Certified Derivation of Small-Step From Big-Step Skeletal Semantics

Guillaume Ambal, Sergueï Lenglet, Alan Schmitt, Camille Noûs

Certified Derivation of Small-Step From Big-Step Skeletal Semantics

Guillaume Ambal
Univ Rennes
France
guillaume.ambal@irisa.fr

Sergueï Lenglet
Université de Lorraine
France
serguei.lenglet@univ-lorraine.fr

Alan Schmitt
Inria
France
alan.schmitt@inria.fr

Camille Noûs
Laboratoire Cogitamus
France
camille.nous@cogitamus.fr

ABSTRACT

We present an automatic translation of a skeletal semantics written in big-step style into an equivalent structural operational semantics. This translation is implemented on top of the Necro tool, which lets us automatically generate an OCaml interpreter for the small step semantics and a Coq mechanization of both semantics. We prove the framework correct in two ways: we provide a paper proof of the core of the transformation, and 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.

CCS CONCEPTS

• **Theory of computation → Operational semantics**.

KEYWORDS

Big-Step, Small-Step, Operational Semantics

ACM Reference Format:
Guillaume Ambal, Sergueï Lenglet, Alan Schmitt, and Camille Noûs. 2022. Certified Derivation of Small-Step From Big-Step Skeletal Semantics. In *24th International Symposium on Principles and Practice of Declarative Programming (PPDP 2022), September 20–22, 2022, Tbilisi, Georgia.* ACM, New York, NY, USA, 48 pages. https://doi.org/10.1145/3551357.3551384

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 [14], a small-step semantics [22], a context-based reduction semantics [28], or even an abstract machine [10, 15]. They all describe identical behaviors of programs, but each 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.

PPDP 2022, September 20–22, 2022, Tbilisi, Georgia

ACM ISBN 978-1-4503-9703-2/22/09…$15.00
https://doi.org/10.1145/3551357.3551384

To capture the essence of operational semantics, Bodin et al. [5] have proposed *skeletal semantics*, a meta-language to describe the specification of a language in a general and systematic way by focusing on the structure of its semantics. Skeletal semantics is a framework generic enough it can express any semantics which can be written with inductive rules. It also benefits from a tool called Necro [9], which can generate an OCaml [17] interpreter or a Coq [8] mechanization of a skeletal semantics given as input. This easy-to-extend tool and the simplicity of skeletal semantics make it a perfect framework for experimenting with inter-semantics transformations.

As an example, consider the skeleton corresponding to a conditional `If(e,s1,s2)`, i.e., `if e then s1 else s2`.

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

In this skeleton, p stands for the recursive evaluation of a subterm. The constructs `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: either compute `isTrue` with b then recursively evaluate `s1`; or compute `isFalse` with b then recursively evaluate `s2`. This piece of syntax becomes a semantics when one provides a meaning for "recursively evaluate", "branch", or the function `isTrue`; skeletal semantics enables the specification of multiple semantics for the same definition just by interpreting differently its building blocks.

A skeletal semantics can be written either in a big-step style, evaluating a term to produce a value, or in a small-step style, i.e., as a partial reduction meant to be iterated. A big-step semantics is closer to an interpreter, which explains why many languages are specified using such a style [1, 19]. However, small-step semantics are better suited to reason about intermediate execution states, e.g., in a debugger, or to prove subject reduction. They also allow for non-terminating or stuck computations, while big-step semantics may express the former only in presence of coinduction [18] and the latter only with explicit error states.

Both styles are therefore desirable, and knowing in advance which one to use when defining a language is not easy. In fact, the semantics of the high-level language of CompCert [16] was initially written in big-step, and was later rewritten as small-step. Being able to automatically derive one from the other would render the

issue moot, and proving the equivalence between the source and target semantics would ensure the correctness of the derivation. Equivalence proofs between big-step and small-step semantics are well-known [18, 21, 24] but repetitive for large languages: they can greatly benefit from additional scrutiny and automation, such as the one provided by theorem provers. While the transformation from small-step to big-step has been significantly studied [7, 11, 12, 23], the opposite direction has been less considered [13, 27], and none of these transformations comes with formal guarantees on their correctness.

In this paper, we propose an automatic translation of a big-step skeletal semantics into a small-step one, with no restriction on the input language. A first approach is to 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`.

Such an approach produces a lot of new constructors, which is not a problem if the resulting semantics is not meant to be manipulated by humans, e.g., if it is only produced for a debugger, where the internal states matter more than the constructors themselves. However, the main challenge of our work is to also generate a minimal, human-readable small-step semantics—more precisely, a structural operational semantics (SOS) [22]—close to what one would write by hand, and to do so independently of the input language. We achieve such a goal by reusing the constructors of the initial big-step semantics as much as possible: in the previous example, we can reuse `If` instead of creating `If2`. Reusing constructors is not always possible: for example, `(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 in such cases only.

Our contributions are: an automatic way of generating small-step skeletal semantics, with or without constructors reuse, from a big-step skeletal semantics of any language, from which we derive OCaml interpreters using Necro. Without reuse, we prove on paper the transformation independently from the input language, by showing that a finite or infinite sequence of small-step reductions is equivalent to an inductive or coinductive big-step evaluation. For the more complex transformation with reuse, we produce for any input language a certificate checkable in Coq of the equivalence in the finite case. We evaluate the approach on various imperative and functional languages with constructs such as pattern matching or exceptions.

The paper is structured as follows. In Section 2, we give a detailed example of skeletal semantics and of its transformation, using a simple imperative language. We formalize skeletal semantics in Section 3 and the transformation in Section 4. Section 5 presents the correspondence results without or with reuse, and shows how to generate a Coq certificate of the equivalence in the latter case. We experiment on several languages in Section 6, and discuss related work in Section 7. The appendix contains the full example introduced in Section 2, as well as the pen-and-paper proof. An implementation is available online [4] with examples of generated semantics, interpreters, and Coq certificates.

2 OVERVIEW ON AN EXAMPLE

2.1 IMP in 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. A skeletal semantics is composed of types, filters, procedures, and rules. We distinguish between *base* and *program* types. Base types are left unspecified and correspond to the base elements of the language, like environments or identifiers for variables. Program types are defined with a list of constructors, each having a precise typing.

Example 1. We write examples using the Necro [9] syntax, and use as a running example a subset of an imperative language called IMP, whose complete definition can be found in Appendix A.1. Types are defined with the keyword `type`; base types (`int`, `bool`, `ident`, `state`, and `value`) are only declared, while program types (`exp` and `stm`) are declared alongside the signature of their constructors, like an algebraic datatype definition in OCaml.

```
type int                    type ident
type bool                   type state
type exp =                  type value
| Iconst of int             type stm =
| Bconst of bool            | Assign of ident * exp
| Var of ident              | Seq of stm * stm
| Plus of exp * exp         | While of exp * stm
```

Types `int` and `bool` represent integers and booleans, collected under the type `value`. The type `ident` represents identifiers for the variables of the language and `state` represents environments mapping variables to values. The two program types 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 types. The only way we can manipulate them is through typed unspecified partial functions called *filters*, which represent the basic operations of the language. For instance, the final representation of `state` does not matter as long as we define read and write operations as filters. These unspecified functions can be seen as the atomic operations of the language.

Example 2. Filters are declared with the keyword `val`, and 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 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
```

Procedures, declared with the `hook` keyword for historical reasons, correspond to the evaluation functions we want to define. Calls to procedures, or *proc calls*, typically represent the recursive

```
hook hexp (s: state, e: exp)
  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 (s1,v1) = hexp(s,e1) in
  let (s2,v2) = hexp(s1,e2) in
  let v = add (v1,v2) in
  (s2,v)
```

```
hook hstm (s: state, t: stm)
  matching t: state =
| Assign (x,e) ->
  let (s1,v) = hexp(s,e) in
  write (x,s1,v)
| Seq (t1,t2) ->
  let s1 = hstm (s,t1) in
  hstm (s1,t2)
| While (e1,t2) ->
  let (s1,v) = hexp(s,e1) in
  branch
    let () = isTrue (v) in
    let s2 = hstm(s1,t2) in
    hstm (s2, While (e1,t2))
  or
    let () = isFalse (v) in
    s1
  end
```

Figure 1: Procedures in IMP

evaluation of subterms, they are thus the point where we want to interrupt the execution when switching to small step. A procedure operates on a term of program type using pattern matching. The behavior of the procedure on a constructor is defined with a *skeleton*. A skeleton represents the semantic behavior of a reduction. It is a sequence of *skeleton elements*, or *skelements*, linked via a LetIn structure. A skelement is either a function call (filter or procedure), a return of values, or a branching corresponding to a non-deterministic choice over several possible skeletons.

Example 3. We define the procedures `hexp` and `hstm` for the evaluation of expressions and statements in Figure 1; the matched term is declared with the keyword `matching`. A branching is written `branch .. (or ..)* end`, while the other skelements (filter or procedure call, return) are not preceded by keywords, as we can easily tell them apart.

The rule for `While` shows how branching works. We evaluate the first term to get a value v; in most programming languages, we would then branch depending on v. Here, instead, we encode this behavior with the non-deterministic branching 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 `hexp` 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.

We now informally describe the transformation of such a semantics into small step with as much constructor reuse as possible. We first add new term constructors (Section 2.2) for the cases where reuse is impossible, and we then transform the skeletons themselves (Section 2.3). Appendix A contains the results of each transformation phase on the complete version of IMP. We also give the semantics without constructor reuse for comparison in Appendix A.6.

2.2 New Constructors

We first determine which new constructors are required in order to produce a small-step semantics. Most reduction rules use 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 $e_1 \longrightarrow e_1'$, 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 reuse the same constructor after a step. The different problematic situations are formally detailed in Section 4.1; 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, we analyze each proc call to determine in which case it falls in. If it is a tail-call, i.e., a final procedure call forwarding its return values, there is nothing to do. If the terms it evaluates are only used once, then we reuse the same constructor. In contrast, if some of the evaluated terms are used elsewhere, then we need to create an additional constructor. In that case, the new constructor mirrors the situation at the program point and carries two copies of the terms evaluated several times. We also extend the corresponding procedure with a new reduction rule for the created constructor, which corresponds to the remainder of the initial skeleton rooted at the analyzed proc call.

We illustrate the analysis on several IMP constructors. We can reuse `Plus` after each proc call, as the arguments of the calls—respectively `s`, `e1` and `s1`, `e2`—are not needed in the rest of the skeleton (cf. Figure 1). For the same reason, we can reuse `Seq` after its first proc call; its second one is a tail-call, so there is no need to worry about reuse.

```
hook hstm (s: state, t: stm) matching t: state =
| Seq (t1, t2) ->
  let s1 = hstm (s, t1) in        (* reuse *)
  hstm (s1, t2)                   (* tail-call *)
```

The analysis gets more interesting for `While`:

```
| While (e1,t2) ->
  let (s1,v) = hexp (s,e1) in     (* new constr: While1 *)
  branch
    let () = isTrue (v) in
    let s2 = hstm (s1,t2) in      (* new constr: While2 *)
    hstm (s2, While (e1,t2))      (* tail-call *)
  or
    let () = isFalse (v) in
    s1
  end
```

The third proc call is a tail-call, as it is the final instruction of one of the reduction paths. When analyzing the first one, we see that `e1` is needed later, thus we cannot reuse `While` here. Similarly, we

cannot reuse the constructor for the second proc call since `t2` is needed in the tail-call.

For each of these two calls, we need to create new constructors 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 proc call, namely `s`, `e1` for the first one, and `s1`, `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 `s1` 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 call—are duplicated. In the end, we extend the `stm` type with the following constructors:

```
| While1 of state * exp * exp * stm
| While2 of state * stm * exp * stm
```

We also add a new rule for each of them. The new skeleton consists in resuming the computation from the corresponding analyzed proc call and 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 skelements. We do not modify the rules for `While` nor the other constructors at this stage.

```
hook hstm (s: state, t: stm) matching t: state =
| ...
| While1 (s0,e0,e1,t2) ->
  let (s1,v) = hexp (s0,e0) in     (* reuse *)
  branch
    let () = isTrue (v) in
    let s2 = hstm (s1,t2) in       (* new constr: While2 *)
    hstm (s2, While (e1,t2))       (* tail-call *)
  or
    let () = isFalse (v) in
    s1
  end
| While2 (s0,t0,e1,t2) ->
  let s2 = hstm (s0,t0) in         (* reuse *)
  hstm (s2, While (e1,t2))         (* tail-call *)
```

The added rules are used next for the last phase of the transformation. The new rules do not need to be analyzed themselves as they are duplicates of analyzed skeletons.

2.3 Make the Skeletons Small-Step

Previous phases set the stage for the main transformation, but the semantics is still big-step, since procedures fully compute their arguments. The last phase of the transformation makes procedures behave in a small-step way.

A small-step reduction transforms a term into another term of the same program type, whereas procedures return values. We thus need to consider these values as terms of the corresponding input type. In our example, we extend the types `exp` and `stm` with constructors corresponding to the return types of each procedure, respectively `Ret_hexp` of type `(state, value) → exp` for `hexp`, and `Ret_hstm` of type `state → stm` for `hstm`. We also define procedures to unpack coerced values, e.g., for expressions:

```
hook getRet_hexp (e: exp) matching e: state * value =
| Ret_hexp (v1, v2) -> (v1, v2)
```

A similar procedure is defined for statements. These procedures are only defined for the corresponding newly created constructors, as trying to unpack the value of a term not fully reduced should fail.

We also need to change the output types of every user-defined procedure to make them match the input ones. The signature of the procedures become:

```
hook hexp (s: state, e: exp) matching e: state * exp = ...
hook hstm (s: state, t: stm) matching t: state * stm = ...
```

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

We then update the skeletons themselves. We recall that skeletons are sequences of operations (skelements), which are mostly composed of filter and proc calls. We consider that only proc calls correspond to reduction steps and should be transformed. The reason is that filters represent atomic operations that are not meant to be interrupted, while proc calls often correspond to the evaluation of a subterm. We distinguish four cases; for each of them, we illustrate the resulting skeleton with informal SOS rules to improve readability. Note that, in each example, the variable `s` is the state coming from the definition of the procedure.

1. If the last element of a skeleton is not a proc call, we need to use coercions to match the new signature. For type checking, we also have to return the other arguments of the procedure, even if they are not of any use. E.g., the rule for `Iconst` returns `(s, v)` in the initial big-step skeleton (cf. Figure 1). Using a coercion, we turn it into `Ret_hexp (s,v)`; we cannot return this term only, as the output type of `hexp` is `(state * exp)`, so we also return a useless copy of `s`.

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

$$\frac{}{s, i \longrightarrow s, (s, \texttt{intToVal}(i))}$$

2. The most interesting case is when we can reuse the constructor. It means 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 take a reduction step and reconstruct; or the subterm has been fully evaluated, and we extract its value and continue the reduction according to the rest of the skeleton. We distinguish the two behaviors using branches. For instance, transforming the first proc call of the `Plus` constructor produces a rule structured as follows.

```
| Plus (e1, e2) ->
  branch
    let (s',e1') = hexp (s, e1) in
    (s', Plus (e1',e2))
  or
    let (s1,v1) = getRet_hexp (e1) in
    ...
  end
```

$$\frac{s, e_1 \longrightarrow s', e_1'}{s, e_1 + e_2 \longrightarrow s', e_1' + e_2} \qquad \frac{e_1 = (s_1, v_1) \qquad \ldots}{s, e_1 + e_2 \longrightarrow \ldots}$$

where the dots stands for the transformation of the second proc call (see Example 6). In the first branch, we reconstruct as `(s', Plus (e1',e2))`, overwriting `s` and `e1` with the new terms resulting from the step. In the second branch, we extract the value using the procedure dedicated to `Ret_hexp`. Even though we use

a branching, the reduction is deterministic, as `getRet_hexp` and `hexp` are restricted to respectively coerced values and terms that are not values.

3. For a proc call for which we created a constructor, we need to change to the new constructor after a reduction step. To simplify the semantics, we consider the change to be a step by itself; the next small step can then reduce the proc call. For instance, the reduction rule for `While` becomes:

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

$$\frac{}{s, \texttt{While}(e_1, t_2) \longrightarrow s, \texttt{While1}(s, e_1, e_1, t_2)}$$

We duplicate `s` and `e1` and return the new configuration. Copying `s` is not required when writing the semantics by hand but comes from the analysis of Section 2.2. Such useless duplication is not uncommon in systematic derivations of semantics, as discussed in Section 6. Calling the `hstm` procedure again would then execute 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 (s0',e0') = hexp (s0,e0) in
    (s, While1 (s0',e0',e1,t2))
  or
    let (s1,v) = getRet_hexp (e0) in
    branch
      let () = isTrue (v) in
      (s, While2 (s1,t2,e1,t2))
    or
      let () = isFalse (v) in
      (s, Ret_hstm s1)
    end
  end
```

$$\frac{s_0, e_0 \longrightarrow s_0', e_0'}{s, \texttt{While1}(s_0, e_0, e_1, t_2) \longrightarrow s, \texttt{While1}(s_0', e_0', e_1, t_2)}$$

$$\frac{e_0 = (s_1, v) \qquad \texttt{isTrue}(v)}{s, \texttt{While1}(s_0, e_0, e_1, t_2) \longrightarrow s, \texttt{While2}(s_1, t_2, e_1, t_2)}$$

$$\frac{e_0 = (s_1, v) \qquad \texttt{isFalse}(v)}{s, \texttt{While1}(s_0, e_0, e_1, t_2) \longrightarrow s, s_1}$$

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, and that the added constructors were sufficient.

4. Finally, we also cut tail-calls to simplify the semantics. This creates administrative steps like $s, \texttt{If}(\texttt{True}, t_1, t_2) \longrightarrow s, t_1$ where no subcomputation takes place, which is closer to usual pen-and-paper definitions. For example, the second proc call of the `Seq` constructor becomes a return—the first one is transformed as previously.

```
| Seq (t1, t2) ->
  branch
    let (s',t1') = hstm (s,t1) in
    (s', Seq (t1',t2))
  or
    let s1 = getRet_hstm (t1) in
    (s1, t2)
  end
```

$$\frac{s, t_1 \longrightarrow s', t_1'}{s, t_1; t_2 \longrightarrow s', t_1'; t_2} \qquad \frac{t_1 = s_1}{s, t_1; t_2 \longrightarrow s_1, t_2}$$

This final phase produces a small-step skeletal semantics where each proc 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 A.4, and we prove the equivalence between the two semantics in Sections 5.

3 SKELETAL SEMANTICS

We present the formal definition of skeletal semantics we use to describe the transformation in Section 4 and state the equivalence results in Sections 5. It mostly differs from the original one [5] by the presence of the LetIn structure and the possibility of defining several procedures, which we believe makes writing the skeletal semantics of a given language significantly easier.

3.1 Syntax and Types

We write $\widetilde{a}$ for a (possibly empty) tuple $(a_1, \ldots, a_n)$, and $\|\widetilde{a}\|$ for its size—here n. We also write $\widetilde{a}, b$ or $\widetilde{a}, \widetilde{b}$ for its extension on the right, and $a_i \in \widetilde{a}$ to state that a_i belongs to $\widetilde{a}$. Given a function or relation f, we write $\widetilde{f(a)}$ for $(f(a_1), \ldots, f(a_n))$ if $\|\widetilde{a}\| = n$, and similarly $\widetilde{g(a,b)}$ for $(g(a_1, b_1), \ldots, g(a_n, b_n))$ if $\|\widetilde{a}\| = \|\widetilde{b}\| = n$.

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

$$t ::= v \mid c(\widetilde{t}) \qquad S ::= \texttt{let}\ \widetilde{v} = K\ \texttt{in}\ S \mid K$$

$$K ::= \texttt{Filter}\ f\ (\widetilde{t}) \mid \texttt{Proc}\ p\ (\widetilde{t}, t) \mid \texttt{Return}\ (\widetilde{t}) \mid \texttt{Branch}\ (\widetilde{S})$$

The skeletal semantics of a language consists of: a set T of types ranged over by s, which includes program types T_g ranged over by s_g; a set C of constructors, with a typing function $\texttt{ctype} : C \to \widetilde{T} \times T_g$; a set F of filters, with $\texttt{ftype} : F \to \widetilde{T} \times \widetilde{T}$; a set P of procedures, with $\texttt{ptype} : P \to (\widetilde{T} \times T_g) \times \widetilde{T}$; a set R of rules $p(\widetilde{y}, c(\widetilde{x})) := S$, assuming $\texttt{ptype}(p) = ((\widetilde{s}, s_g), \widetilde{s'})$, $\texttt{ctype}(c) = (\widetilde{s''}, s_g)$, $\|\widetilde{y}\| = \|\widetilde{s}\|$, and $\|\widetilde{x}\| = \|\widetilde{s''}\|$.

The typing of constructors restricts their output to a term of program type, while filters may produce terms of any type. The input of a procedure $\widetilde{T} \times T_g$ is composed of auxiliary terms of type $\widetilde{T}$ and of the term being evaluated of program type T_g, corresponding to the keyword `matching` in Figure 1. We define projections $\texttt{ptype}_\texttt{g}$ and $\texttt{ptype}_\texttt{out}$: if $\texttt{ptype}(p) = ((\widetilde{s}, s_g), \widetilde{s'})$, then $\texttt{ptype}_\texttt{g}(p) = s_g$ and $\texttt{ptype}_\texttt{out}(p) = \widetilde{s'}$.

In the concrete syntax of Section 2.1, procedures collect behavior for all constructors, whereas we split them here. Formally, a *rule* $p(\widetilde{y}, c(\widetilde{x})) := S$ defines the behavior of p 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

$$\frac{\Sigma(\widetilde{t}) \Downarrow_p \widetilde{b}}{\Sigma \vdash \texttt{Proc}\ p\ \widetilde{t} \Downarrow \widetilde{b}} \qquad \frac{\mathcal{R}_f(\widetilde{\Sigma(t)}) \Downarrow \widetilde{b}}{\Sigma \vdash \texttt{Filter}\ f\ \widetilde{t} \Downarrow \widetilde{b}} \qquad \frac{\widetilde{\Sigma(t)} = \widetilde{b}}{\Sigma \vdash \texttt{Return}\ \widetilde{t} \Downarrow \widetilde{b}}$$

$$\frac{S_i \in \widetilde{S} \qquad \Sigma \vdash S_i \Downarrow \widetilde{b}}{\Sigma \vdash \texttt{Branch}\ \widetilde{S} \Downarrow \widetilde{b}} \qquad \frac{\Sigma \vdash K \Downarrow \widetilde{a} \qquad \Sigma + \widetilde{\{v \mapsto a\}} \vdash S \Downarrow \widetilde{b}}{\Sigma \vdash \texttt{let}\ \widetilde{v} = K\ \texttt{in}\ S \Downarrow \widetilde{b}}$$

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

Figure 2: Inductive Interpretation

$$\frac{\Sigma(\widetilde{t}) \Uparrow_p}{\Sigma \vdash \texttt{Proc}\ p\ \widetilde{t} \Uparrow}$$

$$\frac{p(\widetilde{y}, c(\widetilde{x})) := S \in R \qquad \widetilde{\{y \mapsto a\}} + \widetilde{\{x \mapsto a'\}} \vdash S \Uparrow}{(\widetilde{a}, c(\widetilde{a'})) \Uparrow_p}$$

$$\frac{S_i \in \widetilde{S} \qquad \Sigma \vdash S_i \Uparrow}{\Sigma \vdash \texttt{Branch}\ \widetilde{S} \Uparrow} \qquad \frac{\Sigma \vdash K \Uparrow}{\Sigma \vdash \texttt{let}\ \widetilde{v} = K\ \texttt{in}\ S \Uparrow}$$

$$\frac{\Sigma \vdash K \Downarrow \widetilde{a} \qquad \Sigma + \widetilde{\{v \mapsto a\}} \vdash S \Uparrow}{\Sigma \vdash \texttt{let}\ \widetilde{v} = K\ \texttt{in}\ S \Uparrow}$$

Figure 3: Coinductive Interpretation

pairwise distinct, and to contain the free variables of S. We suppose that at most one rule handling c for p exists in R. The matching does not have to be exhaustive: a procedure p without a rule for c simply cannot reduce terms with c as head constructor.

3.2 Interpretations

The dynamic of a skeletal semantics is given by an *interpretation* of the rules defining its procedures [5], which can be inductive or coinductive. It supposes an instantiation of the base types and filters. For each base type, we assume given a set representing its values. The values of program types are inductively built from its constructors. For each filter $f \in F$, we assume a relation $\mathcal{R}_f$ between its input and output values, and write $\mathcal{R}_f(\widetilde{a}) \Downarrow \widetilde{b}$ when the interpretation of f relates the values $\widetilde{a}$ to $\widetilde{b}$.

For example, instantiating `bool` with Booleans $\{\top; bot\}$, we could interpret `isTrue` to only accept $\top$ so that $\mathcal{R}_{\texttt{isTrue}}(\top) \Downarrow ()$ and similarly for `isFalse`. We could also allow conditional branching on an integer by extending `isTrue` and `isFalse` to `int`, so that $\mathcal{R}_{\texttt{isFalse}}(0) \Downarrow ()$ and $\forall i \neq 0$, $\mathcal{R}_{\texttt{isTrue}}(i) \Downarrow ()$. The strength of skeletal semantics is that this choice only involves filters: we do not have to change anything in the definitions or interpretations of the procedures.

We write Σ for an environment mapping a finite set of variables in $\mathcal{V}$ (its *domain*) to values. We write $\Sigma(v)$ for the value related to v in Σ, and we write $\Sigma(t)$ so that $\Sigma(c(\widetilde{t})) = c(\widetilde{\Sigma(t)})$. The environment $\{v \mapsto b\}$ maps a single variable v to 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 inductive interpretation $\Sigma \vdash S \Downarrow \widetilde{b}$, defined in Figure 2, expresses that S outputs the values $\widetilde{b}$ under Σ; it assumes that the free variables of S are in the domain of Σ. The rules are self-explanatory; e.g., evaluating a proc call consists in evaluating its arguments, which itself consists in finding the skeleton of the appropriate rule and evaluating it. The interpretation is inherently big-step, as a judgment $\Sigma \vdash S \Downarrow \widetilde{b}$ computes the final value returned by S. It is also non-deterministic because of branchings, which return the outcome of one of their successful branches, and filter calls, as $\mathcal{R}_f$ may relate several results to the same input. Figure 2 also uses an auxiliary judgment $\widetilde{a} \Downarrow_p \widetilde{b}$ saying that the procedure p taking $\widetilde{a}$ as input can output $\widetilde{b}$.

To express infinite computations, we define a coinductive semantics of skeletons. The rules are given in Figure 3, where the double lines mean the rules must be interpreted coinductively.

There is no rule for returns and filters, as they cannot diverge. A proc call diverges if the evaluation of the corresponding skeleton diverges. A branching diverges if one of the branches diverges. A LetIn structure diverges either if its first or its second skelement diverges. The coinductive interpretation is used in Section 5.1 to prove that the transformation of Section 4 preserves infinite behaviors.

4 TRANSFORMATION TO SMALL STEP

4.1 New Constructors

Given a big-step semantics, we discriminate its proc calls into three categories: either `Tail` for a tail-call, `Reuse` when constructor reuse is possible, or `New` c when a new constructor c is needed. Formally, we consider an intermediate skeletal semantics where proc calls are annotated: `Proc` $a\ p\ (\widetilde{t}, t)$, with $a ::= \texttt{New}\, c \mid \texttt{Tail} \mid \texttt{Reuse}$. We proceed in two steps: we first analyze and annotate the proc calls before creating the new constructors.

A simple way to make the semantics small-step would be to introduce a new constructor for each call. While it is safe to do so, the resulting semantics can be optimized by reusing constructors. We aim to reuse them as much as possible to obtain a semantics close to a usual SOS. We identify three cases where reuse is not possible: after a filter call, in the continuation of a branching, or if an argument of a proc call is not a variable or is needed several times in the skeleton.

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

Next, we need to account for branching non-determinism. In $\texttt{let}\ \widetilde{v} = \texttt{Branch}(\texttt{Proc}\ p_1\ \widetilde{t_1},\ \texttt{Proc}\ p_2\ \widetilde{t_2})\ \texttt{in}\ \texttt{Proc}\ p\ \widetilde{t}$, two different reduction paths lead to the last call. The premise of constructor reuse is that reevaluating the skeleton from the start should lead to

$$[\texttt{Proc}\ p\ \widetilde{t}]^{pr,V}_{L,b} \triangleq \texttt{Proc Tail}\ p\ \widetilde{t} \qquad \text{if } L = \top, pr = p$$

$$[\texttt{Proc}\ p\ \widetilde{w}]^{pr,V}_{L,b} \triangleq \texttt{Proc Reuse}\ p\ \widetilde{w} \qquad \text{if } b = \top, \widetilde{w} \in V$$

$$[\texttt{Proc}\ p\ \widetilde{t}]^{pr,V}_{L,b} \triangleq \texttt{Proc (New}\,c\texttt{)}\ p\ \widetilde{t} \qquad \text{else, } c \text{ fresh}$$

$$[\texttt{let}\ \widetilde{v} = K\ \texttt{in}\ S]^{pr,V}_{L,b} \triangleq$$
$$\texttt{let}\ \widetilde{v} = [K]^{pr,V}_{bot,b}\ \texttt{in}\ [S]^{pr,V}_{L,bot} \qquad \text{if } K \neq \texttt{Proc}\ p\ \widetilde{t}$$

$$[\texttt{let}\ \widetilde{v} = \texttt{Proc}\ p\ \widetilde{w}\ \texttt{in}\ S]^{pr,V}_{L,b} \triangleq$$
$$\texttt{let}\ \widetilde{v} = \texttt{Proc Reuse}\ p\ \widetilde{w}\ \texttt{in}\ [S]^{pr,V}_{L,b} \qquad \text{if } b = \top, \widetilde{w} \in V$$

$$[\texttt{let}\ \widetilde{v} = \texttt{Proc}\ p\ \widetilde{t}\ \texttt{in}\ S]^{pr,V}_{L,b} \triangleq$$
$$\texttt{let}\ \widetilde{v} = \texttt{Proc (New}\,c\texttt{)}\ p\ \widetilde{t}\ \texttt{in}\ [S]^{pr,V}_{L,b} \qquad \text{else, } c \text{ fresh}$$

Figure 4: Main Cases of the Proc Calls Analysis

the same evaluation context. It is not the case in such a situation, as we may take a different path and reach the last proc call with different values bound to the variables $\widetilde{v}$. As such, we do not reuse constructor if the analyzed proc call is in the continuation of a branching.

Lastly, as illustrated in the overview, reuse 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 call are not variables, or if these variables are used again in the skeleton.

Formally, the annotation process of a given skeleton S is noted $[S]^{pr,V}_{L,b}$, where L is a boolean indicating if we are at the toplevel of the LetIn structure of the main skeleton, used to detect tail-calls; b is a boolean indicating if constructor reuse is possible at this position, 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; pr is the name of the procedure corresponding to the rule being analyzed, also used to detect tail-calls, as detailed below.

The main part of the analysis is defined in Figure 4. Given a rule $p(\widetilde{y}, c(\widetilde{x})) := S$, we compute the set of variables that are used exactly once in S, and we fix V as this set, and pr as p. These parameters are constants while L and b are initialized at $\top$ and can change through the analysis: b is set to *bot* after a filter call or a branching, while L is set to *bot* in the first part of a LetIn structure.

A final proc call is considered a tail-call if and only if it is situated at the toplevel of the main skeleton ($L = \top$) and if the procedure being called is the one being analyzed ($p = pr$). The second condition prevents typing issues. In the initial big-step semantics, a rule from a procedure `h1` can make a final call to a procedure `h2` if they have the same return types, like in the following dummy example.

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

However, a small-step procedure must have the same return type as its input type (see Sections 2.3 and 4.3). The new output type of `h2` (`exp`) does not match the new one of `h1` (`explist`): the call to `h2` has to be modified to make the types match and cannot be a tail-call.

A proc call can only reuse its constructor if it is not after 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 proc call can be annotated with either `Tail` or `Reuse`, we choose to give precedence to the former, because tail-calls are more specific and lead to simpler skeletal semantics at the end of the transformation. Proc calls that cannot be annotated `Tail` or `Reuse` are instead associated with a fresh constructor name created on the fly. We apply the analysis to every skeleton in the semantics.

After the analysis, we process every proc call annotated `New`c to generate the rule of c. The rule contains what is left to compute at the program point of the proc call, i.e., its continuation, that we represent as a context of the form $E ::= [\cdot] \mid \texttt{let}\ \widetilde{x} = [\cdot]\ \texttt{in}\ S$.

