# Lecture: RSC

Marco Patrignani

## 1 Languages

We extend the languages from the Backtranslation lecture notes with a (first order) heap. That is, the heap can not contain locations, just other values.

### 1.1 Source

$$
\begin{aligned}
e &::= \cdots \mid \mathsf{let\ x{=}\ new\ e\ in\ e} \mid \mathsf{let\ x{=}\ read\ e\ in\ e} \mid \mathsf{let\ x{=}\ write\ e\ at\ e\ in\ e} \\
v &::= v \mid \ell \\
E &::= \cdots \mid \mathsf{let\ x{=}\ new\ E\ in\ e} \mid \mathsf{let\ x{=}\ read\ E\ in\ e} \mid \mathsf{let\ x{=}\ write\ E\ at\ e\ in\ e} \\
&\quad \mid \mathsf{let\ x{=}\ write\ v\ at\ E\ in\ e} \\
H &::= \varnothing \mid H; \ell \mapsto n \\
\Omega &::= H \triangleright e
\end{aligned}
$$

$H(\ell)$ returns the number $n$ in $H$ to which $\ell$ maps to.

$H \cup \ell \mapsto n$ updates $H$ with a (possibly new) binding of the form $\ell \mapsto n$. If $\ell$ is in the domain of $H$, the binding for $\ell$ is updated to point to $n$.

#### 1.1.1 Static Semantics (Typing)

Programs are now passed locations as input from the context. The related typing rules are trivially changed accordingly.

$\tau ::= \cdots \mid \mathsf{Ref\ Nat}$

$\vdash f(x) \mapsto e : \mathsf{Ref\ Nat} \to \mathsf{Nat}$ Well-typed program $e$ of type $\mathsf{Ref\ Nat}$ to $\mathsf{Nat}$

$$\frac{\Gamma \vdash e : \mathsf{Nat} \quad \Gamma, x : \mathsf{Ref\ Nat} \vdash e' : \tau}{\Gamma \vdash \mathsf{let\ x{=}\ new\ e\ in\ e'} : \tau}\ (\text{T-alloc})$$

$$\frac{\Gamma \vdash e : \mathsf{Ref\ Nat} \quad \Gamma, x : \mathsf{Nat} \vdash e' : \tau}{\Gamma \vdash \mathsf{let\ x{=}\ read\ e\ in\ e'} : \tau}\ (\text{T-read})$$

$$\frac{\Gamma \vdash e : \mathsf{Nat} \quad \Gamma \vdash e' : \mathsf{Ref\ Nat} \quad \Gamma, x : \mathsf{Nat} \vdash e'' : \tau}{\Gamma \vdash \mathsf{let\ x{=}\ write\ e\ at\ e'\ in\ e''} : \tau}\ (\text{T-write})$$

#### 1.1.2 Dynamic Semantics

Judgement: $\Omega \hookrightarrow \Omega'$

$H \triangleright e \hookrightarrow H \triangleright e'$ if $e \hookrightarrow e'$

$$\mathsf{H} \triangleright \mathsf{let\ x = new\ n\ in\ e} \hookrightarrow \mathsf{H} \cup \ell \mapsto \mathsf{n} \triangleright \mathsf{e}[\mathsf{x} \,/\, \ell] \text{ if } \ell \notin \mathsf{H}$$
$$\mathsf{H} \triangleright \mathsf{let\ x = read\ } \ell \mathsf{\ in\ e} \hookrightarrow \mathsf{H} \triangleright \mathsf{e}[\mathsf{H}(\ell) \,/\, \mathsf{x}]$$
$$\mathsf{H} \triangleright \mathsf{let\ x = write\ n\ at\ } \ell \mathsf{\ in\ e''} \hookrightarrow \mathsf{H} \cup \ell \mapsto \mathsf{n} \triangleright \mathsf{e''}[\mathsf{n} \,/\, \mathsf{x}]$$

## 1.2 Target

$$\begin{aligned}
\mathbf{v} &::= \cdots \mid \mathbf{k} \\
\mathbf{e} &::= \cdots \mid \mathbf{let\ x = new\ e\ in\ e} \mid \mathbf{let\ x = read\ e\ with\ e\ in\ e} \\
&\quad \mid \mathbf{let\ x = write\ e\ at\ e\ with\ e\ in\ e} \mid \mathbf{let\ x = hide\ e\ in\ e} \\
\mathbf{O} &::= \mathbf{let\ y = call\ f\ e\ in\ e} \text{ such that } \mathbf{let\ x = hide\ e\ in\ e} \notin \mathbf{e} \\
\mathbf{E} &::= \cdots \mid \mathbf{let\ x = new\ E\ in\ e} \mid \mathbf{let\ x = read\ E\ with\ e\ in\ e} \\
&\quad \mid \mathbf{let\ x = write\ E\ at\ e\ with\ e\ in\ e} \mid \mathbf{let\ x = write\ v\ at\ E\ with\ e\ in\ e} \\
&\quad \mid \mathbf{let\ x = hide\ E\ in\ e} \\
\mathbf{H} &::= \varnothing \mid \mathbf{H; n \mapsto n : \eta} \mid \mathbf{H; k} \\
\eta &::= \bot \mid \mathbf{k} \\
\mathbf{\Omega} &::= \mathbf{H \triangleright e}
\end{aligned}$$

