; Fd; s_s'' \to \langle Mem', stk', pc', \Phi', nalloc'\rangle$.

  And by the inductive hypothesis, we have s_t'' such that (ASSM2):
  $\langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle \to^* s_t''$,
  $\Sigma; \Delta; \beta; MVar; Fd; s_s'' \cong_{modIDs} s_t''$

  By induction on the relation $\to^*$ in (ASSM2) and by using Lemma 52, we know (*):
  $t \vdash_{exec} s_t''$

  By induction on the relation $\to^*$ in (ASSM1) and by using Lemma 56, we know (**):
  $K_{mod}; K_{fun}; \overline{mods_1} \ltimes \overline{mods_2}; \Sigma; \Delta; \beta; MVar; Fd \vdash_{exec} s_s''$

  Our goal is:

  $\exists \mathcal{M}_d', stk', \mathsf{ddc}', \mathsf{stc}', \mathsf{pcc}', \mathsf{nalloc}'. \ \ s_t'' \to \langle \mathcal{M}_c, \mathcal{M}_d', stk', imp, \phi, \mathsf{ddc}', \mathsf{stc}', \mathsf{pcc}', \mathsf{nalloc}'\rangle \ \wedge$
  $K_{mod}; K_{fun}; \Sigma; \Delta; \beta; MVar; Fd; \langle Mem', stk', pc', \Phi', nalloc'\rangle \cong_{modIDs} \langle \mathcal{M}_c, \mathcal{M}_d', stk', imp, \phi, \mathsf{ddc}', \mathsf{stc}', \mathsf{pcc}', \mathsf{nalloc}'\rangle$

  We apply Lemma 97 obtaining the following subgoals:

- $\Sigma; \Delta; \beta; MVar; Fd; s''_s \to \langle Mem', stk', pc', \Phi', nalloc' \rangle$.
  Immediate by (ASSM1).
- $\Sigma; \Delta; \beta; MVar; Fd; s''_s \cong_{modIDs} s''_t$
  Immediate by (ASSM2).
- $t \vdash_{exec} s''_t$
  Immediate by (*).
- $K_{mod}; K_{fun}; \overline{mods_1} \ltimes \overline{mods_2}; \Sigma; \Delta; \beta; MVar; Fd \vdash_{exec} s''_s$
  Immediate by (**).
- The remaining subgoals are immediate by the antecedents of the current lemma.

This concludes the proof of Lemma 99. □

Theorem 1 (Compiler backward simulation, multiple steps (Compiler correctness))**.**

$\forall K_{mod}, K_{fun}, \Sigma; \Delta; \beta; MVar; Fd, \langle Mem, stk, pc, \Phi, nalloc \rangle, \overline{mods_1},$
$t, \langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc} \rangle.$
$\llbracket mods_1 \rrbracket_{\Delta,\Sigma,\beta,K_{mod},K_{fun}} = t \ \wedge$
$K_{mod}; K_{fun}; \overline{mods_1}; \Sigma; \Delta; \beta; MVar; Fd \vdash_{exec} \langle Mem, stk, pc, \Phi, nalloc \rangle \ \wedge$
$t \vdash_{exec} \langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc} \rangle \ \wedge$
$modIDs = \{modID \mid (modID, \_, \_) \in \overline{mods_1}\} \ \wedge$
$K_{mod}; K_{fun}; \Sigma; \Delta; \beta; MVar; Fd; \langle Mem, stk, pc, \Phi, nalloc \rangle \cong_{modIDs} \langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc} \rangle \ \wedge$
$\langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc} \rangle \to^* \langle \mathcal{M}_c, \mathcal{M}'_d, stk', imp, \phi, \mathsf{ddc}', \mathsf{stc}', \mathsf{pcc}', \mathsf{nalloc}' \rangle$
$\implies$
$\Sigma; \Delta; \beta; MVar; Fd \vdash \langle Mem, stk, pc, \Phi, nalloc \rangle \to^* \langle Mem', stk', pc', \Phi', nalloc' \rangle \ \wedge$
$K_{mod}; K_{fun}; \Sigma; \Delta; \beta; MVar; Fd; \langle Mem', stk', pc', \Phi', nalloc' \rangle \cong_{modIDs} \langle \mathcal{M}_c, \mathcal{M}'_d, stk', imp, \phi, \mathsf{ddc}', \mathsf{stc}', \mathsf{pcc}', \mathsf{nalloc}' \rangle$

Proof. Similar to the proof of Lemma 99. Follows from Lemma 98, Lemma 52, and Lemma 56. □

Lemma 100 (Source and compiled initial states are cross-language related)**.**

$$\begin{aligned}
&\forall \omega \in \mathbb{N}, \overline{m_1}, \overline{m}, s_i, \Delta, \Sigma, \beta, K_{mod}, K_{fun}, MVar, Fd, modIDs, t, t' s_i.\\
&modIDs = \{modID \mid (modID, _, _) \in \overline{m}\} \ \wedge\\
&K_{mod}; K_{fun}; \overline{m}; \Sigma; \Delta + \omega; \beta; MVar; Fd \vdash_i s_i \ \wedge\\
&t' = \llbracket \overline{m} \rrbracket_{\Delta,\Sigma,\beta,K_{mod},K_{fun}} \ \wedge\\
&t = t' + \omega \ \wedge\\
&t \vdash_i s_i \ \wedge\\
&\implies\\
&K_{mod}; K_{fun}; \Sigma; \Delta + \omega; \beta; MVar; Fd; s_i \cong_{modIDs} s_i
\end{aligned}$$

Proof.
By inverting assumption $t \vdash_i s_i$ using rule initial-state, and by instantiating Lemma 5 then inversion using rule exec-state together with assumption $s_i.\mathsf{pcc} \subseteq \mathsf{dom}(t'.\mathcal{M}_c)$, we know (ASSM1):

$\exists mainMod.\ (t'.imp + \omega)(mainMod) = (p, d, \mathit{offs}) \ \wedge \ \mathtt{main} \in \mathsf{dom}(\mathit{offs}) \ \wedge$
$s_i.\mathsf{pcc} = (\kappa, p.\sigma, p.e, \mathit{offs}(\mathtt{main})) \ \wedge \ s_i.\mathsf{ddc} = d \ \wedge \ s_i.\mathsf{stc} = s_i.\mathsf{mstc}(mainMod) \ \wedge \ t'.\phi(mainMod, \mathtt{main}) = (nArgs, nLocal)$
$s_i.\mathsf{stc} = (\delta, \_, \_, nArgs + nLocal)$

Also, by inverting assumption $K_{mod}; K_{fun}; \overline{m}; \Sigma; \Delta + \omega; \beta; MVar; Fd \vdash_i s_i$ using rule Initial-state-src, we know (ASSM2):

$$s_i.pc = (\mathtt{main}, 0) \ \wedge$$
$$s_i.\Phi = \{\mathtt{moduleID}(Fd(\mathtt{main})) \mapsto \mathtt{frameSize}(Fd(\mathtt{main})\} \cup \bigcup_{mid \in \mathsf{dom}(\Delta) \setminus \{\mathtt{moduleID}(Fd(\mathtt{main}))\}} \{mid \mapsto 0\}$$

Furthermore, by Lemma 91, and by inversion of the assumption $K_{mod}; K_{fun}; \overline{m}; \Sigma; \Delta + \omega; \beta; MVar; Fd \vdash_i s_i$ using rules Initial-state-src then Well-formed program and parameters then Well-formed program, we know $mainMod$ of (*) is unique.

Our goal (by unfolding Definition 64) consists of the following subgoals:

- $s_i.nalloc = s_i.\mathsf{nalloc}$

  From the assumptions and by inverting rules initial-state and Initial-state-src, we know $s_i.nalloc = s_i.\mathsf{nalloc} = -1$.

- $A_s = \mathsf{reachable\_addresses}(\Sigma, \Delta + \omega, modIDs, s_i.Mem) \ \wedge$
  $A_t = \mathsf{reachable\_addresses}(\bigcup_{mid \in modIDs} \{(t'.imp + \omega)(mid).\mathsf{ddc}, t'.\mathsf{mstc}(mid)\}, t'.\mathcal{M}_d + \omega) \ \wedge$
  $A_s = A_t \ \wedge \ s_i.Mem|_{A_s} = (t'.\mathcal{M}_d + \omega)|_{A_t}$

  From the assumptions, and by inverting rules initial-state, and Initial-state-src, we get the following substitutions:

  $t'.\mathcal{M}_d + \omega = \{a \mapsto 0 \mid a \in \mathsf{dom}(t'.\mathcal{M}_d + \omega)\}$, and

  $s_i.Mem = \{a \mapsto 0 \mid a \in \bigcup_{mid \in modIDs} \Delta(mid)\}$

  Thus, by Lemma 10 and Lemma 61, we observe that (*):
  $A_s = \mathsf{static\_addresses}(\Sigma, \Delta + \omega, modIDs)$, and

  $A_t = \{a \mid a \in [c.\sigma, c.e) \ \wedge \ c \in \bigcup_{mid \in modIDs} \{(t'.imp + \omega)(mid).\mathsf{ddc}, t'.\mathsf{mstc}(mid)\}\}$

  By Definition 46, we thus know (**):
  $A_s = \{a \mid a \in (\Delta + \omega)(mid) \wedge mid \in modIDs\} \uplus \{a \mid a \in \Sigma(mid) \wedge mid \in modIDs\}$

  The first conjunct of our goal is $A_s = A_t$.

  Substituting using (*) and (**), it suffices to show that:
  $\forall mid \in modIDs.\ (\Delta + \omega)(mid) = [(t'.imp + \omega)(mid).\mathsf{ddc}.\sigma, (t'.imp + \omega)(mid).\mathsf{ddc}.e) \wedge \Sigma(mid) = [t'.\mathsf{mstc}(mid).\sigma, t'.\mathsf{mstc}(mid).e)$

  By applying Definitions 15 and 44, and using simple arithmetic, it suffices to show that:
  $\forall mid \in modIDs.\ \Delta(mid) = [t'.imp(mid).\mathsf{ddc}.\sigma, t'.imp(mid).\mathsf{ddc}.e) \wedge \Sigma(mid) = [t'.\mathsf{mstc}(mid).\sigma, t'.\mathsf{mstc}(mid).e)$

  This follows immediately by Lemma 91.

- $(\Delta + \omega)(\mathtt{moduleID}(Fd(s_i.pc.fid))) = (s_i.\mathsf{ddc}.\sigma, s_i.\mathsf{ddc}.e)$

  By (ASSM1) and (ASSM2), it suffices to show that:

  $(\Delta + \omega)(\mathtt{main}) = [(t'.imp + \omega)(mainMod).\mathsf{ddc}.\sigma, (t'.imp + \omega)(mainMod).\mathsf{ddc}.e)$

  Again, by applying Definitions 15 and 44, and using simple arithmetic, it suffices to show that:
  $\Delta(\mathtt{main}) = [t'.imp(mainMod).\mathsf{ddc}.\sigma, t'.imp(mainMod).\mathsf{ddc}.e)$

  By the uniqueness of $mainMod$ argued above, this goal is immediate by Lemma 91.

- $\Sigma(\mathtt{moduleID}(Fd(s_i.pc.fid))) = (s_i.\mathsf{stc}.\sigma, s_i.\mathsf{stc}.e)$

  By (ASSM1) and (ASSM2), and by rule initial-state giving $t'\mathsf{mstc} \doteq s_i.\mathsf{mstc}$, it suffices to show that:

$\Sigma(\texttt{main}) = [t'.\mathsf{mstc}(mainMod).\sigma, t'.\mathsf{mstc}(mainMod).e)$

By the uniqueness of $mainMod$ argued above, this goal is immediate by Lemma 91.

- $\Phi(\texttt{moduleID}(Fd(s_i.pc.fid))) = s_i.\mathsf{stc}.off$

  By (ASSM1) and (ASSM2), it suffices to show that:
  $\texttt{frameSize}(Fd(\texttt{main})) = t'.\phi(mainMod, \texttt{main}).nArgs + t'.\phi(mainMod, \texttt{main}).nLocal$

  By the definition of $\texttt{frameSize}$, it is equivalent to show that:
  $\texttt{length}(\texttt{args}(Fd(\texttt{main}))) + \texttt{length}(\texttt{localIDs}(Fd(\texttt{main}))) =$
  $t'.\phi(mainMod, \texttt{main}).nArgs + t'.\phi(mainMod, \texttt{main}).nLocal$

  By the uniqueness of $mainMod$ argued above, this goal is immediate by Lemma 91.

- $K_{mod}(\texttt{moduleID}(Fd(s_i.pc.fid))).1 + K_{fun}(s_i.pc.fid).1 + s_i.pc.n = s_i.\mathsf{pcc}.\sigma + s_i.\mathsf{pcc}.off \wedge$
  $K_{mod}(\texttt{moduleID}(Fd(s_i.pc.fid))) = [s_i.\mathsf{pcc}.\sigma, s_i.\mathsf{pcc}.e)]$

  By (ASSM1) and (ASSM2), it suffices to show that:
  $K_{mod}(\texttt{moduleID}(Fd(\texttt{main}))).1 + K_{fun}(\texttt{main}).1 + 0 =$
  $(t'.imp + \omega)(mainMod).\mathsf{pcc}.\sigma + (t'.imp + \omega)(mainMod).offs(\texttt{main}) \wedge$
  $K_{mod}(\texttt{moduleID}(Fd(\texttt{main}))) = [(t'.imp + \omega)(mainMod).\mathsf{pcc}.\sigma, (t'.imp + \omega)(mainMod).\mathsf{pcc}.e)]$

  By Definition 15, it is equivalent to show:

  $K_{mod}(\texttt{moduleID}(Fd(\texttt{main}))).1 + compilation - bounds - preserved K_{fun}(\texttt{main}).1 =$
  $t'.imp(mainMod).\mathsf{pcc}.\sigma + t'.imp(mainMod).offs(\texttt{main}) \wedge$
  $K_{mod}(\texttt{moduleID}(Fd(\texttt{main}))) = [t'.imp(mainMod).\mathsf{pcc}.\sigma, t'.imp(mainMod).\mathsf{pcc}.e)]$

  By the uniqueness of $mainMod$ argued above, this goal is immediate by Lemma 91.

- $K_{mod}; K_{fun}; \Sigma; \Delta + \omega; \beta; MVar; Fd; s_i.stk \cong_{modIDs} s_i.stk$

  Here, by unfolding Definition 62, and choosing $f = \emptyset$, we satisfy all the conjuncts of our goal because $s_i.stk = \texttt{nil}$ and $s_i.stk = \texttt{nil}$.

- $K_{mod}; K_{fun}; \Sigma; \Delta + \omega; \beta; MVar; Fd; s_i.\Phi \cong_{modIDs} s_i.\mathsf{mstc}, s_i.\phi$

  By unfolding Definition 63, it suffices to show:

  – $\forall mid \in modIDs.\ s_i.\Phi(mid) = s_i.\mathsf{mstc}(mid).off$

    Using the definition of $s_i.\Phi$ given by (ASSM2), we distinguish two cases:

    ∗ **Case** $mid = \texttt{main}$**:**
      In this case, our goal follows by (ASSM1), and the uniqueness of $mainMod$ argued above together with Lemma 91.

    ∗ **Case** $mid \neq \texttt{main}$**:**
      In this case, our goal is immediate by (ASSM1) and the precondition
      $\forall sc.\ sc \in \mathsf{range}(\mathsf{mstc}) \setminus \{\mathsf{stc}\} \implies sc = (\delta, \_, \_, 0)$ of rule initial-state which we get by inversion of our assumption $t \vdash_i s_i$.

  – $\forall fid \in \mathsf{dom}(Fd), mid.\ \texttt{moduleID}(Fd(fid)) = mid \implies$
    $(\texttt{frameSize}(Fd(fid)) + \Sigma(mid).1 + s_i.\Phi(mid) < \Sigma(mid).2 \iff$
    $s_i.\phi(mid, fid).1 + s_i.\phi(mid, fid).2 + s_i.\mathsf{mstc}(mid).\sigma + s_i.\mathsf{mstc}(mid).off < s_i.\mathsf{mstc}(mid).e)$

  – $\forall fid \in \mathsf{dom}(Fd), mid.\ \texttt{moduleID}(Fd(fid)) = mid \implies$
    $\texttt{length}(\texttt{args}(Fd(fid))) = s_i.\phi(mid, fid).1$

  – $\forall (mid, fid) \in \mathsf{dom}(s_i.\phi).\ fid \in \mathsf{dom}(Fd) \wedge mid = \texttt{moduleID}(Fd(fid))$

    All of these three subgoals are immediate after substitution using Lemma 91.

This concludes the proof of Lemma 100. □

Definition 65 (Target empty context)**.**

$$\varnothing \stackrel{\mathsf{def}}{=} (\{\}, \{\}, \{\}, \{\}, \{\})$$

Lemma 101 (Target empty context is universally linkable)**.**

$$\forall t : \mathit{TragetSetup}.\ \varnothing \ltimes t = \lfloor t \rfloor$$

Proof.
Immediate by Definition 65 and rule valid-linking. □

Definition 66 (Target whole-program convergence compatible with partial convergence)**.**

$$\omega, \nabla \vdash t \Downarrow \stackrel{\mathsf{def}}{=} \omega, \nabla \vdash \varnothing[t] \Downarrow$$

Definition 67 (Source empty context)**.**

$$\varnothing \stackrel{\mathsf{def}}{=} \mathtt{nil}$$

Lemma 102 (Source empty context is universally linkable and universally order-preserving)**.**

$$\forall p : \mathit{Prog}.\ \mathtt{wfp}(p) \implies \varnothing \ltimes p = \lfloor p \rfloor$$

$$\forall p, K_{mod}.\ \varnothing \rhd_{K_{mod}} p$$
$$\forall p, \Delta.\ p \rhd_{\Delta} \varnothing$$

Proof.
Immediate by Definition 67 and (rule Valid-linking-src + definition 41). □

Definition 68 (Source whole-program convergence compatible with partial convergence)**.**

$$K_{mod}, K_{fun}, \Sigma, \Delta + \omega, \beta, \nabla \vdash \overline{m} \Downarrow \stackrel{\mathsf{def}}{=} K_{mod}, K_{fun}, \Sigma, \Delta + \omega, \beta, \nabla \vdash \varnothing[\overline{m}] \Downarrow$$

Lemma 103 (Cross-language relatedness implies equi-terminality)**.**

$$\begin{array}{l}
\forall K_{mod}, K_{fun}, \Sigma; \Delta; \beta; \mathit{MVar}; \mathit{Fd}, s_s, \overline{mods_1}, \overline{mods_2}, t_1, t_2, t, s_t. \\
[\![mods_1]\!]_{\Delta,\Sigma,\beta,K_{mod},K_{fun}} = t_1\ \wedge \\
K_{mod}; K_{fun}; \overline{mods_1} \ltimes \overline{mods_2}; \Sigma; \Delta; \beta; \mathit{MVar}; \mathit{Fd} \vdash_{exec} s_s\ \wedge \\
t = t_1 \ltimes t_2\ \wedge \\
t \vdash_{exec} s_t\ \wedge \\
\mathit{modIDs} = \{\mathit{modID} \mid (\mathit{modID}, _, _) \in \overline{mods_1}\}\ \wedge \\
\mathtt{moduleID}(\mathit{Fd}(s_s.pc.fid)) \in \mathit{modIDs}\ \wedge \\
K_{mod}; K_{fun}; \Sigma; \Delta; \beta; \mathit{MVar}; \mathit{Fd}; s_s \cong_{\mathit{modIDs}} s_t \\
\quad \implies \\
\vdash_t s_s \iff \vdash_t s_t
\end{array}$$

Proof.
We assume the antecedents.

- **" $\implies$ " direction:**

  We assume $\vdash_t s_s$, and our goal by unfolding Definition 13 is to show that $\mathcal{M}_c(s_t.\mathsf{pcc}) = \mathsf{Exit}$.

  Here, it suffices by assumption $t = t_1 \ltimes t_2$ and rule valid-linking to show that:
  $t_1.\mathcal{M}_c(s_t.\mathsf{pcc}) = \mathsf{Exit}$

  assuming that:
  $s_t.\mathsf{pcc} \in \mathsf{dom}(t_1.\mathcal{M}_c)$

  The latter follows from the assumptions:
  $\mathtt{moduleID}(Fd(pc.fid)) \in modIDs$, $K_{mod}; K_{fun}; \overline{mods_1} \ltimes \overline{mods_2}; \Sigma; \Delta; \beta; MVar; Fd \vdash_{exec} s_s$, and $K_{mod}; K_{fun}; \Sigma; \Delta; \beta; MVar; Fd; s_s \cong_{modIDs} s_t$ after unfolding Definitions 61 and 64.

  For the former goal ($t_1.\mathcal{M}_c(s_t.\mathsf{pcc}) = \mathsf{Exit}$), we apply Lemma 93, to instead get the following three subgoals:

  – $\exists \overline{mods}, \Delta, \Sigma, \beta, K_{mod}, K_{fun}.\ \ [\![\overline{mods}]\!]_{\Delta,\Sigma,\beta,K_{mod},K_{fun}} = \langle t_1.\mathcal{M}_c, \_, \_, \_\rangle$

    We choose $\overline{mods} = \overline{mods}_1$, and $\Delta, \Sigma, \beta, K_{mod}, K_{fun}$ from our assumptions.

  – $\exists mid, fid, n.\ s_t.\mathsf{pcc}.\sigma + s_t.\mathsf{pcc}.off = K_{mod}(mid).1 + K_{fun}(fid).1 + n$

    which follows immediately by choosing $fid = s_s.pc.fid, n = s_s.pc.n, mid = \mathtt{moduleID}(s_s.pc.fid)$ from assumption $K_{mod}; K_{fun}; \Sigma; \Delta; \beta; MVar; Fd; s_s \cong_{modIDs} s_t$ after unfolding Definitions 61 and 64.

  – $(\!|\mathtt{commands}(Fd(s_s.pc.fid))(s_s.pc.n)|\!)_{\_} = \mathsf{Exit}$

    which is immediate by Definition 59 and by inverting assumption $\vdash_t s_s$ using Terminal-state-src-exit.

  This concludes the " $\implies$ " direction.

- **" $\impliedby$ " direction:**

  Here, we assume $\vdash_t s_t$, and our goal is to show $\vdash_t s_s$.

  (Similarly to the " $\implies$ " direction, here we know $s_t.\mathsf{pcc} \in \mathsf{dom}(t_1.\mathcal{M}_c)$, and we know we have all the assumptions of Lemma 93.)

  By inversion using rule Terminal-state-src-exit, our goal is to show that:
  $\mathtt{commands}(Fd(s_s.pc.fid))(s_s.pc.n) = \mathsf{Exit}$

  We assume for the sake of contradiction that (*):
  $\mathtt{commands}(Fd(s_s.pc.fid))(s_s.pc.n) \neq \mathsf{Exit}$

  By Lemma 93 though, we know:
  $(\!|\mathtt{commands}(Fd(s_s.pc.fid))(s_s.pc.n)|\!)_{Fd,K_{fun},s_s.pc.fid,\mathtt{moduleID}(Fd(s_s.pc.fid)),\beta} =$
  $t_1.\mathcal{M}_c(K_{mod}(\mathtt{moduleID}(Fd(s_s.pc.fid))).1 + K_{fun}(s_s.pc.fid).1 + n)$

  Equivalently, by assumptions
  $\mathtt{moduleID}(Fd(pc.fid)) \in modIDs$,
  $K_{mod}; K_{fun}; \overline{mods_1} \ltimes \overline{mods_2}; \Sigma; \Delta; \beta; MVar; Fd \vdash_{exec} s_s$,
  and $K_{mod}; K_{fun}; \Sigma; \Delta; \beta; MVar; Fd; s_s \cong_{modIDs} s_t$ after unfolding Definitions 61 and 64, we thus know:
  $(\!|\mathtt{commands}(Fd(s_s.pc.fid))(s_s.pc.n)|\!)_{Fd,K_{fun},s_s.pc.fid,\mathtt{moduleID}(Fd(s_s.pc.fid)),\beta} = t_1.\mathcal{M}_c(\mathsf{pcc})$

  Equivalently, by assumption $\vdash_t s_t$ after unfolding Definition 13, we thus know:
  $(\!|\mathtt{commands}(Fd(s_s.pc.fid))(s_s.pc.n)|\!)_{Fd,K_{fun},s_s.pc.fid,\mathtt{moduleID}(Fd(s_s.pc.fid)),\beta} = \mathsf{Exit}$

  Thus, by inversion using Definition 59, we know:
  $\mathtt{commands}(Fd(s_s.pc.fid))(s_s.pc.n) = \mathsf{Exit}$

  This contradicts assumption (*), so our goal is proved.

This concludes the proof of Lemma 103.

$\square$

3.2 Compositionality: linking-and-convergence-preserving homomorphism

Lemma 104 (Existence of an initial state is preserved and reflected by $[\![\cdot]\!]$).

$$
\begin{aligned}
&\forall \omega \in \mathbb{N}, \nabla < -1, \Delta, \Sigma, \beta, K_{mod}, K_{fun}, \overline{m}, t, t'.\\
&\texttt{wfp\_params}(\overline{m}, \Delta, \Sigma, \beta, K_{mod}, K_{fun}) \wedge\\
&t' = [\![\overline{m}]\!]_{\Delta,\Sigma,\beta,K_{mod},K_{fun}} \wedge\\
&t = (t'.\mathcal{M}_c, t'.\mathcal{M}_d + \omega, t'.imp + \omega, t'.\mathsf{mstc}, t'.\phi)\\
&\quad \Longrightarrow\\
&(\exists s_i, MVar, Fd.\ K_{mod}; K_{fun}; \overline{m}; \Sigma; \Delta + \omega; \beta; MVar; Fd \vdash_i s_i\\
&\quad \Longleftrightarrow\\
&\exists s_i.\ t \vdash_i s_i)
\end{aligned}
$$

Proof.
We assume the antecedents.

- **" $\Longrightarrow$ " direction:**

  Here we have $s_i, MVar, Fd$ with $K_{mod}; K_{fun}; \overline{m}; \Sigma; \Delta + \omega; \beta; MVar; Fd \vdash_i s_i$.

  By inversion using rules Initial-state-src and Exec-state-src, we obtain the following assumptions:

  1. $s_i.pc = (\texttt{main}, 0)$
  2. $s_i.pc = (funID, \_) \ \wedge\ funID \in \mathsf{dom}(Fd)$
  3. $\texttt{wfp\_params}(\overline{m}, \Delta + \omega, \Sigma, \beta, K_{mod}, K_{fun})$

  And our goal is to show $\exists s_i.\ t \vdash_i s_i$.

  We claim $\exists mainMod.\ t.imp(mainMod) = (p, d, offs) \ \wedge\ \texttt{main} \in \mathsf{dom}(offs)$.

  This claim holds by assumptions 1 and 2 together with Lemma 91.

  We pick:

$$
\begin{aligned}
s_i = \langle &t.\mathcal{M}_c, t.\mathcal{M}_d, \texttt{nil}, t.imp, t.\phi, t.imp(mainMod).ddc,\\
&t.\mathsf{mstc}(mainMod), t.imp(mainMod).pcc, t.\mathsf{mstc}, -1\rangle
\end{aligned}
$$

  Our goal using rules initial-state and exec-state consists of the following subgoals, all of which we prove below:

  – $s_i.\mathsf{pcc} = (\kappa, \_, \_, \_) \ \wedge\ \ s_i.\mathsf{ddc} = (\delta, \_, \_, \_) \ \wedge\ \ s_i.\mathsf{stc} = (\delta, \_, \_, \_)$:
  This is immediate by Lemmas 91 and 92 which describe the range of $t.imp$ (after unfolding Definition 15) and the range of $t.\mathsf{mstc}$.

  – $s_i.\mathsf{nalloc} < 0$:
  Immediate by $s_i.\mathsf{nalloc} = -1$.

  – $modIDs = \mathsf{dom}(s_i.imp) = \mathsf{dom}(s_i.\mathsf{mstc}) = \mathsf{dom}(t.\mathsf{mstc})$
  This is immediate by substitution then by Lemmas 91 and 92 which describe the domain of $t.imp$ (after unfolding Definition 15).

  – $\forall mid \in modIDs.\ s_i.\mathsf{mstc}(mid) \doteq t.\mathsf{mstc}(mid)$
  Immediate by substitution and the reflexivity of $\doteq$.

– $\forall sc \in \mathsf{range}(s_i.\mathsf{mstc}), c \in \mathsf{range}(s_i.imp).\ sc = (\delta, \_, \_, \_) \ \wedge \ sc \cap c.2 = \emptyset$:

The first conjunct is easy by Lemmas 91 and 92.

For the second conjunct, it is equivalent (after unfolding Definition 3, and unfolding the definition of s_i that we gave above) to show the following:

$$\bigcup_{sc \in \mathsf{range}(t'.\mathsf{mstc})} [sc.\sigma, sc.e) \cap \bigcup_{c \in \mathsf{range}(t'.imp+\omega)} [c.2.\sigma, c.2.e) \quad = \quad \emptyset$$

By Definition 15, it is equivalent to show that:

$$\bigcup_{sc \in \mathsf{range}(t'.\mathsf{mstc})} [sc.\sigma, sc.e) \cap \bigcup_{c \in \mathsf{range}(t'.imp)} [c.2.\sigma + \omega, c.2.e + \omega) \quad = \quad \emptyset$$

And by easy axioms, it is equivalent to show that:

$$\bigcup_{mid \in \mathsf{dom}(t'.\mathsf{mstc})} [t'.\mathsf{mstc}(mid).\sigma, t'.\mathsf{mstc}(mid).e) \cap$$
$$\bigcup_{mid \in \mathsf{dom}(t'.imp)} [t'.imp(mid).2.\sigma + \omega, t'.imp(mid).2.e + \omega) \quad = \quad \emptyset$$

By Lemmas 91 and 92 (together with our assumption about t'), and by folding Definition 44, it is equivalent to show that:

$$\bigcup_{mid \in \mathsf{dom}(t'.\mathsf{mstc})} \Sigma(mid) \ \cap \bigcup_{mid \in \mathsf{dom}(t'.imp)} (\Delta + \omega)(mid) \quad = \quad \emptyset$$

But by inverting assumption 3 using rule Well-formed program and parameters, we get the precondition (*):

$\bigcup(\Delta + \omega)(mid) \cap \bigcup \Sigma(mid) = \emptyset$

(*) immediately satisfies our goal by Lemma 92 which describes $\mathsf{dom}(t'.\mathsf{mstc})$ and $\mathsf{dom}(t'.imp)$.

– $\forall a, st.\ st \in \mathsf{range}(s_i.\mathsf{mstc}) \wedge a \in \mathsf{reachable\_addresses}(\{st\}, s_i.\mathcal{M}_d) \implies a \geq s_i.\mathsf{nalloc}$:

Here, assuming the antecedents, by Lemma 10, we know $a \in [st.\sigma, st.e)$.

And by Lemmas 91 and 92, we know $a \in \bigcup \Sigma(mid)$.

And by condition $(\bigcup \Delta(mid) \cup \bigcup \Sigma(mid)) \cap (-\infty, 0) = \emptyset$ which we get by inverting rule Module-list-translation then rule Well-formed program and parameters, we know $a \geq 0$.

Thus from $0 > s_i \mathsf{nalloc}$ which we proved above, we have our goal: $a \geq s_i.\mathsf{nalloc}$ by transitivity of $\geq$.

– $s_i.\mathsf{pcc} \subseteq \mathsf{dom}(s_i.\mathcal{M}_c)$:

This holds by assumptions 1 and 2 together with Lemma 93.

– $\forall a.\ s_i.\mathcal{M}_d(a) = (\kappa, \sigma, e, \_) \implies [\sigma, e) \subseteq \mathsf{dom}(s_i.\mathcal{M}_c)$

Vacuously true by noticing the definition of $s_i.\mathcal{M}_d$.

– $\exists mid \in modIDs.\ s_i.imp(mid) = (cc, dc, \_) \wedge s_i.\mathsf{pcc} \subseteq cc \wedge s_i.\mathsf{ddc} \doteq dc \wedge s_i.\mathsf{mstc}(mid) \doteq s_i.\mathsf{stc}$:

Pick $mid = mainMod$ from above. Then this is immediate by the definition of s_i and reflexivity of $\doteq$.

– $\forall (cc, dc, \_) \in \mathtt{range}(s_i.imp).\ \ (cc = (\kappa, \sigma, e, \_) \wedge [\sigma, e) \subseteq \mathsf{dom}(s_i.\mathcal{M}_c))$
$\wedge \ \ (dc = (\delta, \sigma, e, \_) \wedge [\sigma, e) \subseteq \mathsf{dom}(s_i.\mathcal{M}_d)) \wedge \forall a.\ a \in \mathsf{reachable\_addresses}(\{dc\}, \mathcal{M}_d) \implies a \geq s_i.\mathsf{nalloc}$

Fix arbitrary mid and $(cc, dc, \_)$ where $s_i.imp(mid) = (cc, dc, \_)$.

The first two conjuncts are immediate by Lemmas 91 to 93.

For the third conjunct, we fix arbitrary $a \in \mathsf{reachable\_addresses}(\{dc\}, \mathcal{M}_d)$

Then by Lemma 10, we know $a \in [dc.\sigma, dc.e)$.

And by Lemmas 91 and 92, we know $a \in \bigcup(\Delta + \omega)(mid)$.

And by condition $(\bigcup(\Delta + \omega)(mid) \cup \bigcup \Sigma(mid)) \cap (-\infty, 0) = \emptyset$ which we get by inverting assumption 3 using rule Well-formed program and parameters, we know $a \geq 0$.

Thus from $0 > s_i \mathsf{nalloc}$ which we proved above, we have our goal: $a \geq s_i.\mathsf{nalloc}$ by transitivity of $\geq$.

– $\forall \_ \in \mathtt{elems}(s_i.stk).\ \_$:

Vacuously true because $s_i.stk = \mathtt{nil}$.

This concludes the proof of the " $\implies$ " direction.

- **" $\impliedby$ " direction:**

  Here we have s_i with $t \vdash_i s_i$.

  By inversion using rules initial-state and exec-state, we obtain the following assumptions:

  1. $\exists mainMod.\ imp(mainMod) = (p, d, offs)\ \wedge\ \mathtt{main} \in \mathsf{dom}(offs)\ \wedge$
     $\mathsf{pcc} = (\kappa, p.\sigma, p.e, offs(\mathtt{main}))\ \wedge\ \mathsf{ddc} = d\ \wedge\ \mathsf{stc} = \mathsf{mstc}(mainMod)$
  2. $\forall sc \in \mathsf{range}(s_i.\mathsf{mstc}), c \in \mathsf{range}(s_i.imp).\ sc = (\delta, \_, \_, \_)\ \wedge\ sc \cap c.2 = \emptyset$

  And our goal is to show $\exists s_i, MVar, Fd.\ K_{mod}; K_{fun}; \overline{m}; \Sigma; \Delta + \omega; \beta; MVar; Fd \vdash_i s_i$.

  We pick $s_i = \langle \{a \mapsto 0 \mid a \in \bigcup_{mid \in \mathsf{dom}(\Delta)} (\Delta + \omega)(mid) \cup \Sigma(mid)\}, \mathtt{nil}, (\mathtt{main}, 0), \Phi, -1\rangle$

  where $\Phi = \{\mathtt{moduleID}(Fd(\mathtt{main})) \mapsto \mathtt{frameSize}(Fd(\mathtt{main}))\} \cup \bigcup_{mid \in \mathsf{dom}(\Delta) \setminus \{\mathtt{moduleID}(Fd(\mathtt{main}))\}} \{mid \mapsto 0\}$

  Our goal by inversion of rules Initial-state-src and Exec-state-src consists of the following subgoals, which we prove next (The preconditions of Initial-state-src are immediate by the definition of s_i. The preconditions of Exec-state-src remain.):

  - $\mathtt{wfp\_params}(\overline{m}, \Delta + \omega, \Sigma, \beta, K_{mod}, K_{fun})$

    Using rule Well-formed program and parameters, we need to prove the following subgoals:

    * $\forall mid, mid' \in modIDs.\ mid \neq mid' \implies (\Delta + \omega)(mid) \cap (\Delta + \omega)(mid') = \emptyset$
      By unfolding the definition of $\cap$ on intervals obtaining the characterizing inequalities, and by unfolding Definition 44, it is easy to show that it is equivalent to show that:
      $\forall mid, mid' \in modIDs.\ mid \neq mid' \implies (\Delta)(mid) \cap (\Delta)(mid') = \emptyset$
      But the latter follows immediately from the assumption $t' = [\![\overline{m}]\!]_{\Delta,\Sigma,\beta,K_{mod},K_{fun}}$ after inversion using rule Module-list-translation then rule Well-formed program and parameters.
    * $\bigcup(\Delta + \omega)(mid) \cap \bigcup \Sigma(mid) = \emptyset$
      By Lemma 92 which describes $\mathsf{dom}(t'.\mathsf{mstc})$ and $\mathsf{dom}(t'.imp)$, together with the preconditions defining domains of Σ and Δ which we get from the assumptions by inversion using rule Module-list-translation then rule Well-formed program and parameters, it is equivalent to show that:
      $\bigcup_{mid \in \mathsf{dom}(t'.\mathsf{mstc})} \Sigma(mid)\ \cap \bigcup_{mid \in \mathsf{dom}(t'.imp)} (\Delta + \omega)(mid)\ =\ \emptyset$
      By Lemmas 91 and 92, and by unfolding Definition 44, it is equivalent to show that:
      $\bigcup_{mid \in \mathsf{dom}(t'.\mathsf{mstc})} [t'.\mathsf{mstc}(mid).\sigma, t'.\mathsf{mstc}(mid).e)\ \cap$
      $\bigcup_{mid \in \mathsf{dom}(t'.imp)} [t'.imp(mid).2.\sigma + \omega, t'.imp(mid).2.e + \omega)\ =\ \emptyset$
      And by easy axioms about the domain and range of a function, it is equivalent to show that:
      $\bigcup_{sc \in \mathsf{range}(t'.\mathsf{mstc})} [sc.\sigma, sc.e) \cap \bigcup_{c \in \mathsf{range}(t'.imp)} [c.2.\sigma + \omega, c.2.e + \omega)\ =\ \emptyset$
      By folding Definition 15, it is equivalent to show that:
      $\bigcup_{sc \in \mathsf{range}(t'.\mathsf{mstc})} [sc.\sigma, sc.e) \cap \bigcup_{c \in \mathsf{range}(t'.imp + \omega)} [c.2.\sigma, c.2.e)\ =\ \emptyset$
      But this is immediate by assumption 2.
    * $(\bigcup(\Delta + \omega)(mid) \cup \bigcup \Sigma(mid)) \cap (-\infty, 0) = \emptyset$
      By assumption, $\omega \geq 0$. Thus, this subgoal follows from the corresponding statement about Δ which can be obtained from the assumption $t' = [\![\overline{m}]\!]_{\Delta,\Sigma,\beta,K_{mod},K_{fun}}$ after inversion using rule Module-list-translation then rule Well-formed program and parameters.

* $\forall mid \in modIDs.\ \biguplus_{vid \in MVar(mid)} \beta(vid, \bot, mid) = [0, (\Delta + \omega)(mid).2 - (\Delta + \omega)(mid).1)$

  By unfolding Definition 44, it is equivalent to show that:
  $\forall mid \in modIDs.\ \biguplus_{vid \in MVar(mid)} \beta(vid, \bot, mid) = [0, \Delta(mid).2 + \omega - (\Delta(mid).1 + \omega))$

  By simple arithmetic, it is equivalent to show:
  $\forall mid \in modIDs.\ \biguplus_{vid \in MVar(mid)} \beta(vid, \bot, mid) = [0, \Delta(mid).2 - \Delta(mid).1)$

  The latter is immediate from the assumption $t' = [\![\overline{m}]\!]_{\Delta,\Sigma,\beta,K_{mod},K_{fun}}$ after inversion using rule Module-list-translation then rule Well-formed program and parameters.

* The remaining subgoals are immediate from the assumption $t' = [\![\overline{m}]\!]_{\Delta,\Sigma,\beta,K_{mod},K_{fun}}$ after inversion using rule Module-list-translation then rule Well-formed program and parameters.

– $modIDs = \{modID \mid (modID, \_, \_) \in \overline{m}\} \quad \wedge$
  $funDefs = \{modFunDef \mid modFunDef \in modFunDefs \wedge (\_, \_, modFunDefs) \in \overline{m}\} \quad \wedge$
  $Fd = \{funID \mapsto funDef \mid funDef \in funDefs \wedge funDef = (\_, funID, \_, \_, \_)\}$

  Nothing to prove.

– $\mathsf{dom}(K_{mod}) = \mathsf{dom}(MVar) = \mathsf{dom}(\Sigma) = \mathsf{dom}(\Delta + \omega) = modIDs$

  After unfolding Definition 44, this subgoal is immediate from $\texttt{wfp\_params}(\overline{m}, \Delta, \Sigma, \beta, K_{mod}, K_{fun})$ which we get from the assumption $t' = [\![\overline{m}]\!]_{\Delta,\Sigma,\beta,K_{mod},K_{fun}}$ after inversion using rule Module-list-translation.

– $MVar = \{modID \mapsto \overline{varIDs} \mid (modID, \overline{varIDs}, \_) \in \overline{m}\}$

  Nothing to prove.

– $s_i.pc = (funID, \_) \ \wedge\ funID \in \mathsf{dom}(Fd)$

  The first conjunct is immediate. The second conjunct follows from assumption 1 together with Lemma 92.

– $\forall (fid, \_) \in \texttt{elems}(s_i.stk).\ fid \in \mathsf{dom}(Fd)$

  Vacuously true because $s_i.stk = \texttt{nil}$ by construction.

– $\mathsf{static\_addresses}(\Sigma, \Delta + \omega, modIDs) \subseteq \mathsf{dom}(s_i.Mem)$

  By unfolding Definition 46, and by the choice of $s_i.Mem$, it is immediate that $\mathsf{static\_addresses}(\Sigma, \Delta + \omega, modIDs) = \mathsf{dom}(s_i.Mem)$.

– $\nabla < -1 \implies (s_i.nalloc > \nabla\ \wedge$
  $\forall a \in \mathsf{dom}(s_i.Mem).\ a > \nabla\ \wedge$
  $\forall a, s, e, v.\ v \in \mathsf{range}(s_i.Mem)\ \wedge\ v = (\delta, s, e, \_)\ \wedge\ a \in [s, e) \implies a > \nabla)$

  Conjunct $s_i.nalloc > \nabla$ is immediate by assumption $\nabla < -1$ and the choice $s_i.nalloc = -1$.

  Conjunct $\forall a \in \mathsf{dom}(s_i.Mem).\ a > \nabla$ is immediate by the previously proved subgoal $(\bigcup(\Delta + \omega)(mid) \cup \bigcup \Sigma(mid)) \cap (-\infty, 0) = \emptyset$ and the definition of $\mathsf{dom}(s_i.Mem)$.

  The last conjunct is vacuously true by noticing that $\mathsf{range}(s_i.Mem) = \{0\}$.

– $\forall mid \in modIDs.\ \Sigma(mid).1 + s_i.\Phi(mid) \leq \Sigma(mid).2$

  Immediate by the interval type after noticing the definition of $s_i.\Phi$ which ensures $s_i.\Phi(mid) = 0$.

– $\forall mid \in modIDs.\ s_i.\Phi(mid) =$
  $$\sum_{fid\ \in\ \{fid\ \mid\ \texttt{moduleID}(Fd(fid))\ =\ mid\}} \texttt{frameSize}(Fd(fid)) \times (\texttt{countIn}((fid, \_), s_i.stk) + (s_i.pc = (fid, \_)\ ?\ 1\ :\ 0))$$

  Here, first notice that the sub-term $\texttt{countIn}((fid, \_), s_i.stk)$ is always equal to 0 because $s_i.stk = \texttt{nil}$ and $\texttt{countIn}(\_, \texttt{nil}) = 0$.

  Next, we distinguish two cases for mid:

* **Case** $mid = \mathtt{moduleID}(Fd(\mathtt{main}))$**:**
  In this case, $s_i.\Phi(mid) = \mathtt{frameSize}(Fd(\mathtt{main}))$.
  The right-hand side evaluates also to a non-zero value that corresponds to:
  $\mathtt{frameSize}(Fd(\mathtt{main}))$
  due to the choice on the value of $s_i.pc$.
* **Case** $mid \neq \mathtt{moduleID}(Fd(\mathtt{main}))$**:**
  In this case, the sub-term $(s_i.pc = (fid, \_)\ ?\ 1\ :\ 0)$ is 0 for all the summation terms.
  Also, the $\mathtt{countIn}(\cdots)$ sub-term is 0 as explained above.
  Thus in this case, both sides of the equality evaluate to 0: one side because $s_i.\Phi(mid) = 0$, and the other as explained above.

- $stk = \mathtt{nil} \implies pc.fid = \mathtt{main}$
  Immediate by the choice of $s_i.pc$ made above.
- $s_i.stk \neq \mathtt{nil} \implies s_i.stk(0).fid = \mathtt{main}$
  Vacuously true because $s_i.stk = \mathtt{nil}$.
- $\forall mid, a, \sigma, e.\ s_i.Mem(a) = (\delta, \sigma, e, \_) \ \wedge\ [\sigma, e) \cap \Sigma(mid) \neq \emptyset \implies a \in \Sigma(mid)$
  Vacuously true by choice of $s_i.Mem$.
- $s_i.nalloc < 0$
  Immediate by the choice $s_i.nalloc = -1$ made above.

This concludes the proof of the " $\Longleftarrow$ " direction.

This concludes the proof of Lemma 104. $\square$

Lemma 105 (Convergence is preserved and reflected by $[\![\cdot]\!]$)**.**

$$\begin{aligned}
&\forall \omega \in \mathbb{N}, \nabla \in \mathbb{Z}^-, \Delta, \Sigma, \beta, K_{mod}, K_{fun}, \overline{m}, t'.\\
&t' = [\![\overline{m}]\!]_{\Delta,\Sigma,\beta,K_{mod},K_{fun}} \implies\\
&(K_{mod}, K_{fun}, \Sigma, \Delta + \omega, \beta, \nabla \vdash \overline{m} \Downarrow\\
&\iff\\
&\omega, \nabla \vdash t' \Downarrow)
\end{aligned}$$

Proof.

We assume $t' = [\![\overline{m}]\!]_{\Delta,\Sigma,\beta,K_{mod},K_{fun}}$.

- **We prove the " $\implies$ " direction.**
  Assume $K_{mod}, K_{fun}, \Sigma, \Delta + \omega, \beta, \nabla \vdash \overline{m} \Downarrow$.
  Thus, we have–by unfolding Definitions 43 and 68 and eliminating the tautologies resulting from Lemma 102 that:

$$\begin{aligned}
&\exists s_t.\ \mathsf{initial_state}(\overline{m}, \mathsf{main_module}(\overline{m})) \to^*_{\nabla} s_t \ \wedge\\
&\exists MVar, Fd.\ K_{mod}; K_{fun}; \overline{m}; \Sigma; \Delta + \omega; \beta; MVar; Fd \vdash_t s_t
\end{aligned} \tag{1}$$

  Our goal (by unfolding Definitions 17 and 66 and eliminating the tautologies resulting from Lemma 101) is:

$$\begin{aligned}
&\exists t.\ t = (t'.\mathcal{M}_c, t'.\mathcal{M}_d + \omega, t'.imp + \omega, t'.\mathsf{mstc}, t'.\phi) \wedge\\
&\exists s_i.\ t \vdash_i s_i \wedge\\
&\forall s_i.\ t \vdash_i s_i \implies \exists s_t.\ s_i \to^*_{\nabla} s_t \ \wedge\ \vdash_t s_t
\end{aligned}$$

– **Subgoal** $\exists t.\ t = (t'.\mathcal{M}_c, t'.\mathcal{M}_d + \omega, t'.imp + \omega, t'.\mathsf{mstc}, t'.\phi)$**:**
  By the totality of the operator $+\ \omega$ (Definitions 14 and 15), this subgoal is immediate.
– **Subgoal** $\exists s_i.\ t \vdash_i s_i$**:**
  This follows immediately from Lemma 104.
– **Subgoal** $\forall s_i.\ t \vdash_i s_i \implies \exists s_t.\ s_i \rightarrow^*_{\triangledown} s_t \ \wedge\ \vdash_t s_t$**:**
  Fix an arbitrary s_i and assume $t \vdash_i s_i$.
  From Proposition (1), we obtain $s_i, MVar, Fd$ with:
  $K_{mod}; K_{fun}; \overline{m}; \Sigma; \Delta + \omega; \beta; MVar; Fd \vdash_i s_i$.
  Thus, we can now conclude from Lemma 100 that (INIT-REL):
  $K_{mod}; K_{fun}; \Sigma; \Delta + \omega; \beta; MVar; Fd; s_i \cong_{modIDs} s_i$
  with $modIDs = \{modID \mid (modID, \_, \_) \in \overline{m}\}$
  Now, again from Proposition (1), we obtain s_t with (SOURCE-STEPS):
  $s_i \rightarrow^*_{\triangledown} s_t$.
  For the first conjunct of our goal ($s_i \rightarrow^*_{\triangledown} s_t$), we apply Lemma 99.
  The generated subgoals are provable by:
  * (INIT-REL),
  * (SOURCE-STEPS),
  * obtaining the necessary source $\vdash_{exec}$ statement through inversion of conjunct $\vdash_i$ of Proposition (1) using rule Initial-state-src,
  * obtaining the necessary target $\vdash_{exec}$ statement through inversion of already proved statement $t \vdash_i s_i$ using rule initial-state,
  * choosing $\overline{mods_1} = \overline{m}$,
  * choosing $\overline{mods_2} = \varnothing$ (Definition 67), and
  * inversion of $\vdash_{exec}$ (once before and once after using Lemma 56 to obtain the subgoals $\texttt{moduleID}(Fd(s_i.pc.fid)) \in modIDs$ and $\texttt{moduleID}(Fd(s_t.pc.fid)) \in modIDs$ respectively).

  For the second conjunct of our goal ($\vdash_t s_t$), we apply Lemma 103.
  The generated subgoals are provable by:
  * (for subgoal $K_{mod}; K_{fun}; \Sigma; \Delta + \omega; \beta; MVar; Fd; s_t \cong_{modIDs} s_t$) applying Lemma 99 which is possible as described above,
  * choosing $\overline{mods_1} = \overline{m}$,
  * choosing $\overline{mods_2} = \varnothing$ (Definition 67),
  * (for subgoal $\vdash_t s_t$) using Proposition (1),
  * (for subgoal $t \vdash_{exec} s_t$) applying Lemma 52, and
  * (for subgoal $\_ \vdash_{exec} s_t$) applying Lemma 56.

  Using Lemma 103, we conclude from $\vdash_t s_t$ of Proposition (1) that $\vdash_t s_t$ which satisfies our subgoal.
  This concludes the proof of conjunct $\forall s_i.\ t \vdash_i s_i \implies \exists s_t.\ s_i \rightarrow^*_{\triangledown} s_t \ \wedge\ \vdash_t s_t$.

This concludes all subgoals of the " $\implies$ " direction.

• **We prove the " $\impliedby$ " direction.**
  Assume $\omega, \triangledown \vdash [\![\overline{m}]\!]_{\Delta,\Sigma,\beta,K_{mod},K_{fun}} \Downarrow$.
  Thus, we have–by unfolding Definitions 17 and 66 and eliminating the tautologies resulting from Lemma 101–that:

$$\begin{aligned}
&\exists t.\ t = (t'.\mathcal{M}_c, t'.\mathcal{M}_d + \omega, t'.imp + \omega, t'.\mathsf{mstc}, t'.\phi) \ \wedge \\
&\exists s_i.\ t \vdash_i s_i \ \wedge \\
&\forall s_i.\ t \vdash_i s_i \implies \exists s_t.\ s_i \rightarrow^*_{\triangledown} s_t \ \wedge\ \vdash_t s_t
\end{aligned} \tag{2}$$

Our goal (by unfolding Definitions 43 and 68 and eliminating the tautologies resulting from Lemma 102) is:

$$\exists s_t.\ \mathsf{initial\_state}(\overline{m}, \mathsf{main\_module}(\overline{m})) \rightarrow^*_\nabla s_t \ \wedge$$
$$\exists MVar, Fd.\ K_{mod}; K_{fun}; \overline{m}; \Sigma; \Delta; \beta; MVar; Fd \vdash_t s_t$$

- **Subgoal** $\exists s_i, MVar, Fd.\ K_{mod}; K_{fun}; \overline{m}; \Sigma; \Delta + \omega; \beta; MVar; Fd \vdash_i s_i$**:**
  Here, we apply Lemma 104.
  The generated subgoals are proved using:
  * Proposition (2),
  * assumption $t' = [\![\overline{m}]\!]_{\Delta,\Sigma,\beta,K_{mod},K_{fun}}$, and
  * (for subgoal $\mathtt{wfp\_params}(\overline{m}, \Delta, \Sigma, \beta, K_{mod}, K_{fun})$) inversion of assumption $t' = [\![\overline{m}]\!]_{\Delta,\Sigma,\beta,K_{mod},K_{fun}}$ using rule Module-list-translation.
- **Subgoal** $\forall s_i, MVar, Fd.$
  $K_{mod}; K_{fun}; \overline{m}; \Sigma; \Delta + \omega; \beta; MVar; Fd \vdash_i s_i \implies$
  $\exists s_t.\ s_i \rightarrow^*_\nabla s_t \ \wedge K_{mod}; K_{fun}; \overline{m}; \Sigma; \Delta + \omega; \beta; MVar; Fd \vdash_t s_t$**:**
  We fix arbitrary $s_i, MVar, Fd$ and assume $K_{mod}; K_{fun}; \overline{m}; \Sigma; \Delta + \omega; \beta; MVar; Fd \vdash_i s_i$.
  From Proposition (2), we obtain s_i with $t \vdash_i s_i$.
  This enables us to use Lemma 100 to conclude that (INIT-RELATED):
  $K_{mod}; K_{fun}; \Sigma; \Delta + \omega; \beta; MVar; Fd; s_i \cong_{modIDs} s_i$
  Thus, instantiate Theorem 1 to obtain:
  $\exists s_t.\ s_i \rightarrow^*_\nabla s_t \ \wedge \ K_{mod}; K_{fun}; \Sigma; \Delta + \omega; \beta; MVar; Fd; s_i \cong_{modIDs} s_t$
  Now, the remaining conjunct follows from Lemma 103 as in the proof of the " $\implies$ " direction.

This concludes all the subgoals of the " $\impliedby$ " direction.

□

One key property of many (compositional) compilers is that they are compatible with source and target linking. In particular, our compiler is a linking-preserving homomorphism (Lemma 106).

Lemma 106 (Compilation preserves linkability and convergence, i.e., $[\![\cdot]\!]$ is a linking-preserving homomorphism and more)**.**

$$\omega, \nabla \vdash [\![\mathbb{C}]\!]_{\Delta,\Sigma,\beta,K_{mod},K_{fun}}[\ [\![\overline{m_1}]\!]_{\tilde{\Delta},\tilde{\Sigma},\beta_1,K_{mod1},K_{fun1}}\ ]\Downarrow \iff$$
$$\omega, \nabla \vdash [\![\mathbb{C}[\overline{m_1}]_{\Delta\uplus\tilde{\Delta},\Sigma\uplus\tilde{\Sigma}}]\!]_{\Delta\uplus\tilde{\Delta},\Sigma\uplus\tilde{\Sigma},\beta\uplus\beta_1,K_{mod}\uplus K_{mod1},K_{fun}\uplus K_{fun1}}\Downarrow$$

Proof.
We let

$$\mathbb{C} = [\![\mathbb{C}]\!]_{\Delta,\Sigma,\beta,K_{mod},K_{fun}}$$
$$t_1 = [\![\overline{m_1}]\!]_{\tilde{\Delta},\tilde{\Sigma},\beta_1,K_{mod1},K_{fun1}}$$
$$t'_c = [\![\mathbb{C}[\overline{m_1}]_{\Delta\uplus\tilde{\Delta},\Sigma\uplus\tilde{\Sigma}}]\!]_{\Delta\uplus\tilde{\Delta},\Sigma\uplus\tilde{\Sigma},\beta\uplus\beta_1,K_{mod}\uplus K_{mod1},K_{fun}\uplus K_{fun1}}$$

- We prove the " $\implies$ " direction.

  From the assumption and by unfolding Definition 17 of convergence, we have the following:

$$\begin{aligned}&\exists t'.\ \mathbb{C} \ltimes t_1 = \lfloor t' \rfloor \ \wedge \\ &\exists t.\ t = (t'.\mathcal{M}_c, t'.\mathcal{M}_d + \omega, t'.imp + \omega, t'.\mathsf{mstc}, t'.\phi) \ \wedge \\ &\exists s_i.\ t \vdash_i s_i \ \wedge \\ &\forall s_i.\ t \vdash_i s_i \implies \exists s_t.\ s_i \rightarrow^*_\nabla s_t \ \wedge \ \vdash_t s_t\end{aligned} \tag{3}$$

Our goal, by unfolding Definitions 17 and 66 and after substituting using Lemma 101 is thus:

$$\begin{aligned}
&\exists t_c.\ t_c = (t'_c.\mathcal{M}_c, t'_c.\mathcal{M}_d + \omega, t'_c.imp + \omega, t'_c.\mathsf{mstc}, t'_c.\phi) \wedge \\
&\exists s'_i.\ t_c \vdash_i s'_i \wedge \\
&\forall s'_i.\ t_c \vdash_i s'_i \implies \exists s'_t.\ s'_i \rightarrow^*_{\triangledown} s'_t \ \wedge\ \vdash_t s'_t
\end{aligned}$$

The first conjunct of our goal is always true (see Definition 14).

For the second conjunct, we pick $s'_i = s_i$ from Proposition (3), and we also pick $t_c = t$ from Proposition (3). This allows conjunct $t_c \vdash_i s_i$ and conjunct $\forall s'_i.\ t_c \vdash_i s'_i \implies \exists s'_t.\ s'_i \rightarrow^*_{\triangledown} s'_t \ \wedge\ \vdash_t s'_t$ of our goal to follow immediately from the corresponding ones of Proposition (3).

So, it remains to show that $t = (t'_c.\mathcal{M}_c, t'_c.\mathcal{M}_d + \omega, t'_c.imp, t'_c.\mathsf{mstc}, t'_c.\phi)$. Here are all the subgoals:

- **Subgoal** $(\mathbb{C} \ltimes t_1).\mathcal{M}_c = t'_c.\mathcal{M}_c$**:**
  From rule valid-linking, we know:

  $$(\mathbb{C} \ltimes t_1).\mathcal{M}_c = \mathbb{C}.\mathcal{M}_c \uplus t_1.\mathcal{M}_c$$

  By Lemma 93, and Definition 32 of source linking, we conclude our subgoal.
- **Subgoal** $(\mathbb{C} \ltimes t_1).\mathcal{M}_d + \omega = t'_c.\mathcal{M}_d + \omega$**:**
  After unfolding Definition 14, it suffices to show that:

  $$(\mathbb{C} \ltimes t_1).\mathcal{M}_d = t'_c.\mathcal{M}_d$$

  From rule valid-linking, we know:

  $$(\mathbb{C} \ltimes t_1).\mathcal{M}_d = \mathbb{C}.\mathcal{M}_d \uplus t_1.\mathcal{M}_d$$

  Our subgoal then follows from Definition 32 of source linking and rules Module-list-translation and Module-translation.
- **Subgoal** $(\mathbb{C} \ltimes t_1).imp + \omega = t'_c.imp + \omega$**:**
  After unfolding Definition 15, it suffices to show that:

  $$(\mathbb{C} \ltimes t_1).imp = t'_c.imp$$

  From rule valid-linking, we know:

  $$(\mathbb{C} \ltimes t_1).imp = \mathbb{C}.imp \uplus t_1.imp$$

  By Lemma 91, and Definition 32 of source linking, we conclude our subgoal.
- **Subgoal** $(\mathbb{C} \ltimes t_1).\mathsf{mstc} = t'_c.\mathsf{mstc}$**:**
  From rule valid-linking, we know:

  $$(\mathbb{C} \ltimes t_1).\mathsf{mstc} = \mathbb{C}.\mathsf{mstc} \uplus t_1.\mathsf{mstc}$$

  By Lemma 91, and Definition 32 of source linking, we conclude our subgoal.
- **Subgoal** $(\mathbb{C} \ltimes t_1).\phi = t'_c.\phi$**:**
  From rule valid-linking, we know:

  $$(\mathbb{C} \ltimes t_1).\phi = \mathbb{C}.\phi \uplus t_1.\phi$$

  By Lemma 91, and Definition 32 of source linking, we conclude our subgoal.