Given a rule $r = (\texttt{pr}(\widetilde{y}, c_r(\widetilde{x})) := S_r)$ we turn each call of S_r of the form $E[\texttt{Proc (New}\,c\texttt{)}\ p\ \widetilde{t}]$ into a new rule $\{\texttt{pr}(\widetilde{y}, c(\widetilde{w}, \widetilde{z})) := E[\texttt{Proc Reuse}\ p\ \widetilde{w}]\}$, where $\widetilde{w}$ are fresh variables representing the terms $\widetilde{t}$, and $\widetilde{z}$ are the variables needed to evaluate E. The proc call p is annotated with `Reuse` in the rule as the variables have been defined so that there is no overlap in their uses. The formal definition of this generation phase is given in Appendix B.3; we illustrate it on the following example.

Example 4. We apply the generation process to the second proc call in the rule for `While` presented in Section 2.2. We reach this proc call K in a context E:

$K :=$ `Proc (New While2) hstm (s1, t2)`

$E :=$ `let s2 = [·] in Proc Tail hstm (s2,While(e1,t2))`

The new constructor needs the variables $\widetilde{z} =$ `(e1, t2)` to evaluate E. From the arguments of the proc call, we create the fresh variables $\widetilde{w} =$ `(s0, t0)`. The rule for `While2` is therefore as follows.

```
hstm(s, While2(s0, t0, e1, t2)) :=
  let s2 = Proc Reuse hstm (s0, t0) in
  Proc Tail hstm (s2, While (e1, t2))
```

Example 5 (Without initial constructors reuse)**.** If we do not aim at reusing initial constructors, the analysis simply creates a new constructor for each proc call, which are processed exactly as above. For `Plus(e1, e2)`, we create `Plus1` for the evaluation of `e1` and `Plus2` for `e2`.

```
hexp(s, Plus1(s0, e0, e2)) :=
  let (s', v1) = Proc Reuse hexp (s0,e0) in ...
hexp(s, Plus2(s0, e0, v1)) :=
  let (s'', v2) = Proc Reuse hexp (s0, e0) in ...
```

The generation phase is applied to every annotated rule. This results in new constructors and rules, that we add to the unmodified initial ones.

4.2 Distribute Branchings

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

Reusing constructors is problematic for procedures in nested computations. In `let x = (let y = Proc 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 = Proc eval t in let x = S1 in S2`, and the transformation of Section 4.3 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 $(\texttt{let}\ \widetilde{v} = \texttt{Branch}(S_1, S_2)\ \texttt{in}\ S)$ into a skeleton $\texttt{Branch}(\texttt{let}\ \widetilde{v} = S_1\ \texttt{in}\ S, \texttt{let}\ \widetilde{v} = S_2\ \texttt{in}\ S)$, so that the reusing proc calls in S_1 and S_2 can be transformed in the final phase. To this end, 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.

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

The distribution of `LetIn` over `Branch`, noted $\lceil S \rfloor$ is defined as follows, and is applied to every skeleton at this point of the transformation.

$$\lceil \texttt{Branch}(S_1, .., S_n) \rfloor \triangleq \texttt{Branch}(\lceil S_1 \rfloor, .., \lceil S_n \rfloor)$$
$$\lceil B \rfloor \triangleq B \qquad \text{if } B \neq \texttt{Branch(..)}$$
$$\lceil \texttt{let}\ \widetilde{v} = B\ \texttt{in}\ S \rfloor \triangleq \texttt{let}\ \widetilde{v} = B\ \texttt{in}\ \lceil S \rfloor \quad \text{if } B \neq \texttt{Branch(..)}$$
$$\lceil \texttt{let}\ \widetilde{v} = \texttt{Branch}(S_1, .., S_n)\ \texttt{in}\ S \rfloor \triangleq \texttt{Branch}(\lceil < S_1 \mid \widetilde{v} \mid S > \rfloor, .., \lceil < S_n \mid \widetilde{v} \mid S > \rfloor)$$

4.3 Make the Skeletons Small-Step

With the analysis and generation done so far, we are ready to make the procedures behave in a small-step way. As explained in the overview, we first update their output types to match the input ones. Noting P_0 the initial set of procedures, for all $p \in P_0$ with $\texttt{ptype}(p) = ((\widetilde{s}, s_g), \widetilde{u})$, we redefine: $\texttt{ptype}(p) = ((\widetilde{s}, s_g), (\widetilde{s}, s_g))$.

We also add coercions for return values. For every procedure $\texttt{p} \in P_0$, we create a new constructor `Ret_p` to pack its result and a matching procedure `getRet_p` to unpack it. Coercions turn the return values $\texttt{ptype}_{\texttt{out}}(\texttt{p})$ of the corresponding procedure into an executable program of type $\texttt{ptype}_{\texttt{g}}(\texttt{p})$, while the extracting procedure is doing the opposite. The latter is defined only on the constructor it destructs; the number of variables $\widetilde{v}$ it returns is given by the output type of the procedure, i.e., $\|\texttt{ptype}_{\texttt{out}}(\texttt{p})\| = \|\widetilde{v}\|$.

$$\{\texttt{getRet\_p}(\texttt{Ret\_p}\ \widetilde{v}) := \texttt{Return}\ \widetilde{v} \mid \texttt{p} \in P_0\}$$

We then transform the skeletons defining the procedures, including the rules added in Section 4.1. At this stage of the transformation, the skeletons of our current set of rules respect the following simplified grammar, either directly or with a simple modification.

$$S ::= \texttt{let}\ \widetilde{v} = K\ \texttt{in}\ S \mid \texttt{Branch}\ \widetilde{S} \mid \texttt{Proc Tail}\ p\ \widetilde{t} \mid \texttt{Return}\ \widetilde{t}$$
$$K ::= \texttt{Filter}\ f\ \widetilde{t} \mid \texttt{Proc Reuse}\ p\ \widetilde{w} \mid \texttt{Proc}\ (\texttt{New}\, c)\ p\ \widetilde{t} \mid \texttt{Return}\ \widetilde{t}$$

A tail-call is necessarily a final proc call, so a skeleton $\texttt{let}\ \widetilde{v} = \texttt{Proc Tail}\ p\ \widetilde{t}\ \texttt{in}\ S$ is not possible. Whenever a proc call is annotated `Reuse`, the analysis of Section 4.1 implies that its input terms are all variables $\widetilde{w}$. Because we distribute LetIns over branchings (Section 4.2), we cannot have a skeleton of the form $\texttt{let}\ \widetilde{v} = \texttt{Branch}\ \widetilde{S'}\ \texttt{in}\ S$. Finally, if a skeleton ends with K that is not a tail-call, we transform it into an equivalent one by delaying the return: we replace K with the skeleton $\texttt{let}\ \widetilde{z} = K\ \texttt{in Return}\ \widetilde{z}$, where $\widetilde{z}$ are fresh variables. As such, it is sufficient to define our procedure on skeletons respecting the simplified grammar.

Let $r = (\texttt{pr}(\widetilde{y}, c_r(\widetilde{x})) := S_r)$, and R be the set of rules

$$\| \texttt{Return}\ \widetilde{t} \|^r_\sigma \triangleq \texttt{Return}\ (\widetilde{y}, \texttt{Ret\_pr}(\widetilde{t}))$$
$$\| \texttt{Proc Tail pr}\ \widetilde{t} \|^r_\sigma \triangleq \texttt{Return}\ \widetilde{t}$$
$$\| \texttt{let}\ \widetilde{v} = \texttt{Proc}\ (\texttt{New}\, c)\ p\ \widetilde{t}\ \texttt{in}\ S \|^r_\sigma \triangleq \texttt{Return}\ (\widetilde{y}, c(\widetilde{t}, \widetilde{z_c}))$$
$$\text{where } (\texttt{pr}(\widetilde{y}, c(\widetilde{w_c}, \widetilde{z_c})) := S_c) \in R$$
$$\| \texttt{let}\ \widetilde{v} = \texttt{Proc Reuse}\ p\ (\widetilde{w'}, w)\ \texttt{in}\ S \|^r_\sigma \triangleq \texttt{Branch}(S_1, S_2)$$
$$S_1 = \texttt{let}\ \widetilde{z} = \texttt{Proc}\ p\ \widetilde{w}\ \texttt{in Return}\ (\widetilde{y}, c_r(\widetilde{x}))(\sigma ; \widetilde{[z/w]})$$
$$\text{where } \widetilde{z} \text{ are fresh and } \widetilde{w} = (\widetilde{w'}, w)$$
$$S_2 = \texttt{let}\ \widetilde{v} = \texttt{Proc getRet\_p}\ (w)\ \texttt{in}\ \| S \|^r_{\sigma\,;\,[\texttt{Ret\_p}(\widetilde{v})/w]}$$

Figure 5: Main Cases of the Transformation of a Skeleton

The transformation relies on a substitution to remember how the initial arguments of the rule are changed through the proc calls, as we show in Example 6. A substitution σ is a total mapping from variables to terms that is 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 annotated skeleton S, a rule $r = (\texttt{pr}(\widetilde{y}, c_r(\widetilde{x})) := S_r)$, and a substitution σ representing the knowledge accumulated so far, the transformation $\| S \|^r_\sigma$ results in a plain skeleton—without annotations (cf. Figure 5).

We coerce the results of a final `Return` skelement with the dedicated constructor `Ret_pr`. We remind that we return the environment variables $\widetilde{y}$ of the rule to match the updated typing of the procedure.

A tail-call is simply turned into a return, as the procedure being called is identical to the one where the current rule is defined (see Figure 4).

As explained in Section 2.3, a procedure annotated $(\texttt{New}\, c)$ is turned into a return with a term built with c. One might be surprised the proc call disappears, it is simply delegated to the new rule for c (see Section 4.1 right before Example 4). By construction of c and its rule $(\texttt{pr}(\widetilde{y}, c(\widetilde{w_c}, \widetilde{z_c})) := S_c)$, the variables $\widetilde{w_c}$ are used as input of the analyzed call and $\widetilde{z_c}$ to compute the rest of S_c. The result is therefore $\texttt{Return}\ (\widetilde{y}, c(\widetilde{t}, \widetilde{z_c}))$, where we replace $\widetilde{w_c}$ with the terms $\widetilde{t}$ being reduced.

We change a proc call which can reuse its constructor into a branching representing its possible behaviors. The first branch begins by making a further step $\texttt{let}\ \widetilde{z} = \texttt{Proc}\ p\ \widetilde{w}\ \texttt{in}\ \ldots$, storing

the results in some fresh variables $\widetilde{z}$. We then reconstruct a configuration starting from the initial input $(\widetilde{y}, c_r(\widetilde{x}))$, to which we apply σ before changing $\widetilde{w}$ by their new values $\widetilde{z}$. The substitution σ is necessary if one of the variables $\widetilde{w}$ is not part of the initial arguments but defined from a previous proc call, as shown in the `Plus` example below. The second branch covers the case where the last term represented by w is a coerced set of values: we extract them into $\widetilde{v}$ and continue transforming S, remembering that w is equal to $\texttt{Ret\_p}(\widetilde{v})$.

Example 6. Consider the rule for `Plus`:

```
| Plus (e1, e2) ->
  let (s1, v1) = hexp (s, e1) in      (* reuse *)
  let (s2, v2) = hexp (s1, e2) in     (* reuse *)
  let v = add (v1, v2) in
  (s2, 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_hexp(s1, v1)` in σ.

Transforming the second proc call shows why we need σ. As for the first one, the first branch steps once `(s1, e2)` into some fresh variables `(z3, z4)` and reconstructs a configuration. We see that `s1` does not occur in the initial configuration `(s, Plus (e1, e2))`; we therefore apply the substitution to create `(s, Plus (Ret_hexp(s1, v1), e2))`, and now we can turn `(s1, e2)` into `(z3, z4)`, resulting in the configuration `(s, Plus (Ret_hexp(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 the following skeleton, which behaves as in the three rules below. The comments indicate the origin of the different parts: the first proc call (`'|'`); the second proc call (`'||'`); the unmodified filter call (`'#'`); and the coerced return (`'##'`).

```
hook hexp (s: state, e: exp) matching e: state * exp =
| Plus (e1, e2) ->
  branch                                  (* |     *)
    let (z1, z2) = hexp (s, e1) in        (* |     *)
    (z1, Plus (z2, e2))                   (* |     *)
  or                                      (* |     *)
    let (s1, v1) = getRet_hexp (e1) in    (* |     *)
    branch                                (*  ||   *)
      let (z3, z4) = hexp (s1, e2) in     (*  ||   *)
      (s, Plus (Ret_hexp (z3, v1), z4))   (*  ||   *)
    or                                    (*  ||   *)
      let (s2, v2) = getRet_hexp (e2) in  (*  ||   *)
      let v = add (v1, v2) in             (*    #  *)
      (s, Ret_hexp (s2, v))               (*    ## *)
    end                                   (*  ||   *)
  end                                     (* |     *)
```

$$\frac{s, e_1 \longrightarrow z_1, z_2}{s, e_1 + e_2 \longrightarrow z_1, z_2 + e_2} \qquad \frac{e_1 = (s_1, v_1) \quad s_1, e_2 \longrightarrow z_3, z_4}{s, e_1 + e_2 \longrightarrow s, (z_3, v_1) + z_4}$$

$$\frac{e_1 = (s_1, v_1) \quad e_2 = (s_2, v_2)}{s, e_1 + e_2 \longrightarrow s, (s_2, v_1 + v_2)}$$

If we do not reuse the initial constructors, we do not need the substitution σ as we cannot have nested proc calls where we reuse constructors. Instead, we change constructor, e.g., going from `Plus1` to `Plus2`. As a result, the small-step transformation without reuse is written $\| S \|^r$ and the skeletons S_1 and S_2 of the last case of Figure 5 become:

$$S_1 = \texttt{let}\ \widetilde{z} = \texttt{Proc}\ h\ \widetilde{w}\ \texttt{in Return}\ (\widetilde{y}, c_r(\widetilde{x}))\widetilde{[z/w]}$$
$$\widetilde{z}\ \text{fresh}\ ;\ \widetilde{w} = (\widetilde{w'}, w)$$
$$S_2 = \texttt{let}\ \widetilde{v} = \texttt{Proc getRet\_p}\ (w)\ \texttt{in}\ \| S \|^r$$

Example 7 (Without initial constructors reuse)**.** If we do not reuse `Plus`, the rules for `Plus`, `Plus1`, and `Plus2` behave as follows. Unlike `Plus` in Example 6, the new constructors takes three arguments by default (c.f., Example 5 and Appendix A.6).

$$\frac{}{s, e_1 + e_2 \longrightarrow s, (s, e_1) +_1 e_2} \qquad \frac{s_0, e_1 \longrightarrow s', e_1'}{s, (s_0, e_1) +_1 e_2 \longrightarrow s, (s', e_1') +_1 e_2}$$

$$\frac{}{s, (s_0, v_1) +_1 e_2 \longrightarrow s, v_1 +_2 (s_0, e_2)}$$

$$\frac{s_0, e_2 \longrightarrow s', e_2'}{s, v_1 +_2 (s_0, e_2) \longrightarrow s, v_1 +_2 (s', e_2')}$$

$$\frac{}{s, v_1 +_2 (s_0, v_2) \longrightarrow s, (s_0, v_1 + v_2)}$$

This last phase of the transformation is applied to the rules R defining the procedures $\texttt{p} \in P_0$. The final rules are:

$$\{\texttt{p}(\widetilde{y}, c(\widetilde{x})) := \| S \|_{\epsilon}^{\texttt{p}(\widetilde{y}, c(\widetilde{x})) := S} \mid (\texttt{p}(\widetilde{y}, c(\widetilde{x})) := S) \in R\} \cup$$
$$\{\texttt{getRet\_p}(\texttt{Ret\_p}\ \widetilde{v}) := \texttt{Return}\ \widetilde{v} \mid \texttt{p} \in P_0\}$$

In the end, we obtain a small-step semantics where every initial procedure makes a single step of computation.

The transformation has been implemented in Necro [9], reusing initial constructors by default, but with an option to not reuse them. One of the main features of the framework is the automatic derivation of an OCaml interpreter from a skeletal semantics. The interpreter is parametric in the types and functions representing the base types and filters. Using the transformation, we generate an interpreter providing both big-step and small-step implementations of procedures. The small-step interpreter relies on the same instantiated types and functions for base types and filters as the big-step one, so it provides a small-step semantics at no additional cost. Examples of generated interpreters can be found in the implementation [4].

5 EQUIVALENCE PROOF

We prove that the transformation is correct, i.e., that the initial and transformed semantics are equivalent. To deal with the complexity of the approach, we provide two complementary correctness results. The first one is a pen-and-paper proof that the transformation *without constructor reuse* is correct. The proof is available in Appendix C, and we quickly present the results in Section 5.1. To further our trust in the transformation and to deal with constructor reuse, we provide a mechanized approach to correctness, presented in the rest of this section. This second approach does not attempt to formalize the whole transformation nor the reuse analysis, but instead it generates a fully automatic Coq proof of the equivalence

of the initial and transformed semantics instantiated on a given language.

The pen-and-paper proof gives us confidence that the generation will not fail in the absence of constructor reuse, and the automatic Coq proof shows the correctness of the analysis for constructor reuse for a given language. The generated Coq proof can thus be considered as a certificate for an instance of the transformation.

To state the equivalence theorems between the two semantics, we use the interpretation judgments of Section 3.2 (Figure 2). To simplify notations, we write $\texttt{Ret}_{\texttt{he}}$ (resp. $\texttt{Ret}_{\texttt{hs}}$, $\texttt{Ret}_{\texttt{p}}$) for Ret_hexp (resp. Ret_hstm, Ret_p), as well as $\Downarrow_{\texttt{he}}$ (resp. $\Downarrow_{\texttt{hs}}$) for $\Downarrow_{\texttt{hexp}}$ (resp. $\Downarrow_{\texttt{hstm}}$). We extend the notation to include the semantics, writing $\widetilde{a} \Downarrow_p^{\text{BS}} \widetilde{b}$ to state that p takes the values $\widetilde{a}$ as input and output $\widetilde{b}$ using the initial big-step semantics. We write $\widetilde{a} \Downarrow_p^{\text{SS}} \widetilde{b}$ when we use the generated semantics of Section 4, without reuse in Section 5.1, and with reuse otherwise. We recall that the inductive interpretation is inherently big-step, as a judgment computes the whole skeleton, so we keep the formal notation $\Downarrow_p^{\text{SS}}$ even for the output semantics. However, the resulting set of rules is created such that its interpretation corresponds to a standard small-step reduction.

5.1 Pen-and-paper Proof

As said above, we cover the simplified case where we do not reuse constructors, as proving the analysis of Section 4.1 would make the proof substantially more complex. We would need to justify the correctness of the analysis and that the information is used properly, like in the substitution used in Section 4.3. However, the paper proof applies to both finite and diverging computations.

We prove that a big-step evaluation is possible if and only if a sequence of small-step reductions can lead to the same result up to a $\texttt{Ret}_{\texttt{p}}$ coercion. In the finite case, assuming the terms $\widetilde{a}$ and $\widetilde{b}$ are written using the initial big-step semantics, we show that for all p

$$\widetilde{a} \Downarrow_p^{\text{BS}} \widetilde{b} \iff \exists \widetilde{a'},\ \widetilde{a}\ (\Downarrow_p^{\text{SS}})^*\ (\widetilde{a'}, \texttt{Ret}_{\texttt{p}}(\widetilde{b})).$$

The theorem holds even if the input semantics has an unusual shape. However, it only seems useful if the input follows the big-step paradigm. For instance, if the input is already small-step, then the output would be an odd semantics akin to a malformed abstract machine [4].

For diverging derivations, we define infinite small-step reductions coinductively so that $\widetilde{a} \Downarrow_p^{\text{SS}} \widetilde{b}$ and $\widetilde{b} \overset{\infty}{\to}_p$ implies $\widetilde{a} \overset{\infty}{\to}_p$, and we prove that for all p and initial terms $\widetilde{a}$

$$\widetilde{a} \Uparrow_p^{\text{BS}} \iff \widetilde{a} \overset{\infty}{\to}_p .$$

5.2 Coq Certification Proof Sketch

A complete formalization of the transformation in a proof assistant seems out of reach because of its many steps. Instead, we leverage the Coq generation of Necro, that produces a formalization of an inductive big-step semantics given a skeletal semantics, to *automatically* generate a formalization of the small and big-step semantics and a proof that they coincide.

Unlike the paper proof, our a-posteriori Coq certification includes constructor reuse, as we only check the resulting semantics without having to justify the analysis phase. The Coq script only covers finite evaluations, however, as Necro only generates inductive semantics and some of our proof techniques do not apply in a coinductive setting.

We explain how to generate the scripts using IMP. The implementation [4] contains the script generator and examples of Coq scripts for other languages. For IMP, the correspondence is stated as follows.

Theorem 1. For all $s, s_0 : \texttt{state}$, $e : \texttt{exp}$, $t : \texttt{stm}$, and $v : \texttt{value}$,

$$(s, e) \Downarrow_{\texttt{he}}^{\text{BS}} (s_0, v) \iff \exists s',\ (s, e)\ (\Downarrow_{\texttt{he}}^{\text{SS}})^* (s', \texttt{Ret}_{\texttt{he}}(s_0, v))$$
$$(s, t) \Downarrow_{\texttt{hs}}^{\text{BS}} s_0 \iff \exists s',\ (s, t)\ (\Downarrow_{\texttt{hs}}^{\text{SS}})^* (s', \texttt{Ret}_{\texttt{hs}}(s_0)).$$

The interesting direction is to show that a sequence of small-step reductions implies a big-step evaluation. We use a concatenation lemma [18, 24], stating that we can merge a small step into a big step:

$$e \Downarrow_p^{\text{SS}} e' \Downarrow_p^{\text{BS}} v \implies e \Downarrow_p^{\text{BS}} v$$

Such a result can be stated independently from the semantics, so it is easier to automatize. Its downside is that the big-step and small-step semantics need to be defined on the same constructors. The 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 consider an extended big-step (EBS) semantics defined on all constructors. It corresponds to the situation before the last phase of the transformation (between Sections 4.2 and 4.3), with additional rules to extract values coerced by the new $\texttt{Ret}_{\texttt{p}}$ constructors: $\{p(\widetilde{y}, \texttt{Ret}_{\texttt{p}}(\widetilde{v})) := \texttt{Return}\ \widetilde{v} \mid p \in P_0\}$—we remind that P_0 is the initial set of user-defined procedures. These last new rules allow for judgments of the form $(\widetilde{a}, \texttt{Ret}_{\texttt{p}}(\widetilde{b})) \Downarrow_p^{\text{EBS}} \widetilde{b}$.

Given an input BS semantics, our tool generates the EBS and SS semantics. The EBS and SS semantics are defined on extended types that we prefix with a letter e, writing for instance estate for extended states. The EBS generated for IMP can be found in Appendix A.5. The three semantics are transformed into Coq definitions using the export function of Necro. Necro also provides a Coq definition of the inductive interpretation (Figure 2), which itself depends on interpretations for the base types 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 strategy is 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. The first step is easy to verify since EBS is a conservative extension of BS: we simply match every behavior—every applied rule—in big-step with exactly the same one on the other side.

We note $|t|$ the canonical injection of an initial term t into the extended semantics, e.g., from type stm to estm.

Theorem 2 (BS⇒EBS)**.** For all $(s, s_0 : \texttt{state})$, $(e : \texttt{exp})$, $(t : \texttt{stm})$, and $(v : \texttt{value})$,

$$(s, e) \Downarrow_{\texttt{he}}^{\text{BS}} (s_0, v) \implies (|s|, |e|) \Downarrow_{\texttt{he}}^{\text{EBS}} (|s_0|, |v|)$$
$$(s, t) \Downarrow_{\texttt{hs}}^{\text{BS}} s_0 \implies (|s|, |t|) \Downarrow_{\texttt{hs}}^{\text{EBS}} |s_0|.$$

Theorem 3 (EBS⇒BS)**.** For all $(s : \texttt{state})$, $(e : \texttt{exp})$, $(t : \texttt{stm})$, $(s'_0 : \texttt{estate})$, and $(v' : \texttt{evalue})$,

$$(|s|, |e|) \Downarrow^{\text{EBS}}_{\texttt{he}} (s'_0, v') \implies \exists s_0, v, (s'_0, v') = (|s_0|, |v|) \wedge (s, e) \Downarrow^{\text{BS}}_{\texttt{he}} (s_0, v)$$
$$(|s|, |t|) \Downarrow^{\text{EBS}}_{\texttt{hs}} s'_0 \implies \exists s_0,\ s'_0 = |s_0| \ \wedge\ (s, t) \Downarrow^{\text{BS}}_{\texttt{hs}} s_0$$

5.3 Small-Step Implies Extended Big-Step

As hinted above, we rely on a concatenation lemma.

Lemma 4 (Concatenation)**.** For all s, s', s_0, e, e', t, t', and v,

$$(s, e) \Downarrow^{\text{SS}}_{\texttt{he}} (s', e') \Downarrow^{\text{EBS}}_{\texttt{he}} (s_0, v) \implies (s, e) \Downarrow^{\text{EBS}}_{\texttt{he}} (s_0, v)$$
$$(s, t) \Downarrow^{\text{SS}}_{\texttt{hs}} (s', t') \Downarrow^{\text{EBS}}_{\texttt{hs}} s_0 \implies (s, t) \Downarrow^{\text{EBS}}_{\texttt{hs}} s_0$$

It is a local result, as it considers small steps one at a time, making it easier to prove automatically. Indeed, the proof is done by induction on the small step; in each case, we also decompose the big-step hypothesis which generates numerous subcases. Fortunately, elementary tactics are sufficient for Coq to automatically reconstruct the extended big step in all cases: it is a simple case analysis to find the big-step hypothesis whose source coincides with the conclusion of the small-step induction hypothesis.

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

Theorem 5 (SS⇒EBS)**.** For all s, s', s_0, e, t, and v,

$$(s, e)\ (\Downarrow^{\text{SS}}_{\texttt{he}})^*\ (s', \texttt{Ret}_{\texttt{he}}(s_0, v)) \implies (s, e) \Downarrow^{\text{EBS}}_{\texttt{he}} (s_0, v)$$
$$(s, t)\ (\Downarrow^{\text{SS}}_{\texttt{hs}})^*\ (s', \texttt{Ret}_{\texttt{hs}}(s_0)) \implies (s, t) \Downarrow^{\text{EBS}}_{\texttt{hs}} s_0$$

5.4 Extended Big-Step Implies Small-Step

For the sake of readability, we illustrate this section with a `Plus` example without states. We also write `Ret` for $\texttt{Ret}_{\texttt{he}}$, and $\underset{\text{SS}}{\rightarrow}$ for the interpretation of the small-step semantics.

We prove that big-step implies small-step on extended terms 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 $\texttt{Plus}(e_1, e_2) \Downarrow^{\text{EBS}}_{\texttt{he}} v$ produces the hypotheses $e_1 \underset{\text{SS}}{\rightarrow}^* \texttt{Ret}(v_1)$, $e_2 \underset{\text{SS}}{\rightarrow}^* \texttt{Ret}(v_2)$, and $v = v_1 + v_2$, from which we want to show $\texttt{Plus}(e_1, e_2) \underset{\text{SS}}{\rightarrow}^* \texttt{Ret}(v)$.

We could try a a brute-force search as in the previous section, but it would require Coq to guess the intermediate configurations, such as $\texttt{Plus}(e'_1, e_2)$ if $e_1 \underset{\text{SS}}{\rightarrow} e'_1 \underset{\text{SS}}{\rightarrow}^* \texttt{Ret}(v_1)$. It works on small languages like IMP, but not for larger ones like miniML. Instead, we help by generating congruence results about the small-step reduction. We still need Coq to automatically find the right combination of lemmas to apply, but at least the intermediary configurations are given by the lemmas. For `Plus`, we need results of the form:

$$e_1 \underset{\text{SS}}{\rightarrow}^* e'_1 \text{ implies } \texttt{Plus}(e_1, e_2) \underset{\text{SS}}{\rightarrow}^* \texttt{Plus}(e'_1, e_2)$$
$$e_2 \underset{\text{SS}}{\rightarrow}^* e'_2 \text{ implies } \texttt{Plus}(\texttt{Ret}(v_1), e_2) \underset{\text{SS}}{\rightarrow}^* \texttt{Plus}(\texttt{Ret}(v_1), e'_2)$$
$$v = v_1 + v_2 \text{ implies } \texttt{Plus}(\texttt{Ret}(v_1), \texttt{Ret}(v_2)) \underset{\text{SS}}{\rightarrow}^* \texttt{Ret}(v)$$

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 to derive them from the skeletons; it is the only part of the proof script which really depends on the semantics of the language.

	Lines of Code			Constructors		
Language	BS	SS	Coq	BS	SS	NR
Call-by-Name	28	41	110	3	4	5
Call-by-Value	22	41	100	3	4	5
CBV, choice	29	48	120	4	5	6
CBV, fail	42	60	150	5	6	7
Arithmetic	32	81	160	5	5	13
IMP	79	144	330	11	13	21
IMP, write in exp	84	154	350	12	14	23
IMP, LetIn	85	166	360	12	16	24
IMP, try/catch	123	192	420	15	17	26
MiniML	155	299	720	18	28	33

Table 1: Size of the Generated Semantics and Proof Scripts

Each generated lemma corresponds to a path in a small-step rule: for instance, the three inference rules describing the behavior of the skeleton for `Plus` in Example 6 corresponds to the three results above (with states). For each path, the LetIns definitions of the skeleton contain the hypothesis of the lemma, while the final result is the configuration to step towards. The generated lemmas are automatically 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.

With these results, 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 6 (EBS⇒SS)**.** For all s, s_0, e, t, and v,

$$(s, e) \Downarrow^{\text{EBS}}_{\texttt{he}} (s_0, v) \implies \exists s',\ (s, e)\ (\Downarrow^{\text{SS}}_{\texttt{he}})^*\ (s', \texttt{Ret}_{\texttt{he}}(s_0, v))$$
$$(s, t) \Downarrow^{\text{EBS}}_{\texttt{hs}} s_0 \implies \exists s',\ (s, t)\ (\Downarrow^{\text{SS}}_{\texttt{hs}})^*\ (s', \texttt{Ret}_{\texttt{hs}}(s_0))$$

6 EVALUATION

We compare the size of the generated small-step semantics and equivalence proof scripts for various languages in Table 1. In the constructors columns, BS, SS, and NR contain the number of constructors respectively in the input, output (not counting the constructors coercing values into terms), and in the output without reuse.

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. The examples written in an imperative style include a small arithmetic language, 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, and constructs

to define and handle algebraic datatypes. The generated small-step semantics and proofs for the examples can be found in the implementation [4].

In the certification, about 500 lines of code are completely independent of the input language and contain definitions of skeletons or of the inductive 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 procedures, constructors, and filters): these are general definitions, results, and tactics to manipulate the inductive interpretation. 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. The size of these parts is 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 mirror the different paths in the small-step rules (see Section 5.4).

W.r.t. constructors, we see the impact of constructor reuse especially in the IMP case, where we go from two new constructors with reuse to ten new constructors without reuse.

The generated semantics diverge from the ones written by hand when they duplicate the environment variables in the new constructors, like the state for IMP (Section 2.3). Duplication occurs in coercions such as `Ret_hexp` to make the small steps uniform, with the same input and output types. For new constructors like `While1`, or `While2`, it seems to be the price to pay for a systematic transformation, as it occurs in Vesely and Fisher's work [27, Section 7], but also when writing pretty-big step semantics [6, Section 5.2]. An extra analysis could detect that the state can be removed from `While1` or `While2`, but it is not clear that such an optimization is possible in general. Besides, it would prevent other static analyses, such as detecting read-only states.

7 RELATED WORK

Our work uses skeletal semantics to benefit from the simplicity of its foundations. The $\mathbb{K}$ framework [25] is noticeably more complex and does not provide a translation into Coq. Other semantic tools, such as Lem [20] and Ott [26], are not equipped for the direct manipulation of semantics—they do not give access to a meta-language. Here, the small meta-language allows for a simple transformation, at the cost of harder definitions of some advanced language features.

While several approaches go from a small-step to a big-step setting by manipulating either inference rules [7, 23] or interpreters [11, 12], the other direction has been less pursued.

Vesely and Fisher [27] 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`. Roughly, the transformation starts with a partial CPS-transform of `eval` to turn recursive calls into continuations, considered as newly created terms. After making `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 and Fisher's approach only considers subcomputations as reduction steps, as they transform only `eval` calls—similar to proc 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 the approach scales to several mutually recursive 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 $\mathtt{App}(t_1, t_2)$ with a subreduction $s, t_1 \longrightarrow s', t_1'$, we would reconstruct a configuration as $s, \mathtt{App}(t_1, t_2) \longrightarrow s, \mathtt{App1}(s', t_1', t_2)$ while they would reconstruct a term as $s \vdash \mathtt{App}(t_1, t_2) \longrightarrow \mathtt{App}(C(s', t_1'), t_2)$. 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, Vesely and Fisher [27] claim to have informal proofs of several parts of their transformation. We have a language-independent proof of the transformation without initial constructors reuse for terminating and diverging evaluations. With reuse, we generate an equivalence proof script for terminating evaluations for any input semantics.

Huizing et al. [13] present a transformation from a big-step to a small-step semantics, by directly manipulating inference rules. Small-step configurations are extended with a stack to keep track of the big-step premises that have already been computed. For each 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 premise under consideration, or reduce it. 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 as well as the stack make the resulting semantics very different from usual SOS definitions.

The functional correspondence [2] is a systematic approach to transform a big-step interpreter into an abstract machine. Compared to a small-step SOS, an abstract machine is closer to an implementation. In particular, it usually does not reconstruct the whole term after reducing a redex. In a recent paper [3], we use the functional correspondence to derive an abstract machine for the skeletal semantics meta-language of Section 3, thus providing an executable interpretation of skeletons. The resulting machine is proven sound w.r.t. concrete interpretation using Coq. Unlike the present work, where we transform directly user-defined languages, our previous work stays at the meta-level: the abstract machine is derived once and for all, and it manipulates skeletons. Even when instantiated with a user-defined language, the resulting machine is very difficult to read and far from an efficient abstract

machine for that language, as it reduces skeletons and not language constructs directly. The goal of our previous paper was to get a certified—albeit slow—implementation of a language, not to generate a human-readable semantics, as in this work.

8 CONCLUSION

We present a fully automatic transformation from a big-step to a small-step skeletal semantics with or without reuse of the initial constructors. With reuse, we generate new constructors only for problematic program points. The resulting small-step skeletal semantics can then be given to Necro to generate OCaml interpreters or Coq formalizations. We exploit the latter feature to automatically and a-posteriori certify the correctness of the result of the transformation with reuse on any language for terminating evaluations. The transformation without reuse is proved to be correct independently from the input language in the terminating and diverging cases. The transformations and the proofs make no assumptions about filters, and the filters defined for the big-step input are used by the small-step outputs up to coercions. Our work remains parametric in the implementation of filters, in the spirit of skeletal semantics. Experimenting our method with several languages shows the transformation with reuse produces small-step skeletal semantics close to the SOS ones that one would write by hand.

REFERENCES

[1] 2021. ECMAScript language specification. Standard ECMA-262. https://262.ecma-international.org/

[2] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. 2003. A functional correspondence between evaluators and abstract machines. In *Proceedings of the 5th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, 27-29 August 2003, Uppsala, Sweden*. ACM, 8–19. https://doi.org/10.1145/888251.888254

[3] Guillaume Ambal, Sergueï Lenglet, and Alan Schmitt. 2022. Certified abstract machines for skeletal semantics. In *CPP '22: 11th ACM SIGPLAN International Conference on Certified Programs and Proofs, Philadelphia, PA, USA, January 17 - 18, 2022*, Andrei Popescu and Steve Zdancewic (Eds.). ACM, 55–67. https://doi.org/10.1145/3497775.3503676

[4] Guillaume Ambal, Sergueï Lenglet, and Alan Schmitt. 2022. Certified Derivation of Small-Step From Big-Step Skeletal Semantics. Implementation available at https://gitlab.inria.fr/skeletons/necro/-/tree/PPDP2022.

[5] Martin Bodin, Philippa Gardner, Thomas Jensen, and Alan Schmitt. 2019. Skeletal semantics and their interpretations. *PACMPL* 3, POPL (2019), 44:1–44:31. https://doi.org/10.1145/3290357

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

[7] Ştefan Ciobâcă. 2013. From Small-Step Semantics to Big-Step Semantics, Automatically. In *Integrated Formal Methods, 10th International Conference, IFM 2013, Turku, Finland, June 10-14, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7940)*, Einar Broch Johnsen and Luigia Petre (Eds.). Springer, 347–361. https://doi.org/10.1007/978-3-642-38613-8_24

[8] The Coq Development Team. 2020. *The Coq Proof Assistant Reference Manual, version 8.11*. http://coq.inria.fr

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

[10] Olivier Danvy. 2004. A Rational Deconstruction of Landin's SECD Machine. In *Implementation and Application of Functional Languages, 16th International Workshop, IFL 2004, Lübeck, Germany, September 8-10, 2004, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 3474)*, Clemens Grelck, Frank Huch, Greg Michaelson, and Philip W. Trinder (Eds.). Springer, 52–71. https://doi.org/10.1007/11431664_4

[11] Olivier Danvy. 2005. From Reduction-based to Reduction-free Normalization. *Electron. Notes Theor. Comput. Sci.* 124, 2 (2005), 79–100. https://doi.org/10.1016/j.entcs.2005.01.007

[12] Olivier Danvy, Jacob Johannsen, and Ian Zerny. 2011. A walk in the semantic park. In *Proceedings of the 2011 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2011, Austin, TX, USA, January 24-25, 2011*, Siau-Cheng Khoo and Jeremy G. Siek (Eds.). ACM, 1–12. https://doi.org/10.1145/1929501.1929503

[13] Cornelis Huizing, Ron Koymans, and Ruurd Kuiper. 2010. A Small Step for Mankind. In *Concurrency, Compositionality, and Correctness, Essays in Honor of Willem-Paul de Roever (Lecture Notes in Computer Science, Vol. 5930)*, Dennis Dams, Ulrich Hannemann, and Martin Steffen (Eds.). Springer, 66–73. https://doi.org/10.1007/978-3-642-11512-7_5

[14] Gilles Kahn. 1987. Natural Semantics. In *STACS 87, 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany, February 19-21, 1987, Proceedings (Lecture Notes in Computer Science, Vol. 247)*, Franz-Josef Brandenburg, Guy Vidal-Naquet, and Martin Wirsing (Eds.). Springer, 22–39. https://doi.org/10.1007/BFb0039592

[15] P. J. Landin. 1964. The Mechanical Evaluation of Expressions. *Comput. J.* 6, 4 (1964), 308–320. https://doi.org/10.1093/comjnl/6.4.308

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

[17] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. 2020. *The OCaml system release 4.10*. Inria. https://caml.inria.fr/pub/docs/manual-ocaml/

[18] Xavier Leroy and Hervé Grall. 2009. Coinductive big-step operational semantics. *Inf. Comput.* 207, 2 (2009), 284–304. https://doi.org/10.1016/j.ic.2007.12.004

[19] Robin Milner, Robert Harper, David MacQueen, and Tofte Mads. 1997. *The Definition of Standard ML, Revised Edition*.

[20] Dominic P. Mulligan, Scott Owens, Kathryn E. Gray, Tom Ridge, and Peter Sewell. 2014. Lem: reusable engineering of real-world semantics. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, Johan Jeuring and Manuel M. T. Chakravarty (Eds.). ACM, 175–188. https://doi.org/10.1145/2628136.2628143

[21] Hanne Riis Nielson and Flemming Nielson. 1992. *Semantics with applications - a formal introduction*. Wiley.

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

[23] Casper Bach Poulsen and Peter D. Mosses. 2014. Deriving Pretty-Big-Step Semantics from Small-Step Semantics. In *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 (Lecture Notes in Computer Science, Vol. 8410)*, Zhong Shao (Ed.). Springer, 270–289. https://doi.org/10.1007/978-3-642-54833-8_15

[24] Casper Bach Poulsen and Peter D. Mosses. 2017. Flag-based big-step semantics. *J. Log. Algebraic Methods Program.* 88 (2017), 174–190. https://doi.org/10.1016/j.jlamp.2016.05.001

[25] Traian-Florin Serbanuta, Andrei Arusoaie, David Lazar, Chucky Ellison, Dorel Lucanu, and Grigore Rosu. 2014. The K Primer (version 3.3). *Electron. Notes Theor. Comput. Sci.* 304 (2014), 57–80. https://doi.org/10.1016/j.entcs.2014.05.003

[26] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strnisa. 2010. Ott: Effective tool support for the working semanticist. *J. Funct. Program.* 20, 1 (2010), 71–122. https://doi.org/10.1017/S0956796809990293

[27] Ferdinand Vesely and Kathleen Fisher. 2019. One Step at a Time - A Functional Derivation of Small-Step Evaluators from Big-Step Counterparts. In *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 (Lecture Notes in Computer Science, Vol. 11423)*, Luís Caires (Ed.). Springer, 205–231. https://doi.org/10.1007/978-3-030-17184-1_8

[28] Andrew K. Wright and Matthias Felleisen. 1994. A Syntactic Approach to Type Soundness. *Information and Computation* 115, 1 (1994), 38–94. https://doi.org/10.1006/inco.1994.1093

APPENDICES

Appendix A contains the successive skeletal semantics of the language IMP at each phase of transformation, including the final semantics with and without constructor reuse. The only modification we made to the generated files is a renaming of the generated names to ease the reading.

Appendix B gives more details on the transformation phases of Section 4. Notably, it contains definitions skipped in the paper, and notations used in the following proof.

Appendix C contains a pen-and-paper proof of the small-step transformation presented in the accompanying paper. We only certify the base transformation, without reuse of constructors or tail-calls. We do not certify the analysis, whose main purpose is to optimize the reuse of constructors, and assume initial proc calls are all annotated with fresh constructor names (New c).

A SUCCESSIVE TRANSFORMATIONS OF IMP

A.1 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 (s1, v1) = hexpr (s, e1) in
  let (s2, v2) = hexpr (s1, e2) in
  let v = add (v1, v2) in
```

```
  (s2, v)
| Equal (e1, e2) ->
  let (s1, v1) = hexpr (s, e1) in
  let (s2, v2) = hexpr (s1, e2) in
  let v = eq (v1, v2) in
  (s2, v)
| Not (e1) ->
  let (s1, v) = hexpr (s, e1) in
  let v1 = neg (v) in
  (s1, v1)

hook hstmt (s : state, t : stmt) matching t : state =
| Skip -> s
| Assign (x, e) ->
  let (s1, v) = hexpr (s, e) in
  write (x, s1, v)
| Seq (t1, t2) ->
  let s1 = hstmt (s, t1) in
  hstmt (s1, t2)
| If (e1, t2, t3) ->
  let (s1, v) = hexpr (s, e1) in
  branch
    let () = isTrue (v) in
    hstmt (s1, t2)
  or
    let () = isFalse (v) in
    hstmt (s1, t3)
  end
| While (e1, t2) ->
  let (s1, v) = hexpr (s, e1) in
  branch
    let () = isTrue (v) in
    let s2 = hstmt (s1, t2) in
    hstmt (s2, While (e1, t2))
  or
    let () = isFalse (v) in
    s1
  end
```

