Compositional Compiler Verification via Parametric Simulation

Georg Neis
MPI-SWS, Germany
neis@mpi-sws.org

Chung-Kil Hur
Seoul National University
gil.hur@cse.snu.ac.kr

Jan-Oliver Kaiser
MPI-SWS, Germany
janno@mpi-sws.org

Derek Dreyer
MPI-SWS, Germany
dreyer@mpi-sws.org

Viktor Vafeiadis
MPI-SWS, Germany
viktor@mpi-sws.org

Abstract

Compiler verification is essential for the construction of fully verified software, but most prior work (such as CompCert) has focused on verifying whole-program compilers. To support separate compilation and to enable linking of results from different verified compilers, it is important to develop a compositional notion of compiler correctness that is modular (preserved under linking), transitive (supports multi-pass composition), and flexible (applicable to a wide variety of languages and program transformations).

In this work, building on prior work of Hur et al. [7], we develop a novel approach to compositional compiler verification based on parametric inter-language simulations (PILS). PILS are as modular and flexible as state-of-the-art logical-relations models, but are transitive as well. We demonstrate the effectiveness of PILS by using them to verify Pilsner, a simple but non-trivial multi-pass optimizing compiler for an ML-like language programmed in Coq. This has been a significant undertaking, involving several person-years of work and around 40,000 lines of Coq.

1. Introduction

Most verification tools operate on programs written in high-level languages, which must be compiled down to machine-language programs prior to execution. The compiler is simply trusted to “preserve the semantics” of its source language (and hence preserve confidence in the high-level verification). Unfortunately, this trust is not well founded. For instance, recent work of Le et al. [11] identified 147 confirmed bugs in the industrial-strength GCC and LLVM compilers, of which 95 were violations of semantics preservation.

The goal of compiler verification is to eliminate the need for trust in compilation by providing a formal, machine-checked guarantee that a compiler is semantics-preserving. Toward this end, the most successful project so far has been CompCert [12], a verified optimizing compiler for an ML-like language programmed in Coq. An important question is how to verify separate compilation, and to enable linking of results from different compilers.

To enable verified separate compilation, semantics preservation should be defined at the level of modules, not just whole programs, and it should be preserved under linking. That is, for a module \( m_S \) refined by a target module \( m_T \), and a source module \( m'_S \) refined by a target module \( m'_T \), then the source-level linking of \( m_S \) and \( m'_S \) should be refined by the target-level linking of \( m_T \) and \( m'_T \). (This has been referred to as “horizontal compositionality”.)

- **Modularity**: To enable verified separate compilation, semantics preservation should be defined at the level of modules, not just whole programs, and it should be preserved under linking. That is, for a source module \( m_S \) refined by a target module \( m_T \), and a source module \( m'_S \) refined by a target module \( m'_T \), then the source-level linking of \( m_S \) and \( m'_S \) should be refined by the target-level linking of \( m_T \) and \( m'_T \). (This has been referred to as “horizontal compositionality”.)

- **Transitivity**: Proofs of semantic preservation should be transitive. That is, one should be able to prove a compiler correct by verifying its constituent passes independently and then linking the theorems together by transitivity. (This has been referred to as “vertical compositionality”.)

- **Flexibility**: It should be possible to prove semantics preservation for a wide range of compilers and program transformations, so that the results of different verified compilers (together with hand-optimized and hand-verified machine code) can be safely linked. In particular, we seek a notion of semantic preservation that imposes as few restrictions as possible on the kinds of program transformations and the kinds of source, intermediate, and target languages to which it is applicable.

**Contextual Refinement** The canonical approach to compositionality is to prove that the output of the transformation is a contextual refinement of the input. This means that the output refines the input when placed within an arbitrary enclosing program context.

While contextual refinement is inherently modular and transitive, it only applies if the input and output languages are the same, and is thus a non-starter for realistic compiler verification (unless combined with multi-language semantics, as discussed below).

**Logical Relations** Benton and Hur [3] proposed the idea of using logical relations to define compositional semantics preservation. In addition to being inherently modular, logical relations are highly flexible, having been used in the past as an effective technique for
proving correctness of a wide variety of program transformations in a wide variety of languages [2, 5]. Moreover, unlike contextual refinement, logical relations can be used to relate different source and target languages.

Hur and Dreyer [6] developed this idea further by formalizing the compositional correctness of a simple, single-pass compiler from an ML-like source language to an idealized assembly language. They additionally demonstrated the flexibility of their inter-language logical relations by using them to verify a contrived but illustrative example, wherein a higher-order ML function was implemented in a rather baroque way by some tricky hand-written self-modifying assembly code. Thanks to the modularity of their logical relations method, this highly non-standard assembly code could nonetheless be safely linked with assembly modules produced by their verified compiler, with the resulting assembly program guaranteed to preserve the semantics of the corresponding linked source modules.

Unfortunately, it is not clear how to scale Hur et al.’s approach from single- to multi-pass compilers because, although logical relations are modular and flexible, they are not typically transitive.

**Multi-Language Semantics** Motivated by the goal of supporting compiler verification for programs that interoperate between different languages, Perconti and Ahmed [16] propose an approach based on multi-language semantics [13]. In particular, they define a “big tent” language that comprises the source, target, and intermediate languages of a compiler, and provides “wrapping” operations for embedding terms of each language within the others. They then use logical relations to prove that every source module is contextually equivalent to a suitably wrapped version of the target module to which it is compiled. In this way, their method synthesizes the benefits of logical relations (modularity and different source and target languages) and contextual equivalence (transitivity).

One downside of their approach is that the intermediate languages (ILs) used in a compiler show up explicitly in the statement of compiler correctness. This leads to a loss of modularity: the semantics of source-level linking is not preserved when linking the results of compilers that have different ILs. Another downside concerns flexibility: the approach seems to be restricted to compilers that use typed intermediate and assembly languages. Lastly, Perconti and Ahmed have so far only applied their technique to a compiler for a purely functional source language.

**Compositional Verification for CompCert** Motivated by the goal of compositional compiler verification, Beringer et al. [4] propose an adaptation of the CompCert framework based on a novel operational semantics that differentiates between internal (intra-module) and external (inter-module) function calls. They introduce a notion of “logical simulation relation” that assumes little about the memory transformations performed by external function calls.

Beringer et al.’s approach is transitive, but lacking somewhat in modularity and flexibility. Concerning modularity, it does not yet (according to the authors) support fully separate compilation. Concerning flexibility, it depends on compiler passes only performing a restricted set of memory transformations—permitting additional transformations can potentially break the transitivity property. In addition, their method appears to be geared specifically toward compilers in the style of CompCert, which employ a uniform memory model across source, intermediate, and target languages. It is not clear how to generalize their technique to support richer (e.g., ML-like) source languages, or compilers whose source and target languages have different memory models.

### 1.2 Our Approach: Parametric Inter-Language Simulations

In this paper, we propose a compositional approach to semantics preservation and compiler verification that combines some of the best features of previous methods. In particular, ours is the first to provide strong support for modularity, transitivity, and flexibility.

Our approach is a descendant of parametric simulations (PS), formerly called “relation transition systems”, a method developed by Hur, Dreyer, Neis, and Vafeiadis (HDNV) [7]. The basic idea of PS is to parameterize the definition of similarity (which says when a target-language expression refines a source-language expression) over an “unknown relation” (which says when an unknown target-language function, passed in from the environment, refines an unknown source-language function).

One of HDNV’s main motivations for developing PS was that they have the potential to scale to support compositional compiler verification. Their argument why PS have this potential is three-pronged: (1) PS provide the same kind of modular relational reasoning principles as state-of-the-art logical-relations methods [5]; (2) unlike such state-of-the-art logical-relations methods, PS are transitivity/composable; and (3) unlike other higher-order simulation methods [18, 19, 17], PS do not employ “syntactic” techniques that only make sense for single-language reasoning. However, as formally developed so far [7, 8, 9], PS have in fact only been used for single-language reasoning, i.e., as a proof technique for contextual equivalence/refinement. Whether they could be adapted to support general compiler verification has remained a major open problem.