Semantics.

$$\begin{aligned}
&\text{Judgement: } \mathbf{\Omega} \hookrightarrow \mathbf{\Omega'} \\
\mathbf{H \triangleright e} &\hookrightarrow \mathbf{H \triangleright e'} \text{ if } \mathbf{e} \hookrightarrow \mathbf{e'} \\
\mathbf{H \triangleright let\ x = new\ n'\ in\ e} &\hookrightarrow \mathbf{H \cup n + 1 \mapsto n' \triangleright e[x \,/\, n + 1]} \text{ if the cardinality of } \mathbf{H} = n \\
\mathbf{H \triangleright let\ x = read\ n\ with\ k\ in\ e} &\hookrightarrow \mathbf{H \triangleright e[H(n) \,/\, x]} \text{ if } \mathbf{n \mapsto n' : k} \in \mathbf{H} \\
\mathbf{H \triangleright let\ x = read\ n\ with\ v\ in\ e} &\hookrightarrow \mathbf{H \triangleright e[H(n) \,/\, x]} \text{ if } \mathbf{n \mapsto _ : \bot} \in \mathbf{H} \\
\mathbf{H \triangleright let\ x = write\ n'\ at\ n\ with\ k\ in\ e''} &\hookrightarrow \mathbf{H \cup n \mapsto n' \triangleright e''[n \,/\, x]} \text{ if } \mathbf{n \mapsto v : k} \in \mathbf{H} \\
\mathbf{H \triangleright let\ x = write\ n'\ at\ n\ with\ v\ in\ e''} &\hookrightarrow \mathbf{H \cup n \mapsto n' \triangleright e''[n \,/\, x]} \text{ if } \mathbf{n \mapsto _ : \bot} \in \mathbf{H}
\end{aligned}$$

## 1.3 Common

Assume both languages also contain pairs $\langle e, e \rangle$ and projections $e.1$, $e.2$ as standardly done.

Also, we will use a shorthand $e_1; e_2$ for the expression $let\ \_ = e_1\ in\ e_2$.

To ensure state is carried forward in letins, their semantics is changed to:

$$H \triangleright let\ x = e'\ in\ e \hookrightarrow H' \triangleright let\ x = e''\ in\ e \text{ if } H \triangleright e' \hookrightarrow H' \triangleright e''$$
$$H \triangleright let\ x = v\ in\ e \hookrightarrow H \triangleright e[v \,/\, x]$$

**Definition 1.1 (Plugging).** Given that $O = let\ y = call\ f\ e_{arg}\ in\ e_{cont}$ and $P = f(x) \mapsto e_{fun}$.

$$O\,[P] \stackrel{\text{def}}{=} \varnothing \triangleright let\ y = let\ x = e_{arg}\ in\ e_{fun}\ in\ e_{cont}$$

### 1.3.1 Behaviours

We define a behaviour as a sequence of call/returns from the context to the program. The behaviours of a program is the set of all behaviours it can generate.

Note that while this is analogous to traces, this is for whole programs. Also, while traces capture a sort of context/program interaction, behaviours capture a whole program/environment interaction, e.g., they would capture I/O if our language had any. See later for this addition.

$$b ::= call\ n\ H? \cdot ret\ n\ H!$$

While we write this in black, there are really two behaviours, as the heaps are different between S and T. This will be relevant later.

$$O[P] \rightsquigarrow call\ n\ H? \cdot ret\ n'\ H'! \text{ if } \begin{array}{r} O[=]\ let\ y = call\ f\ e\ in\ ec \\ P = f(x) \mapsto eb \\ \varnothing \triangleright e \hookrightarrow^* H \triangleright n \\ H \triangleright let\ x = n\ in\ eb \hookrightarrow^* H' \triangleright n' \\ H' \triangleright let\ y = n'\ in\ ec \hookrightarrow^* H'' \triangleright n'' \end{array}$$

$$\texttt{Behav}\,(O[P]) \stackrel{\text{def}}{=} \{b \mid O[P] \rightsquigarrow b\}$$

# 2 Compiler

The compiler translates source locations $\ell$ into pairs $\mathbf{n}, \mathbf{k}$ of the natural number $\mathbf{n}$ which is the target location and a capability $\mathbf{k}$ used to hide location $\mathbf{n}$.