A.2 After Adding Coercions

In the implementation, the new constructors and procedures for coercions are created first. Since it is independent from the rest of the transformation, it could be done at any point. It is only presented last in Section 4.3 to facilitate the reading.

```
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 (e : expr) matching e : 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 e1 ->
  let (s1, v) = hexpr (s, e1) in
  let v1 = neg (v) in
  (s1, v1)
| Bconst b ->
  let v = boolToVal (b) in
  (s, v)
| Equal (e1, e2) ->
  let (s1, v1) = hexpr (s, e1) in
  let (s2, v2) = hexpr (s1, e2) in
  let v = eq (v1, v2) in
  (s2, v)
| Iconst i ->
  let v = intToVal (i) in
  (s, v)
| Plus (e1, e2) ->
  let (s1, v1) = hexpr (s, e1) in
  let (s2, v2) = hexpr (s1, e2) in
  let v = add (v1, v2) in
  (s2, v)
| Var x ->
  let v = read (x, s) in
  (s, v)

hook hstmt (s : state, t : stmt) matching t : state =
| Assign (x, e) ->
  let (s1, v) = hexpr (s, e) in
  write (x, s1, v)
| If (e1, t2, t3) ->
  let (s1, v) = hexpr (s, e1) in
```

```
  branch
    let () = isTrue (v) in
    hstmt (s1, t2)
  or
    let () = isFalse (v) in
    hstmt (s1, t3)
  end
| Seq (t1, t2) ->
  let s1 = hstmt (s, t1) in
  hstmt (s1, t2)
| Skip ->
  s
| While (e1, t2) ->
  let (s1, v) = hexpr (s, e1) in
  branch
    let () = isTrue (v) in
    let s2 = hstmt (s1, t2) in
    hstmt (s2, (While (e1, t2)))
  or
    let () = isFalse (v) in
    s1
  end
```

A.3 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 (e : expr) matching e : 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 e1 ->
  let (s1, v) = hexpr (s, e1) in            (* reuse *)
  let v1 = neg (v) in
  (s1, v1)
| Bconst b ->
  let v = boolToVal (b) in
  (s, v)
| Equal (e1, e2) ->
  let (s1, v1) = hexpr (s, e1) in           (* reuse *)
  let (s2, v2) = hexpr (s1, e2) in          (* reuse *)
  let v = eq (v1, v2) in
  (s2, v)
| Iconst i ->
  let v = intToVal (i) in
  (s, v)
| Plus (e1, e2) ->
  let (s1, v1) = hexpr (s, e1) in           (* reuse *)
  let (s2, v2) = hexpr (s1, e2) in          (* reuse *)
  let v = add (v1, v2) in
  (s2, v)
| Var x ->
  let v = read (x, s) in
  (s, v)

hook hstmt (s : state, t : stmt) matching t : state =
| Assign (x, e) ->
  let (s1, v) = hexpr (s, e) in             (* reuse *)
  write (x, s1, v)
| If (e1, t2, t3) ->
  let (s1, v) = hexpr (s, e1) in            (* reuse *)
  branch
    let () = isTrue (v) in
    hstmt (s1, t2)                          (* tail-call *)
  or
    let () = isFalse (v) in
    hstmt (s1, t3)                          (* tail-call *)
  end
| Seq (t1, t2) ->
  let s1 = hstmt (s, t1) in                 (* reuse *)
  hstmt (s1, t2)                            (* tail-call *)
| Skip ->
  s
| While (e1, t2) ->
  let (s1, v) = hexpr (s, e1) in            (* new constr: While1 *)
  branch
    let () = isTrue (v) in
```

```
    let s2 = hstmt (s1, t2) in              (* new constr: While2 *)
    hstmt (s2, (While (e1, t2)))            (* tail-call *)
  or
    let () = isFalse (v) in
    s1
  end
| While1 (s0, e0, e1, t2) ->
  let (s1, v) = hexpr (s0, e0) in           (* reuse *)
  branch
    let () = isTrue (v) in
    let s2 = hstmt (s1, t2) in              (* new constr: While2 *)
    hstmt (s2, (While (e1, t2)))            (* tail-call *)
  or
    let () = isFalse (v) in
    s1
  end
| While2 (s0, t0, e1, t2) ->
  let s2 = hstmt (s0, t0) in                (* reuse *)
  hstmt (s2, (While (e1, t2)))              (* tail-call *)
```

A.4 Final Small-Step Skeletal Semantics

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

type expr =
| Bconst of bool
| Equal of expr * expr
| Iconst of int
| Not of expr
| 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 (e : expr) matching e : 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 =
| 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 (s1, v1) = getRet_hexpr (e1) in
    branch
      let (z3, z4) = hexpr (s1, e2) in
      (s, Equal (Ret_hexpr (z3, v1), z4))
    or
      let (s2, v2) = getRet_hexpr (e2) in
      let v = eq (v1, v2) in
      (s, Ret_hexpr (s2, v))
    end
  end
| Iconst i ->
  let v = intToVal (i) in
  (s, Ret_hexpr (s, v))
| Not e1 ->
  branch
    let (z1, z2) = hexpr (s, e1) in
    (z1, Not z2)
  or
    let (s1, v) = getRet_hexpr (e1) in
    let v1 = neg (v) in
    (s, Ret_hexpr (s1, v1))
  end
| Plus (e1, e2) ->
  branch
    let (z1, z2) = hexpr (s, e1) in
    (z1, Plus (z2, e2))
  or
    let (s1, v1) = getRet_hexpr (e1) in
    branch
      let (z3, z4) = hexpr (s1, e2) in
      (s, Plus (Ret_hexpr (z3, v1), z4))
    or
      let (s2, v2) = getRet_hexpr (e2) in
      let v = add (v1, v2) in
      (s, Ret_hexpr (s2, v))
    end
  end
| Var x ->
  let v = read (x, s) in
  (s, Ret_hexpr (s, v))
```

$$\frac{}{s, b \longrightarrow s, (s, \mathsf{boolToVal}(b))}$$

$$\frac{s, e_1 \longrightarrow z_1, z_2}{s, (e_1 \stackrel{?}{=} e_2) \longrightarrow z_1, (z_2 \stackrel{?}{=} e_2)}$$

$$\frac{e_1 = (s_1, v_1) \qquad s_1, e_2 \longrightarrow z_3, z_4}{s, (e_1 \stackrel{?}{=} e_2) \longrightarrow s, ((z_3, v_1) \stackrel{?}{=} z_4)}$$

$$\frac{e_1 = (s_1, v_1) \qquad e_2 = (s_2, v_2)}{s, (e_1 \stackrel{?}{=} e_2) \longrightarrow s, (s_2, (v_1 \stackrel{?}{=} v_2))}$$

$$\frac{}{s, i \longrightarrow s, (s, \mathsf{intToVal}(i))}$$

$$\frac{s, e_1 \longrightarrow z_1, z_2}{s, \mathsf{Not}(e_1) \longrightarrow z_1, \mathsf{Not}(z_2)}$$

$$\frac{e_1 = (s_1, v)}{s, \mathsf{Not}(e_1) \longrightarrow s, (s_1, \neg v)}$$

$$\frac{s, e_1 \longrightarrow z_1, z_2}{s, e_1 + e_2 \longrightarrow z_1, z_2 + e_2}$$

$$\frac{e_1 = (s_1, v_1) \qquad s_1, e_2 \longrightarrow z_3, z_4}{s, e_1 + e_2 \longrightarrow s, (z_3, v_1) + z_4}$$

$$\frac{e_1 = (s_1, v_1) \qquad e_2 = (s_2, v_2)}{s, e_1 + e_2 \longrightarrow s, (s_2, v_1 + v_2)}$$

$$\frac{}{s, x \longrightarrow s, (s, s[x])}$$

```
hook hstmt (s : state, t : stmt) matching t : state * stmt =
| Assign (x, e) ->
  branch
    let (z1, z2) = hexpr (s, e) in
    (z1, Assign (x, z2))
  or
    let (s1, v) = getRet_hexpr (e) in
    let z3 = write (x, s1, 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 (s1, v) = getRet_hexpr (e1) in
    branch
      let () = isTrue (v) in
      (s1, t2)
    or
      let () = isFalse (v) in
      (s1, t3)
    end
  end
| Seq (t1, t2) ->
  branch
    let (z1, z2) = hstmt (s, t1) in
    (z1, Seq (z2, t2))
  or
    let s1 = getRet_hstmt (t1) in
    (s1, 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 (s1, v) = getRet_hexpr (e0) in
    branch
      let () = isTrue (v) in
      (s, (While2 (s1, t2, e1, t2)))
    or
      let () = isFalse (v) in
      (s, Ret_hstmt s1)
    end
  end
| While2 (s0, t0, e1, t2) ->
  branch
    let (z1, z2) = hstmt (s0, t0) in
    (s, While2 (z1, z2, e1, t2))
  or
    let s2 = getRet_hstmt (t0) in
    (s2, (While (e1, t2)))
  end
```

$$\frac{s, e \longrightarrow z_1, z_2}{s, (x := e) \longrightarrow z_1, (x := z_2)}$$

$$\frac{e = (s_1, v)}{s, (x := e) \longrightarrow s, (s_1[x \mapsto v])}$$

$$\frac{s, e_1 \longrightarrow z_1, z_2}{s, \mathtt{If}(e_1, t_2, t_3) \longrightarrow z_1, \mathtt{If}(z_2, t_2, t_3)}$$

$$\frac{e_1 = (s_1, v) \qquad \mathtt{isTrue}(v)}{s, \mathtt{If}(e_1, t_2, t_3) \longrightarrow s_1, t_2}$$

$$\frac{e_1 = (s_1, v) \qquad \mathtt{isFalse}(v)}{s, \mathtt{If}(e_1, t_2, t_3) \longrightarrow s_1, t_3}$$

$$\frac{s, t_1 \longrightarrow z_1, z_2}{s, t_1; t_2 \longrightarrow z_1, z_2; t_2}$$

$$\frac{t_1 = (s_1)}{s, t_1; t_2 \longrightarrow s_1, t_2}$$

$$\frac{}{s, \mathtt{Skip} \longrightarrow s, (s)}$$

$$\frac{}{s, \mathtt{While}(e_1, t_2) \longrightarrow s, \mathtt{While1}(s, e_1, e_1, t_2)}$$

$$\frac{s_0, e_0 \longrightarrow z_1, z_2}{s, \mathtt{While1}(s_0, e_0, e_1, t_2) \longrightarrow s, \mathtt{While1}(z_1, z_2, e_1, t_2)}$$

$$\frac{e_0 = (s_1, v) \qquad \mathtt{isTrue}(v)}{s, \mathtt{While1}(s_0, e_0, e_1, t_2) \longrightarrow s, \mathtt{While2}(s_1, t_2, e_1, t_2)}$$

$$\frac{e_0 = (s_1, v) \qquad \mathtt{isFalse}(v)}{s, \mathtt{While1}(s_0, e_0, e_1, t_2) \longrightarrow s, (s_1)}$$

$$\frac{s_0, t_0 \longrightarrow z_1, z_2}{s, \mathtt{While2}(s_0, t_0, e_1, t_2) \longrightarrow s, \mathtt{While2}(z_1, z_2, e_1, t_2)}$$

$$\frac{t_0 = (s_2)}{s, \mathtt{While2}(s_0, t_0, e_1, t_2) \longrightarrow s_2, \mathtt{While}(e_1, t_2)}$$

A.5 Extended Big-Step for Coq Certification

```
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 e1 ->
  let (s1, v) = hexpr (s, e1) in
  let v1 = neg (v) in
  (s1, v1)
| Bconst b ->
  let v = boolToVal (b) in
  (s, v)
| Equal (e1, e2) ->
  let (s1, v1) = hexpr (s, e1) in
  let (s2, v2) = hexpr (s1, e2) in
  let v = eq (v1, v2) in
  (s2, v)
| Iconst i ->
  let v = intToVal (i) in
  (s, v)
| Plus (e1, e2) ->
  let (s1, v1) = hexpr (s, e1) in
  let (s2, v2) = hexpr (s1, e2) in
  let v = add (v1, v2) in
  (s2, 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, e) ->
  let (s1, v) = hexpr (s, e) in
  write (x, s1, v)
| If (e1, t2, t3) ->
  let (s1, v) = hexpr (s, e1) in
  branch
    let () = isTrue (v) in
    hstmt (s1, t2)
  or
    let () = isFalse (v) in
    hstmt (s1, t3)
  end
| Seq (t1, t2) ->
  let s1 = hstmt (s, t1) in
  hstmt (s1, t2)
| Skip ->
  s
| While (e1, t2) ->
  let (s1, v) = hexpr (s, e1) in
  branch
    let () = isTrue (v) in
    let s2 = hstmt (s1, t2) in
    hstmt (s2, (While (e1, t2)))
  or
    let () = isFalse (v) in
    s1
  end
| While1 (s0, e0, e1, t2) ->
  let (s1, v) = hexpr (s0, e0) in
  branch
    let () = isTrue (v) in
    let s2 = hstmt (s1, t2) in
    hstmt (s2, (While (e1, t2)))
  or
    let () = isFalse (v) in
    s1
  end
| While2 (s0, t0, e1, t2) ->
  let s2 = hstmt (s0, t0) in
  hstmt (s2, (While (e1, t2)))
| Ret_hstmt v1 -> v1
```

A.6 Resulting Small-Step without Reuse

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

```
type expr =
| Not of expr
| Not1 of state * expr
| Bconst of bool
| Equal of expr * expr
| Equal1 of state * expr * expr
| Equal2 of state * expr * value
| Iconst of int
| Plus of expr * expr
| Plus1 of state * expr * expr
| Plus2 of state * expr * value
| Var of ident
| Ret_hexpr of state * value
type stmt =
| Assign of ident * expr
| Assign1 of state * expr * ident
| If of expr * stmt * stmt
| If1 of state * expr * stmt * stmt
| Seq of stmt * stmt
| Seq1 of state * 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 (e : expr) matching e : 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 e1 ->
    (s, Not1 (s, e1))
  | Not1 (s0, e0) ->
    branch
      let (z1, z2) = hexpr (s0, e0) in
      (s, Not1 (z1, z2))
    or
      let (s1, v) = getRet_hexpr (e0) in
      let v1 = neg (v) in
      (s, Ret_hexpr (s1, v1))
    end
  | Bconst b ->
    let v = boolToVal (b) in
    (s, Ret_hexpr (s, v))
```

$$\frac{}{s, \mathtt{Not}(e_1) \longrightarrow s, \mathtt{Not1}(s, e_1)}$$

$$\frac{s_0, e_0 \longrightarrow z_1, z_2}{s, \mathtt{Not1}(s_0, e_0) \longrightarrow s, \mathtt{Not1}(z_1, z_2)}$$

$$\frac{e_0 = (s_1, v)}{s, \mathtt{Not1}(s_0, e_0) \longrightarrow s, (s_1, \neg v)}$$

$$\frac{}{s, b \longrightarrow s, (s, \mathtt{boolToVal}(b))}$$

```
| Equal (e1, e2) ->
  (s, Equal1 (s, e1, e2))
| Equal1 (s0, e0, e2) ->
  branch
    let (z1, z2) = hexpr (s0, e0) in
    (s, Equal1 (z1, z2, e2))
  or
    let (s1, v1) = getRet_hexpr (e0) in
    (s, Equal2 (s1, e2, v1))
  end
| Equal2 (s0, e0, v1) ->
  branch
    let (z1, z2) = hexpr (s0, e0) in
    (s, Equal2 (z1, z2, v1))
  or
    let (s2, v2) = getRet_hexpr (e0) in
    let v = eq (v1, v2) in
    (s, Ret_hexpr (s2, v))
  end
| Iconst i ->
  let v = intToVal (i) in
  (s, Ret_hexpr (s, v))
| Plus (e1, e2) ->
  (s, Plus1 (s, e1, e2))
| Plus1 (s0, e0, e2) ->
  branch
    let (z1, z2) = hexpr (s0, e0) in
    (s, Plus1 (z1, z2, e2))
  or
    let (s1, v1) = getRet_hexpr (e0) in
    (s, Plus2 (s1, e2, v1))
  end
| Plus2 (s0, e0, v1) ->
  branch
    let (z1, z2) = hexpr (s0, e0) in
    (s, Plus2 (z1, z2, v1))
  or
    let (s2, v2) = getRet_hexpr (e0) in
    let v = add (v1, v2) in
    (s, Ret_hexpr (s2, v))
  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, e) ->
  (s, Assign1 (s, e, x))
| Assign1 (s0, e0, x) ->
  branch
    let (z1, z2) = hexpr (s0, e0) in
    (s, Assign1 (z1, z2, x))
  or
    let (s1, v) = getRet_hexpr (e0) in
    let z3 = write (x, s1, v) in
    (s, Ret_hstmt z3)
  end
```

$$\frac{}{s, (e_1 \stackrel{?}{=} e_2) \longrightarrow s, ((s, e_1) \stackrel{?}{=}_1 e_2)}$$

$$\frac{s_0, e_0 \longrightarrow z_1, z_2}{s, ((s_0, e_0) \stackrel{?}{=}_1 e_2) \longrightarrow s, ((z_1, z_2) \stackrel{?}{=}_1 e_2)}$$

$$\frac{e_0 = (s_1, v_1)}{s, ((s_0, e_0) \stackrel{?}{=}_1 e_2) \longrightarrow s, (v_1 \stackrel{?}{=}_2 (s_1, e_2))}$$

$$\frac{s_0, e_0 \longrightarrow z_1, z_2}{s, (v_1 \stackrel{?}{=}_2 (s_0, e_0)) \longrightarrow s, (v_1 \stackrel{?}{=}_2 (z_1, z_2))}$$

$$\frac{e_0 = (s_2, v_2)}{s, (v_1 \stackrel{?}{=}_2 (s_0, e_0)) \longrightarrow s, (s_2, (v_1 \stackrel{?}{=} v_2))}$$

$$\frac{}{s, i \longrightarrow s, (s, \mathtt{intToVal}(i))}$$

$$\frac{}{s, (e_1 + e_2) \longrightarrow s, ((s, e_1) +_1 e_2)}$$

$$\frac{s_0, e_0 \longrightarrow z_1, z_2}{s, ((s_0, e_0) +_1 e_2) \longrightarrow s, ((z_1, z_2) +_1 e_2)}$$

$$\frac{e_0 = (s_1, v_1)}{s, ((s_0, e_0) +_1 e_2) \longrightarrow s, (v_1 +_2 (s_1, e_2))}$$

$$\frac{s_0, e_0 \longrightarrow z_1, z_2}{s, (v_1 +_2 (s_0, e_0)) \longrightarrow s, (v_1 +_2 (z_1, z_2))}$$

$$\frac{e_0 = (s_2, v_2)}{s, (v_1 +_2 (s_0, e_0)) \longrightarrow s, (s_2, (v_1 + v_2))}$$

$$\frac{}{s, x \longrightarrow s, (s, s(x))}$$

$$\frac{}{s, x := e \longrightarrow s, x :=_1 (s, e)}$$

$$\frac{s_0, e_0 \longrightarrow z_1, z_2}{s, x :=_1 (s_0, e_0) \longrightarrow s, x :=_1 (z_1, z_2)}$$

$$\frac{e_0 = (s_1, v)}{s, x :=_1 (s_0, e_0) \longrightarrow s, (s_1[x \mapsto v])}$$

```
| If (e1, t2, t3) ->
  (s, If1 (s, e1, t2, t3))
| If1 (s0, e0, t2, t3) ->
  branch
    let (z1, z2) = hexpr (s, e0) in
    (s, If1 (z1, z2, t2, t3))
  or
    let (s1, v) = getRet_hexpr (e0) in
    branch
      let () = isTrue (v) in
      (s1, t2)
    or
      let () = isFalse (v) in
      (s1, t3)
    end
  end
| Seq (t1, t2) ->
  (s, Seq1 (s, t1, t2))
| Seq1 (s0, t0, t2) ->
  branch
    let (z1, z2) = hstmt (s0, t0) in
    (s, Seq1 (z1, z2, t2))
  or
    let s1 = getRet_hstmt (t0) in
    (s1, 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 (s1, v) = getRet_hexpr (e0) in
    branch
      let () = isTrue (v) in
      (s, While2 (s1, t2, e1, t2))
    or
      let () = isFalse (v) in
      (s, Ret_hstmt s1)
    end
  end
| While2 (s0, t0, e1, t2) ->
  branch
    let (z1, z2) = hstmt (s0, t0) in
    (s, While2 (z1, z2, e1, t2))
  or
    let s2 = getRet_hstmt (t0) in
    (s2, While (e1, t2))
  end
```

$$\frac{}{s, \mathtt{If}(e_1, t_2, t_3) \longrightarrow s, \mathtt{If1}(s, e_1, t_2, t_3)}$$

$$\frac{s_0, e_0 \longrightarrow z_1, z_2}{s, \mathtt{If1}(s_0, e_0, t_2, t_3) \longrightarrow s, \mathtt{If1}(z_1, z_2, t_2, t_3)}$$

$$\frac{e_0 = (s_1, v) \qquad \mathtt{isTrue}(v)}{s, \mathtt{If1}(s_0, e_0, t_2, t_3) \longrightarrow s_1, t_2}$$

$$\frac{e_0 = (s_1, v) \qquad \mathtt{isFalse}(v)}{s, \mathtt{If1}(s_0, e_0, t_2, t_3) \longrightarrow s_1, t_3}$$

$$\frac{}{s, t_1; t_2 \longrightarrow s, (s, t_1)\ ;_1\ t_2}$$

$$\frac{s_0, t_0 \longrightarrow z_1, z_2}{s, (s_0, t_0)\ ;_1\ t_2 \longrightarrow s, (z_1, z_2)\ ;_1\ t_2}$$

$$\frac{t_0 = (s_1)}{s, (s_0, t_0)\ ;_1\ t_2 \longrightarrow s_1, t_2}$$

$$\frac{}{s, \mathtt{Skip} \longrightarrow s, (s)}$$

$$\frac{}{s, \mathtt{While}(e_1, t_2) \longrightarrow s, \mathtt{While1}(s, e_1, e_1, t_2)}$$

$$\frac{s_0, e_0 \longrightarrow z_1, z_2}{s, \mathtt{While1}(s_0, e_0, e_1, t_2) \longrightarrow s, \mathtt{While1}(z_1, z_2, e_1, t_2)}$$

$$\frac{e_0 = (s_1, v) \qquad \mathtt{isTrue}(v)}{s, \mathtt{While1}(s_0, e_0, e_1, t_2) \longrightarrow s, \mathtt{While2}(s_1, t_2, e_1, t_2)}$$

$$\frac{e_0 = (s_1, v) \qquad \mathtt{isFalse}(v)}{s, \mathtt{While1}(s_0, e_0, e_1, t_2) \longrightarrow s, (s_1)}$$

$$\frac{s_0, t_0 \longrightarrow z_1, z_2}{s, \mathtt{While2}(s_0, t_0, e_1, t_2) \longrightarrow s, \mathtt{While2}(z_1, z_2, e_1, t_2)}$$

$$\frac{t_0 = (s_2)}{s, \mathtt{While2}(s_0, t_0, e_1, t_2) \longrightarrow s_2, \mathtt{While}(e_1, t_2)}$$

$$
\begin{array}{rcll}
[\texttt{Branch}(S_1,..,S_n)]^{pr,V}_{L,b} & \triangleq & \texttt{Branch}([S_1]^{pr,V}_{L,b},..,[S_n]^{pr,V}_{L,b}) & \\
[\texttt{Filter}\ f\ \widetilde{t}]^{pr,V}_{L,b} & \triangleq & \texttt{Filter}\ f\ \widetilde{t} & \\
[\texttt{Return}\ \widetilde{t}]^{pr,V}_{L,b} & \triangleq & \texttt{Return}\ \widetilde{t} & \\
[\texttt{Proc}\ p\ \widetilde{t}]^{pr,V}_{L,b} & \triangleq & \texttt{Proc Tail}\ p\ \widetilde{t} & \text{if } L=\top, pr=p \\
[\texttt{Proc}\ p\ \widetilde{w}]^{pr,V}_{L,b} & \triangleq & \texttt{Proc Reuse}\ p\ \widetilde{w} & \text{if } b=\top, \widetilde{w}\in V \\
[\texttt{Proc}\ p\ \widetilde{t}]^{pr,V}_{L,b} & \triangleq & \texttt{Proc}\ (\texttt{New}\, c)\ p\ \widetilde{t} & \text{otherwise, } c \text{ fresh} \\
[\texttt{let}\ \widetilde{v}=K\ \texttt{in}\ S]^{pr,V}_{L,b} & \triangleq & \texttt{let}\ \widetilde{v}=[K]^{pr,V}_{\bot,b}\ \texttt{in}\ [S]^{pr,V}_{L,\bot} & \text{if } K\neq \texttt{Proc}\ p\ \widetilde{t} \\
[\texttt{let}\ \widetilde{v}=\texttt{Proc}\ p\ \widetilde{w}\ \texttt{in}\ S]^{pr,V}_{L,b} & \triangleq & \texttt{let}\ \widetilde{v}=\texttt{Proc Reuse}\ p\ \widetilde{w}\ \texttt{in}\ [S]^{pr,V}_{L,b} & \text{if } b=\top, \widetilde{w}\in V \\
[\texttt{let}\ \widetilde{v}=\texttt{Proc}\ p\ \widetilde{t}\ \texttt{in}\ S]^{pr,V}_{L,b} & \triangleq & \texttt{let}\ \widetilde{v}=\texttt{Proc}\ (\texttt{New}\, c)\ p\ \widetilde{t}\ \texttt{in}\ [S]^{pr,V}_{L,b} & \text{otherwise, } c \text{ fresh}
\end{array}
$$

Figure 6: Proc Calls Analysis

B TRANSFORMATION DETAILS

B.1 Initial Semantics

The initial big-step skeletal semantics is given as $(T_b, T_g, C_0, F, P_0, \mathrm{R_{BS}}, \texttt{ctype}, \texttt{ftype}, \texttt{ptype})$, where the set T of types is the union of base types T_b and program types T_g. The transformation modifies or completes these elements. Note that `ctype` and `ptype` are slightly modified, but we keep the same name to simplify the notations.

