Small Steps of the Skeleton Dance

Guillaume Ambal, Alan Schmitt, Sergueï Lenglet

Small Steps of the Skeleton Dance

Guillaume Ambal , Alan Schmitt , Sergueï Lenglet

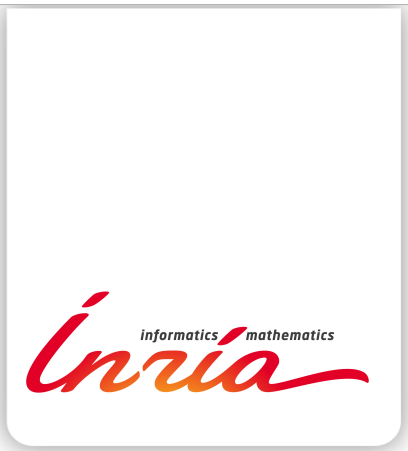
RESEARCH REPORT

N° 9363

September 2020

Project-Teams Celtique

ISSN 0249-6399 ISRN INRIA/RR--9363--FR+ENG



Small Steps of the Skeleton Dance

Guillaume Ambal *, Alan Schmitt †, Sergueï Lenglet ‡

Project-Teams Celtique

Research Report n° 9363 — September 2020 — 44 pages

Abstract: We present an automatic translation of a skeletal semantics written in big-step style into an equivalent semantics in small-step. This translation is implemented on top of the Necro tool, which lets us automatically generate an OCaml interpreter for the small step semantics and Coq mechanization of both semantics. We generate Coq certification scripts alongside the transformation. We illustrate the approach using a simple imperative language and show how it scales to larger languages.

Key-words: Big-Step, Small-Step, Skeletal Semantics, Operational Semantics

* Université Rennes 1, Rennes
† Inria, Rennes
‡ Université de Lorraine, Nancy

RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Transformation automatique d'une sémantique squelettique grand-pas en sémantique petit-pas

Résumé : Nous présentons une transformation automatique d'une sémantique squelettique écrite en style grand-pas vers une sémantique équivalent en style petit-pas. Cette transformation est implémentée dans l'outil Necro, ce qui nous permet de générer automatiquement un interpréteur Ocaml pour la sémantique petit-pas ainsi qu'une formalisation Coq des deux sémantiques. Nous générons un script de certification Coq en parallèle de la transformation. Nous illustrons notre approche sur un langage impératif simple.

Mots-clés : grand-pas, petit-pas, sémantique squelettique, sémantique opérationelle

1 Introduction

Any given programming language may come with many semantics. Even focusing on operational semantics, one can use a big-step (or natural) semantics [10], a small-step semantics [15], a context-based reduction semantics [19], or even an abstract machine [11, 6]. They are all equivalent, in the sense that they describe identical behaviors of programs, but some may be better adapted for some purposes, such as proving properties of the language, dealing with non-terminating or interacting programs, or being close to an implementation. To capture the essence of operational semantics without choosing a particular style, Bodin et al. [1] have recently proposed *skeletal semantics*, a meta language to provide a general and systematic way to describe the specification of a language by focusing on the structure of its semantics.

To illustrate this approach, consider the skeleton corresponding to a conditional `If(e,s1,s2)`, i.e., `if e then s1 else s2`.

$$\texttt{let } b = h(\texttt{e}) \texttt{ in } \begin{pmatrix} \texttt{isTrue}(b); h(\texttt{s1}) \\ \texttt{isFalse}(b); h(\texttt{s2}) \end{pmatrix}$$

In this skeleton, h is left undefined and stands for the recursive evaluation of a subterm, without specifying how it should be done. Similarly, `isTrue` and `isFalse` are not specified: they may complete or fail, in the latter case making the whole branch fail. One may thus read this skeleton as follows. Recursively evaluate `e` as b, then branch on two behaviors: compute `isTrue` with b then recursively evaluate `s1`, or compute `isFalse` with b then recursively evaluate `s2`.

A skeletal semantics is just syntax. To give it meaning, one defines an *interpretation*, which states what "recursively evaluate", "branch", or any primitive function such as `isTrue` mean. The seminal paper on skeletal semantics mainly defines two interpretations, one for a big-step semantics and one for an abstract semantics. It then shows how they can be proven to be related by simply proving that the interpretations of the primitive functions are related. To the best of our knowledge, skeletal semantics is a framework generic enough to express any semantics which can be written with inductive rules. It also benefits from a tool called Necro [5], which can for instance generate an OCaml [13] interpreter or a Coq [4] mechanization of a skeletal semantics given as input.

The most natural interpretation one can give for a skeletal semantics is a big-step one (Section 2.2). In some cases, a small-step semantics is better suited, for instance to describe programs that communicate, that may be interrupted, or that are interleaved. In fact, the semantics of the high-level language of CompCert [12] was initially written in a big-step style, and was later rewritten as small-step. To define small-step semantics for languages described as skeletal semantics, one can easily imagine writing an interruptible interpretation, that only evaluates part of a skeleton and returns the unevaluated part. This is closer to an abstract machine than a small step approach, however: in the latter, one *reconstructs* a term after some computation, and the evaluation is then restarted from this partially evaluated term.

We thus propose a different approach: automatically generate a skeletal semantics that behaves like a small-step semantics. Without reconstruction, the principle is quite simple: replace each recursive evaluation with a new constructor representing what is left to evaluate in the skeleton. For example in the `If` skeleton, we replace the first call with a constructor `If2` so that if `e` reduces to `e'`, `If(e, s1, s2)` reduces to `If2(e', s1, s2)`, and if `e'` is a value, `If2` continues with the branching. We would also create two new constructors for the evaluations of `s1` and `s2`.

While correct, this approach clutters the resulting semantics by introducing a lot of unnecessary constructors, as we could simply reuse `If` instead of creating `If2`. The goal and main challenge of our work is to reconstruct terms using the initial constructors as much as possible

to generate a minimal small-step semantics close to what one would write by hand, and to do so independently of the input language. We thus analyze skeletons to identify where reconstruction is not possible: e.g., `(while e do s)` does not reduce to `(while e' do s)` if `e` reduces to `e'`, as `e` may be needed in further executions of the loop. We create new constructors only in such cases.

We ensure that the resulting small-step semantics corresponds to the big-step one by producing for any input language an equivalence certificate checkable in Coq. The proof scheme we use to show that a sequence of small steps reductions implies a big step evaluation is simple enough that semantics-independent exhaustive search tactics are able to verify it. The reverse implication requires more guidance to avoid a case analysis explosion, so we generate from the resulting semantics lemmas stating congruence properties about the small-step reduction to help the tactics.

Our contributions are: an automatic method to generate a small-step skeletal semantics for any language given as a big-step skeletal semantics, along with a Coq script proving the correctness of the result and a small-step OCaml interpreter.

The paper is structured as follows. We formally introduce skeletal semantics in Section 2. We give a detailed example of the main phases of the transformation in Section 3, before formalizing them in Section 4. We show how to generate a Coq certificate of the equivalence between the resulting and input semantics in Section 5. We describe our implementation as an extension of Necro in Section 6. Finally, we evaluate our approach on several languages in Section 7. The appendix contains the result of each transformation step on a simple imperative language. The source code of our implementation, the generated interpreters and proof scripts for many languages are available at Inria's Gitlab.

2 Skeletal Semantics

Skeletal semantics is an approach to formalize the operational semantics of programming languages. The fundamental idea is to only specify the structure of evaluation functions (e.g., sequences of operations, non-deterministic choices, recursive calls) while keeping abstract basic operations (e.g., updating an environment or comparing two values). The motivation for this semantics is that the structure can be analyzed, transformed, or certified independently from the implementation choices of the basic operations.

2.1 Syntactic Entities

A skeletal semantics is composed of sorts, filters, hooks, and rules. Sorts represent categories for the elements of the language; they can be seen as types. We distinguish between *base* and *program* sorts. Base sorts are left unspecified and correspond to the base elements of the language, like environments or identifiers for variables. Program sorts are defined with a list of constructors, each having a precise typing.

Example 2.1. We write examples using the Necro [5] syntax, and use as a running example an imperative language called IMP. Sorts are defined with the keyword `type`; base sorts (`int`, `bool`, `ident`, `state`, and `value`) are only declared, while program sorts (`expr` and `stmt`) are declared alongside the signature of their constructors, like an algebraic datatype definition in OCaml.

```
type int
type bool
type expr =
| Iconst of int
| Bconst of bool
| Var of ident
| Plus of expr * expr
| Equal of expr * expr
| Not of expr
```

```
type ident
type state
type value
type stmt =
| Skip
| Assign of ident * expr
| Seq of stmt * stmt
| If of expr * stmt * stmt
| While of expr * stmt
```

Sorts `int` and `bool` represent integers and booleans, collected under the sort `value`. The sort `ident` represents identifiers for the variables of the language and `state` represents environments mapping variables to values. The two program sorts define the expressions and statements of the language.

A distinctive feature of skeletal semantics is that we do not need to further specify the implementation of base sorts. The only way we can manipulate them is through typed unspecified functions called *filters*, which represent the basic operations of the language. We can reason on the semantics as a whole by assuming some properties on these filters. For instance, the final representation of `state` does not matter as long as we define read and write operations as filters.

Example 2.2. Filters are declared with the keyword `val`. They are explicitly typed, using the keyword `unit` in case of a missing input or output. We consider the following filters for IMP.

```
val add : value * value -> value
val eq : value * value -> value
val neg : value -> value
val isTrue : value -> unit
val isFalse : value -> unit
```

```
val intToVal : int -> value
val boolToVal : bool -> value
val read : ident * state -> value
val write : ident * state * value -> state
```

Hooks correspond to the evaluation functions we want to define, operating on a program sort it pattern-matches: the behavior of the hook on a constructor is defined with a rule, whose main component is a *skeleton*. A skeleton represents the semantic behavior of a reduction. It is a sequence of bones, linked via a LetIn structure. A bone represents a single operation, that can either be a function call (filter or hook), a return of values, or a branching corresponding to a non-deterministic choice over several possible skeletons.

Example 2.3. We define the hooks `hexpr` and `hstmt` for the evaluation of respectively expressions and statements in Figure 1; the matched term is declared with the keyword `matching`. Branchings are written `branch .. (or ..)* end`, while the other types of bones (filter call, hook call, return) are not preceded by keywords, as we can easily tell them apart.

The rules for `If` and `While` illustrate branchings. In both cases, we evaluate the first term to get a value v. In most programming languages, we would then branch depending on v. We encode this behavior with the non-deterministic branchings of skeletal semantics by starting each branch with a filter either `isTrue` or `isFalse`, which causes one of the branches to fail.

Evaluating expressions with `hexpr` returns a state although it is never modified. This choice prepares for extensions of the language, such as function calls, where the state could be mutated.

Formally, we write $\widetilde{a}$ for a (possibly empty) tuple $(a_1, \ldots, a_n)$, and $\|\widetilde{a}\|$ for its size, here n. Unless explicitly stated, tuples are not expected to have the same arity. We write $\widetilde{a}, b$ and $\widetilde{a}, \widetilde{b}$ for the extension of a tuple on the right, and we have $\|\widetilde{a}, \widetilde{b}\| = \|\widetilde{a}\| + \|\widetilde{b}\|$. We write $a_i \in \widetilde{a}$ to state that a_i is an element of the tuple $\widetilde{a}$. Given a function or relation f, we write $\widetilde{f(a)}$ for $(f(a_1), \ldots, f(a_n))$ assuming $\|\widetilde{a}\| = n$. Similarly, assuming $\|\widetilde{a}\| = \|\widetilde{b}\| = n$, we write $\widetilde{g(a, b)}$ for $(g(a_1, b_1), \ldots, g(a_n, b_n))$.

```
hook hexpr (s : state, e : expr)
     matching e : state * value =
| Iconst (i) ->
  let v = intToVal (i) in
  (s, v)
| Bconst (b) ->
  let v = boolToVal (b) in
  (s, v)
| Var (x) ->
  let v = read (x, s) in
  (s, v)
| Plus (e1, e2) ->
  let (s', v1) = hexpr (s, e1) in
  let (s'', v2) = hexpr (s', e2) in
  let v = add (v1, v2) in
  (s'', v)
| Equal (e1, e2) ->
  let (s', v1) = hexpr (s, e1) in
  let (s'', v2) = hexpr (s', e2) in
  let v = eq (v1, v2) in
  (s'', v)
| Not (e') ->
  let (s', v) = hexpr (s, e') in
  let v' = neg (v) in
  (s', v')
```

```
hook hstmt (s : state, t : stmt)
     matching t : state =
| Skip -> s
| Assign (x, t') ->
  let (s', v) = hexpr (s, t') in
  write (x, s', v)
| Seq (t1, t2) ->
  let s' = hstmt (s, t1) in
  hstmt (s', t2)
| If (e1, t2, t3) ->
  let (s', v) = hexpr (s, e1) in
  branch
    let () = isTrue (v) in
    hstmt (s', t2)
  or
    let () = isFalse (v) in
    hstmt (s', t3)
  end
| While (e1, t2) ->
  let (s', v) = hexpr (s, e1) in
  branch
    let () = isTrue (v) in
    let s'' = hstmt (s', t2) in
    hstmt (s'', While (e1, t2))
  or
    let () = isFalse (v) in
    s'
  end
```

Figure 1: Hooks in IMP

We let c, f, and h range over respectively constructor, filter, and hook names. Assuming a countable set $\mathcal{V}$ of variables ranged over by v (and also w, x, y, and z), the grammar of terms (t), skeletons (S), and bones (B) is defined as follows.

$$
\begin{aligned}
t &::= v \mid c(\widetilde{t}) \\
S &::= \texttt{let}\ \widetilde{v} = B\ \texttt{in}\ S \mid B \\
B &::= \texttt{Filter}\ f\ (\widetilde{t}) \mid \texttt{Hook}\ h\ (\widetilde{t}, t) \mid \texttt{Return}\ (\widetilde{t}) \mid \texttt{Branching}\ (\widetilde{S})
\end{aligned}
$$

The skeletal semantics of a language consists in:

- a set ($T = T_b \cup T_p$) of sorts ranged over by s, combining base sorts T_b, and program sorts T_p ranged over by s_p;
- a set C of constructors, with a typing function $\texttt{csort} : C \to \widetilde{T} \times T_p$;
- a set F of filters, with a typing function $\texttt{fsort} : F \to \widetilde{T} \times \widetilde{T}$;
- a set H of hooks, with a typing function $\texttt{hsort} : H \to (\widetilde{T} \times T_p) \times \widetilde{T}$;
- a set R of rules of the form $h(\widetilde{y}, c(\widetilde{x})) := S$, assuming we have $\texttt{hsort}(h) = ((\widetilde{s}, s_p), \widetilde{s'})$, $\texttt{csort}(c) = (\widetilde{s''}, s_p)$, $\|\widetilde{y}\| = \|\widetilde{s}\|$, and $\|\widetilde{x}\| = \|\widetilde{s''}\|$.

The typing of constructors restricts their output to a term of program sort, while filters may produce terms of any sort. The input of a hook $\widetilde{T} \times T_p$ is composed of auxiliary terms of sort $\widetilde{T}$ and of the term being evaluated of program sort T_p. We define three projections $\texttt{hsort}_{\texttt{in}}$, $\texttt{hsort}_{\texttt{p}}$, and $\texttt{hsort}_{\texttt{out}}$ so that if $\texttt{hsort}(h) = ((\widetilde{s}, s_p), \widetilde{s'})$, then $\texttt{hsort}_{\texttt{in}}(h) = \widetilde{s}$, $\texttt{hsort}_{\texttt{p}}(h) = s_p$, and $\texttt{hsort}_{\texttt{out}}(h) = \widetilde{s'}$.