This concludes the proof of the " $\Longrightarrow$ " direction.

- We prove the " $\Longleftarrow$ " direction.

  From the assumption and by unfolding Definitions 17 and 66 of whole program convergence and partial convergence, we obtain:

$$\begin{aligned}
&\exists t_c.\ t_c = (t'_c.\mathcal{M}_c, t'_c.\mathcal{M}_d + \omega, t'_c.imp + \omega, t'_c.\mathsf{mstc}, t'_c.\phi) \wedge \\
&\exists s'_i.\ t_c \vdash_i s'_i \wedge \\
&\forall s'_i.\ t_c \vdash_i s'_i \implies \exists s'_t.\ s'_i \to^*_{\triangledown} s'_t \ \wedge \ \vdash_t s'_t
\end{aligned} \tag{4}$$

  Also, by unfolding Definition 42 of layout-ordered linking, we obtain:

$$\begin{aligned}
&\mathbb{C} \ltimes \overline{m_1} = \lfloor \overline{m} \rfloor \wedge \\
&\overline{m_1} \triangleright_{\Delta \uplus \tilde{\Delta}, \Sigma \uplus \tilde{\Sigma}} \mathbb{C}
\end{aligned} \tag{5}$$

  Our goal, after unfolding Definition 17, is:

$$\begin{aligned}
&\exists t'.\ \mathbb{C} \ltimes t_1 = \lfloor t' \rfloor \wedge \\
&\exists t.\ t = (t'.\mathcal{M}_c, t'.\mathcal{M}_d + \omega, t'.imp + \omega, t'.\mathsf{mstc}, t'.\phi) \wedge \\
&\exists s_i.\ t \vdash_i s_i \wedge \\
&\forall s_i.\ t \vdash_i s_i \implies \exists s_t.\ s_i \to^*_{\triangledown} s_t \ \wedge \ \vdash_t s_t
\end{aligned}$$

  To prove the first conjunct, we pick $t' = t'_c$, the latter we have from Proposition (4) and we hence verify that all the assumptions of rule valid-linking hold:

  - **Subgoal disjointness** ($t'_c = (\mathbb{C}.\mathcal{M}_c \uplus t_1.\mathcal{M}_c, \mathbb{C}.\mathcal{M}_d \uplus t_1.\mathcal{M}_d, \mathbb{C}.imp \uplus t_1.imp, \mathbb{C}.\mathsf{mstc} \uplus t_1.\mathsf{mstc}, \mathbb{C}.\phi \uplus t_1.\phi)$)**:**
    Here, we apply Lemmas 91 to 93 to both the left- and right-hand sides of our goal and thus, we are left with disjointness subgoals that are provable by inversion of rules Valid-linking-src and Well-formed program (both we get by first inverting rule Module-list-translation).
  - **Subgoal order condition** $\mathtt{min}(\mathsf{dom}(\mathbb{C}.\mathcal{M}_d)) > \mathtt{max}(\mathsf{dom}(t_1.\mathcal{M}_d))$**):**
    Follows from conjunct $\overline{m_1} \triangleright_{\Delta \uplus \tilde{\Delta}, \Sigma \uplus \tilde{\Sigma}} \mathbb{C}$ (Definition 41) of Proposition (5) after applying Lemma 92.
  - **Subgoal distinct function IDs** ($funIDs = [fid \mid fid \in \mathsf{dom}(offs) \wedge (\_, \_, offs) \in \mathsf{range}(\mathbb{C}.imp) \cup \mathsf{range}(t_1.imp)] \ \wedge \ \mathsf{all\_distinct}(funIDs)$)**:**
    Follows from the corresponding condition after inverting rule Well-formed program which we get by first applying Lemmas 91 and 92 and then inverting rule Module-list-translation then inverting the precondition $\mathtt{wfp\_params}(\overline{m}, \cdots)$ using rule Well-formed program and parameters.
  - **Subgoal disjointness of capabilities** $\forall c_1 \in \mathsf{range}(\mathbb{C}.imp), c_2 \in \mathsf{range}(t_1.imp).\ c_1 \cap c_2 = \emptyset$**:**
    Follows from the checks obtained by inverting rule Module-list-translation and inverting the precondition $\mathtt{wfp\_params}(\overline{m}, \cdots)$ using rule Well-formed program and parameters after first applying Lemmas 91 and 92.
  - **Subgoal disjointness of capabilities** $\forall c_1 \in \mathsf{range}(\mathbb{C}.\mathsf{mstc}), c_2 \in \mathsf{range}(t_1.\mathsf{mstc}).\ c_1 \cap c_2 = \emptyset$**:**
    Follows from the checks obtained by inverting rule Module-list-translation and inverting the precondition $\mathtt{wfp\_params}(\overline{m}, \cdots)$ using rule Well-formed program and parameters after first applying Lemmas 91 and 92.

The next three conjuncts of our goal thus follow immediately from the corresponding conjuncts of Proposition (4).

This concludes the proof of Lemma 106. ☐

Lemma 107 (Compiler is a linking-preserving homomorphism)**.**

$$\llbracket \mathbb{C} \rrbracket_{\Delta,\Sigma,\beta,K_{mod},K_{fun}} \ltimes \llbracket \overline{m_1} \rrbracket_{\tilde{\Delta},\tilde{\Sigma},\beta_1,K_{mod1},K_{fun1}} = \llbracket \mathbb{C}[\overline{m_1}]_{\Delta\uplus\tilde{\Delta},\Sigma\uplus\tilde{\Sigma}} \rrbracket_{\Delta\uplus\tilde{\Delta},\Sigma\uplus\tilde{\Sigma},\beta\uplus\beta_1,K_{mod}\uplus K_{mod1},K_{fun}\uplus K_{fun1}}$$

Proof.
Similar to Lemma 106. ☐

4 A sound trace semantics for CHERIExp

We give a sound and complete trace semantics for **CHERIExp**. In this section, we prove soundness only (Lemma 114). Completeness, on the other hand, follows as an immediate corollary (Corollary 12) from results about the compiler of Section 3.

We first give the trace actions $\lambda \in \Lambda$:

$\lambda ::=$	$\checkmark$	termination marker
	$\mid \tau$	silent internal action
	$\mid \texttt{call}(mid, fid)\overline{v}?\mathcal{M}_d, n$	receive a call
	$\mid \texttt{ret}?\mathcal{M}_d, n$	receive a return
	$\mid \texttt{call}(mid, fid)\overline{v}!\mathcal{M}_d, n$	issue a call
	$\mid \texttt{ret}!\mathcal{M}_d, n$	issue a return

We next state useful definitions and lemmas about the trace semantics which we give in Figure 9 and about **CHERIExp** and the compiler.

Trace prefixes $\alpha \in \Lambda^+$ are finite sequences of actions. They describe an abstraction of the behavior of the program as given by a finite sequence of its reduction steps. The emphasis that is made by the abstraction is on the so-called "boundary-crossing" actions. In the interesting case when the boundary is set to be "compiled part of the program" vs. "arbitrary **CHERIExp** linked context", the trace behavior of a program helps in reasoning about the boundary-crossing actions which turn out to be sufficient to capture the observable behavior of compiled programs.

The action $\checkmark$ indicates that execution has reached a terminal state. Silent actions τ are actions that do not change ownership of the program counter capability pcc. Ownership of pcc is whether it points to an address in one partition of the code memory (out of two designated partitions). Actions that are marked with a ? indicate incoming function calls or returns (with respect to a designated partition of the program), and actions that are marked with a ! indicate on the other hand the outgoing-directed function calls or returns. In our proofs, the partition is such that the actions performed by the part of the program that is compiled with our compiler are distinguished from the actions that are performed by the **CHERIExp** context that is linked with the compiled program.

An incoming call action $\texttt{call}(mid, fid)?\mathcal{M}_d, n$ records, as indicated by rule cinvoke-context-to-compiled in Figure 9 that a Cinvoke command has been executed, where the function *fid* in module *mid* is being called, and the projection $\mathcal{M}_d$ of the data memory is the recording of the values in all the data memory locations that have in the past been shared between the two parts of the program. The number n indicates the memory consumption of the program so far. The return action $\texttt{ret}?\mathcal{M}_d, n$ also records the same about the data memory and the memory consumption. And outgoing call and return actions are analogous to incoming ones.

Alternating traces

Let $\dot{?} ::= \texttt{call}\ (mid, fid)?\mathcal{M}_d, n \ \mid \texttt{ret}?\mathcal{M}_d, n$ and $\dot{!} ::= \texttt{call}(mid, fid)!\mathcal{M}_d, n \mid \texttt{ret}!\mathcal{M}_d, n$. And let $\alpha|_{\not\tau} \overset{\mathsf{def}}{=} \pi_{\Lambda\setminus\{\tau\}}(\alpha)$. And define the set Alt of finite alternating traces as follows:

Definition 69 (Alternatingly-communicating finite traces)**.** *We define the set* Alt *of finite traces where communication is alternating as follows:* $\texttt{Alt} \overset{\mathsf{def}}{=} (\dot{?}|\epsilon)\ (\dot{!}\ \dot{?})^*\ (\dot{!}|\epsilon)$

Claim 5 (Extending an alternating prefix to keep it alternating)**.**

1. $(\alpha\lambda \in \texttt{Alt} \land \lambda \in \dot{?} \land \lambda' \in \dot{!}) \implies \alpha\lambda\lambda' \in \texttt{Alt}$
2. $(\alpha\lambda \in \texttt{Alt} \land \lambda \in \dot{!} \land \lambda' \in \dot{?}) \implies \alpha\lambda\lambda' \in \texttt{Alt}$

Figure 9: Trace semantics for **CHERIExp** for an arbitrary compiled component $\overline{c} : \mathit{TargetSetup}$

(assign-silent)

$$\frac{\mathcal{M}_c(\mathsf{pcc}) = \mathsf{Assign}\ \mathcal{E}_L\ \mathcal{E}_R \qquad \langle \mathcal{M}_c, \mathcal{M}_d, \mathit{stk}, \mathit{imp}, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle \rightarrow_{\triangledown} s'}{\langle \mathcal{M}_c, \mathcal{M}_d, \mathit{stk}, \mathit{imp}, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle, \varsigma \xrightarrow{\tau}_{[\overline{c}], \triangledown} s', \varsigma}$$

(alloc-silent)

$$\frac{\mathcal{M}_c(\mathsf{pcc}) = \mathsf{Alloc}\ \mathcal{E}_L\ \mathcal{E}_{size} \qquad \langle \mathcal{M}_c, \mathcal{M}_d, \mathit{stk}, \mathit{imp}, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle \rightarrow_{\triangledown} s'}{\langle \mathcal{M}_c, \mathcal{M}_d, \mathit{stk}, \mathit{imp}, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle, \varsigma \xrightarrow{\tau}_{[\overline{c}], \triangledown} s', \varsigma}$$

(jump-silent)

$$\frac{\mathcal{M}_c(\mathsf{pcc}) = \mathsf{JumpIfZero}\ \mathcal{E}_{cond}\ \mathcal{E}_{off} \qquad \langle \mathcal{M}_c, \mathcal{M}_d, \mathit{stk}, \mathit{imp}, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle \rightarrow_{\triangledown} s'}{\langle \mathcal{M}_c, \mathcal{M}_d, \mathit{stk}, \mathit{imp}, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle, \varsigma \xrightarrow{\tau}_{[\overline{c}], \triangledown} s', \varsigma}$$

(cinvoke-silent-compiled)

$$\frac{\begin{array}{c}\mathcal{M}_c(\mathsf{pcc}) = \mathsf{Cinvoke}\ \mathit{mid}\ \mathit{fid}\ \overline{e} \qquad \langle \mathcal{M}_c, \mathcal{M}_d, \mathit{stk}, \mathit{imp}, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle \rightarrow_{\triangledown} s' \\ \overline{c} = (\mathcal{M}_{c_{[\overline{c}]}}, \mathcal{M}_{d_{[\overline{c}]}}, \mathit{imp}_{[\overline{c}]}) \qquad \mathsf{pcc} \subseteq \mathsf{dom}(\mathcal{M}_{c_{[\overline{c}]}}) \qquad \mathit{mid} \in \mathsf{dom}(\mathit{imp}_{[\overline{c}]})\end{array}}{\langle \mathcal{M}_c, \mathcal{M}_d, \mathit{stk}, \mathit{imp}, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle, \varsigma \xrightarrow{\tau}_{[\overline{c}], \triangledown} s', \varsigma}$$

(cinvoke-silent-context)

$$\frac{\begin{array}{c}\mathcal{M}_c(\mathsf{pcc}) = \mathsf{Cinvoke}\ \mathit{mid}\ \mathit{fid}\ \overline{e} \qquad \langle \mathcal{M}_c, \mathcal{M}_d, \mathit{stk}, \mathit{imp}, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle \rightarrow_{\triangledown} s' \\ \overline{c} = (\mathcal{M}_{c_{[\overline{c}]}}, \mathcal{M}_{d_{[\overline{c}]}}, \mathit{imp}_{[\overline{c}]}) \qquad \mathsf{pcc} \not\subseteq \mathsf{dom}(\mathcal{M}_{c_{[\overline{c}]}}) \qquad \mathit{mid} \notin \mathsf{dom}(\mathit{imp}_{[\overline{c}]})\end{array}}{\langle \mathcal{M}_c, \mathcal{M}_d, \mathit{stk}, \mathit{imp}, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle, \varsigma \xrightarrow{\tau}_{[\overline{c}], \triangledown} s', \varsigma}$$

(cinvoke-context-to-compiled)

$$\frac{\begin{array}{c} s = \langle \mathcal{M}_c, \mathcal{M}_d, \mathit{stk}, \mathit{imp}, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle \\ s' = \langle \mathcal{M}_c, \mathcal{M}'_d, \mathit{stk}', \mathit{imp}, \phi, \mathsf{ddc}', \mathsf{stc}', \mathsf{pcc}', \mathsf{mstc}', \mathsf{nalloc}'\rangle \qquad s \succ_{\approx} s' \\ s_{\bot} = s'[\mathcal{M}_c \mapsto s'.\mathcal{M}_c[s'.\mathsf{pcc} \mapsto \bot]] \qquad s \not\rightarrow_{\triangledown} s' \implies s'' = s_{\bot} \qquad s \rightarrow_{\triangledown} s' \implies s'' = s' \\ \mathcal{M}_c(\mathsf{pcc}) = \mathsf{Cinvoke}\ \mathit{mid}\ \mathit{fid}\ \overline{e} \qquad \overline{v} = [i \mapsto v_i \mid \ \forall i \in [0, \mathit{nArgs})\ \ \overline{e}(i), \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow v_i] \\ r = \mathsf{reachable\_addresses}(\{\overline{v}(i) \mid i \in [0, \mathit{nArgs}) \wedge \overline{v}(i) = (\delta, \_, \_, \_)\}, \mathcal{M}'_d) \\ \varsigma' = \mathsf{reachable\_addresses\_closure}(\varsigma \cup r, \mathcal{M}'_d) \\ \overline{c} = (\mathcal{M}_{c_{[\overline{c}]}}, \mathcal{M}_{d_{[\overline{c}]}}, \mathit{imp}_{[\overline{c}]}) \qquad \mathsf{pcc} \not\subseteq \mathsf{dom}(\mathcal{M}_{c_{[\overline{c}]}}) \qquad \mathit{mid} \in \mathsf{dom}(\mathit{imp}_{[\overline{c}]}) \end{array}}{s, \varsigma \xrightarrow{\mathtt{call}(\mathit{mid}, \mathit{fid})\overline{v}?\mathcal{M}'_d|_{\varsigma'}, \mathsf{nalloc}'}_{[\overline{c}], \triangledown} s'', \varsigma'}$$

(cinvoke-compiled-to-context)

$$\frac{\begin{array}{c} s = \langle \mathcal{M}_c, \mathcal{M}_d, \mathit{stk}, \mathit{imp}, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle \\ s' = \langle \mathcal{M}_c, \mathcal{M}'_d, \mathit{stk}', \mathit{imp}, \phi, \mathsf{ddc}', \mathsf{stc}', \mathsf{pcc}', \mathsf{mstc}', \mathsf{nalloc}'\rangle \qquad s \succ_{\approx} s' \\ s_{\bot} = s'[\mathcal{M}_c \mapsto s'.\mathcal{M}_c[s'.\mathsf{pcc} \mapsto \bot]] \qquad s \not\rightarrow_{\triangledown} s' \implies s'' = s_{\bot} \qquad s \rightarrow_{\triangledown} s' \implies s'' = s' \\ \mathcal{M}_c(\mathsf{pcc}) = \mathsf{Cinvoke}\ \mathit{mid}\ \mathit{fid}\ \overline{e} \qquad \overline{v} = [i \mapsto v_i \mid \ \forall i \in [0, \mathit{nArgs})\ \ \overline{e}(i), \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow v_i] \\ r = \mathsf{reachable\_addresses}(\{\overline{v}(i) \mid i \in [0, \mathit{nArgs}) \wedge \overline{v}(i) = (\delta, \_, \_, \_)\}, \mathcal{M}'_d) \\ \varsigma' = \mathsf{reachable\_addresses\_closure}(\varsigma \cup r, \mathcal{M}'_d) \\ \overline{c} = (\mathcal{M}_{c_{[\overline{c}]}}, \mathcal{M}_{d_{[\overline{c}]}}, \mathit{imp}_{[\overline{c}]}) \qquad \mathsf{pcc} \subseteq \mathsf{dom}(\mathcal{M}_{c_{[\overline{c}]}}) \qquad \mathit{mid} \notin \mathsf{dom}(\mathit{imp}_{[\overline{c}]}) \end{array}}{s, \varsigma \xrightarrow{\mathtt{call}(\mathit{mid}, \mathit{fid})\overline{v}!\mathcal{M}'_d|_{\varsigma'}, \mathsf{nalloc}'}_{[\overline{c}], \triangledown} s'', \varsigma'}$$

(creturn-silent-compiled)

$$\frac{\begin{array}{c}\mathcal{M}_c(\mathsf{pcc}) = \mathsf{Creturn} \qquad s = \langle \mathcal{M}_c, \mathcal{M}_d, \mathit{stk}, \mathit{imp}, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle \\ s' = \langle \mathcal{M}_c, \mathcal{M}'_d, \mathit{stk}', \mathit{imp}, \phi, \mathsf{ddc}', \mathsf{stc}', \mathsf{pcc}', \mathsf{mstc}', \mathsf{nalloc}'\rangle \\ s \rightarrow_{\triangledown} s' \qquad \overline{c} = (\mathcal{M}_{c_{[\overline{c}]}}, \mathcal{M}_{d_{[\overline{c}]}}, \mathit{imp}_{[\overline{c}]}) \qquad \mathsf{pcc} \subseteq \mathsf{dom}(\mathcal{M}_{c_{[\overline{c}]}}) \qquad \mathsf{pcc}' \subseteq \mathsf{dom}(\mathcal{M}_{c_{[\overline{c}]}})\end{array}}{s, \varsigma \xrightarrow{\tau}_{[\overline{c}], \triangledown} s', \varsigma}$$

Figure 9 (Cont.): Trace semantics for **CHERIExp** for an arbitrary compiled component $\bar{c} : \mathit{TargetSetup}$

$$\frac{\begin{array}{c}\mathcal{M}_c(\mathsf{pcc}) = \mathsf{Creturn} \quad s = \langle \mathcal{M}_c, \mathcal{M}_d, \mathit{stk}, \mathit{imp}, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle \\ s' = \langle \mathcal{M}_c, \mathcal{M}'_d, \mathit{stk}', \mathit{imp}, \phi, \mathsf{ddc}', \mathsf{stc}', \mathsf{pcc}', \mathsf{mstc}', \mathsf{nalloc}'\rangle \\ s \to_{\triangledown} s' \quad \bar{c} = (\mathcal{M}_{c_{[\bar{c}]}}, \mathcal{M}_{d_{[\bar{c}]}}, \mathit{imp}_{[\bar{c}]}) \quad \mathsf{pcc} \nsubseteq \mathsf{dom}(\mathcal{M}_{c_{[\bar{c}]}}) \quad \mathsf{pcc}' \nsubseteq \mathsf{dom}(\mathcal{M}_{c_{[\bar{c}]}})\end{array}}{s, \varsigma \xrightharpoonup{\tau}_{[\bar{c}],\triangledown} s', \varsigma}\ \text{(creturn-silent-context)}$$

$$\frac{\begin{array}{c}\mathcal{M}_c(\mathsf{pcc}) = \mathsf{Creturn} \quad s = \langle \mathcal{M}_c, \mathcal{M}_d, \mathit{stk}, \mathit{imp}, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle \\ s \to_{\triangledown} s' \quad s' = \langle \mathcal{M}_c, \mathcal{M}'_d, \mathit{stk}', \mathit{imp}, \phi, \mathsf{ddc}', \mathsf{stc}', \mathsf{pcc}', \mathsf{mstc}', \mathsf{nalloc}\rangle \\ \bar{c} = (\mathcal{M}_{c_{[\bar{c}]}}, \mathcal{M}_{d_{[\bar{c}]}}, \mathit{imp}_{[\bar{c}]}) \quad \mathsf{pcc} \nsubseteq \mathsf{dom}(\mathcal{M}_{c_{[\bar{c}]}}) \quad \mathsf{pcc}' \subseteq \mathsf{dom}(\mathcal{M}_{c_{[\bar{c}]}}) \\ \varsigma' = \mathsf{reachable\_addresses\_closure}(\varsigma, \mathcal{M}'_d)\end{array}}{s, \varsigma \xrightharpoonup{\mathtt{ret?}\mathcal{M}'_d|_{\varsigma'},\mathsf{nalloc}}_{[\bar{c}],\triangledown} s', \varsigma'}\ \text{(creturn-to-compiled)}$$

$$\frac{\begin{array}{c}\mathcal{M}_c(\mathsf{pcc}) = \mathsf{Creturn} \quad s = \langle \mathcal{M}_c, \mathcal{M}_d, \mathit{stk}, \mathit{imp}, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle \quad s \to_{\triangledown} s' \\ s' = \langle \mathcal{M}_c, \mathcal{M}'_d, \mathit{stk}', \mathit{imp}, \phi, \mathsf{ddc}', \mathsf{stc}', \mathsf{pcc}', \mathsf{mstc}', \mathsf{nalloc}\rangle \\ \bar{c} = (\mathcal{M}_{c_{[\bar{c}]}}, \mathcal{M}_{d_{[\bar{c}]}}, \mathit{imp}_{[\bar{c}]}) \quad \mathsf{pcc} \subseteq \mathsf{dom}(\mathcal{M}_{c_{[\bar{c}]}}) \quad \mathsf{pcc}' \nsubseteq \mathsf{dom}(\mathcal{M}_{c_{[\bar{c}]}}) \\ \varsigma' = \mathsf{reachable\_addresses\_closure}(\varsigma, \mathcal{M}'_d)\end{array}}{s, \varsigma \xrightharpoonup{\mathtt{ret!}\mathcal{M}'_d|_{\varsigma'},\mathsf{nalloc}}_{[\bar{c}],\triangledown} s', \varsigma'}\ \text{(creturn-to-context)}$$

$$\frac{\vdash_t s}{s, \varsigma \xrightharpoonup{\checkmark}_{[\bar{c}],\triangledown} s, \varsigma}\ \text{(terminate-checkmark)}$$

Definition 70 (Reflexive transitive closure of trace actions).
We write $s \xrightharpoonup{\alpha}{}^{*}_{[\bar{c}],\triangledown} s'$ where $\rightharpoonup^{*}_{[\bar{c}],\triangledown} \subseteq (\mathit{TargetState} \times 2^{\mathbb{Z}}) \times \overline{\Lambda} \times (\mathit{TargetState} \times 2^{\mathbb{Z}})$ to denote the reflexive transitive closure of the trace actions reduction relation $\rightharpoonup_{[\bar{c}],\triangledown} \subseteq (\mathit{TargetState} \times 2^{\mathbb{Z}}) \times \Lambda \times (\mathit{TargetState} \times 2^{\mathbb{Z}})$ where α collects the individual trace actions in succession.

$$\frac{}{s, \varsigma \xrightharpoonup{\epsilon}{}^{*}_{[\bar{c}],\triangledown} s, \varsigma}\ \textit{(trace-closure-refl)} \qquad \frac{s, \varsigma \xrightharpoonup{\alpha}{}^{*}_{[\bar{c}],\triangledown} s'', \varsigma'' \quad s'', \varsigma'' \xrightharpoonup{\lambda}_{[\bar{c}],\triangledown} s', \varsigma'}{s, \varsigma \xrightharpoonup{\alpha\lambda}{}^{*}_{[\bar{c}],\triangledown} s', \varsigma'}\ \textit{(trace-closure-trans)}$$

where $\rightharpoonup_{[\bar{c}],\triangledown} \subseteq (\mathit{TargetState} \times 2^{\mathbb{Z}}) \times \Lambda \times (\mathit{TargetState} \times 2^{\mathbb{Z}})$ is as defined in Figure 9.

Definition 71 (Non-silent trace steps).
We write $s \xrightharpoonup{\alpha}_{[\bar{c}],\triangledown} s'$ where $\twoheadrightarrow_{[\bar{c}],\triangledown} \subseteq (\mathit{TargetState} \times 2^{\mathbb{Z}}) \times \overline{\Lambda} \times (\mathit{TargetState} \times 2^{\mathbb{Z}})$ to denote that execution on state s generates a sequence α of non-silent trace actions (i.e., excluding τ actions) and reaches state s'. We sometimes drop the parameter $\triangledown$ (which is the upper limit on memory allocation) for convenience.

$$\frac{s, \varsigma \xrightharpoonup{\tau^*}{}^{*}_{[\bar{c}],\triangledown} s'', \varsigma'' \quad s'', \varsigma'' \xrightharpoonup{\lambda}_{[\bar{c}],\triangledown} s', \varsigma' \quad \lambda \neq \tau}{s, \varsigma \xrightharpoonup{\lambda}_{[\bar{c}],\triangledown} s', \varsigma'}\ \textit{(trace-steps-lambda)}$$

(trace-steps-alternating)

$$\frac{s, \varsigma \overset{\alpha}{\rightharpoonup\!\!\rightharpoonup}_{[\overline{c}],\nabla} s''', \varsigma''' \qquad s''', \varsigma''' \overset{\tau^*}{\rightharpoonup}^{*}_{[\overline{c}],\nabla} s'', \varsigma'' \qquad s'', \varsigma'' \overset{\lambda}{\rightharpoonup}_{[\overline{c}],\nabla} s', \varsigma' \qquad \lambda \neq \tau}{s, \varsigma \overset{\alpha\lambda}{\rightharpoonup\!\!\rightharpoonup}_{[\overline{c}],\nabla} s', \varsigma'}$$

Claim 6 (A non-silent trace is not the empty string)**.**

$$\begin{aligned}
&\forall \overline{c}, \alpha, s, \varsigma, s', \varsigma', \nabla. \\
&s, \varsigma \overset{\alpha}{\rightharpoonup\!\!\rightharpoonup}_{[\overline{c}],\nabla} s', \varsigma' \\
&\implies \\
&|\alpha| > 1
\end{aligned}$$

Claim 7 ($\rightharpoonup\!\!\rightharpoonup$ eliminates τ actions)**.**

$$\begin{aligned}
&\forall \overline{c}, \alpha, s, \varsigma, s', \varsigma', \nabla. \\
&s, \varsigma \overset{\alpha\lambda}{\rightharpoonup\!\!\rightharpoonup}_{[\overline{c}],\nabla} s', \varsigma' \\
&\implies \\
&\lambda \neq \tau
\end{aligned}$$

Claim 8 ($\rightharpoonup\!\!\rightharpoonup$ is supported by $\rightharpoonup$)**.**

$$\begin{aligned}
&\forall \overline{c}, \alpha, \lambda, s, \varsigma, s', \varsigma', \nabla. \\
&s, \varsigma \overset{\alpha\lambda}{\rightharpoonup\!\!\rightharpoonup}_{[\overline{c}],\nabla} s', \varsigma' \\
&\implies \\
&\exists s'', \varsigma''. \\
&s'', \varsigma'' \overset{\lambda}{\rightharpoonup}_{[\overline{c}],\nabla} s', \varsigma' \;\wedge \\
&s, \varsigma \overset{\alpha}{\rightharpoonup}^{*}_{[\overline{c}],\nabla} s'', \varsigma''
\end{aligned}$$

Claim 9 ($\rightharpoonup\!\!\rightharpoonup$ decomposes)**.**

$$\begin{aligned}
&\forall \overline{c}, \alpha_1, \alpha_2, s, \varsigma, s', \varsigma', \nabla. \\
&s, \varsigma \overset{\alpha_1\alpha_2}{\rightharpoonup\!\!\rightharpoonup}_{[\overline{c}],\nabla} s', \varsigma' \\
&\implies \\
&\exists s_1, \varsigma_1. \\
&s, \varsigma \overset{\alpha_1}{\rightharpoonup\!\!\rightharpoonup}_{[\overline{c}],\nabla} s_1, \varsigma_1 \;\wedge \\
&s_1, \varsigma_1 \overset{\alpha_2}{\rightharpoonup\!\!\rightharpoonup}_{[\overline{c}],\nabla} s', \varsigma' \;\wedge
\end{aligned}$$

Claim 10 (Non-silent part of $\rightharpoonup^*$ is supported by $\rightharpoonup\!\!\rightharpoonup$)**.**

$$\begin{aligned}
&\forall \overline{c}, \alpha, s, \varsigma, s', \varsigma', \nabla. \\
&|\ \alpha|_{\not\tau}\ | \geq 1 \;\wedge \\
&s, \varsigma \overset{\alpha}{\rightharpoonup}^{*}_{[\overline{c}],\nabla} s', \varsigma' \\
&\implies \\
&\exists s'', \varsigma''.\ s, \varsigma \overset{\alpha|_{\not\tau}}{\rightharpoonup\!\!\rightharpoonup}_{[\overline{c}],\nabla} s'', \varsigma''
\end{aligned}$$

For a target program $\overline{c} : \mathit{TargetSetup}$, we define the set $\mathit{TR}(\overline{c}) \subseteq \Lambda^+$ of finite non-empty prefixes of $\overline{c}$'s possible execution traces as follows:

Definition 72 (A prefix of an execution trace is possible for a component)**.**
A finite prefix α *belonging to a component* $\bar{c}$*'s set* $TR_{\omega,\nabla}(\bar{c})$ *of possible execution trace prefixes is defined as:*

$$\begin{aligned}\alpha \in TR_{\omega,\nabla}(\bar{c}) \iff\ & \exists \mathbb{C}, t' : \mathit{TargetSetup}, s' : \mathit{TargetState}, \varsigma' : 2^{\mathbb{Z}}.\\ & \mathbb{C} \ltimes \bar{c} = \lfloor t' \rfloor \wedge \\ & \mathsf{initial\_state}(t' + \omega, \mathsf{main\_module}(t')), \emptyset \overset{\alpha}{\rightharpoonup}_{[\bar{c}],\nabla} s', \varsigma'\end{aligned}$$

where $\rightharpoonup_{[\bar{c}],\nabla} \subseteq (\mathit{TargetState} \times 2^{\mathbb{Z}}) \times \overline{\Lambda} \times (\mathit{TargetState} \times 2^{\mathbb{Z}})$ *is as defined in Definition 71.*

Definition 73 (Trace equivalence)**.**

$$\overline{c_1} \overset{\mathrm{T}}{=}_{\omega,\nabla} \overline{c_2} \overset{\mathsf{def}}{=} TR_{\omega,\nabla}(\overline{c_1}) = TR_{\omega,\nabla}(\overline{c_2})$$

Claim 11 (Termination markers appear only at the end of an execution trace)**.**

$$\forall \bar{c}.\ \alpha \in TR(\bar{c}) \implies \alpha \in (\Lambda \setminus \{\checkmark\})^* \vee \alpha \in (\Lambda \setminus \{\checkmark\})^* \checkmark$$

Claim 12 (Prefix-closure of trace set membership)**.**

$$\forall \bar{c}, \alpha.\ \alpha \in TR(\bar{c}) \implies (\forall \alpha'.\ \alpha = \alpha' \alpha'' \implies \alpha' \in TR(\bar{c}))$$

Proof.
Follows from Claim 9. Instantiate "$\implies$" direction of Definition 72 using the assumption, and apply its "$\impliedby$" direction to the goal. □

Claim 13 (A state that is reachable by $\rightarrow$ reduction or by $\succ_{\approx}$ is also reachable by $\rightharpoonup$)**.**

$$\begin{aligned}& \forall \bar{c}, t, s, s', \varsigma, \nabla.\\ & (s \rightarrow_{\nabla} s' \vee s \succ_{\approx} s')\\ & \implies\\ & \exists \lambda, \varsigma'.\ s, \varsigma \overset{\lambda}{\rightharpoonup}_{[\bar{c}],\nabla} s', \varsigma'\end{aligned}$$

Claim 14 (A non-$\perp$ state that is reachable by $\rightharpoonup$ is also reachable by $\rightarrow$ reduction)**.**

$$\begin{aligned}& \forall t, \bar{c}, s, s', \varsigma, \varsigma'.\\ & s'.\mathcal{M}_c(s'.\mathsf{pcc}) \neq \perp \wedge\\ & s, \varsigma \overset{\lambda}{\rightharpoonup}_{[\bar{c}],\nabla} s', \varsigma'\\ & \implies\\ & s \rightarrow_{\nabla} s'\end{aligned}$$

Claim 15 (Silent trace steps correspond to $\rightarrow$ steps)**.**

$$\begin{aligned}& \forall \bar{c}, s, s', \varsigma, \varsigma', \nabla.\\ & s, \varsigma \overset{\tau^*}{\rightharpoonup}{}^*_{[\bar{c}],\nabla} s', \varsigma'\\ & \implies\\ & s \rightarrow^*_{\nabla} s'\end{aligned}$$

Claim 16 (Non-stuck trace steps correspond to $\rightarrow$ execution steps)**.**

$$\begin{aligned}& \forall \bar{c}, s, s', s'', \varsigma, \varsigma', \varsigma'', \nabla.\\ & s, \varsigma \overset{\alpha}{\rightharpoonup}{}^*_{[\bar{c}],\nabla} s', \varsigma' \wedge\\ & s', \varsigma' \overset{\lambda}{\rightharpoonup}_{[\bar{c}],\nabla} s'', \varsigma''\\ & \implies\\ & s \rightarrow^*_{\nabla} s'\end{aligned}$$

Claim 17 (The set of shared addresses ς does not change by silent trace steps)**.**

$$\forall s, s', \varsigma, \varsigma', \nabla.$$
$$s, \varsigma \xrightharpoonup{\tau^*}{}^{*}_{[\overline{c}], \nabla} s', \varsigma'$$
$$\implies$$
$$\varsigma = \varsigma'$$

Corollary 5 (Reachability by $\rightarrow^*$ implies reachability by $\rightharpoonup^*$)**.**

$$\forall t_1, t_2, \omega, \nabla, s.$$
$$\mathsf{initial\_state}(t_1 \ltimes t_2 + \omega, \mathsf{main\_module}(t_1 \ltimes t_2)) \rightarrow^*_{\nabla} s$$
$$\implies$$
$$\exists \varsigma, \alpha.\ \mathsf{initial\_state}(t_1 \ltimes t_2 + \omega, \mathsf{main\_module}(t_1 \ltimes t_2)), \emptyset \xrightharpoonup{\alpha}{}^{*}_{[t_2], \nabla} s, \varsigma$$

Corollary 6 (Reachability by $\rightharpoonup^*$ implies reachability by $\rightarrow^*$ when the state is non-$\bot$)**.**

$$\forall t_1, t_2, \omega, \nabla, s, \varsigma, \alpha.$$
$$\mathsf{initial\_state}(t_1 \ltimes t_2 + \omega, \mathsf{main\_module}(t_1 \ltimes t_2)), \emptyset \xrightharpoonup{\alpha}{}^{*}_{[t_2], \nabla} s, \varsigma \wedge$$
$$s.\mathcal{M}_c(s.\mathsf{pcc}) \neq \bot$$
$$\implies$$
$$\mathsf{initial\_state}(t_1 \ltimes t_2 + \omega, \mathsf{main\_module}(t_1 \ltimes t_2)) \rightarrow^*_{\nabla} s$$

Lemma 108 (Non-communication actions do not change context/compiled component's ownership of pcc)**.**

$$\forall \overline{c}, t : \mathit{TargetSetup}, s, s'.$$
$$t \ltimes \overline{c} \vdash_{exec} s \wedge$$
$$s \xrightharpoonup{\tau}_{[\overline{c}]} s'$$
$$\implies$$
$$(s.\mathsf{pcc} \subseteq \mathsf{dom}(\overline{c}.\mathcal{M}_c) \iff s'.\mathsf{pcc} \subseteq \mathsf{dom}(\overline{c}.\mathcal{M}_c))$$

Proof. Fix arbitrary, $\overline{c}$, t, s, and s', and assume the antecedents.

- **Subgoal** $s.\mathsf{pcc} \subseteq \mathsf{dom}(\overline{c}.\mathcal{M}_c) \implies s'.\mathsf{pcc} \subseteq \mathsf{dom}(\overline{c}.\mathcal{M}_c)$**:**

  Assume $s.\mathsf{pcc} \subseteq \mathsf{dom}(\overline{c}.\mathcal{M}_c)$

  Our goal is:
  $s'.\mathsf{pcc} \subseteq \mathsf{dom}(\overline{c}.\mathcal{M}_c)$

  Distinguish the following cases for assumption $s \xrightharpoonup{\tau}_{[\overline{c}]} s'$.

  - **Case assign-silent:**
    Here, by inversion of the preconditions using rule assign, obtain:
    $s.\mathsf{pcc} \doteq s'.\mathsf{pcc}$
    Thus, our goal follows by substitution using assumption $s.\mathsf{pcc} \subseteq \mathsf{dom}(\overline{c}.\mathcal{M}_c)$.
  - **Case alloc-silent:**
    Here, by inversion of the preconditions using rule allocate, obtain:
    $s.\mathsf{pcc} \doteq s'.\mathsf{pcc}$
    Thus, our goal follows by substitution using assumption $s.\mathsf{pcc} \subseteq \mathsf{dom}(\overline{c}.\mathcal{M}_c)$.

- **Case jump-silent:**
  Here, distinguish two cases for inversion of $s \to s'$:
  * **Case jump0:**
    Here, obtain $s'.\mathsf{pcc} = \mathtt{inc}(s.\mathsf{pcc}, \_)$.
    Thus, have:
    $s.\mathsf{pcc} \doteq s'.\mathsf{pcc}$
    Thus, our goal follows by substitution using assumption $s.\mathsf{pcc} \subseteq \mathsf{dom}(\bar{c}.\mathcal{M}_c)$.
  * **Case jump1:**
    Here, obtain $s'.\mathsf{pcc} = \mathtt{inc}(s.\mathsf{pcc}, 1)$.
    Thus, have:
    $s.\mathsf{pcc} \doteq s'.\mathsf{pcc}$
    Thus, our goal follows by substitution using assumption $s.\mathsf{pcc} \subseteq \mathsf{dom}(\bar{c}.\mathcal{M}_c)$.
- **Case cinvoke-silent-compiled:**
  Here, obtain:
  $s.\mathcal{M}_c(s.\mathsf{pcc}) = \mathsf{Cinvoke}\ \mathit{mid}\ \mathit{fid}\ \bar{e}$,
  $\mathit{mid} \in \mathsf{dom}(\bar{c}.\mathit{imp})$, and
  $s \to s'$
  Thus, by inversion using cinvoke then cinvoke-aux, have (*):
  $s'.\mathsf{pcc} \doteq s.\mathit{imp}(\mathit{mid}).\mathsf{pcc}$
  By inversion of lemma antecedents using valid-linking and valid-program, we know:
  $\mathit{mid} \in \mathsf{dom}(\bar{c}.\mathit{imp}) \implies s.\mathit{imp}(\mathit{mid}).\mathsf{pcc} \subseteq \mathsf{dom}(\bar{c}.\mathcal{M}_c)$ (applied Lemma 2)
  Instantiating the latter using our assumptions, and substituting using (*), we have our goal.
- **Case cinvoke-silent-context:**
  Precondition $s.\mathsf{pcc} \not\subseteq \mathsf{dom}(\bar{c}.\mathcal{M}_c)$ contradicts our assumption. So, any goal is provable.
- **Case creturn-silent-compiled:**
  Goal is immediate by the precondition of rule creturn-silent-compiled.
- **Case creturn-silent-context:**
  Precondition $s.\mathsf{pcc} \not\subseteq \mathsf{dom}(\bar{c}.\mathcal{M}_c)$ of rule creturn-silent-context contradicts our assumption. So, any goal is provable.

- **Subgoal $s.\mathsf{pcc} \subseteq \mathsf{dom}(\bar{c}.\mathcal{M}_c) \impliedby s'.\mathsf{pcc} \subseteq \mathsf{dom}(\bar{c}.\mathcal{M}_c)$:**

  Assume $s'.\mathsf{pcc} \subseteq \mathsf{dom}(\bar{c}.\mathcal{M}_c)$

  Our goal is:
  $s.\mathsf{pcc} \subseteq \mathsf{dom}(\bar{c}.\mathcal{M}_c)$

  Distinguish the following cases for assumption $s \xrightarrow{\tau}_{[\bar{c}]} s'$.

  - **Case assign-silent,**
  - **Case alloc-silent, and**
  - **Case jump-silent:**
    Similar to the corresponding cases of the previous subgoal: Goal follows by substitution using the assumption after obtaining $s.\mathsf{pcc} \doteq s'.\mathsf{pcc}$.
  - **Case cinvoke-silent-compiled:**
    Goal is immediate by preconditions of cinvoke-silent-compiled.
  - **Case cinvoke-silent-context:**
    Obtain a contradiction to assumption
    $s'.\mathsf{pcc} \subseteq \mathsf{dom}(\bar{c}.\mathcal{M}_c)$

by proving:
$s'.\mathsf{pcc} \nsubseteq \mathsf{dom}(\bar{c}.\mathcal{M}_c)$
First, obtain:
$s.\mathcal{M}_c(s.\mathsf{pcc}) = \mathsf{Cinvoke}\ mid\ fid\ \bar{e}$,
$mid \notin \mathsf{dom}(\bar{c}.imp)$, and
$s \to s'$
Thus, by inversion using cinvoke then cinvoke-aux, have (*):
$s'.\mathsf{pcc} \doteq s.imp(mid).\mathsf{pcc}$
By inversion of lemma antecedents using valid-linking and valid-program, we know:
$mid \notin \mathsf{dom}(\bar{c}.imp) \implies s.imp(mid).\mathsf{pcc} \nsubseteq \mathsf{dom}(\bar{c}.\mathcal{M}_c)$ (applied Lemma 2)
Instantiating the latter using our assumptions, and substituting using (*), we have our goal.

– **Case creturn-silent-compiled:**
Goal is immediate by the preconditions of creturn-silent-compiled.

– **Case creturn-silent-context:**
Precondition $s'.\mathsf{pcc} \nsubseteq \mathsf{dom}(\bar{c}.\mathcal{M}_c)$, so any goal is provable.

This concludes the proof of Lemma 108. □

Corollary 7 (Non-communication actions do not change ownership of pcc (star-closure))**.**

$$\begin{aligned}
&\forall \bar{c}, t : TargetSetup, s, s'.\\
&t \ltimes \bar{c} \vdash_{exec} s \wedge\\
&s, \varsigma \xrightarrow{\tau}{}^{*}_{[\bar{c}]} s', \varsigma\\
&\implies\\
&(s.\mathsf{pcc} \subseteq \mathsf{dom}(\bar{c}.\mathcal{M}_c) \iff s'.\mathsf{pcc} \subseteq \mathsf{dom}(\bar{c}.\mathcal{M}_c))
\end{aligned}$$

Proof. Follows by Lemma 108, Claim 15 and corollary 2. □

Then, Lemma 109 states a restriction on the form of traces with respect to input actions $\dot{?}$ and output actions $\dot{!}$.

Lemma 109 (Traces consist of alternating input/output actions)**.**

$$\forall \bar{c}, \alpha.\ \alpha \in TR(\bar{c}) \implies \alpha \in \mathtt{Alt}\checkmark^*$$

Proof.

- Fix arbitrary $\bar{c}$ and α, and assume the antecedents.
- By unfolding the assumptions using Definition 72, we obtain (*):
$\exists \mathbb{C}, t', t : TargetSetup, s, s' : TargetState, \varsigma' : 2^{\mathbb{Z}}.$
$\mathbb{C} \ltimes \bar{c} = \lfloor t' \rfloor \wedge$
$t = (t'.\mathcal{M}_c, t'.\mathcal{M}_d + \omega, t'.imp, t'.\mathsf{mstc}, t'.\phi) \wedge$
$t \vdash_i s \wedge$
$s, \emptyset \xrightarrow{\alpha}_{[\bar{c}], \triangledown} s', \varsigma'$
- Our goal is:
$\alpha \in \mathtt{Alt}\checkmark^*$
- By inversion of the last conjunct of (*), we distinguish the following cases:

– **Case trace-steps-lambda:**
Here, we know $\lambda \neq \tau$.
And our goal becomes:
$\lambda \in \texttt{Alt}\checkmark^*$
This follows by regular language identities after unfolding Definition 69.

– **Case trace-steps-alternating:**
Here, we know (**):
$s, \varsigma \xrightarrow{\alpha'}_{[\overline{c}],\triangledown} s''', \varsigma'''$,
$s''', \varsigma''' \xrightarrow{\tau^*}{}^{*}_{[\overline{c}],\triangledown} s'', \varsigma''$,
$s'', \varsigma'' \xrightarrow{\lambda}_{[\overline{c}],\triangledown} s', \varsigma'$, and
$\lambda \neq \tau$
And by the induction hypothesis, we know (IH):
$\alpha' \in \texttt{Alt}\checkmark^*$
By instantiating Claim 6 using (**), we obtain (LAST-ACTION-OF-ALPHA'):
$\alpha' = \alpha''\lambda'$.
We prove our goal ($\alpha''\lambda'\lambda \in \texttt{Alt}\checkmark^*$) by distinguishing the following cases for λ:

* **Case $\lambda = \tau$:**
  By contradiction with (**), any goal is provable.
* **Case $\lambda = \checkmark$:**
  Here, our goal is immediate by regular language identities.
* **Case $\lambda \in \overset{\bullet}{?}$:**
  By regular language identities applied to our goal, it suffices to prove:
  $\alpha''\lambda'\lambda \in \texttt{Alt}$
  By applying Claim 5, we obtain the following subgoals:
  · $\alpha''\lambda' \in \texttt{Alt}$
    Immediate by (IH) after substitution using (LAST-ACTION-OF-ALPHA').
  · $\lambda' \in \overset{\bullet}{!}$
    Unfolding the case condition ($\alpha \in \overset{\bullet}{?}$), distinguish the following cases:
    1. **Case $\lambda = \texttt{call}(\_, \_)\_?\_, \_$:**
       Here, by inversion of (**) using cinvoke-context-to-compiled, we know:
       $s''.\mathsf{pcc} \notin \mathsf{dom}(\overline{c}.\mathcal{M}_c)$
       By instantiating Corollary 7 using (**) and the statement above, we know (S"'-PCC-OWNERSHIP):
       $s'''.\mathsf{pcc} \notin \mathsf{dom}(\overline{c}.\mathcal{M}_c)$
       And by instantiating Claim 8 using (**), after substitution using (LAST-ACTION-OF-ALPHA'), we obtain:
       $\_, \_ \xrightarrow{\lambda'}_{[\overline{c}],\triangledown} s''', \varsigma'''$
       By inversion of the latter statement, we get the following cases:
       (a) **Case $\lambda' = \tau$:** (short for the cases that produce τ)
           By instantiation of Claim 7 using (**), we know: $\lambda' \neq \tau$
           This contradicts the assumption $\lambda' = \tau$. So, our goal is provable.
       (b) **Case terminate-checkmark:**
           Here, we know $\lambda' = \checkmark$.
           Thus, after instantiating Claim 11 using α, we conclude using regular language identities that $\lambda = \checkmark$.
           This contradicts our case assumption $\lambda \in \overset{\bullet}{?}$. So, any goal is provable.

(c) **Case cinvoke-compiled-to-context, and**

(d) **Case creturn-to-context:**

Here, our goal ($\lambda' \in \dot{!}$) is immediate by the obtained preconditions.

(e) **Case cinvoke-context-to-compiled:**

Here, we know:

$mid \in \mathsf{dom}(\bar{c}.imp)$,

and by inversion of the preconditions using cinvoke-aux, we know:

$s'''.\mathsf{pcc} = \mathtt{inc}(s'''.imp(mid).\mathsf{pcc}, \_)$

Thus, by inversion of (*) using valid-linking and valid-program, we know:

$s'''.\mathsf{pcc} \subseteq \mathsf{dom}(\bar{c}.\mathcal{M}_c)$

This contradicts (S"'-PCC-OWNERSHIP). So, any goal is provable.

(f) **Case creturn-to-compiled:**

Here, we have:

$s'''.\mathsf{pcc} \subseteq \mathsf{dom}(\bar{c}.\mathcal{M}_c)$

This contradicts (S"'-PCC-OWNERSHIP). So, any goal is provable.

2. **Case $\lambda = \mathtt{ret?}\_, \_$:**

Here, by inversion of (**) using creturn-to-compiled, we know:

$s''.\mathsf{pcc} \not\subseteq \mathsf{dom}(\bar{c}.\mathcal{M}_c)$

The proof proceeds as in the previous case. We omit it for brevity.

* **Case $\lambda \in \dot{!}$:**

This is dual to **case** $\lambda \in \dot{?}$. We omit the proof for brevity.

- This concludes the proof of Lemma 109.

□

4.1 Soundness

To prove the soundness of trace equivalence, we define a ternary simulation relation on trace states. The simulation relation is called an **A**lternating **S**trong-**W**eak **S**imilarity (ASWS). ASWS is defined in terms of the strong and weak similarity relations that are given in Definition 86. The purpose of using ASWS is to show a *determinacy* result about the trace semantics. Determinacy is stated as a lemma about *three* executions, hence the ternary simulation relation.

Definition 74 (Alternating Strong-Weak Similarity (ASWS))**.**

$$ASWS(s_{12}, \varsigma_{12}, s_{11}, \varsigma_{11}, s_{22}, \varsigma_{22})_{\mathbb{C}_1, t_2, \alpha, i} \overset{\mathsf{def}}{=}$$

$$(\alpha(i) \in \dot{?} \ \vee \ s_{12}.\mathsf{pcc} \subseteq \mathsf{dom}(\mathbb{C}_1.\mathcal{M}_c)) \implies s_{12}, \varsigma_{12} \approx_{[\mathbb{C}_1]} s_{11}, \varsigma_{11} \ \wedge \ s_{12}, \varsigma_{12} \sim_{[t_2], \alpha, i} s_{22}, \varsigma_{22}$$

$$\wedge$$

$$(\alpha(i) \in \dot{!} \ \vee \ s_{12}.\mathsf{pcc} \not\subseteq \mathsf{dom}(\mathbb{C}_1.\mathcal{M}_c)) \implies s_{12}, \varsigma_{12} \sim_{[\mathbb{C}_1], \alpha, i} s_{11}, \varsigma_{11} \ \wedge \ s_{12}, \varsigma_{12} \approx_{[t_2]} s_{22}, \varsigma_{22}$$

where

$$s_1, \varsigma_1 \sim_{[t], \alpha, i} s_2, \varsigma_2 \ \overset{\mathsf{def}}{=} \ s_1, \varsigma_1 \sim_{[t], \rho_{[t]}(s_1, \varsigma_1)} s_2, \varsigma_2$$

Lemma 110 (Initial states are ASWS-related)**.**

$$\begin{aligned}
&\alpha \in Tr(\mathbb{C}_1[t_1]) \land \\
&\alpha \in Tr(\mathbb{C}_2[t_2]) \land \\
&s_{11} = \text{initial_state}(\mathbb{C}_1[t_1], \text{main_module}(\mathbb{C}_1[t_1])) \land \\
&s_{22} = \text{initial_state}(\mathbb{C}_2[t_2], \text{main_module}(\mathbb{C}_2[t_2])) \land \\
&s_{12} = \text{initial_state}(\mathbb{C}_1[t_2], \text{main_module}(\mathbb{C}_1[t_2])) \\
&\implies \\
&ASWS(s_{12}, \emptyset, s_{11}, \emptyset, s_{22}, \emptyset)_{\mathbb{C}_1, t_2, \alpha, 0}
\end{aligned}$$

Proof. (Sketch)
Follows from Lemma 135 and Lemma 136 (similar to the proof of Lemma 171). □

Lemma 111 (Two peripheral terminal states are ASWS-related to only a mixed state that is also terminal)**.**

$$\begin{aligned}
&ASWS(s_{12}, \varsigma_{12}, s_{11}, \varsigma_{11}, s_{22}, \varsigma_{22})_{_,_,_,_} \land \\
&\vdash_t s_{11} \land \\
&\vdash_t s_{22} \\
&\implies \\
&\vdash_t s_{12}
\end{aligned}$$

Proof.
Unfold Definition 74 then distinguish two cases:

- **Case** $s_{12}.\text{pcc} \subseteq \text{dom}(\mathbb{C}_1.\mathcal{M}_c)$**:**

  Here, instantiate Lemma 137 using assumption $\vdash_t s_{11}$ to obtain the goal.

- **Case** $s_{12}.\text{pcc} \not\subseteq \text{dom}(\mathbb{C}_1.\mathcal{M}_c)$**:**

  Here, instantiate Lemma 137 using assumption $\vdash_t s_{22}$ to obtain the goal.

□

Definition 75 (View change of a trace step)**.**

$$\begin{aligned}
\text{view_change}(a \ ? \ b) &\stackrel{\text{def}}{=} a \ ! \ b \\
\text{view_change}(a \ ! \ b) &\stackrel{\text{def}}{=} a \ ? \ b
\end{aligned}$$

Fact 1 (View change is an involution)**.**

$$\lambda \in \texttt{Alt} \implies \text{view\_change}(\text{view\_change}(\lambda)) = \lambda$$

Claim 18 (Existence of a view change of a trace step)**.**

$$\begin{aligned}
&\mathbb{C} \ltimes t \vdash_{border} \alpha[:i], s, \varsigma \land \\
&s, \varsigma \xrightarrow{\alpha(i)}_{[t]} s', \varsigma' \\
&\implies \\
&s, \varsigma \xrightarrow{\text{view_change}(\alpha(i))}_{[\mathbb{C}]} s', \varsigma'
\end{aligned}$$

Proof.
Follows from the bi-partition on the code memory of the linked program. □

Lemma 112 (ASWS satisfies the alternating simulation condition)**.**

$$
\begin{aligned}
&\alpha \in \mathtt{Alt} \ \wedge \\
&ASWS(s_{12}, \varsigma_{12}, s_{11}, \varsigma_{11}, s_{22}, \varsigma_{22})_{\mathbb{C}_1, t_2, \alpha, i} \ \wedge \\
&\mathbb{C}_1 \ltimes t_1 \vdash_{border} \alpha[: i], s_{11}, \varsigma_{11} \ \wedge \\
&\mathbb{C}_2 \ltimes t_2 \vdash_{border} \alpha[: i], s_{22}, \varsigma_{22} \ \wedge \\
&\mathbb{C}_1 \ltimes t_2 \vdash_{border} \alpha[: i], s_{12}, \varsigma_{12} \ \wedge \\
&s_{11}, \varsigma_{11} \xrightarrow{\alpha(i)}_{[t_1]} s'_{11}, \varsigma'_{11} \ \wedge \\
&s_{22}, \varsigma_{22} \xrightarrow{\alpha(i)}_{[t_2]} s'_{22}, \varsigma'_{22} \\
&\implies \\
&\exists s'_{12}, \varsigma'_{12}. \\
&s_{12}, \varsigma_{12} \xrightarrow{\alpha(i)}_{[t_2]} s'_{12}, \varsigma'_{12} \ \wedge \\
&ASWS(s'_{12}, \varsigma'_{12}, s'_{11}, \varsigma'_{11}, s'_{22}, \varsigma'_{22})_{\mathbb{C}_1, t_2, \alpha, i+1}
\end{aligned}
$$

Proof.
By $\alpha \in \mathtt{Alt}$ (unfolding Definition 69),
it suffices to distinguish the following two cases:

- **Case** $\alpha(i) \in \dot{!}$**:**

  Using the case condition together with the assumptions $(s_{11}, \varsigma_{11} \xrightarrow{\alpha(i)}_{[t_1]} s'_{11}, \varsigma'_{11})$ and $(\mathbb{C}_1 \ltimes t_1 \vdash_{border} \alpha[: i], s_{11}, \varsigma_{11})$, we instantiate Claim 18 to obtain:

  (s11-?-step): $s_{11}, \varsigma_{11} \xrightarrow{\mathsf{view\_change}(\alpha(i))}_{[\mathbb{C}_1]} s'_{11}, \varsigma'_{11}$

  By unfolding the assumption using Definition 74, we have:

  (STRONG-SIM-t2): $s_{12}, \varsigma_{12} \approx_{[t_2]} s_{22}, \varsigma_{22}$

  (WEAK-SIM-C1): $s_{12}, \varsigma_{12} \sim_{[\mathbb{C}_1], \alpha, i} s_{11}, \varsigma_{11}$

  Here, we can instantiate Lemma 149 (Weakening of strong similarity) using (STRONG-SIM-t2) and the given step $(s_{22}, \varsigma_{22} \xrightarrow{\alpha(i)}_{[t_2]} s'_{22}, \varsigma'_{22})$ to obtain:

  (G1): $\exists s'_{12}.\ s_{12}, \varsigma_{12} \xrightarrow{\alpha(i)}_{[t_2]} s'_{12}, \varsigma'_{22}$
  and
  (G2): $s'_{12}, \varsigma'_{22} \sim_{[t_2], \alpha, i+1} s'_{22}, \varsigma'_{22}$
  By instantiating Claim 18 using (G1) and the border-state invariant $(\mathbb{C}_1 \ltimes t_2 \vdash_{border} \alpha[: i], s_{12}, \varsigma_{12})$ from the assumptions, we obtain:

  (G1-?-step): $s_{12}, \varsigma_{12} \xrightarrow{\mathsf{view\_change}(\alpha(i))}_{[\mathbb{C}_1]} s'_{12}, \varsigma'_{22}$
  Thus, using (G1-?-step) together with (WEAK-SIM-C1) and (s11-?-step), we instantiate the strengthening lemma (Lemma 153) to obtain:

  (G3): $s'_{11}, \varsigma'_{11} \approx_{[\mathbb{C}_1]} s'_{12}, \varsigma'_{22}$
  After (G1), (G2) and (G3), no subgoals remain. So this concludes this case.