B.2 Analysis

The first phase is the analysis described in Section 4.1. The full rules are given in Figure 6. This analysis is applied to every skeleton of the semantics.

$$\mathrm{R_{lbl}} = \{p(\widetilde{y}, c(\widetilde{x})) := [S]^{p,\texttt{SglUse}(S)}_{\top,\top} \mid (p(\widetilde{y}, c(\widetilde{x})) := S) \in \mathrm{R_{BS}}\}$$

Where $\texttt{SglUse}(S)$ is the set of variables that are used exactly once in S.

B.3 Generation

After the analysis, we process every proc call annotated with a fresh constructor c to compute the type and rule of c. For this, we need to decompose skeletons into a proc call and a context representing the continuation.

The operation $[\![S]\!]^r_E$ inductively goes through S, building the continuation in its parameter E and returning the set of new rules. Assuming r of the form $\texttt{pr}(\widetilde{y}, c_r(\widetilde{x_r})) := S_r$, we define it as follows:

$$
\begin{array}{rcl}
[\![\texttt{Branch}\ (S_1,..,S_n)]\!]^r_E & \triangleq & [\![S_1]\!]^r_E \cup \ldots \cup [\![S_n]\!]^r_E \\
[\![\texttt{Filter}\ f\ \widetilde{t}]\!]^r_E = [\![\texttt{Return}\ \widetilde{t}]\!]^r_E & \triangleq & \emptyset \\
[\![\texttt{Proc Reuse}\ p\ \widetilde{t}]\!]^r_E = [\![\texttt{Proc Tail}\ p\ \widetilde{t}]\!]^r_E & \triangleq & \emptyset \\
[\![\texttt{Proc}\ (\texttt{New}\, c)\ p\ \widetilde{t}]\!]^r_E & \triangleq & \{\texttt{pr}(\widetilde{y}, c(\widetilde{w}, \widetilde{z})) := E[\texttt{Proc Reuse}\ p\ \widetilde{w}]\} \\
 & & \text{where } \widetilde{w} \text{ are fresh and } \widetilde{z} = (\mathrm{fv}(E) \setminus \widetilde{y}) \\
[\![\texttt{let}\ \widetilde{v}=K\ \texttt{in}\ S]\!]^r_E & \triangleq & [\![K]\!]^r_{<[\cdot]\mid\widetilde{v}\mid E[S]>} \cup [\![S]\!]^r_E
\end{array}
$$

where $\mathrm{fv}(E)$ are the free variables of E, defined as expected (see Definition 1 in Appendix C).

We update the semantics by adding c, its typing, and its rule. As explained in Section 4.1, the variables $\widetilde{w}$ can be deduced, and the new skeleton can reuse the fresh constructor so we change the annotation.

The generation phase is applied to every rule in $\mathrm{R_{lbl}}$, resulting in a new set $C_1 \supseteq C_0$ of constructors and a new set $\mathrm{R_{gen}} \supseteq \mathrm{R_{lbl}}$ of rules. The output of this phase is the extended semantics $(T_b, T_g, C_1, F, P_0, \mathrm{R_{gen}}, \texttt{ctype}, \texttt{ftype}, \texttt{ptype})$.

$$
\begin{aligned}
&\text{Assuming } r = (\mathtt{pr}(\widetilde{y}, c_r(\widetilde{x})) := S_r),\\
\|\ \mathtt{Branch}\ (S_1, .., S_n)\ \|^r_\sigma &\triangleq \mathtt{Branch}\ (\|\ S_1\ \|^r_\sigma, .., \|\ S_n\ \|^r_\sigma)\\
\|\ \mathtt{let}\ \widetilde{v} = \mathtt{Return}\ \widetilde{t}\ \ \mathtt{in}\ \ S\ \|^r_\sigma &\triangleq \mathtt{let}\ \widetilde{v} = \mathtt{Return}\ \widetilde{t}\ \ \mathtt{in}\ \ \|\ S\ \|^r_\sigma\\
\|\ \mathtt{let}\ \widetilde{v} = \mathtt{Filter}\ f\ \widetilde{t}\ \ \mathtt{in}\ \ S\ \|^r_\sigma &\triangleq \mathtt{let}\ \widetilde{v} = \mathtt{Filter}\ f\ \widetilde{t}\ \ \mathtt{in}\ \ \|\ S\ \|^r_\sigma\\
\|\ \mathtt{Return}\ \widetilde{t}\ \|^r_\sigma &\triangleq \mathtt{Return}\ (\widetilde{y}, \mathtt{Ret\_pr}(\widetilde{t}))\\
\|\ \mathtt{Proc\ Tail\ pr}\ \widetilde{t}\ \|^r_\sigma &\triangleq \mathtt{Return}\ \widetilde{t}\\
\|\ \mathtt{let}\ \widetilde{v} = \mathtt{Proc}\ (\mathtt{New}\, c)\ p\ \widetilde{t}\ \ \mathtt{in}\ \ S\ \|^r_\sigma &\triangleq \mathtt{Return}\ (\widetilde{y}, c(\widetilde{t}, \widetilde{z_c}))\\
&\qquad \text{where } (\mathtt{pr}(\widetilde{y}, c(\widetilde{w_c}, \widetilde{z_c})) := S_c) \in \mathrm{R}_{\mathrm{dist}}\\
\|\ \mathtt{let}\ \widetilde{v} = \mathtt{Proc\ Reuse}\ p\ (\widetilde{w'}, w)\ \ \mathtt{in}\ \ S\ \|^r_\sigma &\triangleq \mathtt{Branch}(S_1, S_2) \quad \text{where}
\end{aligned}
$$

$$
S_1 = \mathtt{let}\ \widetilde{z} = \mathtt{Proc}\ p\ \widetilde{w}\ \ \mathtt{in\ Return}\ (\widetilde{y}, c_r(\widetilde{x}))(\sigma\,; \widetilde{[z/w]}) \qquad \widetilde{z}\ \text{fresh}\ ;\ \widetilde{w} = (\widetilde{w'}, w)
$$

$$
S_2 = \mathtt{let}\ \widetilde{v} = \mathtt{Proc\ \ getRet\_p}\ (w)\ \ \mathtt{in}\ \ \|\ S\ \|^r_{\sigma\,;\,[\mathtt{Ret\_p}(\widetilde{v})/w]}
$$

Figure 7: Small-Step Transformation of a Skeleton

B.4 Distribute Branchings

This phase is described in Section 4.2. The distribution of `LetIn` over `Branch`, noted $\lceil S \rfloor$ is defined as follows.

$$
\begin{aligned}
\lceil \mathtt{Branch}(S_1, .., S_n) \rfloor &\triangleq \mathtt{Branch}(\lceil S_1 \rfloor, .., \lceil S_n \rfloor)\\
\lceil K \rfloor &\triangleq K \qquad\qquad \text{if } K \neq \mathtt{Branch}(\ldots)\\
\lceil \mathtt{let}\ \widetilde{v} = \mathtt{Branch}(S_1, .., S_n)\ \ \mathtt{in}\ \ S \rfloor &\triangleq \mathtt{Branch}(\lceil < S_1 \mid \widetilde{v} \mid S > \rfloor, .., \lceil < S_n \mid \widetilde{v} \mid S > \rfloor)\\
\lceil \mathtt{let}\ \widetilde{v} = K\ \ \mathtt{in}\ \ S \rfloor &\triangleq \mathtt{let}\ \widetilde{v} = K\ \ \mathtt{in}\ \lceil S \rfloor \qquad \text{if } K \neq \mathtt{Branch}(\ldots)
\end{aligned}
$$

As mentioned, this forces proc calls to be at the toplevel of the program. This way, the next phase is able to modify proc calls to immediately return and stop the computation, which is not possible in nested computations.

We apply this operation to every skeleton of our semantics:

$$
\mathrm{R}_{\mathrm{dist}} = \{\mathtt{p}(\widetilde{y}, c(\widetilde{x})) := \lceil S \rfloor \mid (\mathtt{p}(\widetilde{y}, c(\widetilde{x})) := S) \in \mathrm{R}_{\mathrm{gen}}\}
$$

B.5 Small-Step

Finally, the last phase of the transformation produces the coercions and the small-step rules.

$$
\forall p \in P_0 \text{ with } \mathtt{ptype}(p) = ((\widetilde{s}, s_g), \widetilde{u}),\ \text{we redefine: } \mathtt{ptype}(p) = ((\widetilde{s}, s_g), (\widetilde{s}, s_g))
$$

$$
C_2 = C_1 \cup \{\mathtt{Ret\_p} \mid \mathtt{p} \in P_0\} \qquad\qquad P_{\mathtt{getRet}} = \{\mathtt{getRet\_p} \mid \mathtt{p} \in P_0\}
$$

$$
\mathrm{R}_{\mathrm{getRet}} = \{\mathtt{getRet\_p}(\mathtt{Ret\_p}\ \widetilde{v}) := \mathtt{Return}\ \widetilde{v} \mid \mathtt{p} \in P_0 \wedge \|\mathtt{ptype}_{\mathtt{out}}(\mathtt{p})\| = \|\widetilde{v}\|\}
$$

The full set of cases for the final phase is given in Figure 7. It is applied to the rules defining the procedures of P_0.

$$
\mathrm{R}_{\mathrm{SS}} = \{\mathtt{p}(\widetilde{y}, c(\widetilde{x})) := \|\ S\ \|^{\mathtt{p}(\widetilde{y}, c(\widetilde{x})) := S}_{\epsilon} \mid (\mathtt{p}(\widetilde{y}, c(\widetilde{x})) := S) \in \mathrm{R}_{\mathrm{dist}}\} \cup \mathrm{R}_{\mathrm{getRet}}
$$

In the end, we obtain the semantics $(T_b, T_g, C_2, F, P_0 \cup P_{\mathtt{getRet}}, \mathrm{R}_{\mathrm{SS}}, \mathtt{ctype}, \mathtt{ftype}, \mathtt{ptype})$ where every $\mathtt{p} \in P_0$ makes a single step of computation.

$$\frac{\Sigma(\widetilde{t}) \Downarrow_p \widetilde{b}}{\Sigma \vdash \texttt{Proc}\ p\ \widetilde{t} \Downarrow \widetilde{b}} \qquad \frac{\mathcal{R}_f(\widetilde{\Sigma(t)}) \Downarrow \widetilde{b}}{\Sigma \vdash \texttt{Filter}\ f\ \widetilde{t} \Downarrow \widetilde{b}} \qquad \frac{\widetilde{\Sigma(t)} = \widetilde{b}}{\Sigma \vdash \texttt{Return}\ \widetilde{t} \Downarrow \widetilde{b}} \qquad \frac{S_i \in \widetilde{S} \qquad \Sigma \vdash S_i \Downarrow \widetilde{b}}{\Sigma \vdash \texttt{Branch}\ \widetilde{S} \Downarrow \widetilde{b}}$$

$$\frac{\Sigma \vdash K \Downarrow \widetilde{a} \qquad \Sigma + \widetilde{\{v \mapsto a\}} \vdash S \Downarrow \widetilde{b}}{\Sigma \vdash \texttt{let}\ \widetilde{v} = K\ \texttt{in}\ S \Downarrow \widetilde{b}} \qquad \frac{p(\widetilde{y}, c(\widetilde{x})) := S \in R \qquad \widetilde{\{y \mapsto a\}} + \widetilde{\{x \mapsto a'\}} \vdash S \Downarrow \widetilde{b}}{(\widetilde{a}, c(\widetilde{a'})) \Downarrow_p \widetilde{b}}$$

Figure 8: Inductive Interpretation

$$\frac{\Sigma(\widetilde{t}) \Uparrow_p}{\Sigma \vdash \texttt{Proc}\ p\ \widetilde{t} \Uparrow}\ \textsc{Div-Pc} \qquad \frac{S_i \in \widetilde{S} \qquad \Sigma \vdash S_i \Uparrow}{\Sigma \vdash \texttt{Branch}\ \widetilde{S} \Uparrow}\ \textsc{Div-Br} \qquad \frac{\Sigma \vdash K \Uparrow}{\Sigma \vdash \texttt{let}\ \widetilde{v} = K\ \texttt{in}\ S \Uparrow}\ \textsc{Div-LetL}$$

$$\frac{p(\widetilde{y}, c(\widetilde{x})) := S \in R \qquad \widetilde{\{y \mapsto a\}} + \widetilde{\{x \mapsto a'\}} \vdash S \Uparrow}{(\widetilde{a}, c(\widetilde{a'})) \Uparrow_p}\ \textsc{Div-tuple} \qquad \frac{\Sigma \vdash K \Downarrow \widetilde{a} \qquad \Sigma + \widetilde{\{v \mapsto a\}} \vdash S \Uparrow}{\Sigma \vdash \texttt{let}\ \widetilde{v} = K\ \texttt{in}\ S \Uparrow}\ \textsc{Div-LetR}$$

Figure 9: Coinductive Interpretation

$$\begin{aligned}
&\text{Assuming } r = (\texttt{pr}(\widetilde{y}, c_r(\widetilde{x})) := S_r),\\
\|\ \texttt{Branch}\ (S_1, \ldots, S_n)\ \|^r &\triangleq \texttt{Branch}\ (\|\ S_1\ \|^r, \ldots, \|\ S_n\ \|^r)\\
\|\ \texttt{let}\ \widetilde{v} = \texttt{Return}\ \widetilde{t}\ \texttt{in}\ S\ \|^r &\triangleq \texttt{let}\ \widetilde{v} = \texttt{Return}\ \widetilde{t}\ \texttt{in}\ \|\ S\ \|^r\\
\|\ \texttt{let}\ \widetilde{v} = \texttt{Filter}\ f\ \widetilde{t}\ \texttt{in}\ S\ \|^r &\triangleq \texttt{let}\ \widetilde{v} = \texttt{Filter}\ f\ \widetilde{t}\ \texttt{in}\ \|\ S\ \|^r\\
\|\ \texttt{Return}\ \widetilde{t}\ \|^r &\triangleq \texttt{Return}\ (\widetilde{y}, \texttt{Ret_pr}(\widetilde{t}))\\
\|\ \texttt{let}\ \widetilde{v} = \texttt{Proc}\ (\texttt{New}\, c)\ p\ \widetilde{t}\ \texttt{in}\ S\ \|^r &\triangleq \texttt{Return}\ (\widetilde{y}, c(\widetilde{t}, \widetilde{z_c}))\\
&\quad \text{where } (\texttt{pr}(\widetilde{y}, c(\widetilde{w_c}, \widetilde{z_c})) := S_c) \in \mathrm{R}_{\text{dist}}\\
\|\ \texttt{let}\ \widetilde{v} = \texttt{Proc Reuse}\ p\ (\widetilde{w'}, w)\ \texttt{in}\ S\ \|^r &\triangleq \texttt{Branch}(S_1, S_2) \quad \text{where}\\
S_1 = \texttt{let}\ \widetilde{z} = \texttt{Proc}\ p\ \widetilde{w}\ \texttt{in Return}\ (\widetilde{y}, c_r(\widetilde{x}))\widetilde{[z/w]} &\qquad \widetilde{z} \text{ fresh; } \widetilde{w} = (\widetilde{w'}, w)\\
S_2 = \texttt{let}\ \widetilde{v} = \texttt{Proc getRet_p}\ (w)\ \texttt{in}\ \|\ S\ \|^r &
\end{aligned}$$

Figure 10: Small-Step Transformation of a Skeleton without Analysis

C EQUIVALENCE PROOF

C.1 Definitions and Proof Structure

We present a pen-and-paper proof that the transformation of Section 4 is correct, in the simplified case where we systematically create new constructors.

We recall the rules for the inductive (Figure 8) and coinductive (Figure 9) interpretations of skeletal semantics.

The skeletal semantics as well as the different phases of the transformation are presented in the accompanying paper, and completed in Appendix B. We only redefine the last phase of the transformation (Figure 10), as it can be simplified to not use substitutions.

The following definitions are not explicited in the paper.

Definition 1. We note $\mathrm{fv}(S)$ the free variables of a tuple/term/skelement/skeleton/context.

$$\begin{aligned}
\mathrm{fv}((a_1, \ldots, a_n)) &\triangleq \mathrm{fv}(a_1) \cup \ldots \cup \mathrm{fv}(a_n) & \mathrm{fv}(v) &\triangleq \{v\}\\
\mathrm{fv}(c(\widetilde{t})) &\triangleq \mathrm{fv}(\widetilde{t}) & \mathrm{fv}(\texttt{Filter}\ f\ (\widetilde{t})) &\triangleq \mathrm{fv}(\widetilde{t})\\
\mathrm{fv}(\texttt{Proc}\ p\ (\widetilde{t}, t)) &\triangleq \mathrm{fv}(\widetilde{t}) \cup \mathrm{fv}(t) & \mathrm{fv}(\texttt{Return}\ (\widetilde{t})) &\triangleq \mathrm{fv}(\widetilde{t})\\
\mathrm{fv}(\texttt{Branch}\ (\widetilde{S})) &\triangleq \mathrm{fv}(\widetilde{S}) & \mathrm{fv}(\texttt{let}\ \widetilde{v} = K\ \texttt{in}\ S) &\triangleq (\mathrm{fv}(S) \setminus \{\widetilde{v}\}) \cup \mathrm{fv}(K)\\
\mathrm{fv}([\cdot]) &\triangleq \emptyset & \mathrm{fv}(< [\cdot] \mid \widetilde{v} \mid S >) &\triangleq \mathrm{fv}(S) \setminus \{\widetilde{v}\}
\end{aligned}$$

Definition 2. We note $\mathrm{bv}(S)$ the variables defined in a skeleton.

$$\begin{aligned}
\mathrm{bv}(\texttt{Filter}\ f\ (\widetilde{t})) &\triangleq \emptyset \\
\mathrm{bv}(\texttt{Proc}\ p\ (\widetilde{t}, t)) &\triangleq \emptyset \\
\mathrm{bv}(\texttt{Return}\ (\widetilde{t})) &\triangleq \emptyset \\
\mathrm{bv}(\texttt{Branch}\ (S_1, \ldots, S_n)) &\triangleq \mathrm{bv}(S_1) \cup \ldots \cup \mathrm{bv}(S_n) \\
\mathrm{bv}(\texttt{let}\ \widetilde{v} = K\ \texttt{in}\ S) &\triangleq \{\widetilde{v}\} \cup \mathrm{bv}(S)
\end{aligned}$$

Definition 3. We note $\mathrm{SSA}(S)$ the statement that a skeleton S does not reuse variables names. This can be seen as a Static Single Assignment form.

$\mathrm{SSA}(S) \triangleq \mathrm{NoRedef}(S) \wedge \mathrm{fv}(S) \cap \mathrm{bv}(S) = \emptyset$

$$\begin{aligned}
\mathrm{NoRedef}(\texttt{Filter}\ f\ (\widetilde{t})) &\triangleq \top \\
\mathrm{NoRedef}(\texttt{Proc}\ p\ (\widetilde{t}, t)) &\triangleq \top \\
\mathrm{NoRedef}(\texttt{Return}\ (\widetilde{t})) &\triangleq \top \\
\mathrm{NoRedef}(\texttt{Branch}\ (S_1, \ldots, S_n)) &\triangleq \forall i, \mathrm{NoRedef}(S_i) \\
\mathrm{NoRedef}(\texttt{let}\ \widetilde{v} = K\ \texttt{in}\ S) &\triangleq \begin{cases} \mathrm{bv}(K) \cap \{\widetilde{v}\} = \emptyset \\ \mathrm{bv}(S) \cap \{\widetilde{v}\} = \emptyset \\ \mathrm{bv}(K) \cap \mathrm{bv}(S) = \emptyset \\ \mathrm{NoRedef}(K) \\ \mathrm{NoRedef}(S) \end{cases}
\end{aligned}$$

Our transformation is meant to apply to such skeletons. If need be, a first transformation putting skeletons in SSA form can be necessary. Note that the naming "SSA" is an abuse, as parallel branchings are allowed to define the same variables. However, every execution of the skeleton follows a single branch and never overwrites the content of a variable.

Definition 4. We note $S_1 \in S_2$ when S_1 is a subskeleton of S_2. This property is the reflexive transitive closure of the following rules.

$$\begin{aligned}
K &\in \texttt{let}\ \widetilde{v} = K\ \texttt{in}\ S \\
S &\in \texttt{let}\ \widetilde{v} = K\ \texttt{in}\ S \\
S_i &\in \texttt{Branch}\ (S_1, \ldots, S_n)
\end{aligned}$$

Definition 5. We note $S_1 \triangleleft S_2$ when S_1 is a tail subskeleton of S_2. This property is the reflexive transitive closure of the following two rules.

$$\begin{aligned}
S &\triangleleft \texttt{let}\ \widetilde{v} = K\ \texttt{in}\ S \\
S_i &\triangleleft \texttt{Branch}\ (S_1, \ldots, S_n)
\end{aligned}$$

This definition does not consider subskeletons inside subevaluations.
For instance, $S_i \not\triangleleft \texttt{let}\ \widetilde{v} = \texttt{Branch}\ (S_1, \ldots, S_n)\ \texttt{in}\ S'$.

The proof uses the sets of rules at the different phases of the transformation to state and certify results. As explained in Section 5, we additionally introduce a set $\mathrm{R_{EBS}}$, corresponding to an extended big-step semantics, to simplify the proof strategy. We recall the notations for the different sets and the semantics they represent.

- $\mathrm{R_{BS}}$: initial rules, input of the transformation;
- $\mathrm{R_{gen}}$: rules after generation of new constructors and rules;
- $\mathrm{R_{dist}}$: rules after distributing branchings;
- $\mathrm{R_{EBS}}$: similar to $\mathrm{R_{dist}}$ (after delaying returns), with rules for `Ret_p` constructors;

$$\mathrm{R_{EBS}} \triangleq \mathrm{R_{dist}} \cup \{p(\widetilde{y}, \texttt{Ret\_p}(\widetilde{v})) := \texttt{Return}\ \widetilde{v} \mid p \in P_0\}$$

- $\mathrm{R_{SS}}$: result of the transformation after going small-step, with `getRet_p` procedures.

We use these sets to specify the notations for derivation judgments. We write for instance $\Sigma \vdash S \Downarrow^{\mathrm{R_{BS}}} \widetilde{b}$ and $\widetilde{a} \Downarrow_p^{\mathrm{R_{BS}}} \widetilde{b}$ for the concrete evaluation, and $\Sigma \vdash S \Uparrow^{\mathrm{R_{BS}}}$ and $\widetilde{a} \Uparrow_p^{\mathrm{R_{BS}}}$ for the divergence of the initial big-step semantics.

Definition 6. We note $\widetilde{a} \overset{\infty}{\rightarrow}_p$ the divergence in the small-step skeletal semantics. It is defined coinductively with the following rule.

$$\frac{\widetilde{a} \Downarrow_p^{\mathrm{R_{SS}}} \widetilde{b} \qquad \widetilde{b} \overset{\infty}{\rightarrow}_p}{\widetilde{a} \overset{\infty}{\rightarrow}_p}$$

Other equivalent definitions include:

- $\widetilde{a} \xrightarrow{\infty}_p$ is the maximal predicate Q satisfying the following property.

$$Q(\widetilde{a}, p) \quad \Longrightarrow \quad \exists \widetilde{b},\ \widetilde{a} \Downarrow_p^{\text{RSS}} \widetilde{b} \ \wedge\ Q(\widetilde{b}, p)$$

- Let $F(X) = \{(\widetilde{a}, p) \mid \exists \widetilde{b},\ \widetilde{a} \Downarrow_p^{\text{RSS}} \widetilde{b} \wedge (\widetilde{b}, p) \in X\}$, we have:

$$\xrightarrow{\infty} = \{(\widetilde{a}, p) \mid \widetilde{a} \xrightarrow{\infty}_p\} \ \triangleq\ \nu X.F(X)$$

We also recall notations from the transformation phases (Appendix B). We use the following.

- $< S_1 \mid \widetilde{x} \mid S_2 >$ for the monadic bind of two skeletons;
- $[\![S]\!]_E^r$ for the generation of new constructors, skeletons, and big-step rules (after adding a "$\texttt{New}\, c$" annotation to every proc call);
- $\lceil S \rfloor$ for the distribution of branchings;
- $\| S \|^r$ for the final small-step transformation, redefined with simpler rules in Figure 10.

A first section (C.2) covers simple lemmas about our different definitions, unconnected to the different rule sets. Afterwards, we check that our SSA property is preserved throughout the transformation (C.3). Then we certify important properties of the different phases of the transformation, to make sure they behave as intended (C.4). Using these results, we finally prove the equivalences between the different semantics. Section C.5 covers the equivalence between Big-Step and Extended Big-Step. Section C.6 certifies that an EBS evaluation/divergence implies a small-step (in)finite sequence. Finally, Section C.7 proves the reverse direction: a small-step (in)finite reduction sequence implies an EBS evaluation/divergence.

C.2 Basic Lemmas

The simple proofs of the first few trivial lemmas are omitted.

Lemma 7. For all S_1, $\widetilde{v}$, S_2, $\widetilde{w}$, and S_3,

$$<< S_1 \mid \widetilde{v} \mid S_2 >\mid \widetilde{w} \mid S_3 > = < S_1 \mid \widetilde{v} \mid< S_2 \mid \widetilde{w} \mid S_3 >>$$

Lemma 8. For all Σ_1, Σ_2, and Σ_3,

$$(\Sigma_1 + \Sigma_2) + \Sigma_3 = \Sigma_1 + (\Sigma_2 + \Sigma_3)$$

Lemma 9. For all Σ_1, Σ_2, and Σ_3,

- $\forall x, \quad x \notin (\mathrm{dom}(\Sigma_2) \setminus \mathrm{dom}(\Sigma_3)) \implies (\Sigma_1 + \Sigma_3)(x) = (\Sigma_1 + \Sigma_2 + \Sigma_3)(x)$
- $\forall t, \quad (\mathrm{dom}(\Sigma_2) \setminus \mathrm{dom}(\Sigma_3)) \cap \mathrm{fv}(t) = \emptyset \implies (\Sigma_1 + \Sigma_3)(t) = (\Sigma_1 + \Sigma_2 + \Sigma_3)(t)$
- $\forall \widetilde{t}, \quad (\mathrm{dom}(\Sigma_2) \setminus \mathrm{dom}(\Sigma_3)) \cap \mathrm{fv}(\widetilde{t}) = \emptyset \implies (\Sigma_1 + \Sigma_3)(\widetilde{t}) = (\Sigma_1 + \Sigma_2 + \Sigma_3)(\widetilde{t})$

Lemma 10. For all S, Σ_1, Σ_2, Σ_3, and $\widetilde{a}$, if $(\mathrm{dom}(\Sigma_2) \setminus \mathrm{dom}(\Sigma_3)) \cap \mathrm{fv}(S) = \emptyset$, then

$$\Sigma_1 + \Sigma_3 \vdash S \Downarrow \widetilde{a} \iff \Sigma_1 + \Sigma_2 + \Sigma_3 \vdash S \Downarrow \widetilde{a}$$

Proof. By structural induction on S.

- If $S = \texttt{Return}\ (\widetilde{t})$
  $\Sigma_1 + \Sigma_3 \vdash \texttt{Return}\ (\widetilde{t}) \Downarrow \widetilde{a}$
  $\iff (\widetilde{\Sigma_1 + \Sigma_3)(t)} = \widetilde{a}$
  $\iff (\widetilde{\Sigma_1 + \Sigma_2 + \Sigma_3)(t)} = \widetilde{a}$ using Lemma 9 (because $\mathrm{fv}(\widetilde{t}) = \mathrm{fv}(S)$)
  $\iff \Sigma_1 + \Sigma_2 + \Sigma_3 \vdash \texttt{Return}\ (\widetilde{t}) \Downarrow \widetilde{a}$
- If S is of the form $\texttt{Filter}\ f\ (\widetilde{t})$ or $\texttt{Proc}\ p\ (\widetilde{t}, t)$, similarly
- If $S = \texttt{Branch}\ (\widetilde{S})$
  $\Sigma_1 + \Sigma_3 \vdash \texttt{Branch}\ (\widetilde{S}) \Downarrow \widetilde{a}$
  $\iff \exists S_i \in \widetilde{S}, \Sigma_1 + \Sigma_3 \vdash S_i \Downarrow \widetilde{a}$
  $\iff \exists S_i \in \widetilde{S}, \Sigma_1 + \Sigma_2 + \Sigma_3 \vdash S_i \Downarrow \widetilde{a}$ using induction hypothesis
  $\iff \Sigma_1 + \Sigma_2 + \Sigma_3 \vdash \texttt{Branch}\ (\widetilde{S}) \Downarrow \widetilde{a}$
- If $S = (\texttt{let}\ \widetilde{v} = K\ \texttt{in}\ S')$
  $\Sigma_1 + \Sigma_3 \vdash \texttt{let}\ \widetilde{v} = K\ \texttt{in}\ S' \Downarrow \widetilde{a}$
  $\iff \exists \widetilde{b}, \begin{cases} \Sigma_1 + \Sigma_3 \vdash K \Downarrow \widetilde{b} \\ \Sigma_1 + \Sigma_3 + \widetilde{\{v \mapsto b\}} \vdash S' \Downarrow \widetilde{a} \end{cases}$
  $\iff \exists \widetilde{b}, \begin{cases} \Sigma_1 + \Sigma_2 + \Sigma_3 \vdash K \Downarrow \widetilde{b} \\ \Sigma_1 + \Sigma_2 + (\Sigma_3 + \widetilde{\{v \mapsto b\}}) \vdash S' \Downarrow \widetilde{a} \end{cases}$ using IH twice and Lemma 8.
  $\mathrm{fv}(S') \subset \mathrm{fv}(S) \cup \{\widetilde{v}\}$, so $(\mathrm{dom}(\Sigma_2) \setminus (\mathrm{dom}(\Sigma_3) \cup \{\widetilde{v}\})) \cap \mathrm{fv}(S') = \emptyset$
  $\iff \Sigma_1 + \Sigma_2 + \Sigma_3 \vdash \texttt{let}\ \widetilde{v} = K\ \texttt{in}\ S' \Downarrow \widetilde{a}$

□

Lemma 11. For all S, Σ_1, Σ_2, and Σ_3, if $(\mathrm{dom}(\Sigma_2) \setminus \mathrm{dom}(\Sigma_3)) \cap \mathrm{fv}(S) = \emptyset$, then

$$\Sigma_1 + \Sigma_3 \vdash S \Uparrow \quad \Longleftrightarrow \quad \Sigma_1 + \Sigma_2 + \Sigma_3 \vdash S \Uparrow$$

Proof. This lemma is similar to Lemma 10, but for divergence. See Figure 9 for the coinductive definition of $\Sigma \vdash S \Uparrow$. The proof is done by a straightforward induction on S, and case analysis on the rule used. The only two places where the environment Σ is used are:

- in rule Div-Pc, where both environments agree on the mapping of the variables $\widetilde{t}$ by hypothesis.
- in the first leaf of rule Div-LetR, where we can use Lemma 10 to go from one environment to the other.