A rule $h(\widetilde{y}, c(\widetilde{x})) := S$ defines the behavior of h on the constructor c by the skeleton S, which describes the sequence of reductions to perform using the input variables $\widetilde{x}$ and $\widetilde{y}$. We assume the variables $\widetilde{x}, \widetilde{y}$ to be pairwise distinct, and to contain the free variables of S. We suppose that at most one rule handling c for h exists in R. The matching does not have to be exhaustive: a hook h without a rule for c simply cannot reduce terms with c as head constructor.

The formal definition of skeletal semantics presented in this section differs from the original one [1] mostly by introducing the LetIn structure and allowing more than one hook. We believe our version makes writing the skeletal semantics of a given language significantly easier.

2.2 Concrete Interpretation

The dynamic of a skeletal semantics is given by the *concrete interpretation* of the rules defining its hooks [1], which corresponds to a big-step semantics. This interpretation computes the result of applying a hook to a term by inductively interpreting the skeleton of the rule of the corresponding constructor and hook, under some environment mapping skeleton variables to values.

The interpretation supposes an instantiation of the base sorts and filters. For every base sort $s \in T_b$ we assume given a set $\mathcal{C}(s)$, representing its values. For every program sort $s_p \in T_p$, we inductively construct its set of closed program terms $\mathcal{C}(s_p)$ from the constructors of the skeletal semantics and the values of the different sets $\mathcal{C}(s)$ for $s \in T$. For every filter $f \in F$ with typing $\texttt{fsort}(f) = \widetilde{s}, \widetilde{s'}$, we assume a relation $\mathcal{R}_f$ between the elements of $\widetilde{\mathcal{C}(s)}$ and $\widetilde{\mathcal{C}(s')}$. If $\widetilde{a}$ are values of $\widetilde{\mathcal{C}(s)}$ and $\widetilde{b}$ are values of $\widetilde{\mathcal{C}(s')}$, we write $\mathcal{R}_f(\widetilde{a}) \Downarrow \widetilde{b}$ when the interpretation of f relates $\widetilde{a}$ to $\widetilde{b}$.

Example 2.4. For IMP, we instantiate the base sort `ident` with strings, `int` with integers ($\mathbb{Z}$), `bool` with Booleans ($\mathbb{B} = \{\top; \bot\}$), `value` with the disjoint union ($\mathbb{Z} \cup \mathbb{B}$), and `store` with partial functions from strings to values. The interpretations of the different filters are the following:

- `intToVal` and `boolToVal` inject their arguments in ($\mathbb{Z} \cup \mathbb{B}$);
- `read(x, s)` returns the result of applying `s` to `x`;
- `write(x, s, v)` returns the partial function mapping `x` to `v` and every `y` $\neq$ `x` to `s(y)` ;
- `eq(x, y)` returns $\top$ if both values are equal, $\bot$ otherwise;
- `add(x, y)` is only defined on integers so that $\mathcal{R}_{\texttt{add}}(i_1, i_2) \Downarrow (i_1 + i_2)$ for any i_1 and i_2;
- `neg` is only defined on booleans so that $\mathcal{R}_{\texttt{neg}}(\top) \Downarrow (\bot)$ and $\mathcal{R}_{\texttt{neg}}(\bot) \Downarrow (\top)$;
- `isTrue` only accepts $\top$ so that $\mathcal{R}_{\texttt{isTrue}}(\top) \Downarrow ()$;
- `isFalse` only accepts $\bot$ so that $\mathcal{R}_{\texttt{isFalse}}(\bot) \Downarrow ()$.

The relations above make the distinction between integers and booleans. We can define an interpretation where conditional branching on an integer is allowed, by extending the interpretation of the filters `isTrue` and `isFalse` as follows:

$$\mathcal{R}_{\texttt{isFalse}}(0) \Downarrow (); \qquad \forall i \neq 0,\ \mathcal{R}_{\texttt{isTrue}}(i) \Downarrow ()$$

$$\frac{\widetilde{\Sigma(t)} = \widetilde{b}, c(\widetilde{b'}) \qquad h(\widetilde{y}, c(\widetilde{x})) := S \in R \qquad \{y \widetilde{\mapsto} b\} + \{x \widetilde{\mapsto} b'\} \vdash S \Downarrow \widetilde{a}}{\Sigma \vdash \texttt{Hook}\ h\ \widetilde{t} \Downarrow \widetilde{a}} \qquad \frac{\mathcal{R}_f(\widetilde{\Sigma(t)}) \Downarrow \widetilde{a}}{\Sigma \vdash \texttt{Filter}\ f\ \widetilde{t} \Downarrow \widetilde{a}}$$

$$\frac{\widetilde{\Sigma(t)} = \widetilde{a}}{\Sigma \vdash \texttt{Return}\ \widetilde{t} \Downarrow \widetilde{a}} \qquad \frac{S_i \in \widetilde{S} \qquad \Sigma \vdash S_i \Downarrow \widetilde{a}}{\Sigma \vdash \texttt{Branching}\ \widetilde{S} \Downarrow \widetilde{a}} \qquad \frac{\Sigma \vdash B \Downarrow \widetilde{b} \qquad \Sigma + \{v \widetilde{\mapsto} b\} \vdash S \Downarrow \widetilde{a}}{\Sigma \vdash \texttt{let}\ \widetilde{v} = B\ \texttt{in}\ S \Downarrow \widetilde{a}}$$

Figure 2: Inference Rules for the Concrete Interpretation

The strength of skeletal semantics is that this choice is local to the interpretation of filters: we do not have to change anything in the definitions or interpretations of the hooks.

We write Σ for an environment mapping a finite set of variables in $\mathcal{V}$ (its *domain*) to values in $\cup_{s \in T} \mathcal{C}(s)$. We write $\Sigma(v)$ to access the mapping associated to v in Σ, and we extend the notation to terms $\Sigma(t)$ as follows: $\Sigma(c(\widetilde{t})) = c(\widetilde{\Sigma(t)})$. The environment mapping a single variable v to a value b is written $\{v \mapsto b\}$, and we write $\Sigma + \Sigma'$ for the update of Σ with Σ', so that $(\Sigma + \Sigma')(v) = \Sigma'(v)$ if v is in the domain of Σ', and $(\Sigma + \Sigma')(v) = \Sigma(v)$ otherwise.

The interpretation $\Sigma \vdash S \Downarrow \widetilde{a}$, defined in Figure 2, is a relation stating that S outputs the values $\widetilde{a}$ under the environment Σ; it assumes that the free variables of S are in the domain of Σ. A LetIn structure is evaluated sequentially, starting with B under Σ and then continuing with S under the environment updated with the outputs of B. The environment Σ is used when interpreting bones to turn the input terms $\widetilde{t}$ into values. These values are simply returned in the case of a `Return` bone. A filter call looks for a possible result related to these values in $\mathcal{R}_f$. A branching returns the result of one of its branch; it does not matter if some branches are stuck or non-terminating as long as one branch succeeds. To evaluate a hook call, we first compute the arguments of the hook, and find the rule corresponding to its constructor. We then interpret the skeleton of the rule under a new environment mapping the free variables of the skeleton to the appropriate values taken from Σ. Note that our approach differs from the one of [1]: they take the smallest fixpoint of a functional describing one step of the relation, whereas we directly define the semantics as an inductive definition.

This interpretation is inherently big-step, as a judgment $\Sigma \vdash S \Downarrow \widetilde{a}$ computes the final value returned by S. It is also non-deterministic, as apparent in the rules for branching and filter call, since $\mathcal{R}_f$ may relate several results to the same input.

Example 2.5. We interpret the following statement `st` with the hook `hstmt`:

```
st := (If Equal(Var("length"),Plus(Var("width"),2))
       Assign("width",Iconst(4))
       Assign("flag",Bconst(⊥)) )
```

The program is a simple conditional: if `length` = `width` + 2, then `width` := 4, otherwise `flag` := ⊥. We evaluate this statement in the state a_0 = {`length` ↦ 4 ; `width` ↦ 2}, so we expect the result to be a_1 = {`length` ↦ 4 ; `width` ↦ 4}. We remind that the rule is `hstmt(s, If(e1,t2,t3)) :=` $S_{\texttt{If}}$, where $S_{\texttt{If}}$ is the following skeleton (cf. Figure 1):

```
let (s', v) = hexpr (s, e1) in
branch
  let () = isTrue (v) in          (* | S1  *)
```

```
  hstmt (s', t2)                    (* |     *)
or
  let () = isFalse (v) in           (* || S2 *)
  hstmt (s', t3)                    (* ||    *)
end
```

We interpret this skeleton in the initial environment:

$$\Sigma_0 = \{\texttt{s} \mapsto a_0\,;\texttt{e1} \mapsto \texttt{Equal(Var("length"),Plus(Var("width"),2))};\\ \texttt{t2} \mapsto \texttt{Assign("width",Iconst(4))}\,;\texttt{t3} \mapsto \texttt{Assign("flag",Bconst(}\bot\texttt{))}\}$$

We recursively evaluate the first hook call on the values $\Sigma_0(\texttt{s},\texttt{e1})$ by finding the corresponding rule `hexpr(s, Equal(e1,e2))` $:= S_{\texttt{Equal}}$ in R. We interpret $S_{\texttt{Equal}}$ under the new environment $\Sigma_2 = \{\texttt{s} \mapsto a_0\,;\texttt{e1} \mapsto \texttt{Var("length")}\,;\texttt{e2} \mapsto \texttt{Plus(Var("width"),2)}\}$, which results in $(a_0, \top)$.

We then evaluate the branching in the extended environment $\Sigma_1 = \Sigma_0 + \{\texttt{s}' \mapsto a_0; \texttt{v} \mapsto \top\}$. Only the branch guarded by `isTrue` can succeed; with our choice for $\mathcal{R}_{\texttt{isTrue}}$, we have $\mathcal{R}_{\texttt{isTrue}}(\Sigma_1(v)) \Downarrow ()$, meaning that we pass this filter call.

Finally, we compute the recursive call on the values $(a_0, \texttt{Assign("width",Iconst(4))})$. Once again we look for the corresponding rule in R and create a new environment for this evaluation. As expected, this computation returns a_1 and it closes the derivation of $\Sigma_0 \vdash S_{\texttt{If}} \Downarrow a_1$.

$$\dfrac{\dfrac{\dfrac{\vdots}{\Sigma_2 \vdash S_{\texttt{Equal}} \Downarrow (a_0, \top)}}{\Sigma_0 \vdash \texttt{hexpr (s,e1)} \Downarrow (a_0, \top)} \qquad \dfrac{\dfrac{\mathcal{R}_{\texttt{isTrue}}(\top) \Downarrow ()}{\Sigma_1 \vdash \texttt{isTrue (v)} \Downarrow ()} \qquad \dfrac{\dfrac{\vdots}{\Sigma_3 \vdash S_{\texttt{Assign}} \Downarrow a_1}}{\Sigma_1 \vdash \texttt{hstmt (s',t2)} \Downarrow a_1}}{\dfrac{\Sigma_1 \vdash S_1 \Downarrow a_1}{\Sigma_1 \vdash \texttt{Branching}(S_1, S_2) \Downarrow a_1}}}{\Sigma_0 \vdash S_{\texttt{If}} \Downarrow a_1}$$

3 Overview on an Example

Given a big-step skeletal semantics, we transform it to produce a skeletal semantics whose concrete interpretation behaves like a small-step interpretation of the initial semantics. We present the main steps of the transformation on the IMP language, whose full semantics can be found in Appendix A. As values are a special kind of terms, we introduce their coercion in Section 3.1. We then determine which new constructors need to be introduced and their behavior in Section 3.2. Finally, we generate the small-step semantics in Section 3.3.

3.1 Coercions

The first phase of our transformation is to coerce return values into terms. Since we want small-step reductions to transform a term into another term of the same program sort, it means that values returned by hooks need to be considered as terms of the corresponding input sort.

In our example, we need to add constructors corresponding to the return sorts of the two hooks, one of sort $(\texttt{state}, \texttt{value}) \to \texttt{expr}$ for `hexpr`, and one of sort $\texttt{state} \to \texttt{stmt}$ for `hstmt`. The program sorts become:

```
type expr =
| Iconst of int
| ...
| Ret_hexpr of state * value
```

```
type stmt =
| Skip
| ...
| Ret_hstmt of state
```

In the final small-step semantics, we need to be able to extract these coerced values. To this end, we define hooks to unpack the values for each constructor we introduce. In our example, we get the two following functions:

```
hook getRet_hexpr (e : expr) matching e : state * value =
| Ret_hexpr (v1, v2) -> (v1, v2)
hook getRet_hstmt (t : stmt) matching t : state =
| Ret_hstmt v1 -> v1
```

These hooks are only defined for the corresponding newly created constructors, as trying to unpack the value of a term not fully reduced should fail.

The transitory semantics at this point of the transformation is available in Appendix B.

3.2 New Constructors

The second phase is to determine which new constructors are required in order to produce a small-step semantics. Most reduction rules use the state and the arguments of the constructor only once. For instance, the evaluation of the term $\texttt{Plus}(e_1, e_2)$ consists of first evaluating e_1, then evaluating e_2, then combining the results. If we make progress on one subterm, let us say $e_1 \to e_1'$, then we reconstruct the term as $\texttt{Plus}(e_1', e_2)$. We can discard the initial value of e_1 because the variables standing for e_1 and e_2 appear only once in the skeleton for $\texttt{Plus}(e_1, e_2)$. This allows us to reuse the constructor to rebuild a term after a step of computation.

In some cases, however, we cannot reconstruct using the same constructor after a step. The different problematic situations are detailed in Section 4.2; here we only describe the main problem, namely that we cannot remember two versions of the same term.

Unlike `Plus`, some constructors make use of their arguments several times in their reduction rules, such as `While`. The reduction of $\texttt{While}(e_1, t_2)$ might evaluate both e_1 and t_2 before cycling back to the original term $\texttt{While}(e_1, t_2)$. In a small-step setting, to reduce e_1, we need to remember both a working copy e_1' of the expression and its initial value to cycle back. We cannot store both e_1 and e_1' in the `While` constructor, so we create a new one `While1` to do so.

In practice, the second phase of our transformation analyzes each hook call to determine in which of the following categories if falls in.

- It is a tail-hook, i.e., a final hook call forwarding its return values, so there is no need for reconstruction.
- The terms being evaluated are only used once. In this case, we can reconstruct with the same constructor.
- Some of the evaluated terms are used elsewhere. The naive reconstruction does not work, and we need to create an additional constructor.

In the third case, the additional constructor we create mirrors the situation at the program point and carries two copies of the reused terms. We also extend the corresponding hook with a new reduction rule for this new constructor, which is roughly the remainder of the initial skeleton rooted at the analyzed hook call.

We illustrate our analysis on some of the IMP constructors. For `Plus`, we can reconstruct after each hook call:

```
| Plus (e1, e2) ->
  let (s', v1) = hexpr (s, e1) in       (* reconstruction *)
  let (s'', v2) = hexpr (s', e2) in     (* reconstruction *)
  let v = add (v1, v2) in
  (s'', v)
```

In both cases, the arguments of the calls—respectively `s`, `e1` and `s'`, `e2`—are not reused in the rest of the skeleton, so we can reconstruct at each program point using `Plus`.