In this work, we deliver on the promise of PS by generalizing them to parametric inter-language simulations (PILS). Essentially, we are doing for PS what Hur and Dreyer [6] did for logical relations: namely, taking a proof technique for single-language reasoning and showing how to scale it to inter-language reasoning between high- and low-level languages. The key difference is that, due to its reliance on a logical-relations model, Hur and Dreyer’s method only supports verification of single-pass compilation, whereas PILS support verification of multi-pass compilation. Thus, PILS combine the modularity and flexibility of Hur and Dreyer’s method with the transitivity of PS. Furthermore, PILS are fully mechanized in Coq.

To demonstrate the flexibility of PILS, we have used them to verify Pilsner, a multi-pass optimizing compiler that we have programmed in Coq. Pilsner translates an ML-like Source language with recursive types, abstract types, and general references (S), through a CPS-based Intermediate language (I), down to an idealized assembly-language Target language (T). After CPS conversion, Pilsner performs several simple optimizations at the I level prior to code generation. These optimizations include dead code elimination, hoisting, inlining, and deduplication of redundant bindings.

To show that our method is in no way dependent on the particular transformations employed in the Pilsner compiler, we have also used PILS to verify the challenging self-modifying code example of Hur and Dreyer (mentioned above). The ability to handle such a “wild” example gives strong evidence that PILS are flexible enough to handle a wide range of different program transformations.

The metatheory of PILS—especially the proof of transitivity, which was extremely complex even for PS [8]—is quite involved. To ensure the scalability of PILS as a general method for verifying different compilers, we define PILS in a language-generic fashion, i.e., parameterizing the definition by abstract specifications for the two languages that the PILS are supposed to relate. This language-generic formulation has the major side benefit of enabling us to avoid redundant proof effort by proving much of the PILS metatheory once and for all. Thus, in verifying the Pilsner compiler, we can reuse those metatheoretic results when instantiating the generic PILS model with various pairs of languages (T-S, T-I, I-S).

The overall proof effort has nonetheless been a significant undertaking, involving several person-years of work and around 40,000 lines of Coq. Although fascinating, a full formal presentation of the metatheory of PILS and Pilsner would thus be beyond the scope of this paper. Our aim here instead is to convey a high-level understanding of how PILS work, how they generalize and improve upon the original parametric simulations, and how we use them in verifying Pilsner. We provide as much technical detail as needed to
2. Review of Parametric Simulations

In this section, we briefly review HDNV’s (single-language) parametric simulations (hereafter, PS) [7]. The PS model we present here is based on HDNV’s formulation, but with several modifications (discussed below) that are important for generalizing to PILS.

It is shown only for a fixed, pure language ($\lambda_0^u$) in order to focus attention on the basic structure of a PS model. Despite the various complications introduced by inter-language reasoning later on, the basic structure presented here will remain intact, so we believe this is a useful stepping stone to understanding PILS.

$\lambda_0^u$ is a minimal typed CBV $\lambda$-calculus with numbers, products, sums, recursive types, and numerical I/O (Figure 1). The syntax of types, $\tau \in \text{Type}$, expressions, $e \in \text{Exp}$, and values, $v \in \text{Val}$, is standard. As is evident from the term syntax, we assume that type annotations have already been erased (which is fine since the dynamic semantics of $\lambda_0^u$ ignores them). $k \in \text{CCont}$ are closed evaluation contexts (i.e., continuations), whose definition we omit for brevity. We also assume a small-step labelled transition relation, $e \xrightarrow{\text{ctx}} e'$, stating that $e$ reduces to $e'$ in one execution step. Transitions are labelled with an event, $t$, recording whether the execution step reads a number from the external input ($\gamma(n)$), outputs a number ($\gamma(n)$), or simply performs some internal computation ($\epsilon$).

Figure 1 also defines a notion of compositional correctness, $\Gamma \vdash e : e_\text{ct} : \tau$, saying that under context $\Gamma$, the terms $e$, $e_\text{ct}$, are similar at type $\tau$, in that every execution of $e$ can be simulated by one of $e_\text{ct}$ having the same I/O behavior and similar final values. In the special case where $\Gamma$ is empty, $e : e_\text{ct}$, will imply the global correctness notion $\text{Beh}(e) \subseteq \text{Beh}(e_\text{ct})$, where $\text{Beh}(e)$ denotes the I/O and termination behaviors $e$’s execution can have.

Before discussing the definition of $\leq$, let us first give the basic intuition behind it. The primary difficulty in defining such a relation is in determining when two functions behave similarly. Naively, we would like to say that two functions are similar if one simulates the other whenever applied to similar arguments. But this presupposes that we already know what it means for arguments to be similar; and in a higher-order language, where the arguments could themselves be functions, we are back where we started! In a logical-relations model, this circularity is resolved by defining similarity recursively over the type structure of the language (or over steps of computation if it is a step-indexed logical relation [5]). But as we noted in the introductory, logical relations do not in general lead to a transitive notion of semantics preservation.

Parametric simulations solve this circularity a different way. The key idea behind them—and the reason they are called parametric simulations—is that instead of trying to define what it means for arguments to be similar, they are parameterized over what it means for arguments to be similar. Specifically, the entire simulation is parameterized over an unknown relation $U$, which represents when unknown functions (e.g., functions passed in from the environment as arguments to higher-order functions) are to be considered similar. We call this the unknown relation because it is unknown whether it is a simulation at all—it might very well relate behaviorally dissimilar functions! But that is not “our” problem: we can treat the obligation to prove that $U$ is a simulation as the environment’s responsibility, not ours. Our problem is merely to ensure that the functions related by our parametric simulation behave similarly at all times other than during calls to functions related by $U$.

Given this intuition, we now turn to the formal definitions.

---

1 HDNV called this the “global knowledge”, in contrast to the “local knowledge” $F(U)$. We feel our terminology is more intuitive, and does not conflict with the global/local distinction as it pertains to worlds (§4.1).
The unknown relation, \( U \in \text{VRelF} \), only relates values at function types. We lift it to relate values at arbitrary types using the value closure construction, \( \llbracket \cdot \rrbracket \in \text{VRelF} \to \text{VRel}. \) Here, \( \llbracket R \rrbracket \) removes from \( R \) any related pairs that are not proper function values, and adds the expected equivalent values at the other types. (Note that this is a perfectly valid recursive definition because \( R \) only appears on the r.h.s. in positive positions.)

The auxiliary relation, \( \text{call}(R_f, R_c, R_h) \), relates \( e_a \) and \( e_b \) when they represent the program state just after beta-reduction of related function calls, in which the functions are related by \( R_f \), the arguments by \( R_c \), and the continuations by \( R_h \).

The similarity relations for function values (\( F \)), expressions (\( E \)), and continuations (\( K \)), as well as \( U \) are defined mutually coinductively\(^2\), and parametrically with respect to an arbitrary unknown relation \( U \).

**Function similarity,** \( F(U) \), asserts that two function values \( v_a \) and \( v_b \) are similar if, for any larger unknown relation \( U' \), the functions map arguments that are related by \( \llbracket U' \rrbracket \) to results that are similar according to \( E(U') \). This corresponds directly to the intuition given above. (Note that we require \( v_a \) and \( v_b \) to be “good” functions: this just means they must not be obviously “bad” functions like \( \lambda x.2(3) \); it is needed for technical reasons in the proof of transitivity.)

**Continuation similarity,** \( K(U) \), is defined analogously: \( k_a \) and \( k_b \) are similar if, for any larger \( U' \), they map values related by \( \llbracket U' \rrbracket \) to expressions that are similar according to \( E(U') \).

**Expression similarity,** \( E(U) \), is more interesting. Two expressions \( a \) and \( b \) are similar if, for any larger \( U' \), they map values related by \( \llbracket U' \rrbracket \) to expressions that are similar according to \( E(U') \).

The model takes a small-step approach, saying how for each step of \( e_a \), we should find matching steps of \( e_b \). In contrast, HDNV’s model takes a big-step approach, where multiple steps of each expression are matched. While the big-step approach may result in somewhat simpler proofs for deterministic languages, our small-step approach has some important benefits: it works much better with non-deterministic languages and I/O events, and has the potential to scale to a concurrent language.