□

Lemma 12. For all S_1, x, S_2, and Σ, if $(\mathrm{bv}(S_1) \setminus \{\widetilde{x}\}) \cap \mathrm{fv}(S_2) = \emptyset$, then

$$\Sigma \vdash\!< S_1 \mid \widetilde{x} \mid S_2 >\!\Uparrow \iff \begin{array}{c} \Sigma \vdash S_1 \Uparrow \\ \text{OR} \\ \exists \widetilde{b}, \left\{ \begin{array}{c} \Sigma \vdash S_1 \Downarrow \widetilde{b} \\ \Sigma + \widetilde{\{x \mapsto b\}} \vdash S_2 \Uparrow \end{array} \right. \end{array}$$

I.e., $< S_1 \mid \widetilde{x} \mid S_2 >$ diverges if either S_1 or S_2 diverges.

Proof. This lemma is similar to Lemma 13, but for divergence. It is also done by structural induction on S_1.

If S_1 is a skelement K, then $< K \mid \widetilde{x} \mid S_2 >$ is defined as $\texttt{let}\ \widetilde{x} = K\ \texttt{in}\ S_2$ and the result comes directly from the definition (see rules Div-LetR and Div-LetL of Figure 9).

If $S_1 = (\texttt{let}\ \widetilde{v} = K\ \texttt{in}\ S')$, then $< S_1 \mid \widetilde{x} \mid S_2 >$ is defined as $\texttt{let}\ \widetilde{v} = K\ \texttt{in}\ \ < S' \mid \widetilde{x} \mid S_2 >$. Then:

$$\begin{array}{ll} & \Sigma \vdash\!< S_1 \mid \widetilde{x} \mid S_2 >\!\Uparrow \\ \Longleftrightarrow & \begin{array}{c} \Sigma \vdash K \Uparrow \\ \text{OR} \\ \exists \widetilde{c}, \left\{ \begin{array}{c} \Sigma \vdash K \Downarrow \widetilde{c} \\ \Sigma + \widetilde{\{v \mapsto c\}} \vdash\!< S' \mid \widetilde{x} \mid S_2 >\!\Uparrow \end{array} \right. \end{array} & \text{by either rule Div-LetL or Div-LetR.} \\ \Longleftrightarrow & \begin{array}{c} \Sigma \vdash K \Uparrow \\ \text{OR} \\ \Sigma \vdash K \Downarrow \widetilde{c} \\ \exists \widetilde{c}, \left\{ \Sigma + \widetilde{\{v \mapsto c\}} \vdash S' \Uparrow \ \text{OR} \ \exists \widetilde{b}, \left\{ \Sigma + \widetilde{\{v \mapsto c\}} \vdash S' \Downarrow \widetilde{b} \right. \right. \\ \Sigma + \widetilde{\{v \mapsto c\}} + \widetilde{\{x \mapsto b\}} \vdash S_2 \Uparrow \end{array} & \text{by IH} \\ \Longleftrightarrow & \begin{array}{c} \Sigma \vdash S_1 \Uparrow \\ \text{OR} \\ \exists \widetilde{b}, \left\{ \begin{array}{c} \Sigma \vdash S_1 \Downarrow \widetilde{b} \\ \Sigma + \widetilde{\{v \mapsto c\}} + \widetilde{\{x \mapsto b\}} \vdash S_2 \Uparrow \end{array} \right. \end{array} & \text{by definitions} \\ \Longleftrightarrow & \begin{array}{c} \Sigma \vdash S_1 \Uparrow \\ \text{OR} \\ \exists \widetilde{b}, \left\{ \begin{array}{c} \Sigma \vdash S_1 \Downarrow \widetilde{b} \\ \Sigma + \widetilde{\{x \mapsto b\}} \vdash S_2 \Uparrow \end{array} \right. \end{array} & \text{by Lemma 11} \end{array}$$

$\{\widetilde{v}\} \subset \mathrm{bv}(S_1)$, so our hypothesis implies $(\{\widetilde{v}\} \setminus \{\widetilde{x}\}) \cap \mathrm{fv}(S_2) = \emptyset$, enough to use Lemma 11. □

Lemma 13. For all S_1, x, S_2, Σ, and $\widetilde{a}$, if $(\mathrm{bv}(S_1) \setminus \{\widetilde{x}\}) \cap \mathrm{fv}(S_2) = \emptyset$, then

$$\Sigma \vdash\!< S_1 \mid \widetilde{x} \mid S_2 >\!\Downarrow \widetilde{a} \iff \exists \widetilde{b}, \left\{ \begin{array}{c} \Sigma \vdash S_1 \Downarrow \widetilde{b} \\ \Sigma + \widetilde{\{x \mapsto b\}} \vdash S_2 \Downarrow \widetilde{a} \end{array} \right.$$

Proof. By structural induction on S_1.

If S_1 is a skelement K, then $< K \mid \widetilde{x} \mid S_2 >$ is defined as $\texttt{let}\ \widetilde{x} = K\ \texttt{in}\ S_2$ and the result comes directly from the definition (see Figure 8).

If S_1 is of the form $\texttt{let}\ \widetilde{v} = K\ \texttt{in}\ S'$, then $< S_1 \mid \widetilde{x} \mid S_2 >$ is defined as $\texttt{let}\ \widetilde{v} = K\ \texttt{in}\ \ < S' \mid \widetilde{x} \mid S_2 >$, and we have the following.

$$\begin{array}{ll} & \Sigma \vdash\!< S_1 \mid \widetilde{x} \mid S_2 >\!\Downarrow \widetilde{a} \\ \Longleftrightarrow & \exists \widetilde{c}, \left\{ \begin{array}{c} \Sigma \vdash K \Downarrow \widetilde{c} \\ \Sigma + \widetilde{\{v \mapsto c\}} \vdash\!< S' \mid \widetilde{x} \mid S_2 >\!\Downarrow \widetilde{a} \end{array} \right. & \text{by definition} \\ \Longleftrightarrow & \exists \widetilde{c}, \left\{ \begin{array}{c} \Sigma \vdash K \Downarrow \widetilde{c} \\ \exists \widetilde{b}, \left\{ \begin{array}{c} \Sigma + \widetilde{\{v \mapsto c\}} \vdash S' \Downarrow \widetilde{b} \\ \Sigma + \widetilde{\{v \mapsto c\}} + \widetilde{\{x \mapsto b\}} \vdash S_2 \Downarrow \widetilde{a} \end{array} \right. \end{array} \right. & \text{by induction hypothesis} \end{array}$$

$$\iff \exists \widetilde{c}, \widetilde{b}, \begin{cases} \Sigma \vdash K \Downarrow \widetilde{c} \\ \Sigma + \widetilde{\{v \mapsto c\}} \vdash S' \Downarrow \widetilde{b} \text{ by Lemma 10} \\ \Sigma + \widetilde{\{x \mapsto b\}} \vdash S_2 \Downarrow \widetilde{a} \end{cases}$$

$\{\widetilde{v}\} \subset \text{bv}(S_1)$, so our hypothesis implies $(\{\widetilde{v}\} \setminus \{\widetilde{x}\}) \cap \text{fv}(S_2) = \emptyset$

$$\iff \exists \widetilde{b}, \begin{cases} \Sigma \vdash \texttt{let}\ \widetilde{v} = K\ \texttt{in}\ S' \Downarrow \widetilde{b} \\ \Sigma + \widetilde{\{x \mapsto b\}} \vdash S_2 \Downarrow \widetilde{a} \end{cases} \text{ by definition}$$

□

Lemma 14. For all $S_1, \widetilde{v}$, and S_2, $\lceil S_2 \rfloor \triangleleft \lceil < S_1 \mid \widetilde{v} \mid S_2 > \rfloor$

Proof. By induction on S_1. Note that this lemma holds only because we assume branchings to always contain at least one skeleton.

- If $S_1 = \texttt{Branch}\ (S'_1, \ldots, S'_n)$, then $\lceil < S_1 \mid \widetilde{v} \mid S_2 > \rfloor = \texttt{Branch}\ (\lceil < S'_1 \mid \widetilde{v} \mid S_2 > \rfloor, \ldots.)$ and we have our result by applying our induction hypothesis on S'_1.
- If $S_1 = K$ is not a branching, then $\lceil < S_1 \mid \widetilde{v} \mid S_2 > \rfloor = (\texttt{let}\ \widetilde{v} = K\ \texttt{in}\ \lceil S_2 \rfloor)$ and we immediately have our result.
- If $S_1 = (\texttt{let}\ \widetilde{w} = \texttt{Branch}\ (S'_1, \ldots, S'_n)\ \texttt{in}\ S'_0)$, then
  $\lceil < S_1 \mid \widetilde{v} \mid S_2 > \rfloor = \lceil \texttt{let}\ \widetilde{w} = \texttt{Branch}\ (S'_1, \ldots, S'_n)\ \texttt{in}\ < S'_0 \mid \widetilde{v} \mid S_2 > \rfloor = \texttt{Branch}\ (\lceil < S'_1 \mid \widetilde{w} \mid < S'_0 \mid \widetilde{v} \mid S_2 >> \rfloor, \ldots).$
  We first use our induction hypothesis on S'_1, giving $\lceil < S'_0 \mid \widetilde{v} \mid S_2 > \rfloor \triangleleft \lceil < S_1 \mid \widetilde{v} \mid S_2 > \rfloor$. Then we have our result by applying our induction hypothesis a second time with S'_0.
- If $S_1 = (\texttt{let}\ \widetilde{w} = K\ \texttt{in}\ S'_0)$ where K is not a branching, then $\lceil < S_1 \mid \widetilde{v} \mid S_2 > \rfloor = (\texttt{let}\ \widetilde{w} = K\ \texttt{in}\ \lceil < S'_0 \mid \widetilde{v} \mid S_2 > \rfloor)$ and we have our result from our induction hypothesis on S'_0.

□

Lemma 15. Forall $S_1, \widetilde{v}, S_2, E$, and K_P, where K_P is not a branching, if
$E[K_P] \triangleleft \lceil < S_1 \mid \widetilde{v} \mid S_2 > \rfloor$ and $K_P \notin S_1$, then $E[K_P] \triangleleft \lceil S_2 \rfloor$.

Proof. By structural induction on S_1. All cases are straightforward. The only interesting case is when $S_1 = (\texttt{let}\ \widetilde{w} = \texttt{Branch}(S'_1, \ldots, S'_n)\ \texttt{in}\ S')$ as we need to use our induction hypothesis twice.

In this case, by definition: $\lceil < S_1 \mid \widetilde{v} \mid S_2 > \rfloor \triangleq \texttt{Branch}(\lceil < S'_1 \mid \widetilde{w} \mid < S' \mid \widetilde{v} \mid S_2 >> \rfloor, \ldots, \lceil < S'_n \mid \widetilde{w} \mid < S' \mid \widetilde{v} \mid S_2 >> \rfloor)$. From $E[K_P] \triangleleft \lceil < S_1 \mid \widetilde{v} \mid S_2 > \rfloor$, there is an S'_i such that $E[K_P] \triangleleft \lceil < S'_i \mid \widetilde{w} \mid < S' \mid \widetilde{v} \mid S_2 >> \rfloor$. We use our induction hypothesis a first time knowing that $K_P \notin S'_i$, and we get $E[K_P] \triangleleft \lceil < S' \mid \widetilde{v} \mid S_2 > \rfloor$. Then we use our induction hypothesis a second time, knowing $K_P \notin S'$, to get the desired result.

□

C.3 SSA

Lemma 16. Forall $S' \in S$, if $\text{SSA}(S)$ then $\text{SSA}(S')$.

Proof. By simply following Definitions 3 and 4. □

Lemma 17. For all $S_1, S_2, \widetilde{v}$, if $\text{SSA}(\texttt{let}\ \widetilde{v} = \texttt{Branch}(S_1)\ \texttt{in}\ S_2)$, then:

- $\text{SSA}(< S_1 \mid \widetilde{v} \mid S_2 >)$
- $\text{bv}(\texttt{let}\ \widetilde{v} = \texttt{Branch}(S_1)\ \texttt{in}\ S_2) = \text{bv}(< S_1 \mid \widetilde{v} \mid S_2 >)$
- $\text{fv}(\texttt{let}\ \widetilde{v} = \texttt{Branch}(S_1)\ \texttt{in}\ S_2) = \text{fv}(< S_1 \mid \widetilde{v} \mid S_2 >)$

Remark 17.1. Note that the reverse of the first point is not true. See for instance $S_1 = (\texttt{let}\ y = K\ \texttt{in}\ S)$ and $S_2 = \texttt{Return}\ y$.

Proof. First, by unfolding definitions, note that we always have:

- $\text{NoRedef}(\texttt{Branch}(S_1)) \iff \text{NoRedef}(S_1)$
- $\text{bv}(\texttt{Branch}(S_1)) = \text{bv}(S_1)$
- $\text{fv}(\texttt{Branch}(S_1)) = \text{fv}(S_1)$

We proceed by structural induction on S_1. If $S_1 = K$ is a skelement, then $< S_1 \mid \widetilde{v} \mid S_2 > = (\texttt{let}\ \widetilde{v} = K\ \texttt{in}\ S_2)$. We can easily unfold the definitions and check the desired results hold, using the three points above.

Else S_1 is of the form $(\texttt{let}\ \widetilde{w} = K\ \texttt{in}\ S)$.

Cutting our hypothesis gives us: $\begin{cases} \text{fv}(\ldots) \cap \text{bv}(\ldots) = \emptyset \\ \quad \text{bv}(S1) \cap \widetilde{v} = \emptyset \\ \quad \text{bv}(S2) \cap \widetilde{v} = \emptyset \\ \text{bv}(S1) \cap \text{bv}(S2) = \emptyset \\ \quad \text{NoRedef}(S1) \\ \quad \text{NoRedef}(S2) \end{cases}$

$$\text{We have } \mathrm{bv}(S_1) = \widetilde{w} \cup \mathrm{bv}(K) \cup \mathrm{bv}(S) \text{ so: } \begin{cases} \mathrm{fv}(\ldots) \cap \mathrm{bv}(\ldots) = \emptyset \\ (\widetilde{w} \cup \mathrm{bv}(K) \cup \mathrm{bv}(S)) \cap \widetilde{v} = \emptyset \\ \mathrm{bv}(S2) \cap \widetilde{v} = \emptyset \\ (\widetilde{w} \cup \mathrm{bv}(K) \cup \mathrm{bv}(S)) \cap \mathrm{bv}(S2) = \emptyset \\ \begin{cases} \mathrm{bv}(K) \cap \widetilde{w} = \emptyset \\ \mathrm{bv}(S) \cap \widetilde{w} = \emptyset \\ \mathrm{bv}(K) \cap \mathrm{bv}(S) = \emptyset \\ \mathrm{NoRedef}(K) \\ \mathrm{NoRedef}(S) \end{cases} \\ \mathrm{NoRedef}(S2) \end{cases}$$

$$\text{We can reformulate into: } \begin{cases} \mathrm{fv}(\ldots) \cap \mathrm{bv}(\ldots) = \emptyset \\ \text{elements of } \{\widetilde{v}; \widetilde{w}; \mathrm{bv}(K); \mathrm{bv}(S1); \mathrm{bv}(S2)\} \text{ are 2 by 2 disjoints} \\ \mathrm{NoRedef}(K) \\ \mathrm{NoRedef}(S) \\ \mathrm{NoRedef}(S2) \end{cases}$$

This is enough to first prove $\mathrm{SSA}(\texttt{let } \widetilde{w} = K \texttt{ in let } \widetilde{v} = \texttt{Branch}(S) \texttt{ in } S_2)$ by checking the definition. Using our induction hypothesis, we have:

- $\mathrm{SSA}(< S \mid \widetilde{v} \mid S_2 >)$
- $\mathrm{bv}(< S \mid \widetilde{v} \mid S_2 >) = \widetilde{v} \cup \mathrm{bv}(S) \cup \mathrm{bv}(S_2)$
- $\mathrm{fv}(< S \mid \widetilde{v} \mid S_2 >) = \mathrm{fv}(S) \cup (\mathrm{fv}(S_2) \setminus \widetilde{v}) = \mathrm{fv}(S) \cup \mathrm{fv}(S_2)$

Then, we have enough pieces to prove our desired result, once again by directly checking the definition:

- $\mathrm{SSA}(\texttt{let } \widetilde{w} = K \texttt{ in } < S \mid \widetilde{v} \mid S_2 >)$
- $\mathrm{bv}(\texttt{let } \widetilde{w} = K \texttt{ in } < S \mid \widetilde{v} \mid S_2 >) = \widetilde{v} \cup \widetilde{w} \cup \mathrm{bv}(K) \cup \mathrm{bv}(S) \cup \mathrm{bv}(S_2)$
- $\mathrm{fv}(\texttt{let } \widetilde{w} = K \texttt{ in } < S \mid \widetilde{v} \mid S_2 >) = \mathrm{fv}(K) \cup \mathrm{fv}(S) \cup \mathrm{fv}(S_2)$

□

Lemma 18. SSA form is preserved during the generation phase creating new skeletons. I.e.:

- If $\mathrm{SSA}(E[\texttt{Branch } (S_1, \ldots, S_n)])$ then $\mathrm{SSA}(E[S_i])$.
- If $\mathrm{SSA}(E[\texttt{let } \widetilde{v} = K \texttt{ in } S])$ then $\mathrm{SSA}(E[S])$.
- If $\mathrm{SSA}(E[\texttt{let } \widetilde{v} = K \texttt{ in } S])$ then $\mathrm{SSA}(< [\cdot] \mid \widetilde{v} \mid E[S] > [K])$.

Proof. If $E = [\cdot]$, the three points simplify to:

- If $\mathrm{SSA}(\texttt{Branch } (S_1, \ldots, S_n))$ then $\mathrm{SSA}(S_i)$.
- If $\mathrm{SSA}(\texttt{let } \widetilde{v} = K \texttt{ in } S)$ then $\mathrm{SSA}(S)$.
- If $\mathrm{SSA}(\texttt{let } \widetilde{v} = K \texttt{ in } S)$ then $\mathrm{SSA}(\texttt{let } \widetilde{v} = K \texttt{ in } S)$.

The third case is trivial, the other two are solved using Lemma 16.

Else $E =< [\cdot] \mid \widetilde{w} \mid S_E >$ for some $\widetilde{w}$ and S_E. The three points simplify to:

- If $\mathrm{SSA}(\texttt{let } \widetilde{w} = \texttt{Branch } (S_1, \ldots, S_n) \texttt{ in } S_E)$ then $\mathrm{SSA}(< S_i \mid \widetilde{w} \mid S_E >)$.
- If $\mathrm{SSA}(\texttt{let } \widetilde{v} = K \texttt{ in } < S \mid \widetilde{w} \mid S_E >)$ then $\mathrm{SSA}(< S \mid \widetilde{w} \mid S_E >)$.
- If $\mathrm{SSA}(\texttt{let } \widetilde{v} = K \texttt{ in } < S \mid \widetilde{w} \mid S_E >)$ then $\mathrm{SSA}(\texttt{let } \widetilde{v} = K \texttt{ in } < S \mid \widetilde{w} \mid S_E >)$.

Once again, the third case is trivial, and the second can be solved using Lemma 16. The remaining (first) point can be cut in two halves, by proving the intermediate result $\mathrm{SSA}(\texttt{let } \widetilde{w} = \texttt{Branch } (S_i) \texttt{ in } S_E)$. The first half can be checked directly by following Definition 3. The second half is proved separately (Lemma 17). □

Lemma 19. The distribution of branchings preserves SSA form.

I.e., for all skeleton S, if $\mathrm{SSA}(S)$, then $\mathrm{SSA}(\lceil S \rfloor)$.

Proof. For this, we prove the following by structural induction on S.

$$\forall S, \mathrm{SSA}(S) \implies \begin{cases} \mathrm{SSA}(\lceil S \rfloor) \\ \mathrm{bv}(S) = \mathrm{bv}(\lceil S \rfloor) \\ \mathrm{fv}(S) = \mathrm{fv}(\lceil S \rfloor) \end{cases}$$

Most cases are straightforward. The only interesting case is when S is of the form $\texttt{let } \widetilde{v} = \texttt{Branch}(S_1, \ldots, S_n) \texttt{ in } S'$. Then by definition $\lceil S \rfloor \triangleq \texttt{Branch}(\lceil < S_1 \mid \widetilde{v} \mid S' > \rfloor, \ldots, \lceil < S_n \mid \widetilde{v} \mid S' > \rfloor)$.

First, we check by following Definition 3 that we have $\mathrm{SSA}(\texttt{let } \widetilde{v} = \texttt{Branch}(S_i) \texttt{ in } S')$ for each i. Then, using Lemma 17, we get:

- $\mathrm{SSA}(< S_i \mid \widetilde{v} \mid S' >)$

- $\widetilde{v} \cup \mathrm{bv}(S_i) \cup \mathrm{bv}(S') = \mathrm{bv}(< S_i \mid \widetilde{v} \mid S' >)$
- $\mathrm{fv}(S_i) \cup \mathrm{fv}(S') = \mathrm{fv}(< S_i \mid \widetilde{v} \mid S' >)$

And we can conclude:

- $\mathrm{bv}(\lceil S \rfloor) = \bigcup_i \mathrm{bv}(< S_i \mid \widetilde{v} \mid S' >) = \bigcup_i (\widetilde{v} \cup \mathrm{bv}(S_i) \cup \mathrm{bv}(S')) = (\bigcup_i \mathrm{bv}(S_i)) \cup (\widetilde{v} \cup \mathrm{bv}(S')) = \mathrm{bv}(S)$
- $\mathrm{fv}(\lceil S \rfloor) = \bigcup_i \mathrm{fv}(< S_i \mid \widetilde{v} \mid S' >) = \bigcup_i (\mathrm{fv}(S_i) \cup \mathrm{fv}(S')) = (\bigcup_i \mathrm{fv}(S_i)) \cup \mathrm{fv}(S') = \mathrm{fv}(S)$
- $\mathrm{SSA}(\lceil S \rfloor)$ by definition, using $\mathrm{bv}(S) \cap \mathrm{fv}(S) = \emptyset$ and $\mathrm{SSA}(< S_i \mid \widetilde{v} \mid S' >)$

□

Lemma 20. If all initial skeletons are in SSA form, then the new skeletons created during the generation phase respect the SSA conditions. I.e., if for all rule $(p(\widetilde{y}, c(\widetilde{z})) \coloneqq S) \in \mathrm{R_{BS}}$ we initially have $\mathrm{SSA}(S)$, then for all rule $(p(\widetilde{y}, c(\widetilde{z})) \coloneqq S) \in \mathrm{R_{gen}}$ we also have $\mathrm{SSA}(S)$.

Proof. Initial rules are left untouched, so we only need to check $\mathrm{SSA}(S)$ for new skeletons, constructed from a generation operation $[\![\texttt{Proc (New}\, c)\ p\ \widetilde{t}]\!]^r_E$. The important part of the proof is to check $\mathrm{SSA}(E[\texttt{Proc}\ p\ \widetilde{t}])$, as the new skeleton is then $E[\texttt{Proc}\ p\ \widetilde{w}]$ where $\widetilde{w}$ are fresh variables, and the property trivially ensues.

For this, we check we have $\mathrm{SSA}(E[S])$ at every step $[\![S]\!]^r_E$. Initially, we start with a given skeleton S (from a rule of $\mathrm{R_{BS}}$, for which we now $\mathrm{SSA}(S)$) and an environment $E = [\cdot]$, so it holds. Then, the preservation of the property comes from Lemma 18. □

C.4 Properties of the Transformation Phases

Lemma 21. For all environment Σ, terms $\widetilde{a}$, and skeleton S such that $\mathrm{SSA}(S)$,

$$\Sigma \vdash S \Downarrow \widetilde{a} \iff \Sigma \vdash \lceil S \rfloor \Downarrow \widetilde{a}$$

I.e., the distribution of branchings preserves the concrete interpretation.

Proof. The proof is done by induction on the size of S, and dealing with the different forms S can take. Three of the four cases are trivial (see Section B.4 for the definition). The only interesting case is when S is of the form $\texttt{let}\ \widetilde{v} = \texttt{Branch}(S_1, \ldots, S_n)\ \texttt{in}\ S'$. Then, by definition, $\lceil S \rfloor \triangleq \texttt{Branch}(\lceil < S_1 \mid \widetilde{v} \mid S' > \rfloor, \ldots, \lceil < S_n \mid \widetilde{v} \mid S' > \rfloor)$.

$\Sigma \vdash S \Downarrow \widetilde{a}$

$\iff \exists \widetilde{b}, S_i, \begin{cases} \Sigma \vdash S_i \Downarrow \widetilde{b} \\ \Sigma + \widetilde{\{v \mapsto b\}} \vdash S' \Downarrow \widetilde{a} \end{cases}$ by definition (using two rules)

$\iff \exists S_i, \Sigma \vdash < S_i \mid \widetilde{v} \mid S' > \Downarrow \widetilde{a}$ using Lemma 13

We have $(\mathrm{bv}(S_i) \setminus \{\widetilde{v}\}) \cap \mathrm{fv}(S') = \emptyset$ from $\begin{cases} \mathrm{bv}(S) \cap \mathrm{fv}(S) = \emptyset \text{ from } \mathrm{SSA}(S) \\ \mathrm{fv}(S') \subset \mathrm{fv}(S) \cup \{\widetilde{v}\} \\ \mathrm{bv}(S_i) \subset \mathrm{bv}(S) \end{cases}$

$\iff \exists S_i, \Sigma \vdash \lceil < S_i \mid \widetilde{v} \mid S' > \rfloor \Downarrow \widetilde{a}$ using IH, since $< S_i \mid \widetilde{v} \mid S' >$ is smaller than S

$\iff \Sigma \vdash \lceil S \rfloor \Downarrow \widetilde{a}$

□

Lemma 22. For all environment Σ and skeleton S such that $\mathrm{SSA}(S)$,

$$\Sigma \vdash S \Uparrow \iff \Sigma \vdash \lceil S \rfloor \Uparrow$$

I.e., the distribution of branchings also preserves the coinductive semantics.

Proof. This lemma is similar to Lemma 21, and so is the proof. This is done by a straightforward induction on S. Instead of using Lemma 13, we use the similar Lemma 12 for divergence. Also, we directly use Lemma 21 instead of the induction hypothesis for finite subevaluations. □

Lemma 23. Delaying returns does not impact the inductive interpretation. Thus, the rules common to $\mathrm{R_{dist}}$ and $\mathrm{R_{EBS}}$ behave the same way.

Proof. This is done by induction on skeletons. The induction case is trivial, and the base case is to check that $\Sigma \vdash \texttt{Filter}\ f\ \widetilde{t} \Downarrow \widetilde{a} \iff \Sigma \vdash \texttt{let}\ \widetilde{w} = \texttt{Filter}\ f\ \widetilde{t}\ \texttt{in Return}\ \widetilde{w} \Downarrow \widetilde{a}$
(and similarly for procedures, with the same reasoning). This comes from the two following derivation trees having the same leaf $\mathcal{R}_f(\widetilde{\Sigma(t)}) \Downarrow \widetilde{a}$.

$$\dfrac{\mathcal{R}_f(\widetilde{\Sigma(t)}) \Downarrow \widetilde{a}}{\Sigma \vdash \texttt{Filter}\ f\ \widetilde{t} \Downarrow \widetilde{a}}$$

$$\dfrac{\dfrac{\mathcal{R}_f(\widetilde{\Sigma(t)}) \Downarrow \widetilde{a}}{\Sigma \vdash \texttt{Filter}\ f\ \widetilde{t} \Downarrow \widetilde{a}} \qquad \dfrac{(\Sigma + \widetilde{\{w \mapsto a\}})(\widetilde{w}) = \widetilde{a}}{\Sigma + \widetilde{\{w \mapsto a\}} \vdash \texttt{Return}\ \widetilde{w} \Downarrow^{\mathrm{REBS}} \widetilde{a}}}{\Sigma \vdash \texttt{let}\ \widetilde{w} = \texttt{Filter}\ f\ \widetilde{t}\ \texttt{in Return}\ \widetilde{w} \Downarrow^{\mathrm{REBS}} \widetilde{a}}$$

□

The following two lemmas show that the rules added to create $\mathrm{R_{EBS}}$ (from $\mathrm{R_{dist}}$) behave as intended.

Lemma 24. For all procedure p and terms $\widetilde{a}$, $\widetilde{b}$, and $\widetilde{d}$,

$$\widetilde{b} = \widetilde{d} \iff (\widetilde{a}, \texttt{Ret\_p}(\widetilde{b})) \Downarrow_p^{\mathrm{R_{EBS}}} \widetilde{d}$$

Proof. This comes from the following derivation tree, either by creating it or because there is only one possible inversion of the rules of Figure 8.

$$\frac{(\texttt{p}(\widetilde{y}, \texttt{Ret\_p}(\widetilde{x})) := \texttt{Return}\ \widetilde{x}) \in \mathrm{R_{EBS}} \qquad \dfrac{(\widetilde{\{y \mapsto a\}} + \widetilde{\{x \mapsto b\}})(\widetilde{x}) = \widetilde{b}}{\widetilde{\{y \mapsto a\}} + \widetilde{\{x \mapsto b\}} \vdash \texttt{Return}\ \widetilde{x} \Downarrow^{\mathrm{R_{EBS}}} \widetilde{b}}}{(\widetilde{a}, \texttt{Ret\_p}(\widetilde{b})) \Downarrow_p^{\mathrm{R_{EBS}}} \widetilde{b}}$$

□

Lemma 25. For all procedure p and terms $\widetilde{b}$ and $\widetilde{d}$,

$$\widetilde{b} = \widetilde{d} \iff \texttt{Ret\_p}(\widetilde{b}) \Downarrow_{\texttt{getRet\_p}}^{\mathrm{R_{SS}}} \widetilde{d}$$

Proof. This comes from the following derivation tree, either by creating it or because there is only one possible inversion of the rules of Figure 8.

$$\frac{(\texttt{getRet\_p}(\texttt{Ret\_p}(\widetilde{x})) := \texttt{Return}\ \widetilde{x}) \in \mathrm{R_{SS}} \qquad \dfrac{\widetilde{\{x \mapsto b\}}(\widetilde{x}) = \widetilde{b}}{\widetilde{\{x \mapsto b\}} \vdash \texttt{Return}\ \widetilde{x} \Downarrow^{\mathrm{R_{SS}}} \widetilde{b}}}{\texttt{Ret\_p}(\widetilde{b}) \Downarrow_{\texttt{getRet\_p}}^{\mathrm{R_{SS}}} \widetilde{b}}$$

□

The following two lemmas show the small-step semantics exhibits the usual congruence rules for proc calls labeled `Reuse`.