For `Seq`, we have:

```
| Seq (t1, t2) ->
  let s' = hstmt (s, t1) in             (* reconstruction *)
  hstmt (s', t2)                        (* tail-hook *)
```

As before, we can reconstruct after the first hook call as `s` and `t1` are used only once. The second hook call is simply a tail-hook, so there is no need to worry about reconstruction.

The analysis gets more interesting for `While`:

```
| While (e1, t2) ->
  let (s', v) = hexpr (s, e1) in        (* new cons: While1 *)
  branch
    let () = isTrue (v) in
    let s'' = hstmt (s', t2) in         (* new cons: While2 *)
    hstmt (s'', While (e1, t2))         (* tail-hook *)
  or
    let () = isFalse (v) in
    s'
  end
```

The third hook call is a tail-hook, as it is the final instruction of one of the reduction paths. When analyzing the first one, we see that `e1` is reused later, thus we cannot reconstruct from this point. Similarly, we cannot reconstruct from the second hook since `t2` is reused in the tail-hook.

For each of these two calls, we need to create a new constructor corresponding to their respective program point. The new constructors are built with two different kinds of arguments. Firstly, we create an argument for every term being evaluated at the analyzed hook call, namely `s`, `e1` for the first one, and `s'`, `t2` for the other. Secondly, we create arguments for the variables needed in the rest of the skeleton; in our example, it means keeping `e1` and `t2` in both cases. However, we do not need to duplicate `s'` nor add an argument for `v` as they are no longer necessary. As a result, while most variables appear only once in the arguments of a new constructor, the contentious ones—used in and after the corresponding hook call—are duplicated. In the end, we extend the definition of IMP with the following constructors:

```
type stmt =
| ...
| While of expr * stmt
| While1 of state * expr * expr * stmt
| While2 of state * stmt * expr * stmt
| Ret_hstmt of state
```

We also add a new rule for each constructor introduced. The new skeleton consists in resuming the computation from the corresponding analyzed hook call, updating the input of the call to

make use of the new arguments of the constructor. The resulting rule for `While1` is almost the same as the one for `While`, while the skeleton of `While2` covers only its last two bones. We do not modify the rules for `While` or the other constructors at this stage.

```
hook hstmt (s : state, t : stmt) matching t : state =
| ...
| While1 (s0, e0, e1, t2) ->
  let (s', v) = hexpr (s0, e0) in
  branch
    let () = isTrue (v) in
    let s'' = hstmt (s', t2) in
    hstmt (s'', While (e1, t2))
  or
    let () = isFalse (v) in
    s'
  end
| While2 (s0, t0, e1, t2) ->
  let s'' = hstmt (s0, t0) in
  hstmt (s'', While (e1, t2))
```

The added rules become useful when we change the rules to introduce calls to `While1` and `While2`, as part of the last step of the transformation. Note that these added rules already include the information from the reconstruction analysis, see Section 4.2 for details.

The semantics after this phase of the transformation is provided in Appendix C.

3.3 Make the Skeletons Small-Step

The previous phases set the stage for the main transformation, but our semantics is still big-step at this point, since the hooks fully compute their arguments. The last phase of the transformation makes the hooks behave in a small-step way.

Firstly, we need to change the output sorts of every hook to make them match the input ones. The headers of the hooks become:

```
hook hexpr (s : state, e : expr) matching e : state * expr = ...
hook hstmt (s : state, t : stmt) matching t : state * stmt = ...
```

Doing so makes the sorts coherent with a small-step reduction process meant to be iterated.

More importantly, we need to update the skeletons themselves. We recall that skeletons are sequences of operations (bones), which are mostly composed of filter calls and hook calls. For our transformation, we consider that only hook calls correspond to reduction steps. The reason is that filters represent simple atomic operations that are not meant to be interrupted, while hook calls often correspond to the evaluation of a subterm. Thus, this last phase essentially focuses on transforming hook calls. We also need to take care of the result returned at the end of a skeleton, so that its sort matches the updated output sort of its hook. We distinguish four cases.

1 If the last bone of a skeleton is not a tail-hook, then we need to wrap the results differently to match the new typing of the hook. The final result needs to be coerced to the input program sort using one of the new constructors defined in the first phase (Section 3.1). For type checking, we also have to return the other arguments of the hook, even if they are not of any use. For instance, the output of the rule for the `Iconst` constructor in the initial big-step skeleton is `(s, v)` (cf. Figure 1). Using a coercion, we turn this pair into an expression `Ret_hexpr (s, v)`; if we

could we would return this term only, but the output sort of the `hexpr` hook is `(state * expr)`, so we also return a useless copy of `s`. The rule for `Iconst` is thus as follows.

```
| Iconst i ->
  let v = intToVal (i) in
  (s, Ret_hexpr (s, v))
```

2 The most interesting case is when we reach a hook call where we know we can reconstruct. In this situation, we are at a program point corresponding to the evaluation of a subterm, and we have two possibilities: either the subterm needs to be evaluated further, in which case we need to take a reduction step and reconstruct, or the subterm has been fully evaluated, and we need to extract its value and continue the reduction according to the rest of the skeleton. We distinguish the two behaviors in skeletal semantics using branches. For instance, transforming the first hook call of the `Plus` constructor produces a rule structured as follows:

```
| Plus (e1, e2) ->
  branch
    let (w1, w2) = hexpr (s, e1) in
    (w1, Plus (w2, e2))
  or
    let (s', v1) = getRet_hexpr (e1) in
    ...
  end
```

where the dots correspond to the transformation of the second hook call. In the first branch, we reconstruct as `(w1, Plus (w2, e2))`, overwriting the variables `s` and `e1` with the new terms resulting from the reduction step. In the second branch, we extract the coerced value using the hook defined alongside the constructor in Section 3.1. Even though we use a branching, the reduction is deterministic, as the definition of `getRet_hexpr` is restricted to coerced values, while `hexpr` operates only on terms that are not coerced values.

3 If we reach a hook call where we are not able to reconstruct, i.e., one of the calls for which we created a new constructor during the analysis, then the small-step function has to change the constructor after a reduction step. To simplify the semantics, we can equivalently decide to duplicate the necessary terms and change the constructor before reducing. To simplify even further, we consider the change of constructor to be a reduction step by itself; the next small step can then reduce the hook call. For instance, the reduction rule for the `While` constructor becomes:

```
| While (e1, t2) ->
  (s, While1 (s, e1, e1, t2))
```

We simply duplicate `s` and `e1` using the new constructor `While1`. We immediately return this new configuration. Calling the hook `hstmt` after that would then executes the skeleton for `While1` where the next reduction step and reconstruction actually take place. Similarly, we call the new constructor `While2` in the rule created for `While1`:

```
| While1 (s0, e0, e1, t2) ->
  branch                                (* |                      *)
    let (w1, w2) = hexpr (s0, e0) in    (* | first hook call      *)
    (s, While1 (w1, w2, e1, t2))        (* | transformed as       *)
  or                                    (* | previously           *)
    let (s', v) = getRet_hexpr (e0) in  (* |                      *)
```

```
    branch                                  (*   # initial structure     *)
      let () = isTrue (v) in                (*   #                       *)
      (s, While2 (s', t2, e1, t2))          (*     || second hook call   *)
    or                                      (*   #                       *)
      let () = isFalse (v) in               (*   #                       *)
      (s, Ret_hstmt s')                     (*    ## coerced return      *)
    end                                     (*   #                       *)
  end                                       (* |                        *)
```

This shows that the final transformation phase operates not only on the rules of the initial semantics, but also on the ones created during the analysis.

4 Finally, we also cut the tail-hook calls to simplify the semantics. This creates administrative small-steps of the form $s, \texttt{If}(\texttt{True}, t_2, t_3) \to s, t_2$ where no subcomputation takes place. It generates behaviors closer to usual pen-and-paper definitions. We can see this with the `Seq` constructor: the first hook call is transformed as previously, but the second is turned into a return.

```
| Seq (t1, t2) ->
  branch                                (* |                                *)
    let (w1, w2) = hstmt (s, t1) in     (* | first hook call                 *)
    (w1, Seq (w2, t2))                  (* | transformed as previously       *)
  or                                    (* |                                *)
    let s' = getRet_hstmt (t1) in       (* |                                *)
    (s', t2)                            (*    || second call becomes return  *)
  end                                   (* |                                *)
```

This final phase produces a small-step skeletal semantics where each hook call reduces its arguments only once. It is equivalent to the initial big-step one, in the sense that evaluating a term with either semantics produces the same value. The complete result of the transformation on IMP can be found in Appendix D. We state the equivalence between the two semantics in Section 5.

4 Transformation Steps

We define formally the different phases of our transformation using the notations introduced in Section 2.1. We start with a given skeletal semantics $(T_b, T_p, C_0, F, H_0, R_0, \texttt{csort}, \texttt{fsort}, \texttt{hsort})$ that we modify and expand, introducing new sets of constructors, hooks, or rules at each stage. To keep the number of new symbols to a minimum, we reuse the names `csort` and `hsort` throughout the transformation even if these functions are slightly modified.

4.1 Coercions

The first step is to add coercions for return values like in Section 3.1. For every hook we add a new constructor to pack its result as well as the corresponding hook to unpack it.

$$C_1 = C_0 \cup \{\texttt{Ret\_h} \mid \texttt{h} \in H_0\} \qquad \text{with } \texttt{csort}(\texttt{Ret\_h}) = (\texttt{hsort}_{\texttt{out}}(\texttt{h}), \texttt{hsort}_{\texttt{p}}(\texttt{h}))$$
$$H_1 = H_0 \cup \{\texttt{getRet\_h} \mid \texttt{h} \in H_0\} \qquad \text{with } \texttt{hsort}(\texttt{getRet\_h}) = ((), \texttt{hsort}_{\texttt{p}}(\texttt{h})), \texttt{hsort}_{\texttt{out}}(\texttt{h}))$$

We remind that $\texttt{hsort}_{\texttt{in}}$, $\texttt{hsort}_{\texttt{p}}$, and $\texttt{hsort}_{\texttt{out}}$ are the projections of the typing function $\texttt{hsort} : H_0 \to (\widetilde{T} \times T_p) \times \widetilde{T}$, representing respectively the sorts of input states, the program

sort being reduced, and the output sort of the hook. Each coercion turns the return values $\texttt{hsort}_{\texttt{out}}(\texttt{h})$ of the corresponding hook into an executable program of sort $\texttt{hsort}_{\texttt{p}}(\texttt{h})$.

Each extracting hook takes this program sort as its single input, as it does not depend on any environment sort to reduce. It is defined only for the constructor it destructs. We know how many variables it returns by looking at the output sort of the hook.

$$R_1 = R_0 \cup \{\texttt{getRet\_h}(\texttt{Ret\_h}\ \widetilde{v}) := \texttt{Return}\ \widetilde{v} \mid \texttt{h} \in H_0 \wedge \|\texttt{hsort}_{\texttt{out}}(\texttt{h})\| = \|\widetilde{v}\|\}$$

We finish this phase with the skeletal semantics $(T_b, T_p, C_1, F, H_1, R_1, \texttt{csort}, \texttt{fsort}, \texttt{hsort})$ and extended typing functions `csort` and `hsort`.

4.2 New Constructors

As presented in the overview, the second phase consists in an analysis of the hook calls to split them into three different categories. We exploit the results of the analysis during the final stage of the transformation. For the formal presentation, we introduce an extended skeletal semantics as an intermediate representation to carry over the information we need—the implementation uses ad-hoc data structures instead. We annotate hook calls with either `Tail` for a tail-hook, `Recons` for calls from which we can reconstruct with the same constructor, or `New` c when we create a new constructor c.

$$\begin{aligned} a &::= \texttt{New}\ c \mid \texttt{Tail} \mid \texttt{Recons} \\ S &::= \texttt{let}\ \widetilde{v} = B\ \texttt{in}\ S \mid B \\ B &::= \texttt{Filter}\ f\ \widetilde{t} \mid \texttt{Hook}\ a\ h\ (\widetilde{t}, t) \mid \texttt{Return}\ \widetilde{t} \mid \texttt{Branching}\ \widetilde{S} \end{aligned}$$

We proceed in two steps at this stage: we first analyze the rules then we create new constructors and their rules.

4.2.1 Analysis

A simple way to make the semantics small-step would be to introduce a new constructor for each hook call. While it is safe to do so, the resulting semantics would be unnecessarily bloated, as we can reuse constructors and reconstruct in many cases. Our goal is to reconstruct as much as possible to obtain a semantics close to the usual small-step semantics. It turns out that reconstruction is not possible in the following cases:

- after a filter call;
- in the continuation of a branching;
- if an argument of the hook call is not a variable, or if it is used several times in the skeleton.

Firstly, even if we do not consider computing a filter as a step, we do not want to recompute the same filter several times. A reconstruction implies that the whole skeleton is evaluated up to the hook call at each reduction step, meaning that a filter placed before the hook call would be called at every reduction step. This could have unintended consequences if the filters are defined with side effects. We therefore give up on reconstruction if the analyzed hook is after a filter call.

Secondly, we need to take into account the non-determinism induced by a branching. In a skeleton like $\texttt{let}\ \widetilde{v} = \texttt{Branching}(\texttt{Hook}\ h_1\ \widetilde{t_1},\ \texttt{Hook}\ h_2\ \widetilde{t_2})\ \texttt{in Hook}\ h\ \widetilde{t}$, two different reduction paths lead to the hook call after the branching. The premise of reconstruction is that reevaluating

$$
\begin{array}{rcll}
[\texttt{Branching}(S_1,..,S_n)]^{hr,V}_{L,b} & \triangleq & \texttt{Branching}([S_1]^{hr,V}_{L,b},..,[S_n]^{hr,V}_{L,b}) & \\
[\texttt{Filter}\ f\ \widetilde{t}]^{hr,V}_{L,b} & \triangleq & \texttt{Filter}\ f\ \widetilde{t} & \\
[\texttt{Return}\ \widetilde{t}]^{hr,V}_{L,b} & \triangleq & \texttt{Return}\ \widetilde{t} & \\
[\texttt{Hook}\ h\ \widetilde{t}]^{hr,V}_{L,b} & \triangleq & \texttt{Hook Tail}\ h\ \widetilde{t} & \text{if } L=\top, h=hr \\
[\texttt{Hook}\ h\ \widetilde{w}]^{hr,V}_{L,b} & \triangleq & \texttt{Hook Recons}\ h\ \widetilde{w} & \text{if } b=\top, \widetilde{w}\in V \\
[\texttt{Hook}\ h\ \widetilde{t}]^{hr,V}_{L,b} & \triangleq & \texttt{Hook (New}\ c\texttt{)}\ h\ \widetilde{t} & \text{otherwise, } c \text{ fresh} \\
[\texttt{let}\ \widetilde{v}=B\ \texttt{in}\ S]^{hr,V}_{L,b} & \triangleq & \texttt{let}\ \widetilde{v}=[B]^{hr,V}_{\bot,b}\ \texttt{in}\ [S]^{hr,V}_{L,\bot} & \text{if } B\neq \texttt{Hook}\ h\ \widetilde{t} \\
[\texttt{let}\ \widetilde{v}=\texttt{Hook}\ h\ \widetilde{w}\ \texttt{in}\ S]^{hr,V}_{L,b} & \triangleq & \texttt{let}\ \widetilde{v}=\texttt{Hook Recons}\ h\ \widetilde{w}\ \texttt{in}\ [S]^{hr,V}_{L,b} & \text{if } b=\top, \widetilde{w}\in V \\
[\texttt{let}\ \widetilde{v}=\texttt{Hook}\ h\ \widetilde{t}\ \texttt{in}\ S]^{hr,V}_{L,b} & \triangleq & \texttt{let}\ \widetilde{v}=\texttt{Hook (New}\ c\texttt{)}\ h\ \widetilde{t}\ \texttt{in}\ [S]^{hr,V}_{L,b} & \text{otherwise, } c \text{ fresh}
\end{array}
$$

Figure 3: Hook calls analysis

the skeleton from the start should lead to the same evaluation context. However, reevaluating the skeleton in such a situation may take a different path and reach the last hook call with different values bound to the variables in $\widetilde{v}$. As such, we give up on reconstruction if the analyzed hook is in the continuation of a branching.