Thus, since programs expect locations, it expects a dereferenceable pair as input. Also, note that in the case of programs, the compiler remaps $\mathsf{x}$ as $\mathbf{z}$ in order to perform the typecheck on inputs.

$$\begin{aligned}
&\ldots \\
[\![\mathsf{f(x)} \mapsto \mathsf{e}]\!]^{\mathsf{S}}_{\mathbf{T}} &= \mathbf{f(z) \mapsto let\ x = read\ z.1\ with\ z.2\ in} \\
&\qquad \mathbf{if\ x\ has\ NAT\ then\ } [\![\mathsf{e}]\!]^{\mathsf{S}}_{\mathbf{T}} \mathbf{\ else\ fail} \\
[\![\mathsf{let\ x = new\ e\ in\ e'}]\!]^{\mathsf{S}}_{\mathbf{T}} &= \mathbf{let\ x_l = new\ } [\![\mathsf{e}]\!]^{\mathsf{S}}_{\mathbf{T}} \mathbf{\ in} \\
&\qquad \mathbf{let\ x_c = hide\ x_l\ in} \\
&\qquad \mathbf{let\ } [\![\mathsf{x}]\!]^{\mathsf{S}}_{\mathbf{T}} = \langle \mathbf{x_l}, \mathbf{x_c} \rangle \mathbf{\ in} \\
&\qquad [\![\mathsf{e'}]\!]^{\mathsf{S}}_{\mathbf{T}} \\
[\![\mathsf{let\ x = read\ e\ in\ e'}]\!]^{\mathsf{S}}_{\mathbf{T}} &= \mathbf{let\ x_p = } [\![\mathsf{e}]\!]^{\mathsf{S}}_{\mathbf{T}} \mathbf{\ in} \\
&\qquad \mathbf{let\ } [\![\mathsf{x}]\!]^{\mathsf{S}}_{\mathbf{T}} = \mathbf{read\ x_p.1\ with\ x_p.2\ in} \\
&\qquad [\![\mathsf{e'}]\!]^{\mathsf{S}}_{\mathbf{T}}
\end{aligned}$$

$$[\![\mathsf{let\ x{=}\ write\ e\ at\ e'\ in\ e''}]\!]^{\mathsf{S}}_{\mathbf{T}} = \mathbf{let\ x_p{=}}[\![\mathsf{e'}]\!]^{\mathsf{S}}_{\mathbf{T}}\ \mathbf{in}\ \mathbf{let}\ [\![\mathsf{x}]\!]^{\mathsf{S}}_{\mathbf{T}}\mathbf{= write}\ [\![\mathsf{e}]\!]^{\mathsf{S}}_{\mathbf{T}}\ \mathbf{at\ x_p.1\ with\ x_p.2\ in}\ [\![\mathsf{e'}]\!]^{\mathsf{S}}_{\mathbf{T}}$$

## 2.1 Compiler Properties

At this point, since we have added cases to the compiler, we need to add the missing cases to the compiler correctness proof and to the auxiliary lemmas. These additions are straightforward.

# 3 Backtranslation

In this case we have a single trace to backtranslate into a single source context.

$$\langle\!\langle \mathbf{call\ n\ H?} \cdot \mathbf{ret\ n'\ H'!} \rangle\!\rangle^{\mathbf{T}}_{\mathsf{S}} = \mathsf{let\ y{=}call\ f\ e\ in\ e'} \quad \text{where } \mathsf{e} = \mathsf{let\ z{=}}\langle\!\langle \mathit{allocate}\ \mathbf{H} \rangle\!\rangle^{\mathbf{T}}_{\mathsf{S}}\ \mathsf{in}\ \langle\!\langle \mathbf{n} \rangle\!\rangle^{\mathbf{T}}_{\mathsf{S}}$$

$$\mathsf{e'} = \langle\!\langle \mathit{update}\ \mathbf{H}\ \mathit{from}\ \mathbf{H'} \rangle\!\rangle^{\mathbf{T}}_{\mathsf{S}}$$

$$\langle\!\langle \mathit{allocate}\ \mathbf{H} \rangle\!\rangle^{\mathbf{T}}_{\mathsf{S}} = \mathsf{false} \quad \text{if } \mathbf{H} = \varnothing$$

$$= \mathsf{let\ x_n{=}\ new}\ \langle\!\langle \mathbf{v} \rangle\!\rangle^{\mathbf{T}}_{\mathsf{S}}\ \mathsf{in}\ \langle\!\langle \mathit{allocate}\ \mathbf{H'} \rangle\!\rangle^{\mathbf{T}}_{\mathsf{S}} \quad \text{if } \mathbf{H} = \mathbf{H'}; \mathbf{n} \mapsto \mathbf{v} : \eta$$

$$\langle\!\langle \mathit{update}\ \mathbf{H}\ \mathit{from}\ \mathbf{H'} \rangle\!\rangle^{\mathbf{T}}_{\mathsf{S}} = \mathsf{true}$$

Note that backtranslating the return is not necessary here, the compiled code will do that. However, we keep that structure as it will be useful later.