Lemma 26. If $(p(\widetilde{y}, c(\widetilde{w}, \widetilde{z})) := \texttt{let}\ \widetilde{v} = \texttt{Proc Reuse}\ p_1\ \widetilde{w}\ \texttt{in}\ S) \in \mathrm{R_{EBS}}$, then

$$(\widetilde{e}, e_0) \Downarrow_{p_1}^{\mathrm{R_{SS}}} (\widetilde{e}, e_0') \implies (\widetilde{a}, c((\widetilde{e}, e_0), \widetilde{a'})) \Downarrow_p^{\mathrm{R_{SS}}} (\widetilde{a}, c((\widetilde{e}, e_0'), \widetilde{a'}))$$

Proof. Let r be the rule in the formulation of the lemma.

$$\frac{(\ldots := \| \ldots \|^r) \in \mathrm{R_{SS}} \qquad \dfrac{\dfrac{(\widetilde{e}, e_0) \Downarrow_{p_1}^{\mathrm{R_{SS}}} (\widetilde{e}, e_0') \qquad \dfrac{\Sigma'(\widetilde{y}, c(\widetilde{u}, \widetilde{z})) = (\widetilde{a}, c((\widetilde{e}, e_0'), \widetilde{a'}))}{\Sigma' \vdash \texttt{Return}\ (\widetilde{y}, c(\widetilde{u}, \widetilde{z})) \Downarrow^{\mathrm{R_{SS}}} (\widetilde{a}, c((\widetilde{e}, e_0'), \widetilde{a'}))}}{\Sigma \vdash \texttt{let}\ \widetilde{u} = \texttt{Proc}\ p_1\ \widetilde{w}\ \texttt{in Return}\ (\widetilde{y}, c(\widetilde{u}, \widetilde{z})) \Downarrow^{\mathrm{R_{SS}}} (\widetilde{a}, c((\widetilde{e}, e_0'), \widetilde{a'}))}}{\Sigma \vdash \| \texttt{let}\ \widetilde{v} = \texttt{Proc Reuse}\ p_1\ \widetilde{w}\ \texttt{in}\ S \|^r \Downarrow^{\mathrm{R_{SS}}} (\widetilde{a}, c((\widetilde{e}, e_0'), \widetilde{a'}))}}{(\widetilde{a}, c((\widetilde{e}, e_0), \widetilde{a'})) \Downarrow_p^{\mathrm{R_{SS}}} (\widetilde{a}, c((\widetilde{e}, e_0'), \widetilde{a'}))}$$

where $\Sigma = \widetilde{\{y \mapsto a\}} + \{\widetilde{w} \mapsto (\widetilde{e}, e_0)\} + \widetilde{\{z \mapsto a'\}}$ and $\Sigma' = \Sigma + \{\widetilde{u} \mapsto (\widetilde{e}, e_0')\}$. □

Lemma 27. If $(p(\widetilde{y}, c(\widetilde{w}, \widetilde{z})) := \texttt{let}\ \widetilde{v} = \texttt{Proc Reuse}\ p_1\ \widetilde{w}\ \texttt{in}\ S) \in \mathrm{R_{EBS}}$, and e_0 is not of the form $\texttt{Ret\_p1}(\ldots)$, then

$$(\widetilde{a}, c((\widetilde{e}, e_0), \widetilde{a'})) \Downarrow_p^{\mathrm{R_{SS}}} \widetilde{d} \implies \exists e_0', \begin{cases} \widetilde{d} = (\widetilde{a}, c((\widetilde{e}, e_0'), \widetilde{a'})) \\ \widetilde{e}, e_0 \Downarrow_{p_1}^{\mathrm{R_{SS}}} (\widetilde{e}, e_0') \end{cases}$$

This Lemma is the converse of Lemma 26 above.

Proof. The property holds because the only possible evaluation tree is the one presented in the proof of Lemma 26 above. Indeed $[\Sigma \vdash \texttt{let}\ \widetilde{v} = \texttt{Proc getRet\_p1}\ (w)\ \texttt{in}\ \| S \|^r \Downarrow^{\mathrm{R_{SS}}} \ldots]$, with $\Sigma(w) = e_0$, is not possible since `getRet_p1` only has a rule for the constructor `Ret_p1`. □

The following two lemmas show the small-step semantics can also extract coerced values and continue a computation, for proc calls labeled `Reuse` when the first computation is finished.

Lemma 28. If $(p(\widetilde{y}, c((\widetilde{w}, w), \widetilde{z})) := \texttt{let}\ \widetilde{v} = \texttt{Proc Reuse}\ p_1\ (\widetilde{w}, w)\ \texttt{in}\ S) \in \mathrm{R_{EBS}}$, then

$$\widetilde{\{y \mapsto a\}} + \widetilde{\{z \mapsto a'\}} + \widetilde{\{v \mapsto b\}} \vdash \| S \|^r \Downarrow^{\mathrm{R_{SS}}} \widetilde{d} \implies (\widetilde{a}, c((\widetilde{e}, \texttt{Ret\_p1}(\widetilde{b})), \widetilde{a'})) \Downarrow_p^{\mathrm{R_{SS}}} \widetilde{d}$$

Proof. Let r be the rule in the formulation of the lemma.

$$\frac{(\ldots := \| \ldots \|^r) \in \mathrm{R_{SS}} \quad \dfrac{\dfrac{\texttt{Ret\_p1}(\widetilde{b}) \Downarrow^{\mathrm{R_{SS}}}_{\texttt{getRet\_p1}} \widetilde{b} \qquad \overline{\Sigma' \vdash \| S \|^r \Downarrow^{\mathrm{R_{SS}}} \widetilde{d}}}{\Sigma \vdash \texttt{let}\ \widetilde{v} = \texttt{Proc getRet\_p1}\ (w)\ \texttt{in}\ \| S \|^r \Downarrow^{\mathrm{R_{SS}}} \widetilde{d}}}{\Sigma \vdash \| \texttt{let}\ \widetilde{v} = \texttt{Proc Reuse}\ p_1\ (\widetilde{w}, w)\ \texttt{in}\ S \|^r \Downarrow^{\mathrm{R_{SS}}} \widetilde{d}}}{(\widetilde{a}, c((\widetilde{e}, \texttt{Ret\_p1}(\widetilde{b})), \widetilde{a'})) \Downarrow^{\mathrm{R_{SS}}}_p \widetilde{d}}$$

With $\Sigma = \widetilde{\{y \mapsto a\}} + \widetilde{\{w \mapsto e\}} + \{w \mapsto \texttt{Ret\_p1}(\widetilde{b})\} + \widetilde{\{z \mapsto a'\}}$ and $\Sigma' = \Sigma + \widetilde{\{v \mapsto b\}}$. The first leaf uses Lemma 25. The second leaf uses Lemma 10 to go from $\widetilde{\{y \mapsto a\}} + \widetilde{\{z \mapsto a'\}} + \widetilde{\{v \mapsto b\}}$ to Σ' since the variables $(\widetilde{w}, w)$ do not appear in S by construction. □

Lemma 29. If $(p(\widetilde{y}, c((\widetilde{w}, w), \widetilde{z})) := \texttt{let}\ \widetilde{v} = \texttt{Proc Reuse}\ p_1\ (\widetilde{w}, w)\ \texttt{in}\ S) \in \mathrm{R_{EBS}}$, then

$$(\widetilde{a}, c((\widetilde{e}, \texttt{Ret\_p1}(\widetilde{b})), \widetilde{a'})) \Downarrow^{\mathrm{R_{SS}}}_p \widetilde{d} \implies \widetilde{\{y \mapsto a\}} + \widetilde{\{z \mapsto a'\}} + \widetilde{\{v \mapsto b\}} \vdash \| S \|^r \Downarrow^{\mathrm{R_{SS}}} \widetilde{d}$$

This Lemma is the converse of Lemma 28 above.

Proof. The property holds because the only possible evaluation tree is the one presented in the proof of Lemma 28 above. Indeed $[\Sigma \vdash \texttt{let}\ \widetilde{u} = \texttt{Proc}\ p_1\ (\widetilde{w}, w)\ \texttt{in Return}\ (\widetilde{y}, c(\widetilde{u}, \widetilde{z})) \Downarrow^{\mathrm{R_{SS}}} \ldots]$, with $\Sigma(w) = \texttt{Ret\_p1}(\widetilde{b})$, is not possible since, in $\mathrm{R_{SS}}$, the procedure p_1 does not have a rule for the constructor `Ret_p1`. We also use Lemma 10 to simplify the environment. □

Lemma 30. For every rule $r = (p(\widetilde{y}, c(\widetilde{x})) := S) \in \mathrm{R_{BS}}$, during the generation phase with the skeleton S, every time we apply the generation operation on a subskeleton, i.e. $[\![S_0]\!]^r_E$, we have $\lceil E[S_0] \rfloor \triangleleft \lceil S \rfloor$.

Proof. The property trivially holds at the start ($[\![S]\!]^r_{[\cdot]}$) of the generation, since $\lceil S \rfloor \triangleleft \lceil S \rfloor$ by reflexivity. Now we check that we preserve this property during the generation, assuming we are at the point $[\![S_0]\!]^r_E$ with $\lceil E[S_0] \rfloor \triangleleft \lceil S \rfloor$.

▶ If S_0 is a skelement.

- If S_0 is not a branching, there is no recursive call to the generation operation and we are done.
- If $S_0 = \texttt{Branch}\ (S_1, \ldots, S_n)$, for each i we generate $[\![S_i]\!]^r_E$, and so we want to show $\lceil E[S_i] \rfloor \triangleleft \lceil S \rfloor$. Note that we necessarily have $\lceil E[S_0] \rfloor = \texttt{Branch}\ (\lceil E[S_1] \rfloor, \ldots, \lceil E[S_n] \rfloor)$. If $E = [\cdot]$, this is trivial. If E is of the form $< [\cdot] \mid \widetilde{v} \mid S' >$, we have:

$$\begin{aligned}\lceil E[S_0] \rfloor &= \lceil < \texttt{Branch}\ (S_1, \ldots, S_n) \mid \widetilde{v} \mid S' > \rfloor \\ &= \lceil \texttt{let}\ \widetilde{v} = \texttt{Branch}\ (S_1, \ldots, S_n)\ \texttt{in}\ S' \rfloor \\ &= \texttt{Branch}\ (\lceil < S_1 \mid \widetilde{v} \mid S' > \rfloor, \ldots, \lceil < S_n \mid \widetilde{v} \mid S' > \rfloor) \\ &= \texttt{Branch}\ (\lceil E[S_1] \rfloor, \ldots, \lceil E[S_n] \rfloor)\end{aligned}$$

  So by transitivity we have $\lceil E[S_i] \rfloor \triangleleft \lceil E[S_0] \rfloor \triangleleft \lceil S \rfloor$.

▶ If S_0 is of the form $\texttt{let}\ \widetilde{v} = K\ \texttt{in}\ S_1$. Now we need to check to the property for the two recursive calls $[\![K]\!]^r_{<[\cdot] \mid \widetilde{v} \mid E[S_1]>}$ and $[\![S_1]\!]^r_E$.

We notice that $E[S_0] = (\texttt{let}\ \widetilde{v} = K\ \texttt{in}\ E[S_1]) = (< [\cdot] \mid \widetilde{v} \mid E[S_1] >)[K]$, so the property holds for the first one from our assumption $\lceil E[S_0] \rfloor \triangleleft \lceil S \rfloor$.

We are left with checking the property for $[\![S_1]\!]^r_E$, i.e., we want to show that $\lceil E[S_1] \rfloor \triangleleft \lceil S \rfloor$.

- If we have $K = \texttt{Branch}\ (S'_1, \ldots, S'_n)$, then $\lceil E[S_0] \rfloor = \texttt{Branch}\ (\ldots, \lceil < S'_i \mid \widetilde{v} \mid E[S_1] > \rfloor, \ldots)$. Using Lemma 14, we conclude $\lceil E[S_1] \rfloor \triangleleft \lceil < S'_i \mid \widetilde{v} \mid E[S_1] > \rfloor \triangleleft \lceil E[S_0] \rfloor \triangleleft \lceil S \rfloor$.
- Lastly, if K is not a branching, we simply have $\lceil E[S_0] \rfloor = \texttt{let}\ \widetilde{v} = K\ \texttt{in}\ \lceil E[S_1] \rfloor$, so we also have $\lceil E[S_1] \rfloor \triangleleft \lceil E[S_0] \rfloor \triangleleft \lceil S \rfloor$ by definition.

□

The next two lemmas explain where the rules for new constructors come from. If after distribution, we have an initial skeleton with $E[K_P]$ (where $K_P = \texttt{Proc}\ (\texttt{New}\, c_0)\ p_1\ \widetilde{t}$) as its tail, then it comes from the distribution of a skeleton of the form $E'[K_P]$. The new rule for c_0 is the distribution of $E'[K_P]$, which results in $E[K_P]$ as well.

E.g., if the skeleton of `While` ends with $\texttt{let}\ \widetilde{v} = \texttt{Proc}\ (\texttt{New While2})\ p\ \widetilde{t}\ \texttt{in}\ S$ then the skeleton of `While2` is necessarily of the form $\texttt{let}\ \widetilde{v} = \texttt{Proc Reuse}\ p\ \widetilde{w}\ \texttt{in}\ S$.

Lemma 31. Forall $S_0, E, E_0, r, c_0, p_1, \widetilde{t}$, let $K_P \triangleq \texttt{Proc}\ (\texttt{New}\, c_0)\ p_1\ \widetilde{t}$.

If we generate $[\![S_0]\!]^r_{E_0}$, and know that $K_P \in S_0$, $K_P \notin E_0$, and that $E[K_P] \triangleleft \lceil E_0[S_0] \rfloor$, then there exists E' such that we also generate $[\![K_P]\!]^r_{E'}$ and have $\lceil E'[K_P] \rfloor = E[K_P]$.

Proof. Note that K_P appears exactly once in S_0, since K_P is annotated with a fresh constructor name c_0. We prove this lemma by induction on S_0.

- If $S_0 = \texttt{Branch}\ (S'_1, \ldots, S'_n)$, then there is exactly one S'_i that contains K_P and we generate $[\![S'_i]\!]^r_{E_0}$. Also $\lceil E_0[S_0]\rfloor = \texttt{Branch}\ (\lceil E_0[S'_1]\rfloor, \ldots, \lceil E_0[S'_n]\rfloor)$, so we can deduce that $E[K_P] \triangleleft \lceil E_0[S'_i]\rfloor$. By using our induction hypothesis with S'_i, we immediately get our desired result.
- If $S_0 = K$ is not a branching, then $K_P \in S_0$ implies that $S_0 = K_P$. We take $E' = E_0$ and need to check $\lceil E_0[K_P]\rfloor = E[K_P]$. We know that $E[K_P] \triangleleft \lceil E_0[K_P]\rfloor$. If $E_0 = [\cdot]$, then $E = [\cdot]$ and we have the equality. Else $\lceil E_0[K_P]\rfloor$ is of the form $\texttt{let}\ \ldots = K_P\ \texttt{in}\ \lceil \ldots \rfloor$, and since $K_P \notin E_0$, it does not appear in the "..." parts. Necessarily (from Definition 5), we have $E[K_P] = \lceil E_0[K_P]\rfloor$.
- If $S_0 = (\texttt{let}\ \widetilde{v} = \texttt{Branch}\ (S'_1, \ldots, S'_n)\ \texttt{in}\ S')$ and there is exactly one S'_i that contains K_P. Let $E_1 =< [\cdot] \mid \widetilde{v} \mid E_0[S'] >$, we generate $[\![S'_i]\!]^r_{E_1}$. Also $\lceil E_0[S_0]\rfloor = \texttt{Branch}\ (\lceil < S'_1 \mid \widetilde{v} \mid E_0[S'] >\rfloor, \ldots, \lceil < S'_n \mid \widetilde{v} \mid E_0[S'] >\rfloor)$, so we can deduce that $E[K_P] \triangleleft \lceil < S'_i \mid \widetilde{v} \mid E_0[S'] > \rfloor = \lceil E_1[S'_i]\rfloor$. We can use our induction hypothesis with E_1 and S'_i to immediately get our desired result.
- If $S_0 = (\texttt{let}\ \widetilde{v} = \texttt{Branch}\ (S'_1, \ldots, S'_n)\ \texttt{in}\ S')$ and $K_P \in S'$. Then K_P does not appear in any S'_i and we generate $[\![S']\!]^r_{E_0}$. Also $\lceil E_0[S_0]\rfloor = \texttt{Branch}\ (\lceil < S'_1 \mid \widetilde{v} \mid E_0[S'] >\rfloor, \ldots, \lceil < S'_n \mid \widetilde{v} \mid E_0[S'] >\rfloor)$. Since $E[K_P] \triangleleft \lceil E_0[S_0]\rfloor$, there is an S'_i such that $E[K_P] \triangleleft \lceil < S'_1 \mid \widetilde{v} \mid E_0[S'] >\rfloor$. We use Lemma 15 to get more precisely that $E[K_P] \triangleleft \lceil E_0[S']\rfloor$. Then we can use our induction hypothesis on E_0 and S' to get our desired result.
- If $S_0 = (\texttt{let}\ \widetilde{v} = K_P\ \texttt{in}\ S')$. let $E_1 =< [\cdot] \mid \widetilde{v} \mid E_0[S'] >$, we generate $[\![K_P]\!]^r_{E_1}$. We take $E' = E_1$, and need to show that $E[K_P]$ is equal to $\lceil E_1[K_P]\rfloor = (\texttt{let}\ \widetilde{v} = K_P\ \texttt{in}\ \lceil E_0[S']\rfloor)$. Once again, $E[K_P] \triangleleft \lceil E_1[K_P]\rfloor$ and K_P does not appear in $\lceil E_0[S']\rfloor$, so $E[K_P] \not\triangleleft \lceil E_0[S']\rfloor$ and we have the equality from the only possible case of Definition 5.
- If $S_0 = (\texttt{let}\ \widetilde{v} = K\ \texttt{in}\ S')$ where $K \neq K_P$ is not a branching, then $K_P \in S'$ and we generate $[\![S']\!]^r_{E_0}$. Also $\lceil E_0[S_0]\rfloor = (\texttt{let}\ \widetilde{v} = K\ \texttt{in}\ \lceil E_0[S']\rfloor)$. Thus $E[K_P] \triangleleft \lceil E_0[S']\rfloor$ and we have our desired result by using our induction hypothesis on E_0 and S'.

□

Lemma 32. For all rule $(p(\widetilde{y}, c(\widetilde{x})) := \lceil S\rfloor) \in \mathrm{R}_{\mathrm{dist}}$, if $E[\texttt{Proc}\ (\texttt{New}\, c_0)\ p_1\ \widetilde{t}] \triangleleft \lceil S\rfloor$, then $\mathrm{R}_{\mathrm{dist}}$ contains a rule of the form:

$$p(\widetilde{y}, c_0(\widetilde{w}, \widetilde{z})) := E[\texttt{Proc Reuse}\ p_1\ \widetilde{w}]$$

where $\widetilde{w}$ does not appear in $\mathrm{fv}(E)$.

Proof. Firstly, if c is not an initial constructor, we can recover the same kind of hypotheses with an initial constructor. In this case, our rule is of the form

$$p(\widetilde{y}, c(\widetilde{x_1}, \widetilde{x_2})) := E_S[\texttt{Proc Reuse}\ p_2\ \widetilde{x_1}]\ \in \mathrm{R}_{\mathrm{gen}}$$

with $\widetilde{x} = \widetilde{x_1}, \widetilde{x_2}$. So it has been created from a generation operation $[\![\texttt{Proc}\ (\texttt{New}\, c)\ p_2\ \widetilde{t'}]\!]^r_{E_S}$, with $r = (p(\widetilde{y}, c'(\widetilde{v})) := S_0) \in \mathrm{R}_{\mathrm{BS}}$ where c' is an initial constructor. Using Lemma 30, we have $\lceil S\rfloor = \lceil E_S[\texttt{Proc Reuse}\ p_2\ \widetilde{x_1}]\rfloor \triangleleft \lceil S_0\rfloor$. So by transitivity we recover the following hypothesis: $(p(\widetilde{y}, c'(\widetilde{v})) := \lceil S_0\rfloor) \in \mathrm{R}_{\mathrm{dist}}$ where c' is an initial constructor and $E[\texttt{Proc}\ (\texttt{New}\, c_0)\ p_1\ \widetilde{t}] \triangleleft \lceil S_0\rfloor$.

Now, we can assume without loss of generality that c is an initial constructor, and so that we have $r = (p(\widetilde{y}, c(\widetilde{x})) := S) \in \mathrm{R}_{\mathrm{BS}}$. We use Lemma 31 with $[\![S]\!]^r_{[\cdot]}$ to get E' such that we generate $[\![\texttt{Proc}\ (\texttt{New}\, c_0)\ p_1\ \widetilde{t}]\!]^r_{E'}$ and have $\lceil E'[\texttt{Proc}\ (\texttt{New}\, c_0)\ p_1\ \widetilde{t}]\rfloor = E[\texttt{Proc}\ (\texttt{New}\, c_0)\ p_1\ \widetilde{t}]$. Thus the creation of new skeletons produces a rule $(p(\widetilde{y}, c_0(\widetilde{w}, \widetilde{z})) := E'[\texttt{Proc Reuse}\ p_1\ \widetilde{w}]) \in \mathrm{R}_{\mathrm{gen}}$ with fresh variables $\widetilde{w}$. And after distributing branchings we have

$$(p(\widetilde{y}, c_0(\widetilde{w}, \widetilde{z})) := E[\texttt{Proc Reuse}\ p_1\ \widetilde{w}]) \in \mathrm{R}_{\mathrm{dist}}$$

□

Lemma 33. $\forall (p(\widetilde{y}, c(\widetilde{x})) := S) \in \mathrm{R}_{\mathrm{EBS}}$, if $(\texttt{let}\ \widetilde{v} = \texttt{Proc}\ (\texttt{New}\, c_0)\ p_1\ \widetilde{t}\ \texttt{in}\ S_0) \triangleleft S$, then $\mathrm{R}_{\mathrm{EBS}}$ contains a rule of the form:

$$p(\widetilde{y}, c_0(\widetilde{w}, \widetilde{z})) := \texttt{let}\ \widetilde{v} = \texttt{Proc Reuse}\ p_1\ \widetilde{w}\ \texttt{in}\ S_0$$

where $\widetilde{w}$ does not appear in $\mathrm{fv}(S_0)$.

Proof. Directly from Lemma 32, as we go from $\mathrm{R}_{\mathrm{dist}}$ to $\mathrm{R}_{\mathrm{EBS}}$ by only delaying returns at the end of skeletons. This operation does the same modifications on both rules (we assume it selects the same fresh variables for simplicity), and does not create new free variables. □

C.5 Big-Step and Extended Big-Step

The following three theorems state the equivalence between the initial and extended big-step semantics. For this, we note $|t|$ the canonical injection of a term t into the extended semantics. We show that the behavior is preserved when we limit ourselves to initial terms, i.e., to terms using only constructors in C_0 (see Appendix B), and not newly created constructors (e.g., `While1` or `Ret_h`).

Theorem 34 (BS⇒EBS)**.** For all p and initial terms $\widetilde{a}$, and $\widetilde{b}$,

$$\widetilde{a} \Downarrow_p^{\mathrm{R}_{\mathrm{BS}}} \widetilde{b} \implies \widetilde{|a|} \Downarrow_p^{\mathrm{R}_{\mathrm{EBS}}} \widetilde{|b|}$$

Proof. Our hypothesis comes from a rule of the form $p(\widetilde{y}, c(\widetilde{x})) := S \in \mathrm{R}_{\mathrm{BS}} \subseteq \mathrm{R}_{\mathrm{gen}}$ with $\widetilde{a} = \widetilde{d}, c(\widetilde{d'})$ and a derivation $\{\widetilde{y \mapsto d}\} + \{\widetilde{x \mapsto d'}\} \vdash S \Downarrow \widetilde{b}$. By definition, $p(\widetilde{y}, c(\widetilde{x})) := \lceil S\rfloor$ is a rule of $\mathrm{R}_{\mathrm{dist}}$. And from Lemma 21, we have $\{\widetilde{y \mapsto d}\} + \{\widetilde{x \mapsto d'}\} \vdash \lceil S\rfloor \Downarrow \widetilde{b}$ and $\widetilde{|a|} \Downarrow_p^{\mathrm{R}_{\mathrm{dist}}} \widetilde{|b|}$. Since delaying returns does not change the behavior (Lemma 23), we also have $\widetilde{|a|} \Downarrow_p^{\mathrm{R}_{\mathrm{EBS}}} \widetilde{|b|}$. □

Theorem 35 (EBS⇒BS)**.** For all p, initial terms $\widetilde{a}$ and extended terms $\widetilde{b'}$,

$$\widetilde{|a|} \Downarrow_p^{\mathrm{R_{EBS}}} \widetilde{b'} \implies \exists \widetilde{b}, \widetilde{b'} = \widetilde{|b|} \wedge \widetilde{a} \Downarrow_p^{\mathrm{R_{BS}}} \widetilde{b}$$

Proof. Since $\widetilde{a}$ are initial terms and do not use coercion constructors of the form `Ret_h`, we also have $\widetilde{|a|} \Downarrow_p^{\mathrm{R_{dist}}} \widetilde{b'}$ from Lemma 23.

This comes from a rule of the form $p(\widetilde{y}, c(\widetilde{x})) := \lceil S \rfloor \in \mathrm{R_{dist}}$ with $\widetilde{a} = \widetilde{d}, c(\widetilde{d'})$ and a derivation $\{\widetilde{y \mapsto d}\} + \{\widetilde{x \mapsto d'}\} \vdash \lceil S \rfloor \Downarrow \widetilde{b}$. By definition, $p(\widetilde{y}, c(\widetilde{x})) := S$ is a rule of $\mathrm{R_{gen}}$, and from Lemma 21, we have $\widetilde{|a|} \Downarrow_p^{\mathrm{R_{gen}}} \widetilde{b'}$.

Since $\widetilde{a}$ are initial terms and do not use newly generated constructors (e.g., `While1`), then c is an initial constructor, $p(\widetilde{y}, c(\widetilde{x})) := S$ is also a rule of $\mathrm{R_{BS}}$, and we have $\widetilde{a} \Downarrow_p^{\mathrm{R_{BS}}} \widetilde{b'}$. This shows that $\widetilde{b'}$ is necessarily restricted to initial constructors, which can be stated as

$$\exists \widetilde{b}, \widetilde{b'} = \widetilde{|b|} \wedge \widetilde{a} \Downarrow_p^{\mathrm{R_{BS}}} \widetilde{b}$$

□

Theorem 36 (Div: BS⇔EBS)**.** For all p and initial terms $\widetilde{a}$,

$$\widetilde{a} \Uparrow_p^{\mathrm{R_{BS}}} \iff \widetilde{|a|} \Uparrow_p^{\mathrm{R_{EBS}}}$$

Proof. Similarly to the two previous theorems. Since we only focus on initial terms (no new generated constructors), $\widetilde{a} \Uparrow_p^{\mathrm{R_{BS}}} \iff \widetilde{|a|} \Uparrow_p^{\mathrm{R_{gen}}}$. From the definition of $\mathrm{R_{dist}}$ and Lemma 22, $\widetilde{|a|} \Uparrow_p^{\mathrm{R_{gen}}} \iff \widetilde{|a|} \Uparrow_p^{\mathrm{R_{dist}}}$. Since we only focus on initial terms (no coerced returns) and delaying returns is also equivalent for coinduction, $\widetilde{|a|} \Uparrow_p^{\mathrm{R_{dist}}} \iff \widetilde{|a|} \Uparrow_p^{\mathrm{R_{EBS}}}$. □

C.6 Extended Big-Step Implies Small-Step

We first certify the finite case with the following theorem.

Theorem 37 (EBS ⇒ SS)**.** For all $\widetilde{a}$, c, $\widetilde{a'}$, $\widetilde{b}$, and p,

$$(\widetilde{a}, c(\widetilde{a'})) \Downarrow_p^{\mathrm{R_{EBS}}} \widetilde{b} \implies (\widetilde{a}, c(\widetilde{a'})) \, (\Downarrow_p^{\mathrm{R_{SS}}})^* \, (\widetilde{a}, \texttt{Ret\_p}(\widetilde{b}))$$

Proof. Since $(\widetilde{a}, c(\widetilde{a'})) \Downarrow_p^{\mathrm{R_{EBS}}} \widetilde{b}$ is defined mutually with the evaluation of skeletons $(\Sigma \vdash S \Downarrow^{\mathrm{R_{EBS}}} \widetilde{b})$, we will prove two results at the same time: Forall $\widetilde{a}, c, \widetilde{a'}, \widetilde{b}, p, \widetilde{y}, \widetilde{x}, S_0, \Sigma, S$, we have:

- $(\widetilde{a}, c(\widetilde{a'})) \Downarrow_p^{\mathrm{R_{EBS}}} \widetilde{b} \implies (\widetilde{a}, c(\widetilde{a'})) \, (\Downarrow_p^{\mathrm{R_{SS}}})^* \, (\widetilde{a}, \texttt{Ret\_p}(\widetilde{b}))$
- $\begin{cases} r = (p(\widetilde{y}, c(\widetilde{x})) := S_0) \in \mathrm{R_{EBS}} \\ S \triangleleft S_0 \\ \Sigma(\widetilde{y}) = \widetilde{a} \\ \Sigma \vdash S \Downarrow^{\mathrm{R_{EBS}}} \widetilde{b} \\ S \text{ has no } \texttt{Reuse} \text{ label} \end{cases} \implies \exists d, \begin{cases} \Sigma \vdash \| \, S \, \|^r \Downarrow^{\mathrm{R_{SS}}} (\widetilde{a}, d) \\ (\widetilde{a}, d) \, (\Downarrow_p^{\mathrm{R_{SS}}})^* \, (\widetilde{a}, \texttt{Ret\_p}(\widetilde{b})) \end{cases}$

The second result says that if S evaluates to $\widetilde{b}$ as part of a big-step evaluation, then $\| \, S \, \|^r$ can be (small-step) reduced to an intermediate configuration $(\widetilde{a}, d)$ that can reduce further to $(\widetilde{a}, \texttt{Ret\_p}(\widetilde{b}))$.

We reason by mutual induction on the derivations of $(\widetilde{a}, c(\widetilde{a'})) \Downarrow_p^{\mathrm{R_{EBS}}} \widetilde{b}$ and $\Sigma \vdash S \Downarrow^{\mathrm{R_{EBS}}} \widetilde{b}$.

► For the first result, our hypothesis is $(\widetilde{a}, c(\widetilde{a'})) \Downarrow_p^{\mathrm{R_{EBS}}} \widetilde{b}$.

- If c = `Ret_p`, using Lemma 25 we have $\widetilde{a'} = \widetilde{b}$ and this is trivial as we take zero small-steps.
- If c is a new constructor, the bottom of the derivation is of the form

$$\dfrac{p(\widetilde{y}, c(\widetilde{w}, \widetilde{z})) := \ldots \qquad \dfrac{\overline{\Sigma_0(\widetilde{w}) \Downarrow_{p_1}^{\mathrm{R_{EBS}}} \widetilde{b'}} \qquad \overline{\Sigma_0 + \{\widetilde{v \mapsto b'}\} \vdash S \Downarrow^{\mathrm{R_{EBS}}} \widetilde{b}}}{\Sigma_0 \vdash \texttt{let } \widetilde{v} = \texttt{Proc Reuse } p_1 \ \widetilde{w} \ \texttt{ in } \ S \Downarrow^{\mathrm{R_{EBS}}} \widetilde{b}}}{(\widetilde{a}, c((\widetilde{a'_0}, a'_1), \widetilde{a'_2})) \Downarrow_p^{\mathrm{R_{EBS}}} \widetilde{b}}$$

  where $\widetilde{a'} = ((\widetilde{a'_0}, a'_1), \widetilde{a'_2})$, $\Sigma_0 = \widetilde{\{y \mapsto a\}} + \{\widetilde{w} \mapsto (\widetilde{a'_0}, a'_1)\} + \{\widetilde{z \mapsto a'_2}\}$, and we use the rule $p(\widetilde{y}, c(\widetilde{w}, \widetilde{z})) := \texttt{let } \widetilde{v} = \texttt{Proc Reuse } p_1 \ \widetilde{w} \ \texttt{ in } \ S$. Using our induction hypothesis on the first leaf we get $(\widetilde{a'_0}, a'_1) \, (\Downarrow_{p_1}^{\mathrm{R_{SS}}})^* \, (\widetilde{a'_0}, \texttt{Ret\_p1}(\widetilde{b'}))$. By Lemma 26, we then have $(\widetilde{a}, c((\widetilde{a'_0}, a'_1), \widetilde{a'_2})) \, (\Downarrow_p^{\mathrm{R_{SS}}})^* \, (\widetilde{a}, c((\widetilde{a'_0}, \texttt{Ret\_p1}(\widetilde{b'})), \widetilde{a'_2}))$.