- **Case** $\alpha(i) \in \overset{\bullet}{?}$**:**

  By unfolding the assumption using Definition 74, we have:

  (STRONG-SIM-C1): $s_{12}, \varsigma_{12} \approx_{[\mathbb{C}_1]} s_{11}, \varsigma_{11}$

  (WEAK-SIM-t2): $s_{12}, \varsigma_{12} \sim_{[t_2],\alpha,i} s_{22}, \varsigma_{22}$

  Using the case condition together with the assumptions $(s_{11}, \varsigma_{11} \xrightarrow{\alpha(i)}_{[t_1]} s'_{11}, \varsigma'_{11})$ and $(\mathbb{C}_1 \ltimes t_1 \vdash_{border} \alpha[:\ i], s_{11}, \varsigma_{11})$, we instantiate Claim 18 to obtain:

  (s11-!-step): $s_{11}, \varsigma_{11} \xrightarrow{\mathsf{view\_change}(\alpha(i))}_{[\mathbb{C}_1]} s'_{11}, \varsigma'_{11}$

  Now we can instantiate Lemma 149 (Weakening of strong similarity) using (STRONG-SIM-C1) and (s11-!-step) to obtain:

  (G1): $\exists s'_{12}.\ s_{12}, \varsigma_{12} \xrightarrow{\mathsf{view\_change}(\alpha(i))}_{[\mathbb{C}_1]} s'_{12}, \varsigma'_{11}$
  and
  (G2): $s'_{12}, \varsigma'_{11} \sim_{[\mathbb{C}_1],\alpha,i+1} s'_{11}, \varsigma'_{11}$

  Now after obtaining s'_{12} from (G1) and using the assumption $(\mathbb{C}_1 \ltimes t_2 \vdash_{border} \alpha[:\ i], s_{12}, \varsigma_{12})$, we instantiate Claim 18 to obtain:

  (s12-?-step): $s_{12}, \varsigma_{12} \xrightarrow{\mathsf{view\_change}(\mathsf{view\_change}(\alpha(i)))}_{[t_2]} s'_{12}, \varsigma'_{11}$, which by rewriting using Fact 1 becomes:

  (s12-?-step): $s_{12}, \varsigma_{12} \xrightarrow{\alpha(i)}_{[t_2]} s'_{12}, \varsigma'_{11}$

  Now we use (s12-?-step) together with (WEAK-SIM-t2) and the given step $(s_{22}, \varsigma_{22} \xrightarrow{\alpha(i)}_{[t_2]} s'_{22}, \varsigma'_{22})$ to instantiate the strengthening lemma (Lemma 153) and obtain:

  (G3): $s'_{12}, \varsigma'_{11} \approx_{[t_2]} s'_{22}, \varsigma'_{22}$

  After (G1), (G2) and (G3), no subgoals remain. So this concludes this case.

This concludes the proof of Lemma 112.

□

Lemma 113 (ASWS satisfies the alternating simulation condition – whole trace)**.**

$$
\begin{aligned}
&\alpha \in \mathtt{Alt}\ \wedge \\
&ASWS(s_{12}, \varsigma_{12}, s_{11}, \varsigma_{11}, s_{22}, \varsigma_{22})_{\mathbb{C}_1, t_2, \alpha, 0}\ \wedge \\
&\mathbb{C}_1 \ltimes t_1 \vdash_{border} \alpha, s_{11}, \varsigma_{11}\ \wedge \\
&\mathbb{C}_2 \ltimes t_2 \vdash_{border} \alpha, s_{22}, \varsigma_{22}\ \wedge \\
&\mathbb{C}_1 \ltimes t_2 \vdash_{border} \alpha, s_{12}, \varsigma_{12}\ \wedge \\
&s_{11}, \varsigma_{11} \xrightarrow{\alpha}_{[t_1]} s'_{11}, \varsigma'_{11}\ \wedge \\
&s_{22}, \varsigma_{22} \xrightarrow{\alpha}_{[t_2]} s'_{22}, \varsigma'_{22} \\
&\ \Longrightarrow \\
&\exists s'_{12}, \varsigma'_{12}. \\
&s_{12}, \varsigma_{12} \xrightarrow{\alpha}_{[t_2]} s'_{12}, \varsigma'_{12}\ \wedge \\
&ASWS(s'_{12}, \varsigma'_{12}, s'_{11}, \varsigma'_{11}, s'_{22}, \varsigma'_{22})_{\mathbb{C}_1, t_2, \alpha, |\alpha|}
\end{aligned}
$$

Proof. (Sketch)
Follows by induction on the index of the ASWS relation from Lemma 112. □

Lemma 114 (Soundness of trace equivalence with respect to contextual equivalence)**.**

$$t_1 \overset{\mathrm{T}}{=}_{\omega,\nabla} t_2 \implies t_1 \simeq_{\omega,\nabla} t_2$$

Proof.
Equivalently, we prove the contra-positive, i.e., assuming (*):

$$t_1 \not\simeq_{\omega,\nabla} t_2$$

Our goal is now:

$$t_1 \overset{\mathrm{T}}{\neq}_{\omega,\nabla} t_2$$

Using (*) and by unfolding it using Definition 18, we know (without loss of generality) that:

$$\exists \mathbb{C}.\ \omega, \nabla \vdash \mathbb{C}[t_1] \Downarrow \ \wedge\ \omega, \nabla \nvdash \mathbb{C}[t_2] \Downarrow$$

By further unfolding using Definition 17, we know (**):

$$\begin{aligned}
&\exists \mathbb{C}, t_1'.\ \mathbb{C} \ltimes t_1 = \lfloor t_1' \rfloor \wedge \\
&\exists s_t.\ \mathsf{initial_state}(t_1' + \omega, \mathsf{main_module}(t_1')) \to_{\nabla}^{*} s_t \ \wedge\ \vdash_t s_t \\
&\wedge \\
&\forall t_2', s.\ \mathbb{C} \ltimes t_2 = \lfloor t_2' \rfloor \implies \\
&\mathsf{initial_state}(t_2' + \omega, \mathsf{main_module}(t_2')) \to_{\nabla}^{*} s \implies \nvdash_t s)
\end{aligned}$$

By unfolding our goal using Definition 73, our goal becomes:

$$TR_{\omega,\nabla}(t_1) \neq TR_{\omega,\nabla}(t_2)$$

For this, it suffices to prove (without loss of generality) that:

$$\exists \alpha.\ \alpha \in TR_{\omega,\nabla}(t_1) \ \wedge\ \alpha \notin TR_{\omega,\nabla}(t_2)$$

By unfolding using Definition 72, our goal becomes:

$$\begin{aligned}
&\exists \alpha, \mathbb{C}_1, t_1', s_1', \varsigma_1'. \\
&\mathbb{C}_1 \ltimes t_1 = \lfloor t_1' \rfloor \wedge \\
&\mathsf{initial_state}(t_1' + \omega, \mathsf{main_module}(t_1')), \emptyset \xrightarrow{\alpha}_{[t_1],\nabla} s_1', \varsigma_1' \wedge \\
&\forall \mathbb{C}_2, t_2'.\ \mathbb{C}_2 \ltimes t_2 = \lfloor t_2' \rfloor \implies \\
&\nexists s_2', \varsigma_2'.\ \mathsf{initial_state}(t_2' + \omega, \mathsf{main_module}(t_2')), \emptyset \xrightarrow{\alpha}_{[t_2],\nabla} s_2', \varsigma_2'
\end{aligned}$$

From (**), we obtain $\mathbb{C}$, t_1', and s_t. And by instantiating the $\implies$ direction of Corollary 5, we know (#1):

$$\exists \varsigma, \alpha.\ \mathsf{initial\_state}(t_1' + \omega, \mathsf{main\_module}(t_1')), \emptyset \xrightarrow{\alpha}{}^{*}_{[t_1],\nabla} s_t, \varsigma$$

By obtaining ς from (#1), and by using conjunct $\vdash_t s_t$ of (**) to instantiate rule terminate-checkmark, we know (#2):

$$s_t, \varsigma \xrightarrow{\checkmark}_{[t_1],\triangledown} s_t, \varsigma$$

Using (#1) and (#2), we instantiate rule trace-closure-trans to obtain ($t_1\checkmark$):

$$\mathsf{initial\_state}(t_1' + \omega, \mathsf{main\_module}(t_1')), \emptyset \xrightarrow{\alpha\checkmark}{}^{*}_{[t_1],\triangledown} s_t, \varsigma$$

To prove our existential goal, we pick $\alpha\checkmark|_{\not\tau}$ for α. We have to prove each of the following subgoals (conjuncts):

- **Subgoal** $\exists s_1', \varsigma_1'.\ \mathbb{C} \ltimes t_1 = \lfloor t_1' \rfloor\ \wedge\ \mathsf{initial\_state}(t_1' + \omega, \mathsf{main\_module}(t_1')), \emptyset \xrightarrow{\alpha\checkmark|_{\not\tau}}_{[t_1],\triangledown} s_1', \varsigma_1'$**:**
  Here, we apply Claim 6 obtaining the following subgoals:
  - $|\ \alpha\checkmark|_{\not\tau}\ | \geq 1$:
    Immediate because $\checkmark \neq \tau$.
  - $\mathsf{initial\_state}(t_1' + \omega, \mathsf{main\_module}(t_1')), \emptyset \xrightarrow{\alpha\checkmark}{}^{*}_{[t_1],\triangledown} s_t, \varsigma$
    Immediate by ($t_1\checkmark$).
- **Subgoal** $\forall \mathbb{C}_2, t_2'.\ \mathbb{C}_2 \ltimes t_2 = \lfloor t_2' \rfloor \implies$
  $\nexists s_2', \varsigma_2'.\ \mathsf{initial\_state}(t_2' + \omega, \mathsf{main\_module}(t_2')), \emptyset \xrightarrow{\alpha\checkmark|_{\not\tau}}_{[t_2],\triangledown} s_2', \varsigma_2'$**:**
  Pick arbitrary $\mathbb{C}_2, t_2'$ with $\mathbb{C}_2 \ltimes t_2 = \lfloor t_2' \rfloor$.
  Our goal is to show:
  $\nexists s_2', \varsigma_2'.\ \mathsf{initial\_state}(t_2' + \omega, \mathsf{main\_module}(t_2')), \emptyset \xrightarrow{\alpha\checkmark|_{\not\tau}}_{[t_2],\triangledown} s_2', \varsigma_2'$
  For the sake of contradiction, assume the contrary, i.e.:
  - Assume $\exists s_2', \varsigma_2'.\ \mathsf{initial\_state}(t_2' + \omega, \mathsf{main\_module}(t_2')), \emptyset \xrightarrow{\alpha\checkmark|_{\not\tau}}_{[t_2],\triangledown} s_2', \varsigma_2'$
  - By simplification of the restriction operator, we know:
    $\exists s_2', \varsigma_2'.\ \mathsf{initial\_state}(t_2' + \omega, \mathsf{main\_module}(t_2')), \emptyset \xrightarrow{\alpha|_{\not\tau}\checkmark}_{[t_2],\triangledown} s_2', \varsigma_2'$
  - Thus, by instantiating Claim 8, we know (TRACE-UNTIL-s2"):
    $\exists s_2', \varsigma_2', s_2'', \varsigma_2''.$
    $s_2'', \varsigma_2'' \xrightarrow{\checkmark}_{[t_2],\triangledown} s_2', \varsigma_2'\ \wedge$
    $\mathsf{initial\_state}(t_2' + \omega, \mathsf{main\_module}(t_2')), \emptyset \xrightarrow{\alpha|_{\not\tau}}{}^{*}_{[t_2],\triangledown} s_2'', \varsigma_2''$
  - By inversion of the first conjunct of (TRACE-UNTIL-s2") using terminate-checkmark, we know (TERMINAL-s2"):
    $\vdash_t s_2''$.
  - Similarly, we obtain from the previous (parallel) subgoal the state s_1'', ς_1'' where (TRACE-UNTIL-s1"):
    $s_1'', \varsigma_1'' \xrightarrow{\checkmark}_{[t_1],\triangledown} s_1', \varsigma_1'\ \wedge$
    $\mathsf{initial\_state}(t_1' + \omega, \mathsf{main\_module}(t_1')), \emptyset \xrightarrow{\alpha|_{\not\tau}}{}^{*}_{[t_1],\triangledown} s_1'', \varsigma_1''$
    and thus, we know (TERMINAL-s1"):
    $\vdash_t s_1''$

– Now, we instantiate Lemma 110 (Initial states are ASWS-related) to obtain (INIT-ASWS):

$$
\begin{aligned}
ASWS(&\mathsf{initial\_state}(\mathbb{C} \ltimes t_2 + \omega, \mathsf{main\_module}(\mathbb{C} \ltimes t_2)), \emptyset,\\
&\mathsf{initial\_state}(t_1' + \omega, \mathsf{main\_module}(t_1')), \emptyset,\\
&\mathsf{initial\_state}(t_2' + \omega, \mathsf{main\_module}(t_2')), \emptyset)_{\mathbb{C}, t_2, \alpha, 0}
\end{aligned}
$$

Now instantiate Lemma 113 (ASWS satisfies the alternating simulation condition – whole trace) using (TRACE-UNTIL-s2") and (TRACE-UNTIL-s1") to obtain $s''_{12}, \varsigma''_{12}$ satisfying:

(TRACE-UNTIL-s12"):

$$
\begin{aligned}
&\mathsf{initial\_state}(\mathbb{C} \ltimes t_2 + \omega, \mathsf{main\_module}(\mathbb{C} \ltimes t_2)), \emptyset \xrightarrow{\alpha}_{[t_2]} s''_{12}, \varsigma''_{12}\\
\wedge\ &ASWS(s''_{12}, \varsigma''_{12}, s''_1, \varsigma''_1, s''_2, \varsigma''_2)_{\mathbb{C}, t_2, \alpha, |\alpha|}
\end{aligned}
$$

Now instantiate Lemma 111 using (TERMINAL-s2") and (TERMINAL-s1") to obtain: (TERMINAL-s12"): $\vdash_t s''_{12}$

– Now instantiate Corollary 6 using (TRACE-UNTIL-s12") to obtain: (C-t2-STAR-STEPS-TO-s12"): $\mathsf{initial\_state}(\mathbb{C} \ltimes t_2 + \omega, \mathsf{main\_module}(\mathbb{C} \ltimes t_2)) \to^*_{\triangledown} s''_{12}$

– Now use (C-t2-STAR-STEPS-TO-s12") to instantiate the second conjunct of (**) and to immediately obtain a contradiction to (TERMINAL-s12").

This concludes the proof of the second subgoal.

This concludes the proof of Lemma 114.

□

5 A complete trace semantics for ImpMod

We give a sound and complete trace semantics for **ImpMod**. In this section, we prove completeness only (Lemma 117). Soundness, on the other hand, follows as an immediate corollary (Corollary 13) from results about the compiler of Section 3.

The syntax of the traces is exactly the same as in Section 4.
Figure 10 describes the trace semantics of **ImpMod**.

Definition 76 (Reflexive transitive closure of trace actions).
We write $s \xrightarrow{\alpha}{}^{*}_{[p],\nabla} s'$ where $\rightharpoonup^{*}_{[p],\nabla} \subseteq (\mathit{SourceState} \times 2^{\mathbb{Z}}) \times \overline{\Lambda} \times (\mathit{SourceState} \times 2^{\mathbb{Z}})$ to denote the reflexive transitive closure of the trace actions reduction relation $\rightharpoonup_{[p],\nabla} \subseteq (\mathit{SourceState} \times 2^{\mathbb{Z}}) \times \Lambda \times (\mathit{SourceState} \times 2^{\mathbb{Z}})$ where α collects the individual trace actions in succession.

$$\frac{}{s, \varsigma \xrightharpoonup{\epsilon}{}^{*}_{[p],\nabla} s, \varsigma}\ \textit{(trace-closure-refl-src)} \qquad \frac{s, \varsigma \xrightharpoonup{\alpha}{}^{*}_{[p],\nabla} s'', \varsigma'' \qquad s'', \varsigma'' \xrightharpoonup{\lambda}_{[p],\nabla} s', \varsigma'}{s, \varsigma \xrightharpoonup{\alpha\lambda}{}^{*}_{[p],\nabla} s', \varsigma'}\ \textit{(trace-closure-trans-src)}$$

where $\rightharpoonup_{[p],\nabla} \subseteq (\mathit{SourceState} \times 2^{\mathbb{Z}}) \times \Lambda \times (\mathit{SourceState} \times 2^{\mathbb{Z}})$ is as defined in Figure 10.

Definition 77 (Non-silent trace steps).
We write $s \xrightharpoondown{\alpha}_{[p],\nabla} s'$ where $\rightharpoondown_{[p],\nabla} \subseteq (\mathit{SourceState} \times 2^{\mathbb{Z}}) \times \overline{\Lambda} \times (\mathit{SourceState} \times 2^{\mathbb{Z}})$ to denote that execution on state s generates a sequence α of non-silent trace actions (i.e., excluding τ actions) and reaches state s'. We sometimes drop the parameter ∇ (which is the upper limit on memory allocation) for convenience.

$$\frac{s, \varsigma \xrightharpoonup{\tau^*}{}^{*}_{[p],\nabla} s'', \varsigma'' \qquad s'', \varsigma'' \xrightharpoonup{\lambda}_{[p],\nabla} s', \varsigma' \qquad \lambda \neq \tau}{s, \varsigma \xrightharpoondown{\lambda}_{[p],\nabla} s', \varsigma'}\ \textit{(trace-steps-lambda-src)}$$

$$\frac{s, \varsigma \xrightharpoondown{\alpha}_{[p],\nabla} s''', \varsigma''' \qquad s''', \varsigma''' \xrightharpoonup{\tau^*}{}^{*}_{[p],\nabla} s'', \varsigma'' \qquad s'', \varsigma'' \xrightharpoonup{\lambda}_{[p],\nabla} s', \varsigma' \qquad \lambda \neq \tau}{s, \varsigma \xrightharpoondown{\alpha\lambda}_{[p],\nabla} s', \varsigma'}\ \textit{(trace-steps-alternating-src)}$$

Claim 19 (A non-silent trace is not the empty string).

$$\begin{array}{l}\forall p, \alpha, s, \varsigma, s', \varsigma', \nabla. \\ s, \varsigma \xrightharpoondown{\alpha}_{[p],\nabla} s', \varsigma' \\ \implies \\ |\alpha| > 1\end{array}$$

Claim 20 ($\rightharpoondown$ eliminates τ actions).

$$\begin{array}{l}\forall p, \alpha, s, \varsigma, s', \varsigma', \nabla. \\ s, \varsigma \xrightharpoondown{\alpha\lambda}_{[p],\nabla} s', \varsigma' \\ \implies \\ \lambda \neq \tau\end{array}$$

Figure 10: Trace semantics for **ImpMod** for an arbitrary program p

(assign-silent-src)

$$\frac{\texttt{commands}(Fd(pc.fid))(pc.n) = \mathsf{Assign}\ e_l\ e_r \qquad \Sigma;\Delta;\beta;MVar;Fd \vdash \langle Mem, stk, pc, \Phi, nalloc\rangle \rightarrow_{\triangledown} s'}{\Sigma;\Delta;\beta;MVar;Fd \vdash \langle Mem, stk, pc, \Phi, nalloc\rangle, \varsigma \xrightarrow{\tau}_{[p],\triangledown} s', \varsigma}$$

(alloc-silent-src)

$$\frac{\texttt{commands}(Fd(pc.fid))(pc.n) = \mathsf{Alloc}\ e_l\ e_{size} \qquad \Sigma;\Delta;\beta;MVar;Fd \vdash \langle Mem, stk, pc, \Phi, nalloc\rangle \rightarrow_{\triangledown} s'}{\Sigma;\Delta;\beta;MVar;Fd \vdash \langle Mem, stk, pc, \Phi, nalloc\rangle, \varsigma \xrightarrow{\tau}_{[p],\triangledown} s', \varsigma}$$

(jump-silent-src)

$$\frac{\texttt{commands}(Fd(pc.fid))(pc.n) = \mathsf{JumpIfZero}\ e_c\ e_{off} \qquad \Sigma;\Delta;\beta;MVar;Fd \vdash \langle Mem, stk, pc, \Phi, nalloc\rangle \rightarrow_{\triangledown} s'}{\Sigma;\Delta;\beta;MVar;Fd \vdash \langle Mem, stk, pc, \Phi, nalloc\rangle, \varsigma \xrightarrow{\tau}_{[p],\triangledown} s', \varsigma}$$

(cinvoke-silent-program-src)

$$\frac{\begin{array}{c}\texttt{commands}(Fd(pc.fid))(pc.n) = \mathsf{Call}\ fid_{call}\ \overline{e} \qquad \Sigma;\Delta;\beta;MVar;Fd \vdash \langle Mem, stk, pc, \Phi, nalloc\rangle \rightarrow_{\triangledown} s' \\ \texttt{moduleID}(Fd(pc.fid)) \in \texttt{moduleIDs}(p) \qquad \texttt{moduleID}(Fd(fid_{call})) \in \texttt{moduleIDs}(p)\end{array}}{\Sigma;\Delta;\beta;MVar;Fd \vdash \langle Mem, stk, pc, \Phi, nalloc\rangle, \varsigma \xrightarrow{\tau}_{[p],\triangledown} s', \varsigma}$$

(cinvoke-silent-context-src)

$$\frac{\begin{array}{c}\texttt{commands}(Fd(pc.fid))(pc.n) = \mathsf{Call}\ fid_{call}\ \overline{e} \qquad \Sigma;\Delta;\beta;MVar;Fd \vdash \langle Mem, stk, pc, \Phi, nalloc\rangle \rightarrow_{\triangledown} s' \\ \texttt{moduleID}(Fd(pc.fid)) \notin \texttt{moduleIDs}(p) \qquad \texttt{moduleID}(Fd(fid_{call})) \notin \texttt{moduleIDs}(p)\end{array}}{\Sigma;\Delta;\beta;MVar;Fd \vdash \langle Mem, stk, pc, \Phi, nalloc\rangle, \varsigma \xrightarrow{\tau}_{[p],\triangledown} s', \varsigma}$$

(cinvoke-context-to-program-src)

$$\frac{\begin{array}{c} s = \langle Mem, stk, pc, \Phi, nalloc\rangle \\ s' = \langle Mem', stk', pc', \Phi', nalloc'\rangle \qquad \Sigma;\Delta;\beta;MVar;Fd \vdash s \succ_{\approx} s' \\ s_{\perp} = \langle Mem', stk', \perp, \Phi', nalloc'\rangle \\ \Sigma;\Delta;\beta;MVar;Fd \vdash s \not\rightarrow_{\triangledown} s' \implies s'' = s_{\perp} \qquad \Sigma;\Delta;\beta;MVar;Fd \vdash s \rightarrow_{\triangledown} s' \implies s'' = s' \\ \texttt{commands}(Fd(pc.fid))(pc.n) = \mathsf{Call}\ fid_{call}\ \overline{e} \qquad \overline{v} = [i \mapsto v_i \mid\ i \in [0, |\overline{e}|) \wedge \overline{e}(i), \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow v_i] \\ r = \mathsf{reachable\_addresses}(\{\overline{v}(i) \mid i \in [0, |\overline{e}|) \wedge \overline{v}(i) = (\delta, \_, \_, \_)\}, Mem') \\ \varsigma' = \mathsf{reachable\_addresses\_closure}(\varsigma \cup r, Mem') \\ \texttt{moduleID}(Fd(pc.fid)) \notin \texttt{moduleIDs}(p) \qquad \texttt{moduleID}(Fd(fid_{call})) \in \texttt{moduleIDs}(p)\end{array}}{\Sigma;\Delta;\beta;MVar;Fd \vdash s, \varsigma \xrightarrow{\texttt{call}(\texttt{moduleID}(Fd(fid_{call})), fid_{call})\overline{v}?Mem'|_{\varsigma'}, nalloc'}_{[p],\triangledown} s'', \varsigma'}$$

(cinvoke-program-to-context-src)

$$\frac{\begin{array}{c} s = \langle Mem, stk, pc, \Phi, nalloc\rangle \\ s' = \langle Mem', stk', pc', \Phi', nalloc'\rangle \qquad \Sigma;\Delta;\beta;MVar;Fd \vdash s \succ_{\approx} s' \\ s_{\perp} = \langle Mem', stk', \perp, \Phi', nalloc'\rangle \\ \Sigma;\Delta;\beta;MVar;Fd \vdash s \not\rightarrow_{\triangledown} s' \implies s'' = s_{\perp} \qquad \Sigma;\Delta;\beta;MVar;Fd \vdash s \rightarrow_{\triangledown} s' \implies s'' = s' \\ \texttt{commands}(Fd(pc.fid))(pc.n) = \mathsf{Call}\ fid_{call}\ \overline{e} \qquad \overline{v} = [i \mapsto v_i \mid\ i \in [0, |\overline{e}|) \wedge \overline{e}(i), \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow v_i] \\ r = \mathsf{reachable\_addresses}(\{\overline{v}(i) \mid i \in [0, |\overline{e}|) \wedge \overline{v}(i) = (\delta, \_, \_, \_)\}, Mem') \\ \varsigma' = \mathsf{reachable\_addresses\_closure}(\varsigma \cup r, Mem') \\ \texttt{moduleID}(Fd(pc.fid)) \in \texttt{moduleIDs}(p) \qquad \texttt{moduleID}(Fd(fid_{call})) \notin \texttt{moduleIDs}(p)\end{array}}{\Sigma;\Delta;\beta;MVar;Fd \vdash s, \varsigma \xrightarrow{\texttt{call}(\texttt{moduleID}(Fd(fid_{call})), fid_{call})\overline{v}?Mem'|_{\varsigma'}, nalloc'}_{[p],\triangledown} s'', \varsigma'}$$

(creturn-silent-program-src)

$$\frac{\begin{array}{c}\texttt{commands}(Fd(pc.fid))(pc.n) = \mathsf{Return} \qquad s = \langle Mem, stk, pc, \Phi, nalloc\rangle \\ s' = \langle Mem', stk', pc', \Phi', nalloc'\rangle \\ \Sigma;\Delta;\beta;MVar;Fd \vdash s \rightarrow_{\triangledown} s' \qquad \texttt{moduleID}(Fd(pc.fid)) \in \texttt{moduleIDs}(p) \\ \texttt{moduleID}(Fd(pc'.fid)) \in \texttt{moduleIDs}(p)\end{array}}{\Sigma;\Delta;\beta;MVar;Fd \vdash s, \varsigma \xrightarrow{\tau}_{[p],\triangledown} s', \varsigma}$$

Figure 10 (Cont.): Trace semantics for **ImpMod** for an arbitrary program p

(creturn-silent-context-src)

$$\frac{\begin{array}{c}\texttt{commands}(Fd(pc.fid))(pc.n) = \mathsf{Return} \qquad s = \langle Mem, stk, pc, \Phi, nalloc\rangle \\ s' = \langle Mem', stk', pc', \Phi', nalloc'\rangle \\ \Sigma; \Delta; \beta; MVar; Fd \vdash s \to_{\nabla} s' \qquad \texttt{moduleID}(Fd(pc.fid)) \notin \texttt{moduleIDs}(p) \\ \texttt{moduleID}(Fd(pc'.fid)) \notin \texttt{moduleIDs}(p)\end{array}}{\Sigma; \Delta; \beta; MVar; Fd \vdash s, \varsigma \xrightarrow{\tau}_{[p],\nabla} s', \varsigma}$$

(creturn-to-program-src)

$$\frac{\begin{array}{c}\texttt{commands}(Fd(pc.fid))(pc.n) = \mathsf{Return} \qquad s = \langle Mem, stk, pc, \Phi, nalloc\rangle \\ s' = \langle Mem', stk', pc', \Phi', nalloc\rangle \\ \Sigma; \Delta; \beta; MVar; Fd \vdash s \to_{\nabla} s' \qquad \texttt{moduleID}(Fd(pc.fid)) \notin \texttt{moduleIDs}(p) \\ \texttt{moduleID}(Fd(pc'.fid)) \in \texttt{moduleIDs}(p) \qquad \varsigma' = \mathsf{reachable\_addresses\_closure}(\varsigma, Mem')\end{array}}{\Sigma; \Delta; \beta; MVar; Fd \vdash s, \varsigma \xrightarrow{\texttt{ret}?Mem'|_{\varsigma'}, nalloc}_{[p],\nabla} s', \varsigma'}$$

(creturn-to-context-src)

$$\frac{\begin{array}{c}\texttt{commands}(Fd(pc.fid))(pc.n) = \mathsf{Return} \qquad s = \langle Mem, stk, pc, \Phi, nalloc\rangle \\ s' = \langle Mem', stk', pc', \Phi', nalloc\rangle \\ \Sigma; \Delta; \beta; MVar; Fd \vdash s \to_{\nabla} s' \qquad \texttt{moduleID}(Fd(pc.fid)) \in \texttt{moduleIDs}(p) \\ \texttt{moduleID}(Fd(pc'.fid)) \notin \texttt{moduleIDs}(p) \qquad \varsigma' = \mathsf{reachable\_addresses\_closure}(\varsigma, Mem')\end{array}}{\Sigma; \Delta; \beta; MVar; Fd \vdash s, \varsigma \xrightarrow{\texttt{ret}?Mem'|_{\varsigma'}, nalloc}_{[p],\nabla} s', \varsigma'}$$

(terminate-checkmark-src)

$$\frac{\_ \vdash_t s}{\Sigma; \Delta; \beta; MVar; Fd \vdash s, \varsigma \xrightarrow{\checkmark}_{[p],\nabla} s, \varsigma}$$

Claim 21 ($\rightharpoonup\!\!\!\rightharpoonup$ is supported by $\rightharpoonup$)**.**

$$\begin{array}{l}\forall p, \alpha, \lambda, s, \varsigma, s', \varsigma', \nabla. \\ s, \varsigma \xrightarrow{\alpha\lambda}_{[p],\nabla} s', \varsigma' \\ \implies \\ \exists s'', \varsigma''. \\ s'', \varsigma'' \xrightarrow{\lambda}_{[p],\nabla} s', \varsigma' \ \wedge \\ s, \varsigma \xrightarrow{\alpha}{}^{*}_{[p],\nabla} s'', \varsigma''\end{array}$$

Claim 22 ($\rightharpoonup\!\!\!\rightharpoonup$ decomposes)**.**

$$\begin{array}{l}\forall p, \alpha_1, \alpha_2, s, \varsigma, s', \varsigma', \nabla. \\ s, \varsigma \xrightarrow{\alpha_1\alpha_2}_{[p],\nabla} s', \varsigma' \\ \implies \\ \exists s_1, \varsigma_1. \\ s, \varsigma \xrightarrow{\alpha_1}_{[p],\nabla} s_1, \varsigma_1 \ \wedge \\ s_1, \varsigma_1 \xrightarrow{\alpha_2}_{[p],\nabla} s', \varsigma' \ \wedge\end{array}$$

Claim 23 (Non-silent part of $\rightharpoonup^*$ is supported by $\rightharpoonup\!\!\!\!\rightharpoonup$)**.**

$$\begin{array}{l}\forall p, \alpha, s, \varsigma, s', \varsigma', \nabla. \\ \mid \alpha|_{\not\tau} \mid \geq 1 \;\wedge \\ s, \varsigma \xrightharpoonup{\alpha}{}^{*}_{[p],\nabla} s', \varsigma' \\ \Longrightarrow \\ \exists s'', \varsigma''.\ s, \varsigma \xrightharpoonup{\alpha|_{\not\tau}}{}_{[p],\nabla} s'', \varsigma''\end{array}$$

For a program p, we define the set $TR(p) \subseteq \Lambda^+$ of finite non-empty prefixes of p's possible execution traces as follows:

Definition 78 (A prefix of an execution trace is possible for a component)**.**
A finite prefix α belonging to a component p's set $TR_{\nabla,\Delta,\Sigma,\beta}(p)$ of possible execution trace prefixes is defined as:

$$\begin{array}{l}\alpha \in TR_{\omega,\nabla,\Delta,\Sigma,\beta}(p) \\ \iff \\ \exists \mathbb{C}, \overline{m}, s', \varsigma', \Delta_{\mathbb{C}}, \Sigma_{\mathbb{C}}, \beta_{\mathbb{C}}. \\ \Delta' = \Delta \uplus \Delta_{\mathbb{C}} \;\wedge\; \Sigma' = \Sigma \uplus \Sigma_{\mathbb{C}} \;\wedge\; \beta' = \beta \cup \beta_{\mathbb{C}} \;\wedge \\ \mathbb{C}[p]_{\Delta',\Sigma'} = \overline{m} \;\wedge \\ \Sigma'; \Delta' + \omega; \beta'; \mathsf{mvar}(\overline{m}); \mathsf{fd\_map}(\overline{m}) \vdash \mathsf{initial\_state}(\overline{m}, \Delta' + \omega, \Sigma', \mathsf{main\_module}(\overline{m})), \emptyset \xrightharpoonup{\alpha}{}_{[p],\nabla} s', \varsigma'\end{array}$$

where $\rightharpoonup\!\!\!\!\rightharpoonup_{[p],\nabla} \subseteq (\mathit{SourceState} \times 2^{\mathbb{Z}}) \times \overline{\Lambda} \times (\mathit{SourceState} \times 2^{\mathbb{Z}})$ *is as defined in Definition 77.*

Definition 79 (Trace equivalence)**.**

$$\beta_1, p_1 \stackrel{\mathrm{T}}{=}_{\omega,\nabla,\Delta,\Sigma} \beta_2, p_2 \stackrel{\mathsf{def}}{=} TR_{\omega,\nabla,\Delta,\Sigma,\beta_1}(p_1) = TR_{\omega,\nabla,\Delta,\Sigma,\beta_2}(p_2)$$

Claim 24 (Termination markers appear only at the end of an execution trace)**.**

$$\forall p.\ \alpha \in TR(p) \implies \alpha \in (\Lambda \setminus \{\checkmark\})^* \vee \alpha \in (\Lambda \setminus \{\checkmark\})^* \checkmark$$

Claim 25 (Prefix-closure of trace set membership)**.**

$$\forall p, \alpha.\ \alpha \in TR(\bar{c}) \implies (\forall \alpha'.\ \alpha = \alpha' \alpha'' \implies \alpha' \in TR(p))$$

Proof.
Follows from Claim 22. Instantiate " $\implies$ " direction of Definition 78 using the assumption, and apply its " $\impliedby$ " direction to the goal. ∎

Claim 26 (A state that is reachable by $\rightarrow$ reduction or by $\succ_{\approx}$ is also reachable by $\rightharpoonup$)**.**

$$\begin{array}{l}\forall p, s, s', \varsigma, \nabla. \\ (s \rightarrow_{\nabla} s' \vee s \succ_{\approx} s') \\ \Longrightarrow \\ \exists \lambda, \varsigma'.\ s, \varsigma \xrightharpoonup{\lambda}{}_{[p],\nabla} s', \varsigma'\end{array}$$

Claim 27 (A non-$\bot$ state that is reachable by $\rightharpoonup$ is also reachable by $\rightarrow$ reduction)**.**

$$\begin{aligned}&\forall t, p, s, s', \varsigma, \varsigma'.\\&s'.pc \neq \bot\ \wedge\\&s, \varsigma \xrightharpoonup{\lambda}_{[p],\nabla} s', \varsigma'\\&\Longrightarrow\\&s \rightarrow_{\nabla} s'\end{aligned}$$

Claim 28 (Silent trace steps correspond to $\rightarrow$ steps)**.**

$$\begin{aligned}&\forall p, s, s', \varsigma, \varsigma', \nabla.\\&s, \varsigma \xrightharpoonup{\tau^*}{}^{*}_{[p],\nabla} s', \varsigma'\\&\Longrightarrow\\&s \rightarrow^*_{\nabla} s'\end{aligned}$$

Claim 29 (Non-stuck trace steps correspond to $\rightarrow$ execution steps)**.**

$$\begin{aligned}&\forall p, s, s', s'', \varsigma, \varsigma', \varsigma'', \nabla.\\&s, \varsigma \xrightharpoonup{\alpha}{}^{*}_{[p],\nabla} s', \varsigma'\ \wedge\\&s', \varsigma' \xrightharpoonup{\lambda}_{[p],\nabla} s'', \varsigma''\\&\Longrightarrow\\&s \rightarrow^*_{\nabla} s'\end{aligned}$$

Claim 30 (The set of shared addresses ς does not change by silent trace steps)**.**

$$\begin{aligned}&\forall s, s', \varsigma, \varsigma', \nabla.\\&s, \varsigma \xrightharpoonup{\tau^*}{}^{*}_{[p],\nabla} s', \varsigma'\\&\Longrightarrow\\&\varsigma = \varsigma'\end{aligned}$$

Corollary 8 (Reachability by $\rightarrow^*$ implies reachability by $\rightharpoonup^*$)**.**

$$\begin{aligned}&\mathsf{initial\_state}(\overline{C} \uplus p, \Delta, \Sigma, \mathsf{main\_module}(\overline{C} \uplus p)) \rightarrow^*_{\nabla} s\\&\Longrightarrow\\&\exists \varsigma, \alpha.\ \mathsf{initial\_state}(\overline{C} \uplus p, \Delta, \Sigma, \mathsf{main\_module}(\overline{C} \uplus p)), \emptyset \xrightharpoonup{\alpha}{}^{*}_{[p],\nabla} s, \varsigma\end{aligned}$$

Corollary 9 (Reachability by $\rightharpoonup^*$ implies reachability by $\rightarrow^*$ when the state is non-$\bot$)**.**

$$\begin{aligned}&\mathsf{initial\_state}(\overline{C} \uplus p, \Delta, \Sigma, \mathsf{main\_module}(\overline{C} \uplus p)), \emptyset \xrightharpoonup{\alpha}{}^{*}_{[p],\nabla} s, \varsigma\ \wedge\\&s.pc \neq \bot\\&\Longrightarrow\\&\mathsf{initial\_state}(\overline{C} \uplus p, \Delta, \Sigma, \mathsf{main\_module}(\overline{C} \uplus p)) \rightarrow^*_{\nabla} s\end{aligned}$$

Lemma 115 (Non-communication actions do not change context/compiled component's ownership of pc)**.**

$$\begin{aligned}&K_{mod}; K_{fun}; \mathbb{C} \uplus p; \Sigma; \Delta; \beta; MVar; Fd \vdash_{exec} s\ \wedge\\&s \xrightharpoonup{\tau}_{[p]} s'\\&\Longrightarrow\\&(\mathtt{moduleID}(Fd(s.pc.fid)) \in \mathtt{moduleIDs}(p) \iff \mathtt{moduleID}(Fd(s'.pc.fid)) \in \mathtt{moduleIDs}(p))\end{aligned}$$

Proof. Similar to the proof of Lemma 108. □

Corollary 10 (Non-communication actions do not change ownership of pc (star-closure))**.**

$$
\begin{aligned}
&K_{mod}; K_{fun}; \mathbb{C} \uplus p; \Sigma; \Delta; \beta; MVar; Fd \vdash_{exec} s \land \\
&s, \varsigma \xrightarrow{\tau}{}^{*}_{[p]} s', \varsigma \\
&\implies \\
&(\mathtt{moduleID}(Fd(s.pc.fid)) \in \mathtt{moduleIDs}(p) \iff \mathtt{moduleID}(Fd(s'.pc.fid)) \in \mathtt{moduleIDs}(p))
\end{aligned}
$$

Proof. Follows by Lemma 115, Claim 28 and corollary 4. □

Then, Lemma 116 states a restriction on the form of traces with respect to input actions $\overset{\bullet}{?}$ and output actions $\overset{\bullet}{!}$.

Lemma 116 (Traces consist of alternating input/output actions)**.**

$$\forall p, \alpha.\ \alpha \in TR(p) \implies \alpha \in \mathtt{Alt}\checkmark^{*}$$

Proof.
Similar to the proof of Lemma 109. □

5.1 Completeness using back-translation

Lemma 117 (Completeness of trace equivalence with respect to contextual equivalence)**.**

$$
\begin{aligned}
&\forall \overline{m_1}, \overline{m_2}, \tilde{\Delta}, \beta_1, \beta_2, \tilde{\Sigma}, \nabla. \\
&\mathsf{dom}(\tilde{\Sigma}) = \{\mathtt{moduleID}(m) \mid m \in \overline{m_1}\} = \{\mathtt{moduleID}(m) \mid m \in \overline{m_2}\} \land \\
&\mathsf{dom}(\tilde{\Delta}) = \{\mathtt{moduleID}(m) \mid m \in \overline{m_1}\} = \{\mathtt{moduleID}(m) \mid m \in \overline{m_2}\} \land \\
&\tilde{\Delta}, \beta_1, \overline{m_1}\ \simeq_{\tilde{\Sigma},\omega,\nabla}\ \tilde{\Delta}, \beta_2, \overline{m_2} \\
&\implies \\
&\exists \Delta, \Sigma.\ \beta_1, \overline{m_1}\ \overset{\mathrm{T}}{=}_{\omega,\nabla,\Delta,\Sigma}\ \beta_2, \overline{m_2}
\end{aligned}
$$

(Proof Sketch):
The proof of this lemma is similar to the correctness of the back-translation given by Lemma 168, and additionally relies on Lemma 119.

We omit the details to avoid repetition. The crucial difference is that back-translation is defined for the common prefix of two traces as follows: Back-translation is a function (denoted by $\langle\!\langle \cdot, \cdot \rangle\!\rangle$) that takes as input two traces α_1, α_2 of respectively two programs, c_1 and c_2, and produces a source (partial) program c which is a distinguishing context. A distinguishing context satisfies **either**:

- when c is linked with c_1, it constitutes a *converging* program, and when it is linked with c_2, it constitutes a *diverging* program, **or**
- when c is linked with c_1, it constitutes a *diverging* program, and when it is linked with c_2, it constitutes a *converging* program.

Definition 80 (Distinguishing snippet for equi-flow trace actions).

$$\mathsf{distinguishArgs} : \mathcal{E} \to \mathcal{V} \to \mathcal{V} \to \overline{Cmd}$$

$$\begin{aligned}
&\mathsf{distinguishArgs}(e, v_1, v_2) \stackrel{\mathsf{def}}{=} \\
&\mathsf{capType}(v_1) \neq \mathsf{capType}(v_2) \implies \textit{ifnotzero-then-else}(e - \mathsf{capType}(v_1), \mathsf{converge}, \mathsf{diverge}) \\
&\mathsf{capType}(v_1) = \mathsf{capType}(v_2) = \mathsf{INTEGER} \implies \textit{ifnotzero-then-else}(e - v_1, \mathsf{converge}, \mathsf{diverge}) \\
&\mathsf{capStart}(v_1) \neq \mathsf{capStart}(v_2) \implies \textit{ifnotzero-then-else}(\mathsf{start}(e) - \mathsf{capStart}(v_1), \mathsf{converge}, \mathsf{diverge}) \\
&\mathsf{capEnd}(v_1) \neq \mathsf{capEnd}(v_2) \implies \textit{ifnotzero-then-else}(\mathsf{end}(e) - \mathsf{capEnd}(v_1), \mathsf{converge}, \mathsf{diverge}) \\
&\mathsf{capOff}(v_1) \neq \mathsf{capOff}(v_2) \implies \textit{ifnotzero-then-else}(\mathsf{off}(e) - \mathsf{capOff}(v_1), \mathsf{converge}, \mathsf{diverge})
\end{aligned}$$

Lemma 118 (Value cross-relatedness on integers is compatible with **ImpMod** subtraction).

$$\begin{aligned}
&\forall v_t, v_s, v_1, v_2, s. \\
&v_1 \cong v_t \ \wedge \ v_2 \cong v_t \ \wedge \ v_1 - v_2, _, _, _, _, _, _, _, _ \Downarrow v_s \implies v_s = 0
\end{aligned}$$

Proof. Follows from Definition 60 and rule Evaluate-expr-binop. ☐

Lemma 119 (If two target values are unequal, then distinguishArgs produces code that terminates on exactly one of them).

$$\begin{aligned}
&\forall \Sigma; \Delta; \beta; MVar; Fd, s, e, v_1, v_2. \\
&\mathsf{upcoming_commands}(s, \mathsf{distinguishArgs}(e, v_1, v_2)) \ \wedge \\
&v_1 \neq v_2 \ \wedge \\
&\exists v.\ e, \Sigma; \Delta; \beta; MVar; Fd, s.Mem, s.\Phi, s.pc \Downarrow v \\
&\implies \\
&(v \cong v_2 \implies \exists s_t.\ \Sigma; \Delta; \beta; MVar; Fd \vdash s \to^* s_t \ \wedge \ \vdash_t s_t) \ \wedge \\
&(v \cong v_1 \implies \nexists s_t.\ \Sigma; \Delta; \beta; MVar; Fd \vdash s \to^* s_t \ \wedge \ \vdash_t s_t)
\end{aligned}$$

Proof. Follows by easy case distinction after unfolding Definition 80 from Lemmas 118, 159, 161 and 162. ☐

6 Security guarantee about the compiler: full abstraction

To be convinced about the security of the compiler, we need:

1. a property for a compiler that captures security (for that, we use Definition 81 of full abstraction of a compiler),
2. and a proof that our compiler satisfies this property (Theorem 2).

To express compiler security, one de-facto standard exists: compiler full abstraction [5]. Informally, a compiler is fully abstract if the compilation from source programs to target programs preserves and reflects contextual/behavioral equivalence. In other words, a compiler is fully-abstract if for any two source programs $\overline{m}_1$ and $\overline{m}_2$ and in any possible execution environment, we have that they are behaviorally equivalent ($\overline{m}_1 \simeq \overline{m}_2$) if and only if their compiled counterparts are behaviorally equivalent ($[\![\overline{m}_1]\!] \simeq [\![\overline{m}_2]\!]$). The notion of behavioral equivalence used here is the canonical notion of contextual equivalence: two terms are equivalent if they behave the same when plugged into any valid context.

Source and target contextual equivalence can be stated as in Definitions 18 and 45.

This definition is standard and used by most papers in the literature on secure compilation [6–14].

We say a compiler $[\![\cdot]\!]$ is fully abstract if in all execution environments, it preserves and reflects contextual equivalence. An execution environment determines (1) the stack region $\tilde{\Sigma}(\texttt{moduleID}(m))$ that is allocated for a module m of the compiled program together with (2) the start address ω of the data segment of the compiled program, and (3) the limit ∇ on dynamic memory allocation. So, effectively, full abstraction **requires that for any fixed**: (1) the *stack size* allocated to any of the program's modules (i.e., whether sufficient or not), (2) the *offset in memory* in which a program's data segment lives, and (3) the *heap space* available for dynamic allocation (i.e., whether sufficient or not), the compiler should preserve and reflect the contextual equivalence of the source language programs. Thus, full abstraction of a compiler $[\![\cdot]\!]$ denoted $\mathsf{FA}([\![\cdot]\!])$ is defined as follows.

Definition 81 (Compiler full abstraction)**.**

$$
\begin{aligned}
\mathsf{FA}([\![\cdot]\!]) \overset{\mathsf{def}}{=}\ & \forall \overline{m_1}, \overline{m_2}, \tilde{\Delta}, \beta_1, \beta_2, K_{mod1}, K_{fun1}, K_{mod2}, K_{fun2}, \tilde{\Sigma}, \nabla < -1, t_1, t_2.\\
& \mathsf{dom}(\tilde{\Sigma}) = \{\texttt{moduleID}(m) \mid m \in \overline{m_1}\} = \{\texttt{moduleID}(m) \mid m \in \overline{m_2}\} \wedge\\
& \mathsf{dom}(\tilde{\Delta}) = \{\texttt{moduleID}(m) \mid m \in \overline{m_1}\} = \{\texttt{moduleID}(m) \mid m \in \overline{m_2}\} \wedge\\
& \overline{m_2} \frown \overline{m_2} \wedge\\
& [\![\overline{m_1}]\!]_{\tilde{\Delta}, \tilde{\Sigma}, \beta_1, K_{mod1}, K_{fun1}} = t_1 \wedge\\
& [\![\overline{m_2}]\!]_{\tilde{\Delta}, \tilde{\Sigma}, \beta_2, K_{mod2}, K_{fun2}} = t_2\\
& \implies\\
& \tilde{\Delta}, \beta_1, \overline{m_1} \simeq_{\tilde{\Sigma}, \omega, \nabla} \tilde{\Delta}, \beta_2, \overline{m_2} \iff t_1 \simeq_{\omega, \nabla} t_2
\end{aligned}
$$

Compiler full abstraction can be stated as follows:

Theorem 2 ($[\![\cdot]\!]$ is fully abstract)**.** *$[\![\cdot]\!] \in FA$ where $[\![\cdot]\!]$ is our compiler that is defined in rule Module-list-translation.*

Proof.
Immediate by Lemmas 120 and 121. □

Referring to Definition 81 of a translation being fully abstract, we call the $\implies$ direction of the logical equivalence "*preservation of contextual equivalence*" (Lemma 121), and the other direction $\impliedby$ "*reflection of contextual equivalence*" (Lemma 120).

The proof of Lemma 120 is easy given the correctness and compositionality results we proved in Section 3.

Lemma 120 ($[\![\cdot]\!]$ reflects contextual equivalence)**.**

$$\begin{aligned}
&\forall \overline{m_1}, \overline{m_2}, \tilde{\Delta}, \beta_1, \beta_2, K_{mod1}, K_{fun1}, K_{mod2}, K_{fun2}, \tilde{\Sigma}, \omega, \nabla.\\
&\mathsf{dom}(\tilde{\Sigma}) = \{\mathtt{moduleID}(m) \mid m \in \overline{m_1}\} = \{\mathtt{moduleID}(m) \mid m \in \overline{m_2}\} \wedge\\
&\mathsf{dom}(\tilde{\Delta}) = \{\mathtt{moduleID}(m) \mid m \in \overline{m_1}\} = \{\mathtt{moduleID}(m) \mid m \in \overline{m_2}\} \wedge\\
&\exists t_1.\ [\![\overline{m_1}]\!]_{\tilde{\Delta},\tilde{\Sigma},\beta_1,K_{mod1},K_{fun1}} = t_1 \wedge\\
&\exists t_2.\ [\![\overline{m_2}]\!]_{\tilde{\Delta},\tilde{\Sigma},\beta_2,K_{mod2},K_{fun2}} = t_2\\
&\implies\\
&(\tilde{\Delta}, \beta_1, \overline{m_1} \simeq_{\tilde{\Sigma},\omega,\nabla} \tilde{\Delta}, \beta_2, \overline{m_2} \iff t_1 \simeq_{\omega,\nabla} t_2)
\end{aligned}$$

Proof.
We fix the universally-quantified variables, and assume the antecedents.

Then, in order to prove the implication:

$$\tilde{\Delta}, \beta_1, \overline{m_1} \simeq_{\tilde{\Sigma},\omega,\nabla} \tilde{\Delta}, \beta_2, \overline{m_2} \impliedby t_1 \simeq_{\omega,\nabla} t_2$$

we instead prove its contra-positive. Thus, we assume:

$$\tilde{\Delta}, \beta_1, \overline{m_1} \not\simeq_{\tilde{\Sigma},\omega,\nabla} \tilde{\Delta}, \beta_2, \overline{m_2} \tag{6}$$

And our goal becomes:

$$t_1 \not\simeq_{\omega,\nabla} t_2$$

From Proposition (6), and by unfolding Definition 45, we get (w.l.o.g.):

$$\begin{aligned}
&\exists \Delta, \beta, \Sigma, K_{mod}, K_{fun}, \mathbb{C}.\\
&\mathtt{wfp}(\mathbb{C}) \wedge\\
&K_{mod} \uplus K_{mod1}, K_{fun} \uplus K_{fun1}, \Sigma \uplus \tilde{\Sigma}, (\Delta \uplus \tilde{\Delta}) + \omega, \beta \uplus \beta_1, \nabla \vdash \mathbb{C}[\overline{m_1}] \Downarrow \wedge\\
&K_{mod} \uplus K_{mod2}, K_{fun} \uplus K_{fun2}, \Sigma \uplus \tilde{\Sigma}, (\Delta \uplus \tilde{\Delta}) + \omega, \beta \uplus \beta_2, \nabla \nvdash \mathbb{C}[\overline{m_2}] \Downarrow
\end{aligned} \tag{7}$$

and our goal (by unfolding Definition 18) is to show that:

$$\exists \mathbb{C}.\ \omega, \nabla \vdash \mathbb{C}[t_1] \Downarrow \ \wedge\ \omega, \nabla \nvdash \mathbb{C}[t_2] \Downarrow$$

In order to show this goal, we pick:

$$\mathbb{C} = [\![\mathbb{C}]\!]_{\Delta,\Sigma,\beta,K_{mod},K_{fun}} \tag{8}$$

which we know from rule Module-list-translation that it exists because of conjunct $\mathtt{wfp}(\mathbb{C})$ of Proposition (7). By substitution from the assumptions and from Proposition (8), our goal is thus to show that:

$$\omega, \nabla \vdash [\![\mathbb{C}]\!]_{\Delta,\Sigma,\beta,K_{mod},K_{fun}}[\,[\![\overline{m_1}]\!]_{\tilde{\Delta},\tilde{\Sigma},\beta_1,K_{mod1},K_{fun1}}\,]\Downarrow \ \wedge\ \omega, \nabla \nvdash [\![\mathbb{C}]\!]_{\Delta,\Sigma,\beta,K_{mod},K_{fun}}[\,[\![\overline{m_2}]\!]_{\tilde{\Delta},\tilde{\Sigma},\beta_2,K_{mod2},K_{fun2}}\,]\Downarrow$$

By applying Lemma 106, it suffices to instead prove:

$$\begin{aligned}
&\omega, \nabla \vdash [\![\mathbb{C}[\overline{m_1}]_{\Delta\uplus\tilde{\Delta},\Sigma\uplus\tilde{\Sigma}}]\!]_{\Delta\cup\tilde{\Delta},\Sigma\cup\tilde{\Sigma},\beta\cup\beta_1,K_{mod}\cup K_{mod1},K_{fun}\cup K_{fun1}} \Downarrow \wedge\\
&\omega, \nabla \nvdash [\![\mathbb{C}[\overline{m_2}]_{\Delta\uplus\tilde{\Delta},\Sigma\uplus\tilde{\Sigma}}]\!]_{\Delta\cup\tilde{\Delta},\Sigma\cup\tilde{\Sigma},\beta\cup\beta_2,K_{mod}\cup K_{mod2},K_{fun}\cup K_{fun2}} \Downarrow
\end{aligned}$$

By Lemma 105, we immediately have the two conjuncts of our goal following from respectively the two conjuncts of Proposition (7). This concludes the proof of Lemma 120.

$\square$

Now, we turn to Lemma 121, which states that the compilers preserves contextual equivalence of **ImpMod** programs.

To prove this lemma, we rely on trace equivalence of **CHERIExp** (Definition 73), and trace equivalence of **ImpMod** as a go-between. Thus, preservation of contextual equivalence follows immediately by the following three lemmas:

- Soundness of target trace equivalence (Lemma 114)
- Compilation preserves trace equivalence (Lemma 122)
- Completeness of source trace equivalence (Lemma 117)

Lemma 121 ($[\![\cdot]\!]$ preserves contextual equivalence)**.**

$$\begin{aligned}
&\forall \overline{m_1}, \overline{m_2}, \tilde{\Delta}, \beta_1, \beta_2, K_{mod1}, K_{fun1}, K_{mod2}, K_{fun2}, \tilde{\Sigma}, \omega \in \mathbb{N}, \nabla \in \mathbb{Z}^-.\\
&\mathtt{dom}(\tilde{\Sigma}) = \{\mathtt{moduleID}(m) \mid m \in \overline{m_1}\} = \{\mathtt{moduleID}(m) \mid m \in \overline{m_2}\} \wedge\\
&\mathtt{dom}(\tilde{\Delta}) = \{\mathtt{moduleID}(m) \mid m \in \overline{m_1}\} = \{\mathtt{moduleID}(m) \mid m \in \overline{m_2}\} \wedge\\
&\exists t_1.\ [\![\overline{m_1}]\!]_{\tilde{\Delta}, \tilde{\Sigma}, \beta_1, K_{mod1}, K_{fun1}} = t_1 \wedge\\
&\exists t_2.\ [\![\overline{m_2}]\!]_{\tilde{\Delta}, \tilde{\Sigma}, \beta_2, K_{mod2}, K_{fun2}} = t_2\\
&\implies\\
&(\tilde{\Delta}, \beta_1, \overline{m_1} \simeq_{\tilde{\Sigma}, \omega, \nabla} \tilde{\Delta}, \beta_2, \overline{m_2} \implies t_1 \simeq_{\omega, \nabla} t_2)
\end{aligned}$$

Proof.
Immediate by Lemmas 114, 117 and 122. ☐

Lemma 122 (Compilation preserves trace equivalence)**.**

$$\beta_1, p_1 \overset{\mathrm{T}}{=}_{\omega, \nabla, \Delta, \Sigma} \beta_2, p_2 \implies [\![p_1]\!]_{\Delta, \Sigma, \beta_1, K_{mod1}, K_{fun1}} \overset{\mathrm{T}}{=}_{\omega, \nabla} [\![p_2]\!]_{\Delta, \Sigma, \beta_2, K_{mod2}, K_{fun2}}$$

Proof.
Unfolding using Definitions 73 and 79, we need to prove:

$$Tr_{\omega, \nabla, \Delta, \Sigma, \beta_1}(p_1) = Tr_{\omega, \nabla, \Delta, \Sigma, \beta_2}(p_2) \implies Tr_{\omega, \nabla}([\![p_1]\!]_{\Delta, \Sigma, \beta_1, K_{mod1}, K_{fun1}}) = Tr_{\omega, \nabla}([\![p_2]\!]_{\Delta, \Sigma, \beta_2, K_{mod2}, K_{fun2}})$$

This is immediate by Lemmas 131 and 173. ☐

Lemma 131 follows by lifting compiler forward simulation to the trace semantics.