Last but not least, the HDNV model uses two nested greatest fixed point definitions: one for the \( E \) and \( K \) and one implicit one saying that there exists a function simulation \( L \) that is consistent w.r.t. the generating function of \( F \) (which they call a “consistent local knowledge”). Our model, instead, defines function similarity (the greatest consistent function simulation) directly, in mutual coinduction with \( E \) and \( K \). We believe this is more intuitive.

As a result of these technical differences, our PS model is a better starting point for scaling to the inter-language setting than the original HDNV model.

### 3. Abstract Language and Concrete Instances

It is now time to abstract away language details from the PS model in the previous section, in order to obtain a PILS model that supports both a richer source language as well as a very different low-level target-language. This model will be defined parametrically in two abstract languages \( A \) and \( B \) and a global world that fixes calling conventions and data representations. In this section, we first describe the basic requirements on the abstract languages. Then we presents parts of the concrete languages under consideration and show how they match these requirements. In §4 we present the actual model as well as the concrete global worlds for our languages.

Common to all languages are the set of events from \( \{ \text{ERR}, \text{VRel}, \text{Val}, \text{Rec}, \text{Call}, \text{Ret} \} \) and a countably infinite set of labels \( F_1, F_2, \ldots \in \text{Lbl} \). In this work, we consider a simplistic notion of module as the compilation unit. Labels are used to identify their components. In contrast to modeling linking as function application, we will have proper static linking.

#### 3.1 Language Specification

Figure 2 presents the abstract language in terms of a signature that any concrete language must implement. Hence, a language must come with a set \( \text{Val} \) of values, a set \( \text{Cont} \) of continuations, a set \( \text{Conf} \) of configurations, a set \( \text{Mach} \) of machines, a set \( \text{Mod} \) of modules, and a set \( \text{Anch} \) of anchors (think: load addresses).

The core of the language semantics is given in the form of a transition system \( \langle \cdot \rangle \) of machines, whose transitions are labelled with events. Configurations can be thought of as partial machines—they play different roles in different contexts (e.g., they might represent just a heap or just an expression, or a full machine). If a configuration \( c \) is complete, it may be realized by a machine \( m \in \text{real}(c) \). (In all our instantiations, this is either empty, meaning the configuration is invalid or incomplete, or it contains exactly one machine.) Configurations must form a partial commutative monoid with composition and neutral element \( \emptyset \), except that the partiality is implicit via \( \text{real} \). (Having \( \cdot \) be total is more convenient for mechanization [5]). We say a machine denotes an error, \( m \in \text{error} \), iff it does not realize any configuration.

Intuitively, modules are sets of labelled function values (the exports), and these values may refer to external functions (the imports) by their unique labels. In terms of the language specification, the module interface consists of two operations load and vload. The

---

\(^2\)This can be made formal by first abstracting \( F, U, K \) over what will become \( E \), showing that they are monotone in that argument, and then defining \( E \) as the greatest fixed point of a monotone generating function.

\(^3\)Due to space limitations, it is impossible to list the complete definitions here. They are, of course, given in our appendix and Coq development.
Domains: Val, Cont, Conf, Mach, Mod, Anch

Operators and relations:
- \( \text{cloud} \in \text{Mod} \to \text{Anch} \to (\text{Lbl} \times \text{Val})^* \to \text{P}(\text{Conf}) \)
- \( \text{vload} \in \text{Mod} \to \text{Anch} \to (\text{Lbl} \times \text{Val})^* \to \text{Lbl} \to \text{P}(\text{Val}) \)
- \( \cdot \in \text{Conf} \to \text{Conf} \) (neutral for \( \cdot \))
- \( \bot \in \text{Conf} \) (computative and associative)
- \( \\emptyset \in \text{Conf} \)
- \( \\in \text{P}(\text{Eval} \times \text{Mach} \times \text{Mach}) \)
- \( \text{real} \in \text{Conf} \to \text{P}(\text{Mach}) \)
- \( \text{error} := \{ m \in \text{Mach} \mid \forall c. m \notin \text{real}(c) \} \)

Figure 2. Language specification

\[
\begin{align*}
\tau & := \ldots \mid \forall \alpha. \tau \mid \exists \alpha. \tau \mid \text{ref } \tau \\
e & := \ldots \mid F \mid e_1 \circ e_2 \mid \text{ifnz } e \text{ then } e_1 \text{ else } e_2 \mid \Lambda. e \mid e[] \mid \text{pack } e \mid \text{unpack } e_1 \text{ as } x \text{ in } e_2 \mid l \mid \text{ref } e \mid l[e] \mid e_1 ::= e_2 \mid e_1 \lll e_2 \\
v & := \ldots \mid \Lambda. e \mid \text{pack } v \mid l \\
\text{Val} & := \{ v \mid \text{FV}(v) = \emptyset \} \\
\text{Mod} \ni M & := \{ F_1 = e_1, \ldots, F_n = e_n \} \\
\text{Anch} & := 1 \\
\text{Cont} \ni K & := \bullet \mid K \in \nu K \mid \ldots \\
\text{Env} & := \text{Lbl} \to \text{Val} \\
\text{Heap} & := \text{Heap} \times \text{Env} \times \text{Exp} \ni \emptyset \\
\emptyset & := (\emptyset, \emptyset, \emptyset) \\
\text{real}(c) & := \{ m \mid m = c \circ m \text{hp} \neq \bot \land m \text{hp finite} \} \\
\text{cloud}(M)(\lambda)(\sigma) & := \{ \exists \sigma'. c = (\emptyset, (\sigma, M, \sigma')), \emptyset \} \\
\text{vload}(M)(\lambda)(\sigma)(F) & := \{ v \mid F(v) \in M \} \\
(h, \sigma, K(F)) & \leadsto (h, \sigma, K[v]) \quad \text{(if } F(v) = v) \\
(h, \sigma, K[\text{input } n]) & \xrightarrow{\ell} (h, \sigma, K[n]) \\
(h, \sigma, K[\text{output } n]) & \xrightarrow{\ell} (h, \sigma, K[()]) \\
(h, \sigma, K[\text{ref } v]) & \xrightarrow{\ell} (h, \{l \mapsto v]\), \sigma, K[[]]) \quad \text{(if } h \cdot \{l \mapsto v\} \neq \bot) \\
(h, \sigma, e) & \xrightarrow{\ell} (\bot, \sigma, e) \quad \text{(if } e \notin \{l \mapsto v\} \text{ and no other rule applicable)}
\end{align*}
\]

Figure 3. Source language S

idea is that cloud takes an anchor saying “where” the module is to be loaded, as well as a value for each of its imports. It then returns a set of initial configurations in which the module is considered “loaded”. Given the same inputs and one of the exported function labels, vload returns the module’s value for that label.

3.2 Source Language S

The source language S is a mostly standard PCF-like language extended with products, sums, universals, existentials, general recursive types, general reference types, as well as numeric I/O. Its type and term syntax is given in Figure 3 as an extension of the one in §2.

Machines consist of a heap \( h \), a read-only environment \( \sigma \) for labels, and an expression \( e \). Heaps are either undefined (equal to \( \perp \)) or partial maps from locations to values. We assume the obvious composition operation \( \cdot \) for heaps (overloading notation) that returns the union of two heaps iff both are defined and they don’t overlap (otherwise it returns \( \perp \)). Note that the empty heap \( \emptyset \) is its neutral element. The step relation between machines is a pretty standard substitution-based left-to-right call-by-value reduction and we state only a few rules. Of note is the treatment of labels: their function values are looked up in \( \sigma \) at runtime. Also, if a machine cannot take a successful step (according to the usual rules such as beta reduction), then it steps to an error state by invalidating its heap component.