## 3.1 Properties of the Backtranslation

In order to use the context backtranslation, we need to prove that it is correct:

**Theorem 3.1 (Correctness of the backtranslation of behaviours).**

$$\begin{array}{l} \mathit{if}\ \mathbf{b} \in \mathtt{Behav}\left(\mathbf{O}\left[\!\left[\mathsf{P}\right]\!\right]^{\mathsf{S}}_{\mathbf{T}}\right) \\ \quad \mathsf{b} \approx \mathbf{b} \\ \quad \mathsf{P} = \mathsf{f(x)} \mapsto \mathsf{e} \\ \mathit{then}\ \mathsf{b} \in \mathtt{Behav}\left(\langle\!\langle \mathbf{b} \rangle\!\rangle^{\mathbf{T}}_{\mathsf{S}}\ [\mathsf{P}]\right) \end{array}$$

In order to state this, we need a partial bijection $\beta$ to relate source locations and target numbers that are locations. Moreover, we need to tell when two heaps are related ($\mathsf{H} \approx \mathbf{H}$), that is when two $\beta$-related locations point to related values. Then, we need to tell when two values are related ($\mathsf{v} \approx \mathbf{v}$), but this is

simple as this is the same relation for compiler correctness: a value is related to its compilation. Finally, we can define when behaviours are related ($\mathsf{b} \approx \mathbf{b}$), that is when their values and heaps are related.

With this boilerplate, we can understand Theorem 3.1 and define the necessary auxiliary lemmas. One lemma tells that just before performing the "call", heaps are related. Compiler correctness will then tell us that given related heaps and related arguments, a program and its compilation produce related outputs with related heaps. These two together will tell that the behaviours are related since the first lemma tells that the parameters of the call are related and the second tells that the parameters of the return are related.

# 4 Proving RSC

The RSC statement already is in backtranslation form and follows directly from Theorem 3.1 (Correctness of the backtranslation of behaviours).

**Theorem 4.1.** $\forall \mathsf{P}, \mathbf{O}, \mathbf{b}, \exists \mathsf{O}, \mathsf{b} \approx \mathbf{b}. \mathbf{O}\left[\![\mathsf{P}]\!]^{\mathsf{S}}_{\mathbf{T}}\right] \rightsquigarrow \mathbf{b} \Rightarrow \mathsf{O}[\mathsf{P}] \rightsquigarrow \mathsf{b}$

# 5 Calling Programs Multiple Times

We now implement programs that can be called multiple times. For the backtranslation, traces become a finite sequence of calls to the program and returns from said calls. Thus, in this case we need to update locations that the program may have had access to.

$$O \stackrel{\text{def}}{=} let\ x = e\ in\ Y$$
$$Y \stackrel{\text{def}}{=} let\ x = call\ f\ e\ in\ e; Y \mid e$$

Formally, there is no recursion and therefore the whole program will necessarily terminate.

**Definition 5.1 (Plugging).**

$$O\,[P] \stackrel{\text{def}}{=} let\ x = e\ in\ let\ x_1 = let\ x = e\ in\ e_1\ in\ \cdots let\ x_k = let\ x = e\ in\ e_k\ in\ e'$$
$$O = let\ x = e\ in\ let\ x_1 = call\ f\ e_1\ in\ \cdots let\ x_k = call\ f\ e_k\ in\ e'$$
$$P = f(x) \mapsto e$$

Behaviours now ($a$) concatenations of actions $b$ (the behaviours from before), which are sequences of call-returns.

$$a ::= \varnothing \mid b \cdot a$$

At the top level, the behaviour of a whole program is the concatenation of its sequence of call/returns.

$$O[P] \rightsquigarrow b \cdot a \text{ if } \begin{array}{l} O = \mathit{let}\ x = e\ \mathit{in}\ \mathit{let}\ x_1 = \mathit{call}\ f\ e_1\ \mathit{in}\ Y \\ P = f(x) \mapsto e' \\ \quad \varnothing \triangleright e \hookrightarrow^* H_p \triangleright n \\ \quad H_p \triangleright \mathit{let}\ x_1 = e_1[n\ /\ x]\ \mathit{in}\ e' \rightsquigarrow b, H' \\ \quad H' \triangleright Y, P \rightsquigarrow a, H'' \end{array}$$

Given a sequence of call/returns, we can calculate its behaviour by decomposing its sequence of individual calls. In this case, we rely on the semantics, so we need to carry around the heap to ensure the call is correct. At the end of the sequence of call/returns, we return the empty behaviour.