6.1 Lifting compiler forward and backward simulation to trace semantics

Lemma 123 (Forward simulation of call attempt)**.**

$\forall K_{mod}, K_{fun}, \Sigma; \Delta; \beta; MVar; Fd, \langle Mem, stk, pc, \Phi, nalloc\rangle, \overline{mods_1}, \overline{m}, \lambda, \varsigma, \varsigma'$
$t, \langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle.$
$[\![mods_1]\!]_{\Delta,\Sigma,\beta,K_{mod},K_{fun}} = t \ \wedge$
$K_{mod}; K_{fun}; \overline{mods_1}; \Sigma; \Delta; \beta; MVar; Fd \vdash_{exec} \langle Mem, stk, pc, \Phi, nalloc\rangle \ \wedge$
$t \vdash_{exec} \langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle \ \wedge$
$modIDs = \{modID \mid (modID, \_, \_) \in \overline{mods_1}\} \ \wedge$
$K_{mod}; K_{fun}; \Sigma; \Delta; \beta; MVar; Fd; \langle Mem, stk, pc, \Phi, nalloc\rangle \cong_{modIDs} \langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle \ \wedge$
$\Sigma; \Delta; \beta; MVar; Fd \vdash \langle Mem, stk, pc, \Phi, nalloc\rangle \succ_{\approx} \langle Mem', stk', pc', \Phi', nalloc'\rangle$
$\implies$
$\langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle \succ_{\approx} \langle \mathcal{M}_c, \mathcal{M}'_d, stk', imp, \phi, \mathsf{ddc}', \mathsf{stc}', \mathsf{pcc}', \mathsf{nalloc}'\rangle \ \wedge$
$K_{mod}; K_{fun}; \Sigma; \Delta; \beta; MVar; Fd; \langle Mem', stk', pc', \Phi', nalloc'\rangle \cong_{modIDs} \langle \mathcal{M}_c, \mathcal{M}'_d, stk', imp, \phi, \mathsf{ddc}', \mathsf{stc}', \mathsf{pcc}', \mathsf{nalloc}'\rangle$

Proof.
Similar to case Call of Lemma 97. □

Lemma 124 (Forward simulation of call attempt)**.**

$\forall K_{mod}, K_{fun}, \Sigma; \Delta; \beta; MVar; Fd, \langle Mem, stk, pc, \Phi, nalloc\rangle, \overline{mods_1}, \overline{m}, \lambda, \varsigma, \varsigma'$
$t, \langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle.$
$[\![mods_1]\!]_{\Delta,\Sigma,\beta,K_{mod},K_{fun}} = t \ \wedge$
$K_{mod}; K_{fun}; \overline{mods_1}; \Sigma; \Delta; \beta; MVar; Fd \vdash_{exec} \langle Mem, stk, pc, \Phi, nalloc\rangle \ \wedge$
$t \vdash_{exec} \langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle \ \wedge$
$modIDs = \{modID \mid (modID, \_, \_) \in \overline{mods_1}\} \ \wedge$
$K_{mod}; K_{fun}; \Sigma; \Delta; \beta; MVar; Fd; \langle Mem, stk, pc, \Phi, nalloc\rangle \cong_{modIDs} \langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle \ \wedge$
$\langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle \succ_{\approx} \langle \mathcal{M}_c, \mathcal{M}'_d, stk', imp, \phi, \mathsf{ddc}', \mathsf{stc}', \mathsf{pcc}', \mathsf{nalloc}'\rangle$
$\implies$
$\Sigma; \Delta; \beta; MVar; Fd \vdash \langle Mem, stk, pc, \Phi, nalloc\rangle \succ_{\approx} \langle Mem', stk', pc', \Phi', nalloc'\rangle \ \wedge$
$K_{mod}; K_{fun}; \Sigma; \Delta; \beta; MVar; Fd; \langle Mem', stk', pc', \Phi', nalloc'\rangle \cong_{modIDs} \langle \mathcal{M}_c, \mathcal{M}'_d, stk', imp, \phi, \mathsf{ddc}', \mathsf{stc}', \mathsf{pcc}', \mathsf{nalloc}'\rangle$

Proof.
Similar to case cinvoke of Lemma 98. □

Lemma 125 (Compiler forward simulation lifted to a trace step)**.**

$\forall K_{mod}, K_{fun}, \Sigma; \Delta; \beta; MVar; Fd, \langle Mem, stk, pc, \Phi, nalloc\rangle, \overline{mods_1}, \overline{m}, \lambda, \varsigma, \varsigma'$

$t, \langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle.$

$\overline{m} \subseteq \overline{mods_1} \wedge$

$[\![mods_1]\!]_{\Delta,\Sigma,\beta,K_{mod},K_{fun}} = t \wedge$

$K_{mod}; K_{fun}; \overline{mods_1}; \Sigma; \Delta; \beta; MVar; Fd \vdash_{exec} \langle Mem, stk, pc, \Phi, nalloc\rangle \wedge$

$t \vdash_{exec} \langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle \wedge$

$modIDs = \{modID \mid (modID, \_, \_) \in \overline{mods_1}\} \wedge$

$K_{mod}; K_{fun}; \Sigma; \Delta; \beta; MVar; Fd; \langle Mem, stk, pc, \Phi, nalloc\rangle \cong_{modIDs} \langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle \wedge$

$\Sigma; \Delta; \beta; MVar; Fd \vdash \langle Mem, stk, pc, \Phi, nalloc\rangle, \varsigma \xrightarrow{\lambda}_{[\overline{m}]} \langle Mem', stk', pc', \Phi', nalloc'\rangle, \varsigma'$

$\implies$

$\langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle, \varsigma \xrightarrow{\lambda}_{[[\![\overline{m}]\!]_{\Delta,\Sigma,\beta,K_{mod},K_{fun}}]} \langle \mathcal{M}_c, \mathcal{M}'_d, stk', imp, \phi, \mathsf{ddc}', \mathsf{stc}', \mathsf{pcc}', \mathsf{nalloc}'\rangle, \varsigma' \wedge$

$K_{mod}; K_{fun}; \Sigma; \Delta; \beta; MVar; Fd; \langle Mem', stk', pc', \Phi', nalloc'\rangle \cong_{modIDs} \langle \mathcal{M}_c, \mathcal{M}'_d, stk', imp, \phi, \mathsf{ddc}', \mathsf{stc}', \mathsf{pcc}', \mathsf{nalloc}'\rangle$

Proof.
We distinguish two cases for λ:

- **Case $\lambda = \tau$:**
  Here, after instantiating Claim 28 using the given trace step
  $\Sigma; \Delta; \beta; MVar; Fd \vdash \langle Mem, stk, pc, \Phi, nalloc\rangle, \varsigma \xrightarrow{\lambda}_{[\overline{m}]} \langle Mem', stk', pc', \Phi', nalloc'\rangle, \varsigma'$,
  we obtain our goal immediately by applying Lemma 97.

- **Case $\lambda \neq \tau$:**
  Here, distinguish two cases:

  - **Case $s'.pc = \bot$:**
    Here, the goal is immediate by applying Lemma 123.
  - **Case $s'.pc \neq \bot$:**
    Here, after instantiating Claim 27,
    we obtain our goal immediately again by applying Lemma 97.

This concludes the proof of Lemma 125. □

Lemma 126 (Compiler backward simulation lifted to a trace step)**.**

$\forall K_{mod}, K_{fun}, \Sigma; \Delta; \beta; MVar; Fd, \langle Mem, stk, pc, \Phi, nalloc\rangle, \overline{mods_1}, \overline{m}, \lambda, \varsigma, \varsigma'$
$t, \langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle.$
$\overline{m} \subseteq \overline{mods_1}\ \wedge$
$[\![mods_1]\!]_{\Delta,\Sigma,\beta,K_{mod},K_{fun}} = t\ \wedge$
$K_{mod}; K_{fun}; \overline{mods_1}; \Sigma; \Delta; \beta; MVar; Fd \vdash_{exec} \langle Mem, stk, pc, \Phi, nalloc\rangle\ \wedge$
$t \vdash_{exec} \langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle\ \wedge$
$modIDs = \{modID \mid (modID, \_, \_) \in \overline{mods_1}\}\ \wedge$
$K_{mod}; K_{fun}; \Sigma; \Delta; \beta; MVar; Fd; \langle Mem, stk, pc, \Phi, nalloc\rangle \cong_{modIDs} \langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle\ \wedge$
$\langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle, \varsigma \xrightarrow{\lambda}_{[[\![\overline{m}]\!]_{\Delta,\Sigma,\beta,K_{mod},K_{fun}}]} \langle \mathcal{M}_c, \mathcal{M}'_d, stk', imp, \phi, \mathsf{ddc}', \mathsf{stc}', \mathsf{pcc}', \mathsf{nalloc}'\rangle, \varsigma'$
$\implies$
$\Sigma; \Delta; \beta; MVar; Fd \vdash \langle Mem, stk, pc, \Phi, nalloc\rangle, \varsigma \xrightarrow{\lambda}_{[\overline{m}]} \langle Mem', stk', pc', \Phi', nalloc'\rangle, \varsigma'\ \wedge$
$K_{mod}; K_{fun}; \Sigma; \Delta; \beta; MVar; Fd; \langle Mem', stk', pc', \Phi', nalloc'\rangle \cong_{modIDs} \langle \mathcal{M}_c, \mathcal{M}'_d, stk', imp, \phi, \mathsf{ddc}', \mathsf{stc}', \mathsf{pcc}', \mathsf{nalloc}'\rangle$

Proof.
We distinguish two cases for λ:

- **Case $\lambda = \tau$:**

  Here, after instantiating Claim 15 using the given trace step
  $\langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle, \varsigma \xrightarrow{\lambda}_{[[\![\overline{m}]\!]_{\Delta,\Sigma,\beta,K_{mod},K_{fun}}]} \langle \mathcal{M}_c, \mathcal{M}'_d, stk', imp, \phi, \mathsf{ddc}', \mathsf{stc}', \mathsf{pcc}', \mathsf{nalloc}'\rangle,$
  we obtain our goal immediately by applying Lemma 98.

- **Case $\lambda \neq \tau$:**

  Here, distinguish two cases:

  – **Case $s'.\mathcal{M}_c(s'.\mathsf{pcc}) = \bot$:**
  Here, the goal is immediate by applying Lemma 124.

  – **Case $s'.\mathcal{M}_c(s'.\mathsf{pcc}) \neq \bot$:**
  Here, after instantiating Claim 14,
  we obtain our goal immediately again by applying Lemma 98.

This concludes the proof of Lemma 126. ☐

Lemma 127 (Compiler forward simulation lifted to many trace steps)**.**

$\forall K_{mod}, K_{fun}, \Sigma; \Delta; \beta; MVar; Fd, \langle Mem, stk, pc, \Phi, nalloc\rangle, \overline{mods_1}, \overline{m}, \alpha, \varsigma, \varsigma'$
$t, \langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle.$
$\overline{m} \subseteq \overline{mods_1} \wedge$
$[\![mods_1]\!]_{\Delta,\Sigma,\beta,K_{mod},K_{fun}} = t \wedge$
$K_{mod}; K_{fun}; \overline{mods_1}; \Sigma; \Delta; \beta; MVar; Fd \vdash_{exec} \langle Mem, stk, pc, \Phi, nalloc\rangle \wedge$
$t \vdash_{exec} \langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle \wedge$
$modIDs = \{modID \mid (modID, \_, \_) \in \overline{mods_1}\} \wedge$
$K_{mod}; K_{fun}; \Sigma; \Delta; \beta; MVar; Fd; \langle Mem, stk, pc, \Phi, nalloc\rangle \cong_{modIDs} \langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle \wedge$
$\Sigma; \Delta; \beta; MVar; Fd \vdash \langle Mem, stk, pc, \Phi, nalloc\rangle, \varsigma \xrightarrow{\alpha}{}^{*}_{[\overline{m}]} \langle Mem', stk', pc', \Phi', nalloc'\rangle, \varsigma'$
$\implies$
$\langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle, \varsigma \xrightarrow{\alpha}{}^{*}_{[[\![\overline{m}]\!]_{\Delta,\Sigma,\beta,K_{mod},K_{fun}}]} \langle \mathcal{M}_c, \mathcal{M}'_d, stk', imp, \phi, \mathsf{ddc}', \mathsf{stc}', \mathsf{pcc}', \mathsf{nalloc}'\rangle, \varsigma' \wedge$
$K_{mod}; K_{fun}; \Sigma; \Delta; \beta; MVar; Fd; \langle Mem', stk', pc', \Phi', nalloc'\rangle \cong_{modIDs} \langle \mathcal{M}_c, \mathcal{M}'_d, stk', imp, \phi, \mathsf{ddc}', \mathsf{stc}', \mathsf{pcc}', \mathsf{nalloc}'\rangle$

Proof.
Follows from Lemma 125:
In the inductive step (case trace-closure-trans),
the necessary assumptions about the source, and target execution invariants $\vdash_{exec}$ and $\vdash_{exec}$ follow from Corollary 4 and Corollary 2 respectively,
after instantiating Claim 16, and Claim 29. ☐

Lemma 128 (Compiler backward simulation lifted to many trace steps)**.**

$\forall K_{mod}, K_{fun}, \Sigma; \Delta; \beta; MVar; Fd, \langle Mem, stk, pc, \Phi, nalloc\rangle, \overline{mods_1}, \overline{m}, \alpha, \varsigma, \varsigma'$
$t, \langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle.$
$\overline{m} \subseteq \overline{mods_1} \wedge$
$[\![mods_1]\!]_{\Delta,\Sigma,\beta,K_{mod},K_{fun}} = t \wedge$
$K_{mod}; K_{fun}; \overline{mods_1}; \Sigma; \Delta; \beta; MVar; Fd \vdash_{exec} \langle Mem, stk, pc, \Phi, nalloc\rangle \wedge$
$t \vdash_{exec} \langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle \wedge$
$modIDs = \{modID \mid (modID, \_, \_) \in \overline{mods_1}\} \wedge$
$K_{mod}; K_{fun}; \Sigma; \Delta; \beta; MVar; Fd; \langle Mem, stk, pc, \Phi, nalloc\rangle \cong_{modIDs} \langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle \wedge$
$\langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle, \varsigma \xrightarrow{\alpha}{}^{*}_{[[\![\overline{m}]\!]_{\Delta,\Sigma,\beta,K_{mod},K_{fun}}]} \langle \mathcal{M}_c, \mathcal{M}'_d, stk', imp, \phi, \mathsf{ddc}', \mathsf{stc}', \mathsf{pcc}', \mathsf{nalloc}'\rangle, \varsigma'$
$\implies$
$\Sigma; \Delta; \beta; MVar; Fd \vdash \langle Mem, stk, pc, \Phi, nalloc\rangle, \varsigma \xrightarrow{\alpha}{}^{*}_{[\overline{m}]} \langle Mem', stk', pc', \Phi', nalloc'\rangle, \varsigma' \wedge$
$K_{mod}; K_{fun}; \Sigma; \Delta; \beta; MVar; Fd; \langle Mem', stk', pc', \Phi', nalloc'\rangle \cong_{modIDs} \langle \mathcal{M}_c, \mathcal{M}'_d, stk', imp, \phi, \mathsf{ddc}', \mathsf{stc}', \mathsf{pcc}', \mathsf{nalloc}'\rangle$

Proof.
Follows from Lemma 126:
In the inductive step (case trace-closure-trans-src),
the necessary assumptions about the source, and target execution invariants $\vdash_{exec}$ and $\vdash_{exec}$ follow from Corollary 4 and Corollary 2 respectively,
after instantiating Claim 16, and Claim 29. ☐

Lemma 129 (Compiler forward simulation lifted to compressed trace steps)**.**

$\forall K_{mod}, K_{fun}, \Sigma; \Delta; \beta; MVar; Fd, \langle Mem, stk, pc, \Phi, nalloc\rangle, \overline{mods_1}, \overline{m}, \alpha, \varsigma, \varsigma'$
$t, \langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle.$
$\overline{m} \subseteq \overline{mods_1}\ \wedge$
$[\![mods_1]\!]_{\Delta,\Sigma,\beta,K_{mod},K_{fun}} = t\ \wedge$
$K_{mod}; K_{fun}; \overline{mods_1}; \Sigma; \Delta; \beta; MVar; Fd \vdash_{exec} \langle Mem, stk, pc, \Phi, nalloc\rangle\ \wedge$
$t \vdash_{exec} \langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle\ \wedge$
$modIDs = \{modID \mid (modID, \_, \_) \in \overline{mods_1}\}\ \wedge$
$K_{mod}; K_{fun}; \Sigma; \Delta; \beta; MVar; Fd; \langle Mem, stk, pc, \Phi, nalloc\rangle \cong_{modIDs} \langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle\ \wedge$
$\Sigma; \Delta; \beta; MVar; Fd \vdash \langle Mem, stk, pc, \Phi, nalloc\rangle, \varsigma \xrightarrow{\alpha}_{[\overline{m}]} \langle Mem', stk', pc', \Phi', nalloc'\rangle, \varsigma'$
$\implies$
$\langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle, \varsigma \xrightarrow{\alpha}_{[[\![\overline{m}]\!]_{\Delta,\Sigma,\beta,K_{mod},K_{fun}}]} \langle \mathcal{M}_c, \mathcal{M}_d', stk', imp, \phi, \mathsf{ddc}', \mathsf{stc}', \mathsf{pcc}', \mathsf{nalloc}'\rangle, \varsigma'\ \wedge$
$K_{mod}; K_{fun}; \Sigma; \Delta; \beta; MVar; Fd; \langle Mem', stk', pc', \Phi', nalloc'\rangle \cong_{modIDs} \langle \mathcal{M}_c, \mathcal{M}_d', stk', imp, \phi, \mathsf{ddc}', \mathsf{stc}', \mathsf{pcc}', \mathsf{nalloc}'\rangle$

Proof.
Follows from Lemmas 125 and 127. □

Lemma 130 (Compiler backward simulation lifted to compressed trace steps)**.**

$\forall K_{mod}, K_{fun}, \Sigma; \Delta; \beta; MVar; Fd, \langle Mem, stk, pc, \Phi, nalloc\rangle, \overline{mods_1}, \overline{m}, \alpha, \varsigma, \varsigma'$
$t, \langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle.$
$\overline{m} \subseteq \overline{mods_1}\ \wedge$
$[\![mods_1]\!]_{\Delta,\Sigma,\beta,K_{mod},K_{fun}} = t\ \wedge$
$K_{mod}; K_{fun}; \overline{mods_1}; \Sigma; \Delta; \beta; MVar; Fd \vdash_{exec} \langle Mem, stk, pc, \Phi, nalloc\rangle\ \wedge$
$t \vdash_{exec} \langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle\ \wedge$
$modIDs = \{modID \mid (modID, \_, \_) \in \overline{mods_1}\}\ \wedge$
$K_{mod}; K_{fun}; \Sigma; \Delta; \beta; MVar; Fd; \langle Mem, stk, pc, \Phi, nalloc\rangle \cong_{modIDs} \langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle\ \wedge$
$\langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \phi, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{mstc}, \mathsf{nalloc}\rangle, \varsigma \xrightarrow{\alpha}_{[[\![\overline{m}]\!]_{\Delta,\Sigma,\beta,K_{mod},K_{fun}}]} \langle \mathcal{M}_c, \mathcal{M}_d', stk', imp, \phi, \mathsf{ddc}', \mathsf{stc}', \mathsf{pcc}', \mathsf{nalloc}'\rangle, \varsigma'$
$\implies$
$\Sigma; \Delta; \beta; MVar; Fd \vdash \langle Mem, stk, pc, \Phi, nalloc\rangle, \varsigma \xrightarrow{\alpha}_{[\overline{m}]} \langle Mem', stk', pc', \Phi', nalloc'\rangle, \varsigma'\ \wedge$
$K_{mod}; K_{fun}; \Sigma; \Delta; \beta; MVar; Fd; \langle Mem', stk', pc', \Phi', nalloc'\rangle \cong_{modIDs} \langle \mathcal{M}_c, \mathcal{M}_d', stk', imp, \phi, \mathsf{ddc}', \mathsf{stc}', \mathsf{pcc}', \mathsf{nalloc}'\rangle$

Follows from Lemmas 126 and 128.

Lemma 131 (No trace is removed by compilation)**.**

$$\alpha \in Tr_{\omega,\nabla,\Delta,\Sigma,\beta}(p) \implies \alpha \in Tr_{\omega,\nabla}([\![p]\!]_{\Delta,\Sigma,\beta,K_{mod},K_{fun}})$$

Proof.
Immediate by Lemma 129 after unfolding Definitions 72 and 78. □

6.2 Strong and weak similarity

Definition 82 (Component-controlled memory region)**.**
In a given trace-execution state s, ς *of a program* $t \ltimes \overline{c}$ *(i.e.,* $t \ltimes \overline{c} \vdash_{exec} s$*), we define the function*

$\rho_{[\overline{c}]} : (\mathit{TargetState} \times 2^{\mathbb{Z}}) \to 2^{\mathbb{Z}}$ *which computes the set of memory addresses on which the similarity relation applies. For strong similarity, this set is all the memory that is reachable by* $\overline{c}$*. For weak similarity, this set is only the set of addresses that are private to* $\overline{c}$*.*

$$\begin{aligned}\rho_{[\overline{c}]}(s, \varsigma) \stackrel{\mathsf{def}}{=}\ & \mathit{if}\ s.\mathsf{pcc} \subseteq \mathsf{dom}(\overline{c}.\mathcal{M}_c)\\ & \mathit{then} \bigcup_{mid \in \mathsf{dom}(\overline{c}.imp)} \mathsf{reachable\_addresses}(\{s.\mathsf{mstc}(mid), \overline{c}.imp(mid).\mathsf{ddc}\}, s.\mathcal{M}_d)\\ & \mathit{else}\ (\bigcup_{mid \in \mathsf{dom}(\overline{c}.imp)} \mathsf{reachable\_addresses}(\{s.\mathsf{mstc}(mid), \overline{c}.imp(mid).\mathsf{ddc}\}, s.\mathcal{M}_d)) \setminus \varsigma\end{aligned}$$

Claim 31 (Controlled-region equality implies reachability equality)**.**

$$\begin{aligned}&\forall \overline{c}, s_1, s_2, \varsigma_1, \varsigma_2.\\ &\mathsf{dom}(s_1.\mathcal{M}_d) = \mathsf{dom}(s_2.\mathcal{M}_d)\ \wedge\\ &\varsigma_1 = \varsigma_2\ \wedge\\ &s_1.\mathsf{pcc} = s_2.\mathsf{pcc}\ \wedge\\ &\rho_{[\overline{c}]}(s_1, \varsigma_1) = \rho_{[\overline{c}]}(s_2, \varsigma_2)\\ &\implies\\ &\mathsf{reachable\_addresses}(\{s_1.\mathsf{stc}, s_1.\mathsf{ddc}\}, s_1.\mathcal{M}_d) =\\ &\mathsf{reachable\_addresses}(\{s_2.\mathsf{stc}, s_2.\mathsf{ddc}\}, s_2.\mathcal{M}_d)\end{aligned}$$

Definition 83 (Similarity of stack capabilities)**.** *Two stack capability maps* mstc_1 *and* mstc_2 *are similar up to/with respect to a component* $\overline{c}$ *iff all the* $\overline{c}$ *modules have the same stack capability value given by* mstc_1 *as that given by* mstc_2*. Formally:*
$\mathsf{mstc}_1 \approx_{[\overline{c}]} \mathsf{mstc}_2 \stackrel{\mathsf{def}}{=} \forall mid.\ mid \in \mathsf{dom}(\overline{c}.imp) \implies \mathsf{mstc}_1(mid) = \mathsf{mstc}_2(mid)$

Claim 32 (Similarity of mstc is an equivalence relation)**.**

Proof. Immediate by Definition 83. □

6.3 Stack similarity (successor-preserving isomorphism)

Two stacks stk_1 and stk_2 (of two executions of a program $\overline{c}$) are related whenever the number of alternations of program frames and context frames is the same in stk_1 as in stk_2, and each two corresponding program stack frames (i.e., a program stack-frame from stk_1 that corresponds to one from stk_2) are equal. The correspondence and the guarantee on the number of alternations are given by a function f between indexes of stk_1 and indexes stk_2. The function f satisfies the following conditions:

1. Domain of f is exhaustive of $\overline{c}$ call sites in stk_1, and contains top and bottom sentinel values.
2. Range of f is exhaustive of $\overline{c}$ call sites in stk_2 and contains top and bottom sentinel values.
3. f is sentinel-value preserving.
4. f is strictly monotone.
5. f is compatible with stack-frame equality (i.e., corresponding frames are equal).
6. f is a successor-preserving homomorphism.

A more formal definition is given by Definitions 84 and 85 which differ only in the condition on sentinel values. Weak stack-similarity (Definition 85) drops the top-sentinel-value requirement.

Conditions for strengthening and weakening are given next.

Definition 84 (Strong stack-similarity)**.**

$$
\begin{aligned}
&\mathit{stk}_1 \approx_{[\overline{c}]} \mathit{stk}_2 \\
&\qquad \stackrel{\mathsf{def}}{=} \\
&\qquad \exists f : \mathbb{Z} \rightharpoonup \mathbb{Z}. \\
&\qquad \mathsf{dom}(f) = \{i \in \mathsf{dom}(\mathit{stk}_1) \mid \mathit{stk}_1(i).\mathsf{pcc} \subseteq \mathsf{dom}(\overline{c}.\mathcal{M}_c)\} \uplus \{-1, \mathtt{length}(\mathit{stk}_1)\} \wedge \\
&\qquad \mathsf{range}(f) = \{i \in \mathsf{dom}(\mathit{stk}_2) \mid \mathit{stk}_2(i).\mathsf{pcc} \subseteq \mathsf{dom}(\overline{c}.\mathcal{M}_c)\} \uplus \{-1, \mathtt{length}(\mathit{stk}_2)\} \wedge \\
&\qquad f(-1) = -1 \wedge \\
&\qquad f(\mathtt{length}(\mathit{stk}_1)) = \mathtt{length}(\mathit{stk}_2) \wedge \\
&\qquad \forall i, j.\ i > j \implies f(i) > f(j) \wedge \\
&\qquad \forall i \in \mathsf{dom}(f) \setminus \{-1, \mathtt{length}(\mathit{stk}_1)\}.\ f(i) = j \implies \mathit{stk}_1(i) = \mathit{stk}_2(j) \wedge \\
&\qquad \forall i, j \in \mathsf{dom}(f).\ j = i + 1 \iff f(j) = f(i) + 1
\end{aligned}
$$

Definition 85 (Weak stack-similarity)**.**

$$
\begin{aligned}
&\mathit{stk}_1 \sim_{[\overline{c}]} \mathit{stk}_2 \\
&\qquad \stackrel{\mathsf{def}}{=} \\
&\qquad \exists f : \mathbb{Z} \rightharpoonup \mathbb{Z}. \\
&\qquad \mathsf{dom}(f) = \{i \in \mathsf{dom}(\mathit{stk}_1) \mid \mathit{stk}_1(i).\mathsf{pcc} \subseteq \mathsf{dom}(\overline{c}.\mathcal{M}_c)\} \uplus \{-1\} \wedge \\
&\qquad \mathsf{range}(f) = \{i \in \mathsf{dom}(\mathit{stk}_2) \mid \mathit{stk}_2(i).\mathsf{pcc} \subseteq \mathsf{dom}(\overline{c}.\mathcal{M}_c)\} \uplus \{-1\} \wedge \\
&\qquad f(-1) = -1 \wedge \\
&\qquad \forall i, j.\ i > j \implies f(i) > f(j) \wedge \\
&\qquad \forall i \in \mathsf{dom}(f) \setminus \{-1, \mathtt{length}(\mathit{stk}_1)\}.\ f(i) = j \implies \mathit{stk}_1(i) = \mathit{stk}_2(j) \wedge \\
&\qquad \forall i, j \in \mathsf{dom}(f).\ j = i + 1 \iff f(j) = f(i) + 1
\end{aligned}
$$

Notice that the functions f used in Definitions 84 and 85 are injective because they are strictly monotone.

Lemma 132 (A strictly-monotone function is injective)**.**

$$
\begin{aligned}
&\forall f. \\
&(\forall i, j.\ i > j \implies f(i) > f(j)) \\
&\implies \\
&(\forall i, j.\ i \neq j \implies f(i) \neq f(j))
\end{aligned}
$$

Proof. Immediate by the anti-reflexivity and asymmetry of the $<$ relation. ☐

Definition 86 (Trace-state similarity)**.**
Given two trace states s_1, ς_1 *and* s_2, ς_2*, we define between them two similarity relations: strong similarity* $s_1, \varsigma_1 \approx_{[\overline{c}]} s_2, \varsigma_2$*, and weak similarity* $s_1, \varsigma_1 \sim_{[\overline{c}]} s_2, \varsigma_2$ *where both relations are parametrized with a component* $\overline{c}$ *for which the trace is collected. The intuition is that strong similarity holds as long as* $\overline{c}$ *is executing, and weak similarity holds as long as the context is executing. Strong similarity satisfies lock-step simulation, and weak similarity satisfies option simulation.*

Formally:

$$
\begin{aligned}
s_1, \varsigma_1 \approx_{[\overline{c}]} s_2, \varsigma_2 \stackrel{\mathsf{def}}{=}& \\
& \rho_{[\overline{c}]}(s_1, \varsigma_1) = \rho_{[\overline{c}]}(s_2, \varsigma_2) = r \wedge \\
& s_1.\mathit{stk} \approx_{[\overline{c}]} s_2.\mathit{stk} \wedge \\
& s_1.\mathsf{mstc} \approx_{[\overline{c}]} s_2.\mathsf{mstc} \wedge \\
& \varsigma_1 = \varsigma_2 \wedge \\
& s_1.\mathcal{M}_d|_r = s_2.\mathcal{M}_d|_r \wedge \\
& s_1.\mathsf{ddc} = s_2.\mathsf{ddc} \wedge \\
& s_1.\mathsf{stc} = s_2.\mathsf{stc} \wedge \\
& s_1.\mathsf{pcc} = s_2.\mathsf{pcc} \wedge \\
& s_1.\mathsf{nalloc} = s_2.\mathsf{nalloc}
\end{aligned}
$$

and

$$
\begin{aligned}
s_1, \varsigma_1 \sim_{[\overline{c}], priv} s_2, \varsigma_2 \stackrel{\mathsf{def}}{=}& \\
& (s_1.\mathsf{pcc} \cap \mathsf{dom}(\overline{c}.\mathcal{M}_c) = \emptyset \\
& \iff \\
& s_2.\mathsf{pcc} \cap \mathsf{dom}(\overline{c}.\mathcal{M}_c) = \emptyset) \wedge \\
& s_1.\mathit{stk} \sim_{[\overline{c}]} s_2.\mathit{stk} \wedge \\
& s_1.\mathsf{mstc} \approx_{[\overline{c}]} s_2.\mathsf{mstc} \wedge \\
& \varsigma_1 = \varsigma_2 \wedge \\
& s_1.\mathcal{M}_d|_{priv} = s_2.\mathcal{M}_d|_{priv}
\end{aligned}
$$

Lemma 133 (Strong stack-similarity is an equivalence relation)**.**

- ***Reflexivity:*** $\forall \mathit{stk}, \overline{c}.\ \mathit{stk} \approx_{[\overline{c}]} \mathit{stk}$
- ***Symmetry:*** $\forall \mathit{stk}_1, \mathit{stk}_2, \overline{c}.\ \mathit{stk}_1 \approx_{[\overline{c}]} \mathit{stk}_2 \implies \mathit{stk}_2 \approx_{[\overline{c}]} \mathit{stk}_1$
- ***Transitivity:*** $\forall \mathit{stk}_1, \mathit{stk}_2, \mathit{stk}_3, \overline{c}.\ \mathit{stk}_1 \approx_{[\overline{c}]} \mathit{stk}_2 \wedge \mathit{stk}_2 \approx_{[\overline{c}]} \mathit{stk}_3 \implies \mathit{stk}_1 \approx_{[\overline{c}]} \mathit{stk}_3$

Proof.

- For reflexivity, pick the identity function $f(x) = x$.
- For symmetry, obtain f by unfolding the assumption using Definition 84.
  Then, pick f^{-1} such that $\mathsf{dom}(f^{-1}) := \mathsf{range}(f)$ and $f^{-1}(f(x)) := x$.
  By injectivity of f (Lemma 132), notice that $f^{-1}(f(x))$ is well defined, and that $\mathsf{range}(f^{-1}) = \mathsf{dom}(f)$.
  The "frame-relatedness" condition for f^{-1} follows by symmetry of the frame relation from the frame-relatedness condition on f.
  The remaining conditions are easy.
- For transitivity, obtain f_1 and f_2 by unfolding the assumption using Definition 84.
  Then, pick $f_{1,3} := f_2 \circ f_1$. Notice that $f_{1,3}$ has the desired domain and range.
  The "frame-relatedness" condition for $f_{1,3}$ follows by transitivity of the frame relation from the frame-relatedness conditions on f_1 and f_2.
  The remaining conditions are easy.

□

Claim 33 (Weak stack-similarity is an equivalence relation)**.**

- ***Reflexivity:*** $\forall stk, \overline{c}.\ stk \sim_{[\overline{c}]} stk$
- ***Symmetry:*** $\forall stk_1, stk_2, \overline{c}.\ stk_1 \sim_{[\overline{c}]} stk_2 \implies stk_2 \sim_{[\overline{c}]} stk_1$
- ***Transitivity:*** $\forall stk_1, stk_2, stk_3, \overline{c}.\ stk_1 \sim_{[\overline{c}]} stk_2 \wedge stk_2 \sim_{[\overline{c}]} stk_3 \implies stk_1 \sim_{[\overline{c}]} stk_3$

Proof. Similar to the proof of Lemma 133. □

Claim 34 (State similarity is an equivalence relation)**.**
The relation $\approx_{[\overline{c}]}$ is reflexive, symmetric, and transitive.

- $\forall s, \varsigma, \overline{c}.\ s, \varsigma \approx_{[\overline{c}]} s, \varsigma$
- $\forall s_1, \varsigma_1, s_2, \varsigma_2, \overline{c}.\ s_1, \varsigma_1 \approx_{[\overline{c}]} s_2, \varsigma_2 \implies s_2, \varsigma_2 \approx_{[\overline{c}]} s_1, \varsigma_1$
- $\forall s_1, \varsigma_1, s_2, \varsigma_2, s_3, \varsigma_3, \overline{c}.\ s_1, \varsigma_1 \approx_{[\overline{c}]} s_2, \varsigma_2 \wedge s_2, \varsigma_2 \approx_{[\overline{c}]} s_3, \varsigma_3 \implies s_1, \varsigma_1 \approx_{[\overline{c}]} s_3, \varsigma_3$

Proof. Follows from Claim 32 and Lemma 133. □

Lemma 134 (Similarity of stack capabilities compatible with uniform substitution)**.**

$$\forall \mathsf{mstc}_1, \mathsf{mstc}_2, mid, \mathsf{stc}.\ \mathsf{mstc}_1 \approx_{[\overline{c}]} \mathsf{mstc}_2 \implies \mathsf{mstc}_1[mid \mapsto \mathsf{stc}] \approx_{[\overline{c}]} \mathsf{mstc}_2[mid \mapsto \mathsf{stc}]$$

Proof. Immediate by unfolding Definition 83, and a case distinction on the map's key entry. □

Lemma 135 (Initial states of the program of interest are strongly related)**.**

$$\begin{aligned}
&s_1 = \mathsf{initial_state}(\mathbb{C}_1 \ltimes p, \mathsf{main_module}(\mathbb{C}_1 \ltimes p)) \wedge \\
&s_2 = \mathsf{initial_state}(\mathbb{C}_2 \ltimes p, \mathsf{main_module}(\mathbb{C}_2 \ltimes p)) \wedge \\
&s_1.\mathsf{pcc} \subseteq \mathsf{dom}(p.\mathcal{M}_c) \wedge \\
&s_1.\mathsf{pcc} \subseteq \mathsf{dom}(p.\mathcal{M}_c) \\
&\implies \\
&s_1, \emptyset \approx_{[p]} s_2, \emptyset
\end{aligned}$$

Proof.
Follows by Definition 86. □

Lemma 136 (Initial states of the context are weakly related)**.**

$$\begin{aligned}
&s_1 = \mathsf{initial_state}(\mathbb{C}_1 \ltimes p, \mathsf{main_module}(\mathbb{C}_1 \ltimes p)) \wedge \\
&s_2 = \mathsf{initial_state}(\mathbb{C}_2 \ltimes p, \mathsf{main_module}(\mathbb{C}_2 \ltimes p)) \wedge \\
&s_1.\mathsf{pcc} \nsubseteq \mathsf{dom}(p.\mathcal{M}_c) \wedge \\
&s_1.\mathsf{pcc} \nsubseteq \mathsf{dom}(p.\mathcal{M}_c) \\
&\implies \\
&s_1, \emptyset \sim_{[p], \rho_{[p]}(s_1, \emptyset)} s_2, \emptyset
\end{aligned}$$

Proof.
Follows by Definition 86. □

Lemma 137 (Terminal states are strongly-related to only terminal states)**.**

$$\begin{aligned}
&s_1, \varsigma_1 \approx_{[p]} s_2, \varsigma_2 \ \wedge \\
&\vdash_t s_1 \\
&\ \Longrightarrow \\
&\vdash_t s_2
\end{aligned}$$

Proof.
Follows by unfolding Definition 86 and Definition 13 then rewriting using $s_1.\mathsf{pcc} = s_2.\mathsf{pcc}$. □

Lemma 138 (Equality of expression evaluation between strongly-similar states)**.**

$$\begin{aligned}
&\forall t_1, t_2, s_1, s_2, \varsigma_1, \varsigma_2, \mathcal{E}, r. \\
&t_1 \vdash_{exec} s_1 \ \wedge \\
&t_2 \vdash_{exec} s_2 \ \wedge \\
&r = \mathsf{reachable_addresses}(\{s_1.\mathsf{stc}, s_1.\mathsf{ddc}\}, s_1.\mathcal{M}_d) \ \wedge \\
&s_1.\mathsf{stc} = s_2.\mathsf{stc} \wedge \\
&s_1.\mathsf{ddc} = s_2.\mathsf{ddc} \wedge \\
&s_1.\mathcal{M}_d|_r = s_2.\mathcal{M}_d|_r \\
&\mathcal{E}, s_1.\mathcal{M}_d, s_1.\mathsf{ddc}, s_1.\mathsf{stc}, s_1.\mathsf{pcc} \Downarrow v \\
&\ \Longrightarrow \\
&\mathcal{E}, s_2.\mathcal{M}_d, s_2.\mathsf{ddc}, s_2.\mathsf{stc}, s_2.\mathsf{pcc} \Downarrow v
\end{aligned}$$

Proof.
We assume the antecedents, and prove our goal by induction on the evaluation $\mathcal{E}, s_1.\mathcal{M}_d, s_1.\mathsf{ddc}, s_1.\mathsf{stc}, s_1.\mathsf{pcc} \Downarrow v$:

1. **Case evalconst:**

   Here, observe that $n, \_, \_, \_, \_ \Downarrow n$, so our goal follows.

2. **Case evalddc:**

3. **Case evalstc:**

   Here, we obtain our goals by conjuncts $s_1.\mathsf{ddc} = s_2.\mathsf{ddc}$, and $s_1.\mathsf{stc} = s_2.\mathsf{stc}$ of the antecedent respectively.

4. **Case evalCapType:**

5. **Case evalCapStart:**

6. **Case evalCapEnd:**

7. **Case evalCapOff:**

8. **Case evalBinOp:**

9. **Case evalIncCap:**

10. **Case evalLim:**

    Here, our goals follow by inverting the corresponding rule, applying the induction hypothesis, and re-applying the rule for the s_2 components.

11. **Case evalDeref:**

    - Here, we have $\mathcal{E} = \mathtt{deref}(\mathcal{E}')$, and we obtain the preconditions $\mathcal{E}', s_1.\mathcal{M}_d, s_1.\mathsf{ddc}, s_1.\mathsf{stc}, s_1.\mathsf{ddc} \Downarrow v$, $\vdash_\delta v$, and $v' = s_1.\mathcal{M}_d(v.s + v.\mathit{off})$.

- The induction hypothesis gives us that $\mathcal{E}', s_2.\mathcal{M}_d, s_2.\mathsf{ddc}, s_2.\mathsf{stc}, s_2.\mathsf{ddc} \Downarrow v$.
- So, we need to show that $s_2.\mathcal{M}_d(v.s + v.\mathit{off}) = v' = s_1.\mathcal{M}_d(v.s + v.\mathit{off})$.
- But we have by assumption that $s_2.\mathcal{M}_d|_r = s_1.\mathcal{M}_d|_r$.
  So it suffices to show that $v.s + v.\mathit{off} \in r$.
- But by Lemma 25 about completeness of $\mathsf{reachable\_addresses}$, and the definition of r from the assumption we have that $[v.s, v.e) \subseteq r$.
- So our sufficient goal "$v.s + v.\mathit{off} \in r$" follows by the definition of $\subseteq$ because from the above-obtained precondition $\vdash_\delta v$, and by Definition 2, we know that $v.s + v.\mathit{off} \in [v.s, v.e)$.
  (Notice that Lemma 25 is applicable by the preconditions of rule exec-state of conjunct $t_1 \ltimes \bar{c} \vdash_{exec} s_1$ of the assumption, and the preconditions $\mathcal{E}', s_1.\mathcal{M}_d, s_1.\mathsf{ddc}, s_1.\mathsf{stc}, s_1.\mathsf{ddc} \Downarrow v$ and $\vdash_\delta v$.)

□

Lemma 139 (The empty stack is in a singleton equivalence class of strong stack-similarity)**.**

$$\begin{array}{l}\forall \mathit{stk}, \bar{c}.\\ \mathtt{nil} \approx_{[\bar{c}]} \mathit{stk}\\ \implies\\ \mathit{stk} = \mathtt{nil}\end{array}$$

Proof.
By unfolding the assumption using Definition 84, obtain f where the following hold:
$f(-1) = -1$, and $f(0) = \mathtt{length}(\mathit{stk})$.

But by instantiating the successor-preservation assumption, know that $f(0) = 0$,
hence $\mathtt{length}(\mathit{stk}) = 0$, thus it must be that $\mathit{stk} = \mathtt{nil}$.

□

Lemma 140 (Adequacy of strong stack-similarity (syncing border-crossing return to non-$\bar{c}$ call-site))**.**

$$\begin{array}{l}\forall \mathit{stk}_1, \mathit{stk}_2, \bar{c}, \mathit{pcc}_1, \mathit{pcc}_2.\\ \mathit{pcc}_1 \not\subseteq \mathsf{dom}(\bar{c}.\mathcal{M}_c) \wedge\\ \mathit{stk}_1{+}{+}[\mathit{pcc}_1] \approx_{[\bar{c}]} \mathit{stk}_2{+}{+}[\mathit{pcc}_2]\\ \implies\\ \mathit{pcc}_2 \not\subseteq \mathsf{dom}(\bar{c}.\mathcal{M}_c)\end{array}$$

Proof.

- Suppose the negation were true: $\mathit{pcc}_2 \subseteq \mathsf{dom}(\bar{c}.\mathcal{M}_c)$.
- Then, by assumption (unfolding Definition 84), we obtain (*):
  f where $\mathtt{length}(\mathit{stk}_2) \in \mathsf{range}(f)$.
- But we also know by the sentinel-preservation assumption that (**):
  $f(\mathtt{length}(\mathit{stk}_1) + 1) = \mathtt{length}(\mathit{stk}_2) + 1$.
- But then using (*) and (**) to instantiate the " $\Longleftarrow$ " direction of the successor-preservation assumption, we know that
  $f(\mathtt{length}(\mathit{stk}_1)) = \mathtt{length}(\mathit{stk}_2)$.

- This last assertion together with the assumption that defines $\mathsf{dom}(f)$ gives us $pcc_1 \subseteq \mathsf{dom}(\bar{c}.\mathcal{M}_c)$.
- This last assertion in turn immediately contradicts our assumption.

$\square$

Lemma 141 (Weak stack-similarity is preserved by a unilateral silent return)**.**

$$\begin{aligned}
&\forall stk_1, stk_2, \bar{c}.\\
&stk_1 \sim_{[\bar{c}]} stk_2 \wedge\\
&\mathtt{top}(stk_1).\mathsf{pcc} \not\subseteq \mathsf{dom}(\bar{c}.\mathcal{M}_c)\\
&\implies\\
&\mathtt{pop}(stk_1).stk \sim_{[\bar{c}]} stk_2
\end{aligned}$$

Proof.
By unfolding Definition 85, we obtain f satisfying:
$\mathsf{dom}(f) = \{i \in \mathsf{dom}(stk_1) \mid stk_1(i).\mathsf{pcc} \subseteq \mathsf{dom}(\bar{c}.\mathcal{M}_c)\} \uplus \{-1\}$

Moreover, we infer from our assumption about $\mathtt{top}(stk_1)$ that (*):
$\mathtt{length}(stk_1) - 1 \notin \mathsf{dom}(f)$.

We also know by the spec. of $\mathtt{pop}$ that (**):
$\mathsf{dom}(\mathtt{pop}(stk_1).stk) = \mathsf{dom}(stk_1) \uplus \{\mathtt{length}(stk_1) - 1\}$

By unfolding our goal using Definition 85, it suffices to pick the same f obtained above, if we prove all the following:

1. Domain of f is exhaustive of $\bar{c}$ call sites in $\mathtt{pop}(stk_1).stk$.

   Immediate by assumption after noticing by (**) and (*) that $\mathsf{dom}(stk_1) = \mathsf{dom}(\mathtt{pop}(stk_1).stk)$.

2. Range of f is exhaustive of $\bar{c}$ call sites in stk_2

   Immediate by assumption.

3. f is sentinel-value preserving.

   Immediate by assumption.

4. f is strictly monotone.

   Immediate by assumption.

5. f is compatible with stack-frame equality.

   Immediate by assumption.

6. f is successor-preserving.

   Immediate by assumption.

This concludes our proof of Lemma 141. $\square$

Lemma 142 (Weak stack-similarity is preserved by a unilateral silent call)**.**

$$\begin{aligned}
&\forall stk_1, stk_2, \bar{c}, pcc.\\
&stk_1 \sim_{[\bar{c}]} stk_2 \wedge\\
&pcc \not\subseteq \mathsf{dom}(\bar{c}.\mathcal{M}_c)\\
&\implies\\
&\mathtt{push}(stk_1, (_, pcc, _, _)) \sim_{[\bar{c}]} stk_2
\end{aligned}$$

Proof. Similar to the proof of Lemma 141. We avoid repetition. $\square$

Lemma 143 (Weakening of strong stack-similarity)**.**

$$\begin{aligned}
&\forall stk_1, stk_2, \overline{c}.\\
&stk_1 \approx_{[\overline{c}]} stk_2\\
&\implies\\
&stk_1 \sim_{[\overline{c}]} stk_2
\end{aligned}$$

Proof.
By unfolding the assumption using Definition 84, we obtain f.
Then, by unfolding the goal using Definition 85, we pick:
$f' := f \setminus \{\mathtt{length}(stk_1) \mapsto \mathtt{length}(stk_2)\}$
Thus, it remains to prove all of the following:

1. Domain of f' is exhaustive of $\overline{c}$ call sites in stk_1
   $(\mathsf{dom}(f') = \{i \in \mathsf{dom}(stk_1) \mid stk_1(i).\mathsf{pcc} \subseteq \mathsf{dom}(\overline{c}.\mathcal{M}_c)\} \uplus \{-1\})$.

   Immediate by the corresponding assumption about f, and the choice of f'.

2. Range of f' is exhaustive of $\overline{c}$ call sites in stk_2
   $(\mathsf{range}(f') = \{i \in \mathsf{dom}(stk_2) \mid stk_2(i).\mathsf{pcc} \subseteq \mathsf{dom}(\overline{c}.\mathcal{M}_c)\} \uplus \{-1\})$

   Immediate by the corresponding assumption about f, and the choice of f'.

3. f' is sentinel-value preserving
   $(f'(-1) = -1)$.

   Immediate by the corresponding assumption about f and the choice of f'.

4. f' is strictly monotone
   $(\forall i, j.\ i > j \implies f'(i) > f'(j))$.

   Pick arbitrary $i, j \in \mathsf{dom}(f')$.
   Notice that $i, j \in \mathsf{dom}(f)$.
   Thus, our goal is immediate by the corresponding assumption about f.

5. f' is compatible with stack-frame equality
   $(\forall i \in \mathsf{dom}(f') \setminus \{-1, \mathtt{length}(stk_1)\}.\ f'(i) = j \implies stk_1(i) = stk_2(j))$.

   Proof is the same as the previous subgoal.

6. f' is successor-preserving
   $(\forall i, j \in \mathsf{dom}(f').\ j = i + 1 \iff f'(j) = f'(i) + 1)$.

   Proof is the same as the previous subgoal.

This concludes the proof of Lemma 143. □

Lemma 144 (Strong stack-similarity is preserved by a bilateral call (from same $\overline{c}$-call-site))**.**

$$\begin{aligned}
&\forall stk_1, stk_2, \overline{c}, pcc.\\
&stk_1 \approx_{[\overline{c}]} stk_2\ \wedge\\
&pcc \subseteq \mathsf{dom}(\overline{c}.\mathcal{M}_c)\\
&\implies\\
&\mathtt{push}(stk_1, (_, pcc, _, _)) \approx_{[\overline{c}]} \mathtt{push}(stk_2, (_, pcc, _, _))
\end{aligned}$$

Proof.
By unfolding the assumption using Definition 84, we obtain f.

Then, by unfolding the goal using Definition 84, we pick:
$f' := f \uplus \{\mathtt{length}(stk_1) + 1 \mapsto \mathtt{length}(stk_2) + 1\}$.
It thus remains to prove all of the following:

1. Domain of f' is exhaustive of $\overline{c}$ call sites in $\texttt{push}(stk_1, (\_, pcc, \_, \_))$, and
2. Range of f' is exhaustive of $\overline{c}$ call sites in stk_2

   Immediate by the corresponding assumptions and by the choice of f'.
3. f is sentinel-value preserving.

   The bottom sentinel value is preserved: $f'(-1) = -1$ follows from $f(-1) = -1$.

   The top sentinel value is preserved by choice of f'.
4. f is strictly monotone.

   Pick arbitrary $i, j \in \mathsf{dom}(f')$ where $i < j$.

   Show $f'(i) < f'(j)$.

   Distinguish three cases:

   - **Case** $i, j \in \mathsf{dom}(f)$
     Immediate by strict monotonicity of f.
   - **Case** $i \notin \mathsf{dom}(f)$**:**
     Know $i = \texttt{length}(stk_1) + 1$.
     Thus, $j > \texttt{length}(stk_1) + 1$.
     Thus, this case is impossible by the definition of $\mathsf{dom}(f')$.
   - **Case** $j \notin \mathsf{dom}(f)$**:**
     Know $j = \texttt{length}(stk_1) + 1$, and
     know $i \in \mathsf{dom}(f)$ (by choice of f').
     Thus, the goal becomes
     $f(i) < f'(\texttt{length}(stk_1)) + 1$
     By choice of f', the goal becomes $f(i) < \texttt{length}(stk_2) + 1$
     This is immediate by the definition of $\mathsf{range}(f)$.
5. f is compatible with stack-frame equality.

   Immediate by the choice of f', and the corresponding assumption about f.
6. f is successor-preserving.

   Pick arbitrary $i, j \in \mathsf{dom}(f)$ with $i = j + 1$.

   Show $f'(i) = f'(j) + 1$.

   Distinguish the following cases:

   - **Case** $i, j \in \mathsf{dom}(f)$**:**
     Immediate by the corresponding assumption about f.
   - **Case** $i \notin \mathsf{dom}(f)$**:**
     Know $i = \texttt{length}(stk_1) + 1$
     Goal becomes $\texttt{length}(stk_2) = f'(\texttt{length}(stk_1))$.
     Immediate by the choice of f'.
   - **Case** $j \notin \mathsf{dom}(f)$**:**
     Know $j = \texttt{length}(stk_1) + 1$.
     Thus, $i = \texttt{length}(stk_1) + 2$ which is impossible by the definition of $\mathsf{dom}(f')$.

This concludes the proof of Lemma 144. $\square$

Lemma 145 (Strong stack-similarity is weakened by a bilateral return to a non-$\overline{c}$-call-site)**.**

$$\begin{aligned}&\forall stk_1, stk_2, \overline{c}, pcc_1, pcc_2.\\ &stk_1{+}{+}[pcc_1] \approx_{[\overline{c}]} stk_2{+}{+}[pcc_2] \wedge\\ &pcc_1 \not\subseteq \mathsf{dom}(\overline{c}.\mathcal{M}_c)\\ &\implies\\ &stk_1 \sim_{[\overline{c}]} stk_2\end{aligned}$$

Proof.
Assume the antecedents.
By instantiating Lemma 140 using the assumptions, we know that
$pcc_2 \not\subseteq \mathsf{dom}(\overline{c}.\mathcal{M}_c)$ (*).
Also, by instantiating Lemma 143 using the assumptions, we know
$stk_1{+}{+}[pcc_1] \sim_{[\overline{c}]} stk_2{+}{+}[pcc_2]$ (**).
Thus, by instantiating Lemma 141 using (*) and (**), we know
$stk_1 \sim_{[\overline{c}]} stk_2{+}{+}[pcc_2]$ (POPPED-LEFT).
By instantiating symmetry (Claim 33) with (POPPED-LEFT), we thus know
$stk_2{+}{+}[pcc_2] \sim_{[\overline{c}]} stk_1$.
Now again by instantiating Lemma 141, we know
$stk_2 \sim_{[\overline{c}]} stk_1$.
Finally, by instantiating symmetry (Claim 33), we know
$stk_1 \sim_{[\overline{c}]} stk_2$, which is our goal. □

Lemma 146 (Strong stack-similarity is preserved by a bilateral return to a $\overline{c}$-call-site)**.**

$$\begin{aligned}&\forall stk_1, stk_2, \overline{c}, pcc_1, pcc_2.\\ &stk_1{+}{+}[pcc_1] \approx_{[\overline{c}]} stk_2{+}{+}[pcc_2] \wedge\\ &pcc_1 \subseteq \mathsf{dom}(\overline{c}.\mathcal{M}_c)\\ &\implies\\ &stk_1 \approx_{[\overline{c}]} stk_2\end{aligned}$$

Proof.
Assume the antecedents (unfold by Definition 84 to obtain f).
By the assumptions, know that $pcc_2 \subseteq \mathsf{dom}(\overline{c}.\mathcal{M}_c)$:

- Suppose the negation were true: $pcc_2 \not\subseteq \mathsf{dom}(\overline{c}.\mathcal{M}_c)$.
- By instantiating symmetry (Lemma 133) using our assumption, then instantiating Lemma 140, we know $pcc_1 \not\subseteq \mathsf{dom}(\overline{c}.\mathcal{M}_c)$ which contradicts the case condition.

In particular, by instantiating the definition of $\mathsf{dom}(f)$ using the assumption, we know that
$f(\mathtt{length}(stk_1)) = \mathtt{length}(stk_2)$ (*)
by instantiating the "$\implies$" direction of the successor-preservation assumption (about f) using the sentinel-value preservation assumption (about f).

For our goal (unfolding Definition 84), we pick
$f' := f \setminus \{\mathtt{length}(stk_1) + 1 \mapsto \mathtt{length}(stk_2) + 1\}$.

1. Domain of f' is exhaustive of $\overline{c}$ call sites in stk_1.

   Follows from the corresponding assumption about f and from the choice of f'.

   The sentinel value follows from $pcc_1 \subseteq \mathsf{dom}(\overline{c}.\mathcal{M}_c)$.

2. Range of f' is exhaustive of $\overline{c}$ call sites in stk_2.

   Follows from the corresponding assumption about f and from the choice of f'.

   The sentinel value follows from $pcc_2 \subseteq \mathsf{dom}(\overline{c}.\mathcal{M}_c)$.

3. f' is sentinel-value preserving.

   Follows from the corresponding assumption about f and from the choice of f', .

4. f' is strictly monotone:
   $(\forall i, j.\ i > j \implies f'(i) > f'(j))$.

   Notice that $f' \subseteq f$, so for arbitrary $i, j \in \mathsf{dom}(f')$, the consequent holds by instantiating the strict-monotonicity assumption about f.

5. f' is compatible with stack-frame equality.

   Pick an arbitrary i where $i \in \mathsf{dom}(f') \setminus \{-1, \mathtt{length}(stk_1)\}$.

   Show that $stk_1(i) = stk_2(f'(i))$.

   This is immediate by instantiating the corresponding assumption (compatibility with stack-frame equality) for f.

6. f' is successor-preserving.

   Pick arbitrary $i, j \in \mathsf{dom}(f')$.

   Show that $j = i + 1 \iff f'(j) = f'(i) + 1$.

   Observe that $\mathsf{dom}(f') \subseteq \mathsf{dom}(f)$.

   Thus, the goal is immediate successor preservation about f.

This concludes the proof of Lemma 146. □

Lemma 147 (Strengthening of weak stack-similarity by a bilateral call from non-$\overline{c}$ call-sites)**.**

$$
\begin{aligned}
&\forall stk_1, stk_2, \overline{c}, pcc_1, pcc_2.\\
&stk_1 \sim_{[\overline{c}]} stk_2 \ \wedge\\
&pcc_1 \not\subseteq \mathsf{dom}(\overline{c}.\mathcal{M}_c) \ \wedge\\
&pcc_2 \not\subseteq \mathsf{dom}(\overline{c}.\mathcal{M}_c)\\
&\implies\\
&\mathtt{push}(stk_1, (\_, pcc_1, \_, \_)) \approx_{[\overline{c}]} \mathtt{push}(stk_2, (\_, pcc_2, \_, \_))
\end{aligned}
$$

Proof.
By unfolding the assumption using Definition 85, we obtain f.

Then, by unfolding the goal using Definition 84, we pick:
$f' := f \uplus \{\mathtt{length}(stk_1) + 1 \mapsto \mathtt{length}(stk_2) + 1\}$.

It thus remains to prove all of the following:

1. Domain of f' is exhaustive of $\overline{c}$ call sites in $\mathtt{push}(stk_1, (\_, pcc_1, \_, \_))$:
   $(\mathsf{dom}(f') = \{i \in \mathsf{dom}(\mathtt{push}(stk_1, (\_, pcc_1, \_, \_))) \mid \mathtt{push}(stk_1, (\_, pcc_1, \_, \_))(i).\mathsf{pcc} \subseteq \mathsf{dom}(\overline{c}.\mathcal{M}_c)\} \uplus \{-1, \mathtt{length}(\mathtt{push}(stk_1, (\_, pcc_1, \_, \_)))\})$.

   Immediate by choice of f' after noticing the corresponding assumption about f, the assumption about pcc_1, and that $\mathtt{length}(\mathtt{push}(stk_1, (\_, pcc_1, \_, \_))) = \mathtt{length}(stk_1) + 1$.

2. Range of f' is exhaustive of $\bar{c}$ call sites in $\texttt{push}(stk_2, (\_, pcc_2, \_, \_))$:
   $(\mathsf{range}(f') = \{i \in \mathsf{dom}(\texttt{push}(stk_2, (\_, pcc_2, \_, \_))) \mid \texttt{push}(stk_2, (\_, pcc_2, \_, \_))(i).\mathsf{pcc} \subseteq \mathsf{dom}(\bar{c}.\mathcal{M}_c)\} \uplus \{-1, \texttt{length}(\texttt{push}(stk_2, (\_, pcc_2, \_, \_)))\})$

   Proof is similar to the previous subgoal.

3. f' is sentinel-value preserving:
   $(f'(-1) = -1 \ \wedge \ f'(\texttt{length}(stk_1) + 1) = \texttt{length}(stk_2 + 1))$.

   Immediate by the choice of f' and by the corresponding assumption about f.

4. f' is strictly monotone: $(\forall i, j.\ i > j \implies f'(i) > f'(j))$.

   Pick arbitrary $i, j \in \mathsf{dom}(f')$, and distinguish these cases:

   - **Case** $i, j \in \mathsf{dom}(f)$**:**
     Here, our goal is immediate by the corresponding assumption about f.
   - **Case** $i \notin \mathsf{dom}(f)$**:**
     Infer $i = \texttt{length}(stk_1) + 1$.
     Thus, infer $f'(i) = \texttt{length}(stk_2) + 1$.
     Thus, the goal becomes:
     $\forall j.\ j < \texttt{length}(stk_1) + 1 \implies \texttt{length}(stk_2) + 1 > f'(j)$
     But assuming $j < \texttt{length}(stk_1) + 1$ gives us $j \in \mathsf{dom}(f)$.
     Thus, $f'(j) = f(j)$.
     But then by the assumption about the range of f, we have our goal.
   - **Case** $j \notin \mathsf{dom}(f)$**:**
     Infer $j = \texttt{length}(stk_1) + 1$.
     Thus, goal follows vacuously because no index $i \in \mathsf{dom}(\texttt{push}(stk_1, \_))$ satisfies $i > \texttt{length}(stk_1) + 1$.

5. f' is compatible with stack-frame equality:
   $(\forall i \in \mathsf{dom}(f') \setminus \{-1, \texttt{length}(stk_1) + 1\}.\ f'(i) = j \implies$
   $\texttt{push}(stk_1, (\_, pcc_1, \_, \_))(i) = \texttt{push}(stk_2, (\_, pcc_2, \_, \_))(j))$

   Fix $i \in \mathsf{dom}(f') \setminus \{-1, \texttt{length}(stk_1) + 1\}$, and distinguish two cases:

   - **Case** $i \in \mathsf{dom}(f)$**:**
     Know by the assumption about $\mathsf{dom}(f)$ from unfolding Definition 85 that $i \in \mathsf{dom}(stk_1)$.
     Thus, our goal follows after instantiating the corresponding assumption about f (i.e., compatibility of f with stack-frame equality), and substitution using simple facts about $\texttt{push}$.
   - **Case** $i \notin \mathsf{dom}(f)$**:**
     By choice of f', and the condition on the fixed i, this case is impossible.