Configurations are machines where one or more components may be missing or invalid. Regarding the heap, this is already possible (a missing heap is an empty heap \( \emptyset \)). Regarding environment and expression, we lift their domain by explicitly adding two elements \( \perp \) and \( \emptyset \), with the obvious composition operation for which \( \perp \) is absorbing and \( \emptyset \) is neutral. Using this, the composition of configurations is simply defined pointwise. For a machine \( m \) to realize a configuration \( c \), it must match the configuration \( m = c \) with the obvious embedding of \( \text{Mach} \) in \( \text{Conf} \). Moreover, it must carry a valid and finite heap (finiteness guarantees that allocation will succeed). Hence a heap can successfully be split across several configurations, but environment and expression cannot (they must be defined in exactly one component in order for the composition to be realizable).

We assume a standard typing judgment \( \Gamma \vdash e : \tau \), where \( \Gamma \) assigns types to both labels and variables. It implies that \( \tau \) and the types in \( \Gamma \) are closed and that all labels and free variables in \( e \) are in the domain of \( \Gamma \).

A module \( M \) is simply an ordered list of uniquely labelled function definitions. The above typing judgment is lifted to modules as \( \Gamma \vdash M : \Gamma' \), but requires that both \( \Gamma \) and \( \Gamma' \) only contain labels and, for simplicity, that the module components are all functions \( \text{fix}(f)(x).e \) or \( \Lambda. e \). Moreover it imposes a strict left-to-right dependency order on the module components, to keep the module semantics simple. As there is no need for anchors in the source language, we define them as a singleton set.

Linking two modules \( \langle M, N \rangle \) simply concatenates them (assuming their labels are disjoint). Note that this is an asymmetric operation as \( \langle M, N \rangle \) may resolve imports of the right module, but not of the left. The semantics of a program, i.e., a complete module containing a designated main function \( F_{\text{main}} \) of type unit \( \to \) unit, is the semantics of the machine consisting of an empty heap, the module itself as environment, and the call of the main function as the expression component. The definition of cloud generalizes this machine creation by allowing for additional functions in the environment, which intuitively come from other modules that the module in question might have been linked with. Because of non-determinism, a program may have many executions, each producing a possibly infinite trace of events. If an execution produces an error, then from that point on an arbitrary event trace is produced. We write \( \text{Behav}(M) \) for the set of a program’s execution traces after erasing all finite sequences of error events.

3.3 Intermediate Language I

The intermediate language I is an untyped, or rather, dynamically typed CPS-variant of S. Parts of it is shown in Figure 4. Being in continuation-passing style, every subexpression is explicitly named and functions never “return”. Concretely, we distinguish between atomic expressions \( a \), which are evaluated in let-bindings, and control expressions \( e \). Ignoring conditionals, every expression \( e \) is essentially a sequence of bindings followed by a function or continuation call. For instance, let \( k \ y = e_1 \) in \( e_2 \) defines a new continuation \( k \) with argument \( y \) and body \( e_1 \), and then executes \( e_2 \) (which may use \( k \)). In addition to term variables \( y \in \text{TVar} \), \( S \) features continuation variables \( k \in \text{KV} \) (see also the discussion of code generation in §6). We write \( x \) for either a term variable or a label. In contrast to the source language, \( I \) is defined using an environment-based semantics where continuations and functions evaluate to closures of code and environment. This avoids the need to reason about substitutions when verifying optimizations, which is often a hassle.

Modules, anchors, configurations, etc. are quite similar to those in the source language. Because continuations are already values in the language, we define \( \text{Cont} \) simply as \( \text{Val} \). Since the semantics is closure-based, it does not make sense to allow configurations containing an environment but no expression, or vice versa. The definitions of cloud and vload are omitted for brevity; the notions of linking and program behaviour are pointless here because we never link or execute intermediate programs.
We now define the generic model that is parameterized over "output" language \( A \) and "input" language \( B \) as well as a global world for them (part of this involves defining what exactly a global world is). Instantiating the generic model therefore really means two things:

1. Instantiating the generic model therefore really means two things:

   - Instantiating the generic model therefore really means two things:
   - (1) plugging in two concrete languages that satisfy the specification from §3, and (2) plugging in a concrete global world for these two languages. We will construct exactly one global world for each pair of languages that we wish to have a similarity relation for.
   - To be precise, we actually define two generic models: a typed one and an untyped one. The former is used if the input language is \( S \), and bakes in \( S \)'s type structure; the latter is used in all other cases. However, we will continue to refer to them as just "the generic model", because the untyped one is obtained simply by erasing all the type arguments from the typed one (highlighted in brown in the figures to come).

2. For the sake of presentation, we simplify the generic model by omitting support for universal and existential types as well as several other features (but see §5). Moreover, we leave out a few side conditions in both the definitions and theorems. Full details can be found in the appendix and of course in the Coq development.

### 3.4 Target Language \( T \)

As shown in Figure 5, our target language \( T \) is an idealized assembly language featuring instructions for arithmetic, control flow, memory access, and I/O. Some of them support multiple addressing modes. For instance, if \( o = r_1 \pm n \), then \( st o r \) stores the contents of register \( r_2 \) on the stack at the address contained in register \( r_2 \), offset by \( \pm n \). If \( o = r_1 r \), then it stores it on the heap instead. The \( l p e \) instruction loads the current program counter into the given register.

### 4. Generics Model and Concrete Instances

We now define the generic model that is parameterized over "output" language \( A \) and "input" language \( B \) as well as a global world for them (part of this involves defining what exactly a global world is). Instantiating the generic model therefore really means two things:
The above example translates into
\[ \text{v} \quad \text{CR} \quad \text{Our source and intermediate language comprise} \]

\[ \text{argument} \quad \text{asks for the configurations that represent a call of function} \]

\[ \text{configurations, handled by} \quad \text{this must be a} \]

\[ \text{handlers must be monotone in the state argument w.r.t.} \]

\[ \text{to answer questions such as: in state} \]

\[ \text{much depends on the memory in which it is considered.} \]

\[ \text{several primitive data structures such as products and coproducts.} \]

\[ \text{w} \quad \text{systems (defined in the canonical way). Combining a local world} \]

\[ \text{are like full worlds, with one twist: their configuration relation} \]

\[ \text{can \quad This mechanism takes the form of two query} \]

\[ \text{and \quad e.g.,} \]

\[ \text{fun} \quad \text{in Figure 6). They take} \]

\[ \text{over pairs of register file and} \]

\[ \text{association configuration predicate} \quad \text{vqh} \text{and cqhb. The part concerned with the input language is} \]

\[ \text{vqh} \text{and cqhb of the query handlers vqa and cqha. The part} \]

\[ \text{with \quad this corresponds to the fact that the environment stays} \]

\[ \text{fixed throughout the whole execution, since it is read-only. The} \]

\[ \text{query handlers vqh and cqhb do not depend on the state and} \]

\[ \text{assoc. configuration predicate cqhb} \text{simply requires that the} \]

\[ \text{configuration contains the state’s environments and nothing else.} \]

\[ \text{the configuration query handlers cqhb are straightforward.} \]

\[ \text{T’s component is more interesting. Its state space} \]

\[ \text{over pairs of register file and value database. The latter is consulted by} \]

\[ \text{the query handlers and keeps track of allocated pairs, sums,} \]

\[ \text{and functions. (Unboxed value forms such as numbers or} \]

\[ \text{functions (intuitively: pointers to closures). The contents of such a} \]

\[ \text{cell (the code pointer) is part of the database entry, as can be seen in} \]

\[ \text{vqh, s.db(fun n')} \text{is the set of functions with code pointer n’}. \text{(The global} \]

\[ \text{information to reality. We will see in this section what this looks like} \]

\[ \text{Recall that we need to construct four global worlds WVS, WFT,} \]

\[ \text{W1S—one for each concrete model. To avoid duplicate work,} \]

\[ \text{we will define all four following the schema shown at the top of} \]

\[ \text{Figure 7. Accordingly, each global world consists of three parts: one} \]

\[ \text{solely regarding the output language, one solely regarding the} \]

\[ \text{input language, and one regarding both. The last of these deals with} \]

\[ \text{global references. It consists of a transition system} \]

\[ \text{consists of a transition system Tref'}, \text{a configuration relation} \]

\[ \text{crel}^{ref'} \text{and the implementation of rqh. The details of that} \]