$$H_p \triangleright \mathit{let}\ x_1 = \mathit{call}\ f\ e_1\ \mathit{in}\ Y, P \rightsquigarrow a, H'' \text{ if } \begin{array}{l} P = f(x) \mapsto e' \\ \quad H_p \triangleright \mathit{let}\ x_1 = e_1[n\ /\ x]\ \mathit{in}\ e' \rightsquigarrow b, H' \\ \quad H' \triangleright Y, P \rightsquigarrow a, H'' \end{array}$$

$$H_p \triangleright e, P \rightsquigarrow \varnothing, H'' \text{ if } H_p \triangleright e \hookrightarrow^* H'' \triangleright n$$

Given a single call to the program, starting from a heap $H_p$ it generates a call/return sequence and a new heap $H'$.

$$H_p \triangleright \mathit{let}\ x = e'\ \mathit{in}\ e \rightsquigarrow \mathit{call}\ n\ H?\cdot \mathit{ret}\ n'\ H'!, H' \text{ if } \begin{array}{l} H_p \triangleright e' \hookrightarrow^* H \triangleright n \\ \quad H \triangleright \mathit{let}\ x = n\ \mathit{in}\ e' \hookrightarrow^* H' \triangleright n' \end{array}$$

The set of behaviours of a whole program is calculated as expected.

$$\texttt{Behav}\,(O[P]) \stackrel{\text{def}}{=} \{a \mid O[P] \rightsquigarrow a\}$$

## 5.1 Backtranslation of Behaviours

To define the backtranslation of behaviours we can reuse the idea of the previous backtranslation. We, mainly, need to carry forward the heap between subsequent calls.

Take a close look at the *types* of the different backtranslation parts to understand what part relies on what subpart. We overload the backtranslation symbol for all subparts.

$$\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{T}}_{\mathsf{S}} : \mathbf{a} \to \mathsf{O}$$

$$\langle\!\langle \varnothing \rangle\!\rangle^{\mathbf{T}}_{\mathsf{S}} \stackrel{\text{def}}{=} \mathsf{let\ x{=}true\ in\ x}$$

$$\langle\!\langle \mathbf{b} \cdot \mathbf{a} \rangle\!\rangle^{\mathbf{T}}_{\mathsf{S}} \stackrel{\text{def}}{=} \mathsf{let\ x{=}true\ in}\ \langle\!\langle \varnothing, \mathbf{b}, \mathbf{H_f}, \mathsf{Y} \rangle\!\rangle^{\mathbf{T}}_{\mathsf{S}} \quad \text{where}$$

$$\mathsf{Y} = \langle\!\langle \mathbf{H_f}, \mathbf{a}, \mathbf{H'}, \mathsf{true} \rangle\!\rangle^{\mathbf{T}}_{\mathsf{S}}$$

The backtranslation of a single action is analogous to before. The main change is that we have a previous heap to consider. That is used before the call

to understand what parts of $\mathbf{H}$ are new and must be allocated and what parts are old and must be updated.

$$\langle\!\langle \cdot \rangle\!\rangle_{\mathsf{S}}^{\mathbf{T}} : \mathbf{H} \times \mathbf{b} \times \mathbf{H} \times \mathsf{Y} \to \mathsf{Y}$$

$$\langle\!\langle \mathbf{H_p}, \mathbf{call\ n\ H?} \cdot \mathbf{ret\ n'\ H'!}, \mathbf{H'}, \mathsf{Y} \rangle\!\rangle_{\mathsf{S}}^{\mathbf{T}} \stackrel{\text{def}}{=} \text{simple adaptation from Section 3}$$

The recursive definition decomposes a behaviour into small subcalls to individual call/returns.

$$\langle\!\langle \cdot \rangle\!\rangle_{\mathsf{S}}^{\mathbf{T}} : \mathbf{H} \times \mathbf{a} \times \mathbf{H} \times \mathsf{Y} \to \mathsf{Y}$$

$$\langle\!\langle \mathbf{H_p}, \mathbf{b} \cdot \mathbf{a}, \mathbf{H_f}, \mathsf{Y} \rangle\!\rangle_{\mathsf{S}}^{\mathbf{T}} \stackrel{\text{def}}{=} \langle\!\langle \mathbf{H_p}, \mathbf{b}, \mathbf{H'}, \mathsf{Y'} \rangle\!\rangle_{\mathsf{S}}^{\mathbf{T}} \qquad \text{where } \mathsf{Y'} = \langle\!\langle \mathbf{H'}, \mathbf{a}, \mathbf{H_f}, \mathsf{Y} \rangle\!\rangle_{\mathsf{S}}^{\mathbf{T}}$$

$$\langle\!\langle \mathbf{H_p}, \varnothing, \mathbf{H_f}, \mathsf{Y} \rangle\!\rangle_{\mathsf{S}}^{\mathbf{T}} \stackrel{\text{def}}{=} \mathsf{Y}$$

### 5.2 Correctness of the Backtranslation and Proving RSC

Again, RSC follows directly from backtranslation correctness. The same boilerplate concerning related values, heaps and behaviours is needed here as well.