Lastly, as illustrated in the overview, reconstruction means that we should be able to store the partially reduced terms in the constructor being evaluated. It is not possible if some of the arguments of the constructor are not variables, or if these variables are reused in the skeleton.

Formally, the annotation process of a given skeleton S is noted $[S]^{hr,V}_{L,b}$, where:

- L is a boolean indicating if we are at the toplevel of the LetIn structure of the main skeleton, used to detect tail-hook calls;
- b is a boolean indicating if we are at a position allowing for reconstruction, i.e., indicating whether we are after a filter call or in the continuation of a branching;
- V is the list of the variables that are only used once throughout the whole initial skeleton;
- hr is the name of the hook function corresponding to the rule being analyzed, also used to detect tail-hook calls, as detailed below.

The analysis is defined in Figure 3. Given a rule $h(\widetilde{y}, c(\widetilde{x})) := S$, we compute the set of variables that are used exactly once in S, written $\texttt{SglUse}(S)$, and we fix V and hr as respectively $\texttt{SglUse}(S)$ and h. The parameter V and hr are constants while L and b are initialized at $\top$ and may change during the analysis.

The analysis goes through the skeleton, leaving filters calls and returned values unchanged. As expected, the boolean b is set to $\bot$ after going through a filter call or a branching. Similarly, L is switched to $\bot$ in the first part of a LetIn structure.

A final hook call is considered a tail-hook if and only if it is situated at the toplevel of the main skeleton ($L = \top$) and if the hook being called is the one being analyzed ($h = hr$). The

second condition prevents typing issues. In the initial big-step semantics, a rule from a hook h_1 can make a final call to a hook h_2 if they have the same return sorts. For instance, we could define the evaluation of a list of expressions as simply evaluating the head of the list:

```
hook h2 (e : expr) matching e : value = ...
hook h1 (l : exprlist) matching l : value =
| Cons(e, l') -> h2 (e)
```

However, a small-step hook should have the same return sort as its input sort, a change we do in the last phase of the transformation:

```
hook h2 (e : expr) matching e : expr = ...
hook h1 (l : exprlist) matching l : exprlist = ...
```

The call to h_2 has to be modified to make the sorts match, hence it cannot be a tail-hook.

A hook call can only be reconstructed if we did not pass a filter call or a branching ($b = \top$) and every term is a variable not used elsewhere ($\widetilde{w} \in V$). In the case a hook call can be annotated with either `Tail` or `Recons`, we choose to give precedence to the former, because tail-hooks are more specific and lead to simpler skeletal semantics at the end of the transformation. Hook calls that cannot be annotated `Tail` or `Recons` are instead associated with a fresh constructor name created on the fly.

We apply the analysis to every skeleton in the semantics, updating the set of rules as follows.

$$R_2 = \{h(\widetilde{y}, c(\widetilde{x})) := [S]^{h,\texttt{SglUse}(S)}_{\top,\top} \mid (h(\widetilde{y}, c(\widetilde{x})) := S) \in R_1\}$$

4.2.2 Processing

After the analysis, we process every hook call annotated with a fresh constructor name c to compute the sort and rule of c. When traversing a skeleton to find such a call, we construct its continuation—the rest of the computation, represented as a context—as it is used to create the fresh skeleton of the new rule. To do so, we define a monadic bind on skeletons, noted $< S_1 \mid \widetilde{x} \mid S_2 >$, which executes S_1, binds the results to $\widetilde{x}$ then executes S_2.

$$< \texttt{let}\ \widetilde{y} = B\ \texttt{in}\ S_1 \mid \widetilde{x} \mid S_2 > \triangleq \texttt{let}\ \widetilde{y} = B\ \texttt{in}\ < S_1 \mid \widetilde{x} \mid S_2 >$$
$$< B \mid \widetilde{x} \mid S_2 > \triangleq \texttt{let}\ \widetilde{x} = B\ \texttt{in}\ S_2$$

We define contexts representing the continuation of a specific bone or skeleton as follows.

$$E ::= [\cdot] \mid < E \mid \widetilde{x} \mid S >$$

Processing a hook call annotated with c in a given rule $r = (\texttt{hr}(\widetilde{y}, c_r(\widetilde{x})) := S_r)$ is done in two steps: we go through S_r to find the call and build its continuation, then we update the semantics by adding c, its typing, and its rule. We write $[\![S]\!]^r_E$ for the first step, where r is the rule under consideration, E the continuation built so far, and S the skeleton we traverse (initially S_r). We write $\texttt{process}(c, h, r, E)$ for the second step, i.e., extending the hook h with a rule for c built from r and E.

The operation $[\![S]\!]^r_E$ inductively goes through S, building the continuation in its parameter

E and returning the set of new rules.

$$\begin{aligned}
[\![\texttt{Branching}\ (S_1,..,S_n)]\!]^r_E &\triangleq [\![S_1]\!]^r_E \cup \ldots \cup [\![S_n]\!]^r_E \\
[\![\texttt{Filter}\ f\ \widetilde{t}]\!]^r_E = [\![\texttt{Return}\ \widetilde{t}]\!]^r_E &\triangleq \emptyset \\
[\![\texttt{Hook Recons}\ h\ \widetilde{t}]\!]^r_E = [\![\texttt{Hook Tail}\ h\ \widetilde{t}]\!]^r_E &\triangleq \emptyset \\
[\![\texttt{Hook}\ (\texttt{New}\ c)\ h\ \widetilde{t}]\!]^r_E &\triangleq \{\texttt{process}(c,h,r,E)\} \\
[\![\texttt{let}\ \widetilde{v} = B\ \texttt{in}\ S]\!]^r_E &\triangleq [\![B]\!]^r_{<E|\widetilde{v}|S>} \cup [\![S]\!]^r_E
\end{aligned}$$

Once a hook call h needing c is found, we generate the corresponding rule using $\texttt{process}(c,h,r,E)$. Assuming $r = (\texttt{hr}(\widetilde{y}, c_r(\widetilde{x})) := S_r)$, we proceed as follows.

We first compute the variables needed as arguments of c and their respective sort. We start with the variables $\widetilde{z}$ (with sort $\widetilde{s}$) needed to evaluate E, which are the ones occurring in E, but not defined in E and not in $\widetilde{y}$. We do not need to include the variables $\widetilde{y}$ as arguments of c since they are already accessible at that program point. We then introduce fresh variables $\widetilde{w}$ to evaluate the current hook call h, as it is the first bone of the new rule for c. The sorts $\widetilde{u}$ of the variables $\widetilde{w}$ are given by the input sorts of h, i.e., $\widetilde{u} = (\texttt{hsort}_{\texttt{in}}(h), \texttt{hsort}_{\texttt{p}}(h))$. Finally, the typing of c is $\texttt{csort}(c) = ((\widetilde{u}, \widetilde{s}), \texttt{hsort}_{\texttt{p}}(\texttt{hr}))$, since the new constructor should build the program sort evaluated by the analyzed rule.

We then create the new rule for c as $\texttt{hr}(\widetilde{y}, c(\widetilde{w}, \widetilde{z})) := E[\texttt{Hook Recons}\ h\ \widetilde{w}]$: it evaluates h with the variables $\widetilde{w}$ and then E with $\widetilde{y}$ and $\widetilde{z}$. The call h can be reconstructed as the variables have been defined so that there is no overlap in their uses.

Example 4.1. We apply the process to the second hook call in the rule for `While` presented in Section 3.2. We reach this hook call B in a context E:

$$\begin{aligned}
B &:= \texttt{ Hook (New While2) hstmt (s', t2)} \\
E &:= < [\cdot] \mid \texttt{s''} \mid \texttt{Hook Tail hstmt (s'',While (e1, t2))} >
\end{aligned}$$

The new constructor needs the variables `(e1 : expr)` and `(t2 : stmt)` to evaluate E, but not `s''`, as it is defined by E. We get the input sorts $\widetilde{u} = (\texttt{state}, \texttt{stmt})$ of the hook call, for which we create the fresh variables $\widetilde{w} = (\texttt{s0}, \texttt{t0})$. From there we type the new constructor and create its rule:

$\texttt{csort(While2)} = (\texttt{state}, \texttt{stmt}, \texttt{expr}, \texttt{stmt}), \texttt{stmt}$

```
hstmt(s, While2(s0, t0, e1, t2)) :=
  let s'' = Hook Recons hstmt (s0, t0) in Hook Tail hstmt (s'',While (e1, t2))
```

To sum up, the processing phase consists of running the scanning $[\![S_r]\!]^r_{[\cdot]}$ for every rule $r = (\texttt{hr}(\widetilde{y}, c_r(\widetilde{x})) := S_r) \in R_2$, resulting in a new set $C_3 \supseteq C_1$ of constructors and a new set $R_3 \supseteq R_2$ of rules. The output of this phase is the extended semantics $(T_b, T_p, C_3, F, H_1, R_3, \texttt{csort}, \texttt{fsort}, \texttt{hsort})$.

4.2.3 Optimization

With the first analysis, we determine which hook calls can be reconstructed using the same constructor. The resulting annotations are still present in the new rules we create, but may not be as accurate as they could be in presence of the new constructors. With a constructor restarting the computation at a closer program point, some hook calls can now be reconstructed. A common example, presented below, is a rule evaluating two values with the same state.

As an optional optimization, we can repeat the analysis on the newly created skeletons to increase the number of reconstructions. However, such an optimization also requires to garbage collect constructors and rules that have been introduced but are no longer needed. For this, we go through the final skeletal semantics and remove the constructors and rules that are no longer reachable from an initial term.

Example 4.2. Consider the following hook and constructor:

```
hook h (s:state, t:term) matching t : value :=
| C(t1,t2) ->
  let v1 = h (s, t1) in
  let v2 = h (s, t2) in
  merge (v1, v2)
```

Both hook calls make use of the variable `s`, so `s` is not part of the set $\texttt{SglUse}(S)$ used for the analysis. As a result, none of the hook calls can be reconstructed from, and the analysis creates two constructors `C1` and `C2`. Then the processing phase builds the corresponding rules and we reach the following situation:

```
hook h (s:state, t:term) matching t : value :=
| C(t1,t2) ->
  let v1 = h (s, t1) in    (* new cons: C1 *)
  let v2 = h (s, t2) in    (* new cons: C2 *)
  merge (v1, v2)
| C1(s0,t0,t2) ->
  let v1 = h (s0, t0) in   (* reconstruction *)
  let v2 = h (s, t2) in    (* new cons: C2 *)
  merge (v1, v2)
| C2(s0,t0,v1) ->
  let v2 = h (s0, t0) in   (* reconstruction *)
  merge (v1, v2)
```

Inside the new evaluation rule of `C1`, the second hook call can be reconstructed since `s` is now only used once. Intuitively, there is no need for a second new constructor `C2`. By repeating the analysis on the new rules we find a new possible reconstruction:

```
| C1(s0,t0,t2) ->
  let v1 = h (s0, t0) in   (* reconstruction *)
  let v2 = h (s, t2) in    (* reconstruction *)
  merge (v1, v2)
```

At this point, `C2` still appears in the rule for `C` so we cannot get rid of it. However, at the end of the full transformation it will be apparent that `C` immediately uses the constructor `C1` and that `C2` cannot be reached. We garbage collect it at this point.

4.3 Distribute Branchings

This phase is not present in the extended example as the issue it solves does not occur in IMP.

Reconstructing terms is problematic for hooks in nested computations. In the structure `let x = (let y = Hook eval t in S1) in S2`, a small-step transformation of `eval` may return a partially evaluated term which ends up stored in `x`, while `S2` may expect `x` to contain a value; for example `S2` may start by filtering `x` with `isTrue`. We avoid the issue by sequencing such nested computations as `let y = Hook eval t in let x = S1 in S2`, and the hook transformation of Section 4.4 ensures that `x` may only contain a value.

The grammar of skeletal semantics of Section 2.1 does not allow for nested `LetIn`, but the same issue is present for branchings inside `LetIn`. We therefore transform a skeleton of the form $(\texttt{let}\ \widetilde{v} = \texttt{Branching}(S_1, S_2)\ \texttt{in}\ S)$ into $\texttt{Branching}(\texttt{let}\ \widetilde{v} = S_1\ \texttt{in}\ S, \texttt{let}\ \widetilde{v} = S_2\ \texttt{in}\ S)$, so that the reconstructing hook calls in S_1 and S_2 can be transformed in the final phase.

The distribution of `LetIn` over `Branching`, noted $\{S\}$ is recursive and makes use of the binding on skeletons $< S_1 \mid \widetilde{v} \mid S_2 >$ defined previously.

$$
\begin{aligned}
\{\texttt{Branching}(S_1, .., S_n)\} &\triangleq \texttt{Branching}(\{S_1\}, .., \{S_n\}) \\
\{B\} &\triangleq B \qquad \text{if } B \neq \texttt{Branching(..)} \\
\{\texttt{let}\ \widetilde{v} = \texttt{Branching}(S_1, .., S_n)\ \texttt{in}\ S\} &\triangleq \texttt{Branching}(\{< S_1 \mid \widetilde{v} \mid S >\}, .., \{< S_n \mid \widetilde{v} \mid S >\}) \\
\{\texttt{let}\ \widetilde{v} = B\ \texttt{in}\ S\} &\triangleq \texttt{let}\ \widetilde{v} = B\ \texttt{in}\ \{S\} \qquad \text{if } B \neq \texttt{Branching(..)}
\end{aligned}
$$

We apply this operation to every skeleton of our semantics:

$$R_4 = \{\texttt{h}(\widetilde{y}, c(\widetilde{x})) := \{S\} \mid (\texttt{h}(\widetilde{y}, c(\widetilde{x})) := S) \in R_3\}$$

Note that duplicating the continuations recursively will not exponentially grow the size of the final small-step semantics. We generate new constructors before this phase, hence the generated constructors are shared between branches, avoiding any unwanted bloat. As an optimization, this distribution and duplication could also be skipped for branchings only containing filter calls and no hook calls.

4.4 Make the Skeletons Small-Step

With the results of the analysis and the preprocessing done so far, we are ready to make the initial hooks small-step. As explained in the overview, we first update their output sorts.

$$\forall h \in H_0 \text{ with } \texttt{hsort}(h) = ((\widetilde{s}, s_p), \widetilde{u}),\ \text{we redefine: } \texttt{hsort}(h) = ((\widetilde{s}, s_p), (\widetilde{s}, s_p))$$

We only change the initial hooks—the ones in H_0—as the hooks `getRet_h` added in H_1 have been created with the desired, and different, output sorts: they actually extract the value from a term.