\[ \text{closely follow earlier work and are omitted here. In short, the} \]

\[ \text{state space is a partial bijection on locations and} \]

\[ \text{crel}^{ref'} \text{ensures that these locations are allocated and contain related} \]

\[ \text{values. The part concerned with the output language consists of} \]

\[ \text{a transition system} \quad \text{Tref'}, \text{a configuration predicate} \]

\[ \text{crel}^{ref'} \text{and implementations vqh, cqhb of the query handlers} \]

\[ \text{vqh and cqhb. The part concerned with the input language is} \]

\[ \text{Figure 7 contains some details for the S and T languages. Regarding} \]

\[ \text{the former, the state space} \quad \text{ranges over label environments (from Figure 3),} \]

\[ \text{but once a particular environment has been picked it can never be} \]

\[ \text{changed. This corresponds to the fact that the environment stays} \]

\[ \text{fixed throughout the whole execution, since it is read-only. The} \]

\[ \text{query handlers vqh and cqhb do not depend on the state and} \]

\[ \text{simply return values based on syntactic criteria. For instance, value} \]

\[ \text{v represents a pair of v1 and v2 if it is the pair value} \]

\[ \text{v1, v2). The associated configuration predicate} \text{crel}^{ref'} \text{simply} \]

\[ \text{requires that the configuration contains the state’s environments and} \]

\[ \text{nothing else. The configuration query handlers} \text{cqhb} \text{are straightforward.} \]

\[ \text{T’s component is more interesting. Its state space} \]

\[ \text{over pairs of register file and value database. The latter is consulted by} \]

\[ \text{the query handlers and keeps track of allocated pairs, sums,} \]

\[ \text{and functions. (Unboxed value forms such as numbers or rolls} \]

\[ \text{are simply represented as themselves and do not require an entry} \]

\[ \text{in the database.) The first conjunct in the configuration} \]

\[ \text{predicate} \text{crel}^{ref'} \text{dictates how the allocated values are laid out in} \]

\[ \text{memory. For instance, it will say that each pair in the database is actually a heap} \]

\[ \text{in the database.) The first conjunct in the configuration} \]

\[ \text{predicate} \text{crel}^{ref'} \text{dictates how the allocated values are laid out in} \]

\[ \text{memory. For instance, it will say that each pair in the database is actually a heap} \]

\[ \text{in the database.) The first conjunct in the configuration} \]

\[ \text{predicate} \text{crel}^{ref'} \text{dictates how the allocated values are laid out in} \]

\[ \text{memory. For instance, it will say that each pair in the database is actually a heap} \]

\[ \text{in the database.) The first conjunct in the configuration} \]

\[ \text{predicate} \text{crel}^{ref'} \text{dictates how the allocated values are laid out in} \]

\[ \text{memory. For instance, it will say that each pair in the database is actually a heap} \]
world only “owns” this first cell of a closure; code and possible environment are subject to the local world.

The definition of cqhT formalizes part of T’s calling convention. The other part is ensured by T’s public transition relation: the callee-saved registers sp and env must retain their initial value. Moreover, cpredT also owns the unused part of the stack (second conjunct), which is always one consecutive and cofinite piece—everything above the stack pointer. When reasoning about code that pushes data on the stack, increasing the stack pointer sp effectively transfers the ownership of these previously unused stack cells from the global world to the proof-local world, whose state space must be set up to remember which piece of the stack it currently owns.

4.3 Simulations

We now come to Figure 8. While these definitions may look daunting at first, it is easy to see that they follow the structure of the PS model from Figure 1. The main difference in generalization to the inter-language setting is that the constructions are now abstracted over the details of the particular languages being related, with the help of language specification and worlds. All definitions but the last implicitly take as arguments languages A and B as well as a full world\(^4\) W ∈ World\(_{A,B}\) (intuitively, composed of a global and a local one).

At the top of the figure we have the value closure, this time separating out the function case (more on that later). Both operations take the current state as argument and, in each case, query the world to ensure that the two values in question are of the appropriate “form”.

The “expression similarity” E from \(\S\)2 now relates configurations. In the case of our source language S, this typically continues to be just an expression (i.e., a configuration with empty heap and environment components). In the case of T, it typically is just the program counter. So before we can talk about executions, we need to complete these configurations to full machines \(m_a, m_b\). To do so, we consider any pair \(c_a, c_b\) of configurations currently related by the world as well as arbitrary frame configurations \(\eta_a, \eta_b\), and compose them together (see the use of config). We only care about successful compositions, i.e., if they are not realized by machines, there is nothing to show.

\(^4\) In order to avoid a cycle, it is important that the value closure only depends on the query handlers of the global world, not on its configuration relation.
The step case of $E$ is pretty much the same as before, except that we have to show that the resulting machines are again suitable compositions of configurations. This means that the frame parts $\eta_o, \eta_i$ must not have been modified, and that the machines again contain configurations $c_s', c_s''$ that satisfy the world. As is typical for such world-based models, the latter does not need to hold at what was the current state $s$ (an argument to $E$) but at some future state $s'$ of our own choosing. In the assembly language for instance, if the machine just finished the allocation of a function closure, we would at the very least pick $s'$ to be $s$ extended with an entry for the new function in its value database—otherwise the function wouldn’t even pass as a function in the rest of the proof.

In the RET case of $E$, we generalize “terminating with a value” to “returning a value to the initial continuation”, where the initial continuations $k^0, k^0$ are passed in as additional parameters to $E$. (Note how we assert that $c_s'$ and $c_s''$ “take the form” of $\text{ret } v, k^0$ and $\text{ret } v, k^0$ by appeal to the configuration query handlers of $W$.) Again, we get to choose a future state $s'$ at which the values are related by the unknown relation $U$ (which is now state-indexed as well). But here, since the expressions are returning to their environment, $s'$ must additionally be a public extension of the “initial” state $s$ that was given as a parameter to $E$. (The treatment of public vs. private extension here closely follows the original PS model of HDNV.)

**Module similarity** At the bottom of the figure we finally define what it means for a module $M_t$ from language $B$ to simulate a module $M_s$ from language $A$, relative to a global world $W \in \text{World}_A, B$. We write this as $\Gamma \vdash M_s \leq_U M_t : \Gamma'$, where $\Gamma$ are the imported labels and $\Gamma'$ the exported labels (in the typed model with type annotations). Simply put, this says that if the imports are related, then so are the exports. The details are more complicated because the exports may make sense only in the proper context.

More formally, we require that there exists a local world $w$ and an initial state $s^0$ (of the full world) such that (1) the configurations obtained from loading the modules are related at $s'$, and (2) for any exported label, the module provides value that are related at any future state and unknown relation at which the imports are also related. The example presented below in §4.6 illustrates this.

### 4.4 A Note on the Untyped Model

Since our source programs are typesafe and therefore “don’t go wrong”, neither will correctly produced IL or target programs. One may thus wonder why our model takes faulty programs into account (the Err case in $E$). This feature is actually crucial for verifying transformations in the untyped version of the model. (Recall that we obtain this version by erasing all the type arguments from the definitions in Figures 6–8.)

To see this, first consider the following optimization at the source level (where $x$ is a variable of type nat):

$$\text{fix } f(x). \text{if } n \text{ then } e \text{ else } e \rightarrow \text{fix } f(x). e$$

In the process of showing that $\text{fix } f(x). e$ is similar to the original function, we will be given arguments related at some unknown relation $U$ and state $s$, $(n, v_n, v_n) \in \langle (U(s))^4 \rangle$. Now, by inverting the definition of the value closure, we learn that $v_n = v_n = n$, for some number $n$.

Let us ignore the remaining proof steps and instead consider this transformation at the IL level, where we would be working in the untyped version of the model. There, we will still be given related arguments, $(v_n, v_n) \in \langle (U(s))^4 \rangle$, but this time the type information is missing. Consequently, when inverting the value closure, we don’t end up with the single case above (where $v_n = v_n = n$), but must also consider all the other cases such as $v_n$ and $v_n$ being pairs. Now, note two important points: First, the global world $W_t$ ensures that whenever $v_n$ is a number, then so is $v_n$. In that case we can proceed as we would above in the typed model. Second, if $v_n$ is not a number, then the original program produces an error and, thanks to Err, there is nothing more to show.