6. f' is successor-preserving:
   $(\forall i, j \in \mathsf{dom}(f').\ j = i + 1 \iff f'(j) = f'(i) + 1)$.

   Fix arbitrary $i, j \in \mathsf{dom}(f')$, and distinguish the following cases:

   - **Case** $i, j \in \mathsf{dom}(f)$**:**
     Here, the goal is immediate by the corresponding assumption about f (after noticing the choice of f').
   - **Case** $i \notin \mathsf{dom}(f)$**:**
     Know by the choice of f' that $i = \texttt{length}(stk_1) + 1$.

- $\Longrightarrow$:
  Here, know $j = \texttt{length}(stk_1) + 2$.
  Thus, our goal is immediate by deriving a contradiction to $j \in \mathsf{dom}(f')$.
- $\Longleftarrow$:
  Here, know $f'(j) = f'(\texttt{length}(stk_1) + 1) + 1$.
  Thus, know $f'(j) = \texttt{length}(stk_2) + 2$.
  This contradicts the subgoal proved earlier about $\mathsf{range}(f')$.

• **Case** $j \notin \mathsf{dom}(f)$**:**
Know by the choice of f' that $j = \texttt{length}(stk_1) + 1$.

- $\Longrightarrow$:
  Here, know $i = \texttt{length}(stk_1)$.
  By the specification of $\texttt{push}$ together with the subgoal proved above about $\mathsf{dom}(f')$, derive a contradiction to $i \in \mathsf{dom}(f')$.
  Thus, our goal is immediate.
- $\Longleftarrow$:
  Here, know $f'(i) = \texttt{length}(stk_2)$.
  By the specification of $\texttt{push}$ together with the subgoal proved above about $\mathsf{range}(f')$, derive a contradiction to $i \in \mathsf{dom}(f')$.

This concludes the proof of f' being successor-preserving.

This concludes the proof of Lemma 147. □

Lemma 148 (A silent action on strongly-similar states satisfies lock-step simulation)**.**

$$\begin{aligned}
&\forall \overline{c}, t_1, s_1, \varsigma_1, t_2, s_2, \varsigma_2, s_1', \varsigma_1'.\\
&\overline{c} \in \mathsf{range}(\llbracket\cdot\rrbracket) \ \wedge\\
&t_1 \ltimes \overline{c} \vdash_{exec} s_1 \ \wedge\\
&t_2 \ltimes \overline{c} \vdash_{exec} s_2 \ \wedge\\
&s_1.\mathsf{pcc} \subseteq \mathsf{dom}(\overline{c}.\mathcal{M}_c) \ \wedge\\
&s_1, \varsigma_1 \approx_{[\overline{c}]} s_2, \varsigma_2 \ \wedge\\
&s_1, \varsigma_1 \xrightarrow{\tau}_{[\overline{c}]} s_1', \varsigma_1'\\
&\Longrightarrow\\
&\exists s_2', \varsigma_2'.\\
&s_2, \varsigma_2 \xrightarrow{\tau}_{[\overline{c}]} s_2', \varsigma_2' \ \wedge\\
&s_1', \varsigma_1' \approx_{[\overline{c}]} s_2', \varsigma_2'
\end{aligned}$$

Proof. We fix arbitrary $\overline{c}, t_1, s_1, \varsigma_1, t_2, s_2, \varsigma_2, s_1', \varsigma_1'$, and assume the antecedent:

$$\begin{aligned}
&\overline{c} \in \mathsf{range}(\llbracket\cdot\rrbracket) \ \wedge\\
&t_1 \ltimes \overline{c} \vdash_{exec} s_1 \ \wedge\\
&t_2 \ltimes \overline{c} \vdash_{exec} s_2 \ \wedge\\
&s_1, \varsigma_1 \approx_{[\overline{c}]} s_2, \varsigma_2 \ \wedge\\
&s_1, \varsigma_1 \xrightarrow{\tau}_{[\overline{c}]} s_1', \varsigma_1'
\end{aligned} \tag{9}$$

From conjunct $s_1, \varsigma_1 \approx_{[\overline{c}]} s_2, \varsigma_2$ of Proposition (9) and by Definition 86, we have (after substituting

$s_1.\mathsf{pcc} \subseteq \mathsf{dom}(\bar{c}.\mathcal{M}_c)$ in Definition 82) the following assumptions:

$$\begin{aligned}
&s_1.\mathsf{pcc} \subseteq \mathsf{dom}(\bar{c}.\mathcal{M}_c) \wedge \\
&s_2.\mathsf{pcc} \subseteq \mathsf{dom}(\bar{c}.\mathcal{M}_c) \wedge \\
&\bigcup_{mid \in \mathsf{dom}(\bar{c}.imp)} \mathsf{reachable_addresses}(\{s_1.\mathsf{mstc}(mid), \bar{c}.imp(mid).\mathsf{ddc}\}, s_1.\mathcal{M}_d) = r \wedge \\
&\bigcup_{mid \in \mathsf{dom}(\bar{c}.imp)} \mathsf{reachable_addresses}(\{s_2.\mathsf{mstc}(mid), \bar{c}.imp(mid).\mathsf{ddc}\}, s_2.\mathcal{M}_d) = r \wedge \\
&s_1.\mathsf{ddc} = s_2.\mathsf{ddc} \wedge \\
&s_1.\mathsf{stc} = s_2.\mathsf{stc} \wedge \\
&s_1.\mathsf{pcc} = s_2.\mathsf{pcc} \wedge \\
&s_1.\mathsf{nalloc} = s_2.\mathsf{nalloc} \wedge \\
&s_1.stk \approx_{[\bar{c}]} s_2.stk \wedge \\
&s_1.\mathsf{mstc} \approx_{[\bar{c}]} s_2.\mathsf{mstc} \wedge \\
&\varsigma_1 = \varsigma_2 \wedge \\
&s_1.\mathcal{M}_d|_r = s_2.\mathcal{M}_d|_r
\end{aligned} \tag{10}$$

From $s_1.\mathsf{pcc} \subseteq \mathsf{dom}(\bar{c}.\mathcal{M}_c)$ and $s_2.\mathsf{pcc} \subseteq \mathsf{dom}(\bar{c}.\mathcal{M}_c)$ of Proposition (10), and by substitution in Proposition (9) after inversion using exec-state and valid-linking, we know:

$$s_1.\mathcal{M}_c(s_1.\mathsf{pcc}) = s_2.\mathcal{M}_c(s_2.\mathsf{pcc}) \tag{11}$$

Our goal $\exists s_2', \varsigma_2'.\ s_2, \varsigma_2 \xrightarrow{\tau}_{[\bar{c}]} s_2', \varsigma_2' \ \wedge\ s_1', \varsigma_1' \approx_{[\bar{c}]} s_2', \varsigma_2'$ consists by unfolding it using Definition 86 then Definition 82 of the following subgoals:

$$\begin{aligned}
&\exists s_2', \varsigma_2'.\ s_2, \varsigma_2 \xrightarrow{\tau}_{[\bar{c}]} s_2', \varsigma_2' \wedge \\
&s_1'.\mathsf{pcc} \subseteq \mathsf{dom}(\bar{c}.\mathcal{M}_c) \wedge \\
&s_2'.\mathsf{pcc} \subseteq \mathsf{dom}(\bar{c}.\mathcal{M}_c) \wedge \\
&\bigcup_{mid \in \mathsf{dom}(\bar{c}.imp)} \mathsf{reachable_addresses}(\{s_1'.\mathsf{mstc}(mid), \bar{c}.imp(mid).\mathsf{ddc}\}, s_1'.\mathcal{M}_d) = r \wedge \\
&\bigcup_{mid \in \mathsf{dom}(\bar{c}.imp)} \mathsf{reachable_addresses}(\{s_2'.\mathsf{mstc}(mid), \bar{c}.imp(mid).\mathsf{ddc}\}, s_2'.\mathcal{M}_d) = r \wedge \\
&s_1'.\mathsf{ddc} = s_2'.\mathsf{ddc} \wedge \\
&s_1'.\mathsf{stc} = s_2'.\mathsf{stc} \wedge \\
&s_1'.\mathsf{pcc} = s_2'.\mathsf{pcc} \wedge \\
&s_1'.\mathsf{nalloc} = s_2'.\mathsf{nalloc} \wedge \\
&s_1'.stk \approx_{[\bar{c}]} s_2'.stk \wedge \\
&s_1'.\mathsf{mstc} \approx_{[\bar{c}]} s_2'.\mathsf{mstc} \wedge \\
&\varsigma_1' = \varsigma_2' \wedge \\
&s_1'.\mathcal{M}_d|_r = s_2'.\mathcal{M}_d|_r
\end{aligned}$$

Notice that subgoals
$s_1'.\mathsf{pcc} \subseteq \mathsf{dom}(\bar{c}.\mathcal{M}_c)$ and $s_2'.\mathsf{pcc} \subseteq \mathsf{dom}(\bar{c}.\mathcal{M}_c)$
follow by Lemma 108 from respectively
the assumption $s_1, \varsigma_1 \xrightarrow{\tau}_{[\bar{c}]} s_1', \varsigma_1'$

and the subgoal $s_2, \varsigma_2 \xrightarrow{\tau}_{[\bar{c}]} s_2', \varsigma_2'$.

We prove the remaining subgoals by considering all the possible cases of the rule $s_1, \varsigma_1 \xrightarrow{\tau}_{[\bar{c}]} s_1', \varsigma_1'$ of Proposition (9):

1. **Case assign-silent:**

   - We obtain the precondition $s_1.\mathcal{M}_c(s_1.\mathsf{pcc}) = \mathsf{Assign}\ \mathcal{E}_l\ \mathcal{E}_r$, so by Proposition (11), we have $s_2.\mathcal{M}_c(s_2.\mathsf{pcc}) = \mathsf{Assign}\ \mathcal{E}_l\ \mathcal{E}_r$. So, the only rule possibly-applicable to $s_2, \varsigma_2 \xrightarrow{\lambda'}_{[\bar{c}]} s_2', \varsigma_2'$ is assign-silent. So, if λ' exists, then $\lambda' = \tau$.
   - **Now, we show that indeed s_2', ς_2' exist by showing that $s_2 \to s_2'$ using rule assign.**
     - By Lemma 138, and given $\mathcal{E}_l, s_1.\mathcal{M}_d, s_1.\mathsf{ddc}, s_1.\mathsf{stc}, s_1.\mathsf{pcc} \Downarrow c_1$ (which we do have by inversion), we have that $\mathcal{E}_l, s_2.\mathcal{M}_d, s_2.\mathsf{ddc}, s_2.\mathsf{stc}, s_2.\mathsf{pcc} \Downarrow c_1$. Also by Lemma 138, and given $\mathcal{E}_r, s_1.\mathcal{M}_d, s_1.\mathsf{ddc}, s_1.\mathsf{stc}, s_1.\mathsf{pcc} \Downarrow v_1$ (which we do have by inversion), we have that $\mathcal{E}_r, s_2.\mathcal{M}_d, s_2.\mathsf{ddc}, s_2.\mathsf{stc}, s_2.\mathsf{pcc} \Downarrow v_1$.
     - The preconditions on $s_2.\mathsf{pcc}$ and on $s_2.\mathsf{stc}$ then follow by substitution using respectively conjuncts $s_1.\mathsf{pcc} = s_2.\mathsf{pcc}$ and $s_1.\mathsf{stc} = s_2.\mathsf{stc}$ of Proposition (10).
     - Thus, we can now conclude that $s_2 \to s_2'$ since all the preconditions of rule assign hold.
     - Thus, by rule assign-silent, we have the first conjunct of our goal: $\exists s_2', \varsigma_2'.\ s_2, \varsigma_2 \xrightarrow{\tau}_{[\bar{c}]} s_2', \varsigma_2'$.
   - **We show the remaining subgoals**:
     - We observe from rule assign that $s_2'.\mathsf{ddc} = s_2.\mathsf{ddc}$, which by Proposition (10) gives $s_2'.\mathsf{ddc} = s_1.\mathsf{ddc}$, which by rule assign gives us $s_2'.\mathsf{ddc} = s_1'.\mathsf{ddc}$
     - A similar argument shows that $s_2'.stk = s_1'.stk$, $s_2'.\mathsf{mstc} = s_1'.\mathsf{mstc}$, $s_2'.\mathsf{stc} = s_1'.\mathsf{stc}$, and $s_2'.\mathsf{nalloc} = s_1'.\mathsf{nalloc}$.
     - Using the necessary preconditions $s_1'.\mathsf{pcc} = \mathtt{inc}(s_1.\mathsf{pcc}, 1)$ and $s_2'.\mathsf{pcc} = \mathtt{inc}(s_2.\mathsf{pcc}, 1)$ of rule assign, and by substitution using $s_1.\mathsf{pcc} = s_2.\mathsf{pcc}$ of Proposition (10), we get $s_2'.\mathsf{pcc} = s_1'.\mathsf{pcc}$.
     - Moreover, we have by rule assign-silent, that $\varsigma_2' = \varsigma_2$, which by Proposition (10) gives us that $\varsigma_2' = \varsigma_1$, which by rule assign-silent gives us $\varsigma_2' = \varsigma_1'$.
     - From the above, we have obtained the following conjuncts:
       * $s_1'.stk \approx_{[\bar{c}]} s_2'.stk \wedge s_1'.\mathsf{mstc} \approx_{[\bar{c}]} s_2'.\mathsf{mstc}$ by reflexivity of both the $\approx_{[\bar{c}]}$ overloaded relations after substituting from $s_1'.stk = s_2'.stk$, and $s_1'.\mathsf{mstc} = s_2'.\mathsf{mstc}$ respectively.
       * $s_1'.\mathsf{ddc} = s_2'.\mathsf{ddc} \wedge s_1'.\mathsf{stc} = s_2'.\mathsf{stc} \wedge s_1'.\mathsf{pcc} = s_2'.\mathsf{pcc} \wedge s_1'.\mathsf{nalloc} = s_2'.\mathsf{nalloc} \wedge \varsigma_1' = \varsigma_2'$ which we obtained successively by the arguments detailed above.
     - **Thus, it remains to show that $r' = \rho_{[\bar{c}]}(s_1', \varsigma_1') = \rho_{[\bar{c}]}(s_2', \varsigma_2')$ and $s_1'.\mathcal{M}_d|_{r'} = s_2'.\mathcal{M}_d|_{r'}$.**
     - We show that (S1'-PCC-SUBSET-C):
       $s_1'.\mathsf{pcc} \subseteq \mathsf{dom}(\bar{c}.\mathcal{M}_c)$
       To prove this, we apply Lemma 108 obtaining subgoals that are provable by the assumptions.
       From (S1'-PCC-SUBSET-C), we obtain by substitution using the previously proven subgoals:
       (S2'-PCC-SUBSET-C):
       $s_2'.\mathsf{pcc} \subseteq \mathsf{dom}(\bar{c}.\mathcal{M}_c)$
       Now, by substituting (S1'-PCC-SUBSET-C), and (S2'-PCC-SUBSET-C) in our goal after unfolding it using Definition 82, our goal becomes:
       $$\bigcup_{mid \in \mathsf{dom}(\bar{c}.imp)} \mathsf{reachable\_addresses}(\{s_1'.\mathsf{mstc}(mid), \bar{c}.imp(mid).\mathsf{ddc}\}, s_1'.\mathcal{M}_d) =$$
       $$\bigcup_{mid \in \mathsf{dom}(\bar{c}.imp)} \mathsf{reachable\_addresses}(\{s_2'.\mathsf{mstc}(mid), \bar{c}.imp(mid).\mathsf{ddc}\}, s_2'.\mathcal{M}_d)$$

By additivity of $\mathsf{reachable\_addresses}$ (Lemma 18), it suffices to show that:
$\mathsf{reachable\_addresses}(\bigcup_{mid \in \mathsf{dom}(\bar{c}.imp)} \{s_1'.\mathsf{mstc}(mid), \bar{c}.imp(mid).\mathsf{ddc}\}, s_1'.\mathcal{M}_d) =$
$\mathsf{reachable\_addresses}(\bigcup_{mid \in \mathsf{dom}(\bar{c}.imp)} \{s_2'.\mathsf{mstc}(mid), \bar{c}.imp(mid).\mathsf{ddc}\}, s_2'.\mathcal{M}_d)$
By conjunct $s_1'.\mathsf{mstc} \approx_{[\bar{c}]} s_2'.\mathsf{mstc}$ that we already proved above, it suffices to show that:
$\mathsf{reachable\_addresses}(\bigcup_{mid \in \mathsf{dom}(\bar{c}.imp)} \{s_1'.\mathsf{mstc}(mid), \bar{c}.imp(mid).\mathsf{ddc}\}, s_1'.\mathcal{M}_d) =$
$\mathsf{reachable\_addresses}(\bigcup_{mid \in \mathsf{dom}(\bar{c}.imp)} \{s_1'.\mathsf{mstc}(mid), \bar{c}.imp(mid).\mathsf{ddc}\}, s_2'.\mathcal{M}_d)$.

– So, we would like to use Lemma 29 about preservation of reachability equivalence with the instantiation $C := \bigcup_{mid \in \mathsf{dom}(\bar{c}.imp)} \{s_1'.\mathsf{mstc}(mid), \bar{c}.imp(mid).\mathsf{ddc}\}$, but we have first to satisfy the premise: $C, s_1.\mathcal{M}_d \vDash v \vee v \notin \{\delta\} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$.
We know $\{s_1'.\mathsf{stc}, s_1'.\mathsf{ddc}\}, s_1.\mathcal{M}_d \vDash v \vee v \notin \{\delta\} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$.
The latter follows immediately by Lemma 25 about completeness of $\mathsf{reachable\_addresses}$, and by simplifying Definition 23 of $\{s_1'.\mathsf{stc}, s_1'.\mathsf{ddc}\}, s_1.\mathcal{M}_d \vDash v$.
(Note that the premises of Lemma 25 are satisfied by conjunct $t_1 \ltimes \bar{c} \vdash_{exec} s_1$ of Proposition (9).)
By Lemma 27,
we thus have the premise $C, s_1.\mathcal{M}_d \vDash v \vee v \notin \{\delta\} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$ for Lemma 29.

– So, now we can use Lemma 29 which gives us (**):
$\mathsf{reachable\_addresses}(\bigcup_{mid \in \mathsf{dom}(\bar{c}.imp)} \{s_1'.\mathsf{mstc}(mid), \bar{c}.imp(mid).\mathsf{ddc}\}, s_1'.\mathcal{M}_d) =$
$\mathsf{reachable\_addresses}(\bigcup_{mid \in \mathsf{dom}(\bar{c}.imp)} \{s_1'.\mathsf{mstc}(mid), \bar{c}.imp(mid).\mathsf{ddc}\}, s_2'.\mathcal{M}_d)$.
This was sufficient for proving the subgoal $r' = \rho_{[\bar{c}]}(s_1', \varsigma_1') = \rho_{[\bar{c}]}(s_2', \varsigma_2')$.

– Now, it remains to show the subgoal $s_1'.\mathcal{M}_d|_{r'} = s_2'.\mathcal{M}_d|_{r'}$.

– By the precondition $\vdash_\delta c_1$, we can apply Lemma 25 to conclude that $c_1.s + c_1.\mathit{off} \in r$.
Thus, by Definition 23, we have the premises for Lemma 38.
By Lemma 38, in order to show that $s_1'.\mathcal{M}_d|_{r'} = s_2'.\mathcal{M}_d|_{r'}$, it suffices to show that $s_1'.\mathcal{M}_d|_r = s_2'.\mathcal{M}_d|_r$.
We show that $\forall a \in r\ s_1'.\mathcal{M}_d(a) = s_2'.\mathcal{M}_d(a)$ by distinguishing two cases:

  * **Case** $a = c_1.s + c_1.\mathit{off}$**:**
  Here, address a is the one assigned in both reduction rules ($s_1 \to s_1'$ and $s_2 \to s_2'$).
  So, the preconditions $s_1'.\mathcal{M}_d = s_1.\mathcal{M}_d[c_1 \mapsto v_1]$ and $s_2'.\mathcal{M}_d = s_2.\mathcal{M}_d[c_1 \mapsto v_1]$ clearly show our goal in this case because they update this address with the same value v_1.
  * **Case** $a \neq c_1.s + c_1.\mathit{off}$**:**
  In this case, similarly to above, we obtain the preconditions $s_1'.\mathcal{M}_d = s_1.\mathcal{M}_d[c_1 \mapsto v_1]$ and $s_2'.\mathcal{M}_d = s_2.\mathcal{M}_d[c_1 \mapsto v_1]$ which show that in this case, the memories $s_1'.\mathcal{M}_d$ and $s_2'.\mathcal{M}_d$ at address a are not updated.
  So, our goal follows from the assumption $s_1.\mathcal{M}_d|_r = s_2.\mathcal{M}_d|_r$ of Proposition (10).

This concludes case assign-silent. Cases alloc-silent and jump-silent are not surprisingly different; a so-far-convinced reader may well skip them.

2. **Case alloc-silent:**

  - We obtain the precondition $s_1.\mathcal{M}_c(s_1.\mathsf{pcc}) = \mathsf{Alloc}\ \mathcal{E}_l\ \mathcal{E}_{size}$, so by Proposition (11), we have $s_2.\mathcal{M}_c(s_2.\mathsf{pcc}) = \mathsf{Alloc}\ \mathcal{E}_l\ \mathcal{E}_{size}$. So, the only rule possibly-applicable to $s_2, \varsigma_2 \xrightarrow{\lambda'}_{[\bar{c}]} s_2', \varsigma_2'$ is alloc-silent. So, if λ' exists, then $\lambda' = \tau$.

- **Now, it remains to show that it is indeed applicable (i.e., $\exists s_2', \varsigma_2'.\ s_2, \varsigma_2 \xrightarrow{\lambda}_{[\overline{c}]} s_2', \varsigma_2'$) and that $s_1', \varsigma_1' \approx_{[\overline{c}]} s_2', \varsigma_2'$.**
- We show that $s_2 \to s_2'$ for some s_2', and in particular that rule allocate is applicable.
- By Lemma 138, and given $\mathcal{E}_l, s_1.\mathcal{M}_d, s_1.\mathsf{ddc}, s_1.\mathsf{stc}, s_1.\mathsf{pcc} \Downarrow c_1$ (which we do have by inversion), we have that $\mathcal{E}_l, s_2.\mathcal{M}_d, s_2.\mathsf{ddc}, s_2.\mathsf{stc}, s_2.\mathsf{pcc} \Downarrow c_1$. Also by Lemma 138, and given $\mathcal{E}_{size}, s_1.\mathcal{M}_d, s_1.\mathsf{ddc}, s_1.\mathsf{stc}, s_1.\mathsf{pcc} \Downarrow v_1$ (which we do have by inversion), we have that $\mathcal{E}_{size}, s_2.\mathcal{M}_d, s_2.\mathsf{ddc}, s_2.\mathsf{stc}, s_2.\mathsf{pcc} \Downarrow v_1$.
- The preconditions on $s_2.\mathsf{pcc}$ and on $s_2.\mathsf{nalloc}$ then follow by substitution using respectively conjuncts $s_1.\mathsf{pcc} = s_2.\mathsf{pcc}$ and $s_1.\mathsf{nalloc} = s_2.\mathsf{nalloc}$ of Proposition (10).
- Thus, we can now conclude that $s_2 \to s_2'$ since all the preconditions of rule allocate hold.
- Moreover, by the precondition $\vdash_\delta c_1$, we can apply Lemma 25 to conclude that $c_1.s + c_1.\mathit{off} \in r$.
- We observe from rule allocate that $s_2'.\mathsf{ddc} = s_2.\mathsf{ddc}$, which by Proposition (10) gives $s_2'.\mathsf{ddc} = s_1.\mathsf{ddc}$, which by rule allocate gives us $s_2'.\mathsf{ddc} = s_1'.\mathsf{ddc}$
- A similar argument shows that $s_2'.\mathit{stk} = s_1'.\mathit{stk}$, $s_2'.\mathsf{mstc} = s_1'.\mathsf{mstc}$ (thus, implying the desired stack and stack-capability-map similarities (definitions 83 and 84) respectively), and $s_2'.\mathsf{stc} = s_1'.\mathsf{stc}$.
- Using the necessary preconditions $s_1'.\mathsf{pcc} = \mathtt{inc}(s_1.\mathsf{pcc}, 1)$ and $s_2'.\mathsf{pcc} = \mathtt{inc}(s_2.\mathsf{pcc}, 1)$ of rule allocate, and by substitution using $s_1.\mathsf{pcc} = s_2.\mathsf{pcc}$ of Proposition (10), we get $s_2'.\mathsf{pcc} = s_1'.\mathsf{pcc}$.
- Also, we have that $s_2'.\mathsf{nalloc} = s_1'.\mathsf{nalloc}$ by substituting conjunct $s_1.\mathsf{nalloc} = s_2.\mathsf{nalloc}$ of Proposition (10) in the preconditions $s_2'.\mathsf{nalloc} = s_2.\mathsf{nalloc} - v_1$ and $s_1'.\mathsf{nalloc} = s_1.\mathsf{nalloc} - v_1$, where the same v_1 appears in both expressions due to the equal-evaluation that is shown above of the expression $\mathcal{E}_{size}$.
- Moreover, we have by rule alloc-silent, that $\varsigma_2' = \varsigma_2$, which by Proposition (10) gives us that $\varsigma_2' = \varsigma_1$, which by rule alloc-silent gives us $\varsigma_2' = \varsigma_1'$.
- Next, we show that $r' = \rho_{[\overline{c}]}(s_1', \varsigma_1') = \rho_{[\overline{c}]}(s_2', \varsigma_2')$ by the same argument as in case assign. We avoid repetition.
- Now, it remains to show that $s_1'.\mathcal{M}_d|_{r'} = s_2'.\mathcal{M}_d|_{r'}$.
- By Lemma 40, it suffices to show that $s_1'.\mathcal{M}_d|_r = s_2'.\mathcal{M}_d|_r$.
  We show that $\forall a \in r\ s_1'.\mathcal{M}_d(a) = s_2'.\mathcal{M}_d(a)$ by distinguishing three cases that are exhaustive (we do not prove that they are mutually exclusive because that is not needed, although we believe them to be mutually exclusive):
  - **Case** $a = c_1.s + c_1.\mathit{off}$**:**
    Here, address a is updated in both reduction rules ($s_1 \to s_1'$ and $s_2 \to s_2'$). So, the preconditions $s_1'.\mathcal{M}_d(c_1) = (\delta, s_1.\mathsf{nalloc} - v_1, s_1.\mathsf{nalloc}, 0)$ and $s_2'.\mathcal{M}_d(c_1) = (\delta, s_2.\mathsf{nalloc} - v_1, s_2.\mathsf{nalloc}, 0)$ show our goal in this case because by substitution using conjunct $s_1.\mathsf{nalloc} = s_2.\mathsf{nalloc}$ of Proposition (10), they update address a with the same value.
  - **Case** $a \in [s_2.\mathsf{nalloc} - v_1, s_2.\mathsf{nalloc})$**:**
    Here, similarly to the previous case, address a is one that is assigned in both reduction rules ($s_1 \to s_1'$ and $s_2 \to s_2'$ because $s_2.\mathsf{nalloc} = s_1.\mathsf{nalloc}$ by Proposition (10)). So, the updated value 0 of both $s_1'.\mathcal{M}_d(a)$ and $s_2'.\mathcal{M}_d(a)$ is the same, so we have our goal.
  - **Case** $a \neq c_1.s + c_1.\mathit{off} \land a \notin [s_2.\mathsf{nalloc} - \mathsf{v}_1, s_2.\mathsf{nalloc})$**:**
    In this case, similarly to above, we obtain the preconditions $s_1'.\mathcal{M}_d = s_1.\mathcal{M}_d[c_1 \mapsto v_1]$ and $s_2'.\mathcal{M}_d = s_2.\mathcal{M}_d[c_1 \mapsto v_1]$ which show that in this case, the memories $s_1'.\mathcal{M}_d$ and $s_2'.\mathcal{M}_d$ at address a are not updated.
    So, our goal follows from the assumption $s_1.\mathcal{M}_d|_r = s_2.\mathcal{M}_d|_r$ of Proposition (10).

This concludes case alloc-silent.

3. **Case jump-silent:**

   - We obtain the precondition $s_1.\mathcal{M}_c(s_1.\mathsf{pcc}) = \mathsf{JumpIfZero}\ \mathcal{E}_{cond}\ \mathcal{E}_{cap}$, so by Proposition (11), we have $s_2.\mathcal{M}_c(s_2.\mathsf{pcc}) = \mathsf{JumpIfZero}\ \mathcal{E}_{cond}\ \mathcal{E}_{cap}$. So, the only rule possibly-applicable to $s_2, \varsigma_2 \xrightarrow{\lambda'}_{[\bar{c}]} s_2', \varsigma_2'$ is jump-silent. So, if λ' exists, then $\lambda' = \lambda = \tau$.
   - **Now, it remains to show that it is indeed applicable (i.e., $\exists s_2', \varsigma_2'.\ s_2, \varsigma_2 \xrightarrow{\lambda}_{[\bar{c}]} s_2', \varsigma_2'$) and that $s_1', \varsigma_1' \approx_{[\bar{c}]} s_2', \varsigma_2'$.**
   - We show that $s_2 \rightarrow s_2'$ for some s_2', and in particular that either rule jump1 or jump0 is applicable.
   - For that, we distinguish the two possible cases for $s_1 \rightarrow s_1'$:
     - **Case jump1:**
       * By Lemma 138, and given $\mathcal{E}_{cond}, s_1.\mathcal{M}_d, s_1.\mathsf{ddc}, s_1.\mathsf{stc}, s_1.\mathsf{pcc} \Downarrow v_1$ (which we do have by inversion), we have that $\mathcal{E}_{cond}, s_2.\mathcal{M}_d, s_2.\mathsf{ddc}, s_2.\mathsf{stc}, s_2.\mathsf{pcc} \Downarrow v_1$.
       * The precondition on $s_2.\mathsf{pcc}$ then follows by substitution using conjunct $s_1.\mathsf{pcc} = s_2.\mathsf{pcc}$ of Proposition (10) and the precondition on v_1 still holds as well because $\mathcal{E}_{cond}$ evaluates to the same v_1 as in rule $s_1 \rightarrow s_1'$ as shown above.
       * Thus, we can now conclude that $s_2 \rightarrow s_2'$ since all the preconditions of rule jump1 hold.
       * The similarities $s_1'.stk \approx_{[\bar{c}]} s_2'.stk \wedge s_1'.\mathsf{mstc} \approx_{[\bar{c}]} s_2'.\mathsf{mstc}$ hold by substitution using the corresponding equalities in Proposition (10).
       * Also, we have that all the required equalities (namely, $\varsigma_1' = \varsigma_2'$, $s_1'.\mathcal{M}_d|_{r'} = s_2'.\mathcal{M}_d|_{r'}$, and $s_1'.\mathsf{ddc} = s_2'.\mathsf{ddc}$) follow from the corresponding ones in Proposition (10) by noticing that $s_2'.\mathcal{M}_d = s_2.\mathcal{M}_d$ and $s_1.\mathcal{M}_d = s_1'.\mathcal{M}_d$ and similarly for ς_2', $s_2'.\mathsf{ddc}$, $s_2'.\mathsf{stc}$, and $s_2'.\mathsf{nalloc}$.
       * So all conjuncts of our goal are proved.
     - **Case jump0:**
       This case is exactly the same as jump1, except that $s_2 \rightarrow s_2'$ holds by rule jump0.

   This concludes case jump-silent.

4. **Case cinvoke-silent-compiled:**

   - We obtain the precondition $s_1.\mathcal{M}_c(s_1.\mathsf{pcc}) = \mathsf{Cinvoke}\ mid\ fid\ \bar{e}$, so by Proposition (11), we have $s_2.\mathcal{M}_c(s_2.\mathsf{pcc}) = \mathsf{Cinvoke}\ mid\ fid\ \bar{e}$.

     Also, by $s_1.\mathsf{pcc} = s_2.\mathsf{pcc}$ of Proposition (10), we know that the precondition $s_2.\mathsf{pcc} \in \mathsf{dom}(\bar{c}.\mathcal{M}_c)$ holds.

     Thus, this, together with the precondition $mid \in \mathsf{dom}(\bar{c}.imp)$ give us that the only rule possibly-applicable to $s_2, \varsigma_2 \xrightarrow{\lambda'}_{[\bar{c}]} s_2', \varsigma_2'$ is cinvoke-silent-compiled. So, if λ' exists, then $\lambda' = \lambda = \tau$.
   - **Now, it remains to show that it is indeed applicable (i.e., $\exists s_2', \varsigma_2'.\ s_2, \varsigma_2 \xrightarrow{\lambda}_{[\bar{c}]} s_2', \varsigma_2'$) and that $s_1', \varsigma_1' \approx_{[\bar{c}]} s_2', \varsigma_2'$.**
   - We show that $s_2 \rightarrow s_2'$ for some s_2', and in particular that rule cinvoke is applicable.
   - We obtain the preconditions $s_1.\phi(mid, fid) = (nArgs, nLocal)$, and $(c, d, offs) = s_1.imp(mid)$.

     So, by Lemma 2, and by our earlier statement $s_2.\mathcal{M}_c(s_2.\mathsf{pcc}) = \mathsf{Cinvoke}\ mid\ fid\ \bar{e}$, we notice that we have $s_2.\phi(mid, fid) = (nArgs, nLocal)$, and $(c, d, offs) = s_2.imp(mid)$.

     This gives us the equalities $s_1'.\mathsf{ddc} = s_2'.\mathsf{ddc}$ and $s_1'.\mathsf{stc} = s_2'.\mathsf{stc}$, and $s_1'.\mathsf{pcc} = s_2'.\mathsf{pcc}$ of our goal.

- We also conclude that expression evaluation of the arguments in state s_2 gives the same values as evaluation in state s_1.
  I.e., given $\overline{e}(i), s_1.\mathcal{M}_d, s_1.\mathsf{ddc}, s_1.\mathsf{stc}, s_1.\mathsf{pcc} \Downarrow \overline{v}(i) \forall i \in [0, nArgs)$ (which we get by inverting $s_1 \succ_{\approx} s_1'$ using cinvoke-aux), we have by Lemma 138 that
  $\overline{e}(i), s_2.\mathcal{M}_d, s_2.\mathsf{ddc}, s_2.\mathsf{stc}, s_2.\mathsf{pcc} \Downarrow \overline{v}(i) \forall i \in [0, nArgs)$.
  This, consequently, gives us that $s_2'.\mathcal{M}_d|_r = s_1'.\mathcal{M}_d|_r$ by case distinction on the updated vs. non-updated locations and substitution in both cases.
  Similarly to case assign-silent, this suffices to prove subgoal $s_2'.\mathcal{M}_d|_{r'} = s_1'.\mathcal{M}_d|_{r'}$.
- We obtain subgoal $s_1'.\mathsf{mstc} \approx_{[\overline{c}]} s_2'.\mathsf{mstc}$ by Lemma 134.
- We would like to prove $s_1'.stk \approx_{[\overline{c}]} s_2'.stk$.
  This is immediate by Lemma 144
- The equalities $s_1'.\mathsf{nalloc} = s_2'.\mathsf{nalloc}$ and $\varsigma_1' = \varsigma_2'$ follow immediately by substitution and the equalities of Proposition (10).
- All subgoals are proved.

5. **Case cinvoke-silent-context:**

   We obtain the precondition $s_1.\mathsf{pcc} \not\subseteq \mathsf{dom}(\overline{c}.\mathcal{M}_c)$, which immediately contradicts conjunct $s_1.\mathsf{pcc} \subseteq \mathsf{dom}(\overline{c}.\mathcal{M}_c)$ of Proposition (9).

   So, any goal is provable.

6. **Case creturn-silent-compiled:**

   - We obtain the precondition $s_1.\mathcal{M}_c(s_1.\mathsf{pcc}) = \mathsf{Creturn}$, so by Proposition (11), we have $s_2.\mathcal{M}_c(s_2.\mathsf{pcc}) = \mathsf{Creturn}$.
     Also, by $s_1.\mathsf{pcc} = s_2.\mathsf{pcc}$ of Proposition (10), we know that the precondition $s_2.\mathsf{pcc} \in \mathsf{dom}(\overline{c}.\mathcal{M}_c)$ holds.
     Now, we have the precondition $s_1'.\mathsf{pcc} \in \mathsf{dom}(\overline{c}.\mathcal{M}_c)$, and we argue that $s_2'.\mathsf{pcc} \in \mathsf{dom}(\overline{c}.\mathcal{M}_c)$ holds. But first, we show s_2' exists.
     In particular, we argue that $s_2 \to s_2'$ using rule creturn.
   - For that, we need to ensure that the precondition
     $s_2'.stk, (s_2'.\mathsf{ddc}, s_2'.\mathsf{pcc}, \_, \_) = \mathsf{pop}(s_2.stk)$ holds, i.e., we need to show that the computation $\mathsf{pop}(s_2.stk)$ is not stuck.
   - We know by $s_1 \to s_1'$ that $s_1.stk \neq \mathtt{nil}$.
   - For showing non-stuckness of $\mathsf{pop}(s_2.stk)$, we use conjunct $s_1.stk \approx_{[\overline{c}]} s_2.stk$ of Proposition (10), where by unfolding Definition 84, we have by $s_1'.\mathsf{pcc} \in \mathsf{dom}(\overline{c}.\mathcal{M}_c)$ that
     $\mathsf{top}(s_1.stk) = \mathsf{top}(s_2.stk) = (s_1'.\mathsf{ddc}, s_1'.\mathsf{pcc}, \_, \_)$.
   - The above suffices to prove that $s_2 \to s_2'$ using rule creturn, and that $s_2'.\mathsf{ddc} = s_1'.\mathsf{ddc}$, $s_2'.\mathsf{stc} = s_1'.\mathsf{stc}$, and $s_2'.\mathsf{pcc} = s_1'.\mathsf{pcc}$.
   - It is also immediate by substitution and transitivity of equality that $s_2'.\mathsf{nalloc} = s_1'.\mathsf{nalloc}$
   - Thus, this, together with the precondition $s_1'.\mathsf{pcc} \in \mathsf{dom}(\overline{c}.\mathcal{M}_c)$ give us that $s_2'.\mathsf{pcc} \in \mathsf{dom}(\overline{c}.\mathcal{M}_c)$.
   - So, the only rule possibly-applicable to $s_2, \varsigma_2 \xrightarrow{\lambda'}_{[\overline{c}]} s_2', \varsigma_2'$ is creturn-silent-compiled. So $\lambda' = \lambda = \tau$.
   - And thus, we have $\varsigma_2' = \varsigma_1'$.
   - Thus, it remains to show that $s_1'.stk \approx_{[\overline{c}]} s_2'.stk$, $s_1'.\mathsf{mstc} \approx_{[\overline{c}]} s_2'.\mathsf{mstc}$, and $s_1'.\mathcal{M}_d|_{r'} = s_2'.\mathcal{M}_d|_{r'}$.
   - The former follows by obtaining from $s_1.stk \approx_{[\overline{c}]} s_2.stk$ the isomorphism f by unfolding Definition 84.
     This is immediate by instantiating Lemma 146.

- For $s_1'.\mathsf{mstc} \approx_{[\bar{c}]} s_2'.\mathsf{mstc}$ we notice that the definition of $\mathit{off}' = \mathit{off} - \mathit{nArgs} - \mathit{nLocal}$ is the same in both $s_1 \to s_1'$ and $s_2 \to s_2'$ (by in-turn the similarity of the definitions of off, nArgs and nLocal).

  And thus, by Lemma 134, we have that $s_1'.\mathsf{mstc} \approx_{[\bar{c}]} s_2'.\mathsf{mstc}$.
- Conjunct $s_1'.\mathcal{M}_d|_r = s_2'.\mathcal{M}_d|_r$ follows immediately by $s_1.\mathcal{M}_d|_r = s_2.\mathcal{M}_d|_r$ of Proposition (10) and substitution.

  Also, notice that $r = r'$. Thus, subgoal $s_2'.\mathcal{M}_d|_{r'} = s_1'.\mathcal{M}_d|_{r'}$ follows by substitution.
- This concludes our case.

7. **Case creturn-silent-context:**

   We obtain the precondition $s_1.\mathsf{pcc} \not\subseteq \mathsf{dom}(\bar{c}.\mathcal{M}_c)$, which immediately contradicts conjunct $s_1.\mathsf{pcc} \subseteq \mathsf{dom}(\bar{c}.\mathcal{M}_c)$ of Proposition (9).

   So, any goal is provable.

This concludes all cases for $s_1, \varsigma_1 \xrightarrow{\tau}_{[\bar{c}]} s_1', \varsigma_1'$, which concludes the proof of Lemma 148.

□

Corollary 11 (Star silent actions on strongly-similar states satisfy simulation)**.**

$$\begin{aligned}
&\forall \bar{c}, t_1, s_1, \varsigma_1, t_2, s_2, \varsigma_2, s_1', \varsigma_1'.\\
&\bar{c} \in \mathsf{range}(\llbracket\cdot\rrbracket) \;\wedge\\
&t_1 \ltimes \bar{c} \vdash_{exec} s_1 \;\wedge\\
&t_2 \ltimes \bar{c} \vdash_{exec} s_2 \;\wedge\\
&s_1, \varsigma_1 \approx_{[\bar{c}]} s_2, \varsigma_2 \;\wedge\\
&s_1, \varsigma_1 \xrightarrow{\tau^*}{}^{*}_{[\bar{c}]} s_1', \varsigma_1'\\
&\implies\\
&\exists s_2', \varsigma_2'.\\
&s_2, \varsigma_2 \xrightarrow{\tau^*}{}^{*}_{[\bar{c}]} s_2', \varsigma_2' \;\wedge\\
&s_1', \varsigma_1' \approx_{[\bar{c}]} s_2', \varsigma_2'
\end{aligned}$$

Proof. Follows from Lemma 148 and claim 14 and Corollary 2. □

Lemma 149 (Strong state-similarity determines non-silent output actions and is weakened by them)**.**

$$\begin{aligned}
&\forall \bar{c}, t_1, s_1, \varsigma, t_2, s_2, \varsigma, s_1', \varsigma'.\\
&t_1 \ltimes \bar{c} \vdash_{silent} s_1, \varsigma, c, r_1, na, \mathcal{M}_d \;\wedge\\
&t_2 \ltimes \bar{c} \vdash_{silent} s_2, \varsigma, c, r_2, na, \mathcal{M}_d \;\wedge\\
&s_1, \varsigma \approx_{[\bar{c}]} s_2, \varsigma \;\wedge\\
&s_1, \varsigma \xrightarrow{\lambda}_{[\bar{c}]} s_1', \varsigma' \;\wedge\\
&\lambda \in \dot{!}\\
&\implies\\
&\exists s_2'.\\
&s_2, \varsigma \xrightarrow{\lambda}_{[\bar{c}]} s_2', \varsigma' \;\wedge\\
&s_1', \varsigma' \sim_{[\bar{c}]} s_2', \varsigma'
\end{aligned}$$

Proof. We fix arbitrary $\overline{c}, t_1, s_1, \varsigma, t_2, s_2, s_1', \varsigma'$, and assume the antecedent:

$$
\begin{aligned}
& t_1 \ltimes \overline{c} \vdash_{exec} s_1 \;\wedge\; t_2 \ltimes \overline{c} \vdash_{exec} s_2 \\
& \wedge\; s_1.\mathsf{pcc} \in \mathsf{dom}(\overline{c}.\mathcal{M}_c) \\
& \wedge\; s_1, \varsigma \approx_{[\overline{c}]} s_2, \varsigma \;\wedge\; s_1, \varsigma \xrightarrow{\lambda}_{[\overline{c}]} s_1', \varsigma' \wedge \lambda \in \overset{\bullet}{!}
\end{aligned}
\tag{12}
$$

From conjunct $s_1, \varsigma \approx_{[\overline{c}]} s_2, \varsigma$ of Proposition (12) and by Definition 86, we have (after substituting $s_1.\mathsf{pcc} \in \mathsf{dom}(\overline{c}.\mathcal{M}_c)$ of Proposition (12) in Definition 82):

$$
\begin{aligned}
& r = \bigcup_{mid \in \mathsf{dom}(\overline{c}.imp)} \mathsf{reachable\_addresses}(\{s_1.\mathsf{mstc}(mid), \overline{c}.imp(mid).\mathsf{ddc}\}, s_1.\mathcal{M}_d) \\
& \wedge\; s_1.stk \approx_{[\overline{c}]} s_2.stk \wedge s_1.\mathsf{mstc} \approx_{[\overline{c}]} s_2.\mathsf{mstc} \\
& \wedge\; s_1.\mathsf{stc} = s_2.\mathsf{stc} \wedge s_1.\mathsf{pcc} = s_2.\mathsf{pcc} \wedge s_1.\mathsf{nalloc} = s_2.\mathsf{nalloc} \\
& \wedge\; s_1.imp = s_2.imp \wedge s_1.\phi = s_2.\phi \\
& \wedge\; s_1.\mathsf{ddc} = s_2.\mathsf{ddc} \wedge s_1.\mathcal{M}_d|_r = s_2.\mathcal{M}_d|_r \\
& \wedge\; \mathsf{dom}(s_1.\mathcal{M}_d) = \mathsf{dom}(s_2.\mathcal{M}_d)
\end{aligned}
\tag{13}
$$

By substituting $s_1.\mathsf{pcc} = s_2.\mathsf{pcc}$ of Proposition (13) in conjunct $s_1.\mathsf{pcc} \in \mathsf{dom}(\overline{c}.\mathcal{M}_c)$ of Proposition (12), we get:

$$
s_2.\mathsf{pcc} \in \mathsf{dom}(\overline{c}.\mathcal{M}_c) \tag{14}
$$

But from conjuncts $t_1 \ltimes \overline{c} \vdash_{exec} s_1 \;\wedge\; t_2 \ltimes \overline{c} \vdash_{exec} s_2$ of Proposition (12), we know by rules valid-linking and exec-state (after inversion using Silent-state invariant) that:

$$
s_1.\mathcal{M}_c = t_1.\mathcal{M}_c \uplus \overline{c}.\mathcal{M}_c \tag{15}
$$

and

$$
s_2.\mathcal{M}_c = t_2.\mathcal{M}_c \uplus \overline{c}.\mathcal{M}_c \tag{16}
$$

respectively.

So, we obtain that $s_1.\mathcal{M}_c(s_1.\mathsf{pcc}) = \overline{c}.\mathcal{M}_c(s_1.\mathsf{pcc})$ by Propositions (12) and (15);
thus $\overline{c}.\mathcal{M}_c(s_1.\mathsf{pcc}) = \overline{c}.\mathcal{M}_c(s_2.\mathsf{pcc})$ by $s_1.\mathsf{pcc} = s_2.\mathsf{pcc}$ of Proposition (13);
thus $\overline{c}.\mathcal{M}_c(s_2.\mathsf{pcc}) = s_2.\mathcal{M}_c(s_2.\mathsf{pcc})$ by Propositions (14) and (16);
thus by transitivity, we obtain:

$$
s_1.\mathcal{M}_c(s_1.\mathsf{pcc}) = s_2.\mathcal{M}_c(s_2.\mathsf{pcc}) \tag{17}
$$

We then show our goal $\exists s_2'.\; s_2, \varsigma \xrightarrow{\lambda}_{[\overline{c}]} s_2', \varsigma' \;\wedge\; s_1', \varsigma' \sim_{[\overline{c}]} s_2', \varsigma'$. The second conjunct unfolds by Definition 86 into:

$$
\begin{aligned}
r' = \rho_{[\overline{c}]}(s_1', \varsigma') = \rho_{[\overline{c}]}(s_2', \varsigma') \;\wedge\; & s_1'.stk \approx_{[\overline{c}]} s_2'.stk \wedge s_1'.\mathsf{mstc} \approx_{[\overline{c}]} s_2'.\mathsf{mstc} \\
\wedge\; & s_1'.imp = s_2'.imp \wedge s_1'.\phi = s_2'.\phi \wedge s_1'.\mathcal{M}_d|_{r'} = s_2'.\mathcal{M}_d|_{r'}
\end{aligned}
$$

The proof is by considering all the possible cases of the rule $s_1, \varsigma_1 \xrightarrow{\lambda}_{[\overline{c}]} s_1', \varsigma_1'$ subject to $\lambda \in \overset{\bullet}{!}$:

1. **Case cinvoke-compiled-to-context:**
   - In this case, we obtain the precondition $s_1.\mathcal{M}_c(s_1.\mathsf{pcc}) = \mathsf{Cinvoke}\ mid\ fid\ \overline{e}$ from which by Proposition (17), we know $s_2.\mathcal{M}_c(s_2.\mathsf{pcc}) = \mathsf{Cinvoke}\ mid\ fid\ \overline{e}$.
   - We also obtain the precondition $s_1 \succ_{\approx} s_1'$, and we would like to conclude $s_2 \succ_{\approx} s_2'$. So by rule cinvoke-aux, we want to show that all the preconditions on s_2 that are necessary for $s_2 \succ_{\approx} s_2'$ are satisfied.

- In particular, we have to verify that $(mid, fid) \in \mathsf{dom}(s_2.\phi)$, but this follows immediately from $(mid, fid) \in \mathsf{dom}(s_1.\phi)$ by conjunct $s_1.\phi = s_2.\phi$ of Proposition (13).
- We also have to verify that $mid \in \mathsf{dom}(s_2.imp)$, but this follows immediately from $mid \in \mathsf{dom}(s_1.imp)$ by conjunct $s_1.imp = s_2.imp$ of Proposition (13).
- We also have to verify that $mid \in \mathsf{dom}(s_2.\mathsf{mstc})$, but this follows immediately by inverting conjunct $\_ \vdash_{exec} s_2$ of Proposition (12) using rule exec-state and by knowing $mid \in \mathsf{dom}(s_2.imp)$ (the latter we just obtained).
- Finally, in order to show $s_2 \succ_{\approx} s_2'$, we need to verify that $\forall i \in [0, nArgs).\ \overline{e}(i), s_2.\mathcal{M}_d, s_2.\mathsf{ddc}, s_2.\mathsf{stc}, s_2.\mathsf{pcc} \Downarrow v_i$. This follows by Lemma 138, since we already know that: $\forall i \in [0, nArgs).\ \overline{e}(i), s_1.\mathcal{M}_d, s_1.\mathsf{ddc}, s_1.\mathsf{stc}, s_1.\mathsf{pcc} \Downarrow v_i$.
- Having satisfied all the possibly-unsatisfiable preconditions of cinvoke-aux, we know $\exists s_2'.\ s_2 \succ_{\approx} s_2'$.
- Conjuncts $s_1'.imp = s_2'.imp$ and $s_1'.\phi = s_2'.\phi$ of our goal follow by Lemma 2 and by substitution using the corresponding conjuncts of Proposition (13).
- Conjunct $s_1'.\mathsf{mstc} \approx_{[\overline{c}]} s_2'.\mathsf{mstc}$ follows immediately from $s_1.\mathsf{mstc} \approx_{[\overline{c}]} s_2.\mathsf{mstc}$ by the precondition $mid \notin \mathsf{dom}(\overline{c}.imp)$.
- Conjunct $s_1'.stk \approx_{[\overline{c}]} s_2'.stk$ follows by instantiating Lemma 144 then Lemma 143.
- For proving conjunct $\varsigma_1' = \varsigma_2'$ of our goal, we have the following obligation:
  $\mathsf{reachable\_addresses\_closure}(\varsigma_1 \cup r_1, s_1'.\mathcal{M}_d) =$
  $\mathsf{reachable\_addresses\_closure}(\varsigma_2 \cup r_2, s_2'.\mathcal{M}_d)$
  where:
  $r_1 = \mathsf{reachable\_addresses}(\{\overline{v}(i) \mid i \in [0, nArgs) \wedge \overline{v}(i) = (\delta, \_, \_, \_)\}, s_1'.\mathcal{M}_d)$, and
  $r_2 = \mathsf{reachable\_addresses}(\{\overline{v}(i) \mid i \in [0, nArgs) \wedge \overline{v}(i) = (\delta, \_, \_, \_)\}, s_2'.\mathcal{M}_d)$.
  (By Lemma 138, we were able to use the same values $\overline{v}$ for both $s_1 \to s_1'$ and $s_2 \to s_2'$.)
  - By conjunct $\varsigma_1 = \varsigma_2$ of Proposition (13), our subgoal becomes:
    $\mathsf{reachable\_addresses\_closure}(\varsigma_1 \cup r_1, s_1'.\mathcal{M}_d) =$
    $\mathsf{reachable\_addresses\_closure}(\varsigma_1 \cup r_2, s_2'.\mathcal{M}_d)$
  - Now, we argue that $r_1 = r_2$.
    We first notice that by Lemma 25, we have that:
    $\forall i \in [0, nArgs).\ \overline{v}(i) = (\delta, \sigma, e, \_) \implies [\sigma, e) \subseteq \mathsf{reachable\_addresses}(\{s_1.\mathsf{stc}, s_1.\mathsf{ddc}\}, s_1.\mathcal{M}_d)$.
    By rule cinvoke-aux, we would like to show that
    $\mathsf{reachable\_addresses}(\{\overline{v}(i) \mid i \in [0, nArgs) \wedge \overline{v}(i) = (\delta, \_, \_, \_)\},$
    $s_1.\mathcal{M}_d[off_1 + i \mapsto v_i\ \forall i \in [0, nArgs)][off_1 + nArgs + i \mapsto 0\ \forall i \in [0, nLocal)]) =$
    $\mathsf{reachable\_addresses}(\{\overline{v}(i) \mid i \in [0, nArgs) \wedge \overline{v}(i) = (\delta, \_, \_, \_)\},$
    $s_2.\mathcal{M}_d[off_2 + i \mapsto v_i\ \forall i \in [0, nArgs)][off_2 + nArgs + i \mapsto 0\ \forall i \in [0, nLocal)])$.
    (Sketch) By relying on inverting our assumptions (twice) using rule Silent-state invariant, we should obtain facts that enable us to simply apply Lemma 21 $nArgs + nLocal$-many times to each side of the goal, then we obtain the equivalent goal:
    $\mathsf{reachable\_addresses}(\{\overline{v}(i) \mid i \in [0, nArgs) \wedge \overline{v}(i) = (\delta, \_, \_, \_)\},\ s_1.\mathcal{M}_d) =$
    $\mathsf{reachable\_addresses}(\{\overline{v}(i) \mid i \in [0, nArgs) \wedge \overline{v}(i) = (\delta, \_, \_, \_)\},\ s_2.\mathcal{M}_d)$.
    (Sketch) By completeness of reachable addresses (Lemma 25), and again by invariance to unreachable memory (Lemma 21), we can satisfy this goal from $s_1.\mathcal{M}_d|_r = s_2.\mathcal{M}_d|_r$ of Proposition (13).
  - Moreover, observe that $\varsigma \cup r_1 \subseteq r$, and hence the same for $\varsigma \cup r_2$.
    Thus, our subgoal above follows by instantiating Lemma 29 using Proposition (13).
- For proving conjunct $\rho_{[\overline{c}]}(s_1', \varsigma') = \rho_{[\overline{c}]}(s_2', \varsigma')$ of our goal, we conclude from rule valid-linking that $s_1'.\mathsf{pcc} \notin \mathsf{dom}(\overline{c}.\mathcal{M}_c)$ and $s_2'.\mathsf{pcc} \notin \mathsf{dom}(\overline{c}.\mathcal{M}_c)$.

- This gives us by Definition 82 the following obligation:
$(\bigcup_{mid \in \mathsf{dom}(\overline{c}.imp)} \mathsf{reachable\_addresses}(\{s_1'.\mathsf{mstc}(mid), \overline{c}.imp(mid).\mathsf{ddc}\}, s_1'.\mathcal{M}_d)) \setminus \varsigma' =$
$(\bigcup_{mid \in \mathsf{dom}(\overline{c}.imp)} \mathsf{reachable\_addresses}(\{s_2'.\mathsf{mstc}(mid), \overline{c}.imp(mid).\mathsf{ddc}\}, s_2'.\mathcal{M}_d)) \setminus \varsigma'.$
  - By conjunct $s_1'.\mathsf{mstc} \approx_{[\overline{c}]} s_2'.\mathsf{mstc}$ of our goal that we already obtained above, and by noticing the condition $mid \in \mathsf{dom}(\overline{c}.imp)$ on the expressions $s_1'.\mathsf{mstc}(mid)$ and $s_2'.\mathsf{mstc}(mid)$, our subgoal is equivalent to:
  $(\bigcup_{mid \in \mathsf{dom}(\overline{c}.imp)} \mathsf{reachable\_addresses}(\{s_1'.\mathsf{mstc}(mid), \overline{c}.imp(mid).\mathsf{ddc}\}, s_1'.\mathcal{M}_d)) \setminus \varsigma' =$
  $(\bigcup_{mid \in \mathsf{dom}(\overline{c}.imp)} \mathsf{reachable\_addresses}(\{s_1'.\mathsf{mstc}(mid), \overline{c}.imp(mid).\mathsf{ddc}\}, s_2'.\mathcal{M}_d)) \setminus \varsigma'.$
  (Sketch) This should follow by easy substitutions after relying on the assumptions we get by inverting (twice) the antecedents using rule Silent-state invariant.

2. **Case creturn-to-context:** (Sketch) Similar to the previous case; except the subgoal about stack similarity relies on instantiating Lemma 145.

□

Lemma 150 (Option simulation: preservation of stack similarity by a silent action)**.**

$$
\begin{aligned}
&\forall \overline{c}, t_1, s_1, \varsigma_1, t_2, s_2, \varsigma_2, s_1', \varsigma_1'.\\
&t_1 \ltimes \overline{c} \vdash_{exec} s_1 \ \wedge\\
&t_2 \ltimes \overline{c} \vdash_{exec} s_2 \ \wedge\\
&s_1.\mathsf{pcc} \cap \mathsf{dom}(\overline{c}.\mathcal{M}_c) = \emptyset \ \wedge\\
&s_1.stk \sim_{[\overline{c}]} s_2.stk \ \wedge\\
&s_1, \varsigma_1 \xrightarrow{\tau}{}^{*}_{[\overline{c}]} s_1', \varsigma_1'\\
&\implies\\
&s_1'.stk \sim_{[\overline{c}]} s_2.stk
\end{aligned}
$$

Proof.
We assume the antecedents.

By unfolding the assumptions using Definition 85, we obtain f with:

We prove our goal by induction:

- **Case trace-closure-refl:**

  Here, the goal is immediate by assumption.

- **Case trace-closure-trans:**

  Here, we know:

  (S1-STAR-STEPS-S1”):
  $s_1, \varsigma_1 \xrightarrow{\tau}{}^{*}_{[\overline{c}]} s_1'', \varsigma_1''$

  (S1”-STEPS-S1’):
  $s_1'', \varsigma_1'' \xrightarrow{\tau}_{[\overline{c}]} s_1', \varsigma_1'$

  And by the induction hypothesis, we know:

  (S1”-STK-SIM-S2-STK):
  $s_1''.stk \approx_{[\overline{c}]} s_2.stk,$

By instantiation of Corollary 7 (twice), we know:
$s_1''.\mathsf{pcc} \cap \mathsf{dom}(\bar{c}.\mathcal{M}_c) = \emptyset$

and

$s_1'.\mathsf{pcc} \cap \mathsf{dom}(\bar{c}.\mathcal{M}_c) = \emptyset$

To prove our goal, we distinguish the following cases of (S1"-STEPS-S1'):

– **Case assign-silent,**

– **Case alloc-silent, and**

– **Case jump-silent:**

  In these cases, picking the obtained f suffices to prove our goal, and the frame relatedness condition holds by assumption after substitution using $s_1'.stk = s_1''.stk$.