We then treat the skeletons defining these hooks, including the rules added in Section 4.2.2. At this stage of the transformation, we argue these skeletons respect the following simplified grammar, either directly or with a simple modification.

$$
\begin{aligned}
S &::= \texttt{let}\ \widetilde{v} = B\ \texttt{in}\ S \mid \texttt{Branching}\ \widetilde{S} \mid \texttt{Hook Tail}\ h\ \widetilde{t} \mid \texttt{Return}\ \widetilde{t} \\
B &::= \texttt{Filter}\ f\ \widetilde{t} \mid \texttt{Hook Recons}\ h\ \widetilde{w} \mid \texttt{Hook}\ (\texttt{New}\ c)\ h\ \widetilde{t} \mid \texttt{Return}\ \widetilde{t}
\end{aligned}
$$

A skeleton $\texttt{let}\ \widetilde{v} = \texttt{Branching}\ \widetilde{S}\ \texttt{in}\ S$ is impossible because of the distribution step of Section 4.3. A tail-hook is necessarily a final hook, so a skeleton $\texttt{let}\ \widetilde{v} = \texttt{Hook Tail}\ h\ \widetilde{t}\ \texttt{in}\ S$ is also not possible. Whenever a hook call is annotated `Recons`, the analysis of Section 4.2.1 implies that its input terms are all variables $\widetilde{w}$. If a skeleton ends with B that is not a tail-hook (i.e., a bone $\texttt{Filter}\ f\ \widetilde{t}$ or $\texttt{Hook}\ a\ h\ \widetilde{t}$ with $a \neq \texttt{Tail}$), we can transform it into an equivalent skeleton following the grammar above by delaying the return. For this, we replace B with the skeleton $\texttt{let}\ \widetilde{z} = B\ \texttt{in Return}\ \widetilde{z}$, where $\widetilde{z}$ are freshly created variables, in number corresponding to the output sort of B. As such, it is sufficient to define our procedure on skeletons respecting the simplified grammar.

Assuming $r = (\texttt{hr}(\widetilde{y}, c_r(\widetilde{x})) := S_r)$,

$$
\begin{aligned}
\| \texttt{Branching}\ (S_1, .., S_n) \|^r_\sigma &\triangleq \texttt{Branching}\ (\| S_1 \|^r_\sigma, .., \| S_n \|^r_\sigma) \\
\| \texttt{let}\ \widetilde{v} = \texttt{Return}\ \widetilde{t}\ \texttt{in}\ S \|^r_\sigma &\triangleq \texttt{let}\ \widetilde{v} = \texttt{Return}\ \widetilde{t}\ \texttt{in}\ \| S \|^r_\sigma \\
\| \texttt{let}\ \widetilde{v} = \texttt{Filter}\ f\ \widetilde{t}\ \texttt{in}\ S \|^r_\sigma &\triangleq \texttt{let}\ \widetilde{v} = \texttt{Filter}\ f\ \widetilde{t}\ \texttt{in}\ \| S \|^r_\sigma \\
\| \texttt{Return}\ \widetilde{t} \|^r_\sigma &\triangleq \texttt{Return}\ (\widetilde{y}, \texttt{Ret\_hr}(\widetilde{t})) \\
\| \texttt{Hook Tail hr}\ \widetilde{t} \|^r_\sigma &\triangleq \texttt{Return}\ \widetilde{t} \\
\| \texttt{let}\ \widetilde{v} = \texttt{Hook}\ (\texttt{New}\ c)\ h\ \widetilde{t}\ \texttt{in}\ S \|^r_\sigma &\triangleq \texttt{Return}\ (\widetilde{y}, c(\widetilde{t}, \widetilde{z_c})) \\
&\qquad \text{where } (\texttt{hr}(\widetilde{y}, c(\widetilde{w_c}, \widetilde{z_c})) := S_c) \in R_4 \\
\| \texttt{let}\ \widetilde{v} = \texttt{Hook Recons}\ h\ (\widetilde{w'}, w)\ \texttt{in}\ S \|^r_\sigma &\triangleq \texttt{Branching}(S_1, S_2) \quad \text{where}
\end{aligned}
$$

$$
\begin{aligned}
S_1 &= \texttt{let}\ (\widetilde{z'}, z) = \texttt{Hook}\ h\ (\widetilde{w'}, w)\ \texttt{in Return}\ \ (\widetilde{y}, c_r(\widetilde{x}))(\sigma\,;[\widetilde{z'/w'}]\,;[z/w]) \quad \widetilde{z'}, z \text{ fresh} \\
S_2 &= \texttt{let}\ \widetilde{v} = \texttt{Hook getRet\_h}\ (w)\ \texttt{in}\ \| S \|^r_{\sigma\,;[\texttt{Ret\_h}(\widetilde{v})/w]}
\end{aligned}
$$

Figure 4: Transformation of a Skeleton

The transformation relies on a substitution to remember how the initial arguments of the rule are changed through the different hook calls, as we show in Example 4.4. A substitution σ is a total mapping from variables to terms equal to the identity except on a finite set of variables called its domain. We write $x\sigma$ for the application of σ to x, ϵ for the identity substitution, and $[t/x]$ for the substitution whose domain is $\{x\}$ and such that $x\sigma = t$. We extend the notion to terms $t\sigma$ and tuples $\widetilde{t}\sigma$ as expected. Given two substitutions σ and σ', we define their sequence $\sigma\,;\sigma'$ so that $x(\sigma\,;\sigma') = (x\sigma)\sigma'$ for all x.

Given an extended skeleton S_r, a rule $r = (\texttt{hr}(\widetilde{y}, c_r(\widetilde{x})) := S_r)$, and a substitution σ representing the knowledge accumulated so far, the transformation $\| S_r \|^r_\sigma$ defined in Figure 4 results in a plain skeleton—without annotations. The first three rules are simple inductive cases, while the last four are the cases sketched in the overview (Section 3.3).

We coerce the results of a final `Return` bone with the constructor `Ret_hr` defined in Section 4.1. We remind that we also return the environment variables $\widetilde{y}$ of the rule to respect the updated typing of the hook.

A tail-hook is simply turned into a return, as the hook being called is identical to the one where the current rule is defined (see Figure 3).

As explained in Section 3.3, a hook annotated (`New` c) is turned into a return with a term built with c, so that the rule created for c in Section 4.2.2 can later perform the expected small-step reduction. One might be surprised the hook call disappears, it is simply delegated to the rule for the new constructor c (see Section 4.2.2 right before Example 4.1). To compute the arguments of c, we distinguish in its rule $r_c = (\texttt{hr}(\widetilde{y}, c(\widetilde{w_c}, \widetilde{z_c})) := S_c)$ the variables $\widetilde{w_c}$ used as input of the analyzed call from the ones $\widetilde{z_c}$ necessary to compute S_c. The resulting bone is then `Return` $(\widetilde{y}, c(\widetilde{t}, \widetilde{z_c}))$, where we replace $\widetilde{w_c}$ with the terms $\widetilde{t}$ being reduced. We know the variables $\widetilde{z_c}$ exist at the program point we are transforming, because they have been extracted from the same hook call in r during the analysis. Similarly, the variables $\widetilde{y}$ of r_c are the same as the environment variables of r by construction, so we can reuse them.

Example 4.3. The call for which we need to create `While2` is of the form (cf. Section 3.2):

```
hook hstmt (s : state, t : stmt) matching t : state =
...
| While1 (s0, e0, e1, t2) ->
...
let (s', v) =  ... in
...
let s'' = hstmt (new While2) (s', t2) in  ...
```

The rule created for `While2` in Example 4.1 is of the form `hstmt(s, While2(s0, t0, e1, t2)) := S`, where $\widetilde{w_c} =$ `(s0,t0)` and $\widetilde{z_c} =$ `(e1,t2)` are used to compute respectively the analyzed call and the rest of the skeleton. Replacing `(s0, t0)` with the arguments of the call `(s', t2)`, the resulting bone is `Return (s, While2 (s', t2, e1, t2))`, and we see that `s`, `e1`, and `t2` are bound at the point we transform.

We change a hook call that can be reconstructed into a branching representing its possible behaviors. The first branch begins by reducing the hook one step further `let` $\widetilde{z'}, z =$ `Hook` h $\widetilde{w'}, w$ `in` ..., storing the results in some fresh variables $\widetilde{z'}, z$.[1] We then reconstruct a configuration using the constructor c_r of the rule being processed. Starting from the initial input $(\widetilde{y}, c_r(\widetilde{x}))$, we apply σ before changing $\widetilde{w'}, w$ by their new values $\widetilde{z'}, z$. The substitution σ is necessary if one of the variables $\widetilde{w'}, w$ is not part of the initial arguments but defined from a previous hook call, as we can see in the `Plus` example below. The second branch covers the case where the term represented by w is a coerced set of values. We extract the content of w into the variables $\widetilde{v}$ of the initial skeleton and continue transforming S, remembering that w is equal to `Ret_h`$(\widetilde{v})$.

Example 4.4. Consider the rule for `Plus`:

```
| Plus (e1, e2) ->
  let (s', v1) = hexpr Recons (s, e1) in
  let (s'', v2) = hexpr Recons (s', e2) in
  let v = add (v1, v2) in
  (s'', v)
```

The first call is turned into two branches, the first one stepping once `(s, e1)` into some fresh variables `(z1, z2)`. The reconstructed configuration is simply the input `(s, Plus (e1, e2))` where `s` and `e1` are replaced by `z1` and `z2`. In the second branch, we extract the content of `e1`, and then transform the rest of the skeleton, remembering that `e1 = Ret_hexpr(s', v1)` in σ.

Transforming the second hook call illustrates why we need σ. As for the first call, the first branch steps once `(s', e2)` into some fresh variables `(z3, z4)` and then reconstructs a configuration. We see that `s'` does not occur in the initial configuration `(s, Plus (e1, e2))`; we therefore apply the substitution to create `(s, Plus (Ret_hexpr(s', v1), e2))`, and now we can turn `(s', e2)` into `(z3, z4)`, resulting in the configuration `(s, Plus (Ret_hexpr(z3, v1), z4))`. The second branch continues transforming the rest of the skeleton, where we no longer need the substitution. In the end, we obtain:

```
hook hexpr (s : state, e : expr) matching e : state * expr =
| Plus (e1, e2) ->
  branch                                    (* |                          *)
    let (z1, z2) = hexpr (s, e1) in         (* | first hook call           *)
```

[1] We individualize the last argument of the hook call because it plays a key role in the second branch, but not in the first one.

```
    (z1, Plus (z2, e2))                          (* |                              *)
  or                                             (* |                              *)
    let (s', v1) = getRet_hexpr (e1) in          (* |                              *)
    branch                                       (*   ||                           *)
      let (z3, z4) = hexpr (s', e2) in           (*   || second hook call          *)
      (s, Plus (Ret_hexpr (z3, v1), z4))         (*   || <- need substitution      *)
    or                                           (*   ||                           *)
      let (s'', v2) = getRet_hexpr (e2) in       (*   ||                           *)
      let v = add (v1, v2) in                    (*      # filter unchanged        *)
      (s, Ret_hexpr (s'', v))                    (*        ## coerced return      *)
    end                                          (*   ||                           *)
  end                                            (* |                              *)
```

The final phase of the transformation is applied only to the rules defining the hooks $\mathtt{h} \in H_0$, as the hooks `getRet_h` added in H_1 have already been built with the right skeletons. If we partition the set of rules R_4 into $R_4 = R'_4 \cup R_{\mathtt{getRet}}$ so that $R_{\mathtt{getRet}}$ contains only the rules defining hooks `getRet_h`, then

$$R_5 = \{\mathtt{h}(\widetilde{y}, c(\widetilde{x})) := \parallel S \parallel_{\epsilon}^{\mathtt{h}(\widetilde{y},c(\widetilde{x})):=S} \mid (\mathtt{h}(\widetilde{y}, c(\widetilde{x})) := S) \in R'_4\} \cup R_{\mathtt{getRet}}$$

In the end, we obtain the skeletal semantics $(T_b, T_p, C_3, F, H_1, R_5, \mathtt{csort}, \mathtt{fsort}, \mathtt{hsort})$ (where `hsort` has been updated) where every $\mathtt{h} \in H_0$ makes a single step of computation.

5 Coq Certification

A complete formalization of the transformation in a proof assistant seems out of reach because of its many intermediary steps. Instead, we follow the easier route of providing an a-posteriori certification alongside the resulting semantics. To this end, we rely on the possibility offered by Necro to export a skeletal semantics into a Coq representation. We automatically generate proof scripts showing that the produced small-step semantics corresponds to the initial big-step one. We explain how to generate the scripts using IMP as an example; the Gitlab repository contains examples of Coq scripts for other languages as well as the script generator.

5.1 Proof Sketch

To state the equivalence theorems between the two semantics, we decorate the concrete interpretation judgment of Section 2.2 with the hook it involves: we write $\widetilde{a} \Downarrow_h \widetilde{b}$ to state that h takes the values $\widetilde{a}$ as input and output the values $\widetilde{b}$.

$$\frac{h(\widetilde{y}, c(\widetilde{x})) := S \in R \qquad \{y \widetilde{\mapsto} a\} + \{x \widetilde{\mapsto} a'\} \vdash S \Downarrow \widetilde{b}}{(\widetilde{a}, c(\widetilde{a'})) \Downarrow_h \widetilde{b}}$$

We write `sshexpr` and `sshstmt` for the hooks of the small-step skeletal semantics resulting from our transformation; the reflexive and transitive closure $\Downarrow_h^*$, with $\mathtt{h} \in \{\mathtt{sshexpr}, \mathtt{sshstmt}\}$, represent sequences of small-step reductions. We recall that the concrete interpretation (Section 2.2) is inherently big-step, as a judgment computes the whole skeleton, so we keep the formal notation $\Downarrow_h$ even for the modified semantics. However, the new hooks are precisely generated such that their concrete (big-step) interpretation corresponds to a standard small-step reduction. The equivalence theorem between the semantics is as follows.

Theorem 5.1. For all `s,s0:state`, `e:expr`, `t:stmt`, and `v:value`,

$$(\mathtt{s},\mathtt{e}) \Downarrow_{\mathtt{hexpr}} (\mathtt{s0},\mathtt{v}) \iff \exists \mathtt{s}',\ (\mathtt{s},\mathtt{e}) \Downarrow^{*}_{\mathtt{sshexpr}} (\mathtt{s}',\mathtt{Ret\_hexpr(s0,v)})$$
$$(\mathtt{s},\mathtt{t}) \Downarrow_{\mathtt{hstmt}} \mathtt{s0} \iff \exists \mathtt{s}',\ (\mathtt{s},\mathtt{t}) \Downarrow^{*}_{\mathtt{sshstmt}} (\mathtt{s}',\mathtt{Ret\_hstmt(s0)})$$

A big-step evaluation is possible if and only if a sequence of small-step reductions can lead to the same result up to coercions.

For the sake of readability, we limit the examples of this section to expressions and deliberately ignore the states. We also write `Ret` for `Ret_hexpr`, and use more intuitive notations for the concrete interpretation of the initial big-step semantics ($\Downarrow_{\mathtt{BS}}$) and of the small-step semantics ($\underset{\mathtt{SS}}{\rightarrow}$).