This characteristic of the untyped value closure is also the reason why, in Figure 8, we use (→) instead of (→) wherever we expect only functions.

### 4.5 Key Results

Instantiating the generic development with our languages and the global worlds $W_T, W_T, W_T, W_1, W_1$ from §4.2 yields four models. We write $\leq_{TM}$ short for $\leq_{W_{TA}}$ (the simulation between target and source modules), and similarly for the rest. Out of three key results, two are exclusively about $\leq_{TM}$ (and self-explanatory):

**Theorem 1** (Preservation under linking, a.k.a. modularity).

$$
\Gamma \vdash M_1 \leq_{TM} M_2 : \Gamma_1 \quad \Gamma_1 \vdash M_3 \leq_{TM} M_4 : \Gamma_2
\Rightarrow
\Gamma \vdash (M_1 \Rightarrow M_2) \leq_{TM} (M_3 \Rightarrow M_4) : \Gamma_1, \Gamma_2
$$

**Theorem 2** (Adequacy for whole programs).

$$
\vdash M_1 \leq_{TM} M_2 : \Gamma \quad F_{\text{main}} : \text{unit } \rightarrow \text{unit } \in \Gamma
\Rightarrow
\text{Behav}(M_1) \subseteq \text{Behav}(M_2)
$$

The third key result is transitivity, which we split into two properties whose proofs mirror each other thanks to our generic definitions:

**Theorem 3** (Transitivity).

$$
\Gamma \vdash M_1 \leq_{TM} M_2 : \Gamma' \quad \Gamma' \vdash M_3 \leq_{TM} M_4 : \Gamma''
\Rightarrow
\Gamma \vdash M_1 \leq_{TM} M_4 : \Gamma''
$$

(Here $\neg -$ erases the type from the given context, leaving just a list of labels.) Note that from the second property we immediately get:

$$
\Gamma \vdash M_1 \leq_{TM} M_2 : \Gamma' \quad \Gamma' \vdash M_3 \leq_{TM} M_4 : \Gamma''
\Rightarrow
\Gamma \vdash M_1 \leq_{TM} M_4 : \Gamma''
$$

Consequently, it is not necessary to prove that $\leq_{TM}$ itself is a transitive relation—we can just always use $\leq_{TM}$ instead. Combining the last rule with the first one gives us exactly what we need in order to split our compiler proof into one separate proof per pass.

### 4.6 Simulation Example

As an example of using the PILS model, we now sketch the proof of a very simple example simulation. Consider a source language module $M_s := \{ \text{E } = \lambda . F \}$, exporting a single function $E$ of type $\tau := \text{unit } \rightarrow \text{unit }$, and a target language module $M_T$ that also exports a single function $E$ with the following (pseudo-)code:

$$\text{ld clo } F; \text{ jmp } \text{clo} + 0$$

Both functions simply call an unknown external function $F$ and then return to whatever continuation they were started with. We will show $(F: \tau) \Rightarrow M_T \leq_{TM} M_S : (E: \tau)$.

Before we get started, it is important to understand how loading the target module works. Given an anchor (load address) $\Psi$ and a value $v$ for the imported function $F$, a configuration $c \in \text{cloud}_{\text{RT}}(M_T)(\Psi)(\{F \rightarrow v\})$ will contain a heap $h$, the complete stack, a register file with the stack pointer sp set to 0, and no program counter. The heap $h$ will consist of four cells, namely address $v$ (with arbitrary content) and addresses $\Psi \oplus 2$. Address $\Psi$ will be exported as the value of $E$, i.e., $\Psi \in \text{cloud}_{\text{RT}}(M_T)(\Psi)(\{F \rightarrow v\})(E)$, and represents a closure with empty environment. Its code pointer, i.e., the value stored at address $\Psi$, will be $\Psi + 1$, where $E$'s code starts. This code refers to $F$ via the concrete value $v$. Formally, the subheap containing the code is defined as code($\Psi + 1, v$), where:

$$\text{code}(n, v) := \{ n \mapsto \text{ld clo } v, n + 1 \mapsto \text{jmp } \text{clo} + 0 \}$$

(We write “-” for the embedding of instructions into words.)
Let us now start the proof. First we define a local world \( w \) to govern precisely this code segment. We give it state space \( \text{Word} \times \text{Word} \), ranging over all possible combinations of load address for \( \mathcal{M}_F \) and value for \( F \) (both on the assembly side). We do not allow any transitions other than self-transitions because the code of \( E \) is never mutated. The configuration relation is defined as follows (it ignores the state of the global world):

\[
\text{w.crel}(U,(\Psi, v)) := \mathcal{O}((\text{code}(\Psi + 1), v, \emptyset, \emptyset, \emptyset, \emptyset))
\]

Now we are given a concrete anchor (i.e., load address) \( \Psi_T \) for the target module and also values \( v_T \) and \( \text{vrgv} \) of the imported function \( F \), as well as \( \text{cloud} \)ed configurations. On the source side, this means that the function environment \( \sigma \) will map \( F \) to \( v_T \). We define our initial state \( s^0 := (s^0_T, s^0) \) such that the global world owns the imported and the exported function closure, while our local world owns \( E \)'s code:

\[
s^0_T := ((R, \text{db})_T, \sigma, \emptyset) \quad s^0 := ((\Psi_T, v_T))
\]

Here, the target's register file \( R \) and the source's function environment \( \sigma \) are those from the given configurations; the target's value database \( \text{db} := \{ (\text{fun}(n), v_T), (\text{fun}(\Psi + 1), \Psi_T) \} \) says that \( v_T \) and \( \Psi_T \) are closures with code pointers \( n \) and \( \Psi_T \) and \( \Psi_T \) respectively, where \( n \) is whatever happens to be stored in the given heap at address \( v_T \). It is easy to check that the given configurations are indeed related at this initial state.

We now come to the meat of the proof, where we reason about values and can forget about all the tedious details of modules. Given \( U \in \mathcal{U} \) and \( s \supseteq s^0 \), we assume that the imported values are related:

\[
(U, v_T, v_S) \in (U(s))^s
\]

And must show that the exported values are then related as well:

\[
(U, \Psi_T, \lambda, F (\cdot)) \in (U(s))^s
\]

The first part of this is showing that they count as functions in \( s \), which is easy to do by relying on \( s \supseteq s^0 \), because we explicitly registered \( \Psi_T \) as a closure in the database of \( s^0 \).

The harder part is showing that the functions are related by the unknown relation \( U(s) \). Relying on \( \text{F}(U) \subseteq U \), it suffices to show that they are similar, for which we consider their applications in a future state. So we are given \( s' \supseteq s \), \( U' \supseteq U \) and initial continuations \( k_T, k_S \), and must show:

\[
(\text{unit}, v_T, v_S) \in (\text{E}(k_T, k_S))(U')(s')
\]

Here, \( c_T := (0, 0, \emptyset, \Psi_T + 1) \) and \( c_S := (\emptyset, 0, k_S[F (\cdot)]) \) are the call-configurations. Moreover, due to the definition of \( \text{eqh}^F \), we know \( k_T = s'.R(\text{ret}) \), i.e., the target continuation is loaded in the return register.