## 6 Second-order Heap

If we allow the source and the target to have higher-order heaps, the backtranslation complicates even more. Specifically, program and context now can share locations. Thus, when a program has returned, it can update a location that the context passed to it, or it can pass a new location to the context.

In the backtranslation, this affects how the returns are translated. There we now need to keep track of what new locations are made available to the context from the program. In fact, later on the context can update that location and pass it to the program, possibly altering its behaviour this way.

### 6.1 (semi)-Formal Details

For simplicity, we can study this in a second order heap, where locations can store naturals or locations that point to naturals.

#### 6.1.1 Source Changes

To lift the first-order imposition on the heap, we simply change the source types:

$$\tau ::= \cdots \mid \mathsf{Ref\ Ref\ Nat}$$

Otherwise, the source does not change.

#### 6.1.2 Target Changes

The target does not change, we keep the simplification that the context cannot protect its locations with capabilities for simplicity.

### 6.1.3 Compiler Changes

The compiler does noes not change. Crucially, when we share a location with the context, we know we will share it as a pair $\langle \mathbf{n}, \mathbf{k} \rangle$ including both the location $\mathbf{n}$ and the capability $\mathbf{k}$ used to protect it.

### 6.1.4 Backtranslation Changes

The backtranslation needs to change as follows. The general structure is the same: we will generate code that sets up the heap correctly and then performs the related call with the related argument. In this case, however, the program can communicate to the context locations, which are now accessible to the context. The context may change the value of these locations, so we need to have a way to access all locations that are accessible to the context in order to modify them.

For this, the source context needs to keep a list where it stores all locations it knows of. However, this list is in the source, so it is just a list of abstract identifiers, and we need to relate them to target locations. Luckily, target locations are natural numbers, so we can store a map in the source list, from source natural numbers to source locations. For convenience, this map is a pair of type $\mathsf{Nat} \times \mathsf{Ref}\ \tau$. So when we are registering a target location $\mathbf{n}$ we will see a source location $\ell$ and we store the pair $\langle \mathsf{n}, \ell \rangle$. This way we can perform lookup of a certain target location $\mathbf{m}$ looking for element $\langle \mathsf{m}, \ell \rangle$ in the list.

At the top level, before we start the first call, we need to set up the list of known locations.

Additionally, at the top level we keep track of the index of the action we are translating to ensure we generate non-conflicting variable names. This means passing around variable $n$ and annotating all variable names with $n$ in the following subcalls. We avoid doing that to avoid polluting the code too much, be aware that this is ensures that we can then assume that all identifiers are distinct in the backtranslated context.

$$\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{T}}_{\mathsf{S}} : \mathbf{a} \to \mathsf{O}$$

$$\langle\!\langle \varnothing \rangle\!\rangle^{\mathbf{T}}_{\mathsf{S}} \stackrel{\text{def}}{=} \mathsf{let\ y{=}true\ in\ y}$$

$$\langle\!\langle \mathbf{b} \cdot \mathbf{a} \rangle\!\rangle^{\mathbf{T}}_{\mathsf{S}} \stackrel{\text{def}}{=} \mathsf{let\ y{=}e_{start}\ in\ Y}$$

where

$$\mathsf{Y} = \langle\!\langle \varnothing, \mathbf{b}, \mathbf{H_f}, \mathsf{Y'} \rangle\!\rangle^{\mathbf{T}}_{\mathsf{S}}$$

$$\mathsf{Y'} = \langle\!\langle \mathbf{H_f}, \mathbf{a}, \mathbf{H'}, \mathsf{true}, n \rangle\!\rangle^{\mathbf{T}}_{\mathsf{S}}$$

$$n = \mathtt{length}\,(\mathbf{b} \cdot \mathbf{a})$$

$$\mathsf{e_{start}} = \mathsf{let\ x{=}}\langle 0, 0 \rangle\ \mathsf{\ in\ x}$$

We know that the source context will evaluate $\mathsf{e_{start}}$, so y will contain the pointer where we store the list.