The interesting direction is to show that a sequence of small-step reductions implies a big-step evaluation. The textbook proof method [14] relies on auxiliary lemmas to split the sequences of reductions in order to recreate big-step derivation trees, such as the following result for `Plus`:

$$\mathtt{Plus}(e_1,e_2) \underset{\mathtt{SS}}{\rightarrow}^{k} v \implies \exists k_1,k_2,v_1,v_2.\ (k=k_1+k_2+1) \wedge (v=v_1+v_2)\ \wedge e_1 \underset{\mathtt{SS}}{\rightarrow}^{k_1} v_1 \wedge e_2 \underset{\mathtt{SS}}{\rightarrow}^{k_2} v_2$$

This technique depends a lot on the semantics of the language, as each lemma must be derived from the constructor skeleton. We instead use a simpler strategy [17] which relies on a concatenation lemma, stating that we can merge a small step into a big step.

$$e \underset{\mathtt{SS}}{\rightarrow} e' \Downarrow_{\mathtt{BS}} v \qquad \mathtt{implies} \qquad e \Downarrow_{\mathtt{BS}} v$$

Such a local result can be verified with language-independent tactics that simply decompose the small-step reduction. We then iterate the lemma to get the desired result.

The downside of this approach is that the big-step and small-step semantics need to be defined on the same constructors—an issue Poulsen and Mosses [17] do not face as the semantics they define does not introduce new constructors. Currently, our big-step semantics is not defined on the newly created constructors, such as `While1` and `While2`. To bridge the gap between the initial big-step (BS) and the small-step (SS) semantics, we also generate an extended big-step semantics (EBS) defined on all constructors. For IMP, the corresponding hooks are written `ehexpr` and `ehstmt`, and the examples use the notation $\Downarrow_{\mathtt{EBS}}$.

We thus manipulate three semantics in this certification. The first is the initial BS semantics written by the user; the second is the EBS semantics generated in the middle of the transformation; the third is the SS semantics, result of the transformation. These semantics are transformed into Coq definitions using the export function of Necro. Necro also provides a Coq definition of the concrete interpretation, which itself depends on interpretations for the base sorts and filters. Our proof script takes as global parameters such interpretations for the initial BS, which are carried to the other two semantics through coercions. This way, the certification is independent from the behavior and implementation of these basic elements.

The proof strategy is therefore to show that BS and EBS are equivalent on initial terms, and then prove that EBS is equivalent to SS on all terms (including the extended ones). In the end, we get that BS is equivalent to SS on initial terms, as stated in Theorem 5.1. The first equivalence is straightforward since EBS has exactly the same rules as BS for the initial constructors. Proving that EBS implies SS relies on the concatenation lemma, as explained above. The opposite direction is done as in the pen-and-paper proof [14], by induction on the EBS derivation. We detail each step and their automation in Coq in the following subsections.

5.2 Initial and Extended Big-Step

The intermediate EBS semantics extends the initial BS one on all constructors. It contains the unchanged, user-defined rules for the initial constructors (e.g. `Plus`), and the rules generated

during the processing phase of Section 4.2.2 for the added constructors; see for instance the rules generated for `While1` and `While2` in Section 3.2. Finally, we add a rule for each return constructor to extract its resulting values, such as $\texttt{Ret}(v) \Downarrow_{\texttt{EBS}} v$. We prefix the sorts of the extended semantics with a letter `e`, writing for instance `estate` for extended states, and we write $|t|$ for the the canonical injection of an initial term t into the extended semantics. The complete EBS generated for IMP can be found in Appendix E.

The first step of the certification checks that EBS and BS agree on the initial terms. It is easy to verify since EBS is a conservative extension of BS: we simply match every behavior—every applied rule—of each big-step semantics with exactly the same one on the other side. The only difficulty is the manipulation of coercions back and forth between initial and extended terms, but a few general lemmas are enough to automate the rewriting.

Theorem 5.2 (BS⇒EBS)**.** For all `s,s0:state`, `e:expr`, `t:stmt`, and `v:value`,

$$\begin{aligned}(\texttt{s},\texttt{e}) \Downarrow_{\texttt{hexpr}} (\texttt{s0},\texttt{v}) &\implies (|\texttt{s}|,|\texttt{e}|) \Downarrow_{\texttt{ehexpr}} (|\texttt{s0}|,|\texttt{v}|)\\ (\texttt{s},\texttt{t}) \Downarrow_{\texttt{hstmt}} \texttt{s0} &\implies (|\texttt{s}|,|\texttt{t}|) \Downarrow_{\texttt{ehstmt}} |\texttt{s0}|\end{aligned}$$

Theorem 5.3 (EBS⇒BS)**.** For all `s:state`, `e:expr`, `t:stmt`, `s0':estate`, and `v':evalue`,

$$\begin{aligned}(|\texttt{s}|,|\texttt{e}|) \Downarrow_{\texttt{ehexpr}} (\texttt{s0}',\texttt{v}') &\implies \exists \texttt{s0},\texttt{v},\ (\texttt{s0}',\texttt{v}') = (|\texttt{s0}|,|\texttt{v}|) \ \wedge\ (\texttt{s},\texttt{e}) \Downarrow_{\texttt{hexpr}} (\texttt{s0},\texttt{v})\\ (|\texttt{s}|,|\texttt{t}|) \Downarrow_{\texttt{ehstmt}} \texttt{s0}' &\implies \exists \texttt{s0},\ \texttt{s0}' = |\texttt{s0}| \ \wedge\ (\texttt{s},\texttt{t}) \Downarrow_{\texttt{hstmt}} \texttt{s0}\end{aligned}$$

This proof step is the only one at the interface between initial and extended terms and as such, the only one using coercions. Henceforth, the results are stated on extended terms (`eexpr` and `eestmt`).

5.3 Small-Step Implies Extended Big-Step

As explained before, the strategy for this direction relies on a concatenation lemma merging a small step and an extended big step together. The proof is done by induction on the small step; in each case, we also need a case analysis on the big-step hypothesis which generates numerous subcases. Fortunately, the proof principle is simple enough that elementary tactics are sufficient for Coq to automatically reconstruct the extended big step in all cases. For instance, if the small step comes from a congruence, i.e.,

$$\texttt{Plus}(e_1,e_2) \underset{\texttt{SS}}{\to} \texttt{Plus}(e_1',e_2) \Downarrow_{\texttt{EBS}} v \text{ from } e_1 \underset{\texttt{SS}}{\to} e_1', e_1' \Downarrow_{\texttt{EBS}} v_1, e_2 \Downarrow_{\texttt{EBS}} v_2, \text{ and } v = v_1 + v_2,$$

we apply the induction hypothesis to get $e_1 \Downarrow_{\texttt{EBS}} v_1$ and reconstruct a big step from our pieces.

To automate this reasoning, we need tactics to automatically apply the induction hypothesis, which are straightforward and language-independent, and tactics to recreate an extended big step from the right hypotheses. Concluding each case can be automated without prior knowledge of the language because we only have to check the result and there is nothing to guess. For `Plus`, we want $\texttt{Plus}(e_1,e_2) \Downarrow_{\texttt{EBS}} v$: we know the term to evaluate, we know the resulting value, and we have the necessary sub-evaluations and filter hypotheses. The checking tactic simply opens the skeleton and verifies that there is indeed a path from the initial term to the resulting one.

Lemma 5.4 (Concatenation)**.** For all `s,s',s0:estate`, `e,e':eexpr`, `t,t':estmt`, `v:evalue`,

$$\begin{aligned}(\texttt{s},\texttt{e}) \Downarrow_{\texttt{sshexpr}} (\texttt{s}',\texttt{e}') \Downarrow_{\texttt{ehexpr}} (\texttt{s0},\texttt{v}) &\implies (\texttt{s},\texttt{e}) \Downarrow_{\texttt{ehexpr}} (\texttt{s0},\texttt{v})\\ (\texttt{s},\texttt{t}) \Downarrow_{\texttt{sshstmt}} (\texttt{s}',\texttt{t}') \Downarrow_{\texttt{ehstmt}} \texttt{s0} &\implies (\texttt{s},\texttt{t}) \Downarrow_{\texttt{ehstmt}} \texttt{s0}\end{aligned}$$

Then, using this concatenation lemma, a simple induction on the reflexive and transitive closure gives us the desired results:

Theorem 5.5 (SS⇒EBS)**.** For all `s,s',s0:estate`, `e:eexpr`, `t:estmt`, and `v:evalue`,

$$(\mathtt{s},\mathtt{e}) \Downarrow^*_{\mathtt{sshexpr}} (\mathtt{s}',\mathtt{Ret\_hexpr(s0,v)}) \implies (\mathtt{s},\mathtt{e}) \Downarrow_{\mathtt{ehexpr}} (\mathtt{s0},\mathtt{v})$$
$$(\mathtt{s},\mathtt{t}) \Downarrow^*_{\mathtt{sshstmt}} (\mathtt{s}',\mathtt{Ret\_hstmt(s0)}) \implies (\mathtt{s},\mathtt{t}) \Downarrow_{\mathtt{ehstmt}} \mathtt{s0}$$

5.4 Extended Big-Step Implies Small-Step

Proving that a big-step evaluation of an extended term corresponds to a small-step sequence of reduction is done by induction on the derivation of the big-step evaluation. In each case, we need to build a complete sequence of small steps from the partial sequences we get from using the induction hypothesis. For instance, decomposing the evaluation $\mathtt{Plus}(e_1,e_2) \Downarrow_{\mathtt{EBS}} v$ produces the hypotheses $e_1 \underset{\mathtt{SS}}{\to}^* \mathtt{Ret}(v_1)$, $e_2 \underset{\mathtt{SS}}{\to}^* \mathtt{Ret}(v_2)$, and $v = v_1 + v_2$, from which we want to show $\mathtt{Plus}(e_1,e_2) \underset{\mathtt{SS}}{\to}^* \mathtt{Ret}(v)$.

As in the previous section, we could let Coq conclude in a bruteforce way, by trying to derive the desired conclusion from the hypotheses, without any knowledge of the semantics of the language. It requires Coq to guess the intermediary configurations; e.g., if $e_1 \underset{\mathtt{SS}}{\to} e_1' \underset{\mathtt{SS}}{\to}^* \mathtt{Ret}(v_1)$, then the final sequence should go through $\mathtt{Plus}(e_1',e_2)$. On complex cases, we may also need to change constructors (e.g. $\mathtt{While1}(..) \underset{\mathtt{SS}}{\to} \mathtt{While2}(..)$). A lot of trial-and-error may be necessary to find these intermediary configurations, notably for cases involving non-determinism where several small-step reductions are possible. Overall, such bruteforce tactics are not efficient enough: it works on small languages such as IMP, but not for more complex languages such as our miniML example.

Instead, we help Coq by generating congruence results about the small-step reduction. With them, we still might need backtracking to find the right combination of lemmas to apply, but all possible small-step reduction cases have been defined so we are not repeatedly losing exploration time to regenerate them in each search. In the case of `Plus`, we need lemmas of the form:

$$e_1 \underset{\mathtt{SS}}{\to}^* e_1' \text{ implies } \mathtt{Plus}(e_1,e_2) \underset{\mathtt{SS}}{\to}^* \mathtt{Plus}(e_1',e_2)$$
$$e_2 \underset{\mathtt{SS}}{\to}^* e_2' \text{ implies } \mathtt{Plus}(\mathtt{Ret}(v_1),e_2) \underset{\mathtt{SS}}{\to}^* \mathtt{Plus}(\mathtt{Ret}(v_1),e_2')$$
$$v = v_1 + v_2 \text{ implies } \mathtt{Plus}(\mathtt{Ret}(v_1),\mathtt{Ret}(v_2)) \underset{\mathtt{SS}}{\to}^* \mathtt{Ret}(v)$$

Combining them lets us prove the desired result. Such congruence results come for free if the reduction is written using small-step inference rules. In our case, we have skeletons and the concrete interpretation, so we need to derive them. It is the only part of the proof script that really depends on the semantics of the language, as we read the skeletons to generate these lemmas.

Each lemma corresponds to a path in a small-step rule. In the case of `Plus` (whose skeleton is detailed in Example 4.4), following the different branches gives us three different paths:

```
- let (z1, z2) = sshexpr (s, e1) in (z1, Plus (z2, e2))
- let (s', v1) = getRet_hexpr (e1) in
  let (z3, z4) = sshexpr (s', e2) in (s, Plus (Ret_hexpr (z3, v1), z4))
- let (s', v1) = getRet_hexpr (e1) in let (s'', v2) = getRet_hexpr (e2) in
  let v = add (v1, v2) in (s, Ret_hexpr (s'', v))
```

For each path, the LetIn definition contains the hypothesis of the lemma, while the final result is the configuration to step towards. If we forget about the state, we see that we obtain exactly the three previous lemmas. They are proved either by unfolding the definitions or doing a straightforward induction; each proof is simple since the structure of the lemma matches a path of the skeleton.

Once this is done, the proof of the main theorem is simply done by induction on the extended big step. In each case, we apply the induction hypothesis on every big-step premise, which gives us several small-step sequences on sub-computations. Then, the congruence lemmas are automatically applied and the results merged together by Coq to create the wanted small-step sequence.

Theorem 5.6 (EBS⇒SS)**.** For all `s,s0:estate`, `e:eexpr`, `t:estmt`, and `v:evalue`,

$$(\mathtt{s},\mathtt{e}) \Downarrow_{\mathtt{ehexpr}} (\mathtt{s0},\mathtt{v}) \implies \exists \mathtt{s}',\ (\mathtt{s},\mathtt{e}) \Downarrow^{*}_{\mathtt{sshexpr}} (\mathtt{s}', \mathtt{Ret\_hexpr}(\mathtt{s0},\mathtt{v}))$$
$$(\mathtt{s},\mathtt{t}) \Downarrow_{\mathtt{ehstmt}} \mathtt{s0} \implies \exists \mathtt{s}',\ (\mathtt{s},\mathtt{t}) \Downarrow^{*}_{\mathtt{sshstmt}} (\mathtt{s}', \mathtt{Ret\_hstmt}(\mathtt{s0}))$$

6 Implementation

The transformation and the self-certification have been implemented in Necro [5], with a number of options (Section 6.1) to tailor the transformation to specific needs. Using the existing Necro tools, we can generate a Coq version of the small-step semantics to prove properties on it—on top of the equivalence with the big-step one already automatically proved as presented in Section 5. We also generate an OCaml interpreter for the language allowing for small-step and big-step reductions (Section 6.2).

6.1 Options and Optimization

The small-step transformation is part of the expanding toolbox of Necro for the manipulation of skeletal semantics. We make the transformation more flexible by providing a few options for it.

6.1.1 Limiting the Transformation

Skeletal semantics can be defined with different levels of precision. For instance, booleans and their basic operations can either be left abstract by considering them a basic sort with filters, or explicitly defined as a program sort with True/False constructors and hooks. In the second case, these hooks can be converted to small-step reduction processes.

However, sometimes we are only interested in the big-picture of the reduction of main expressions and do not wish to reconstruct when computing boolean operations. For these situations, we propose an option to only transform a specific subset of hooks. High-level evaluation processes can thus be turned into a small-step reduction while keeping low-level operations in their big-step form.

On the IMP language example, we could wish to only transform the evaluation of statements. This is akin to considering a small-step operational semantics for statements and a denotational semantics for expressions.