  We can also use our induction hypothesis on the second leaf (with S) to get d such that $\Sigma_0 + \{\widetilde{v \mapsto b'}\} \vdash \| \, S \, \|^r \Downarrow^{\mathrm{R_{SS}}} (\widetilde{a}, d)$ and $(\widetilde{a}, d) \, (\Downarrow_p^{\mathrm{R_{SS}}})^* \, (\widetilde{a}, \texttt{Ret\_p}(\widetilde{b}))$.

Finally, using Lemma 28 (and Lemma 10 to rewrite the environment), we have $(\widetilde{a}, c((\widetilde{a'_0}, \texttt{Ret\_p1}(\widetilde{b'})), \widetilde{a'_2})) \Downarrow_p^{\mathrm{R_{SS}}} (\widetilde{a}, d)$, and we just have to assemble the three pieces by transitivity.

- Otherwise, if c is an initial constructor, the bottom of the tree derivation is of the form:

$$\frac{p(\widetilde{y}, c(\widetilde{x})) := S \in \mathrm{R_{EBS}} \qquad \dfrac{\cdots}{\Sigma_0 \vdash S \Downarrow^{\mathrm{R_{EBS}}} \widetilde{b}}}{(\widetilde{a}, c(\widetilde{a'})) \Downarrow_p^{\mathrm{R_{EBS}}} \widetilde{b}}$$

Where $\Sigma_0 = \widetilde{\{y \mapsto a\}} + \widetilde{\{x \mapsto a'\}}$ and S does not start with a label `Reuse`.
We use our induction hypothesis with $\Sigma_0 \vdash S \Downarrow^{\mathrm{R_{EBS}}} \widetilde{b}$ to obtain d such that $\Sigma_0 \vdash \| S \|^r \Downarrow^{\mathrm{R_{SS}}} (\widetilde{a}, d)$ and $(\widetilde{a}, d)\ (\Downarrow_p^{\mathrm{R_{SS}}})^* (\widetilde{a}, \texttt{Ret\_p}(\widetilde{b}))$.
Now we can complete our goal as such:

$$\frac{\dfrac{p(\widetilde{y}, c(\widetilde{x})) := \| S \|^r \in \mathrm{R_{SS}} \qquad \overline{\Sigma_0 \vdash \| S \|^r \Downarrow^{\mathrm{R_{SS}}} (\widetilde{a}, d)}}{(\widetilde{a}, c(\widetilde{a'})) \Downarrow_p^{\mathrm{R_{SS}}} (\widetilde{a}, d)} \qquad \overline{(\widetilde{a}, d)\ (\Downarrow_p^{\mathrm{R_{SS}}})^* (\widetilde{a}, \texttt{Ret\_p}(\widetilde{b}))}}{(\widetilde{a}, c(\widetilde{a'}))\ (\Downarrow_p^{\mathrm{R_{SS}}})^* (\widetilde{a}, \texttt{Ret\_p}(\widetilde{b}))}$$

▶ For the second result, we assume $\begin{cases} r = (p(\widetilde{y}, c(\widetilde{x})) := S_0) \in \mathrm{R_{EBS}} \\ S \triangleleft S_0 \\ \Sigma(\widetilde{y}) = \widetilde{a} \\ \Sigma \vdash S \Downarrow^{\mathrm{R_{EBS}}} \widetilde{b} \\ S \text{ has no } \texttt{Reuse} \text{ label} \end{cases}$

Note that we have $\mathrm{SSA}(S_0)$, either from assumption for the initial skeletons, or by using Lemmas 20 and 19 for new skeletons. Thus we know extending the environment Σ will not change the mapping of variables $\widetilde{y}$.

As said previously, we proceed by induction on the derivation of $\Sigma \vdash S \Downarrow^{\mathrm{R_{EBS}}} \widetilde{b}$.

- If S is of the form $\texttt{Return}\ \widetilde{t}$, then we choose $d = \texttt{Ret\_p}(\widetilde{b})$:

$$\frac{\Sigma(\widetilde{t}) = \widetilde{b}}{\Sigma \vdash \texttt{Return}\ \widetilde{t} \Downarrow^{\mathrm{R_{EBS}}} \widetilde{b}} \implies \begin{cases} \dfrac{\Sigma(\widetilde{y}, \texttt{Ret\_p}(\widetilde{t})) = (\widetilde{a}, \texttt{Ret\_p}(\widetilde{b}))}{\Sigma \vdash \texttt{Return}\ (\widetilde{y}, \texttt{Ret\_p}(\widetilde{t})) \Downarrow^{\mathrm{R_{SS}}} (\widetilde{a}, \texttt{Ret\_p}(\widetilde{b}))} \\ \\ (\widetilde{a}, \texttt{Ret\_p}(\widetilde{b}))\ (\Downarrow_p^{\mathrm{R_{SS}}})^* (\widetilde{a}, \texttt{Ret\_p}(\widetilde{b})) \qquad \text{(with zero steps)} \end{cases}$$

- If S is of the form $\texttt{Branch}\ (S_1, \ldots, S_n)$, using the induction hypothesis with S_i:

$$\frac{\overline{\Sigma \vdash S_i \Downarrow^{\mathrm{R_{EBS}}} \widetilde{b}}}{\Sigma \vdash \texttt{Branch}\ (S_1, \ldots, S_n) \Downarrow^{\mathrm{R_{EBS}}} \widetilde{b}} \implies \begin{cases} \dfrac{\overline{\Sigma \vdash \| S_i \|^r \Downarrow^{\mathrm{R_{SS}}} (\widetilde{a}, d)}}{\Sigma \vdash \texttt{Branch}\ (\| S_1 \|^r, \ldots, \| S_n \|^r) \Downarrow^{\mathrm{R_{SS}}} (\widetilde{a}, d)} \\ \\ (\widetilde{a}, d)\ (\Downarrow_p^{\mathrm{R_{SS}}})^* (\widetilde{a}, \texttt{Ret\_p}(\widetilde{b})) \end{cases}$$

- If S is of the form $\texttt{let}\ \widetilde{v} = \texttt{Filter}\ f\ \widetilde{t}\ \texttt{in}\ S_0$, using the induction hypothesis with S_0:

$$\frac{\dfrac{\mathcal{R}_f(\widetilde{\Sigma(t)}) \Downarrow \widetilde{b'}}{\Sigma \vdash \texttt{Filter}\ f\ \widetilde{t} \Downarrow \widetilde{b'}} \qquad \overline{\Sigma + \widetilde{\{v \mapsto b'\}} \vdash S_0 \Downarrow^{\mathrm{R_{EBS}}} \widetilde{b}}}{\Sigma \vdash \texttt{let}\ \widetilde{v} = \texttt{Filter}\ f\ \widetilde{t}\ \texttt{in}\ S_0 \Downarrow^{\mathrm{R_{EBS}}} \widetilde{b}} \implies$$

$$\begin{cases} \dfrac{\dfrac{\mathcal{R}_f(\widetilde{\Sigma(t)}) \Downarrow \widetilde{b'}}{\Sigma \vdash \texttt{Filter}\ f\ \widetilde{t} \Downarrow \widetilde{b'}} \qquad \overline{\Sigma + \widetilde{\{v \mapsto b'\}} \vdash \| S_0 \|^r \Downarrow^{\mathrm{R_{SS}}} (\widetilde{a}, d)}}{\Sigma \vdash \texttt{let}\ \widetilde{v} = \texttt{Filter}\ f\ \widetilde{t}\ \texttt{in}\ \| S_0 \|^r \Downarrow^{\mathrm{R_{SS}}} (\widetilde{a}, d)} \\ \\ (\widetilde{a}, d)\ (\Downarrow_p^{\mathrm{R_{SS}}})^* (\widetilde{a}, \texttt{Ret\_p}(\widetilde{b})) \end{cases}$$

(similarly for $S := \texttt{let}\ \widetilde{v} = \texttt{Return}\ \widetilde{t}\ \texttt{in}\ S_0$)

• If S is of the form $\texttt{let}\ \widetilde{v} = \texttt{Proc}\ (\texttt{New}\, c_0)\ p_1\ \widetilde{t}\ \ \texttt{in}\ \ S_0$, we have:

$$\frac{\dfrac{\Sigma(\widetilde{t}) \Downarrow_{p_1}^{\mathrm{R_{EBS}}} \widetilde{b'}}{\Sigma \vdash \texttt{Proc}\ (\texttt{New}\, c_0)\ p_1\ \widetilde{t} \Downarrow^{\mathrm{R_{EBS}}} \widetilde{b'}} \qquad \dfrac{}{\Sigma + \{\widetilde{v \mapsto b'}\} \vdash S_0 \Downarrow^{\mathrm{R_{EBS}}} \widetilde{b}}}{\Sigma \vdash \texttt{let}\ \widetilde{v} = \texttt{Proc}\ (\texttt{New}\, c_0)\ p_1\ \widetilde{t}\ \ \texttt{in}\ \ S_0 \Downarrow^{\mathrm{R_{EBS}}} \widetilde{b}}$$

Let us name $\Sigma(\widetilde{t}) = (\widetilde{e}, e_t)$. We use our induction hypothesis on the first leaf and get $(\widetilde{e}, e_t)\ (\Downarrow_{p_1}^{\mathrm{R_{SS}}})^*\ (\widetilde{e}, \texttt{Ret\_p1}(\widetilde{b'}))$. We also use our induction hypothesis on the second leaf and we have d_0 such that $\Sigma + \{\widetilde{v \mapsto b'}\} \vdash \parallel S_0 \parallel^r \Downarrow^{\mathrm{R_{SS}}} (\widetilde{a}, d_0)$ and $(\widetilde{a}, d_0)\ (\Downarrow_p^{\mathrm{R_{SS}}})^*\ (\widetilde{a}, \texttt{Ret\_p}(\widetilde{b}))$.

From Lemma 33, $\mathrm{R_{EBS}}$ contains a rule of the form $p(\widetilde{y}, c_0(\widetilde{w}, \widetilde{z})) := \texttt{let}\ \widetilde{v} = \texttt{Proc Reuse}\ p_1\ \widetilde{w}\ \ \texttt{in}\ \ S_0$.

With $d = c_0(\Sigma(t), \Sigma(z)) = c_0(\widetilde{e}, e_t, \Sigma(z))$, we have $\parallel S \parallel^r = \texttt{Return}\ (\widetilde{y}, c_0(\widetilde{t}, \widetilde{z}))$ and thus:

$$\frac{\Sigma(\widetilde{y}, c_0(\widetilde{t}, \widetilde{z})) = (\widetilde{a}, d)}{\Sigma \vdash \texttt{Return}\ (\widetilde{y}, c_0(\widetilde{t}, \widetilde{z})) \Downarrow^{\mathrm{R_{SS}}} (\widetilde{a}, d)}$$

To conclude, we are missing $(\widetilde{a}, d)\ (\Downarrow_p^{\mathrm{R_{SS}}})^*\ (\widetilde{a}, d_0)$ which can be cut into two pieces:

$$\frac{\dfrac{\dots}{(\widetilde{a}, d)\ (\Downarrow_p^{\mathrm{R_{SS}}})^*\ (\widetilde{a}, c_0(\widetilde{e}, \texttt{Ret\_p1}(\widetilde{b'}), \Sigma(\widetilde{z})))} \qquad \dfrac{\dots}{(\widetilde{a}, c_0(\widetilde{e}, \texttt{Ret\_p1}(\widetilde{b'}), \Sigma(\widetilde{z}))) \Downarrow_p^{\mathrm{R_{SS}}} (\widetilde{a}, d_0)}}{(\widetilde{a}, d)\ (\Downarrow_p^{\mathrm{R_{SS}}})^*\ (\widetilde{a}, d_0)}$$

The first part comes directly from iterating Lemma 26. Also, Lemma 28 gives us:

$$\frac{\widetilde{\{y \mapsto a\}} + \widetilde{\{z \mapsto \Sigma(z)\}} + \widetilde{\{v \mapsto b'\}} \vdash \parallel S_0 \parallel^r \Downarrow^{\mathrm{R_{SS}}} (\widetilde{a}, d_0)}{(\widetilde{a}, c_0(\widetilde{e}, \texttt{Ret\_p1}(\widetilde{b'}), \Sigma(\widetilde{z}))) \Downarrow_p^{\mathrm{R_{SS}}} (\widetilde{a}, d_0)}$$

Which is enough for the second part, using $\Sigma + \{\widetilde{v \mapsto b'}\} \vdash \parallel S_0 \parallel^r \Downarrow^{\mathrm{R_{SS}}} (\widetilde{a}, d_0)$ and Lemma 10. □

We know tackle the infinite case. The proof can be separated in two successive lemmas.

Lemma 38. For all p, $\widetilde{y}$, c, $\widetilde{x}$, S_0, Σ, S, and $\widetilde{a}$,

$$\begin{cases} r = (p(\widetilde{y}, c(\widetilde{x})) := S_0) \in \mathrm{R_{EBS}} \\ S \triangleleft S_0 \\ \Sigma(\widetilde{y}) = \widetilde{a} \\ \Sigma \vdash S \Uparrow^{\mathrm{R_{EBS}}} \\ S \text{ has no } \texttt{Reuse} \text{ label} \end{cases} \implies \exists d, \begin{cases} \Sigma \vdash \parallel S \parallel^r \Downarrow^{\mathrm{R_{SS}}} (\widetilde{a}, d) \\ (\widetilde{a}, d) \Uparrow_p^{\mathrm{R_{EBS}}} \end{cases}$$

In other words, if evaluating S diverges during a (big-step) evaluation, then $\parallel S \parallel^r$ can be (small-step) reduced into an intermediate configuration $(\widetilde{a}, d)$ whose (big-step) evaluation also diverges.

Proof. We prove this by induction on S.

► If S is of the form $\texttt{Branch}\ (S_1, \dots, S_n)$, using the induction hypothesis with S_i:

$$\frac{\dfrac{}{\Sigma \vdash S_i \Uparrow^{\mathrm{R_{EBS}}}}}{\Sigma \vdash \texttt{Branch}\ (S_1, \dots, S_n) \Uparrow^{\mathrm{R_{EBS}}}} \implies \begin{cases} \dfrac{\dfrac{}{\Sigma \vdash \parallel S_i \parallel^r \Downarrow^{\mathrm{R_{SS}}} (\widetilde{a}, d)}}{\Sigma \vdash \texttt{Branch}\ (\parallel S_1 \parallel^r, \dots, \parallel S_n \parallel^r) \Downarrow^{\mathrm{R_{SS}}} (\widetilde{a}, d)} \\ (\widetilde{a}, d) \Uparrow_p^{\mathrm{R_{EBS}}} \end{cases}$$

► If S is of the form $\texttt{let}\ \widetilde{v} = K\ \ \texttt{in}\ \ S_0$, where K is a filter or a return, using the induction hypothesis with S_0:

$$\frac{\Sigma \vdash K \Downarrow \widetilde{b'} \qquad \dfrac{}{\Sigma + \{\widetilde{v \mapsto b'}\} \vdash S_0 \Uparrow^{\mathrm{R_{EBS}}}}}{\Sigma \vdash \texttt{let}\ \widetilde{v} = K\ \ \texttt{in}\ \ S_0 \Uparrow^{\mathrm{R_{EBS}}}} \implies \begin{cases} \dfrac{\Sigma \vdash K \Downarrow \widetilde{b'} \qquad \dfrac{}{\Sigma + \{\widetilde{v \mapsto b'}\} \vdash \parallel S_0 \parallel^r \Downarrow^{\mathrm{R_{SS}}} (\widetilde{a}, d)}}{\Sigma \vdash \texttt{let}\ \widetilde{v} = K\ \ \texttt{in}\ \parallel S_0 \parallel^r \Downarrow^{\mathrm{R_{SS}}} (\widetilde{a}, d)} \\ (\widetilde{a}, d) \Uparrow_p^{\mathrm{R_{EBS}}} \end{cases}$$

► If S is of the form $\texttt{let}\ \widetilde{v} = \texttt{Proc}\ (\texttt{New}\, c_0)\ p_1\ \widetilde{t}\ \ \texttt{in}\ \ S_0$, we have:

From Lemma 33, $\mathrm{R_{EBS}}$ contains a rule of the form

$$p(\widetilde{y}, c_0(\widetilde{w}, \widetilde{z})) := \texttt{let}\ \widetilde{v} = \texttt{Proc Reuse}\ p_1\ \widetilde{w}\ \texttt{in}\ S_0$$

We choose $d = c_0(\Sigma(\widetilde{t}), \Sigma(\widetilde{z}))$, we have $\| S \|^r = \texttt{Return}\ (\widetilde{y}, c_0(\widetilde{t}, \widetilde{z}))$ and thus:

$$\frac{\Sigma(\widetilde{y}, c_0(\widetilde{t}, \widetilde{z})) = (\widetilde{a}, d)}{\Sigma \vdash \texttt{Return}\ (\widetilde{y}, c_0(\widetilde{t}, \widetilde{z})) \Downarrow^{\mathrm{R_{SS}}} (\widetilde{a}, d)}$$

We also need to check $(\widetilde{a}, d) \Uparrow_p^{\mathrm{R_{EBS}}}$. For this we have the following tree:

$$\frac{p(\widetilde{y}, c_0(\widetilde{w}, \widetilde{z})) := \ldots \in \mathrm{R_{EBS}} \qquad \dfrac{}{\Sigma' \vdash \texttt{let}\ \widetilde{v} = \texttt{Proc Reuse}\ p_1\ \widetilde{w}\ \texttt{in}\ S_0 \Uparrow^{\mathrm{R_{EBS}}}}}{(\widetilde{a}, d) \Uparrow_p^{\mathrm{R_{EBS}}}}\ \textsc{Div-tuple}$$

where $\Sigma' = \{\widetilde{y \mapsto a}\} + \{\widetilde{w \mapsto \Sigma(t)}\} + \{\widetilde{z \mapsto \Sigma(z)}\}$. The leaf comes from our hypothesis $\Sigma \vdash S \Uparrow^{\mathrm{R_{EBS}}}$. If the proc call $(\texttt{Proc}\ (\texttt{New}\, c_0)\ p_1\ \widetilde{t})$ diverges, this is straightforward as $\Sigma(\widetilde{t}) = \Sigma'(\widetilde{w})$. If evaluating S_0 diverges, we go from Σ to Σ' using Lemma 11.

□

Lemma 39. For all $\widetilde{a}$, c, $\widetilde{a'}$, and p,

$$(\widetilde{a}, c(\widetilde{a'})) \Uparrow_p^{\mathrm{R_{EBS}}} \quad \Longrightarrow \quad \exists \widetilde{b},\ (\widetilde{a}, c(\widetilde{a'})) \Downarrow_p^{\mathrm{R_{SS}}} \widetilde{b} \ \wedge\ \widetilde{b} \Uparrow_p^{\mathrm{R_{EBS}}}$$

Proof. We prove this result by induction on $(\widetilde{a}, c(\widetilde{a'}))$, keeping p universally quantified. We distinguish the different cases of $(\widetilde{a}, c(\widetilde{a'})) \Uparrow_p^{\mathrm{R_{EBS}}}$.

• If it comes from a tree

$$\frac{p(\widetilde{y}, c(\widetilde{w}, \widetilde{z})) := \ldots \in \mathrm{R_{EBS}} \qquad \dfrac{\dfrac{\dfrac{\widetilde{a_1} \Uparrow_{p_1}^{\mathrm{R_{EBS}}}}{\Sigma \vdash \texttt{Proc}\ p_1\ \widetilde{w} \Uparrow^{\mathrm{R_{EBS}}}}\ \textsc{Div-Pc}}{\Sigma \vdash \texttt{let}\ \widetilde{v} = \texttt{Proc Reuse}\ p_1\ \widetilde{w}\ \texttt{in}\ S \Uparrow^{\mathrm{R_{EBS}}}}\ \textsc{Div-LetL}}{(\widetilde{a}, c(\widetilde{a_1}, \widetilde{a_3})) \Uparrow_p^{\mathrm{R_{EBS}}}}\ \textsc{Div-tuple}$$

with $\Sigma = \{\widetilde{y \mapsto a}\} + \{\widetilde{w \mapsto a_1}\} + \{\widetilde{z \mapsto a_3}\}$ and $\widetilde{a'} = (\widetilde{a_1}, \widetilde{a_3})$, then we can use our induction hypothesis since $\widetilde{a_1}$ is a subterm of $(\widetilde{a}, c(\widetilde{a'}))$.

So there is $\widetilde{b'}$ such that $\widetilde{a_1} \Downarrow_{p_1}^{\mathrm{R_{SS}}} \widetilde{b'}$ and $\widetilde{b'} \Uparrow_{p_1}^{\mathrm{R_{EBS}}}$. We choose $\widetilde{b} = (\widetilde{a}, c(\widetilde{b'}, \widetilde{a_3}))$ and we have $(\widetilde{a}, c(\widetilde{a_1}, \widetilde{a_3}))) \Downarrow_p^{\mathrm{R_{SS}}} \widetilde{b}$ from Lemma 26 and $\widetilde{b} \Uparrow_p^{\mathrm{R_{EBS}}}$ from a tree similar to the one above (replacing $\widetilde{a_1}$ by $\widetilde{b'}$).

• If it comes from a tree

$$\frac{p(\widetilde{y}, c(\widetilde{w}, \widetilde{z})) := \ldots \in \mathrm{R_{EBS}} \qquad \dfrac{\dfrac{\dfrac{\widetilde{a_1}, a_2 \Downarrow_{p_1}^{\mathrm{R_{EBS}}} \widetilde{d}}{\Sigma \vdash \texttt{Proc}\ p_1\ \widetilde{w} \Downarrow^{\mathrm{R_{EBS}}} \widetilde{d}} \qquad \dfrac{}{\Sigma' \vdash S \Uparrow^{\mathrm{R_{EBS}}}}}{\Sigma \vdash \texttt{let}\ \widetilde{v} = \texttt{Proc Reuse}\ p_1\ \widetilde{w}\ \texttt{in}\ S \Uparrow^{\mathrm{R_{EBS}}}}\ \textsc{Div-LetR}}{(\widetilde{a}, c(\widetilde{a_1}, a_2, \widetilde{a_3})) \Uparrow_p^{\mathrm{R_{EBS}}}}\ \textsc{Div-tuple}$$

with $\Sigma = \{\widetilde{y \mapsto a}\} + \{\widetilde{w} \mapsto (\widetilde{a_1}, a_2)\} + \{\widetilde{z \mapsto a_3}\}$, $\Sigma' = \Sigma + \{\widetilde{v \mapsto d}\}$ and $\widetilde{a'} = (\widetilde{a_1}, a_2, \widetilde{a_3})$, there is two subcases.

If a_2 is not of the form $\texttt{Ret\_p1}(..)$, then Theorem 37 gives us $\widetilde{a_1}, a_2\ (\Downarrow_{p_1}^{\mathrm{R_{SS}}})^*\ \widetilde{a_1}, \texttt{Ret\_p1}(\widetilde{d})$ which implies there is a_2' such that $\widetilde{a_1}, a_2 \Downarrow_{p_1}^{\mathrm{R_{SS}}} \widetilde{a_1}, a_2'\ (\Downarrow_{p_1}^{\mathrm{R_{SS}}})^*\ \widetilde{a_1}, \texttt{Ret\_p1}(\widetilde{d})$. We take $\widetilde{b} = (\widetilde{a}, c(\widetilde{a_1}, a_2', \widetilde{a_3}))$, we have $(\widetilde{a}, c(\widetilde{a_1}, a_2, \widetilde{a_3}))) \Downarrow_p^{\mathrm{R_{SS}}} \widetilde{b}$ from Lemma 26 and $\widetilde{b} \Uparrow_p^{\mathrm{R_{EBS}}}$ from a tree similar to the one above (replacing a_2 by a_2').

Otherwise we have $a_2 = \texttt{Ret\_p1}(\widetilde{d})$ from Lemma 24. Using Lemma 38 above, there is d_0 such that $\Sigma' \vdash \| S \|^r \Downarrow^{\mathrm{R_{SS}}} (\widetilde{a}, d_0)$ and $(\widetilde{a}, d_0) \Uparrow_p^{\mathrm{R_{EBS}}}$, where r is the rule on the tree above. We choose $\widetilde{b} = (\widetilde{a}, d_0)$ and can conclude $(\widetilde{a}, c(\widetilde{a'})) \Downarrow_p^{\mathrm{R_{SS}}} \widetilde{b}$ using Lemma 28.

• Lastly, if it comes from a tree

$$\frac{p(\widetilde{y}, c(\widetilde{x})) := S \in \mathrm{R_{EBS}} \qquad \dfrac{}{\Sigma_0 \vdash S \Uparrow^{\mathrm{R_{EBS}}}}}{(\widetilde{a}, c(\widetilde{a'})) \Uparrow_p^{\mathrm{R_{EBS}}}}\ \textsc{Div-tuple}$$

where S does not start with a Reuse label, with $\Sigma_0 = \{\widetilde{y \mapsto a}\} + \{\widetilde{x \mapsto a'}\}$.

Once again we use Lemma 38 and there is d such that $\Sigma_0 \vdash \| S \|^r \Downarrow^{\mathrm{R_{SS}}} (\widetilde{a}, d)$ and $(\widetilde{a}, d) \Uparrow_p^{\mathrm{R_{EBS}}}$, where $r = (p(\widetilde{y}, c(\widetilde{x})) := S)$. Then we complete our goal as such:

$$\frac{p(\widetilde{y}, c(\widetilde{x})) := \| S \|^r \in \mathrm{R_{SS}} \qquad \overline{\Sigma_0 \vdash \| S \|^r \Downarrow^{\mathrm{R_{SS}}} (\widetilde{a}, d)}}{(\widetilde{a}, c(\widetilde{a'})) \Downarrow_p^{\mathrm{R_{SS}}} (\widetilde{a}, d)}$$

□

Theorem 40 (Div: EBS ⇒ SS)**.** For all proc p and terms $\widetilde{a}$,

$$\widetilde{a} \Uparrow_p^{\mathrm{R_{EBS}}} \implies \widetilde{a} \xrightarrow{\infty}_p$$

Proof. Given Definition 6 of $\widetilde{a} \xrightarrow{\infty}_p$, by coinduction, we want to show that $\widetilde{a} \Uparrow_p^{\mathrm{R_{EBS}}}$ satisfies the stated property. Thus it is sufficient to prove:

$$\forall \widetilde{a}, p, \quad \widetilde{a} \Uparrow_p^{\mathrm{R_{EBS}}} \quad \implies \quad \exists \widetilde{b},\ \widetilde{a} \Downarrow_p^{\mathrm{R_{SS}}} \widetilde{b} \ \wedge\ \widetilde{b} \Uparrow_p^{\mathrm{R_{EBS}}}$$

We prove this separately in Lemma 39 above. □

C.7 Small-Step implies Extended Big-Step

We first certify the finite case. This is done by iterating the following concatenation lemma.

Lemma 41 (Concatenation)**.** For all $\widetilde{a}$, c, $\widetilde{a'}$, d, $\widetilde{b}$, and p,

$$\begin{cases} (\widetilde{a}, c(\widetilde{a'})) \Downarrow_p^{\mathrm{R_{SS}}} (\widetilde{a}, d) \\ (\widetilde{a}, d) \Downarrow_p^{\mathrm{R_{EBS}}} \widetilde{b} \end{cases} \implies (\widetilde{a}, c(\widetilde{a'})) \Downarrow_p^{\mathrm{R_{EBS}}} \widetilde{b}$$

Proof. Once again $(\widetilde{a}, c(\widetilde{a'})) \Downarrow_p^{\mathrm{R_{SS}}} (\widetilde{a}, d)$ is defined mutually with the evaluation of skeletons ($\Sigma \vdash \| S \|^r \Downarrow^{\mathrm{R_{SS}}} (\widetilde{a}, d)$), so we will prove two results at the same time:
Forall $\widetilde{a}, c, \widetilde{a'}, d, \widetilde{b}, p, \widetilde{y}, \widetilde{x}, S_0, \Sigma, S$, we have:

- $\begin{cases} (\widetilde{a}, c(\widetilde{a'})) \Downarrow_p^{\mathrm{R_{SS}}} (\widetilde{a}, d) \\ (\widetilde{a}, d) \Downarrow_p^{\mathrm{R_{EBS}}} \widetilde{b} \end{cases} \implies (\widetilde{a}, c(\widetilde{a'})) \Downarrow_p^{\mathrm{R_{EBS}}} \widetilde{b}$
- $\begin{cases} r = (p(\widetilde{y}, c(\widetilde{x})) := S_0) \in \mathrm{R_{EBS}} \\ S \lhd S_0 \\ \Sigma(\widetilde{y}) = \widetilde{a} \\ \Sigma \vdash \| S \|^r \Downarrow^{\mathrm{R_{SS}}} (\widetilde{a}, d) \\ (\widetilde{a}, d) \Downarrow_p^{\mathrm{R_{EBS}}} \widetilde{b} \\ S \text{ has no } \texttt{Reuse} \text{ label} \end{cases} \implies \Sigma \vdash S \Downarrow^{\mathrm{R_{EBS}}} \widetilde{b}$

The second result says that, if $\| S \|^r$ (small-step) reduces to a configuration $(\widetilde{a}, d)$ that (big-step) evaluates to $\widetilde{b}$, then we can create a big-step evaluation of S to $\widetilde{b}$.

We reason by mutual induction on $(\widetilde{a}, c(\widetilde{a'})) \Downarrow_p^{\mathrm{R_{SS}}} \widetilde{b}$ and on S.

► First result, by induction on $(\widetilde{a}, c(\widetilde{a'})) \Downarrow_p^{\mathrm{R_{SS}}} (\widetilde{a}, d)$.

- If c is a new constructor, the main rule $r \in \mathrm{R_{EBS}}$ is of the form $p(\widetilde{y}, c(\widetilde{w}, \widetilde{z})) := \texttt{let}\ \widetilde{v} = \texttt{Proc Reuse}\ p_1\ \widetilde{w}\ \texttt{in}\ S_0$. and we have $\widetilde{a'} = ((\widetilde{a'_0}, a'_1), \widetilde{a'_2})$.
  ⊛ If a'_1 is not of the form $\texttt{Ret\_p1}(\ldots)$, then the small-step reduction uses the first branch and our small-step hypothesis corresponds to the following tree.

$$\dfrac{(\ldots := \| \ldots \|^r) \in \mathrm{R_{SS}} \qquad \dfrac{\dfrac{(\widetilde{a'_0}, a'_1) \Downarrow_{p_1}^{\mathrm{R_{SS}}} (\widetilde{a'_0}, a''_1) \qquad \dfrac{\Sigma'(\widetilde{y}, c(\widetilde{u}, \widetilde{z})) = (\widetilde{a}, d)}{\Sigma' \vdash \texttt{Return}\ (\widetilde{y}, c(\widetilde{u}, \widetilde{z})) \Downarrow^{\mathrm{R_{SS}}} (\widetilde{a}, d)}}{\Sigma_0 \vdash \texttt{let}\ \widetilde{u} = \texttt{Proc}\ p_1\ \widetilde{w}\ \texttt{in Return}\ (\widetilde{y}, c(\widetilde{u}, \widetilde{z})) \Downarrow^{\mathrm{R_{SS}}} (\widetilde{a}, d)}}{\Sigma_0 \vdash \| \texttt{let}\ \widetilde{v} = \texttt{Proc Reuse}\ p_1\ \widetilde{w}\ \texttt{in}\ S_0 \|^r \Downarrow^{\mathrm{R_{SS}}} (\widetilde{a}, d)}}{(\widetilde{a}, c((\widetilde{a'_0}, a'_1), \widetilde{a'_2})) \Downarrow_p^{\mathrm{R_{SS}}} (\widetilde{a}, d)}$$

  where $d = c((\widetilde{a'_0}, a''_1), \widetilde{a'_2})$, and $\Sigma_0 = \widetilde{\{y \mapsto a\}} + \{\widetilde{w} \mapsto (\widetilde{a'_0}, a'_1)\} + \widetilde{\{z \mapsto a'_2\}}$, and $\Sigma' = \Sigma_0 + \{\widetilde{u} \mapsto (\widetilde{a'_0}, a''_1)\}$.