– **Case cinvoke-silent-context:**

  Here, again we pick $f' := f$.

  We have $s_1'.stk = s_1''.stk{+}{+}[frame]$ where $frame.\mathsf{pcc} \not\subseteq \mathsf{dom}(\bar{c}.\mathcal{M}_c)$.

  Thus, we obtain by the required condition on $\mathsf{dom}(f')$ from Definition 85 the subgoal:

  $\mathtt{length}(s_1'.stk) - 1 \notin \mathsf{dom}(f')$

  That is immediate by the choice that $f' = f$ (unfolding Definition 85).

  The remaining conditions about f' from Definition 85 are also immediate by the choice that $f' = f$.

– **Case creturn-silent-context:**

  Here, again we pick $f' := f$.

  The subgoals from Definition 85 about $\mathsf{dom}(f')$ and $\mathsf{range}(f')$ are immediate by noticing that:

  $s_1''.stk = s_1'.stk{+}{+}[frame]$ where $frame.\mathsf{pcc} \not\subseteq \mathsf{dom}(\bar{c}.\mathcal{M}_c)$.

  The remaining conditions about f' from Definition 85 are immediate by the choice that $f' = f$.

The remaining cases are impossible.

This concludes the proof of Lemma 150. □

Lemma 151 (Option simulation: preservation of mstc similarity by a silent action)**.**

$$
\begin{aligned}
&\forall \bar{c}, t_1, s_1, \varsigma_1, t_2, s_2, \varsigma_2, s_1', \varsigma_1'.\\
&t_1 \ltimes \bar{c} \vdash_{exec} s_1 \;\wedge\\
&t_2 \ltimes \bar{c} \vdash_{exec} s_2 \;\wedge\\
&s_1.\mathsf{pcc} \cap \mathsf{dom}(\bar{c}.\mathcal{M}_c) = \emptyset \;\wedge\\
&s_1.\mathsf{mstc} \approx_{[\bar{c}]} s_2.\mathsf{mstc} \;\wedge\\
&s_1, \varsigma_1 \xrightarrow{\tau}{}^{*}_{[\bar{c}]} s_1', \varsigma_1'\\
&\Longrightarrow\\
&s_1'.\mathsf{mstc} \approx_{[\bar{c}]} s_2.\mathsf{mstc}
\end{aligned}
$$

Proof.
We assume the antecedents.

By unfolding the assumptions using Definition 83, we obtain:
$\forall mid.\ mid \in \mathsf{dom}(\bar{c}.imp) \implies s_1.\mathsf{mstc}(mid) = s_2.\mathsf{mstc}(mid)$

We prove our goal by induction:

- **Case trace-closure-refl:**

  Here, the goal is immediate by assumption.

- **Case trace-closure-trans:**

  Here, we know:

  (S1-STAR-STEPS-S1"):

  $s_1, \varsigma_1 \xrightarrow{\tau}{}^{*}_{[\bar{c}]} s_1'', \varsigma_1''$

  (S1"-STEPS-S1'):

  $s_1'', \varsigma_1'' \xrightarrow{\tau}_{[\bar{c}]} s_1', \varsigma_1'$

  And by the induction hypothesis, we know:

  (S1"-MSTC-SIM-S2-STK):

  $s_1''.\mathsf{mstc} \approx_{[\bar{c}]} s_2.\mathsf{mstc}$,

  By instantiation of Corollary 7 (twice), we know:

  $s_1''.\mathsf{pcc} \cap \mathsf{dom}(\bar{c}.\mathcal{M}_c) = \emptyset$

  and

  $s_1'.\mathsf{pcc} \cap \mathsf{dom}(\bar{c}.\mathcal{M}_c) = \emptyset$

  To prove our goal ($\forall mid.\ mid \in \mathsf{dom}(\bar{c}.imp) \implies s_1'.\mathsf{mstc}(mid) = s_2.\mathsf{mstc}(mid)$), we distinguish the following cases of (S1"-STEPS-S1'):

  - **Case assign-silent,**
  - **Case alloc-silent, and**
  - **Case jump-silent:**

    Here, our goal is immediate from the assumption after substitution using $s_1'.\mathsf{mstc} = s_1''.\mathsf{mstc}$.
  - **Case cinvoke-silent-context:**

    Here, by the preconditions and by inversion using cinvoke and cinvoke-aux we have:

    $s_1'.\mathsf{mstc} = s_1''.\mathsf{mstc}[mid \mapsto \_]$

    where

    $mid \notin \mathsf{dom}(\bar{c}.imp)$

    Thus, our goal follows from (S1"-MSTC-SIM-S2-STK).
  - **Case creturn-silent-context:**

    Here, by the preconditions and by inversion using creturn, we have:

    $s_1'.\mathsf{mstc} = s_1''.\mathsf{mstc}[modID \mapsto \_]$

    where

    $modID = \mathtt{top}(s_1''.stk).mid$

    It suffices for our goal to show:

    $modID \notin \mathsf{dom}(\bar{c}.imp)$

    By rule exec-state, it suffices to show the following two subgoals:

    * $t_1 \ltimes \bar{c} \vdash_{exec} s_1''$

      Here, apply Corollary 2 obtaining the following subgoals:

      · $t_1 \ltimes \bar{c} \vdash_{exec} s_1$

        Immediate by assumption.

      · $s_1 \to^* s_1''$

        Here, apply Claim 15 obtaining a subgoal that is immediate by (S1-STAR-STEPS-S1").

* $s_1''.\mathsf{pcc} \cap \mathsf{dom}(\overline{c}.\mathcal{M}_c) = \emptyset$
  This follows from the obtained preconditions of rule creturn-silent-context and by inversion of the previous subgoal using exec-state.

The remaining cases are impossible.

This concludes the proof of Lemma 151. □

Lemma 152 (Option simulation: preservation of weak similarity by a silent action)**.**

$$\begin{aligned}
&\forall \overline{c}, t_1, s_1, \varsigma_1, t_2, s_2, \varsigma_2, s_1', \varsigma_1', \mathcal{M}_{border}, \mathsf{na}_{border}, r_{t_1}, r_{t_2}.\\
&s_1.\mathsf{pcc} \cap \mathsf{dom}(\overline{c}.\mathcal{M}_c) = \emptyset \;\wedge\\
&t_1 \ltimes \overline{c} \vdash_{silent} s_1, \varsigma_1, _, r_{t_1}, \mathsf{na}_{border}, \mathcal{M}_{border} \;\wedge\\
&t_2 \ltimes \overline{c} \vdash_{silent} s_2, \varsigma_2, _, r_{t_2}, \mathsf{na}_{border}, \mathcal{M}_{border} \;\wedge\\
&s_1, \varsigma_1 \sim_{[\overline{c}], \mathsf{dom}(\mathcal{M}_{border})} s_2, \varsigma_2 \;\wedge\\
&s_1, \varsigma_1 \xrightarrow{\tau}{}^{*}_{[\overline{c}]} s_1', \varsigma_1'\\
&\Longrightarrow\\
&s_1', \varsigma_1' \sim_{[\overline{c}], \mathsf{dom}(\mathcal{M}_{border})} s_2, \varsigma_2
\end{aligned}$$

Proof.
We assume the antecedents.

By instantiating Lemma 157, we additionally obtain:
$t_1 \ltimes \overline{c} \vdash_{silent} s_1', \varsigma_1', \_, r_{t_1}, \mathsf{na}_{border}, \mathcal{M}_{border}$

By unfolding the assumptions using Definition 86, and by inversion using rule Silent-state invariant, we obtain:

EXEC-1
$t_1 \ltimes \overline{c} \vdash_{exec} s_1$

EXEC-2
$t_2 \ltimes \overline{c} \vdash_{exec} s_2$

TAU-STEPS-1
$s_1, \varsigma_1 \xrightarrow{\tau}{}^{*}_{[\overline{c}]} s_1', \varsigma_1'$

PCC-1-NOT-C
$s_1.\mathsf{pcc} \cap \mathsf{dom}(\overline{c}.\mathcal{M}_c) = \emptyset$

PCC-2-NOT-C
$s_2.\mathsf{pcc} \cap \mathsf{dom}(\overline{c}.\mathcal{M}_c) = \emptyset$

STK-SIM
$s_1.stk \sim_{[\overline{c}]} s_2.stk$

MSTC-SIM
$s_1.\mathsf{mstc} \approx_{[\overline{c}]} s_2.\mathsf{mstc}$

VARSIGMA-EQ
$\varsigma_1 = \varsigma_2$

PRIVATE-MEM-EQ
$s_1.\mathcal{M}_d|_{\mathsf{dom}(\mathcal{M}_{border})} = s_2.\mathcal{M}_d|_{\mathsf{dom}(\mathcal{M}_{border})}$

PRIVATE-MEM-S1-IS-MBORDER

$s_1.\mathcal{M}_d|_{\mathsf{dom}(\mathcal{M}_{border})} = \mathcal{M}_{border}$

PRIVATE-MEM-S1'-IS-MBORDER

$s_1'.\mathcal{M}_d|_{\mathsf{dom}(\mathcal{M}_{border})} = \mathcal{M}_{border}$

Our goal is $s_1', \varsigma_1' \sim_{[\overline{c}],\mathsf{dom}(\mathcal{M}_{border})} s_2, \varsigma_2$.
By unfolding it using Definition 86, we obtain the following subgoals:

- $s_1'.\mathsf{pcc} \cap \mathsf{dom}(\overline{c}.\mathcal{M}_c) = \emptyset$

  Follows by instantiating Corollary 7 using assumptions (**EXEC-1**) and (**TAU-STEPS-1**) then substitution using assumption (**PCC-1-NOT-C**).

- $s_1'.stk \sim_{[\overline{c}]} s_2.stk$

  Follows by applying Lemma 150 obtaining subgoals that are immediate by assumptions (**EXEC-1**), (**EXEC-2**), (**EXEC-2**), (**STK-SIM**), and (**PCC-1-NOT-C**).

- $s_1'.\mathsf{mstc} \approx_{[\overline{c}]} s_2.\mathsf{mstc}$

  Follows by applying Lemma 151 obtaining subgoals that are immediate by assumptions (**EXEC-1**), (**EXEC-2**), (**EXEC-2**), (**STK-SIM**), and (**PCC-1-NOT-C**).

- $\varsigma_1' = \varsigma_2$

  Follows by instantiating Claim 17 using assumption (**TAU-STEPS-1**) then substitution using assumption (**VARSIGMA-EQ**).

- $s_1'.\mathcal{M}_d|_{\mathsf{dom}(\mathcal{M}_{border})} = s_2.\mathcal{M}_d|_{\mathsf{dom}(\mathcal{M}_{border})}$

  Immediate by substitution using assumptions (**PRIVATE-MEM-S1-IS-MBORDER**) then (**PRIVATE-MEM-S1'-IS-MBORDER**) in assumption (**PRIVATE-MEM-EQ**).

This concludes the proof of Lemma 152.

□

Lemma 153 (Matching input actions retrieve back strong state-similarity)**.**

$$\begin{aligned}
&\forall \overline{c}, t_1, s_1, \varsigma, t_2, s_2, s_1', \varsigma', s_2', \mathcal{M}_{border}, \mathsf{na}_{border}, r_{t_1}, r_{t_2}.\\
&s_1.\mathsf{pcc} \cap \mathsf{dom}(\overline{c}.\mathcal{M}_c) = \emptyset \ \wedge\\
&t_1 \ltimes \overline{c} \vdash_{silent} s_1, \varsigma, _, r_{t_1}, \mathsf{na}_{border}, \mathcal{M}_{border} \ \wedge\\
&t_2 \ltimes \overline{c} \vdash_{silent} s_2, \varsigma, _, r_{t_2}, \mathsf{na}_{border}, \mathcal{M}_{border} \ \wedge\\
&s_1, \varsigma \sim_{[\overline{c}],\mathsf{dom}(\mathcal{M}_{border})} s_2, \varsigma \ \wedge\\
&s_1, \varsigma \xrightarrow{\lambda}_{[\overline{c}]} s_1', \varsigma' \ \wedge\\
&s_2, \varsigma \xrightarrow{\lambda}_{[\overline{c}]} s_2', \varsigma' \ \wedge\\
&\lambda \in \dot{?}\\
&\implies\\
&s_1', \varsigma' \approx_{[\overline{c}]} s_2', \varsigma'
\end{aligned}$$

Proof.
(Sketch)
After unfolding using Definition 86 and inversion using rule Silent-state invariant,
we proceed by case distinction on the step $s_1, \varsigma_1 \xrightarrow{\lambda}_{[\overline{c}]} s_1', \varsigma_1'$.

Figure 11: Border-state invariant for compiled programs

$$\frac{\begin{array}{c}\text{(Border-state invariant)}\\ t_{ctx} \ltimes \overline{c} = \lfloor t \rfloor \qquad \overline{c} \in \mathsf{range}(\llbracket\cdot\rrbracket) \qquad t \vdash_{exec} s \\ R_{ctx} = \bigcup_{mid \in \mathsf{dom}(t_{ctx}.imp)} \mathsf{reachable\_addresses}(\{s.\mathsf{mstc}(mid), t_{ctx}.imp(mid).\mathsf{ddc}\}, s.\mathcal{M}_d) \\ R_{\overline{c}} = \bigcup_{mid \in \mathsf{dom}(\overline{c}.imp)} \mathsf{reachable\_addresses}(\{s.\mathsf{mstc}(mid), \overline{c}.imp(mid).\mathsf{ddc}\}, s.\mathcal{M}_d) \\ \mathsf{mem}(\alpha(|\alpha|-1)) = s.\mathcal{M}_d|_{\varsigma} \qquad R_{ctx} \cap R_{\overline{c}} \subseteq \varsigma \qquad I_{ctx} = \mathsf{allocation\_intervals}(?, \alpha) \\ I_{\overline{c}} = \mathsf{allocation\_intervals}(!, \alpha) \qquad \forall a \in R_{ctx} \setminus \varsigma.\ s.\mathcal{M}_d(a) = (\delta, \sigma, e, \_) \implies \\ (\exists i \in I_{ctx}.\ [\sigma, e) \subseteq i \ \vee\ \exists a' \in \varsigma, idx \in [0, |\alpha|).\ [\sigma, e) \subseteq \mathsf{mem}(\alpha(idx))(a') \vee \\ \exists mid \in \mathsf{dom}(t_{ctx}.imp).\ [\sigma, e) \subseteq t_{ctx}.imp(mid).\mathsf{ddc} \vee \\ \exists mid \in \mathsf{dom}(t_{ctx}.imp).\ [\sigma, e) \subseteq t_{ctx}.\mathsf{mstc}(mid)) \\ \forall a \in R_{\overline{c}} \setminus \varsigma.\ s.\mathcal{M}_d(a) = (\delta, \sigma, e, \_) \implies \\ (\exists i \in I_{\overline{c}}.\ [\sigma, e) \subseteq i \ \vee\ \exists a' \in \varsigma, idx \in [0, |\alpha|).\ [\sigma, e) \subseteq \mathsf{mem}(\alpha(idx))(a') \vee \\ \exists mid \in \mathsf{dom}(\overline{c}.imp).\ [\sigma, e) \subseteq \overline{c}.imp(mid).\mathsf{ddc} \vee \\ \exists mid \in \mathsf{dom}(\overline{c}.imp).\ [\sigma, e) \subseteq \overline{c}.\mathsf{mstc}(mid))\end{array}}{t_{ctx} \ltimes \overline{c} \vdash_{border} \alpha, s, \varsigma}$$

Figure 12: Silent-state invariant for compiled programs

$$\frac{\begin{array}{c}\text{(Silent-state invariant)}\\ t_{ctx} \ltimes \overline{c} = \lfloor t_0 \rfloor \\ \overline{c} \in \mathsf{range}(\llbracket\cdot\rrbracket) \qquad t_0 \vdash_{exec} s \qquad t \in \{t_{ctx}, \overline{c}\} \qquad \overline{t} \in \{t_{ctx}, \overline{c}\} \setminus \{t\} \\ s.\mathsf{pcc} \subseteq \mathsf{dom}(t.\mathcal{M}_c) \qquad \forall a \in \mathsf{dom}(\mathcal{M}_{\overline{t},border}).\ \mathcal{M}_{\overline{t},border}(a) = s.\mathcal{M}_d(a) \\ ((-\infty, \mathsf{na}_{border}) \cup r_t) \cap \mathsf{dom}(\mathcal{M}_{\overline{t},border}) = \emptyset \qquad \varsigma \cap \mathsf{dom}(\mathcal{M}_{\overline{t},border}) = \emptyset \\ R_t = \bigcup_{mid \in \mathsf{dom}(t.imp)} \mathsf{reachable\_addresses}(\{s.\mathsf{mstc}(mid), t.imp(mid).\mathsf{ddc}\}, s.\mathcal{M}_d) \\ s.\mathsf{nalloc} \le \mathsf{na}_{border} \qquad R_t \subseteq (r_t \cup [s.\mathsf{nalloc}, \mathsf{na}_{border})) \\ \bigcup_{mid \in \mathsf{dom}(t.imp)} \{t.\mathsf{mstc}(mid), t.imp(mid).\mathsf{ddc}\} \subseteq caps_{4origin,border} \\ \forall a \in R_t.\ s.\mathcal{M}_d(a) = (\delta, \sigma, e, \_) \implies \\ \exists cap \in caps_{4origin,border}.\ [\sigma, e) \subseteq cap \ \vee\ [\sigma, e) \subseteq [s.\mathsf{nalloc}, \mathsf{na}_{border})\end{array}}{t_{ctx} \ltimes \overline{c} \vdash_{silent} s, \varsigma, caps_{4origin,border}, r_t, \mathsf{na}_{border}, \mathcal{M}_{\overline{t},border}}$$

In both cases that arise, we strengthen the memory equality conjunct by observing that the same memory appears also on the matching step $(s_2, \varsigma_2 \xrightarrow{\lambda}_{[\overline{c}]} s_2', \varsigma_2')$.
Also, in both cases, we strengthen the stack similarity by instantiating Lemma 147.
The other subgoals of strong similarity (from Definition 86) are straightforward. ☐

Definition 87 (Per-subject state-universal predicate)**.** *A predicate $P : \mathcal{V} \to \mathbb{B}$ holds universally for* *all values of a program state* s *where* t *is the subject of* s *when:*

$$\begin{array}{l}
\mathsf{per_subject_state_universal}(P, s, t) \stackrel{\mathsf{def}}{=} \\
s.\mathsf{pcc} \subseteq \mathsf{dom}(t.\mathcal{M}_c) \ \wedge \\
\forall a.\ a \in \bigcup\limits_{mid \in \mathsf{dom}(t.imp)} \mathsf{reachable_addresses}(\{s.\mathsf{mstc}(mid), t.imp(mid).\mathsf{ddc}\}, s.\mathcal{M}_d) \implies P(s.\mathcal{M}_d(a)) \\
\wedge \\
P(s.\mathsf{ddc}) \ \wedge \ P(s.\mathsf{stc}) \ \wedge \ P(s.\mathsf{pcc}) \ \wedge \\
\forall mid \in \mathsf{dom}(t.imp).\ P(s.imp(mid).\mathsf{pcc}) \ \wedge \ P(s.imp(mid).\mathsf{ddc}) \ \wedge \ P(s.\mathsf{mstc}(mid)) \ \wedge \\
\forall (cc, dc, _, _) \in s.stk.\ cc \subseteq \mathsf{dom}(t.\mathcal{M}_c) \implies P(cc) \ \wedge \ P(dc)
\end{array}$$

Lemma 154 (Predicates that are guaranteed to hold on the result of expression evaluation under the execution of a specific subject)**.**

$$\begin{array}{l}
\forall t, t_1, t_2, \mathcal{E}, s, v. \\
\mathcal{E}, s.\mathcal{M}_d, s.\mathsf{ddc}, s.\mathsf{stc}, s.\mathsf{pcc} \Downarrow v \ \wedge \\
t \in \{t_1, t_2\} \ \wedge \\
t_1 \ltimes t_2 \vdash_{exec} s \ \wedge \\
\mathsf{per_subject_state_universal}(P, s, t) \ \wedge \\
\mathsf{offset_oblivious}(P) \ \wedge \\
\mathsf{z_trivial}(P) \ \wedge \\
\mathsf{subcap_closed}(P) \\
\implies \\
P(v)
\end{array}$$

Proof. Similar to Lemma 44. □

Lemma 155 (Preservation of per-subject state universality of predicates)**.**

$$\begin{array}{l}
\forall P, t, t_{ctx}, \overline{c}, s, s', \nabla. \\
s.\mathsf{nalloc} < 0 \ \wedge \\
t \in \{t_{ctx}, \overline{c}\} \ \wedge \\
t_{ctx} \ltimes \overline{c} \vdash_{exec} s \ \wedge \\
\mathsf{per_subject_state_universal}(P, s, t) \ \wedge \\
\mathsf{allocation_compatible}(P, s'.\mathsf{nalloc} - 1) \ \wedge \\
\mathsf{offset_oblivious}(P) \ \wedge \\
\mathsf{z_trivial}(P) \ \wedge \\
\mathsf{subcap_closed}(P) \ \wedge \\
s, \varsigma \xrightarrow{\tau^*}{}^{*}_{[\overline{c}], \nabla} s', \varsigma \\
\implies \\
\mathsf{per_subject_state_universal}(P, s', t) \ \wedge \ s'.\mathsf{nalloc} < 0
\end{array}$$

Proof. Similar to Lemma 45. □

Definition 88 (Four-origin policy)**.**

$$
\begin{aligned}
&\mathsf{four\_origin\_policy}_{t,s,\varsigma,\alpha}(v) \stackrel{\mathsf{def}}{=} \\
&v = (\delta, \sigma, e, \_) \implies \\
&\exists mid \in \mathsf{dom}(t.imp).\ [\sigma, e) \subseteq t.imp(mid).\mathsf{ddc} \vee \\
&\exists mid \in \mathsf{dom}(t.imp).\ [\sigma, e) \subseteq s.\mathsf{mstc}(mid) \vee \\
&\exists a' \in \varsigma, idx \in [0, |\alpha|).\ [\sigma, e) \subseteq \mathsf{mem}(\alpha(idx))(a') \vee \\
&\exists i \in \mathsf{allocation\_intervals}(?, \alpha).\ [\sigma, e) \subseteq i
\end{aligned}
$$

Claim 35 (Border state invariant to silent state invariant - $\overline{c}$ executing)**.**

$$
\begin{aligned}
&t_{ctx} \ltimes \overline{c} \vdash_{border} \alpha, s, \varsigma \wedge \\
&caps = \{v \mid \mathsf{four\_origin\_policy}_{t_{ctx} \ltimes \overline{c}, s, \varsigma, \alpha}(v)\} \wedge \\
&r_t = \bigcup_{mid \in \mathsf{dom}(t_{ctx}.imp)} \mathsf{reachable\_addresses}(\{s.\mathsf{mstc}(mid), t_{ctx}.imp(mid).\mathsf{ddc}\}, s.\mathcal{M}_d) \\
&\implies \\
&\exists \mathcal{M}_d. \\
&t_{ctx} \ltimes \overline{c} \vdash_{silent} s, \varsigma, caps, r_t, s.\mathsf{nalloc}, \mathcal{M}_d
\end{aligned}
$$

(Proof Sketch): Follows from Definition 88 after inversion of rule Border-state invariant.

Claim 36 (Border state invariant to silent state invariant - t_{ctx} executing)**.**

$$
\begin{aligned}
&t_{ctx} \ltimes \overline{c} \vdash_{border} \alpha, s, \varsigma \wedge \\
&caps = \{v \mid \mathsf{four\_origin\_policy}_{t_{ctx} \ltimes \overline{c}, s, \varsigma, \alpha}(v)\} \wedge \\
&r_t = \bigcup_{mid \in \mathsf{dom}(\overline{c}.imp)} \mathsf{reachable\_addresses}(\{s.\mathsf{mstc}(mid), \overline{c}.imp(mid).\mathsf{ddc}\}, s.\mathcal{M}_d) \\
&\implies \\
&\exists \mathcal{M}_d. \\
&t_{ctx} \ltimes \overline{c} \vdash_{silent} s, \varsigma, caps, r_t, s.\mathsf{nalloc}, \mathcal{M}_d
\end{aligned}
$$

Similar to Claim 35.

Lemma 156 (Possible origins of capability values at border states)**.**

$$
\begin{aligned}
&\forall t_{ctx}, \overline{c}, \alpha, s, \varsigma, \mathcal{E}, \sigma, e.\\
&t_{ctx} \ltimes \overline{c} \vdash_{border} \alpha, s, \varsigma \;\wedge\\
&\mathcal{E}, s.\mathcal{M}_d, s.\mathsf{ddc}, s.\mathsf{stc}, s.\mathsf{pcc} \Downarrow (\delta, \sigma, e, \_) \;\wedge\\
&I_{ctx} = \mathsf{allocation\_intervals}(?, \alpha) \;\wedge\\
&I_{\overline{c}} = \mathsf{allocation\_intervals}(!, \alpha)\\
&\implies\\
&s.\mathsf{pcc} \subseteq \mathsf{dom}(\overline{c}.\mathcal{M}_c) \;\wedge\\
&(\exists i \in I_{\overline{c}}.\ [\sigma, e) \subseteq i \;\vee\\
&\exists a' \in \varsigma, idx \in [0, |\alpha|).\ [\sigma, e) \subseteq \mathsf{mem}(\alpha(idx))(a') \;\vee\\
&\exists mid \in \mathsf{dom}(\overline{c}.imp).\ [\sigma, e) \subseteq \overline{c}.imp(mid).\mathsf{ddc} \;\vee\\
&\exists mid \in \mathsf{dom}(\overline{c}.imp).\ [\sigma, e) \subseteq \overline{c}.\mathsf{mstc}(mid))\\
&\quad\vee\\
&s.\mathsf{pcc} \subseteq \mathsf{dom}(t_{ctx}.\mathcal{M}_c) \;\wedge\\
&(\exists i \in I_{ctx}.\ [\sigma, e) \subseteq i \;\vee\\
&\exists a' \in \varsigma, idx \in [0, |\alpha|).\ [\sigma, e) \subseteq \mathsf{mem}(\alpha(idx))(a') \;\vee\\
&\exists mid \in \mathsf{dom}(t_{ctx}.imp).\ [\sigma, e) \subseteq t_{ctx}.imp(mid).\mathsf{ddc} \;\vee\\
&\exists mid \in \mathsf{dom}(t_{ctx}.imp).\ [\sigma, e) \subseteq t_{ctx}.\mathsf{mstc}(mid))
\end{aligned}
$$

Proof.

- We assume the antecedents, and prove our lemma by induction on the evaluation of $\mathcal{E}$.
  - **Case evalconst**,
  - **Case evalCapType**,
  - **Case evalCapStart**,
  - **Case evalCapEnd**,
  - **Case evalCapOff**, and
  - **Case evalBinOp:**
    These cases are vacuous.
  - **Case evalddc:**
    Here, we distinguish the following two cases:
    * **Case** $s.\mathsf{pcc} \subseteq \mathsf{dom}(\overline{c}.\mathcal{M}_c)$**:**
      In this case, we choose to prove the left disjunct of our goal.
      Further, we choose to prove the following disjunct:
      $\exists mid \in \mathsf{dom}(\overline{c}.imp).\ [s.\mathsf{ddc}.\sigma, s.\mathsf{ddc}.e) \subseteq \overline{c}.imp(mid).\mathsf{ddc}$
      Now this latter goal follows by inverting assumption $t_{ctx} \ltimes \overline{c} \vdash_{border} \alpha, s, \varsigma$ using rule Border-state invariant, and then inverting its preconditions using rule exec-state.
    * **Case** $s.\mathsf{pcc} \subseteq \mathsf{dom}(t_{ctx}.\mathcal{M}_c)$**:**
      In this case, we choose to prove the right disjunct of our goal.
      Further, we choose to prove the following disjunct:
      $\exists mid \in \mathsf{dom}(t_{ctx}.imp).\ [s.\mathsf{ddc}.\sigma, s.\mathsf{ddc}.e) \subseteq t_{ctx}.imp(mid).\mathsf{ddc}$
      Now this latter goal follows by inverting assumption $t_{ctx} \ltimes \overline{c} \vdash_{border} \alpha, s, \varsigma$ using rule Border-state invariant, and then inverting its preconditions using rule exec-state.

- **Case evalstc:**
  Here, we distinguish the following two cases:
  * **Case $s.\mathsf{pcc} \subseteq \mathsf{dom}(\overline{c}.\mathcal{M}_c)$:**
    In this case, we choose to prove the left disjunct of our goal.
    Further, we choose to prove the following disjunct:
    $\exists mid \in \mathsf{dom}(\overline{c}.imp).\ [s.\mathsf{ddc}.\sigma, s.\mathsf{ddc}.e) \subseteq \overline{c}.\mathsf{mstc}(mid)$
    Now this latter goal follows by inverting assumption $t_{ctx} \ltimes \overline{c} \vdash_{border} \alpha, s, \varsigma$ using rule Border-state invariant, and then inverting its preconditions using rule exec-state.
  * **Case $s.\mathsf{pcc} \subseteq \mathsf{dom}(t_{ctx}.\mathcal{M}_c)$:**
    In this case, we choose to prove the right disjunct of our goal.
    Further, we choose to prove the following disjunct:
    $\exists mid \in \mathsf{dom}(t_{ctx}.imp).\ [s.\mathsf{ddc}.\sigma, s.\mathsf{ddc}.e) \subseteq t_{ctx}.\mathsf{mstc}(mid)$
    Now this latter goal follows by inverting assumption $t_{ctx} \ltimes \overline{c} \vdash_{border} \alpha, s, \varsigma$ using rule Border-state invariant, and then inverting its preconditions using rule exec-state.
- **Case evalIncCap:**
  Here, $\mathcal{E} = \texttt{inc}(\mathcal{E}_c, \mathcal{E}_z)$, and we have the preconditions:
  (Ec-eval):
  $\mathcal{E}_c, s.\mathcal{M}_d, s.\mathsf{ddc}, s.\mathsf{stc}, s.\mathsf{pcc} \Downarrow (x, \sigma, e, \mathit{off})$, and
  (Ez-eval):
  $\mathcal{E}_z, s.\mathcal{M}_d, s.\mathsf{ddc}, s.\mathsf{stc}, s.\mathsf{pcc} \Downarrow z$
  We distinguish two cases:
  * **Case $x = \delta$:**
    Here, our goal follows immediately from the induction hypothesis on (Ec-eval) after substitution.
  * **Case $x \neq \delta$:**
    Here, our goal is vacuously true.
- **Case evalLim:**
  Here, $\mathcal{E} = \texttt{inc}(\mathcal{E}_c, \mathcal{E}_z)$, and we have the preconditions:
  (Ec-eval):
  $\mathcal{E}_c, s.\mathcal{M}_d, s.\mathsf{ddc}, s.\mathsf{stc}, s.\mathsf{pcc} \Downarrow (x, \sigma, e, \_)$, and
  (CAP-BOUNDS-SUB):
  $[\sigma', e') \subseteq [\sigma, e)$
  We distinguish two cases:
  * **Case $x = \delta$:**
    Here, our goal follows immediately from the induction hypothesis on (Ec-eval) after applying transitivity of $\subseteq$ using (CAP-BOUNDS-SUB).
  * **Case $x \neq \delta$:**
    Here, our goal is vacuously true.
- **Case evalDeref:**
  Here, $\mathcal{E} = \texttt{deref}(\mathcal{E}_c)$.
  We have the following preconditions:
  (Ec-eval):
  $\mathcal{E}_c, s.\mathcal{M}_d, s.\mathsf{ddc}, s.\mathsf{stc}, s.\mathsf{pcc} \Downarrow (x, \sigma', e', \mathit{off})$,
  (Ec-delta):
  $\vdash_\delta (x, \sigma', e', \mathit{off})$, and
  (Mem-deref):
  $s.\mathcal{M}_d(\sigma' + \mathit{off}) = (\delta, \sigma, e, \_)$
  We claim (Bounds-reachable):
  $[\sigma', e') \subseteq \mathsf{reachable\_addresses}(\{s.\mathsf{stc}, s.\mathsf{ddc}\}, s.\mathcal{M}_d)$
  We apply Lemma 25 to this claim to obtain the following subgoals:

* $s.\mathsf{pcc} = (\kappa, \_, \_, \_)$,
* $s.\mathsf{ddc} = (\delta, \_, \_, \_)$, and
* $s.\mathsf{stc} = (\delta, \_, \_, \_)$
  All of these follow by inverting assumption $t_{ctx} \ltimes \bar{c} \vdash_{border} \alpha, s, \varsigma$ using rule Border-state invariant, and then inverting its preconditions using rule exec-state.
* $\mathcal{E}_c, s.\mathcal{M}_d, s.\mathsf{ddc}, s.\mathsf{stc}, s.\mathsf{pcc} \Downarrow (\delta, \sigma', e', \mathit{off})$
  Immediate by (Ec-eval) and (Ec-delta).

Using (Bounds-reachable) and (Ec-delta)–unfolding Definition 2, we know (Addr-reachable):
$\sigma' + \mathit{off} \in \mathsf{reachable\_addresses}(\{s.\mathsf{stc}, s.\mathsf{ddc}\}, s.\mathcal{M}_d)$
Now, we distinguish the following two cases:

* **Case** $s.\mathsf{pcc} \subseteq \mathsf{dom}(\bar{c}.\mathcal{M}_c)$**:**
  We choose to prove the left disjunct of our goal.
  Here, we claim (Addr-reachable-all):
  $\sigma' + \mathit{off} \in \bigcup_{mid \in \mathsf{dom}(\bar{c}.imp)} \mathsf{reachable\_addresses}(\{s.\mathsf{mstc}(mid), \bar{c}.imp(mid).\mathsf{ddc}\}, s.\mathcal{M}_d)$
  We apply Lemma 18 to this claim obtaining the following subgoals:
  · $\{s.\mathsf{stc}, s.\mathsf{ddc}\} \subseteq \bigcup_{mid \in \mathsf{dom}(\bar{c}.imp)} \{s.\mathsf{mstc}(mid), \bar{c}.imp(mid).\mathsf{ddc}\}$
    This follows by substituting the case condition in the preconditions obtained by inverting assumption $t_{ctx} \ltimes \bar{c} \vdash_{border} \alpha, s, \varsigma$ using rule Border-state invariant, and then inverting its preconditions using rule exec-state.
  · $\sigma' + \mathit{off} \in \mathsf{reachable\_addresses}(\{s.\mathsf{stc}, s.\mathsf{ddc}\}, s.\mathcal{M}_d)$
    This is immediate by (Addr-reachable).

  We now distinguish two cases:
  · **Case** $\sigma' + \mathit{off} \in \varsigma$**:**
    Here, we choose to prove the following disjunct of (the necessary top-level left disjunct of) our goal:
    $\exists a' \in \varsigma, idx \in [0, |\alpha|).\ [\sigma, e) \subseteq \mathsf{mem}(\alpha(idx))(a')$
    We pick:
    $a' := \sigma' + \mathit{off}$, and
    $idx := |\alpha| - 1$
    Thus, it remains to show that:
    $[\sigma, e) \subseteq \mathsf{mem}(\alpha(|\alpha| - 1))(\sigma' + \mathit{off})$
    We apply the substitution:
    $\mathsf{mem}(\alpha(|\alpha| - 1)) = s.\mathcal{M}_d$
    obtaining the following two subgoals:
    1. $\mathsf{mem}(\alpha(|\alpha| - 1)) = s.\mathcal{M}_d$
       This is immediate by inverting assumption $t_{ctx} \ltimes \bar{c} \vdash_{border} \alpha, s, \varsigma$ using rule Border-state invariant.
    2. $[\sigma, e) \subseteq s.\mathcal{M}_d(\sigma' + \mathit{off})$
       Here, we apply reflexivity of $\subseteq$, so our goal is immediate by (Mem-deref).
  · **Case** $\sigma' + \mathit{off} \notin \varsigma$**:**
    Here, by inverting assumption $t_{ctx} \ltimes \bar{c} \vdash_{border} \alpha, s, \varsigma$ using rule Border-state invariant, we obtain the following preconditions:
    (Rc-def):
    $R_{\bar{c}} = \bigcup_{mid \in \mathsf{dom}(\bar{c}.imp)} \mathsf{reachable\_addresses}(\{s.\mathsf{mstc}(mid), \bar{c}.imp(mid).\mathsf{ddc}\}, s.\mathcal{M}_d)$,
    and
    (All-privately-held-caps):
    $\forall a \in R_{\bar{c}} \setminus \varsigma.\ s.\mathcal{M}_d(a) = (\delta, \sigma, e, \_) \implies$
    $(\exists i \in I_{\bar{c}}.\ [\sigma, e) \subseteq i\ \ \vee$

$\exists a' \in \varsigma, idx \in [0, |\alpha|).\ [\sigma, e) \subseteq \mathsf{mem}(\alpha(idx))(a') \vee$
$\exists mid \in \mathsf{dom}(\overline{c}.imp).\ [\sigma, e) \subseteq \overline{c}.imp(mid).\mathsf{ddc} \vee$
$\exists mid \in \mathsf{dom}(\overline{c}.imp).\ [\sigma, e) \subseteq \overline{c}.\mathsf{mstc}(mid))$
We instantiate the latter (All-privately-held-caps) with $a := \sigma' + \mathit{off}$ obtaining the following two subgoals:

1. $\sigma' + \mathit{off} \in R_{\overline{c}}$
   By unfolding $R_{\overline{c}}$ using (Rc-def), this goal is immediate by (Addr-reachable-all).
2. $\sigma' + \mathit{off} \notin \varsigma$
   This is immediate by the case condition.

The instantiation immediately gives us our goal.

* **Case** $s.\mathsf{pcc} \subseteq \mathsf{dom}(t_{ctx}.\mathcal{M}_c)$**:**
  We choose to prove the right disjunct of our goal. The proof is analogous to the previous case. We omit it for brevity.

This concludes the proof of **case** **evalDeref**.

This concludes the proof of Lemma 156. □

Silent-state invariant

Lemma 157 (Preservation of the silent-state invariant)**.**

$$\begin{array}{l}\forall t_{ctx}, \overline{c}, s, \varsigma, caps_{4origin,border}, r_{t,border}, \mathsf{na}_{border}, \mathcal{M}_{\overline{t},border}, s', \nabla.\\ t_{ctx} \ltimes \overline{c} \vdash_{silent} s, \varsigma, caps_{4origin,border}, r_{t,border}, \mathsf{na}_{border}, \mathcal{M}_{\overline{t},border} \wedge\\ s, \varsigma \xrightarrow{\tau^*}{}^{*}_{[\overline{c}],\nabla} s', \varsigma\\ \implies\\ t_{ctx} \ltimes \overline{c} \vdash_{silent} s', \varsigma, caps_{4origin,border}, r_{t,border}, \mathsf{na}_{border}, \mathcal{M}_{\overline{t},border}\end{array}$$

Proof.

- We assume the antecedents, and prove our goal by induction on the relation $s, \varsigma \xrightarrow{\tau^*}{}^{*}_{[\overline{c}],\nabla} s', \varsigma$
- **Case** **trace-closure-refl:**

  Here, the goal is immediate by assumption.
- **Case** **trace-closure-trans:**

  Here, by assumption, we have s'' with:
  $s, \varsigma \xrightarrow{\tau^*}{}^{*}_{[\overline{c}],\nabla} s'', \varsigma$, and
  $s'', \varsigma \xrightarrow{\tau}_{[\overline{c}],\nabla} s', \varsigma$,
  and the induction hypothesis
  $t_{ctx} \ltimes \overline{c} \vdash_{silent} s'', \varsigma, caps_{4origin,border}, r_{t,border}, \mathsf{na}_{border}, \mathcal{M}_{\overline{t},border}$.
  By inversion of the induction hypothesis using rule Silent-state invariant, we obtain the following assumptions:

  Valid linking:
  $t_{ctx} \ltimes \overline{c} = \lfloor t_0 \rfloor$

  Compiled component:
  $\overline{c} \in \mathsf{range}(\llbracket\cdot\rrbracket)$

Exec state:
$t_0 \vdash_{exec} s''$

Arbitrary t:
$t \in \{t_{ctx}, \overline{c}\}$

Arbitrary $\bar{t}$:
$\bar{t} \in \{t_{ctx}, \overline{c}\} \setminus \{t\}$

t is executing:
$s''.\mathsf{pcc} \subseteq \mathsf{dom}(t.\mathcal{M}_c)$

Private memory of $\bar{t}$ is untouched:
$\forall a \in \mathsf{dom}(\mathcal{M}_{\bar{t},border}).\ \mathcal{M}_{\bar{t},border}(a) = s''.\mathcal{M}_d(a)$

Private memory was indeed private:
$((-\infty, \mathsf{na}_{border}) \cup r_t) \cap \mathsf{dom}(\mathcal{M}_{\bar{t},border}) = \emptyset$

Private memory is compatible with the history of sharing:
$\varsigma \cap \mathsf{dom}(\mathcal{M}_{\bar{t},border}) = \emptyset$

Reachable addresses of t:
$$R''_t = \bigcup_{mid \in \mathsf{dom}(t.imp)} \mathsf{reachable\_addresses}(\{s''.\mathsf{mstc}(mid), t.imp(mid).\mathsf{ddc}\}, s''.\mathcal{M}_d)$$

New allocation is bounded by na_{border}:
$s''.\mathsf{nalloc} \leq \mathsf{na}_{border}$

Reachable addresses of t can grow only by allocation:
$R''_t \subseteq (r_t \cup [s''.\mathsf{nalloc}, \mathsf{na}_{border}))$

The border capabilities contain capabilities on t's static memory:
$$\bigcup_{mid \in \mathsf{dom}(t.imp)} \{t.\mathsf{mstc}(mid), t.imp(mid).\mathsf{ddc}\} \subseteq caps_{4origin,border}$$

Five-origin policy:
$\forall a \in R''_t.\ s''.\mathcal{M}_d(a) = (\delta, \sigma, e, \_) \implies$
$\exists cap \in caps_{4origin,border}.\ [\sigma, e) \subseteq cap \ \vee\ [\sigma, e) \subseteq [s''.\mathsf{nalloc}, \mathsf{na}_{border})$

By applying rule Silent-state invariant to our goal, we obtain subgoals about s' that we refer to using the names given above to the corresponding assumptions:

- Subgoals "**Valid linking**", "**Compiled component**", "**Arbitrary t**", "**Arbitrary $\bar{t}$**", "**Private memory was indeed private**", "**Private memory is compatible with the history of sharing**", and "**The border capabilities contain capabilities on t's static memory**" are immediate.
- There is nothing to prove about the definition **Reachable addresses of t**.
- To prove subgoal **Exec state**, we apply Corollary 2 obtaining the following subgoals:
  * $t \vdash_{exec} s''$
    This is immediate by assumption **Exec state**.

* $s'' \to^* s'$
To prove this, we apply Claim 15 obtaining the following subgoal: $s'', \varsigma \xrightarrow{\tau^*}{}^{*}_{[\bar{c}],\nabla} s', \varsigma'$
This is immediate by assumptions after applying rule trace-closure-refl.

– To prove the remaining subgoals, we distinguish the possible cases of assumption $s'', \varsigma \xrightarrow{\tau}_{[\bar{c}],\nabla} s', \varsigma$:

* **Case assign-silent:**
By inversion of the assumptions of assign-silent using rule assign, we obtain
(S'-MEM):
$s'.\mathcal{M}_d = s''.\mathcal{M}_d[c \mapsto v]$,
(v-EVAL'd-IN-t):
$\mathcal{E}_R, s''.\mathcal{M}_d, s''.\mathsf{ddc}, s''.\mathsf{stc}, s''.\mathsf{pcc} \Downarrow v$,
(c-EVAL'd-IN-t):
$\mathcal{E}_L, s''.\mathcal{M}_d, s''.\mathsf{ddc}, s''.\mathsf{stc}, s''.\mathsf{pcc} \Downarrow c$,
(c-IN-BOUNDS):
$\vdash_\delta c$,
(EQUAL-MSTC):
$s''.\mathsf{mstc} = s'.\mathsf{mstc}$, and
(EQUAL-NALLOC):
$s''.\mathsf{nalloc} = s'.\mathsf{nalloc}$
We first prove the goal **Reachable addresses of t can grow only by allocation**.
Assuming $R'_t = \bigcup_{mid \in \mathsf{dom}(t.imp)} \mathsf{reachable\_addresses}(\{s'.\mathsf{mstc}(mid), t.imp(mid).\mathsf{ddc}\}, s'.\mathcal{M}_d)$,
our goal is $R'_t \subseteq (r_t \cup (s'.\mathsf{nalloc}, \mathsf{na}_{border}])$.
By the transitivity of $\subseteq$, it suffices to show that:
$R'_t \subseteq r_t$.
We prove our goal by applying transitivity of $\subseteq$ obtaining the following two subgoals:
  1. $R''_t \subseteq r_t$
  Immediate by assumption **Reachable addresses of t can grow only by allocation**.
  2. $R'_t \subseteq R''_t$
  Here, we apply Lemma 38 obtaining the following subgoals:
     (a) $c.\sigma + c.\mathit{off} \in R''_t$
     Here, we apply Lemma 25 obtaining subgoals that are immediate by (c-EVAL'd-IN-t), (c-IN-BOUNDS), and by inversion of assumption **Exec state** using rule exec-state.
     (b) $v = (\delta, \sigma, e, \_) \implies \bigcup_{mid \in \mathsf{dom}(t.imp)} \{s''.\mathsf{mstc}(mid), t.imp(mid).\mathsf{ddc}\}, s''.\mathcal{M}_d \vDash v$
     Assuming $v = (\delta, \sigma, e, \_)$ and by unfolding Definition 23, this goal becomes:
     $[\sigma, e) \subseteq \mathsf{reachable\_addresses}(\bigcup_{mid \in \mathsf{dom}(t.imp)} \{s''.\mathsf{mstc}(mid), t.imp(mid).\mathsf{ddc}\}, s''.\mathcal{M}_d)$
     By applying Lemmas 6 and 18, we obtain the following two subgoals:
        i. $[\sigma, e) \subseteq \mathsf{reachable\_addresses}(\{s''.\mathsf{stc}, s''.\mathsf{ddc}\}, s''.\mathcal{M}_d)$
        Here, we apply Lemma 25 obtaining subgoals that are immediate by (v-EVAL'd-IN-t), and by inversion of assumption **Exec state** using rule exec-state.
        ii. $\exists mid \in \mathsf{dom}(t.imp).\ s''.\mathsf{mstc}(mid) \doteq s''.\mathsf{stc}$
        iii. $\exists mid \in \mathsf{dom}(t.imp).\ s''.imp(mid).\mathsf{ddc} \doteq s''.\mathsf{ddc}$
        These two subgoals are immediate by inverting assumption **Exec state** using rule exec-state and substituting in the preconditions using assumption **t is executing**.
     (c) (applying Lemma 6) $s''.\mathsf{mstc} \doteq s'.\mathsf{mstc}$
     Immediate by (EQUAL-MSTC).

Next, we prove the goal **Five-origin policy**.
We fix an arbitrary $a \in R'_t$, and assume $s'.\mathcal{M}_d(a) = (\delta, \sigma, e, \_)$.
Out goal is (after substitution using (EQUAL-NALLOC)):
$\exists cap \in caps_{4origin,border}.\ [\sigma, e) \subseteq cap \ \vee\ [\sigma, e) \subseteq [s''.\mathsf{nalloc}, \mathsf{na}_{border})$
We distinguish the following two cases:

1. **Case** $a = c.\sigma + c.\mathit{off}$**:**
   We instantiate Lemma 26 using (v-EVAL'd-IN-t) and using subgoal **Exec state** inverted by rule exec-state to obtain
   (3-ORIGINS):
   $[\sigma, e) \subseteq s''.\mathsf{ddc}\ \vee$
   $[\sigma, e) \subseteq s''.\mathsf{stc}\ \vee$
   $\exists a_o.\ [\sigma, e) \subseteq s''.\mathcal{M}_d(a_o)\ \wedge\ a_o \in \mathsf{reachable\_addresses}(\{s''.\mathsf{ddc}, s''.\mathsf{stc}\}, s''.\mathcal{M}_d)$
   We distinguish the following three cases of (3-ORIGINS):
   (a) **Case** $[\sigma, e) \subseteq s''.\mathsf{ddc}$, and
   (b) **Case** $[\sigma, e) \subseteq s''.\mathsf{stc}$
   In these two cases, we apply the transitivity of $\subseteq$ obtaining the subgoals $[\sigma, e) \subseteq s''.\mathsf{ddc}$ and $[\sigma, e) \subseteq s''.\mathsf{stc}$ respectively.
   Both of these subgoals are immediate by the assumption "t **is executing**" together with the assumption "**The border capabilities contain capabilities on** t**'s static memory**".
   (c) **Case** $\exists a_o.\ [\sigma, e) \subseteq s''.\mathcal{M}_d(a_o)\ \wedge\ a_o \in \mathsf{reachable\_addresses}(\{s''.\mathsf{ddc}, s''.\mathsf{stc}\}, s''.\mathcal{M}_d)$**:**
   Here, we obtain a_o, and use it to instantiate assumption **Five origin policy** thus immediately proving our goal.
   (The instantiation is possible by Lemma 18.)
2. **Case** $a \neq c.\sigma + c.\mathit{off}$**:**
   Here, we apply assumption **Five-origin policy** obtaining the following subgoals:
   (a) $s''.\mathcal{M}_d(a) = (\delta, \sigma, e, \_)$
   Immediate by (S'-MEM).
   (b) $a \in R''_t$
   Follows from assumption $a \in R'_t$ and $R'_t \subseteq R''_t$. The latter was proved in the previous goal.

Next, we prove the goal t **is executing**.
Immediate from the corresponding assumption by noticing that $s''.\mathsf{pcc}\ \doteq\ s'.\mathsf{pcc}$.

Next, we prove the goal **New allocation is bounded by** na_{border}.
This is immediate from the corresponding assumption after substitution using (EQUAL-NALLOC).

Next, we prove the goal **Private memory of** $\bar{t}$ **is untouched.**
We pick an arbitrary $a \in \mathsf{dom}(\mathcal{M}_{\bar{t},border})$,
and our goal is to show that $s'.\mathcal{M}_d(a) = \mathcal{M}_{\bar{t},border}(a)$.
By the corresponding assumption (i.e., assumption **Private memory of** $\bar{t}$ **is untouched**) about s'', it suffices by the transitivity of equality to show that:
$s'.\mathcal{M}_d(a) = s''.\mathcal{M}_d(a)$
By (S'-MEM), it thus suffices to show that:
$a \neq c.\sigma + c.\mathit{off}$
For this, it suffices to show that $\mathsf{dom}(\mathcal{M}_{\bar{t},border}) \cap R'_t = \emptyset$
But since by the previously proven subgoal **Reachable addresses of** t **can grow only by allocation**, we know $R'_t \subseteq (r_t \cup [s'.\mathsf{nalloc}, \mathsf{na}_{border}))$, then it suffices to show that

$\mathsf{dom}(\mathcal{M}_{\bar{t},border}) \cap (r_t \cup [s'.\mathsf{nalloc}, \mathsf{na}_{border})) = \emptyset$
The latter is immediate by subgoal **Private memory was indeed private** using simple arithmetic and interval arithmetic identities.
This concludes the proof of **case assign-silent**.

* **Case alloc-silent:**
By inversion of the assumptions of alloc-silent using rule allocate, we obtain
(c-EVAL's-IN-t):
$\mathcal{E}_L, s''.\mathcal{M}_d, s''.\mathsf{ddc}, s''.\mathsf{stc}, s''.\mathsf{pcc} \Downarrow c$,
(v-POSITIVE):
$v \in \mathbb{Z}^+$,
(c-IN-BOUNDS):
$\vdash_\delta c$,
(S'-MEM):
$s'.\mathcal{M}_d = s''.\mathcal{M}_d[c \mapsto (\delta, s''.\mathsf{nalloc} - v, s''.\mathsf{nalloc}, 0), i \mapsto 0 \ \forall i \in [s''.\mathsf{nalloc} - v, s''.\mathsf{nalloc})]$,
(S'-NALLOC):
$s'.\mathsf{nalloc} = s''.\mathsf{nalloc} - v$, and
(EQUAL-MSTC):
$s''.\mathsf{mstc} = s'.\mathsf{mstc}$

We first prove the goal **Reachable addresses of t can grow only by allocation**.
Assuming $R'_t = \bigcup_{mid \in \mathsf{dom}(t.imp)} \mathsf{reachable\_addresses}(\{s'.\mathsf{mstc}(mid), t.imp(mid).\mathsf{ddc}\}, s'.\mathcal{M}_d)$,
our goal is $R'_t \subseteq (r_t \cup (s'.\mathsf{nalloc}, \mathsf{na}_{border}])$.
By inversion of assumption **Exec state** using rule exec-state, and by rewriting using Lemma 18,
we know that (*):
$\forall a.\ a \in R''_t \implies a \geq s''.\mathsf{nalloc}$

Let $\mathcal{M}_{enlarged} = s''.\mathcal{M}_d[i \mapsto 0 \ | \ i \in [s''.\mathsf{nalloc} - v, s''.\mathsf{nalloc})]$
And let $R_{t,enlarged} = \bigcup_{mid \in \mathsf{dom}(t.imp)} \mathsf{reachable\_addresses}(\{s'.\mathsf{mstc}(mid), t.imp(mid).\mathsf{ddc}\}, \mathcal{M}_{enlarged})$

We claim (DECOMPOSED-REACHABILITY):
$R_{t,enlarged} = R''_t$

We prove this claim by induction on $k \in [s''.\mathsf{nalloc} - v, s''.\mathsf{nalloc})$
where $\mathcal{M}_k = s''.\mathcal{M}_d[i \mapsto 0 \ | \ i \in [s''.\mathsf{nalloc} - v, k)]$, and
$R_{t,k} = \bigcup_{mid \in \mathsf{dom}(t.imp)} \mathsf{reachable\_addresses}(\{s'.\mathsf{mstc}(mid), t.imp(mid).\mathsf{ddc}\}, \mathcal{M}_k)$.

The base case is immediate by reflexivity after substitution using (EQUAL-MSTC).

In the inductive step, our goal is $R_{t,k} = R''_t$.
We apply Lemma 21 (after substitution using (EQUAL-MSTC)) obtaining the subgoal:
$k - 1 \notin R_{t,k-1}$
Using the induction hypothesis, we can instead prove:
$k - 1 \notin R''_t$
Because $k < s''.\mathsf{nalloc}$ by choice, then we know $k - 1 < s''.\mathsf{nalloc}$.
But then by instantiating the contrapositive of (*) using $k - 1$, we immediately obtain our subgoal.

Now notice from (S'-MEM) and by the definition of partial maps that (S'-MEM-DECOMPOSED):
$s'.\mathcal{M}_d = \mathcal{M}_{enlarged}[c \mapsto (\delta, s''.\mathsf{nalloc} - v, s''.\mathsf{nalloc}, 0)]$

We pick an arbitrary $a \in R'_t$, and our goal is to show that $a \in r_t \cup [s'.\mathsf{nalloc}, \mathsf{na}_{border})$.

By instantiating Lemma 40 using the rewriting (S'-MEM-DECOMPOSED), and using:
$\mathcal{M}_d = \mathcal{M}_{enlarged}, a_a = a, \hat{a} = c.\sigma + c.\mathit{off}, \sigma = s''.\mathsf{nalloc} - v, e = s''.\mathsf{nalloc}$,
we know:
$a \in R_{t,enlarged} \ \vee \ a \in [s''.\mathsf{nalloc} - v, s''.\mathsf{nalloc})$
Thus, by rewriting using (DECOMPOSED-REACHABILITY) and using (S'-NALLOC), we know:
$a \in R''_t \ \vee \ a \in [s'.\mathsf{nalloc}, s''.\mathsf{nalloc})$
We now distinguish these two cases:

1. $a \in R''_t$
   Here, by the induction hypothesis, we know $a \in r_t \cup [s''.\mathsf{nalloc}, \mathsf{na}_{border})$.
   But by (S'-NALLOC), we know $[s''.\mathsf{nalloc}, \mathsf{na}_{border}) \subseteq [s'.\mathsf{nalloc}, \mathsf{na}_{border})$
   Thus, using both and by the definition of $\subseteq$, our goal is immediate.
2. $a \in [s'.\mathsf{nalloc}, s''.\mathsf{nalloc})$
   Again, here by (S'-NALLOC), and the assumption **New allocation is bounded by** na_{border}, we know
   $[s'.\mathsf{nalloc}, s''.\mathsf{nalloc}) \subseteq [s'.\mathsf{nalloc}, \mathsf{na}_{border})$, which by the definition of $\subseteq$ gives us our goal.

This concludes the proof of the goal **Reachable addresses of t can grow only by allocation**.

Next, we prove the goal **Five-origin policy**.
We fix an arbitrary $a \in R'_t$, and assume $s'.\mathcal{M}_d(a) = (\delta, \sigma, e, \_)$.
Out goal is:
$\exists cap \in caps_{4origin,border}.\ [\sigma, e) \subseteq cap \ \vee \ [\sigma, e) \subseteq [s'.\mathsf{nalloc}, \mathsf{na}_{border})$
We distinguish the following three cases:

1. **Case** $a = c.\sigma + c.\mathit{off}$**:**
   Here, we know $\sigma = s'.\mathsf{nalloc}, e = s''.\mathsf{nalloc}$.
   We prove the right disjunct of our goal.
   So it suffices to prove that
   $[s'.\mathsf{nalloc}, s''.\mathsf{nalloc}) \subseteq [s'.\mathsf{nalloc}, \mathsf{na}_{border})$
   Thus, it suffices to prove that
   $s''.\mathsf{nalloc} \leq \mathsf{na}_{border}$
   This is immediate by assumption **New allocation is bounded by** na_{border}.
2. **Case** $a \in [s'.\mathsf{nalloc}, s''.\mathsf{nalloc})$**:**
   Here, the assumption $s'.\mathcal{M}_d(a) = (\delta, \sigma, e, \_)$ is false. So our goal holds vacuously.
3. **Case** $a \notin \{c.\sigma + c.\mathit{off}\} \cup [s'.\mathsf{nalloc}, s''.\mathsf{nalloc})$**:**
   Here, we know by (S'-MEM) that $s''.\mathcal{M}_d(a) = s'.\mathcal{M}_d(a)$
   Thus, we know (*):
   $s''.\mathcal{M}_d(a) = (\delta, \sigma, e, \_)$
   We instantiate Lemma 40 using $C = \bigcup_{mid \in \mathsf{dom}(t.imp)} \{s'.\mathsf{mstc}(mid), t.imp.\mathsf{ddc}\}$,
   $\mathcal{M}_d = \mathcal{M}_{enlarged}, a_a = a, \hat{a} = c.\sigma + c.\mathit{off}, \sigma = s''.\mathsf{nalloc} - v, e = s''.\mathsf{nalloc}$
   to obtain:
   $a \in R'_t \implies a \in \mathsf{reachable\_addresses}(C, \mathcal{M}_{enlarged}) \ \vee \ a \in [s'.\mathsf{nalloc}, s''.\mathsf{nalloc})$
   Thus, by instantiation using our assumption about a, then by elimination using

our case condition, we conclude:
$a \in \mathsf{reachable\_addresses}(C, \mathcal{M}_{enlarged})$
By substitution using (DECOMPOSED-REACHABILITY), we obtain (**):
$a \in R''_t$
(Notice that we reuse the claim (DECOMPOSED-REACHABILITY) that was defined in the proof of a previous subgoal. The same goes for the definition of $\mathcal{M}_{enlarged}$, etc..)
Using (*) and (**), we instantiate assumption **Five-origin policy** obtaining:
$\exists cap \in caps_{4origin,border}.\ [\sigma, e) \subseteq cap\ \lor\ [\sigma, e) \subseteq [s''.\mathsf{nalloc}, \mathsf{na}_{border})$
We distinguish the following two cases:

(a) **Case** $\exists cap \in caps_{4origin,border}.\ [\sigma, e) \subseteq cap$**:**
Here, the left disjunct of our goal is immediate.