The subcalls are primarily unchanged.

$$\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{T}}_{\mathsf{S}} : \mathbf{H} \times \mathbf{a} \times \mathbf{H} \times \mathsf{Y} \times \mathit{Nat} \to \mathsf{Y}$$

$$\langle\!\langle \mathbf{H_p}, \mathbf{b} \cdot \mathbf{a}, \mathbf{H_f}, \mathsf{Y}, n \rangle\!\rangle^{\mathbf{T}}_{\mathsf{S}} \stackrel{\text{def}}{=} \langle\!\langle \mathbf{H_p}, \mathbf{b}, \mathbf{H}', \mathsf{Y}', n\text{-}1 \rangle\!\rangle^{\mathbf{T}}_{\mathsf{S}} \qquad \text{where } \mathsf{Y}' = \langle\!\langle \mathbf{H}', \mathbf{a}, \mathbf{H_f}, \mathsf{Y}, n \rangle\!\rangle^{\mathbf{T}}_{\mathsf{S}}$$

$$\langle\!\langle \mathbf{H_p}, \varnothing, \mathbf{H_f}, \mathsf{Y}, n \rangle\!\rangle^{\mathbf{T}}_{\mathsf{S}} \stackrel{\text{def}}{=} \mathsf{Y}$$

At each call we need to extend the list the context keeps track of with the new locations it creates. At each return, we need to extend that list to store the new locations created by the program and shared with us. Finally, at each call we need to modify the contents of existing locations according to what the target level trace tells, and we need to lookup those locations from the list.

For all of this, recall that we have the full target level trace. So we have the trace that the compiled program will produce and we can inspect that to know both what known locations it changed and what new locations it revealed to the context. This way we can, at runtime in the source, starting from known pointers, traverse the heap to access new pointers and then register those new pointers in the list.

Then, in order to update a known location, we need to lookup the location based on the target identifier and update its value. Notice that we are not feeding complete heaps to the subroutines but either the new heap ($\setminus$) or the existing one ($\cap$).

$$\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{T}}_{\mathsf{S}} : \mathbf{H} \times \mathbf{b} \times \mathbf{H} \times \mathsf{Y} \times \mathit{Nat} \to \mathsf{Y}$$

$$\left\langle\!\left\langle \begin{array}{l} \mathbf{H_p}, \\ \mathbf{call\ n\ H?} \cdot \mathbf{ret\ n'\ H'!}, \\ \mathbf{H}', \mathsf{Y}, m \end{array} \right\rangle\!\right\rangle^{\mathbf{T}}_{\mathsf{S}} \stackrel{\text{def}}{=} \begin{array}{l} \mathsf{let\ y_m{=}call\ f\ e\ in} \\ \quad \mathsf{e'; e''; Y} \end{array} \qquad \text{where}$$

$$\mathsf{e} = \mathsf{let\ \_{=}}\langle\!\langle \mathit{allocate}\ \mathbf{H} \setminus \mathbf{H_p} \rangle\!\rangle^{\mathbf{T}}_{\mathsf{S}}\ \mathsf{in}\ \langle\!\langle \mathbf{n} \rangle\!\rangle^{\mathbf{T}}_{\mathsf{S}}$$

$$\mathsf{e'} = \langle\!\langle \mathit{update}\ \mathbf{H} \cap \mathbf{H_p}, z \rangle\!\rangle^{\mathbf{T}}_{\mathsf{S}}$$

$$\mathsf{e''} = \langle\!\langle \mathit{register}\ \mathbf{H}' \setminus \mathbf{H}\ \mathit{in}\ \mathbf{H}', z \rangle\!\rangle^{\mathbf{T}}_{\mathsf{S}}$$

$$z = \|\mathtt{dom}\,(\mathbf{H}')\|$$

$$\begin{array}{lll} \langle\!\langle \mathit{allocate}\ \mathbf{H} \rangle\!\rangle^{\mathbf{T}}_{\mathsf{S}} = \mathsf{false} & & \text{if } \mathbf{H} = \varnothing \\ \quad = \mathsf{let\ x_n{=}\ new}\ \langle\!\langle \mathbf{v} \rangle\!\rangle^{\mathbf{T}}_{\mathsf{S}}\ \mathsf{in} & & \text{if } \mathbf{H} = \mathbf{H}'; \mathbf{n} \mapsto \mathbf{v} : \eta \\ \quad\quad \mathsf{let\ x_e{=}\langle n, x_n \rangle\ in} & & \\ \quad\quad\quad \mathsf{let\ x_c{=}\ read\ y\ in} & & \\ \quad\quad\quad\quad \mathsf{let\ \_{=}\ write\ \langle x_e, x_c \rangle\ at\ y\ in} & & \\ \quad\quad\quad\quad\quad \langle\!\langle \mathit{allocate}\ \mathbf{H}' \rangle\!\rangle^{\mathbf{T}}_{\mathsf{S}} & & \end{array}$$