Our second hypothesis is still $(\widetilde{a}, c((\widetilde{a'_0}, a''_1), \widetilde{a'_2}))) \Downarrow_p^{\text{REBS}} \widetilde{b}$, corresponding to the tree:

$$\dfrac{(\dots := \dots) \in \text{R}_{\text{EBS}} \qquad \dfrac{(\widetilde{a'_0}, a''_1) \Downarrow_{p_1}^{\text{REBS}} \widetilde{b'} \qquad \Sigma_1 + \{\widetilde{u \mapsto b'}\} \vdash S_0 \Downarrow^{\text{REBS}} \widetilde{b}}{\Sigma_1 \vdash \texttt{let}\ \widetilde{v} = \texttt{Proc Reuse}\ p_1\ \widetilde{w}\ \texttt{in}\ S_0 \Downarrow^{\text{REBS}} \widetilde{b}}}{(\widetilde{a}, c((\widetilde{a'_0}, a''_1), \widetilde{a'_2})) \Downarrow_p^{\text{REBS}} \widetilde{b}}$$

with $\Sigma_1 = \{\widetilde{y \mapsto a}\} + \{\widetilde{w} \mapsto (\widetilde{a'_0}, a''_1)\} + \{\widetilde{z \mapsto a'_2}\}$.
We can use our first induction hypothesis to combine $(\widetilde{a'_0}, a'_1) \Downarrow_{p_1}^{\text{RSS}} (\widetilde{a'_0}, a''_1)$ and $(\widetilde{a'_0}, a''_1) \Downarrow_{p_1}^{\text{REBS}} \widetilde{b'}$ and produce our goal, i.e. $(\widetilde{a}, c(\widetilde{a'})) \Downarrow_p^{\text{REBS}} \widetilde{b}$:

$$\dfrac{(\dots := \dots) \in \text{R}_{\text{EBS}} \qquad \dfrac{(\widetilde{a'_0}, a'_1) \Downarrow_{p_1}^{\text{REBS}} \widetilde{b'} \qquad \Sigma_0 + \{\widetilde{u \mapsto b'}\} \vdash S_0 \Downarrow^{\text{REBS}} \widetilde{b}}{\Sigma_0 \vdash \texttt{let}\ \widetilde{v} = \texttt{Proc Reuse}\ p_1\ \widetilde{w}\ \texttt{in}\ S_0 \Downarrow^{\text{REBS}} \widetilde{b}}}{(\widetilde{a}, c((\widetilde{a'_0}, a'_1), \widetilde{a'_2})) \Downarrow_p^{\text{REBS}} \widetilde{b}}$$

where we go from Σ_1 to Σ_0 with Lemma 10, since $\widetilde{w}$ does not appear in S_0.
⊛ If $a'_1 = \texttt{Ret\_p1}(\widetilde{b'})$, then the small-step reduction uses the second branch. We can single out the last variable of the proc call to simplify notations:
$p(\widetilde{y}, c((\widetilde{w}, w), \widetilde{z})) := \texttt{let}\ \widetilde{v} = \texttt{Proc Reuse}\ p_1\ (\widetilde{w}, w)\ \texttt{in}\ S_0$, with a state
$\Sigma_0 = \{\widetilde{y \mapsto a}\} + \{\widetilde{w \mapsto a'_0}\} + \{w \mapsto \texttt{Ret\_p1}(\widetilde{b'})\} + \{\widetilde{z \mapsto a'_2}\}$. Our first hypothesis then corresponds to the following tree.

$$\dfrac{(\dots := \| \dots \|^r) \in \text{R}_{\text{SS}} \qquad \dfrac{\dfrac{\texttt{Ret\_p1}(\widetilde{b'}) \Downarrow_{\texttt{getRet\_p1}}^{\text{RSS}} \widetilde{b'} \qquad \Sigma_0 + \{\widetilde{v \mapsto b'}\} \vdash \| S_0 \|^r \Downarrow^{\text{RSS}} (\widetilde{a}, d)}{\Sigma_0 \vdash \texttt{let}\ \widetilde{v} = \texttt{Proc getRet\_p1}\ (w)\ \texttt{in}\ \| S_0 \|^r \Downarrow^{\text{RSS}} (\widetilde{a}, d)}}{\Sigma_0 \vdash \| \texttt{let}\ \widetilde{v} = \texttt{Proc Reuse}\ p_1\ (\widetilde{w}, w)\ \texttt{in}\ S_0 \|^r \Downarrow^{\text{RSS}} (\widetilde{a}, d)}}{(\widetilde{a}, c((\widetilde{a'_0}, \texttt{Ret\_p1}(\widetilde{b'})), \widetilde{a'_2})) \Downarrow_p^{\text{RSS}} (\widetilde{a}, d)}$$

We can now use our second induction hypothesis (with S_0 and the second leaf) to get $\Sigma_0 + \{\widetilde{v \mapsto b'}\} \vdash S_0 \Downarrow^{\text{REBS}} \widetilde{b}$, which allows us to create our goal derivation:

$$\dfrac{p(\widetilde{y}, c((\widetilde{w}, w), \widetilde{z})) := \dots \qquad \dfrac{\overline{(\widetilde{a'_0}, \texttt{Ret\_p1}(\widetilde{b'})) \Downarrow_{p_1}^{\text{REBS}} \widetilde{b'}} \qquad \overline{\Sigma_0 + \{\widetilde{v \mapsto b'}\} \vdash S_0 \Downarrow^{\text{REBS}} \widetilde{b}}}{\Sigma_0 \vdash \texttt{let}\ \widetilde{v} = \texttt{Proc Reuse}\ p_1\ (\widetilde{w}, w)\ \texttt{in}\ S_0 \Downarrow^{\text{REBS}} \widetilde{b}}}{(\widetilde{a}, c((\widetilde{a'_0}, \texttt{Ret\_p1}(\widetilde{b'})), \widetilde{a'_2})) \Downarrow_p^{\text{REBS}} \widetilde{b}}$$

where the other leaf comes from Lemma 24.

- If c is an initial constructor, the bottom of the tree derivation is of the form:

$$\dfrac{p(\widetilde{y}, c(\widetilde{x})) := \| S \|^r \in \text{R}_{\text{SS}} \qquad \dfrac{\dots}{\Sigma_0 \vdash \| S \|^r \Downarrow^{\text{RSS}} (\widetilde{a}, d)}}{(\widetilde{a}, c(\widetilde{a'})) \Downarrow_p^{\text{RSS}} (\widetilde{a}, d)}$$

Where $r = (p(\widetilde{y}, c(\widetilde{x})) := S) \in \text{R}_{\text{EBS}}$, S does not start with a `Reuse` label, and $\Sigma_0 = \{\widetilde{y \mapsto a}\} + \{\widetilde{x \mapsto a'}\}$.
We use our induction hypothesis on the leaf, and we complete our goal as such:

$$\dfrac{p(\widetilde{y}, c(\widetilde{x})) := S \in \text{R}_{\text{EBS}} \qquad \dfrac{\dots}{\Sigma_0 \vdash S \Downarrow^{\text{REBS}} \widetilde{b}}}{(\widetilde{a}, c(\widetilde{a'})) \Downarrow_p^{\text{REBS}} \widetilde{b}}$$

► Second result, with hypotheses $\begin{cases} r = (p(\widetilde{y}, c(\widetilde{x})) := S_0) \in \text{R}_{\text{EBS}} \\ S \triangleleft S_0 \\ \Sigma(\widetilde{y}) = \widetilde{a} \\ \Sigma \vdash \| S \|^r \Downarrow^{\text{RSS}} (\widetilde{a}, d) \\ (\widetilde{a}, d) \Downarrow_p^{\text{REBS}} \widetilde{b} \\ S \text{ has no } \texttt{Reuse} \text{ label} \end{cases}$

We perform an induction on S.

• If S is of the form $\texttt{Return}\ \widetilde{t}$:

$$\begin{cases} \dfrac{\Sigma(\widetilde{y}, \texttt{Ret\_p}(\widetilde{t})) = (\widetilde{a}, d)}{\Sigma \vdash \texttt{Return}\ (\widetilde{y}, \texttt{Ret\_p}(\widetilde{t})) \Downarrow^{\mathrm{R_{SS}}} (\widetilde{a}, d)} \\ \\ (\widetilde{a}, d) \Downarrow_p^{\mathrm{R_{EBS}}} \widetilde{b} \end{cases}$$

implies, from Lemma 24, that we have $d = \texttt{Ret\_p}(\widetilde{b})$ and so $\widetilde{b} = \Sigma(\widetilde{t})$. So we have our goal:

$$\frac{\Sigma(\widetilde{t}) = \widetilde{b}}{\Sigma \vdash \texttt{Return}\ \widetilde{t} \Downarrow^{\mathrm{R_{EBS}}} \widetilde{b}}$$

• If S is of the form $\texttt{Branch}\ (S_1, \ldots, S_n)$, using the induction hypothesis with S_i:

$$\begin{cases} \dfrac{\Sigma \vdash \parallel S_i \parallel^r \Downarrow^{\mathrm{R_{SS}}} (\widetilde{a}, d)}{\Sigma \vdash \texttt{Branch}\ (\parallel S_1 \parallel^r, \ldots, \parallel S_n \parallel^r) \Downarrow^{\mathrm{R_{SS}}} (\widetilde{a}, d)} \\ \\ (\widetilde{a}, d) \Downarrow_p^{\mathrm{R_{EBS}}} \widetilde{b} \end{cases} \implies \frac{\Sigma \vdash S_i \Downarrow^{\mathrm{R_{EBS}}} \widetilde{b}}{\Sigma \vdash \texttt{Branch}\ (S_1, \ldots, S_n) \Downarrow^{\mathrm{R_{EBS}}} \widetilde{b}}$$

• If S is of the form $\texttt{let}\ \widetilde{v} = K\ \texttt{in}\ S_0$, where K is a filter or a return, using the induction hypothesis with S_0:

$$\begin{cases} \dfrac{\Sigma \vdash K \Downarrow \widetilde{b'} \qquad \overline{\Sigma + \{\widetilde{v \mapsto b'}\} \vdash \parallel S_0 \parallel^r \Downarrow^{\mathrm{R_{SS}}} (\widetilde{a}, d)}}{\Sigma \vdash \texttt{let}\ \widetilde{v} = K\ \texttt{in}\ \parallel S_0 \parallel^r \Downarrow^{\mathrm{R_{SS}}} (\widetilde{a}, d)} \\ \\ (\widetilde{a}, d) \Downarrow_p^{\mathrm{R_{EBS}}} \widetilde{b} \end{cases} \implies \frac{\Sigma \vdash K \Downarrow \widetilde{b'} \qquad \overline{\Sigma + \{\widetilde{v \mapsto b'}\} \vdash S_0 \Downarrow^{\mathrm{R_{EBS}}} \widetilde{b}}}{\Sigma \vdash \texttt{let}\ \widetilde{v} = K\ \texttt{in}\ S_0 \Downarrow^{\mathrm{R_{EBS}}} \widetilde{b}}$$

• If S is of the form $\texttt{let}\ \widetilde{v} = \texttt{Proc}\ (\texttt{New}\, c_0)\ p_1\ \widetilde{t}\ \texttt{in}\ S_0$, then from Lemma 33, $\mathrm{R_{EBS}}$ contains a rule of the form $p(\widetilde{y}, c_0(\widetilde{w}, \widetilde{z})) := \texttt{let}\ \widetilde{v} = \texttt{Proc Reuse}\ p_1\ \widetilde{w}\ \texttt{in}\ S_0$. We have:

$$\begin{cases} \dfrac{\Sigma(\widetilde{y}, c_0(\widetilde{t}, \widetilde{z})) = (\widetilde{a}, d)}{\Sigma \vdash \texttt{Return}\ (\widetilde{y}, c_0(\widetilde{t}, \widetilde{z})) \Downarrow^{\mathrm{R_{SS}}} (\widetilde{a}, d)} \\ \\ (\widetilde{a}, d) \Downarrow_p^{\mathrm{R_{EBS}}} \widetilde{b} \end{cases}$$

From which we can deduce $d = c_0(\Sigma(\widetilde{t}), \Sigma(\widetilde{z}))$, and our second hypothesis corresponds, with $\Sigma' = \widetilde{\{y \mapsto a\}} + \widetilde{\{w \mapsto \Sigma(t)\}} + \widetilde{\{z \mapsto \Sigma(b)\}}$, to a tree of the form:

$$\frac{p(\widetilde{y}, c_0(\widetilde{w}, \widetilde{z})) := \ldots \in \mathrm{R_{EBS}} \qquad \dfrac{\dfrac{\Sigma'(\widetilde{w}) \Downarrow_{p_1}^{\mathrm{R_{EBS}}} \widetilde{b'}}{\Sigma' \vdash \texttt{Proc Reuse}\ p_1\ \widetilde{w} \Downarrow \widetilde{b'}} \qquad \Sigma' + \widetilde{\{v \mapsto b'\}} \vdash S_0 \Downarrow^{\mathrm{R_{EBS}}} \widetilde{b}}{\Sigma' \vdash \texttt{let}\ \widetilde{v} = \texttt{Proc Reuse}\ p_1\ \widetilde{w}\ \texttt{in}\ S_0 \Downarrow^{\mathrm{R_{EBS}}} \widetilde{b}}}{(\widetilde{a}, d) \Downarrow_p^{\mathrm{R_{EBS}}} \widetilde{b}}$$

Note that $\Sigma'(\widetilde{w}) = \Sigma(\widetilde{t})$. Also, using Lemma 10, we can deduce $\Sigma + \widetilde{\{v \mapsto b'\}} \vdash S_0 \Downarrow^{\mathrm{R_{EBS}}} \widetilde{b}$ and create our goal derivation:

$$\frac{\dfrac{\Sigma(\widetilde{t}) \Downarrow_{p_1}^{\mathrm{R_{EBS}}} \widetilde{b'}}{\Sigma \vdash \texttt{Proc}\ (\texttt{New}\, c_0)\ p_1\ \widetilde{t} \Downarrow \widetilde{b'}} \qquad \Sigma + \widetilde{\{v \mapsto b'\}} \vdash S_0 \Downarrow^{\mathrm{R_{EBS}}} \widetilde{b}}{\Sigma \vdash \texttt{let}\ \widetilde{v} = \texttt{Proc}\ (\texttt{New}\, c_0)\ p_1\ \widetilde{t}\ \texttt{in}\ S_0 \Downarrow^{\mathrm{R_{EBS}}} \widetilde{b}}$$

□

Theorem 42 (SS ⇒ EBS)**.**

$$\forall \widetilde{a}, c, \widetilde{a'}, \widetilde{b}, p, \quad (\widetilde{a}, c(\widetilde{a'}))\ (\Downarrow_p^{\mathrm{R_{SS}}})^*\ (\widetilde{a}, \texttt{Ret\_p}(\widetilde{b})) \implies (\widetilde{a}, c(\widetilde{a'})) \Downarrow_p^{\mathrm{R_{EBS}}} \widetilde{b}$$

Proof. We prove it by induction on the number of small-step reductions. If $c(\widetilde{a'}) = \texttt{Ret\_p}(\widetilde{b})$, i.e. there is zero small steps, the result holds from Lemma 24. Otherwise, the induction case comes directly from Lemma 41 above. □

We now certify the infinite case. This time, we cannot use a concatenation result. Instead, we check the infinite sequence satisfies the properties of the coinductive interpretation. Before this, we need the following small lemma to split the infinite sequence when needed.

Lemma 43. Assuming $(p(\widetilde{y}, c((\widetilde{w}, w), \widetilde{z})) := \texttt{let}\ \widetilde{v} = \texttt{Proc Reuse}\ p_1\ (\widetilde{w}, w)\ \texttt{in}\ S) \in \mathrm{R_{EBS}}$, if we have a small-step divergence

$$(\widetilde{b}, c((\widetilde{a}, a_0), \widetilde{b'})) \overset{\infty}{\to}_p$$

then either the first proc call diverges

$$(\widetilde{a}, a_0) \overset{\infty}{\to}_{p_1}$$

or the first proc call finishes and the rest of the computation diverges

$$\exists \widetilde{a'}, \begin{cases} (\widetilde{a}, a_0)\ (\Downarrow_{p_1}^{\mathrm{R_{SS}}})^*\ (\widetilde{a}, \texttt{Ret\_p1}(\widetilde{a'})) \\ (\widetilde{b}, c((\widetilde{a}, \texttt{Ret\_p1}(\widetilde{a'})), \widetilde{b'})) \overset{\infty}{\to}_p \end{cases}$$

Proof. We use the excluded middle axiom, and perform a case disjunction on whether there is a tuple of the form $(\widetilde{b}, c((\widetilde{a}, \texttt{Ret\_p1}(\widetilde{a'})), \widetilde{b'}))$ in the infinite sequence of reduction.

• If there is, we take the first such tuple and we have:

$$(\widetilde{b}, c((\widetilde{a}, a_0), \widetilde{b'}))\ (\Downarrow_p^{\mathrm{R_{SS}}})^*\ (\widetilde{b}, c((\widetilde{a}, \texttt{Ret\_p1}(\widetilde{a'})), \widetilde{b'})) \overset{\infty}{\to}_p$$

Now we show by induction on $(\widetilde{b}, c((\widetilde{a}, a_0), \widetilde{b'}))\ (\Downarrow_p^{\mathrm{R_{SS}}})^*\ (\widetilde{b}, c((\widetilde{a}, \texttt{Ret\_p1}(\widetilde{a'})), \widetilde{b'}))$ that we have $(\widetilde{a}, a_0)\ (\Downarrow_{p_1}^{\mathrm{R_{SS}}})^*\ (\widetilde{a}, \texttt{Ret\_p1}(\widetilde{a'}))$. The base case (zero steps) is trivial. For the induction case, note that a_0 cannot be of the form $\texttt{Ret\_p1}(\ldots)$ as we chose $(\widetilde{b}, c((\widetilde{a}, \texttt{Ret\_p1}(\widetilde{a'})), \widetilde{b'}))$ to be the first of this form. We then use Lemma 27 to transform the first step, and the induction hypothesis to transform the rest.

• If there is no such tuple, we show $(\widetilde{a}, a_0) \overset{\infty}{\to}_{p_1}$ by coinduction. We cut our hypothesis and there is $\widetilde{t}$ such that $(\widetilde{b}, c((\widetilde{a}, a_0), \widetilde{b'}))\ \Downarrow_p^{\mathrm{R_{SS}}}\ \widetilde{t} \overset{\infty}{\to}_p$. From Lemma 27, we have $\widetilde{t} = (\widetilde{b}, c((\widetilde{a}, a_0'), \widetilde{b'}))$ with $(\widetilde{a}, a_0)\ \Downarrow_{p_1}^{\mathrm{R_{SS}}}\ (\widetilde{a}, a_0')$. Also, $(\widetilde{b}, c((\widetilde{a}, a_0'), \widetilde{b'})) \overset{\infty}{\to}_p$ still has the property of not having any intermediate tuple of the form $(\widetilde{b}, c((\widetilde{a}, \texttt{Ret\_p1}(\widetilde{a'})), \widetilde{b'}))$, so we can use our induction hypothesis with it to conclude.

□

Theorem 44 (Div: SS ⇒ EBS)**.** For all proc p and terms $\widetilde{a}$,

$$\widetilde{a} \overset{\infty}{\to}_p \implies \widetilde{a} \Uparrow_p^{\mathrm{R_{EBS}}}$$

Proof. Since $\widetilde{a} \Uparrow_p^{\mathrm{R_{EBS}}}$ is defined mutually with $\Sigma \vdash S \Uparrow^{\mathrm{R_{EBS}}}$, we show two properties at the same time:

- $\forall \widetilde{a}, p, \quad \widetilde{a} \overset{\infty}{\to}_p \implies \widetilde{a} \Uparrow_p^{\mathrm{R_{EBS}}}$
- $\forall p, \widetilde{y}, c, \widetilde{x}, S_0, \Sigma, S, \widetilde{b}, \quad \begin{cases} r = (p(\widetilde{y}, c(\widetilde{x})) := S_0) \in \mathrm{R_{EBS}} \\ S \triangleleft S_0 \\ \Sigma \vdash \| S \|^r \Downarrow^{\mathrm{R_{SS}}} \widetilde{b} \\ \widetilde{b} \overset{\infty}{\to}_p \\ S \text{ has no Reuse label} \end{cases} \implies \Sigma \vdash S \Uparrow^{\mathrm{R_{EBS}}}$

We note the predicates

$$Q = \{(\widetilde{a}, p) \mid \widetilde{a} \overset{\infty}{\to}_p\}$$

$$Q' = \left\{ (\Sigma, S) \;\middle|\; \begin{array}{c} \exists p, \widetilde{b}, (p(\ldots) := S_0) \in \mathrm{R_{EBS}}, \text{ such that} \\ S \triangleleft S_0 \wedge \Sigma \vdash \| S \|^r \Downarrow^{\mathrm{R_{SS}}} \widetilde{b} \wedge \widetilde{b} \overset{\infty}{\to}_p \wedge S \text{ has no Reuse label} \end{array} \right\}$$

To prove both these result by coinduction, we show that the two predicates satisfy the properties corresponding to the rules of Figure 9. I.e., for every element of Q and Q', we can create a derivation of the goal statement using at least one rule of Figure 9 and using elements of Q and Q' as leaves.

▶ For elements of Q. We assume $(\widetilde{a}, c(\widetilde{a'})) \overset{\infty}{\to}_p$.

• If c is an initial constructor, then $\mathrm{R_{EBS}}$ contains a rule r of the form $p(\widetilde{y}, c(\widetilde{x})) := S_0$ where S_0 has no label Reuse. We note $\Sigma = \{\widetilde{y \mapsto a}\} + \{\widetilde{x \mapsto a'}\}$. Splitting the first small-step of the infinite reduction, we have $\widetilde{b}$ such that $\Sigma \vdash \| S_0 \|^r \Downarrow^{\mathrm{R_{SS}}} \widetilde{b}$ and $\widetilde{b} \overset{\infty}{\to}_p$. We immediately have $(\Sigma, S_0) \in Q'$ using r and $S_0 \triangleleft S_0$, so we can start constructing a derivation as follows.

$$\frac{(p(\widetilde{y}, c(\widetilde{x})) := S_0) \in \mathrm{R_{EBS}} \qquad (\Sigma, S_0) \in Q'}{(\widetilde{a}, c(\widetilde{a'})) \Uparrow_p^{\mathrm{R_{EBS}}}}\ \textsc{Div-tuple}$$

• If c is a new constructor, then $\mathrm{R_{EBS}}$ contains a rule r of the form

$$p(\widetilde{y}, c(\widetilde{w}, \widetilde{z})) := \texttt{let}\ \widetilde{v} = \texttt{Proc Reuse}\ p_1\ \widetilde{w}\ \texttt{in}\ S_0$$

where S_0 has no label Reuse. We have $\widetilde{a'} = (\widetilde{a_1}, \widetilde{a_2})$, we note $\Sigma = \{\widetilde{y \mapsto a}\} + \{\widetilde{w \mapsto a_1}\} + \{\widetilde{z \mapsto a_2}\}$. We use Lemma 43 to perform a case disjunction on the behavior of $\widetilde{a}, c(\widetilde{a_1}, \widetilde{a_2}) \overset{\infty}{\to}_p$.

⊛ If $\widetilde{a_1} \xrightarrow{\infty}_{p_1}$, then we can start constructing our goal as such:

$$\dfrac{p(\widetilde{y}, c(\widetilde{w}, \widetilde{z})) := \ldots \in \mathrm{R_{EBS}} \qquad \dfrac{\dfrac{(\widetilde{a_1}, p_1) \in Q}{\Sigma \vdash \mathtt{Proc}\ p_1\ \widetilde{w} \Uparrow^{\mathrm{R_{EBS}}}}\ \textsc{Div-Pc}}{\Sigma \vdash \mathtt{let}\ \widetilde{v} = \mathtt{Proc\ Reuse}\ p_1\ \widetilde{w}\ \mathtt{in}\ S_0 \Uparrow^{\mathrm{R_{EBS}}}}\ \textsc{Div-LetL}}{\widetilde{a}, c(\widetilde{a_1}, \widetilde{a_2}) \Uparrow_p^{\mathrm{R_{EBS}}}}\ \textsc{Div-tuple}$$

⊛ Else there is $\widetilde{a_1'}$ such that $\widetilde{a_1}\ (\Downarrow_{p_1}^{\mathrm{R_{SS}}})^*\ (\ldots, \mathtt{Ret\_p1}(\widetilde{a_1'}))$ and $(\widetilde{a}, c(\ldots, \mathtt{Ret\_p1}(\widetilde{a_1'}), \widetilde{a_2})) \xrightarrow{\infty}_p$. Using Theorem 42 on the first point, we get $\widetilde{a_1} \Downarrow_{p_1}^{\mathrm{R_{EBS}}} \widetilde{a_1'}$. Using Lemma 29 on the second point (after splitting a step), we get there is $\widetilde{b}$ such that $\Sigma' \vdash \parallel S_0 \parallel^r \Downarrow^{\mathrm{R_{SS}}} \widetilde{b}$ and $\widetilde{b} \xrightarrow{\infty}_p$, where $\Sigma' = \widetilde{\{y \mapsto a\}} + \widetilde{\{z \mapsto a_2\}} + \widetilde{\{v \mapsto a_1'\}}$. We can add the mapping of the unused variables $\widetilde{w}$ with Lemma 10, and so $(\Sigma + \widetilde{\{v \mapsto a_1'\}}, S_0) \in Q'$ using rule r, and we can construct our goal as such:

$$\dfrac{p(\widetilde{y}, c(\widetilde{w}, \widetilde{z})) := \ldots \in \mathrm{R_{EBS}} \qquad \dfrac{\dfrac{\widetilde{a_1} \Downarrow_{p_1}^{\mathrm{R_{EBS}}} \widetilde{a_1'}}{\Sigma \vdash \mathtt{Proc}\ p_1\ \widetilde{w} \Downarrow^{\mathrm{R_{EBS}}} \widetilde{a_1'}} \qquad (\Sigma + \widetilde{\{v \mapsto a_1'\}}, S_0) \in Q'}{\Sigma \vdash \mathtt{let}\ \widetilde{v} = \mathtt{Proc\ Reuse}\ p_1\ \widetilde{w}\ \mathtt{in}\ S_0 \Uparrow^{\mathrm{R_{EBS}}}}\ \textsc{Div-LetR}}{\widetilde{a}, c(\widetilde{a_1}, \widetilde{a_2}) \Uparrow_p^{\mathrm{R_{EBS}}}}\ \textsc{Div-tuple}$$

► For elements of Q'. We assume $\begin{cases} r = (p(\widetilde{y}, c(\widetilde{x})) := S_0) \in \mathrm{R_{EBS}} \\ S \triangleleft S_0 \\ \Sigma \vdash \parallel S \parallel^r \Downarrow^{\mathrm{R_{SS}}} \widetilde{b} \\ \widetilde{b} \xrightarrow{\infty}_p \\ S \text{ has no } \mathtt{Reuse} \text{ label} \end{cases}$

First, we can see certain cases for S are not possible. S cannot be a procedure or filter, because we delayed returns to create the set $\mathrm{R_{EBS}}$. Also, S cannot be a return, because $\widetilde{b}$ would be of the form $(\ldots, \mathtt{Ret\_p}(\ldots))$ and $\widetilde{b} \xrightarrow{\infty}_p$ contradicts this.

• If $S = \mathtt{Branch}\ (S_1, \ldots, S_n)$, then $\Sigma \vdash \mathtt{Branch}\ (\parallel S_1 \parallel^r, \ldots, \parallel S_n \parallel^r) \Downarrow^{\mathrm{R_{SS}}} \widetilde{b}$ implies there is S_i such that $\Sigma \vdash \parallel S_i \parallel^r \Downarrow^{\mathrm{R_{SS}}} \widetilde{b}$. We have $(\Sigma, S_i) \in Q'$ using the same rule and terms, since $S_i \triangleleft S \triangleleft S_0$. We can then start the derivation of the goal as follows.

$$\dfrac{S_i \in (S_1, \ldots, S_n) \qquad (\Sigma, S_i) \in Q'}{\Sigma \vdash \mathtt{Branch}\ (S_1, \ldots, S_n) \Uparrow^{\mathrm{R_{EBS}}}}\ \textsc{Div-Br}$$

• If $S = (\mathtt{let}\ \widetilde{v} = K\ \ \mathtt{in}\ \ S')$ where K is a return or a filter, then there is $\widetilde{b'}$ such that $\Sigma \vdash K \Downarrow^{\mathrm{R_{SS}}} \widetilde{b'}$ and $\Sigma' \vdash \parallel S' \parallel^r \Downarrow^{\mathrm{R_{SS}}} \widetilde{b}$, where we note $\Sigma' = \Sigma + \widetilde{\{v \mapsto b'\}}$. We have $(\Sigma', S') \in Q'$ and can start the derivation of the goal as follows.

$$\dfrac{\Sigma \vdash K \Downarrow^{\mathrm{R_{EBS}}} \widetilde{b'} \qquad (\Sigma', S') \in Q'}{\Sigma \vdash \mathtt{let}\ \widetilde{v} = K\ \ \mathtt{in}\ \ S' \Uparrow^{\mathrm{R_{EBS}}}}\ \textsc{Div-LetR}$$

Note that we have $\Sigma \vdash K \Downarrow^{\mathrm{R_{SS}}} \widetilde{b'} \iff \Sigma \vdash K \Downarrow^{\mathrm{R_{EBS}}} \widetilde{b'}$ for filters and returns, as evaluating these skelements do not look up any rule.

• If $S = (\mathtt{let}\ \widetilde{v} = \mathtt{Proc}\ (\mathtt{New}\, c_0)\ p_1\ \widetilde{t}\ \ \mathtt{in}\ \ S')$ then, from Lemma 33, $\mathrm{R_{EBS}}$ contains a rule r' of the form

$$p(\widetilde{y}, c_0(\widetilde{w}, \widetilde{z})) := \mathtt{let}\ \widetilde{v} = \mathtt{Proc\ Reuse}\ p_1\ \widetilde{w}\ \ \mathtt{in}\ \ S'$$

Let $\widetilde{a_0} = \Sigma(\widetilde{y})$, $\widetilde{a_1} = \Sigma(\widetilde{t})$, and $\widetilde{a_2} = \Sigma(\widetilde{z})$. We have $\parallel S \parallel^r = \mathtt{Return}\ (\widetilde{y}, c_0(\widetilde{t}, \widetilde{z}))$ and so $\widetilde{b} = (\widetilde{a_0}, c_0(\widetilde{a_1}, \widetilde{a_2}))$. We use Lemma 43 to perform a case disjunction on the behavior of $\widetilde{a_0}, c_0(\widetilde{a_1}, \widetilde{a_2}) \xrightarrow{\infty}_p$.

⊛ If $\widetilde{a_1} \xrightarrow{\infty}_{p_1}$, then we can start constructing our goal as such:

$$\dfrac{\dfrac{\dfrac{(\widetilde{a_1}, p_1) \in Q}{\Sigma \vdash \mathtt{Proc}\ p_1\ \widetilde{t} \Uparrow^{\mathrm{R_{EBS}}}}\ \textsc{Div-Pc}}{\Sigma \vdash \mathtt{let}\ \widetilde{v} = \mathtt{Proc}\ (\mathtt{New}\, c_0)\ p_1\ \widetilde{t}\ \ \mathtt{in}\ \ S' \Uparrow^{\mathrm{R_{EBS}}}}\ \textsc{Div-LetL}}$$

⊛ Else there is $\widetilde{a_1'}$ such that $\widetilde{a_1}\ (\Downarrow_{p_1}^{\mathrm{R_{SS}}})^*\ (\ldots, \mathtt{Ret\_p1}(\widetilde{a_1'}))$ and $(\widetilde{a_0}, c_0(\ldots, \mathtt{Ret\_p1}(\widetilde{a_1'}), \widetilde{a_2})) \xrightarrow{\infty}_p$. Using Theorem 42 on the first point, we get $\widetilde{a_1} \Downarrow_{p_1}^{\mathrm{R_{EBS}}} \widetilde{a_1'}$. Using Lemma 29 on the second point (after splitting a step), we get there is $\widetilde{b'}$ such that $\Sigma' \vdash \parallel S' \parallel^{r'} \Downarrow^{\mathrm{R_{SS}}} \widetilde{b'}$ and $\widetilde{b'} \xrightarrow{\infty}_p$, where $\Sigma' = \widetilde{\{y \mapsto a_0\}} + \widetilde{\{z \mapsto a_2\}} + \widetilde{\{v \mapsto a_1'\}}$. We can add the mapping of the unused variables $\widetilde{w}$ with Lemma 10, and so $(\Sigma + \widetilde{\{v \mapsto a_1'\}}, S') \in Q'$

using rule r', and we can construct our goal as such:

$$\dfrac{\dfrac{\widetilde{a_1} \Downarrow_{p_1}^{\mathrm{R_{EBS}}} \widetilde{a'_1}}{\Sigma \vdash \mathtt{Proc}\ p_1\ \widetilde{t} \Downarrow^{\mathrm{R_{EBS}}} \widetilde{a'_1}} \qquad (\Sigma + \{\widetilde{v \mapsto a'_1}\}, S') \in Q'}{\Sigma \vdash \mathtt{let}\ \widetilde{v} = \mathtt{Proc}\ (\mathtt{New}\, c_0)\ p_1\ \widetilde{t}\ \mathtt{in}\ S' \Uparrow^{\mathrm{R_{EBS}}}} \textsc{Div-LetR}$$

□