(b) **Case** $[\sigma, e) \subseteq [s''.\mathsf{nalloc}, \mathsf{na}_{border}]$**:**
Here, we prove the right disjunct of our goal by applying the transitivity of $\subseteq$ obtaining the subgoal $s'.\mathsf{nalloc} \leq s''.\mathsf{nalloc}$ which is immediate by (S'-NALLOC) and the condition on v being positive.

This concludes the proof of subgoal **Five-origin policy**.

Next, we prove the goal t **is executing**.
Immediate from the corresponding assumption by noticing that $s''.\mathsf{pcc}\ \doteq\ s'.\mathsf{pcc}$.

Next, we prove the goal **New allocation is bounded by** na_{border}.
This is immediate from the corresponding assumption and (S'-NALLOC).

Next, we prove the goal **Private memory of** $\bar{t}$ **is untouched**.
We pick an arbitrary $a \in \mathsf{dom}(\mathcal{M}_{\bar{t},border})$,
and our goal is to show that $s'.\mathcal{M}_d(a) = \mathcal{M}_{\bar{t},border}(a)$.
By the corresponding assumption (i.e., assumption **Private memory of** $\bar{t}$ **is untouched**) about s'', it suffices by the transitivity of equality to show that:
$s'.\mathcal{M}_d(a) = s''.\mathcal{M}_d(a)$
By (S'-MEM), it thus suffices to show that:
$a \notin \{c.\sigma + c.\mathit{off}\} \cup [s'.\mathsf{nalloc}, s''.\mathsf{nalloc})$
Showing that $a \notin \{c.\sigma + c.\mathit{off}\}$ is the same proof as in **case assign-silent**.
We show that $a \notin [s'.\mathsf{nalloc}, s''.\mathsf{nalloc})$.
For this, it suffices to show that:
$\mathsf{dom}(\mathcal{M}_{\bar{t},border}) \cap [s'.\mathsf{nalloc}, s''.\mathsf{nalloc}) = \emptyset$
By assumption **New allocation is bounded by** na_{border} about $s''.\mathsf{nalloc}$, it suffices to show that:
$\mathsf{dom}(\mathcal{M}_{\bar{t},border}) \cap [s'.\mathsf{nalloc}, \mathsf{na}_{border}) = \emptyset$
By interval identities, it suffices to show that:
$\mathsf{dom}(\mathcal{M}_{\bar{t},border}) \cap (-\infty, \mathsf{na}_{border}) = \emptyset$
By set identities, this follows from assumption **Private memory was indeed private**.
This concludes the proof of subgoal **Private memory of** $\bar{t}$ **is untouched**.

* **Case jump-silent:**
From the assumptions of jump-silent, we distinguish the following two cases.
  - **Case jump0:**
  Here, we have the following assumptions:
  (JUMP-INSTR):
  $s''.\mathcal{M}_c(s''.\mathsf{pcc}) = \mathsf{JumpIfZero}\ \mathcal{E}_{cond}\ \mathcal{E}_{size}$
  (size-EVAL):
  $\mathcal{E}_{size}, s''.\mathcal{M}_d, s''.\mathsf{ddc}, s''.\mathsf{stc}, s''.\mathsf{pcc} \Downarrow v$

(S'-PCC):
$s'.\mathsf{pcc} = \mathtt{inc}(s''.\mathsf{pcc}, v)$
(S'-MEM):
$s'.\mathcal{M}_d = s''.\mathcal{M}_d$
(S'-NALLOC):
$s'.\mathsf{nalloc} = s''.\mathsf{nalloc}$
(S'-MSTC):
$s'.\mathsf{mstc} = s''.\mathsf{mstc}$

We first prove the goal **Reachable addresses of t can grow only by allocation**.
Assuming $R'_t = \bigcup_{mid \in \mathsf{dom}(t.imp)} \mathsf{reachable\_addresses}(\{s'.\mathsf{mstc}(mid), t.imp(mid).\mathsf{ddc}\}, s'.\mathcal{M}_d)$,
our goal is $R'_t \subseteq (r_t \cup (s'.\mathsf{nalloc}, \mathsf{na}_{border}])$.
After substitution using (S'-MEM), (S'-MSTC), and (S'-NALLOC), this goal is immediate by assumptions **Reachable addresses of t** and **Reachable addresses of t can grow only by allocation**.

Next, we prove the goal **Five-origin policy**.
We fix an arbitrary $a \in R'_t$, and assume $s'.\mathcal{M}_d(a) = (\delta, \sigma, e, \_)$.
Out goal is:
$\exists cap \in caps_{4origin,border}.\ [\sigma, e) \subseteq cap \ \vee\ [\sigma, e) \subseteq [s'.\mathsf{nalloc}, \mathsf{na}_{border})$
After substitution using (S'-MEM), (S'-MSTC), and (S'-NALLOC), this goal is immediate by assumptions **Reachable addresses of t** and **Five-origin policy**.

Next, we prove the goal **t is executing**.
This is immediate by the corresponding assumption after noticing from (S'-PCC) that $s'.\mathsf{pcc} \doteq s''.\mathsf{pcc}$.

Next, we prove the goal **New allocation is bounded by na_{border}**.
This is immediate from the corresponding assumption after substitution using (S'-NALLOC).

Next, we prove the goal **Private memory of $\bar{t}$ is untouched**.
We pick an arbitrary $a \in \mathsf{dom}(\mathcal{M}_{\bar{t},border})$,
and our goal is to show that $s'.\mathcal{M}_d(a) = \mathcal{M}_{\bar{t},border}(a)$.
This is immediate from the corresponding assumption after substitution using (S'-MEM).

- **Case jump1:**
Here, we have the following assumptions:
(S'-PCC):
$s'.\mathsf{pcc} = \mathtt{inc}(s''.\mathsf{pcc})$
(S'-MEM):
$s'.\mathcal{M}_d = s''.\mathcal{M}_d$
(S'-NALLOC):
$s'.\mathsf{nalloc} = s''.\mathsf{nalloc}$
(S'-MSTC):
$s'.\mathsf{mstc} = s''.\mathsf{mstc}$
We prove the goal **t is executing**.
From (S'-PCC) and by unfolding the definition of $\mathtt{inc}$, we immediately have that $s'.\mathsf{pcc} \doteq s''.\mathsf{pcc}$. So, our goal is immediate from the assumption **t is executing** about $s''.\mathsf{pcc}$.
All other goals are identical to the corresponding goals of **case jump0** above.

∗ **Case cinvoke-silent-compiled:**
By the assumptions of cinvoke-silent-compiled and by their inversion using rule cinvoke and then cinvoke-aux, we obtain:
(IN-BOUNDS-S"-PCC):
$\vdash_\kappa s''.\mathsf{pcc}$
(S"-PCC):
$s''.\mathsf{pcc} \in \mathsf{dom}(\bar{c}.\mathcal{M}_c)$
(S'-IMP-MID):
$mid \in \mathsf{dom}(\bar{c}.imp)$
(S'-PCC):
$s'.\mathsf{pcc} = \mathtt{inc}(s''.imp(mid).\mathsf{pcc}, s''.imp(mid).offs(fid))$
(S'-DDC):
$s'.\mathsf{ddc} = s''.imp(mid).\mathsf{ddc}$
(S'-STC):
$s'.\mathsf{stc} = \mathtt{inc}(s''.\mathsf{mstc}(mid), nArgs + nLocal)$
(IN-BOUNDS-S'-STC):
$\vdash_\delta s'.\mathsf{stc}$
(STC-POINTER):
$s''.\mathsf{mstc}(mid) = (\delta, \sigma, e, off)$
(S'-MEM):
$s'.\mathcal{M}_d = s''.\mathcal{M}_d[\sigma + off + i \mapsto v_i\ \forall i \in [0, nArgs)][\sigma + off + nArgs + i \mapsto 0\ \forall i \in [0, nLocal)]$
(S'-NALLOC):
$s'.\mathsf{nalloc} = s''.\mathsf{nalloc}$
(S'-MSTC):
$s'.\mathsf{mstc} = s''.\mathsf{mstc}[mid \mapsto (\delta, s''.\mathsf{mstc}(mid).\sigma, s''.\mathsf{mstc}(mid).e, \_)]$

We first prove the goal **Reachable addresses of t can grow only by allocation**.
Assuming $R'_t = \bigcup_{mid \in \mathsf{dom}(t.imp)} \mathsf{reachable\_addresses}(\{s'.\mathsf{mstc}(mid), t.imp(mid).\mathsf{ddc}\}, s'.\mathcal{M}_d)$,
our goal is $R'_t \subseteq (r_t \cup (s'.\mathsf{nalloc}, \mathsf{na}_{border}])$.
By substitution using (S'-NALLOC), our goal becomes:
$R'_t \subseteq (r_t \cup (s''.\mathsf{nalloc}, \mathsf{na}_{border}])$.
Thus, using assumption **Reachable addresses of t can grow only by allocation**, and by the transitivity of $\subseteq$, it suffices to prove:
$R'_t \subseteq R''_t$
Similarly to the proof of the corresponding goal in **case alloc-silent**, the proof proceeds by induction on the number of memory updates defining intermediate memories indexed by the updated address.

For updates at addresses in $[\sigma + off + nArgs, \sigma + off + nArgs + nLocal)$, we apply Lemma 37 that immediately solves our goal.

For updates at addresses in $[\sigma + off, \sigma + off + nArgs)$, we apply Lemma 38 that immediately solves our goal.

We omit the details because they are very similar to the proof of the same goal in **case alloc-silent**.

Next, we prove the goal **Five-origin policy**.
We fix an arbitrary $a \in R'_t$, and assume $s'.\mathcal{M}_d(a) = (\delta, \sigma, e, \_)$.
Our goal is:
$\exists cap \in caps_{4origin,border}.\ [\sigma, e) \subseteq cap\ \vee\ [\sigma, e) \subseteq [s'.\mathsf{nalloc}, \mathsf{na}_{border})$
By substitution using (S'-NALLOC), our goal becomes:

$\exists cap \in caps_{4origin,border}.\ [\sigma, e) \subseteq cap \ \vee\ [\sigma, e) \subseteq [s''.\mathsf{nalloc}, \mathsf{na}_{border})$
By using the proposition $R'_t \subseteq R''_t$ proved above, we know $a \in R''_t$.
We then distinguish three cases:

1. **Case** $a \in [\sigma + \mathit{off} + nArgs, \sigma + \mathit{off} + nArgs + nLocal)$**:**
   Here, from the contradiction to the assumption $s'.\mathcal{M}_d(a) = (\delta, \sigma, e, \_)$ obtained by instantiating (S'-MEM), we have our goal.
2. **Case** $a \in [\sigma + \mathit{off}, \sigma + \mathit{off} + nArgs)$**:**
   This case is similar to the proof of the corresponding goal (**Five-origin policy**) of **case assign-silent**. We omit it for brevity.
3. **Case** $a \notin [\sigma + \mathit{off}, \sigma + \mathit{off} + nArgs + nLocal)$**:**
   Here, we know by instantiating (S'-MEM) that
   $s''.\mathcal{M}_d(a) = s'.\mathcal{M}_d(a)$.
   Thus, our goal is immediate by instantiating assumption **Five-origin policy**.

Next, we prove the goal t **is executing**.
From (S"-PCC), (IN-BOUNDS-S"-PCC), and the assumption (t **is executing**),
we know $t = \bar{c}$,
which we substitute in our goal obtaining instead the subgoal:
$s'.\mathsf{pcc} \subseteq \mathsf{dom}(\bar{c}.\mathcal{M}_c)$.
Using (S'-PCC) and (S'-IMP-MID), we instantiate the preconditions obtained by inverting **Exec state** using rule exec-state. The instantiation immediately solves our goal.

Next, we prove the goal **New allocation is bounded by** na_{border}.
This is immediate from the corresponding assumption after substitution using (S'-NALLOC).

Next, we prove the goal **Private memory of** $\bar{t}$ **is untouched.**
We pick an arbitrary $a \in \mathsf{dom}(\mathcal{M}_{\bar{t},border})$,
and our goal is to show that $s'.\mathcal{M}_d(a) = \mathcal{M}_{\bar{t},border}(a)$.
By the corresponding assumption (i.e., assumption **Private memory of** $\bar{t}$ **is untouched**) about s'', it suffices by the transitivity of equality to show that:
$s'.\mathcal{M}_d(a) = s''.\mathcal{M}_d(a)$
By (S'-MEM), it thus suffices to show that:
$a \notin [\sigma + \mathit{off}, \sigma + \mathit{off} + nArgs + nLocal)$
For this, it suffices by set identities to show both that:
$[\sigma + \mathit{off}, \sigma + \mathit{off} + nArgs + nLocal) \subseteq R'_t$
and that:
$\mathsf{dom}(\mathcal{M}_{\bar{t},border}) \cap R'_t = \emptyset$

1. **Subgoal** $[\sigma + \mathit{off}, \sigma + \mathit{off} + nArgs + nLocal) \subseteq R'_t$**:**
   Using the proposition $R'_t \subseteq R''_t$ proved in a previous goal and by the transitivity of $\subseteq$, it suffices to show that:
   $[\sigma + \mathit{off}, \sigma + \mathit{off} + nArgs + nLocal) \subseteq R''_t$

   Using (IN-BOUNDS-S'-STC), (S'-STC), and (STC-POINTER), and by unfolding Definition 2, we conclude:
   $[\sigma + \mathit{off}, \sigma + \mathit{off} + nArgs + nLocal) \subseteq s''.\mathsf{mstc}(mid)$
   Thus, it suffices for our goal by the transitivity of $\subseteq$ to show that:
   $[s''.\mathsf{mstc}(mid).\sigma, s''.\mathsf{mstc}(mid).e) \subseteq R''_t$

   By unfolding R''_t using assumption **Reachable addresses of** t (after substitution using $t = \bar{c}$ that we proved in an earlier subgoal and instantiation using (S'-

IMP-MID)) then unfolding Definition 22, it suffices by easy set identities and by additivity (Lemma 17) to show that:
$[s''.\mathsf{mstc}(mid).\sigma, s''.\mathsf{mstc}(mid).e) \subseteq \mathsf{access}_{\_,s''.\mathcal{M}_d}[s''.\mathsf{mstc}(mid).\sigma, s''.\mathsf{mstc}(mid).e)$
The latter is immediate by expansiveness (Lemma 8).

2. **Subgoal $\mathsf{dom}(\mathcal{M}_{\bar{t},border}) \cap R'_t = \emptyset$:**
Since by the previously proven subgoal **Reachable addresses of t can grow only by allocation**, we know $R'_t \subseteq (r_t \cup [s'.\mathsf{nalloc}, \mathsf{na}_{border}))$, then it suffices to show that
$\mathsf{dom}(\mathcal{M}_{\bar{t},border}) \cap (r_t \cup [s'.\mathsf{nalloc}, \mathsf{na}_{border})) = \emptyset$
The latter is immediate by subgoal **Private memory was indeed private** using simple arithmetic and interval arithmetic identities.

* **Case cinvoke-silent-context:**
This is very similar to the previous case. We omit the proof for brevity.

* **Case creturn-silent-compiled:**
By the assumptions of creturn-silent-compiled and by their inversion using rule creturn, we obtain:
(IN-BOUNDS-S”-PCC):
$\vdash_\kappa s''.\mathsf{pcc}$
(S”-PCC):
$s''.\mathsf{pcc} \subseteq \mathsf{dom}(\bar{c}.\mathcal{M}_c)$
(S’-PCC):
$s'.\mathsf{pcc} \subseteq \mathsf{dom}(\bar{c}.\mathcal{M}_c)$
(S’-MEM):
$s'.\mathcal{M}_d = s''.\mathcal{M}_d$
(S’-NALLOC):
$s'.\mathsf{nalloc} = s''.\mathsf{nalloc}$
(S’-PCC-SAME-MID-STC):
$\exists mid'.\ s'.\mathsf{pcc} \subseteq s''.imp(mid').\mathsf{pcc}\ \wedge\ s'.\mathsf{stc} = \mathsf{mstc}(mid')$
(S’-MSTC):
$s'.\mathsf{mstc} = s''.\mathsf{mstc}[mid \mapsto \mathtt{inc}(s''.\mathsf{mstc}(mid), \_)]$
(S’-DDC):
$s'.stk, (s'.\mathsf{ddc}, s'.\mathsf{pcc}, \_, \_) = \mathsf{pop}(s''.stk)$

We first prove the goal **Reachable addresses of t can grow only by allocation**.
Assuming $R'_t = \bigcup_{mid \in \mathsf{dom}(t.imp)} \mathsf{reachable\_addresses}(\{s'.\mathsf{mstc}(mid), t.imp(mid).\mathsf{ddc}\}, s'.\mathcal{M}_d)$,
our goal is $R'_t \subseteq (r_t \cup (s'.\mathsf{nalloc}, \mathsf{na}_{border}])$.
By substitution using (S’-NALLOC), our goal becomes:
$R'_t \subseteq (r_t \cup (s''.\mathsf{nalloc}, \mathsf{na}_{border}])$.
But by substitution using (S’-MEM) in the definition of R'_t, we have:
$R'_t = \bigcup_{mid \in \mathsf{dom}(t.imp)} \mathsf{reachable\_addresses}(\{s'.\mathsf{mstc}(mid), t.imp(mid).\mathsf{ddc}\}, s''.\mathcal{M}_d)$
By applying Lemma 18, and then using induction on the size of $\{mid \mid mid \in \mathsf{dom}(t.imp)\}$, we can show that $R'_t = R''_t$.
(The proof instantiates Lemma 6 using (S’-MSTC).)

Thus, by substitution using $R'_t = R''_t$ in our goal, it becomes immediate by the assumption **Reachable addresses of t can grow only by allocation**.

Next, we prove the goal **Five-origin policy**.
We fix an arbitrary $a \in R'_t$, and assume $s'.\mathcal{M}_d(a) = (\delta, \sigma, e, \_)$.
Our goal is:
$\exists cap \in caps_{4origin,border}.\ [\sigma, e) \subseteq cap\ \vee\ [\sigma, e) \subseteq [s'.\mathsf{nalloc}, \mathsf{na}_{border})$

By substitution using (S'-NALLOC), our goal becomes:
$\exists cap \in caps_{4origin,border}.\ [\sigma, e) \subseteq cap \ \vee \ [\sigma, e) \subseteq [s''.\mathsf{nalloc}, \mathsf{na}_{border})$
By using the proposition $R'_t = R''_t$ proved above, we know $a \in R''_t$.
But also using the (S'-MEM), we know $s''.\mathcal{M}_d(a) = (\delta, \sigma, e, \_)$
Now our goal is immediate by instantiating assumption **Five-origin policy** using a.

Next, we prove the goal t **is executing**.
By substitution in assumption t **is executing** using (S"-PCC) and (IN-BOUNDS-S"-PCC), our goal becomes:
$s'.\mathsf{pcc} \subseteq \mathsf{dom}(\overline{c}.\mathcal{M}_c)$
Immediate by (S'-PCC).

Next, we prove the goal **New allocation is bounded by** na_{border}.
This is immediate from the corresponding assumption after substitution using (S'-NALLOC).

Next, we prove the goal **Private memory of** $\bar{t}$ **is untouched**.
We pick an arbitrary $a \in \mathsf{dom}(\mathcal{M}_{\bar{t},border})$,
and our goal is to show that $s'.\mathcal{M}_d(a) = \mathcal{M}_{\bar{t},border}(a)$.
This is immediate from the corresponding assumption after substitution using (S'-MEM).

* **Case creturn-silent-context:**
  This case is similar to the previous case. We omit the proof for brevity.

□

Lemma 158 (Preservation of the border-state invariant $\vdash_{border}$)**.**

$$\begin{aligned}
&\forall t_{ctx}, \overline{c}, \alpha, s, \varsigma, \lambda, s', \varsigma'.\\
&t_{ctx} \ltimes \overline{c} \vdash_{border} \alpha, s, \varsigma \ \wedge\\
&s, \varsigma \xrightarrow{\lambda}_{[\overline{c}],\triangledown} s', \varsigma' \ \wedge\\
&\lambda \neq \checkmark\\
&\Longrightarrow\\
&t_{ctx} \ltimes \overline{c} \vdash_{border} \alpha\lambda, s', \varsigma'
\end{aligned}$$

Proof.

- We fix arbitrary $t_{ctx}, \overline{c}, \alpha, s, \varsigma, \lambda, s', \varsigma'$, and assume the antecedents.
- By inversion of our assumptions using rule trace-steps-lambda, we obtain the following preconditions:

  (STAR-TAU-STEPS):
  $s, \varsigma \xrightarrow{\tau^*}{}^{*}_{[\overline{c}],\triangledown} s'', \varsigma''$,

  (NON-SILENT-STEP):
  $s'', \varsigma'' \xrightarrow{\lambda}_{[\overline{c}],\triangledown} s', \varsigma' \ \wedge \ \lambda \neq \tau$
- By inversion of the assumptions using rule Border-state invariant, we obtain the following preconditions:

Valid linking

$t_{ctx} \ltimes \overline{c} = \lfloor t \rfloor$

Compiled program

$\overline{c} \in \mathsf{range}(\llbracket\cdot\rrbracket)$

Exec invariant

$t \vdash_{exec} s$

Reachable addresses of the context

$$R_{ctx} = \bigcup_{mid \in \mathsf{dom}(t_{ctx}.imp)} \mathsf{reachable\_addresses}(\{s.\mathsf{mstc}(mid), t_{ctx}.imp(mid).\mathsf{ddc}\}, s.\mathcal{M}_d)$$

Reachable addresses of the compiled program

$$R_{\overline{c}} = \bigcup_{mid \in \mathsf{dom}(\overline{c}.imp)} \mathsf{reachable\_addresses}(\{s.\mathsf{mstc}(mid), \overline{c}.imp(mid).\mathsf{ddc}\}, s.\mathcal{M}_d)$$

Memory at the border is described by the trace label

$\mathsf{mem}(\alpha(|\alpha| - 1)) = s.\mathcal{M}_d|_{\varsigma}$

All mutually reachable addresses were recorded as shared

$R_{ctx} \cap R_{\overline{c}} \subseteq \varsigma$

Allocation intervals of the context

$I_{ctx} = \mathsf{allocation\_intervals}(?, \alpha)$

Allocation intervals of the compiled program

$I_{\overline{c}} = \mathsf{allocation\_intervals}(!, \alpha)$

Four-origin policy for privately-held capabilities of the context

$\forall a \in R_{ctx} \setminus \varsigma.\ s.\mathcal{M}_d(a) = (\delta, \sigma, e, \_) \implies$
$(\exists i \in I_{ctx}.\ [\sigma, e) \subseteq i\ \lor$
$\exists a' \in \varsigma, idx \in [0, |\alpha|).\ [\sigma, e) \subseteq \mathsf{mem}(\alpha(idx))(a') \lor$
$\exists mid \in \mathsf{dom}(t_{ctx}.imp).\ [\sigma, e) \subseteq t_{ctx}.imp(mid).\mathsf{ddc} \lor$
$\exists mid \in \mathsf{dom}(t_{ctx}.imp).\ [\sigma, e) \subseteq t_{ctx}.\mathsf{mstc}(mid))$

Four-origin policy for privately-held capabilities of the compiled program

$\forall a \in R_{\overline{c}} \setminus \varsigma.\ s.\mathcal{M}_d(a) = (\delta, \sigma, e, \_) \implies$
$(\exists i \in I_{\overline{c}}.\ [\sigma, e) \subseteq i\ \lor$
$\exists a' \in \varsigma, idx \in [0, |\alpha|).\ [\sigma, e) \subseteq \mathsf{mem}(\alpha(idx))(a') \lor$
$\exists mid \in \mathsf{dom}(\overline{c}.imp).\ [\sigma, e) \subseteq \overline{c}.imp(mid).\mathsf{ddc} \lor$
$\exists mid \in \mathsf{dom}(\overline{c}.imp).\ [\sigma, e) \subseteq \overline{c}.\mathsf{mstc}(mid))$

- We apply rule Border-state invariant to our goal obtaining subgoals (about $\alpha\lambda$, s', and ς') that are analogous to the preconditions above (about α, s, and ς). We skip the explicit stating of the subgoals for the sake of brevity, and re-use the names for the preconditions that are introduced above.

  We let:

  $R'_{ctx} = \bigcup_{mid \in \mathsf{dom}(t_{ctx}.imp)} \mathsf{reachable\_addresses}(\{s'.\mathsf{mstc}(mid), t_{ctx}.imp(mid).\mathsf{ddc}\}, s'.\mathcal{M}_d)$,

  $R'_{\overline{c}} = \bigcup_{mid \in \mathsf{dom}(\overline{c}.imp)} \mathsf{reachable\_addresses}(\{s'.\mathsf{mstc}(mid), \overline{c}.imp(mid).\mathsf{ddc}\}, s'.\mathcal{M}_d)$,

  $I'_{ctx} = \mathsf{allocation\_intervals}(?, \alpha\lambda)$, and

  $I'_{\overline{c}} = \mathsf{allocation\_intervals}(!, \alpha\lambda)$

- We claim (EXEC-S”):

  $t \vdash_{exec} s''$

  To prove it, we apply Corollary 2 obtaining the following subgoals:

  - $t \vdash_{exec} s$

    This is immediate by assumption **Exec invariant**.

- $s \to^* s''$
  To prove this, we apply Claim 15 obtaining the following subgoal:
  $s, \varsigma \xrightarrow{\tau^*}{}^{*}_{[\bar{c}],\triangledown} s'', \varsigma''$
  This is immediate by (STAR-TAU-STEPS).

- From our lemma assumption, we know by instantiating (conditionally on $s.\mathsf{pcc} \subseteq \mathsf{dom}(t_{ctx}.\mathcal{M}_c)$) either Claim 35 or Claim 36, that:
  $\exists caps, r_t, \mathcal{M}_d.\ t_{ctx} \ltimes \bar{c} \vdash_{silent} s, \varsigma, caps, r_t, s.\mathsf{nalloc}, \mathcal{M}_d$

- Thus, by instantiating Lemma 157 using (STAR-TAU-STEPS), we know that:
  (SILENT-S"): $\exists caps, r_t, \mathcal{M}_d.\ t_{ctx} \ltimes \bar{c} \vdash_{silent} s'', \varsigma, caps, r_t, s.\mathsf{nalloc}, \mathcal{M}_d$

- Goals **Valid linking** and **Compiled program** are immediate.

- The remaining goals are proved by distinguishing the following cases for (NON-SILENT-STEP):

  - **Case cinvoke-context-to-compiled:**
    To prove the goal **Exec invariant**, we apply Lemma 53 obtaining the following subgoals:
    - $s'' \succ_{\approx} s'$
      This is immediate by inversion of rule cinvoke-context-to-compiled.
    - $t \vdash_{exec} s''$
      This is immediate by (EXEC-S").

    The goal **Memory at the border is described by the trace label**, i.e., $\mathsf{mem}(\alpha\lambda(|\alpha\lambda| - 1)) = s'.\mathcal{M}_d|_{\varsigma'}$ is immediate by definition of λ that we get by inversion of rule cinvoke-context-to-compiled.
    To prove the goal **Four-origin policy for privately-held capabilities of the context**, we pick an arbitrary $a \in R'_{ctx} \setminus \varsigma'$, and assume $s'.\mathcal{M}_d(a) = (\delta, \sigma, e, \_)$
    Our goal is:
    $\exists i \in I'_{ctx}.\ [\sigma, e) \subseteq i\ \vee$
    $\exists a' \in \varsigma', idx \in [0, |\alpha\lambda|).\ [\sigma, e) \subseteq \mathsf{mem}(\alpha\lambda(idx))(a') \vee$
    $\exists mid \in \mathsf{dom}(t_{ctx}.imp).\ [\sigma, e) \subseteq t_{ctx}.imp(mid).\mathsf{ddc} \vee$
    $\exists mid \in \mathsf{dom}(t_{ctx}.imp).\ [\sigma, e) \subseteq t_{ctx}.\mathsf{mstc}'(mid)$
    We distinguish the following cases:
    - **Case** $[\sigma, e) \subseteq (s'.\mathsf{nalloc}, -1]$**:**
    - **Case** $[\sigma, e) \not\subseteq (s'.\mathsf{nalloc}, -1]$**:**
      Here, we claim
      (NO-MIXED-STATIC-DYNAMIC-CAPABILITY):
      $[\sigma, e) \cap (s'.\mathsf{nalloc}, -1] = \emptyset$
      (Sketch) Then the proof follows in both cases from (SILENT-S") by inversion of rule Silent-state invariant.

    To prove the goal **All mutually-reachable addresses were recorded as shared**, we pick an arbitrary $a \in R'_{ctx} \cap R'_{\bar{c}}$. The goal is to show that:
    $a \in \varsigma'$
    By substitution from the preconditions of rule cinvoke-context-to-compiled, the goal becomes:
    $a \in \mathsf{reachable\_addreses\_closure}(\varsigma'' \cup r, s'.\mathcal{M}_d)$
    (where $r = \mathsf{reachable\_addresses}(\{\bar{v}(i) \mid i \in [0, nArgs) \wedge \bar{v}(i) = (\delta, \_, \_, \_)\}, s'.\mathcal{M}_d)$, and $\bar{v} = [i \mapsto v_i \mid\ \forall i \in [0, nArgs)\ \ \bar{e}(i), s''.\mathcal{M}_d, s''.\mathsf{ddc}, s''.\mathsf{stc}, s''.\mathsf{pcc} \Downarrow v_i]$)

By unfolding our goal using Definition 22, our goal becomes:
$a \in \bigcup_{k \in [0, |s'.\mathcal{M}_d|]} \mathsf{access}_{k, s'.\mathcal{M}_d}(\varsigma'' \cup r)$
After instantiating Claim 17 using (STAR-TAU-STEPS), our goal by substitution becomes:
$a \in \bigcup_{k \in [0, |s'.\mathcal{M}_d|]} \mathsf{access}_{k, s'.\mathcal{M}_d}(\varsigma \cup r)$
(Sketch) The proof of this is tedious, but should follow from the conditions on $s''.\mathcal{M}_d$ that we obtain by inversion of (SILENT-S") using rule Silent-state invariant.

The goal **Four-origin policy for privately-held capabilities of the program** is similar to the previous one.

- **Case cinvoke-compiled-to-context:**
- **Case creturn-to-compiled:**
- **Case creturn-to-context:**
  (Sketch): These cases are similar to the representative one above.

□

Back-Translation

Structure of the emulating context

Definition 89 (Main module of the emulating context)**.**

$$\mathtt{mainModule}(\alpha) \stackrel{\mathsf{def}}{=} (\text{“}\mathit{mainModule}\text{”}, \mathtt{mainGlobalVars}(\alpha), \mathtt{mainModuleFuncs})$$

where mainGlobalVars and mainModuleFuncs are as defined below (Definitions 102 and 105). We first give some auxiliary definitions.

Definition 90 (Context module IDs of a trace)**.**

$$\mathtt{contextModIDs}(\alpha) \stackrel{\mathsf{def}}{=} \{\mathit{mid} \mid \mathtt{call}(\mathit{mid}, \mathit{fid})\_!\_, \_ \in \alpha\}$$

Definition 91 (Context function IDs of a trace)**.**

$$\mathtt{contextFunIDs}(\alpha) \stackrel{\mathsf{def}}{=} \{\text{“}\mathit{mid\_fid}\text{”} \mid \mathtt{call}(\mathit{mid}, \mathit{fid})\_!\_, \_ \in \alpha\}$$

Definition 92 (Number of arguments of a function inferred from either the trace α_1 or the trace α_2)**.**

$$\forall \overline{v}.\ \mathtt{call}(\mathit{mid}, \mathit{fid})\overline{v}!\_, \_ \in \alpha_1 \ \vee\ \mathtt{call}(\mathit{mid}, \mathit{fid})\overline{v}!\_, \_ \in \alpha_2 \implies \mathtt{nArgs}(\text{“}\mathit{mid\_fid}\text{”}, \alpha_1, \alpha_2) = |\overline{v}|$$

Definition 93 (Memory of a trace label)**.**

$$\mathsf{mem}(\tau) \stackrel{\mathsf{def}}{=} \bot$$
$$\mathsf{mem}(\checkmark) \stackrel{\mathsf{def}}{=} \bot$$
$$\mathsf{mem}(\mathtt{ret}\ \_\ \mathcal{M}_d, \_) \stackrel{\mathsf{def}}{=} \mathcal{M}_d$$
$$\mathsf{mem}(\mathtt{call}(\_, \_)\ \_\ \_\ \mathcal{M}_d, \_) \stackrel{\mathsf{def}}{=} \mathcal{M}_d$$

Definition 94 (Allocation status of a trace label).

$$
\begin{aligned}
\mathsf{nalloc}(\tau) &\stackrel{\mathsf{def}}{=} \bot \\
\mathsf{nalloc}(\checkmark) &\stackrel{\mathsf{def}}{=} \bot \\
\mathsf{nalloc}(\texttt{ret}\ \_\ \mathcal{M}_d, n) &\stackrel{\mathsf{def}}{=} n \\
\mathsf{nalloc}(\texttt{call}(\_,\_)\ \_\ \_\ \mathcal{M}_d, n) &\stackrel{\mathsf{def}}{=} n
\end{aligned}
$$

Definition 95 (Shared addresses throughout a trace prefix α).

$$
\texttt{sharedAddresses}(\alpha) \stackrel{\mathsf{def}}{=} \bigcup_i \mathsf{dom}(\mathsf{mem}(\alpha(i)))
$$

Definition 96 (Context addresses collected from a trace).

$$
\mathsf{ctx\_addresses}(\alpha) \stackrel{\mathsf{def}}{=} \bigcup_{\{i \ \mid\ \alpha(i) \in \dot{?}\}} \mathsf{dom}(\mathsf{mem}(\alpha(i))) \setminus \mathsf{dom}(\mathsf{mem}(\alpha(i-1)))
$$

Definition 97 (Data segment that the context shares (collected from a trace)).

$$
\begin{aligned}
&\mathsf{shareable\_data\_segment\_ctx}(\alpha) \stackrel{\mathsf{def}}{=} \\
&[\mathsf{min}(\mathsf{ctx\_addresses}(\alpha) \cap [0, \infty)), \mathsf{max}(\mathsf{ctx\_addresses}(\alpha) \cap [0, \infty)) + 1]
\end{aligned}
$$

Definition 98 (A trace compatible with a program's data segment).

$$
\begin{aligned}
&\mathsf{data\_segment\_compatible\_trace}(\alpha, \Sigma, \Delta, \mathit{modIDs}) \stackrel{\mathsf{def}}{=} \\
&\mathsf{min}(\mathsf{shareable\_data\_segment\_ctx}(\alpha)) > \mathsf{max}(\mathsf{static\_addresses}(\Sigma, \Delta, \mathit{modIDs}))
\end{aligned}
$$

Definition 99 (A trace satisfies monotonic sharing).

$$
\begin{aligned}
&\mathsf{monotonic\_sharing}(\alpha) \stackrel{\mathsf{def}}{=} \\
&\forall i.\ \mathsf{mem}(\alpha(i+1)) \supseteq \mathsf{mem}(\alpha(i))
\end{aligned}
$$

Definition 100 (A trace satisfies no-deallocation).

$$
\begin{aligned}
&\mathsf{no\_dealloc}(\alpha) \stackrel{\mathsf{def}}{=} \\
&\forall i.\ \mathsf{nalloc}(\alpha(i+1)) \leq \mathsf{nalloc}(\alpha(i))
\end{aligned}
$$

Definition 101 (Syntactically-sane trace).

$$
\begin{aligned}
&\mathsf{syntactically\_sane}(\alpha, \Sigma, \Delta, \mathit{modIDs}) \stackrel{\mathsf{def}}{=} \\
&\alpha \in \texttt{Alt}\checkmark^* \wedge \\
&\mathsf{no\_dealloc}(\alpha) \wedge \\
&\mathsf{monotonic\_sharing}(\alpha) \wedge \\
&\mathsf{data\_segment\_compatible\_trace}(\alpha, \Sigma, \Delta, \mathit{modIDs})
\end{aligned}
$$

Definition 102 (Global variables of the module `mainModule`).

$$
\begin{aligned}
\texttt{mainGlobalVars}(\alpha) \overset{\text{def}}{=}\; & \{\text{“}static\_universal\_array\text{”}, \text{“}current\_trace\_index\text{”}\} \\
& \uplus \{\text{“}arg\_store\_\texttt{tIdx\_fid\_arg}\text{”} \mid \texttt{tIdx} \in [0, |\alpha|) \wedge \\
& \quad \texttt{fid} \in \texttt{contextFunIDs}(\alpha) \wedge \\
& \quad \texttt{arg} \in [0, \texttt{nArgs}(\texttt{fid}, \alpha)\,)\,\} \\
& \uplus \{\text{“}snapshot\_\texttt{tIdx\_addr}\text{”} \mid \texttt{tIdx} \in [0, |\alpha|) \wedge \\
& \quad \texttt{addr} \in \texttt{sharedAddresses}(\alpha) \\
& \uplus \{\text{“}own\_allocation\_ptr\_\texttt{tIdx}\text{”} \mid \texttt{tIdx} \in [0, |\alpha|)\}
\end{aligned}
$$

Before we give Definition 105 of the functions defined by the `mainModule`, we explain intuitively what these functions are for. The purpose of the `mainModule` is to perform various bookkeeping tasks. All the bookkeeping *data* is stored in the global variables `mainGlobalVars` which are statically allocated (because we know upfront as a function of the trace α what variables we need). Thus, for the bookkeeping, no extra memory allocation is performed. This is important because memory allocation is an observable event. And, we do not want the bookkeeping that our source context will perform to interfere with the events observable by the source program. Remember that intuitively our goal is that observable source-level events mimic the target-level observable events precisely.

The bookkeeping tasks are initiated whenever the `mainModule` is informed that a call/return to any of the context's modules took place.

Definition 103 (The function `readAndIncrementTraceIdx`).

$$
\begin{aligned}
\texttt{readAndIncrementTraceIdx} \overset{\text{def}}{=}\; & (mainModule, \\
& readAndIncerementTraceIdx, \\
& [ptrRetVal], \\
& [\;], \\
& [\\
& \mathsf{Assign}\;\; ptrRetVal\;\; current\_trace\_index, \\
& \mathsf{Assign}\;\; \mathsf{addr}(current\_trace\_index)\;\; current\_trace\_index + 1, \\
& \mathsf{Return} \\
&])
\end{aligned}
$$

Definition 104 (The functions `saveArgs`).

$$
\begin{aligned}
\texttt{saveArgs}(\texttt{fid}, \texttt{tIdx}, \alpha) \overset{\text{def}}{=}\; & (mainModule, \\
& saveArgs\_\texttt{fid\_tIdx}, \\
& [argVal\_\texttt{i} \mid \texttt{i} \;\in\; [0, \texttt{nArgs}(\texttt{fid}, \alpha)]\;], \\
& [\;], \\
& [\mathsf{Assign}\;\; \mathsf{addr}(arg\_store\_\texttt{tIdx\_fid\_i})\;\; argVal\_\texttt{i} \;\mid\; \texttt{i} \;\in\; [0, \texttt{nArgs}(\texttt{fid}, \alpha))\;] \\
& \texttt{++} \\
& [\mathsf{Return}]\;)
\end{aligned}
$$

Definition 105 (Functions of the module `mainModule`).

$$\texttt{mainModuleFuncs}(\alpha) \stackrel{\text{def}}{=} \{\texttt{readAndIncrementTraceIdx}\} \cup \{\texttt{saveArgs}(\texttt{fid}, \alpha) \mid \texttt{fid} \in \texttt{contextFunIDs}(\alpha)\}$$

Definition 106 (Constructing dereferences from path).

$$\mathsf{construct\_derefs} : \overline{\mathbb{Z}} \to \mathcal{E} \to \mathcal{E}$$

$$\mathsf{construct\_derefs}([\,], \mathit{expr}) \stackrel{\text{def}}{=} \mathit{expr}$$

$$\mathsf{construct\_derefs}(\mathit{off} :: p, \mathit{expr}) \stackrel{\text{def}}{=} \mathsf{construct\_derefs}(p, \mathsf{deref}(\mathit{expr}[\mathit{off}]))$$

Definition 107 (Constructing path to target address).

$$\mathsf{path} : (\{\delta\} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}) \to \mathbb{Z} \to \mathit{DataMemory} \to \overline{\mathbb{Z}}$$
$$\mathsf{path\_depthlimited} : (\{\delta\} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}) \to \mathbb{Z} \to \mathit{DataMemory} \to \mathbb{N} \to \overline{\mathbb{Z}}$$
$$\mathsf{find} : \forall \alpha, \beta.\ \overline{\alpha} \to (\alpha \to \mathsf{Option}\ \beta) \to \mathsf{Option}\ (\alpha \times \beta)$$

$$\begin{aligned}
&\mathsf{find}\ [\,]\ _ \stackrel{\text{def}}{=} \mathsf{None}\\
&\mathsf{find}\ (x :: xs)\ f \stackrel{\text{def}}{=} \mathsf{case}\ f(x)\ \mathsf{of}\\
&\quad \mid \mathsf{Some}\ y \ \to\ \mathsf{Some}\ (x, y)\\
&\quad \mid \mathsf{None} \ \to\ \mathsf{find}\ xs\ f
\end{aligned}$$

$$\begin{aligned}
&\mathsf{path_depthlimited}((\delta, \sigma, e, _), a, \mathcal{M}_d, -1) \stackrel{\text{def}}{=} [\,]\\
&\mathsf{path_depthlimited}((\delta, \sigma, e, _), a, \mathcal{M}_d, k+1) \stackrel{\text{def}}{=}\\
&\quad \mathsf{if}\ a \in [\sigma, e)\ \mathsf{then}\ [a - \sigma]\\
&\quad \mathsf{else\ let}\ f = \lambda x.\ \mathsf{case}\ \mathcal{M}_d(x)\ \mathsf{of}\\
&\qquad\quad \mid (\delta, \sigma', e', _) \ \to\ \mathsf{let}\ p = \mathsf{path_depthlimited}((\delta, \sigma', e', _), a, \mathcal{M}_d, k)\ \mathsf{in}\\
&\qquad\qquad \mathsf{case}\ p\ \mathsf{of} \mid [\,] \ \to\ \mathsf{None} \mid _ \ \to\ \mathsf{Some}\ p\\
&\qquad\quad \mid _ \ \to\ \mathsf{None}\\
&\qquad\quad \mathsf{in\ case\ find}\ [\sigma, e)\ f\ \mathsf{of}\\
&\qquad\qquad \mid \mathsf{None} \ \to\ [\,]\\
&\qquad\qquad \mid \mathsf{Some}\ (a', p) \ \to\ [a' - \sigma] \mathbin{+\!\!+} p\\
&\mathsf{path}((\delta, \sigma, e, _), a, \mathcal{M}_d) \stackrel{\text{def}}{=} \mathsf{path_depthlimited}((\delta, \sigma, e, _), a, \mathcal{M}_d, |\mathcal{M}_d|)
\end{aligned}$$

Definition 108 (Construct address back-translation for addresses reachable from a capability argument).

$$\mathsf{cap\_arg\_reachable\_map} : (\{\delta\} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}) \to \mathit{DataMemory} \to \mathit{VarID} \to (\mathbb{Z} \to \mathcal{E})$$

$$\mathsf{cap\_arg\_reachable\_map}(dc, \mathcal{M}_d, \mathit{vid}) \stackrel{\text{def}}{=} \bigcup_{a\ \in\ \mathsf{reachable\_addresses}(\{dc\}, \mathcal{M}_d)} a \mapsto \mathsf{construct\_derefs}(\mathsf{path}(dc, a, \mathcal{M}_d), \mathit{vid})$$

Definition 109 (Construct address back-translation map from a call-/return to- context label)**.**

$$\overset{\ltimes}{\cup} : \forall \alpha, \beta.\ (\alpha \to \beta) \to (\alpha \to \beta) \to (\alpha \to \beta)$$

$$m_1 \overset{\ltimes}{\cup} m_2 \stackrel{\mathsf{def}}{=} m_1[a \mapsto m_2(a) \mid a \in \mathsf{dom}(m_2)]$$

$$\mathsf{args\_back\_translate} : \lambda \to \mathbb{N} \to (\mathbb{Z} \to \mathcal{E})$$

$$\mathsf{args\_back\_translate}(\mathtt{call}(mid, fid)\overline{v}!\mathcal{M}_d, n, cur\_idx) \stackrel{\mathsf{def}}{=}$$

$$\overset{\ltimes}{\bigcup}\{\mathsf{cap\_arg\_reachable\_map}(v, \mathcal{M}_d, arg\_mid\_fid\_i\_cur\_idx) \mid i \in [1, \mathtt{len}(\overline{v})] \land v = \overline{v}(i)\}$$

Notice that Definition 109 provides a way for finding a valid capability for any reachable address (i.e., including also for every shared address). We assume that relying on this definition, we can define functions that using these capabilities read the shared locations and stores them in *mainModule*'s book-keeping variables.

Definition 110 (Diverging block of code)**.**

$$\mathsf{diverge} \stackrel{\mathsf{def}}{=} [\mathsf{JumpIfZero}\ 0\ 0]$$

Definition 111 (Converging block of code)**.**

$$\mathsf{converge} \stackrel{\mathsf{def}}{=} [\mathsf{Exit}]$$

Definition 112 (If-then-else in **ImpMod**)**.**

$$\textit{ifnotzero-then-else} : \mathcal{E} \to \overline{Cmd} \to \overline{Cmd} \to \overline{Cmd}$$

$$\begin{aligned}
&\textit{ifnotzero-then-else}(e_{cond}, cmds_{then}, cmds_{else}) \stackrel{\mathsf{def}}{=}\\
&[\mathsf{JumpIfZero}\ e_{cond}\ |cmds_{then}| + 2]\\
&{+\!+}\ cmds_{then}\\
&{+\!+}\ [\mathsf{JumpIfZero}\ 0\ |cmds_{else}| + 1]\\
&{+\!+}\ cmds_{else}
\end{aligned}$$

Definition 113 (Switch-block for integers in **ImpMod**)**.**

$$\textit{switch} : \mathcal{E} \to \overline{\mathbb{Z}} \to \overline{\overline{Cmd}} \to \overline{Cmd}$$

$$\textit{switch}(\_, [\,], [\,]) \stackrel{\mathsf{def}}{=} [\,]$$

$$\begin{aligned}
&\textit{switch}(e_{cond}, z :: zl, cmdsl :: cmdsl_per_val) \stackrel{\mathsf{def}}{=}\\
&\textit{ifnotzero-then-else}(e_{cond} - z, \textit{switch}(e_{cond}, zl, cmdsl_per_val), cmdsl)
\end{aligned}$$

Definition 114 (Upcoming commands at an execution state)**.**

$$\mathsf{upcoming\_commands} \subseteq ProgState \times \overline{Cmd}$$

$$\begin{aligned}
&\mathsf{upcoming_commands}(s, \overline{cmds}) \iff\\
&s.pc = (fid, n, _) \land \forall i \in [0, |\overline{cmds}|).\ \mathtt{commands}(s.Fd(fid))(n + i) = \overline{cmds}(i)
\end{aligned}$$

Lemma 159 (If-then-else construction is correct)**.**

$$\begin{aligned}
&\forall s, \Sigma, \Delta, \beta, MVar, Fde_{cond}, cmds_{then}, cmds_{else}, cmds.\\
&\mathsf{upcoming_commands}(s, \textsf{\textit{ifnotzero-then-else}}(e_{cond}, cmds_{then}, cmds_{else}) \mathbin{+\!+} cmds)\\
&\;\Longrightarrow\\
&\quad(e_{cond}, \Sigma, \Delta, \beta, MVar, Fd, s.Mem, s.\Phi, s.pc \Downarrow 0 \implies\\
&\quad\Sigma; \Delta; \beta; MVar; Fd \vdash s \to s' \wedge \mathsf{upcoming_commands}(s', cmds_{else} \mathbin{+\!+} cmds)) \wedge\\
&\quad(e_{cond}, \Sigma, \Delta, \beta, MVar, Fd, s.Mem, s.\Phi, s.pc \Downarrow v \;\wedge\; v \neq 0 \implies\\
&\quad\Sigma; \Delta; \beta; MVar; Fd \vdash s \to s' \wedge \mathsf{upcoming_commands}(s', cmds_{then}))
\end{aligned}$$

Proof. Follows from Definitions 112 and 114 and rules Jump-zero and Jump-non-zero. □

Lemma 160 (Switch construction is correct)**.**

$$\begin{aligned}
&\forall i, s, \Sigma; \Delta; \beta; MVar; Fd, e_{cond}, zlist, cmdslist.\\
&|zlist| = |cmdslist| \wedge\\
&\mathsf{upcoming_commands}(s, \textsf{\textit{switch}}(e_{cond}, zlist, cmdslist)) \wedge\\
&e_{cond}, \Sigma, \Delta, \beta, MVar, Fd, s.Mem, s.\Phi, s.pc \Downarrow zlist(i)\\
&\;\Longrightarrow\\
&\exists s'.\ \Sigma; \Delta; \beta; MVar; Fd \vdash s \to^{i+1} s' \wedge\\
&\mathsf{upcoming_commands}(s', cmdslist(i))
\end{aligned}$$

Proof. Prove it by nested induction on $zlist$ and on i after unfolding Definition 113 and then inversion using rule Evaluate-expr-binop. Follows from Lemma 159. □

Lemma 161 (A $\mathsf{converge}$ block leads to a terminal state)**.**

$$\forall s.\ \mathsf{upcoming\_commands}(s, \mathsf{converge}) \implies \exists s_t.\ s \to^* s_t \;\wedge\; \vdash_t s_t$$

Proof. Follows by Definition 40 of a terminal state "$\vdash_t$", after unfolding Definitions 110 and 114, and taking s_t to be s. □

Lemma 162 (A $\mathsf{diverge}$ block does not lead to a terminal state)**.**

$$\forall s.\ \mathsf{upcoming\_commands}(s, \mathsf{diverge}) \implies \nexists s_t.\ s \to^* s_t \;\wedge\; \vdash_t s_t$$

Proof. Follows by unfolding Definitions 110 and 114, then simulating execution and noticing from case Jump-zero that the following holds by induction on $s \to^* s'$:

$$\forall s'.\ s \to^* s' \implies \mathsf{upcoming\_commands}(s', [\mathsf{JumpIfZero}\ 0\ 0])$$

Thus, by Definition 40, we get our thesis. □

Lemma 163 (Effect of calling `readAndIncrementTraceIdx`).

$\forall K_{mod}, K_{fun}, \overline{mods}, \Sigma; \Delta; \beta; MVar; Fd, s, \alpha, v, vid.$
$\mathsf{emulating\_modules}(\alpha) = \overline{mods} \ \wedge$
$K_{mod}; K_{fun}; \overline{mods} \ltimes \_; \Sigma; \Delta; \beta; MVar; Fd \vdash_{exec} s \ \wedge$
$s.Mem(\Delta(mainModule).1 + \beta(current\_trace\_index, \bot, mainModule).1) = v \ \wedge \ v \in \mathbb{Z} \ \wedge$
$\mathsf{upcoming\_commands}(s, [\mathsf{Call} \ \ readAndIncrementTraceIdx \ \ \mathsf{addr}(vid)] \mathbin{+\!\!+} cmds) \wedge$
$vid \notin \mathtt{localIDs}(Fd(pc.fid)) \cup \mathtt{args}(Fd(pc.fid)) \ \wedge$
$\Sigma(mainModule).1 + s.\Phi(mainModule) + 1 < \Sigma(mainModule).2 \ \wedge$
$\mathsf{addr}(vid), \Sigma; \Delta; \beta; MVar; Fd \Downarrow (\delta, \sigma, e, \mathit{off}) \ \wedge$
$[\sigma, e) \cap \Sigma(\mathtt{moduleID}(Fd(pc.fid))) = \emptyset \ \wedge \ \sigma \leq \sigma + \mathit{off} < e \ \wedge$
$\mathtt{moduleID}(Fd(pc.fid)) \neq mainModule$
$\implies$
$\exists s'. \ \Sigma; \Delta; \beta; MVar; Fd \vdash s \rightarrow^4 s' \ \wedge$
$s'.Mem = s.Mem$
$\quad [\Sigma(mainModule).1 + s.\Phi(mainModule) + 1$
$\quad\ + \beta(ptrRetVal, readAndIncrementTraceIdx, mainModule) \mapsto \_]$
$\quad [\Delta(mainModule).1 + \beta(current\_trace\_index, \bot, mainModule).1] \mapsto v + 1]$
$\quad [\Delta(\mathtt{moduleID}(Fd(pc.fid))).1 + \beta(vid, \bot, \mathtt{moduleID}(Fd(pc.fid))).1 \mapsto v] \ \wedge$
$s'.\Phi = s.\Phi \ \wedge$
$\mathsf{upcoming\_commands}(s', cmds)$

Proof.

- We first show $\exists s_1. \ s \rightarrow s_1$.
  - We apply rule Call obtaining the following subgoals:
    - $\mathtt{commands}(Fd(pc.fid))(pc.n) = \mathsf{Call} \ fid_{call} \ \overline{e}$
      Immediate by unfolding Definition 114 instantiating $fid_{call} = readAndIncrementTraceIdx$.
    - Assuming $modID = \mathtt{moduleID}(Fd(fid_{call}))$, and $frameSize = \mathtt{frameSize}(Fd(fid_{call}))$, we prove:
      $\Sigma(modID).1 + \Phi(modID) + frameSize < \Sigma(modID).2$
      By Definition 103, we know $\mathtt{frameSize}(Fd(readAndIncrementTraceIdx)) = 1$.
      Thus, after substitution in the goal, it becomes immediate by assumptions.
    - $\mathsf{addr}(vid), \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow (\delta, \sigma, e, \mathit{off})$
    - $[\sigma, e) \cap \Sigma(curModID) = \emptyset$
      These two goals are immediate by assumption.
  - And we know by unfolding the assumptions using Definition 124 then Definitions 89, 103 and 105 that we obtain s_1 with
    (S1-UPCOMING-CMDS):
    $\mathsf{upcoming\_commands}(s_1, [$
    $\mathsf{Assign} \ \ ptrRetVal \ \ current\_trace\_index,$
    $\mathsf{Assign} \ \ \mathsf{addr}(current\_trace\_index) \ \ current\_trace\_index + 1,$
    Return
    $])$
    (S1-PC):
    $s_1.pc = (readAndIncrementTraceIdx, 0)$

(S1-STK):
$s_1.stk = [s.pc] \texttt{++}\ s.stk$
(S1-PHI):
$s_1.\Phi = s.\Phi[mainModule \mapsto s.\Phi(mainModule) + 1]$
(S1-MEM):
$s_1.Mem = s.Mem[\Sigma(mainModule).1 + s_1.\Phi(mainModule)+$
$\beta(ptrRetVal, readAndIncrementTraceIdx, mainModule).1 \mapsto (\delta, \sigma, e, off)]$

- So, now we show $\exists s_2.\ s_1 \to s_2$
  - We apply rule Assign-to-var-or-arr to obtain the following subgoals:
    * $\texttt{commands}(Fd(s_1.pc.fid))(s_1.pc.n) = \mathsf{Assign}\ e_l\ e_r$
      Immediate by (S1-PC) and (S1-UPCOMING-CMDS) after unfolding using Definition 114.
    * $ptrRetVal, \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow (\delta, \sigma, e, off)$
      We apply rule Evaluate-expr-var then Evaluate-expr-addr-local obtaining the following subgoals:
      · $ptrRetval \in \texttt{localIDs}(Fd(readAndIncrementTraceIdx)) \cup$
      $\texttt{args}(Fd(readAndIncrementTraceIdx))$
      Immediate by Definition 103.
      · $\beta(ptrRetVal, readAndIncrementTraceIdx, mainModule) = [\sigma_p, e_p)$
      · $\Sigma(mainModule).1 + \Phi(mainModule) + \sigma_p < \Sigma(mainModule).1 + \Phi(mainModule) + e_p$
      These are immediate by inversion of the assumptions using rules Exec-state-src, and Well-formed program and parameters.
      · $s_1.Mem(\Sigma(mainModule).1 + s1.\Phi(mainModule) + \sigma_p) = (\delta, \sigma, e, off)$
      Immediate by (S1-MEM).
    * $current\_trace\_index, \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow v$
      We apply rule Evaluate-expr-var, and the generated subgoals are immediate by assumptions.
    * $\forall s', e'.\ v = (\delta, s', e', \_) \implies \_$
      Vacuously true by assumptions.
    * $\sigma \leq \sigma + off < e$
      Immediate by assumptions.
  - And we know that s_2 satisfies
    (S2-MEM):
    $s_2.Mem = s_1.Mem[\sigma + off \mapsto v]$, and
    (S2-PC):
    $s_2.pc = (readAndIncrementTraceIdx, 1)$

- Next, we show $\exists s_3.\ s_2 \to s_3$
  - We apply rule Assign-to-var-or-arr to obtain the following subgoals:
    * $\texttt{commands}(Fd(s_2.pc.fid))(s_2.pc.n) = \mathsf{Assign}\ e_l\ e_r$
      Immediate by (S2-PC), (S1-PC) and (S1-UPCOMING-CMDS) after unfolding using Definition 114.
    * $\texttt{addr}(current\_trace\_index), \Sigma, \Delta, \beta, MVar, Fd, Mem, \Phi, pc \Downarrow (\delta, \sigma_c, e_c, off_c)$
      We apply rule Evaluate-expr-addr-module and obtain the following subgoals:
      · $current\_trace\_index \notin \texttt{localIDs}(Fd(readAndIncrementTraceIdx)) \cup$
      $\texttt{args}(Fd(readAndIncrementTraceIdx))$
      Immediate by Definition 103.

        · $current\_trace\_index \in MVar(mainModule)$
        Immediate by Definitions 89 and 102.
        · $\beta(current\_trace\_index, \perp, mainModule) = [\sigma'_c, e'_c)$
        Immediate by inversion of the assumptions using rules Exec-state-src, and Well-formed program and parameters.

    We obtain the following substitutions:
    $\sigma_c = \Delta(mainModule).1 + \sigma'_c, e_c = \Delta(mainModule).1 + e'_c, \mathit{off}_c = 0$

    * $current\_trace\_index + 1, \Sigma; \Delta; \beta; MVar; Fd \Downarrow v + 1$
      We apply rule Evaluate-expr-binop then rules Evaluate-expr-const in parallel with (rule Evaluate-expr-var then Evaluate-expr-addr-module).
      All subgoals are immediate by assumptions and Definitions 89 and 102.
    * $\forall s', e'.\ v + 1 = (\delta, s', e', \_) \implies \_$
      Vacuously true by disjointness of $\mathbb{Z}$ and data capabilities.
    * $\sigma_c < e_c$
      Immediate by inversion of the assumptions using rules Exec-state-src, and Well-formed program and parameters.

  – And we know that s_3 satisfies
    (S3-MEM):
    $s_3.Mem = s_2.Mem[\sigma_c + \mathit{off}_c \mapsto v + 1]$, and
    (S3-PC):
    $s_3.pc = (readAndIncrementTraceIdx, 2)$

- And finally, we show $\exists s_4.\ s_3 \to s_4$

  – We apply rule Return to obtain the following subgoals:

    * $s_3.stk \neq \texttt{nil}$
      This is immediate by (S1-STK), and observing that $s_3.stk = s_2.stk = s_1.stk$.