The inductive case of $\langle\!\langle \mathit{allocate}\ \mathbf{H} \rangle\!\rangle^{\mathbf{T}}_{\mathsf{S}}$ allocates a new location $\mathsf{x_n}$, that corresponds to target location $\mathbf{n}$. Then it creates an entry for the list of context known locations of the form $\langle \mathsf{n}, \mathsf{x_n} \rangle$. Then it prepends that entry to the list, knowning that the list starts from $\mathsf{y}$ (so it reads the contents of $\mathsf{y}$ and then writes the new list in $\mathsf{y}$).

$$\langle\!\langle update\ \mathbf{H}, z \rangle\!\rangle^{\mathbf{T}}_{\mathsf{S}} = \mathsf{false} \qquad \text{if } \mathbf{H} = \varnothing$$

$$\langle\!\langle update\ \mathbf{H}, z \rangle\!\rangle^{\mathbf{T}}_{\mathsf{S}} = \mathsf{let}\ \mathsf{x_l}{=}\mathsf{LOOKUP}(\mathsf{n},\mathsf{y},\mathsf{z})\ \mathsf{in} \qquad \text{if}$$
$$\mathsf{let}\ \_ = \mathsf{write}\ \langle\!\langle \mathbf{v} \rangle\!\rangle^{\mathbf{T}}_{\mathsf{S}}\ \mathsf{at}\ \mathsf{x_l}\ \mathsf{in}\ \langle\!\langle update\ \mathbf{H}' \rangle\!\rangle^{\mathbf{T}}_{\mathsf{S}} \qquad \mathbf{H} = \mathbf{H}'; \mathbf{n} \mapsto \mathbf{v} : \eta$$
$$\mathsf{z} = z$$

We write $\mathsf{LOOKUP}(\mathsf{n},\mathsf{y},\mathsf{z})$ for the expression that traverses the list starting from $\mathsf{y}$ and returns the pair whose first element is $\mathsf{n}$. Even if we do not have recursion, we know that the size of the list at this time is at most $\mathsf{z}$, so we can simply lookup $\mathsf{z}$ elements when doing this. Since the language is very simple, this is a bit convoluted to write, so we leave this abstract.

$$\langle\!\langle register\ \mathbf{H}\ in\ \mathbf{H}', z \rangle\!\rangle^{\mathbf{T}}_{\mathsf{S}} = \mathsf{false} \qquad \text{if } \mathbf{H} = \varnothing$$

$$\langle\!\langle register\ \mathbf{H}\ in\ \mathbf{H}', z \rangle\!\rangle^{\mathbf{T}}_{\mathsf{S}} = \mathsf{let}\ \mathsf{x_m}{=}\mathsf{LOOKUP}(\mathsf{m},\mathsf{y},\mathsf{z})\ \mathsf{in} \qquad \text{if}$$
$$\mathsf{let}\ \mathsf{x_r} = \mathsf{read}\ \mathsf{x_m}\ \mathsf{in} \qquad \mathbf{H} = \mathbf{H}'; \mathbf{n} \mapsto \mathbf{v} : \eta$$
$$\mathsf{let}\ \mathsf{x_e} = \left\langle \langle\!\langle \mathbf{n} \rangle\!\rangle^{\mathbf{T}}_{\mathsf{S}}, \mathsf{x_r} \right\rangle\ \mathsf{in} \qquad \mathbf{m} \mapsto \langle \mathbf{n}, \eta \rangle \in \mathbf{H}'$$
$$\mathsf{let}\ \mathsf{x_c} = \mathsf{read}\ \mathsf{y}\ \mathsf{in} \qquad \mathsf{z} = z$$
$$\mathsf{let}\ \_ = \mathsf{write}\ \langle \mathsf{x_e}, \mathsf{x_c} \rangle\ \mathsf{at}\ \mathsf{y}\ \mathsf{in}$$
$$\langle\!\langle register\ \mathbf{H}' \rangle\!\rangle^{\mathbf{T}}_{\mathsf{S}}$$

Here we use the information from the whole heap $\mathbf{H}'$ to see how the new location $\mathbf{n} \mapsto \mathbf{v} : \eta$ is reachable from an old location. Because the heap is second order, we know that there can only be a location $\mathbf{m}$ that contains $\langle \mathbf{n}, \eta \rangle$ (since it's generated by the compiled program).

For simplicity we assume that $\mathbf{m}$ is a known location. In a more concrete case, $\mathbf{m}$ itself may be not accessible, but it may be revealed via location $\mathbf{l}$ which is the accessible one. We simplify this and assume we know this ordering or the locations we lookup are known.

### 6.2 Properties of the Backtranslation

The properties of the backtranslation are unchanged, the proofs however get more complex due to all the additional boilerplate.

To a slightly higher degree of complexity, doing this for full higher-order heaps is discussed precisely and proven formally in [1].

## References


[1] Marco Patrignani and Deepak Garg. Robustly Safe Compilation. In *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019*, ESOP'19, 2019.