6.1.2 No Reconstruction

Our transformation tries to reuse the user defined constructors as much as possible in order to reduce the number of additional constructors. If need be, an option forces the creation of new

constructors fir each hook call. This would lead to more terms, but also to significantly simpler skeletons as every constructor would focus on a specific hook call. Depending on the purpose of the small-step semantics, it might be an interesting trade-off.

6.1.3 No Additional Steps

The transformation presented above makes additional administrative steps when changing constructor or refocusing, as it is common on paper to reduce the number of inference rules of the language. However, this is not necessary, and an option forces the transformation to keep tail-calls and perform a recursive call after changing constructor. This leads for instance to skeletons mimicking the rules:

$$\frac{s, t_2 \to s', t_2'}{s, \texttt{If}(\texttt{True}, t_2, t_3) \to s', t_2'} \qquad \frac{s, e_1 \to s', e_1'}{s, \texttt{While}(e_1, t_2) \to s, \texttt{While1}(s', e_1', e_1, t_2)}$$

The Coq certification generator is compatible with the two previous options but not this one, as the implemented proof script expects the administrative steps.

6.1.4 Optimization

The difficulty of the transformation is to reconstruct and reuse as much of the initial semantics as possible, as well as simplify the necessary new constructors. To help and clean up the results, we implemented a few optimizations, and further extensions are still possible.

For instance, we implemented a small optimization to reduce the number of unnecessary arguments. When a hook call is the only one to use a state, and we need a new constructor for it, duplicating the state is not necessary. This happens for instance for `While1`, where our implementation outputs:

```
| While (e1, t2) ->
  (s, While1 (e1, e1, t2))
| While1 (e0, e1, t2) ->
  branch
    let (w1, w2) = hexpr (s, e0) in
    (w1, While1 (w2, e1, t2))
  or ... end
```

However, this check is not sufficient for all cases, and `While2` still has 4 arguments as it evaluates `s'` which is not the main state. To recover the usual hand-written semantics, we would need to overwrite `s` with the content of `s'`. We are not certain overwriting states is always preferable, as it might conflict with other potential optimizations such as detecting read-only states (see Section 7).

6.2 Ocaml Interpreter

The main feature of Necro is its ability to create an OCaml interpreter from a skeletal semantics. The interpreter is parameterized by the types and functions representing the base sorts and filters. Once these have been instantiated, the interpreter provides a module containing an evaluation function for each hook of the skeletal semantics. The evaluation follows the approach of the concrete interpretation presented in Section 2.2, recursively calling the evaluation function each time a hook is encountered.

Using the transformation of this paper as an intermediate step, we are able to create an interpreter providing both big-step and small-step implementations of hooks. As terms now

Example Language	Lines of Code			Constructors	
	Big-Step	Small-Step	Coq	Big-Step	Small-Step
Call-by-Value	22	41	100	3	4
CBV, choice	29	48	120	4	5
CBV, fail	42	60	150	5	6
Call-by-Name	28	41	110	3	4
Arithmetic	32	81	160	5	5
IMP	79	144	330	11	13
IMP (w/o reconstruction)	79	168	360	11	21
IMP, write in expr	84	154	350	12	14
IMP, LetIn	85	166	360	12	16
IMP, try/catch	123	192	420	15	17
MiniML	155	299	720	18	28
MiniML (w/o reconstruction)	155	314	740	18	33

Table 1: Size of the Generated Semantics and Proof Scripts

include values, which may be present as subterms of constructors, the small-step interpreter operates on an semantics extending the one used in big-step. We therefore automatically create separate program sorts for extended terms (e.g., `eexpr` and `estmt` for the IMP example), and generate new procedures to bridge the gap between the initial and extended program sorts (e.g., `ext_expr` : `expr` → `eexpr` and `unext_eexpr` : `eexpr` → `expr`). The small-step interpreter requires the same instantiated types and functions as the big-step one, so it provides a small-step semantics at no additional cost.

7 Evaluation

We compare the sizes of the generated small-step semantics and equivalence proof scripts for various languages in Table 1. The examples include variants of the call-by-value (CBV) and call-by-name (CBN) λ-calculus implemented with closures, and evaluated in an environment mapping variables to closures; CBV has also been extended with non-determinism and exceptions. We also consider substitution-based implementations where fresh name generation is handled via a filter. The examples written in an imperative style include a small language roughly corresponding to IMP expressions (Arithmetic), and extensions of IMP with local (IMP, LetIn) or global (IMP, write) state modification, and with exceptions and handlers (IMP, try/catch). Lastly, miniML is an ML-like language, extending the λ-calculus with arithmetic and boolean operations as well as constructs to define algebraic datatypes and perform pattern-matching on them. The generated small-step semantics and proofs for all the examples can be found in the Gitlab repository.

In the certification, about 500 lines of code are completely independent of the input language and contain definitions of skeletons or concrete interpretation. About 450 lines of code are templates which are filled with basic information about the syntax of the input semantics (the names of the hooks, constructors, and filters): these are general definitions, results, and tactics to manipulate concrete interpretation or coercions. They are part of the generated proof script, but are not counted in Table 1, where we lists the sizes of the language-dependent parts of the proof.

We see that the resulting small-step semantics are significantly longer than the initial big-step ones. This is because recursive computations are replaced with case disjunctions using branchings, quickly increasing the number of lines of the rules but not their complexity, as we

can see with IMP. We also observe that the language-dependent part of the Coq proof scripts remains linear in the size of the small-step semantics. Indeed, it is composed mostly of the generated lemmas of the EBS implies SS part of the proof, which themselves depend on the number of different paths in the small-step rules, as explained in Section 5.4.

W.r.t. to the number of constructors, we see the impact of reconstruction for IMP and miniML by comparing the generated small-step semantics with or without reconstruction. While our transformation on IMP only produces the required `While1` and `While2`, a similar transformation that does not reuse the initial semantics would create 10 intermediate constructors. But the effectiveness of reconstruction depends on how the input skeletal semantics is written. First, it depends on the ordering of the bones in the initial rules. The process is more efficient when hook calls are grouped at the beginning of skeletons, since we do not reconstruct after filter calls. For instance, the two following equivalent rules lead to respectively one and two new constructors.

```
hook eval (s : env, t : lambdaterm) matching t : clos =
| App (t1, t2) ->
  let c1 = eval (s, t1) in
  let c2 = eval (s, t2) in
  let (x, t3, s') = getClos (c1) in
  let s'' = extEnv (s', x, c2) in
  eval (s'', t3)
```

```
| App (t1, t2) ->
  let c1 = eval (s, t1) in
  let (x, t3, s') = getClos (c1) in
  let c2 = eval (s, t2) in
  let s'' = extEnv (s', x, c2) in
  eval (s'', t3)
```

Second, our transformation does not identify read-only states and forcefully copies them. For instance, the evaluation of expressions of an IMP language could return only values, as the state is never modified.

```
hook hexpr (s : state, e : expr) matching e : value =
| ...
| Plus (e1, e2) ->
  let v1 = hexpr (s, e1) in
  let v2 = hexpr (s, e2) in
  add (v1, v2)
```

The transformation would compute that the state `s` is used twice and thus create a new constructor with a copy of it to evaluate `e1`. A small-step operational semantics written by hand would rely on the fact that `hexpr` does not modify the state to avoid introducing a new constructor in that case. Such an analysis requires a global understanding of the semantics which goes beyond the local study of hook calls our transformation is based on. This discrepancy occurs in the miniML language, for which we create 10 new constructors (cf. Table 1), when only 9 are strictly necessary.

8 Related Work

While several approaches go from a small-step to a big-step setting by manipulating either inference rules [16, 3] or interpreters [8, 7], the other direction has been less investigated.

Vesely et al. [18] propose an automatic transformation from a big-step to a small-step interpreter. The input interpreter contains functions for small operations (e.g., updating a state) and a single evaluation function `eval` describing the computation of any term of the language. Roughly, the transformation starts with a partial CPS-transform of `eval` to turn recursive calls into continuations, considered as newly created terms. After some manipulations to make `eval` a stepping function, it ends with an inverse CPS-transform recreating a small-step interpreter. As in our work, creating a new constructor for every continuation would be correct but the authors

aim for an output closer to a semantics written by hand. For this, they substitute continuations that can be expressed as initial terms in order to simplify the resulting interpreter.

Vesely et al.'s approach only considers subcomputations as reduction steps, similarly to the option described in Section 6.1.3, as they transform only `eval` calls—corresponding to hook calls in our case—and ignore other simple functions calls—filter calls—or focus changes. The input interpreter, defined in an ad-hoc language, may not be as expressive as skeletal semantics, in particular because only one evaluation function is possible. It is not clear whether their approach scales to several mutually recursive evaluation functions.

An important difference is that their resulting small-step function only recreates a term and not a configuration. This systematically leads to a new constructor C packing a state and a term. It is not necessarily less efficient than our approach, as they do not need new constructors when only state variables are reused. For instance, to reduce a λ-calculus term $\texttt{App}(t_1, t_2)$ with a subreduction $s, t_1 \to s', t_1'$, we would reconstruct a configuration as $s, \texttt{App}(t_1, t_2) \to s, \texttt{App1}(s', t_1', t_2)$ while they would reconstruct a term as $s \vdash \texttt{App}(t_1, t_2) \to \texttt{App}(C(s', t_1'), t_2)$. As a result, it is difficult to compare the outputs of the two approaches based on the number of additional constructors or rules, but the output semantics are very close to usual small-step definitions, with a minimal number of created constructors in both cases.

Finally, the authors of [18] claim to have informal proofs of several parts of their transformation. We do not have a language-independent proof of ours, but we instead can generate an equivalence proof script for any input semantics.

Huizing et al. [9] present a transformation from a big-step to a small-step operational semantics, by manipulating directly inference rules. Small-step configurations are extended with a stack to keep track of the big-step premises that have already been computed. For every non-axiom big-step rule, they create several terms to indicate which premise is under evaluation, and a multitude of small-step rules to either initiate/conclude the big-step rule, change the focused premise, or reduce the premise under consideration. Rules for focusing on a new premise also guess an input state for the subcomputation; coherence is only checked when concluding the big-step rule. Guessing intermediate states, and delaying the unification until the end of the corresponding big-step rule, make the transformation very generic and interesting for languages with complicated control flow. However, the large number of small-step rules and new terms, and the presence of a stack, make the resulting semantics very different from usual SOS definitions.

9 Conclusion

We have presented a fully automatic transformation from a big-step to a small-step skeletal semantics. It first prepares the ground by creating coercions and new constructors for problematic program points, before transforming every skeleton to generate small-step reduction functions. The resulting small-step skeletal semantics can then be given to Necro to generate an OCaml interpreter or a Coq formalization. We exploit the latter feature to automatically certify a-posteriori the correctness of the result of the transformation on any language. Experimenting our method with several languages shows it produces small-step skeletal semantics close to what we would write by hand.

Our work extends the scope of the Necro project, which aims to provide a framework for formalizing and certifying languages. This transformation is a first step towards inter-deriving different semantics styles; in particular, we plan to target semantics in the pretty-big step format [2] as well as abstract machines.

References

[1] Martin Bodin, Philippa Gardner, Thomas Jensen, and Alan Schmitt. Skeletal semantics and their interpretations. *PACMPL*, 3(POPL):44:1–44:31, 2019.

[2] Arthur Charguéraud. Pretty-big-step semantics. In *Proceedings of the 22nd European Symposium on Programming (ESOP 2013)*, pages 41–60. Springer, 2013.

[3] Ştefan Ciobâcă. From small-step semantics to big-step semantics, automatically. In Einar Broch Johnsen and Luigia Petre, editors, *Integrated Formal Methods, 10th International Conference, IFM 2013, Turku, Finland, June 10-14, 2013. Proceedings*, volume 7940 of *Lecture Notes in Computer Science*, pages 347–361. Springer, 2013.

[4] The Coq Development Team. *The Coq Proof Assistant Reference Manual, version 8.11*, January 2020.

[5] Nathanaël Courant, Enzo Crance, and Alan Schmitt. Necro: Animating Skeletons. In *ML 2019*, Berlin, Germany, August 2019.

[6] Olivier Danvy. A rational deconstruction of landin's SECD machine. In Clemens Grelck, Frank Huch, Greg Michaelson, and Philip W. Trinder, editors, *Implementation and Application of Functional Languages, 16th International Workshop, IFL 2004, Lübeck, Germany, September 8-10, 2004, Revised Selected Papers*, volume 3474 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 2004.

[7] Olivier Danvy. From reduction-based to reduction-free normalization. *Electron. Notes Theor. Comput. Sci.*, 124(2):79–100, 2005.

[8] Olivier Danvy, Jacob Johannsen, and Ian Zerny. A walk in the semantic park. In Siau-Cheng Khoo and Jeremy G. Siek, editors, *Proceedings of the 2011 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2011, Austin, TX, USA, January 24-25, 2011*, pages 1–12. ACM, 2011.

[9] Cornelis Huizing, Ron Koymans, and Ruurd Kuiper. A small step for mankind. In Dennis Dams, Ulrich Hannemann, and Martin Steffen, editors, *Concurrency, Compositionality, and Correctness, Essays in Honor of Willem-Paul de Roever*, volume 5930 of *Lecture Notes in Computer Science*, pages 66–73. Springer, 2010.

[10] Gilles Kahn. Natural semantics. In Franz-Josef Brandenburg, Guy Vidal-Naquet, and Martin Wirsing, editors, *STACS 87, 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany, February 19-21, 1987, Proceedings*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer, 1987.

[11] P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.

[12] Xavier Leroy. Formal verification of a realistic compiler. *CACM*, 52(7):107–115, 2009.

[13] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The OCaml system release 4.10*. Inria, feb 2020.

[14] Hanne Riis Nielson and Flemming Nielson. *Semantics with applications - a formal introduction*. Wiley professional computing. Wiley, 1992.

[15] Gordon D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:17–139, 2004.

[16] Casper Bach Poulsen and Peter D. Mosses. Deriving pretty-big-step semantics from small-step semantics. In Zhong Shao, editor, *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, volume 8410 of *Lecture Notes in Computer Science*, pages 270–289. Springer, 2014.

[17] Casper Bach Poulsen and Peter D. Mosses. Flag-based big-step semantics. *J. Log. Algebraic Methods Program.*, 88:174–190, 2017.

[18] Ferdinand Vesely and Kathleen Fisher. One step at a time - A functional derivation of small-step evaluators from big-step counterparts. In Luís Caires, editor, *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings*, volume 11423 of *Lecture Notes in Computer Science*, pages 205–231. Springer, 2019.