Unfolding \( E \) and \( \text{configure} \), we are given machines \( m_T, m_S \) that realize compositions \( c_T \cdot c_T \cdot \Psi_T \) and \( c_S \cdot c_S \cdot \text{ret} \), respectively. Moreover, we get to assume that \( (c_T, c_S) \in w^U \), \( \text{crel}(U')(s') \) satisfy the world, and hence we know their shape: \( c_T \) contains a heap, a stack, the register file from \( s' \), and no program counter (since \( c_T \) already contains one); \( c_S \) contains a heap, the environment \( \sigma \) from \( s' \) (given at load time), and no expression (since \( c_S \) already contains one). For the frame configurations \( \Psi_T, \text{ret} \), this leaves but one option: \( \Psi_T \) contains only a heap and a stack; \( \text{ret} \) contains only a heap. All in all, the machines have the following form (for some \( h_T, h_S \)):

\[
m_T = (h_T, s', R, \Psi_T + 1) \quad m_S = (h_S, \sigma, k_S[F (\cdot)])
\]

Next, we proceed according to the step case of \( E \), because we know exactly that \( m_S \) is going to execute the load instruction:

\[
m_T \leadsto (h_T, s', R\text{clo} \circ \text{v_T} + 1, \Psi_T + 2)
\]

We match this by the label-lookup step of the source program (recall that \( \sigma(F) = v_S \)):

\[
m_S \leadsto (h_S, \sigma, k_S[F (\cdot)])
\]

We choose the future state \( s'' \) to be the same as \( s' \), except that the register file has been updated to match the new target machine. It is easy to see that these new machines realize corresponding configurations (in particular the frame components are untouched) and that all constraints imposed by the world are again satisfied at the new state \( s'' \) (no memory has changed except for the register file). Using the \( \text{REC-subcase} \), it now suffices to show

\[
(\text{unit}, c_T^F, c_k^F) \in E(k_T, k_S)(U')(s'')(s'')
\]

where \( c_T^F := (0, 0, \emptyset, \Psi_T + 2) \) and \( c_k^F := (\emptyset, 0, k_S[F (\cdot)]) \). We repeat the whole process with newly configure'd machines \( m_T^F \) and \( m_S^F \). Recall our assumption \( (\tau, v_T, v_S) \in (U(s'))^s' \) about the imports (after using monotonocity). This tells us that they represent valid functions:

1. Closure \( v_T \) is allocated in \( m_T^F \)'s heap: \( m_T^F.h(v_T) = n \).
2. The value \( v_S \) is of the form \( f(x), e \).

Due to (1) we know which step the target takes, and due to (2) the source can match it appropriately:

\[
m_T^F \leadsto (\ldots, n) \quad m_k^F \leadsto (\ldots, k_S[f(\cdot)/x][v_S/f])
\]

(We elide components that haven’t changed.) Keeping the state the same, it is trivial to show that the world is once again satisfied (no memory has changed). Now, notice that according to the global world at the current state \( s'' \), the new configurations \( c_T := (0, 0, 0, n') \) and \( c_k^F := (\emptyset, 0, k_S[f(\cdot)/x][v_S/f]) \) satisfy function call query \( \text{app} v_T(\tau.s''(R(\text{arg})) (s''(R(\text{ret}))) \text{app v_S (k_S) } \text{ret} \), respectively. We can therefore finish the proof via the \( \text{CALL-subcase} \) if we are able to show that the functions, arguments, and continuations are related. The first we know already. The second is easy to check by following the definition of the value closure and using the fact that the global world accepts anything as a unit value on the target side. It remains to show the continuations related:

\[
(\text{unit}, \text{unit}, s''(\text{R(\text{ret})}), k_S) \in K(k_T, k_S)(U'')(s'')
\]

Since \( s'' \cdot R(\text{ret}) = k_T \), this is an instance of a general lemma saying that the initial continuations are related as long the current state publicly extends the initial state. Indeed, \( s'' \vdash \text{sub} \) \( s' \) holds because they only differ in the contents of a caller-save register.

5. Extensions

5.1 Supporting Existential and Universal types

The model from the previous section does not support existential or universal types. The model that we formalized in Coq, however, does support them (as does our Pilnsor compiler). We achieve this by employing the same mechanism that was used in HDNV: values of universal type are essentially treated like functions and existential types are handled by the value closure; lastly, abstract types are modeled as type names, which may be used to attach arbitrary relational interpretations to abstract types.

5.2 Allowing Stuttering

In the definition of \( E \), the source program is forced to take at least as many steps as the target program. Of course, this is harmfully restrictive and one would like to relax it by changing the \( \text{REC} \) case: if the step taken by the target was silent, then the source may \"stutter\", i.e., not take any steps at all. However, we must be careful to forbid infinite stuttering, as that would relate a diverging target program to any source program and thus render the model unsound.

We do so by following the standard approach [14] of indexing \( E \) by an element of a (proof-local) well-founded order, and then demand that this element be decreased in the case of stuttering. However, we require such well-founded orders to provide a little extra structure that lets us employ different product constructions in our metatheory (in particular in the proofs of modularity and
transitivity). We show how this extra structure can always be obtained by considering trees of ordinary well-founded orders.

5.3 Unchaining Internal Computation

Although we are focusing on a sequential setting, the presented model was designed with future application to a concurrent setting in mind. Recall the definition of \( E \): after each step of the target program and matching steps of the source program, we must show that the memory constraints currently imposed by the world are again met. Moreover, in reasoning about the next step of the target program, we are forced to quantify over old and new configurations. In a sequential setting, this is unnecessarily strict. Intuitively, we should not need to show that the world’s conditions are satisfied again until the point where we pass control back to the environment; similarly, we should not need to quantify over new configurations except at points where control is passed to us, because there is no way that the state could have changed in between the internal steps of our computation.

We relax this as desired for our sequential setting through only a few very minor changes to the definitions. The idea is that when we start a computation and are given configurations related by the world, we temporarily acquire them by merging them into our own configurations. As we continue executing, a flag added to configure will remember that we are currently freed from any worldly obligations. However, before we are allowed to use the \( \text{RET} \) or \( \text{CALL} \) case—i.e., when we want to give up control to the environment—we will be forced to release our grip on the world, at which point we must show that all the constraints are again satisfied.

5.4 Enabling Reasoning in Forward Simulation Style

Our model’s formulation lets one conveniently relate a single low-level step to several high-level steps (so-called “backward” simulation). Usually, however, one wants to do exactly the opposite: relate a single high-level step to several low-level steps (so-called “forward” simulation). Intuitively, this is sound assuming that the uniqueness property of bound variables, we immediately follow it with a “freshening” pass that re-establishes uniqueness.

Subsequently, we do a simple dead code (and variable) elimination, which rewrites \( \text{let } x = a \text{ in } e \) to \( e \) if \( x \) does not occur in \( e \). This is justified because, in our IL, evaluation of an atomic expression \( a \) does not have an observable side effect. In the same manner, we also eliminate unused read operations as well as unused allocations (but not write operations as that would be unsound). Following DCE, we hoist let-bindings out of function and continuation definitions, subject to some syntactic constraints. For example:

\[
F \ x \ k' \; \sim \; e[F/f][x/y][k'/k]
\]

If the function is recursive (i.e., \( f \) is used in \( e \)), this is just an unrolling (but note how the first recursive call in the transformed program will conceptually go through the module-level since \( f \) has been rewritten to the label \( F \)). Since inlining destroys the uniqueness property of bound variables, we immediately follow it with a “freshening” pass that re-establishes uniqueness.

Next comes a pass that commutes let-bindings (where possible) in order to group together bindings that assign names to the same expression. For instance:

\[
\text{let } f = (f(x,y,k)), z = x, \text{ in } e \; \sim \; \text{let } z = x, \text{ in } \text{let } f = (f(x,y,k), e) \text{ in } e'
\]

if \( x \) is none of \( f, y, k \). This avoids recomputation of the projection each time \( f \) is called.

Next comes a pass that commutes let-bindings (where possible) in order to group together bindings that assign names to the same expression. For instance:

\[
\text{let } x = a \text{ in } \text{let } y = b \text{ in } z = a \text{ in } e \; \sim \; \text{let } x = a \text{ in } \text{let } z = a \text{ in } \text{let } y = b \text{ in } e
\]

The last transformation, deduplication, gets rid of such consecutive duplicate bindings by rewriting the above expression as follows:

\[
\sim \text{let } x = a \text{ in } \text{let } y = b \text{ in } e[x/z]
\]

This can be seen as a common subexpression elimination.

6. The Pilsner Compiler

Using the machinery from the previous section, we have proven the correctness of Pilsner, a simple yet non-trivial compiler from \( S \) to \( T \) that we have implemented and verified in Coq. Pilsner’s structure is depicted in Figure 9.