  – By (S3-PC), Definition 103, and rule Return, we know
    (S4-PHI):
    $s_4.\Phi = s_3.\Phi[mainModule \mapsto s_3.\Phi(mainModule) - 1]$, and
    (S4-PC):
    $s_4.pc = \texttt{inc}(\texttt{top}(s_3.stk))$

- Thus, we know $s \to^4 s_4$.

- We now show:
  $s_4.Mem =$
  $s.Mem[\Sigma(mainModule).1 + s.\Phi(mainModule) + 1 \mapsto \_]$
  $[\Delta(mainModule).1 + \beta(current\_trace\_index, \perp, mainModule).1] \mapsto v + 1]$
  $[\Delta(\texttt{moduleID}(Fd(pc.fid))).1 + \beta(vid, \perp, \texttt{moduleID}(Fd(pc.fid))).1 \mapsto v]$

  This follows by (S1-MEM), (S2-MEM), (S3-MEM) and by noticing that $s_4.Mem = s_3.Mem$ by rule Return.

  But, it remains to show that the update locations are distinct:
  $\Delta(\texttt{moduleID}(Fd(pc.fid))).1 + \beta(vid, \perp, \texttt{moduleID}(Fd(pc.fid))).1 \neq$
  $\Delta(mainModule).1 + \beta(current\_trace\_index, \perp, mainModule).1 \neq$
  $\Sigma(mainModule).1 + s.\Phi(mainModule) + 1$

  This follows from assumption $\texttt{moduleID}(Fd(pc.fid)) \neq mainModule$ and by the disjointness preconditions given by inversion of the assumptions using Exec-state-src and Well-formed program and parameters.

- Then, we show:
  $s_4.\Phi = s.\Phi$

  This follows from (S1-PHI) and (S4-PHI) together with observing that $s_3.\Phi = s_2.\Phi = s_1.\Phi$.

- Finally, we show $\mathsf{upcoming\_commands}(s_4, \mathit{cmds})$

  Immediate by substitution from (S4-PC), (S1-STK), $s_3.\mathit{stk} = s_2.\mathit{stk} = s_1.\mathit{stk}$, and assumption $\mathsf{upcoming\_commands}(s, [\mathsf{Call}\ \ \mathit{readAndIncrementTraceIdx}\ \ \mathsf{addr}(\mathit{vid})] \mathbin{+\!\!+} \mathit{cmds})$ after unfolding it using Definition 114.

This concludes the proof of Lemma 163.

□

Definition 115 (Independent set of assignments)**.** *A set of assignment commands is independent if all assigned addresses are distinct.*

Lemma 164 (Effect of calling `saveArgs`)**.**

$\forall K_{mod}, K_{fun}, \overline{mods}, \Sigma; \Delta; \beta; \mathit{MVar}; \mathit{Fd}, s, \alpha, \mathtt{fid}, \mathtt{tIdx}, n, \overline{\mathtt{argNames}}, \overline{\mathtt{argVals}}$

$\mathsf{emulating\_modules}(\alpha) = \overline{mods}\ \wedge$

$K_{mod}; K_{fun}; \overline{mods} \ltimes \_; \Sigma; \Delta; \beta; \mathit{MVar}; \mathit{Fd} \vdash_{exec} s\ \wedge$

$n = \mathtt{nArgs}(\mathtt{fid}, \alpha) = |\overline{\mathtt{argNames}}| = |\overline{\mathtt{argVals}}|\ \wedge$

$s.pc.\mathit{fid} = \mathtt{fid}\ \wedge$

$\forall i \in [0, n).$

$\quad \overline{\mathtt{argNames}}(i) \in \mathtt{args}(\mathit{Fd}(s.pc.\mathit{fid}))\ \wedge$

$\quad s.\mathit{Mem}(\Sigma(\mathtt{moduleID}(\mathit{Fd}(s.pc.\mathit{fid}))).1 + s.\Phi(\mathtt{moduleID}(\mathit{Fd}(s.pc.\mathit{fid})))$

$\quad + \beta(\overline{\mathtt{argNames}}(i), s.pc.\mathit{fid}, \mathtt{moduleID}(\mathit{Fd}(s.pc.\mathit{fid}))).1) = \overline{\mathtt{argVals}}(i)\ \wedge$

$\quad \overline{\mathtt{argVals}}(i) = (\delta, \_, \_, \_) \implies [\overline{\mathtt{argVals}}(i).\sigma, \overline{\mathtt{argVals}}(i).e) \cap \Sigma(\mathtt{moduleID}(\mathit{Fd}(s.pc.\mathit{fid}))) = \emptyset$

$\mathsf{upcoming\_commands}(s, [\mathsf{Call}\ \ \mathit{saveArgs}\_\mathtt{fid}\_\mathtt{tIdx}\ \ \overline{\mathtt{argNames}}] \mathbin{+\!\!+} \mathit{cmds})\ \wedge$

$\Sigma(\mathit{mainModule}).1 + s.\Phi(\mathit{mainModule}) + n < \Sigma(\mathit{mainModule}).2\ \wedge$

$\mathtt{moduleID}(\mathit{Fd}(s.pc.\mathit{fid})) \neq \mathit{mainModule}$

$\implies$

$\exists s'.\ \Sigma; \Delta; \beta; \mathit{MVar}; \mathit{Fd} \vdash s \rightarrow^{n+2} s'\ \wedge$

$s'.\mathit{Mem} = s.\mathit{Mem}$

$\quad [\Delta(\mathit{mainModule}).1 + \beta(\mathit{arg\_store}\_\mathtt{tIdx}\_\mathtt{fid}\_\mathtt{i}, \bot, \mathit{mainModule}).1 \mapsto \overline{\mathtt{argVals}}(i) \mid i \in [0, n)]$

$\quad [\Sigma(\mathit{mainModule}).1 + s.\Phi(\mathit{mainModule}) + \beta(\overline{\mathtt{argNames}}(i), \mathit{saveArgs}\_\mathtt{fid}\_\mathtt{tIdx}, \mathit{mainModule})$

$\quad\ \mapsto \overline{\mathtt{argVals}}(i) \mid i \in [0, n)]\ \wedge$

$s'.\Phi = s.\Phi\ \wedge$

$\mathsf{upcoming\_commands}(s', \mathit{cmds})$

Proof.

- We prove $\exists s_{-1}.\ s \rightarrow s_{-1}$.

  - We choose s_{-1} such that:
    (S-MINUS-1-PC):
    $s_{-1}.pc = (\mathit{saveArgs}\_\mathtt{fid}\_\mathtt{tIdx}, 0)$
    (S-MINUS-1-STK):
    $s_{-1}.\mathit{stk} = s.\mathit{stk} \mathbin{+\!\!+} [s.pc]$,

(S-MINUS-1-MEM):
$s_{-1}.Mem = s.Mem$
$[\Sigma(mainModule).1 + s.\Phi(mainModule) + \beta(argVal\_i, saveArgs\_\texttt{fid\_tIdx}, mainModule).1$
$\mapsto \overline{\texttt{argVals}}(i) \mid i \in [0,n)]$,
(S-MINUS-1-PHI):
$s_{-1}.\Phi = s.\Phi[mainModule \mapsto s.\Phi(mainModule) + n]$, and
(S-MINUS-1-UPCOMING-CMDS):
upcoming_commands(s_{-1},
[Assign addr($arg\_store\_\texttt{tIdx\_fid\_i}$) $argVal\_\texttt{i}$ | $\texttt{i} \in [0,n)$]
++
[Return])

- We apply rule Call to obtain the following subgoals:
  * $\texttt{commands}(Fd(pc.fid))(pc.n) = \mathsf{Call}\ fid_{call}\ \overline{e}$
    Immediate by unfolding Definition 114 instantiating $fid_{call} = saveArgs\_\texttt{fid\_tIdx}$.
  * Assuming $modID = \texttt{moduleID}(Fd(fid_{call}))$, and $frameSize = \texttt{frameSize}(Fd(fid_{call}))$, we prove:
    $\Sigma(modID).1 + \Phi(modID) + frameSize < \Sigma(modID).2$
    By Definition 104, we know $\texttt{frameSize}(Fd(saveArgs\_\texttt{fid\_tIdx})) = n$.
    Thus, after substitution in the goal, it becomes immediate by assumptions.
  * $\forall i \in [0,n).\ \overline{\texttt{argNames}}(i), \Sigma, \Delta, \beta, MVar, Fd, s.Mem, s.\Phi, s.pc \Downarrow \overline{\texttt{argVals}}(i)$
    Here, we fix an arbitrary i, and we apply rule Evaluate-expr-var then Evaluate-expr-addr-local obtaining the following subgoals:
    · $\overline{\texttt{argNames}}(i) \in \texttt{args}(Fd(s.pc.fid))$
      Immediate by assumptions.
    · $\beta(\overline{\texttt{argNames}}(i), s.pc.fid, \texttt{moduleID}(Fd(s.pc.fid))) = [\sigma, e)$
      Immediate by assumptions.
    · $\phi = \Sigma(\texttt{moduleID}(Fd(s.pc.fid))).1 + \Phi(\texttt{moduleID}(Fd(s.pc.fid)))$
      This subgoal is immediate by the fact that the given keys exist in the maps Σ, Φ, and β which is immediate by inverting the assumptions using Exec-state-src then Well-formed program and parameters.
    · $\sigma < e$
      Follows by inversion of the assumptions using Well-formed program and parameters.
    · $s.Mem(\sigma + \phi) = \overline{\texttt{argVals}}(i)$
      Follows by assumptions.
  * $\forall i \in [0,n).\ \overline{\texttt{argVals}}(i) = (\delta, \_, \_, \_) \implies [\overline{\texttt{argVals}}(i).\sigma, \overline{\texttt{argVals}}(i).e) \cap \Sigma(curModID) = \emptyset$
    Immediate by assumptions.
  * The remaining subgoals are immediate by (S-MINUS-1-STK), (S-MINUS-1-MEM), and (S-MINUS-1-PHI). Also, (S-MINUS-1-UPCOMING-CMDS) becomes a proof obligation after substitution, and it follows immediately by Definition 104.

- Next, our goal is:
  $\exists s_{n-1}.\ s_{-1} \to^n s_{n-1}\ \wedge$
  $s_{n-1}.\Phi = s_{-1}.\Phi\ \wedge$
  $s_{n-1}.Mem = s_{-1}.Mem$
  $[\Delta(mainModule).1 + \beta(arg\_store\_\texttt{tIdx\_fid\_i}, \perp, mainModule).1 \mapsto \overline{\texttt{argVals}}(i) \mid i \in [0,n)]\ \wedge$
  $s_{n-1}.stk = s_{-1}.stk\ \wedge$
  upcoming_commands(s_{n-1}, [Return])

  We distinguish the following two cases for n:

– **Case** $n = 0$**:**
Here, our goal is immediate by choosing s_{-1}, and by the reflexivity of $\rightarrow^0$.

– **Case** $n > 0$**:**

∗ First, we prove the following by induction on k:

$k \in [0, n) \implies$
$\exists s_k, s_{k-1}.$
$s_{k-1} \rightarrow s_k \land$
$s_k.Mem = s_{k-1}.Mem$
$\quad[\Delta(mainModule).1 + \beta(arg\_store\_\texttt{tIdx\_fid\_k}, \bot, mainModule).1 \mapsto \overline{\texttt{argVals}}(k)] \land$
$s_k.\Phi = s_{-1}.\Phi \land$
$s_k.stk = s_{-1}.stk \land$
upcoming_commands(s_k,
$\quad$[Assign addr($arg\_store\_\texttt{tIdx\_fid\_i}$) $argVal\_\texttt{i}$ | i $\in$ $[k+1, n)$]
$\quad$++
$\quad$[Return])

· **Base case (**$k = 0$**):**
We choose the state s_{-1} that is given above in the proof of $s \rightarrow s_{-1}$.

We choose s_0 such that:
(S0-STK):
$s_0.stk = s_{-1}.stk$,
(S0-MEM):
$s_0.Mem = s_{-1}.Mem$
$[\Delta(mainModule).1 + \beta(arg\_store\_\texttt{tIdx\_fid\_0}, \bot, mainModule).1 \mapsto \overline{\texttt{argVals}}(0)]$,
(S0-PHI):
$s_0.\Phi = s_{-1}.\Phi$
Now we prove that $s_{-1} \rightarrow s_0$ and
upcoming_commands(s_0,
[Assign addr($arg\_store\_\texttt{tIdx\_fid\_i}$) $argVal\_\texttt{i}$ | i $\in$ $[1, n)$]
++
[Return])
Using (S-MINUS-1-UPCOMING-CMDS), and Definition 114 we know:
upcoming_commands(s_{-1}, [Assign addr($arg\_store\_\texttt{tIdx\_fid\_}0$) $argVal\_0$])
Thus, we apply rule Assign-to-var-or-arr to our goal obtaining the following subgoals:

1. addr($arg\_store\_\texttt{tIdx\_fid\_}0$), $\Sigma, \Delta, \beta, MVar, Fd, s_{-1}.Mem, s_{-1}.\Phi, s_{-1}.pc \Downarrow (\delta, \sigma_0, e_0, off_0)$
   Here, we apply rule Evaluate-expr-addr-module all of whose subgoals follow by simplification after unfolding the lemma assumptions using Definitions 89, 102, 104, 105 and 124, inversion of the lemma assumptions using Well-formed program and parameters, and substitution using (S-MINUS-1-PC).
2. $argVal\_0, \Sigma, \Delta, \beta, MVar, Fd, s_{-1}.Mem, s_{-1}.\Phi, s_{-1}.pc \Downarrow \overline{\texttt{argVals}}(0)$
   Here, we apply rules Evaluate-expr-var then Evaluate-expr-addr-local obtaining the following subgoals:
   (a) $argVal\_0 \in \texttt{args}(saveArgs\_\texttt{fid\_tIdx})$
   Immediate by Definition 104 and the assumptions about n after unfolding the assumptions using Definitions 89 and 124.
   (b) $s_{-1}.Mem(\Sigma(mainModule).1 + s_{-1}.\Phi(mainModule)$
   $+ \beta(argVal\_0, saveArgs\_\texttt{fid\_tIdx}, mainModule).1) = \overline{\texttt{argVals}}(0)$

Immediate by (S-MINUS-1-MEM).

(c) The remaining subgoals follow from Well-formed program and parameters by unfolding the assumptions using first Exec-state-src.

3. $\sigma_0 < e_0$
   Follows from unfolding the assumptions using Exec-state-src then Well-formed program and parameters.

4. $\overline{\texttt{argVals}}(0) = (\delta, \_, \_, \_) \implies [\overline{\texttt{argVals}}(0).\sigma, \overline{\texttt{argVals}}(0).e) \cap \Sigma(\mathit{mainModule}) = \emptyset$
   Assume the contrary (for contradiction)
   (ARGVAL0-IS-STACK-CAPABILITY):
   $\overline{\texttt{argVals}}(0) = (\delta, \_, \_, \_) \wedge [\overline{\texttt{argVals}}(0).\sigma, \overline{\texttt{argVals}}(0).e) \cap \Sigma(\mathit{mainModule}) \neq \emptyset$
   Now, by inversion of the assumptions using Exec-state-src,
   we know by instantiating the precondition "**Stack addresses only live on the stack**"
   using (ARGVAL0-IS-STACK-CAPABILITY) that
   (CONTRADICTORY-LOCATION-FOR-ARGVAL0):
   $\forall a\ s.\mathit{Mem}(a) = \overline{\texttt{argVals}}(0) \implies a \in \Sigma(\mathit{mainModule})$
   Now, we instantiate (CONTRADICTORY-LOCATION-FOR-ARGVAL0) using the assumption
   $s.\mathit{Mem}(\Sigma(\texttt{moduleID}(\mathit{Fd}(s.\mathit{pc}.\mathit{fid}))).1 + s.\Phi(\texttt{moduleID}(\mathit{Fd}(s.\mathit{pc}.\mathit{fid})))$
   $+ \beta(\overline{\texttt{argNames}}(0), s.\mathit{pc}.\mathit{fid}, \texttt{moduleID}(\mathit{Fd}(s.\mathit{pc}.\mathit{fid}))).1) = \overline{\texttt{argVals}}(0))$
   to conclude that:
   $\Sigma(\texttt{moduleID}(\mathit{Fd}(s.\mathit{pc}.\mathit{fid}))).1 + s.\Phi(\texttt{moduleID}(\mathit{Fd}(s.\mathit{pc}.\mathit{fid})))$
   $+ \beta(\overline{\texttt{argNames}}(0), s.\mathit{pc}.\mathit{fid}, \texttt{moduleID}(\mathit{Fd}(s.\mathit{pc}.\mathit{fid}))).1 \in \Sigma(\mathit{mainModule})$
   We can derive a contradiction from this last statement using the preconditions of Well-formed program and parameters together with the lemma assumption $\texttt{moduleID}(\mathit{Fd}(s.\mathit{pc}.\mathit{fid})) \neq \mathit{mainModule}$.

5. The remaining subgoals that justify the choice of (S0-MEM), (S0-STK), and (S0-PHI) are immediate.

· **Inductive case** $(0 < k < n)$**:**
  The induction step is very similar to the base case. We avoid repetition.

This concludes the inductive proof.

∗ We instantiate the inductive statement obtained above with $k = n - 1$ obtaining our goal.

This concludes the proof for **case** $n > 0$.

- Now, it remains to show that:
  $\exists s'.\ s_{n-1} \to s' \wedge$
  $s'.\mathit{Mem} = s_{n-1}.\mathit{Mem} \wedge$
  $s'.\Phi = s_{n-1}.\Phi[\mathit{mainModule} \mapsto s_{n-1}.\Phi(\mathit{mainModule}) - n]$

  Here, we apply rule Return obtaining the following subgoals:

  – $s_{n-1}.\mathit{stk} \neq \texttt{nil}$, and
  – upcoming_commands(s', cmds)

  These follow from (S-MINUS-1-STK), and (S-N-1-STK) together with our lemma assumption about the upcoming commands of s after unfolding Definition 114.

This concludes the proof of Lemma 164. □

Definition 116 (Logged memory correct).

$$
\begin{aligned}
&\mathsf{logged\_mem\_correct}(s)_{\alpha,i,\Delta,\beta} \stackrel{\mathsf{def}}{=}\\
&\forall j, a.\\
&j < i \wedge\\
&a \in \mathsf{dom}(\mathsf{mem}(\alpha(j)))\\
&\implies\\
&s.Mem(\Delta(mainModule).1 + \beta(snapshot\_j\_a, \perp, mainModule)) = \mathsf{mem}(\alpha(j))(a)
\end{aligned}
$$

Definition 117 (Arguments saved correctly).

$$
\begin{aligned}
&\mathsf{arguments\_saved\_correctly}(s)_{\alpha,i,\Delta,\beta} \stackrel{\mathsf{def}}{=}\\
&\forall j, argIdx, fid.\\
&j < i \wedge\\
&\alpha(j) = \mathtt{call}(\_, fid)\overline{v}!\_ \ \wedge\\
&argIdx \in [0, \mathtt{len}(\overline{v}))\\
&\implies\\
&s.Mem(\Delta(mainModule).1 + \beta(arg\_store\_j\_fid\_argIdx, \perp, mainModule)) = \overline{v}(argIdx)
\end{aligned}
$$

Definition 118 (Allocation pointers saved).

$$
\begin{aligned}
&\mathsf{allocation\_pointers\_saved}(s)_{\alpha,i,\Delta,\beta} \stackrel{\mathsf{def}}{=}\\
&\forall j.\\
&j < i \wedge\\
&\alpha(j) \in \overset{\bullet}{?}\\
&\implies\\
&s.Mem(\Delta(mainModule).1 + \beta(own\_allocation\_ptr\_j, \perp, mainModule)) = (\delta, \mathsf{nalloc}(\alpha(j)) + 1, \mathsf{nalloc}(\alpha(j-1)), 0)
\end{aligned}
$$

Claim 37 (There is a source function that does allocations according to $\mathsf{allocation\_pointers\_saved}$).

$$
\begin{aligned}
&\exists cmd.\\
&\mathsf{upcoming\_commands}(s, [cmd]) \wedge\\
&\mathsf{allocation\_pointers\_saved}(s)_{\alpha,i,\Delta,\beta} \wedge\\
&s \to s'\\
&\implies\\
&\mathsf{allocation\_pointers\_saved}(s')_{\alpha,i+1,\Delta,\beta}
\end{aligned}
$$

Definition 119 (Emulate call or return or exit command of i-th output action).

$$
\begin{aligned}
&\mathsf{emulate\_ith\_action\_last\_cmd}(\alpha, i) \stackrel{\mathsf{def}}{=}\\
&[\mathsf{Call}\ fid\ [\mathsf{emulate\_value}(\overline{v}(i), \alpha(:i)) \mid i \in [0, \mathtt{len}(\overline{v}))\ ]\ ] && where\ \alpha(i) = \mathtt{call}(\_, fid)?\overline{v}\_\\
&[\mathsf{Return}] && where\ \alpha(i) = \mathtt{ret}\_\\
&[\mathsf{Exit}] && where\ \alpha(i) = \checkmark
\end{aligned}
$$

(Notice that the existence of a function $\mathsf{emulate\_value}(\overline{v}(i), \alpha(:i))$ relies on Definition 108.)

Definition 120 (Emulate i-th output action).

$$\begin{aligned}
&\mathsf{emulate_ith_action}(\alpha, i, \texttt{mid}, \texttt{fid}) \stackrel{\mathsf{def}}{=} \\
&[\mathsf{Call}\ \mathit{readAndIncrementTraceIdx}\ \mathsf{addr}(\mathit{current_trace_index}_\texttt{mid}), \\
&\ \mathsf{Call}\ \mathit{saveArgs}_\texttt{fid}_i\ \texttt{argNamesList}(\alpha, i, \texttt{fid}), \\
&\ \mathsf{Call}\ \mathit{saveSnapshot}_i - \mathit{1}, \\
&\ \mathsf{Call}\ \mathit{doAllocations}_i, \\
&\ \mathsf{Call}\ \mathit{mimicMemory}_i \\
&] \\
&\ \text{++} \\
&\ \mathsf{emulate_ith_action_last_cmd}(\alpha, i)
\end{aligned}$$

Definition 121 (Responses for suffix).

$$\begin{aligned}
&\mathsf{emulate_responses_for_suffix}(\alpha, i, \texttt{mid}, \texttt{fid}) \stackrel{\mathsf{def}}{=} \\
&\mathit{switch}(\\
&\mathit{current_trace_index}_\texttt{mid}, \\
&[i, i+2, i+4, \cdots, |\alpha|), \\
&\ [\mathsf{emulate_ith_action}(\alpha, j, \texttt{mid}, \texttt{fid})\ \text{++}\ \mathsf{emulate_responses_for_suffix}(\alpha, j+2, \texttt{mid}, \texttt{fid}) \mid j \in [i, i+2, i+4, \cdots, |\alpha|)] \\
&)
\end{aligned}$$

Lemma 165 (Adequacy of $\mathsf{emulate\_responses\_for\_suffix}$).

$$\begin{aligned}
&(\mathbb{C}_{emul}, \Delta_{emul}, \Sigma_{emul}, \beta_{emul}, K_{modemul}, K_{funemul}) = \mathsf{emulate}(\alpha, p, \Delta, \Sigma, \beta)\ \wedge \\
&p' = \mathbb{C}_{emul} \ltimes p\ \wedge \\
&(\Delta', \Sigma', \beta', K'_{mod}, K'_{fun}) = \\
&\quad (\Delta \uplus \Delta_{emul}, \Sigma \uplus \Sigma_{emul}, \beta \cup \beta_{emul}, K_{mod} \uplus K_{modemul}, K_{fun} \uplus K_{funemul})\ \wedge \\
&p' \vdash_{exec} s\ \wedge \\
&\mathsf{upcoming_commands}(s, \mathsf{emulate_responses_for_suffix}(\alpha, i, \texttt{moduleID}(\mathsf{fd_map}(p)(s.pc.fid)), s.pc.fid))\ \wedge \\
&\implies \\
&\exists s'.\ s, _ \xrightarrow{\alpha(i)}_{[p]} s', _
\end{aligned}$$

Proof.
After unfolding Definition 121 and Definition 120, the goal follows by successively instantiating Lemma 163 then Lemma 164, and Claim 37, together with unproved assumptions about the existence of functions *saveSnapshot*, and *mimicMemory* which rely on Definition 108. ☐

Definition 122 (Emulating function).

$$\begin{aligned}
&\mathsf{emulating_function}(\alpha, \texttt{mid}, \texttt{fid}) \stackrel{\mathsf{def}}{=} \\
&(\\
&\texttt{mid}, \\
&\texttt{fid}, \\
&[\mathit{argVal}_\texttt{i}\ \mid\ \texttt{i}\ \in\ [0, \texttt{nArgs}(\texttt{fid}, \alpha))\ ], \\
&[\ ], \\
&\mathsf{emulate_responses_for_suffix}(\alpha, 0, \texttt{mid}, \texttt{fid}) \\
&)
\end{aligned}$$

Definition 123 (Emulating module).

$$
\begin{aligned}
&\mathsf{emulating\_module}(\alpha, \texttt{mid}) \stackrel{\mathsf{def}}{=} \\
&(\\
&\texttt{mid}, \\
&[\mathit{current\_trace\_index}\_\texttt{mid}], \\
&\{\mathsf{emulating\_function}(\alpha, \texttt{mid}, \texttt{fid}) \mid \alpha(i) = \texttt{call}(\texttt{mid}, \texttt{fid})\_!\_\} \\
&)
\end{aligned}
$$

Definition 124 (Emulating modules).

$$
\mathsf{emulating\_modules}(\alpha) \stackrel{\mathsf{def}}{=} [\texttt{mainModule}(\alpha)]{+}{+}[\mathsf{emulating\_module}(\alpha, \texttt{mid}) \mid \texttt{mid} \in \texttt{contextModIDs}(\alpha)]
$$

Definition 125 (The emulating context).

$$
\begin{aligned}
&\mathsf{emulate}(\alpha, p, \Delta, \Sigma, \beta, K_{mod}, K_{fun}) \stackrel{\mathsf{def}}{=} \\
&(\mathsf{emulating\_modules}(\alpha), \\
&\mathsf{data\_segment\_map\_extension}(p, \mathsf{emulating\_modules}(\alpha), \Delta), \\
&\mathsf{stack\_map\_extension}(p, \mathsf{emulating\_modules}(\alpha), \Sigma), \\
&\mathsf{variable\_bounds\_extension}(p, \mathsf{emulating\_modules}(\alpha), \beta), \\
&\mathsf{Kmod\_extension}(p, \mathsf{emulating\_modules}(\alpha), K_{mod}), \\
&\mathsf{Kfun\_extension}(p, \mathsf{emulating\_modules}(\alpha), K_{fun}))
\end{aligned}
$$

Lemma 166 (The emulating context is linkable and loadable).

$$
\begin{aligned}
&(\mathbb{C}_{emul}, \Delta_{emul}, \Sigma_{emul}, \beta_{emul}, K_{modemul}, K_{funemul}) = \mathsf{emulate}(\alpha, p, \Delta, \Sigma, \beta, K_{mod}, K_{fun}) \wedge \\
&\mathbb{C} \ltimes [\![p]\!]_{\Delta,\Sigma,\beta,K_{mod},K_{fun}} = \lfloor t' \rfloor \wedge \\
&\mathsf{initial\_state}(t' + \omega, \mathsf{main\_module}(t')), \emptyset \xrightarrow{\alpha}_{[[\![p]\!]_{\Delta,\Sigma,\beta,K_{mod},K_{fun}}],\nabla} s_t', \varsigma' \\
&\implies \\
&\exists \overline{m}. \\
&\mathbb{C}_{emul}[p]_{\Delta,\Sigma} = \overline{m} \wedge \\
&\texttt{wfp\_params}(\overline{m}, \\
&\quad \Delta \uplus \Delta_{emul}, \Sigma \uplus \Sigma_{emul}, \beta \cup \beta_{emul}, K_{mod} \uplus K_{modemul}, K_{fun} \uplus K_{funemul}) \wedge \\
&\mathsf{main\_module}(\overline{m}) \neq \mathit{None}
\end{aligned}
$$

Proof.
(Sketch) By inverting the assumption using rule valid-linking, and unfolding it using Definition 125 then Definitions 90 and 91,
we are able to instantiate rule Valid-linking-src satisfying our goal after instantiating Lemma 92 using our assumption.

Then, subgoal `wfp_params` follows by applying rule Well-formed program and parameters where all the generated subgoals follow by unfolding Definition 125 recursively (assuming there are suitable definitions for extending the linking and loading information, i.e., suitable definitions for data_segment_map_extension, stack_map_extension, variable_bounds_extension, Kmod_extension, and Kfun_extension). ☐

Definition 126 (Emulate invariants)**.**

$\mathsf{emulate\_invariants}(s)_{\alpha,i,p,\Delta,\Sigma,\beta} \stackrel{\mathsf{def}}{=}$
$(\forall pc \in s.stk, s'.\ s'.pc = pc \implies$
$\exists j.\ j \leq i\ \wedge\ \mathsf{upcoming\_commands}(s', \mathsf{emulate\_responses\_for\_suffix}(\alpha, j, \mathtt{moduleID}(\mathsf{fd\_map}(p)(pc.fid)), pc.fid)))\ \wedge$
$(\alpha(i) \in \dot{?} \implies$
$\exists j.\ j \leq i\ \wedge\ \mathsf{upcoming\_commands}(s, \mathsf{emulate\_responses\_for\_suffix}(\alpha, j, \mathtt{moduleID}(\mathsf{fd\_map}(p)(s.pc.fid)), s.pc.fid)))\ \wedge$
$\mathsf{logged\_mem\_correct}(s)_{\alpha,i,\Delta,\beta}\ \wedge$
$\mathsf{arguments\_saved\_correctly}(s)_{\alpha,i,\Delta,\beta}\ \wedge$
$\mathsf{allocation\_pointers\_saved}(s)_{\alpha,i,\Delta,\beta}$

Lemma 167 (Initial state of emulate satisfies emulate_invariants)**.**

$$\begin{aligned}
&(\mathbb{C}_{emul}, \Delta_{emul}, \Sigma_{emul}, \beta_{emul}, K_{modemul}, K_{funemul}) = \mathsf{emulate}(\alpha, p, \Delta, \Sigma, \beta)\ \wedge\\
&p' = \mathbb{C}_{emul} \ltimes p\ \wedge\\
&(\Delta', \Sigma', \beta', K'_{mod}, K'_{fun}) =\\
&\quad (\Delta \uplus \Delta_{emul}, \Sigma \uplus \Sigma_{emul}, \beta \cup \beta_{emul}, K_{mod} \uplus K_{modemul}, K_{fun} \uplus K_{funemul})\ \wedge\\
&s_{emul} = \mathsf{initial_state}(p', \Delta', \Sigma', \mathsf{main_module}(p'))\\
&\implies\\
&\mathsf{emulate_invariants}(s_{emul})_{\alpha,0,p,\Delta',\Sigma',\beta'}
\end{aligned}$$

Proof.
By unfolding Definition 126, we have the following subgoals:

- Vacuous subgoal because $s.stk = \mathtt{nil}$
- Assuming $\alpha(i) \in \dot{?}$, show:
  $\mathsf{upcoming\_commands}(s_{emul}, \mathsf{emulate\_responses\_for\_suffix}(\alpha, i, \mathtt{moduleID}(\mathsf{fd\_map}(p)(s_{emul}.pc.fid)), s_{emul}.pc.fid))$
  Follows from unfolding Definition 125 then Definition 124 then Definition 123.
- $\mathsf{logged\_mem\_correct}(s_{emul})_{\_,0,\_}$
  Immediate after unfolding Definition 116 by noticing that $\alpha(-1) = \bot$.
- $\mathsf{arguments\_saved\_correctly}(s_{emul})_{\_,0,\_}$
  Immediate after unfolding Definition 117 by noticing that $\alpha(-1) = \bot$.
- $\mathsf{allocation\_pointers\_saved}(s_{emul})_{\_,0,\_}$
  Immediate after unfolding Definition 118 by noticing that $\alpha(-1) = \bot$.

∎

Lemma 168 (Adequacy of emulate_invariants)**.**

$$\begin{aligned}
&\mathbb{C}_{emul} \ltimes p \vdash_{exec} s_{emul}\ \wedge\\
&\alpha(i) \in \dot{?}\ \wedge\\
&\mathsf{emulate_invariants}(s_{emul})_{\alpha,i,p,\Delta,\Sigma,\beta}\\
&\implies\\
&\exists s'_{emul}.\ s_{emul}, _ \xrightarrow{\alpha(i)}_{[p]} s'_{emul}, _
\end{aligned}$$

Proof.
After unfolding the assumption using Definition 126, the goal follows from Lemma 165. □

Lemma 169 (Preservation of emulate_invariants)**.**

$$\begin{aligned}
&\mathbb{C}_{emul} \ltimes p \vdash_{exec} s_{emul} \wedge \\
&\mathsf{emulate_invariants}(s_{emul})_{\alpha,i,p,\Delta,\Sigma,\beta} \\
&s_{emul}, _ \xrightarrow{\alpha(i)}_{[p]} s'_{emul}, _ \\
&\implies \\
&\mathsf{emulate_invariants}(s_{emul})_{\alpha,i+1,p,\Delta,\Sigma,\beta}
\end{aligned}$$

Proof.
(Sketch) After unfolding Definition 121 then instantiating Lemma 160, this should follow from Lemma 163 then Lemma 164, and Claim 37, together with unproved assumptions about the existence of functions *saveSnapshot*, and *mimicMemory* which rely on Definition 108. □

6.4 Trace-Indexed Cross-Language (TrICL) simulation relation

Definition 127 (Trace-Indexed Cross-Language (TrICL) simulation relation)**.**

$$\begin{aligned}
&\mathsf{TrICL}(s_{emul}, s_{compiled}, s_{given}, \varsigma)_{\alpha,i,p,\mathbb{C}_{emul},\Delta,\Sigma,\beta} \stackrel{\mathsf{def}}{=} \\
&\mathsf{emulate_invariants}(s_{emul})_{\alpha,i,p,\Delta,\Sigma,\beta} \wedge \\
&s_{emul} \cong_{\mathbb{C}_{emul} \ltimes p} s_{compiled} \wedge \\
&(\alpha(i) \in \dot{!} \implies s_{compiled}, \varsigma \approx_{[\![p]\!]} s_{given}, \varsigma) \wedge \\
&(\alpha(i) \in \dot{?} \implies s_{compiled}, \varsigma \sim_{[\![p]\!],\alpha,i} s_{given}, \varsigma)
\end{aligned}$$

where

$$s_1, \varsigma_1 \sim_{[p],\alpha,i} s_2, \varsigma_2 \stackrel{\mathsf{def}}{=} s_1, \varsigma_1 \sim_{[p],\rho_{[p]}(s_1,\varsigma_1)} s_2, \varsigma_2$$

(Notice that at border states (s, ς) where program part p is ***not*** *executing, the expression $\rho_{[p]}(s, \varsigma)$ gives the domain of the private memory of p at the border.)*

Lemma 170 (TrICL satisfies the alternating simulation condition)**.**

$$\begin{aligned}
&\alpha \in \mathtt{Alt} \wedge \\
&\mathsf{TrICL}(s_{emul}, s_{compiled}, s_{given}, \varsigma)_{\alpha,i,p,\mathbb{C}_{emul},\Delta,\Sigma,\beta} \wedge \\
&_ \ltimes s_{emul} \vdash_{exec} s_{emul} \wedge \\
&\mathbb{C}_{given} \ltimes [\![p]\!] \vdash_{border} \alpha[: i], s_{given}, \varsigma \wedge \\
&s_{given}, \varsigma \xrightarrow{\alpha(i)}_{[\![p]\!]} s'_{given}, \varsigma' \\
&\implies \\
&\exists s'_{compiled}, s'_{emul}. \\
&s_{compiled}, \varsigma \xrightarrow{\alpha(i)}_{[\![p]\!]} s'_{compiled}, \varsigma' \wedge \\
&s_{emul}, \varsigma \xrightarrow{\alpha(i)}_{[p]} s'_{emul}, \varsigma' \wedge \\
&\mathsf{TrICL}(s'_{emul}, s'_{compiled}, s'_{given}, \varsigma')_{\alpha,i+1,p,\mathbb{C}_{emul},\Delta,\Sigma,\beta}
\end{aligned}$$

Proof.
By $\alpha \in \texttt{Alt}$ (unfolding Definition 69),
it suffices to distinguish the following two cases:

- **Case** $\alpha(i) \in \dot{!}$**:**

  By unfolding the assumption using Definition 127, we have:
  (EMUL-INVAR): $\mathsf{emulate\_invariants}(s_{emul})_{\alpha,i,p,\Delta,\Sigma,\beta}$

  (COMPILER-REL): $s_{emul} \cong_{\mathbb{C}_{emul} \ltimes p} s_{compiled}$

  (STRONG-SIM): $s_{compiled}, \varsigma \approx_{[\llbracket p \rrbracket]} s_{given}, \varsigma$

  Here, we can instantiate Lemma 149 (Weakening of strong similarity) using (STRONG-SIM) and the given step to obtain:

  (G1): $s_{compiled}, \varsigma \xrightarrow{\alpha(i)}_{[\llbracket p \rrbracket]} s'_{compiled}, \varsigma'$
  and
  (G2): $s'_{compiled}, \varsigma' \sim_{\llbracket p \rrbracket, \alpha, i+1} s'_{given}, \varsigma'$

  But then using (G1), and (COMPILER-REL), we can instantiate Lemma 130 (lifted compiler backward-simulation) to obtain:

  (G3): $s_{emul}, \varsigma \xrightarrow{\alpha(i)}_{[p]} s'_{emul}, \varsigma'$
  and
  (G4): $s'_{emul} \cong_{\mathbb{C}_{emul} \ltimes p} s'_{compiled}$

  But then using (G3) and (EMUL-INVAR), we can instantiate Lemma 169 (preservation of the emulate invariants) to obtain:

  (G5): $\mathsf{emulate\_invariants}(s_{emul})_{\alpha,i+1,p,\Delta,\Sigma,\beta}$

  After (G1), (G2), (G3), (G4), and (G5), no subgoals remain, so this concludes this case.

- **Case** $\alpha(i) \in \dot{?}$**:**

  By unfolding the assumption using Definition 127, we have:
  (EMUL-INVAR): $\mathsf{emulate\_invariants}(s_{emul})_{\alpha,i,p,\Delta,\Sigma,\beta}$

  (COMPILER-REL): $s_{emul} \cong_{\mathbb{C}_{emul} \ltimes p} s_{compiled}$

  (WEAK-SIM): $s_{compiled}, \varsigma \sim_{\llbracket p \rrbracket, \alpha, i} s_{given}, \varsigma$

  Here, we can instantiate Lemma 168 (adequacy of the emulate invariants) using (EMUL-INVAR) to obtain:

  (G1): $s_{emul}, \varsigma \xrightarrow{\alpha(i)}_{[p]} s'_{emul}, \varsigma'$
  (Notice that $\alpha(i)$ determines ς')

  Then, we can instantiate Lemma 169 (preservation of the emulate invariants) using (G1) above to obtain:
  (G2): $\mathsf{emulate\_invariants}(s_{emul})_{\alpha,i+1,p,\Delta,\Sigma,\beta}$

  Also, using the same emulating step (G1), together with (COMPILER-REL), we can instantiate Lemma 129 (lifted compiler forward-simulation) to obtain:

(G3): $s_{compiled}, \varsigma \xrightarrow{\alpha(i)}_{[\![p]\!]} s'_{compiled}, \varsigma'$
and
(G4): $s'_{emul} \cong_{\mathbb{C}_{emul} \ltimes p} s'_{compiled}$

But then using the last step (G3), the given step (from the assumption), and (WEAK-SIM) we can instantiate the strengthening lemma (Lemma 153) to obtain:
(G5): $s'_{compiled}, \varsigma' \approx_{[\![p]\!]} s'_{given}, \varsigma'$

After (G1), (G2), (G3), (G4), and (G5), no subgoals remain, so this concludes this case.

This concludes the proof of Lemma 170. □

Lemma 171 (Initial states are TrICL-related)**.**

$$
\begin{aligned}
&\alpha \in Tr_{\omega,\nabla}([\![p]\!]_{\Delta,\Sigma,\beta,K_{mod},K_{fun}}) \wedge \\
&(\mathbb{C}_{emul}, \Delta_{emul}, \Sigma_{emul}, \beta_{emul}, K_{modemul}, K_{funemul}) = \mathsf{emulate}(\alpha, p, \Delta, \Sigma, \beta) \wedge \\
&p' = \mathbb{C}_{emul} \ltimes p \wedge \\
&(\Delta', \Sigma', \beta', K'_{mod}, K'_{fun}) = \\
&\quad (\Delta \uplus \Delta_{emul}, \Sigma \uplus \Sigma_{emul}, \beta \cup \beta_{emul}, K_{mod} \uplus K_{modemul}, K_{fun} \uplus K_{funemul}) \wedge \\
&s_{emul} = \mathsf{initial\_state}(p', \Delta', \Sigma', \mathsf{main\_module}(p')) \wedge \\
&s_{compiled} = \mathsf{initial\_state}([\![p']\!]_{\Delta',\Sigma',\beta',K'_{mod},K'_{fun}}, \mathsf{main\_module}(p')) \wedge \\
&s_{given} = \mathsf{initial\_state}(\mathbb{C}_{given} \ltimes [\![p]\!]_{\Delta,\Sigma,\beta,K_{mod},K_{fun}}, \mathsf{main\_module}(p')) \\
&\implies \\
&\mathsf{TrICL}(s_{emul}, s_{compiled}, s_{given}, \emptyset)_{\alpha,0,p,\mathbb{C}_{emul},\Delta',\Sigma',\beta'}
\end{aligned}
$$

Proof.
By unfolding Definition 127, we have the following subgoals:

- emulate invariants:

  Follows by instantiating Lemma 167.

- $s_{emul} \cong_{\mathbb{C}_{emul} \ltimes p} s_{compiled}$:

  Follows by instantiating Lemma 100.

- Assuming $\alpha(0) \in \dot{!}$, show $s_{compiled}, \varsigma \approx_{[\![p]\!]} s_{given}, \varsigma$:

  Here, know by relying on Lemma 166, and by distinguishing the cases for $\alpha(i)$ that: $s_{given}.\mathsf{pcc} \not\subseteq \mathsf{dom}(\mathbb{C}_{given}.\mathcal{M}_c)$.

  Thus, our goal follows by Lemma 135.

- Assuming $\alpha(0) \in \dot{?}$, show $s_{compiled}, \varsigma \sim_{[\![p]\!],\alpha,i} s_{given}, \varsigma$:

  Here, know by relying on Lemma 166 and by distinguishing the cases for $\alpha(i)$ that: $s_{given}.\mathsf{pcc} \subseteq \mathsf{dom}(\mathbb{C}_{given}.\mathcal{M}_c)$

  Thus, our goal follows by Lemma 136.

□

Lemma 172 (TrICL-related states are co-terminal)**.**

$$
\begin{aligned}
&\mathsf{TrICL}(s_{emul}, s_{compiled}, s_{given}, \_)_{\_} \\
&\implies \\
&(\vdash_t s_{emul} \iff \vdash_t s_{compiled} \iff \vdash_t s_{given})
\end{aligned}
$$

Proof.
Follows from Lemma 103, and by unfolding Definition 127 then Definition 119. □

Lemma 173 (No trace is added by compilation)**.**

$$\alpha \in Tr_{\omega,\nabla,\Delta,\Sigma,\beta}(p) \impliedby \alpha \in Tr_{\omega,\nabla}(\llbracket p \rrbracket_{\Delta,\Sigma,\beta,K_{mod},K_{fun}})$$

Proof.
By assumption (unfolding Definition 72), we have (*):

$$\exists \mathbb{C}_{given}, t' : TargetSetup, s'_t : TargetState, \varsigma' : 2^{\mathbb{Z}}.$$
$$\mathbb{C}_{given} \ltimes \llbracket p \rrbracket_{\Delta,\Sigma,\beta,K_{mod},K_{fun}} = \lfloor t' \rfloor \wedge$$
$$\mathsf{initial\_state}(t' + \omega, \mathsf{main\_module}(t')), \emptyset \xrightarrow{\alpha}_{[\llbracket p \rrbracket_{\Delta,\Sigma,\beta,K_{mod},K_{fun}}],\nabla} s'_t, \varsigma'$$

And our goal (unfolding Definition 78) is:

$$\exists \mathbb{C}, \overline{m}, s', \varsigma', \Delta_{\mathbb{C}}, \Sigma_{\mathbb{C}}, \beta_{\mathbb{C}}.$$
$$\Delta' = \Delta \uplus \Delta_{\mathbb{C}} \ \wedge \ \Sigma' = \Sigma \uplus \Sigma_{\mathbb{C}} \ \wedge \ \beta' = \beta \cup \beta_{\mathbb{C}} \wedge$$
$$\mathbb{C}[p]_{\Delta',\Sigma'} = \overline{m} \ \wedge$$
$$\Sigma'; \Delta' + \omega; \beta'; \mathsf{mvar}(\overline{m}); \mathsf{fd\_map}(\overline{m}) \vdash \mathsf{initial\_state}(\overline{m}, \Delta' + \omega, \Sigma', \mathsf{main\_module}(\overline{m})), \emptyset \xrightarrow{\alpha}_{[p],\nabla} s', \varsigma'$$

We pick for our goal the following instantiation:
$\mathbb{C} := \mathbb{C}_{emul}, \Delta_{\mathbb{C}} := \Delta_{emul}, \Sigma_{\mathbb{C}} := \Sigma_{emul}, \beta_{\mathbb{C}} := \beta_{emul}$
where (**):
$(\mathbb{C}_{emul}, \Delta_{emul}, \Sigma_{emul}, \beta_{emul}, K_{modemul}, K_{funemul}) = \mathsf{emulate}(\alpha, p, \Delta, \Sigma, \beta)$

By instantiating Lemma 166 using (*) and (**), we know $\overline{m}$ exists, and that (WF-PARAMS):

$$\mathbb{C}_{emul}[p]_{\Delta,\Sigma} = \overline{m} \wedge$$
$$\mathtt{wfp\_params}(\overline{m}, \Delta \uplus \Delta_{emul}, \Sigma \uplus \Sigma_{emul}, \beta \cup \beta_{emul}, K_{mod} \uplus K_{modemul}, K_{fun} \uplus K_{funemul})$$

Using (WF-PARAMS), we obtain by instantiating rule Module-list-translation a compiled program:

$$p'_{compiled} = \llbracket p' \rrbracket_{\Delta \uplus \Delta_{emul}, \Sigma \uplus \Sigma_{emul}, \beta \cup \beta_{emul}, K_{mod} \uplus K_{modemul}, K_{fun} \uplus K_{funemul}}$$

Now, by instantiating Lemma 171 using our assumption and (WF-PARAMS) and (**), we have (INIT-TrICL):

$$\mathsf{TrICL}(\mathsf{initial\_state}(p', \Delta', \Sigma', \mathsf{main\_module}(p')),$$
$$\mathsf{initial\_state}(p'_{compiled}, \mathsf{main\_module}(p')),$$
$$\mathsf{initial\_state}(\mathbb{C}_{given} \ltimes \llbracket p \rrbracket_{\Delta,\Sigma,\beta,K_{mod},K_{fun}}, \mathsf{main\_module}(p')), \emptyset)_{\alpha,0,p,\mathbb{C}_{emul}}$$

By Lemma 109, and Lemma 172, it suffices to show the following for the alternating prefix $\alpha|_{\checkmark}$:

$$\forall i \in [0, |\alpha|_{\checkmark}|] \ \exists s'_{emul}, s'_{compiled}, s'_{given}, \varsigma_i.$$
$$\Sigma'; \Delta'; \beta'; \mathsf{mvar}(p'); \mathsf{fd\_map}(p') \vdash \mathsf{initial\_state}(p', \Delta', \Sigma', \mathsf{main\_module}(p')), \emptyset \xrightarrow{\alpha(0)\ldots\alpha(i)}_{[p],\nabla} s'_{emul}, \varsigma_i \wedge$$
$$\mathsf{initial\_state}(p'_{compiled}, \mathsf{main\_module}(p')), \emptyset \xrightarrow{\alpha(0)\ldots\alpha(i)}_{[\llbracket p \rrbracket_{\Delta,\Sigma,\beta,K_{mod},K_{fun}}],\nabla} s'_{compiled}, \varsigma_i \wedge$$
$$\mathsf{TrICL}(s'_{emul}, s'_{compiled}, s'_{given}, \varsigma_i)_{\alpha,i,p,\mathbb{C}_{emul}}$$

We are able to show the above sufficient subgoal by proving an inductive version of Lemma 170 (relying on Lemma 158):

- The base case follows from (INIT-TrICL) and instantiation of Lemma 170.
- The inductive case follows by instantiation of Claim 9 using (*) then Lemma 170, followed by instantiation of the following:
  Claim 21 and rule trace-steps-alternating-src for the source trace, and
  Claim 8 and rule trace-steps-alternating for the compiled trace.

This concludes the proof of Lemma 173. □

$$
\begin{tikzcd}
\llbracket p_{s1} \rrbracket \not\simeq \llbracket p_{s2} \rrbracket \arrow[r, dashed, "\text{Contrapositive of Lemma 121 by Lemmas 114, 117 and 122}"] \arrow[d, shift right, "\text{Lemma 114}"'] & p_{s1} \not\simeq p_{s2} \arrow[l, bend right, "\text{Contrapositive of Lemma 120}"'] \arrow[d, dashed, shift right, "\text{Corollary 13 by the cycle}"'] \\
\llbracket p_{s1} \rrbracket \overset{\mathrm{T}}{\neq} \llbracket p_{s2} \rrbracket \arrow[r, "\text{Lemma 122}"] \arrow[u, dashed, shift right, "\text{Corollary 12 by the cycle}"'] & p_{s1} \overset{\mathrm{T}}{\neq} p_{s2} \arrow[u, shift right, "\text{Lemma 117}"']
\end{tikzcd}
$$

Figure 13: The contrapositive of Lemma 121 ($\llbracket \cdot \rrbracket$ preserves contextual equivalence) follows from Lemma 114 (soundness of target trace equivalence), Lemma 122 (compilation preserves trace equivalence), and Lemma 117 (completeness of source trace equivalence). Also, the bent arrow (the contrapositive of Lemma 120 ($\llbracket \cdot \rrbracket$ reflects contextual equivalence)) closes the cycle. Thus, from the cycle, the two vertical dashed arrows follow. The left one (Corollary 12), together with Lemma 114, gives that the target traces are fully abstract. Similarly, the source ones are fully abstract by the right one Corollary 13 , together with Lemma 117.

7 Corollaries for free

7.1 Completeness of the trace semantics of CHERIExp

Corollary 12 (Completeness of target trace equivalence for contextual equivalence of compiled components)**.**

$$\llbracket p_1 \rrbracket_{\Delta,\Sigma,\beta,K_{mod},K_{fun}} \overset{\mathrm{T}}{=}_{\omega,\nabla} \llbracket p_2 \rrbracket_{\Delta,\Sigma,\beta,K_{mod},K_{fun}} \impliedby \llbracket p_1 \rrbracket_{\Delta,\Sigma,\beta,K_{mod},K_{fun}} \simeq_{\omega,\nabla} \llbracket p_2 \rrbracket_{\Delta,\Sigma,\beta,K_{mod},K_{fun}}$$

Proof. Follows from the cycle in Figure 13 (i.e., the contrapositive of our goal is immediate by instantiating Lemma 122 then Lemma 117 then Lemma 120). □

7.2 Soundness of the trace semantics of ImpMod

Corollary 13 (Soundness of source traces)**.**

$$\begin{aligned}
&\forall \overline{m_1}, \overline{m_2}, \tilde{\Delta}, \beta_1, \beta_2, \tilde{\Sigma}, \nabla, \Delta, \Sigma. \\
&\mathtt{dom}(\tilde{\Sigma}) = \{\mathtt{moduleID}(m) \mid m \in \overline{m_1}\} = \{\mathtt{moduleID}(m) \mid m \in \overline{m_2}\} \land \\
&\mathtt{dom}(\tilde{\Delta}) = \{\mathtt{moduleID}(m) \mid m \in \overline{m_1}\} = \{\mathtt{moduleID}(m) \mid m \in \overline{m_2}\} \land \\
&\beta_1, \overline{m_1} \overset{\mathrm{T}}{=}_{\nabla,\Delta,\Sigma} \beta_2, \overline{m_2} \\
&\implies \\
&\tilde{\Delta}, \beta_1, \overline{m_1} \simeq_{\tilde{\Sigma},\nabla} \tilde{\Delta}, \beta_2, \overline{m_2}
\end{aligned}$$

Proof. Follows from the cycle in Figure 13 (i.e., the contrapositive of our goal is immediate by instantiating Lemma 120—after compiling both programs, then Lemma 114, then Lemma 122). □

8 Note on non-commutative linking

The fact that we chose to define linking as non-commutative is just a side effect of trying to avoid some tedious proof [15], but linking being non-commutative is not really essential for security.

We use non-commutativity to require that the program parts are first all linked together and used as the right operand of the linking operator. The left operand then represents the context in which this program runs. Having distinguished the program of interest from its context, we then define linking in such a way that the context's data segment is placed in memory *after* the program's data segment. There is no security motivation for this enforced order; it just makes the proof easier: the construction of the emulating context will occupy a data segment whose size is in principle larger (due to meta-data) than the size of the data segment of the target context that we are emulating. This order of placing the data segments in memory ensures that this increase in size (due to metadata) does not impact the position of the program of interest's variables in memory (in a simulating run compared to a given run).

However, lots of the metadata we store is *redundant*—we store this redundant data to make our life simpler. But in principle, we do believe one should be able to prove that the *non-redundant* metadata will at every execution state always fit within a data segment of the original size (i.e., the size from the given run). By proving this, there will be no need to define linking to be non-commutative.

9 Example output of the source-to-source transformation

```
struct cheri_object main_obj;
static struct sandbox_object *main_objectp;


__attribute__((cheri_ccall))
__attribute__((cheri_method_suffix("_cap")))
__attribute__((cheri_method_class(main_obj)))
extern int main(int argc, char *argv[]);

int init(int argc, char *argv[])
{
      sandbox_chain_load("main", &main_objectp);
      main_obj = sandbox_object_getobject(main_objectp);

      main(argc, argv);
}
```

Listing 1: Source-to-source compilation output. Initialization module init.c

```
struct cheri_object lib1;
struct cheri_object lib2;

__attribute__((cheri_ccall))
__attribute__((cheri_method_class(lib1)))
int f1(void);

__attribute__((cheri_ccall))
__attribute__((cheri_method_class(lib2)))
int f2(void);

__attribute__((cheri_ccallee))
__attribute__((cheri_method_class(main_obj)))
int main(void);

__attribute__ ((constructor)) static void
sandboxes_init(void)
{
      lib2 = fetch_object("lib2");
      lib1 = fetch_object("lib1");
}

int main(void)
{
      f1();
      f2();

      return 0;
```

```
}
```

Listing 2: Source-to-source compilation output. Transformed main.c

```
extern struct cheri_object lib1;
struct cheri_object lib2;

__attribute__((cheri_ccallee))
__attribute__((cheri_method_class(lib1)))
int f1(void);

__attribute__((cheri_ccall))
__attribute__((cheri_method_class(lib2)))
int f2(void);

__attribute__ ((constructor)) static void
sandboxes_init(void)
{
      lib2 = fetch_object("lib2");
}

int f1(void)
{
      f2();
}
```

Listing 3: Source-to-source compilation output. Transformed lib1.c

```
extern struct cheri_object lib2;

__attribute__((cheri_ccallee))
__attribute__((cheri_method_class(lib2)))
int f2(void);

int f2(void)
{
      [..]
}
```

Listing 4: Source-to-source compilation output. Transformed lib2.c

References

[1] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe, "The cheri capability model: Revisiting risc in an age of risk," *SIGARCH Comput. Archit. News*, vol. 42, no. 3, pp. 457–468, Jun. 2014. [Online]. Available: http://doi.acm.org/10.1145/2678373.2665740

[2] R. N. M. Watson, P. G. Neumann, J. Woodruff, M. Roe, J. Anderson, D. Chisnall, B. Davis, A. Joannou, B. Laurie, S. W. Moore, S. J. Murdoch, R. Norton, S. Son, and H. Xia, "Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 6)," University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-907, Apr. 2017. [Online]. Available: http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-907.pdf

[3] A. El-Korashy, "A Formal Model for Capability Machines: An Illustrative Case Study towards Secure Compilation to CHERI," Max-Planck Institute for Software Systems, Saarbrücken, Tech. Rep., Sep. 2016. [Online]. Available: https://people.mpi-sws.org/~elkorashy/

[4] "Rigorous Engineering of Mainstream Systems," 2016, [Online; accessed 06-September-2016]. [Online]. Available: https://www.cl.cam.ac.uk/~pes20/rems/

[5] M. Abadi, "Protection in programming-language translations," in *International Colloquium on Automata, Languages, and Programming*. Springer, 1998, pp. 868–883.

[6] P. Agten, R. Strackx, B. Jacobs, and F. Piessens, "Secure compilation to modern processors," in *CSF '12*. IEEE, 2012, pp. 171 – 185. [Online]. Available: http://dx.doi.org/10.1109/CSF.2012.12

[7] M. Patrignani, D. Devriese, and F. Piessens, "On Modular and Fully-Abstract Compilation," in *Proceedings of the 29th IEEE Computer Security Foundations Symposium CSF 2016, Lisbon, Portugal*, ser. CSF 2016, 2016.

[8] M. Patrignani, P. Agten, R. Strackx, B. Jacobs, D. Clarke, and F. Piessens, "Secure compilation to protected module architectures," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 37, no. 2, p. 6, 2015.

[9] C. Fournet, N. Swamy, J. Chen, P.-E. Dagand, P.-Y. Strub, and B. Livshits, "Fully abstract compilation to javascript," *SIGPLAN Not.*, vol. 48, no. 1, pp. 371–384, Jan. 2013. [Online]. Available: http://doi.acm.org/10.1145/2480359.2429114

[10] A. Ahmed and M. Blume, "Typed closure conversion preserves observational equivalence," *SIGPLAN Not.*, vol. 43, no. 9, pp. 157–168, Sep. 2008. [Online]. Available: http://doi.acm.org/10.1145/1411203.1411227

[11] R. Jagadeesan, C. Pitcher, J. Rathke, and J. Riely, "Local memory via layout randomization," in *Proceedings of the 2011 IEEE 24th Computer Security Foundations Symposium*, ser. CSF '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 161–174. [Online]. Available: http://dx.doi.org/10.1109/CSF.2011.18

[12] M. Abadi and G. Plotkin, "On protection by layout randomization," in *CSF '10*. IEEE, 2010, pp. 337–351. [Online]. Available: http://dx.doi.org/10.1109/CSF.2010.30

[13] Y. Juglaret, C. Hriţcu, A. Azevedo de Amorim, and B. C. Pierce, "Beyond good and evil: Formalizing the security guarantees of compartmentalizing compilation," in *29th IEEE Symposium on Computer Security Foundations (CSF)*. IEEE Computer Society Press, Jul. 2016. [Online]. Available: http://arxiv.org/abs/1602.04503

[14] D. Devriese, M. Patrignani, and F. Piessens, "Fully-abstract compilation by approximate back-translation," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, 2016, pp. 164–177. [Online]. Available: http://doi.acm.org/10.1145/2837614.2837618

[15] T. C. Murray and P. C. van Oorschot, "BP: formal proofs, the fine print and side effects," in *2018 IEEE Cybersecurity Development, SecDev 2018, Cambridge, MA, USA, September 30 - October 2, 2018*, 2018, pp. 1–10. [Online]. Available: https://doi.org/10.1109/SecDev.2018.00009