[19] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

A Initial IMP Skeletal Semantics

```
type int
type bool
type ident
type state
type value

type expr =
| Not of expr
| Bconst of bool
| Equal of expr * expr
| Iconst of int
| Plus of expr * expr
| Var of ident
type stmt =
| Assign of ident * expr
| If of expr * stmt * stmt
| Seq of stmt * stmt
| Skip
| While of expr * stmt

val add : value * value -> value
val boolToVal : bool -> value
val eq : value * value -> value
val intToVal : int -> value
val isFalse : value -> unit
```

```
val isTrue : value -> unit
val neg : value -> value
val read : ident * state -> value
val write : ident * state * value -> state

hook hexpr (s : state, e : expr) matching e : state * value =
| Iconst (i) ->
  let v = intToVal (i) in
  (s, v)
| Bconst (b) ->
  let v = boolToVal (b) in
  (s, v)
| Var (x) ->
  let v = read (x, s) in
  (s, v)
| Plus (e1, e2) ->
  let (s', v1) = hexpr (s, e1) in
  let (s'', v2) = hexpr (s', e2) in
  let v = add (v1, v2) in
  (s'', v)
| Equal (e1, e2) ->
  let (s', v1) = hexpr (s, e1) in
  let (s'', v2) = hexpr (s', e2) in
  let v = eq (v1, v2) in
  (s'', v)
| Not (e') ->
  let (s', v) = hexpr (s, e') in
  let v' = neg (v) in
  (s', v')

hook hstmt (s : state, t : stmt) matching t : state =
| Skip -> s
| Assign (x, t') ->
  let (s', v) = hexpr (s, t') in
  write (x, s', v)
| Seq (t1, t2) ->
  let s' = hstmt (s, t1) in
  hstmt (s', t2)
| If (e1, t2, t3) ->
  let (s', v) = hexpr (s, e1) in
  branch
    let () = isTrue (v) in
    hstmt (s', t2)
  or
    let () = isFalse (v) in
    hstmt (s', t3)
  end
| While (e1, t2) ->
  let (s', v) = hexpr (s, e1) in
```

```
  branch
    let () = isTrue (v) in
    let s'' = hstmt (s', t2) in
    hstmt (s'', While (e1, t2))
  or
    let () = isFalse (v) in
    s'
  end
```

B After Adding Coercions

```
type int
type bool
type ident
type state
type value

type expr =
| Not of expr
| Bconst of bool
| Equal of expr * expr
| Iconst of int
| Plus of expr * expr
| Var of ident
| Ret_hexpr of state * value
type stmt =
| Assign of ident * expr
| If of expr * stmt * stmt
| Seq of stmt * stmt
| Skip
| While of expr * stmt
| Ret_hstmt of state

val add : value * value -> value
val boolToVal : bool -> value
val eq : value * value -> value
val intToVal : int -> value
val isFalse : value -> unit
val isTrue : value -> unit
val neg : value -> value
val read : ident * state -> value
val write : ident * state * value -> state

hook getRet_hexpr (t : expr) matching t : state * value =
| Ret_hexpr (v1, v2) ->
  (v1, v2)

hook getRet_hstmt (t : stmt) matching t : state =
```

```
| Ret_hstmt v1 ->
  v1

hook hexpr (s : state, e : expr) matching e : state * value =
| Not e' ->
  let (s', v) = hexpr (s, e') in
  let v' = neg (v) in
  (s', v')
| Bconst b ->
  let v = boolToVal (b) in
  (s, v)
| Equal (e1, e2) ->
  let (s', v1) = hexpr (s, e1) in
  let (s'', v2) = hexpr (s', e2) in
  let v = eq (v1, v2) in
  (s'', v)
| Iconst i ->
  let v = intToVal (i) in
  (s, v)
| Plus (e1, e2) ->
  let (s', v1) = hexpr (s, e1) in
  let (s'', v2) = hexpr (s', e2) in
  let v = add (v1, v2) in
  (s'', v)
| Var x ->
  let v = read (x, s) in
  (s, v)

hook hstmt (s : state, t : stmt) matching t : state =
| Assign (x, t') ->
  let (s', v) = hexpr (s, t') in
  write (x, s', v)
| If (e1, t2, t3) ->
  let (s', v) = hexpr (s, e1) in
  branch
    let () = isTrue (v) in
    hstmt (s', t2)
  or
    let () = isFalse (v) in
    hstmt (s', t3)
  end
| Seq (t1, t2) ->
  let s' = hstmt (s, t1) in
  hstmt (s', t2)
| Skip ->
  s
| While (e1, t2) ->
  let (s', v) = hexpr (s, e1) in
  branch
```

```
    let () = isTrue (v) in
    let s'' = hstmt (s', t2) in
    hstmt (s'', (While (e1, t2)))
  or
    let () = isFalse (v) in
    s'
  end
```

C After Creating New Constructors

```
type int
type bool
type ident
type state
type value

type expr =
| Not of expr
| Bconst of bool
| Equal of expr * expr
| Iconst of int
| Plus of expr * expr
| Var of ident
| Ret_hexpr of state * value
type stmt =
| Assign of ident * expr
| If of expr * stmt * stmt
| Seq of stmt * stmt
| Skip
| While of expr * stmt
| While1 of state * expr * expr * stmt
| While2 of state * stmt * expr * stmt
| Ret_hstmt of state

val add : value * value -> value
val boolToVal : bool -> value
val eq : value * value -> value
val intToVal : int -> value
val isFalse : value -> unit
val isTrue : value -> unit
val neg : value -> value
val read : ident * state -> value
val write : ident * state * value -> state

hook getRet_hexpr (t : expr) matching t : state * value =
| Ret_hexpr (v1, v2) ->
  (v1, v2)
```

```
hook getRet_hstmt (t : stmt) matching t : state =
| Ret_hstmt v1 ->
  v1

hook hexpr (s : state, e : expr) matching e : state * value =
| Not e' ->
  let (s', v) = hexpr (s, e') in              (* reconstruction *)
  let v' = neg (v) in
  (s', v')
| Bconst b ->
  let v = boolToVal (b) in
  (s, v)
| Equal (e1, e2) ->
  let (s', v1) = hexpr (s, e1) in             (* reconstruction *)
  let (s'', v2) = hexpr (s', e2) in           (* reconstruction *)
  let v = eq (v1, v2) in
  (s'', v)
| Iconst i ->
  let v = intToVal (i) in
  (s, v)
| Plus (e1, e2) ->
  let (s', v1) = hexpr (s, e1) in             (* reconstruction *)
  let (s'', v2) = hexpr (s', e2) in           (* reconstruction *)
  let v = add (v1, v2) in
  (s'', v)
| Var x ->
  let v = read (x, s) in
  (s, v)

hook hstmt (s : state, t : stmt) matching t : state =
| Assign (x, t') ->
  let (s', v) = hexpr (s, t') in              (* reconstruction *)
  write (x, s', v)
| If (e1, t2, t3) ->
  let (s', v) = hexpr (s, e1) in              (* reconstruction *)
  branch
    let () = isTrue (v) in
    hstmt (s', t2)                            (* tail-hook *)
  or
    let () = isFalse (v) in
    hstmt (s', t3)                            (* tail-hook *)
  end
| Seq (t1, t2) ->
  let s' = hstmt (s, t1) in                   (* reconstruction *)
  hstmt (s', t2)                              (* tail-hook *)
| Skip ->
  s
| While (e1, t2) ->
  let (s', v) = hexpr (s, e1) in              (* new cons: While1 *)
```

```
  branch
    let () = isTrue (v) in
    let s'' = hstmt (s', t2) in              (* new cons: While2 *)
    hstmt (s'', (While (e1, t2)))            (* tail-hook *)
  or
    let () = isFalse (v) in
    s'
  end
| While1 (s0, e0, e1, t2) ->
  let (s', v) = hexpr (s0, e0) in            (* reconstruction *)
  branch
    let () = isTrue (v) in
    let s'' = hstmt (s', t2) in              (* new cons: While2 *)
    hstmt (s'', (While (e1, t2)))            (* tail-hook *)
  or
    let () = isFalse (v) in
    s'
  end
| While2 (s0, t0, e1, t2) ->
  let s'' = hstmt (s0, t0) in                (* reconstruction *)
  hstmt (s'', (While (e1, t2)))              (* tail-hook *)
```

D Final Small-Step Skeletal Semantics

```
type int
type bool
type ident
type state
type value

type expr =
| Not of expr
| Bconst of bool
| Equal of expr * expr
| Iconst of int
| Plus of expr * expr
| Var of ident
| Ret_hexpr of state * value
type stmt =
| Assign of ident * expr
| If of expr * stmt * stmt
| Seq of stmt * stmt
| Skip
| While of expr * stmt
| While1 of state * expr * expr * stmt
| While2 of state * stmt * expr * stmt
| Ret_hstmt of state
```

```
val add : value * value -> value
val boolToVal : bool -> value
val eq : value * value -> value
val intToVal : int -> value
val isFalse : value -> unit
val isTrue : value -> unit
val neg : value -> value
val read : ident * state -> value
val write : ident * state * value -> state

hook getRet_hexpr (t : expr) matching t : state * value =
| Ret_hexpr (v1, v2) ->
  (v1, v2)

hook getRet_hstmt (t : stmt) matching t : state =
| Ret_hstmt v1 ->
  v1

hook hexpr (s : state, e : expr) matching e : state * expr =
| Not e' ->
  branch
    let (z1, z2) = hexpr (s, e') in
    (z1, Not z2)
  or
    let (s', v) = getRet_hexpr (e') in
    let v' = neg (v) in
    (s, Ret_hexpr (s', v'))
  end
| Bconst b ->
  let v = boolToVal (b) in
  (s, Ret_hexpr (s, v))
| Equal (e1, e2) ->
  branch
    let (z1, z2) = hexpr (s, e1) in
    (z1, Equal (z2, e2))
  or
    let (s', v1) = getRet_hexpr (e1) in
    branch
      let (z3, z4) = hexpr (s', e2) in
      (s, Equal (Ret_hexpr (z3, v1), z4))
    or
      let (s'', v2) = getRet_hexpr (e2) in
      let v = eq (v1, v2) in
      (s, Ret_hexpr (s'', v))
    end
  end
| Iconst i ->
  let v = intToVal (i) in
  (s, Ret_hexpr (s, v))
```

```
| Plus (e1, e2) ->
  branch
    let (z1, z2) = hexpr (s, e1) in
    (z1, Plus (z2, e2))
  or
    let (s', v1) = getRet_hexpr (e1) in
    branch
      let (z3, z4) = hexpr (s', e2) in
      (s, Plus (Ret_hexpr (z3, v1), z4))
    or
      let (s'', v2) = getRet_hexpr (e2) in
      let v = add (v1, v2) in
      (s, Ret_hexpr (s'', v))
    end
  end
| Var x ->
  let v = read (x, s) in
  (s, Ret_hexpr (s, v))

hook hstmt (s : state, t : stmt) matching t : state * stmt =
| Assign (x, t') ->
  branch
    let (z1, z2) = hexpr (s, t') in
    (z1, Assign (x, z2))
  or
    let (s', v) = getRet_hexpr (t') in
    let z3 = write (x, s', v) in
    (s, Ret_hstmt z3)
  end
| If (e1, t2, t3) ->
  branch
    let (z1, z2) = hexpr (s, e1) in
    (z1, If (z2, t2, t3))
  or
    let (s', v) = getRet_hexpr (e1) in
    branch
      let () = isTrue (v) in
      (s', t2)
    or
      let () = isFalse (v) in
      (s', t3)
    end
  end
| Seq (t1, t2) ->
  branch
    let (z1, z2) = hstmt (s, t1) in
    (z1, Seq (z2, t2))
  or
    let s' = getRet_hstmt (t1) in
```

```
    (s', t2)
  end
| Skip ->
  (s, Ret_hstmt s)
| While (e1, t2) ->
  (s, (While1 (s, e1, e1, t2)))
| While1 (s0, e0, e1, t2) ->
  branch
    let (z1, z2) = hexpr (s0, e0) in
    (s, While1 (z1, z2, e1, t2))
  or
    let (s', v) = getRet_hexpr (e0) in
    branch
      let () = isTrue (v) in
      (s, (While2 (s', t2, e1, t2)))
    or
      let () = isFalse (v) in
      (s, Ret_hstmt s')
    end
  end
| While2 (s0, t0, e1, t2) ->
  branch
    let (z1, z2) = hstmt (s0, t0) in
    (s, While2 (z1, z2, e1, t2))
  or
    let s'' = getRet_hstmt (t0) in
    (s'', (While (e1, t2)))
  end
```

E Extended Big-Step for Coq

```
type int
type bool
type ident
type state
type value

type expr =
| Not of expr
| Bconst of bool
| Equal of expr * expr
| Iconst of int
| Plus of expr * expr
| Var of ident
| Ret_hexpr of state * value
type stmt =
| Assign of ident * expr
| If of expr * stmt * stmt
```

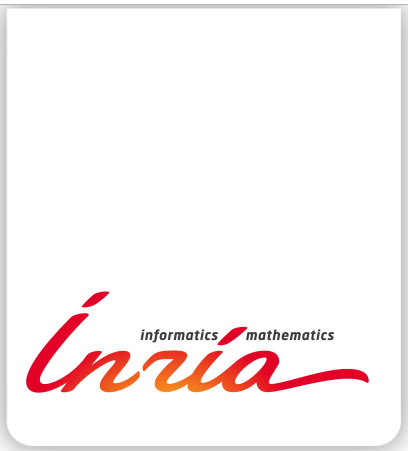
```
| Seq of stmt * stmt
| Skip
| While of expr * stmt
| While1 of state * expr * expr * stmt
| While2 of state * stmt * expr * stmt
| Ret_hstmt of state

val add : value * value -> value
val boolToVal : bool -> value
val eq : value * value -> value
val intToVal : int -> value
val isFalse : value -> unit
val isTrue : value -> unit
val neg : value -> value
val read : ident * state -> value
val write : ident * state * value -> state

hook hexpr (s : state, e : expr) matching e : state * value =
| Not e' ->
  let (s', v) = hexpr (s, e') in
  let v' = neg (v) in
  (s', v')
| Bconst b ->
  let v = boolToVal (b) in
  (s, v)
| Equal (e1, e2) ->
  let (s', v1) = hexpr (s, e1) in
  let (s'', v2) = hexpr (s', e2) in
  let v = eq (v1, v2) in
  (s'', v)
| Iconst i ->
  let v = intToVal (i) in
  (s, v)
| Plus (e1, e2) ->
  let (s', v1) = hexpr (s, e1) in
  let (s'', v2) = hexpr (s', e2) in
  let v = add (v1, v2) in
  (s'', v)
| Var x ->
  let v = read (x, s) in
  (s, v)
| Ret_hexpr (v1, v2) ->
  (v1, v2)

hook hstmt (s : state, t : stmt) matching t : state =
| Assign (x, t') ->
  let (s', v) = hexpr (s, t') in
  write (x, s', v)
| If (e1, t2, t3) ->
```

```
  let (s', v) = hexpr (s, e1) in
  branch
    let () = isTrue (v) in
    hstmt (s', t2)
  or
    let () = isFalse (v) in
    hstmt (s', t3)
  end
| Seq (t1, t2) ->
  let s' = hstmt (s, t1) in
  hstmt (s', t2)
| Skip ->
  s
| While (e1, t2) ->
  let (s', v) = hexpr (s, e1) in
  branch
    let () = isTrue (v) in
    let s'' = hstmt (s', t2) in
    hstmt (s'', (While (e1, t2)))
  or
    let () = isFalse (v) in
    s'
  end
| While1 (s0, e0, e1, t2) ->
  let (s', v) = hexpr (s0, e0) in
  branch
    let () = isTrue (v) in
    let s'' = hstmt (s', t2) in
    hstmt (s'', (While (e1, t2)))
  or
    let () = isFalse (v) in
    s'
  end
| While2 (s0, t0, e1, t2) ->
  let s'' = hstmt (s0, t0) in
  hstmt (s'', (While (e1, t2)))
| Ret_hstmt v1 ->
  v1
```



RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399