First, it translates the source module to the intermediate language via a CPS-transformation. It also takes care to alpha-rename all bound variables such that in the resulting IL module, every variable is bound at most once. This uniqueness condition simplifies the implementation of most of the subsequent optimization passes, as one does not have to worry about accidental variable capturing when rearranging code. Another nice characteristic of the produced intermediate code (not of the IL per se) is that continuations are used in a linear fashion [10], i.e., called at most once. This property is preserved by all other IL transformations and enables a more efficient treatment of continuation variables compared to ordinary variables in the code generation pass.

Code generation, the final pass in the chain, translates from the IL to the machine language. Recall that in the former, there are three kinds of “variables”: term variables \( x \), continuation variables \( k \), and labels \( l \). Labels are translated to absolute addresses according to the import table. Term variable accesses are translated to lookups (based on position) in a linked list on the heap, pointed to by the \( \text{env} \) register. Functions are converted to closures, i.e., pairs of environment and code pointer (module-level functions simply have an empty environment), which live on the heap. A closure’s environment is loaded into the \( \text{env} \) register when the function is called. Finally, continuations are allocated on the stack. Accordingly, continuation variable accesses are translated to lookups (based on position) on the stack, with the side effect that the continuation in question, as well as all more-recently defined ones (above it on the stack), are popped. This is safe because the linearity property mentioned earlier ensures that they won’t be needed anymore.

Let us now come to the transformations happening at the intermediate level. We first perform function inlining. For instance, if a module defines \( F = \text{fix } (y,k), e \), then a call to \( F \) inside a subsequently defined function will be rewritten as follows:

\[
F \ x \ k' \; \sim \; e[F/f][x/y][k'/k]
\]

For the local IL transformations in Pilsner, we developed a simple framework of transformations as expression annotations. The idea is to split a module-level transformation into two parts: (1) an analysis that is applied to each top-level function and annotates selected subexpressions with to-be-performed microtransformations (but does not actually rewrite the code); and (2) the
micro-transformations themselves, together with their correctness proofs. Given these, we automatically produce a verified module transformation that analyzes the input module and performs transformations according to the generated annotations in a bottom-up manner.

A micro-transformation is a partial function on expressions—it must fail if the preconditions for its correctness do not hold (except for a few standard conditions such as well-formedness and uniqueness of bound variables, which it can take for granted). For instance, here is the (only) micro-transformation used in the commute-pass of Pilsner:

\[
\text{commute} \in \text{exp} \rightarrow \text{exp} \\
\text{commute}(e) := \text{let } y = b \text{ in let } x = a \text{ in } e_0 \\
\text{if } \text{is(is(let } x = a \text{ in let } y = b \text{ in } e_0) \text{ and } x \notin \text{FV}(b) \\
\]

In the case that a micro-transformation fails (for which the analysis is to blame), the subexpression that was being transformed simply stays unchanged, or, alternatively, the whole module transformation (and thus the compiler) fails. In either case, if the module transformation succeeds, the output module is guaranteed to correctly implement the input module. This means that the analysis does not need to be verified—in the worst case, the transformation doesn’t optimize the code.

The concrete correctness property demanded by the framework for each micro-transformation \(f\) is twofold. \textit{Syntactic correctness} states the following:

\[
f(e) = e' \quad \Gamma \vdash e \quad \text{unique(BV(e),} \Gamma) \quad w \in \text{WorldL}(W_{11}) \\
\Gamma \vdash e' \triangleright w_{11} e
\]

The relation in the conclusion is the reflexive transitive closure of \(\Gamma \vdash e' \triangleright w_{11} e\), a fairly straightforward lifting of \(E\) from (closed) configurations to open expressions. Its definition considers the expressions under an arbitrary unknown relation and state of \(w'\), with pointwise-related environments providing values for the term variables and labels in \(\Gamma\) as well as continuations for the conclusion variables. \textit{Semantic correctness} requires a proof by coinduction, as one would expect. Together with an easy inductive argument, the compatibility lemmas yield reflexivity for well-formed expressions:

\[
\Gamma \vdash e \implies \Gamma \vdash e' \triangleright w_{11} e
\]

All these lemmas are very helpful in verifying transformations that leave parts of the module unchanged, not just those that make use of our annotation framework.

6.2 Key Result

Theorem 4 (Reinheitsgebot: Compositional correctness of Pilsner).

\[
\Gamma \vdash M_S : \Gamma' \\
\Gamma \vdash \text{Pilsner}(M_S) \triangleright w_{12} M_S : \Gamma''
\]

Using Theorem 2 (if \(\Gamma\) is empty), this gives us the same result as traditional whole-program compiler verification would. However, in combination with Theorem 1, our result becomes much stronger! We can freely link Pilsner-produced code with other Pilsner-produced code while preserving correctness. Even better: since Theorem 1 makes no assumptions about where the target modules come from, we can also link with code that was produced by other compilers or even hand-optimized, and still preserve correctness—as long as those translations have also been verified w.r.t. \(\triangleright w_{12}\).

7. Discussion

Proof of Transitivity Our transitivity proofs closely follow the transitivity proof of the original PS model [8]. However, in our case, since the language “in the middle” is untyped, the complexity having to do with abstract types can be dispensed with. On the other hand, due to our asymmetric small-step formulation of \(E\) and the possibility of stuttering, the part of the proof dealing with \(E\) becomes much more complicated, requiring a distinction of over twenty situations that all ask for somewhat different arguments.

One of the main complexities in the transitivity proof (theirs and ours) lies in dealing with an ambiguity regarding reference allocation: while in one of the two given proofs, an allocation of the middle program may be treated as public (extending the global state), the same allocation may be treated as private (extending the local state) in the other proof. This is a result of transitivity being proven for completely arbitrary local worlds! One might wonder if we could not simplify matters significantly by resorting to an instrumentation of the IL that makes the choice of public vs. private allocation explicit in the program code. It is stated to us that this approach only makes sense if one is willing to a priori decide on all subsequent optimizations. The issue is that a later added optimization might, for instance, figure out that some allocation is never used and therefore can be removed. Such can only be proven correct if the allocation was marked as private, which it may not have been. For the sake of modularity, we therefore believe it is better to deal with the ambiguity issue semantically, e.g., in the way we did.

Related Work In the introduction, we described a variety of existing approaches to compositional compiler verification, and situated PILS in relation to them. We would like to additionally note some important prior work that led to parametric simulations in the first place.

As described by HDNV [7], parametric simulations were inspired by a number of different methods for relational reasoning about higher-order stateful languages: notably, Kripke logical relations [5], normal form (bi-)simulations [18], and environmental (bi-)simulations [19, 17]. From Kripke logical relations, PS adapted the mechanism of possible worlds as state transition systems, enabling the enforcement of protocols on local or global state. From normal form bisimulations, PS took the central idea of viewing unknown functions as black boxes—in particular, the CALL case in Figure 1 is highly reminiscent of a similar case in normal form bisimulations. From environmental bisimulations, PS borrowed the treatment of abstract types and polymorphism via type names.

The key advance of PS was to show how to combine all these mechanisms in a way that supported transitive composition and did not rely on “syntactic” devices employed by the other higher-order simulation methods (e.g., modeling related unknown functions as a common free variable [18], or using context closure operations [19, 17]), because such syntactic devices would preclude a generalization to inter-language reasoning. But it was far from obvious whether PS would necessarily fare any better in this regard.

In this paper, we have demonstrated through PILS that the claims of HDNV were indeed correct, and that PS do in fact generalize to inter-language reasoning as promised.

Future Work Future work includes verifying a certification pass (in progress) as well as providing support for garbage collection and for FFI semantics. Also, our model does not currently support the eta law for function values. This is a known issue of the original PS model, with a known solution [9], which we expect to carry over.

Acknowledgement

The first author is currently funded by a Google European Doctoral Fellowship.

\textsuperscript{3} This separation is convenient, not necessary.

\textsuperscript{5} We actually allow variables in the “source” expression to be renamings of those in the “target” expression, as needed e.g. in the proof of deduplication.
References