Skeletal semantics transformations

Guillaume Ambal

UNIVERSITÉ DE RENNES 1

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE RENNES 1

ÉCOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : *Informatique*

Par
Guillaume AMBAL

Skeletal Semantics Transformations

Thèse présentée et soutenue à Rennes, le 19 Octobre 2022
Unité de recherche : IRISA (UMR 6074)

Rapporteurs avant soutenance :

Dariusz BIERNACKI Associate Professor — University of Wrocław, Poland
Christine TASSON Professeure — Sorbonne Université, Paris

Composition du Jury :

Président :	Nathalie BERTRAND	Directrice de recherche — Inria Rennes
Examinateurs :	Dariusz BIERNACKI	Associate Professor — University of Wrocław, Pologne
	Christine TASSON	Professeure — Sorbonne Université, Paris
	Daniel HIRSCHKOFF	Maître de Conférences — LIP, ENS de Lyon
	Jean-Marie MADIOT	Chargé de Recherche — Inria Paris
Dir. de thèse :	Alan SCHMITT	Directeur de Recherche — Inria Rennes
Co-enc. de thèse :	Sergueï LENGLET	Maître de Conférences — Université de Lorraine, Nancy

Remerciements

Merci à mes directeur et ~~co-directeur~~ co-encadrant de thèse de m'avoir guidé jusque là.
Merci aux rapporteurs d'avoir pris le temps de relire ce pavé.

Merci à toute la liste Platypus pour les discussions et les soirées.

Merci à ma famille pour son soutien et les kiwis.
Merci à ma sœur pour ses élans chaotiques arbitraires.

Muxu.

RÉSUMÉ EN FRANÇAIS

Langages et Sémantiques

En informatique, le développement d'algorithmes et de logiciels se fait au moyen de *langages de programmation*. Ces langages permettent, indirectement, la communication entre les programmeurs et les ordinateurs. Définir formellement ces langages devient donc primordial afin d'éviter les problèmes d'interprétation entre humains et machines.

La définition d'un langage se compose d'une *syntaxe* et d'une *sémantique*. La syntaxe exprime quels objets sont considérés comme des programmes valides, et peut facilement être formalisée à l'aide d'une grammaire. Par exemple, en supposant l'existence de variables (x) et d'entiers (n), on peut définir des expressions (e) et des programmes (c) avec la grammaire suivante :

$$e ::= x \mid n \mid \texttt{Plus}(e_1, e_2) \mid \dots$$
$$c ::= \texttt{skip} \mid \texttt{while}\, e\, \texttt{do}\, c \mid \dots$$

Ensuite, la sémantique d'un langage donne un sens à ces programmes. Dans ce document, nous nous concentrons sur les sémantiques opérationnelles, où un programme est interprété comme une séquence de calculs. Une sémantique opérationnelle explique donc comment exécuter un programme. Formaliser une sémantique opérationnelle peut s'avérer délicat. Pour de petits langages, il est préférable d'utiliser des règles d'inférence qui seront sans ambiguïté. Pour la syntaxe ci-dessus, on définit les règles suivantes.

$$\frac{s(x) = n}{s, x \Downarrow s, n} \qquad \frac{s, e_1 \Downarrow s_1, n_1 \qquad s_1, e_2 \Downarrow s_2, n_2 \qquad n_1 + n_2 = n}{s, \texttt{Plus}(e_1, e_2) \Downarrow s_2, n} \qquad \dots$$

$$\frac{s, e \Downarrow s_1, \texttt{true} \qquad s_1, c \Downarrow s_2 \qquad s_2, \texttt{while}\, e\, \texttt{do}\, c \Downarrow s_3}{s, \texttt{while}\, e\, \texttt{do}\, c \Downarrow s_3} \qquad \frac{s, e \Downarrow s_1, \texttt{false}}{s, \texttt{while}\, e\, \texttt{do}\, c \Downarrow s_1}$$

Évaluer $\texttt{Plus}(e_1, e_2)$ consiste à évaluer successivement e_1 et e_2, puis à additionner les résultats. Pour évaluer une boucle $\texttt{while}\, e\, \texttt{do}\, c$, on évalue d'abord la condition e ; si le résultat est vrai, on évalue la commande c avant de boucler sur $\texttt{while}\, e\, \texttt{do}\, c$; sinon, on arrête le calcul.

Pour des langages plus conséquents, les règles d'inférence peuvent être encombrantes et difficiles à lire, écrire et modifier. Il est donc fréquent de décrire la sémantique d'un langage à l'aide d'explications en anglais. Tous les langages ne sont ainsi pas définis avec le même niveau

de précision. Par exemple, la sémantique de JavaScript est très détaillée et peut être vue comme une traduction en anglais de règles d'inférence, alors que la documentation de Python est une description de plus haut niveau du comportement attendu.

Ces définitions informelles limitent les utilisations possibles des sémantiques des langages de programmation. Il est difficile de les utiliser pour de la vérification ou de la certification. À l'inverse, une sémantique mécanisée sur ordinateur peut servir de référence pour prouver des propriétés du langage, ou pour certifier certains outils associés tels qu'un interpréteur ou un débogueur.

Cette mécanisation peut être faite dans des logiciels génériques, comme des assistants de preuve (Coq [52], Isabelle [39], ...), ou avec des outils spécialisés tels que $\mathbb{K}$ [48] et les sémantiques squelettiques [17]. Dans ce document, nous utilisons les sémantiques squelettiques qui permettent une manipulation très simple des sémantiques opérationnelles.

Sémantiques Squelettiques

Les sémantiques squelettiques sont un cadre logique pour formaliser les sémantiques opérationnelles de langages de programmation à l'aide d'un petit métalangage nommé Skel. La sémantique d'un langage est représentée par un objet écrit dans la syntaxe de Skel. Cette définition peut être vue comme un morceau de code (du métalangage) ou un interpréteur. Cela permet aux sémantiques d'être plus facile à écrire, lire et manipuler. Une telle définition est rendue formelle et non ambiguë indirectement, via la sémantique du métalangage Skel définie par des règles d'inférence.

Le métalangage Skel est pensé pour permettre une grande modularité des sémantiques squelettiques. Le principe fondamental est que l'on peut définir précisément la structure d'une fonction d'évaluation (séquences d'opérations, appels récursifs, branchements non déterministes, ...) tout en gardant abstrait le comportement des opérations de base (comparaison de deux entiers, mise à jour d'un environnement, ...). La structure peut ainsi être analysée, transformée, ou certifiée indépendamment des choix d'implémentation des opérations de base.

Par exemple, on peut formaliser le petit langage précédent de la manière suivante :

```
type ident

type expr =
| Var ident
| Plus of expr * expr
| ...
type stmt =
| Skip
| While of expr * stmt
| ...
```

```
val add : ...

hook hexpr (s: state, e: expr) matching e =
| Var (x) -> ...
| Plus (e1, e2) ->
  let (s1, n1) = hexpr (s, e1) in
  let (s2, n2) = hexpr (s1, e2) in
  let n = add (n1, n2) in
  (s2, n)
| ...
```

On garde ainsi l'idée qu'évaluer $\texttt{Plus}(e_1, e_2)$ consiste à évaluer successivement e_1 et e_2 (par appels récursifs), puis à additionner les résultats. Ceci est indépendant du comportement de l'addition (`add`).

Les sémantiques squelettiques, et leur implémentation en OCaml [36] appelée Necro [21], peuvent ensuite traiter automatiquement ces définitions. Différents outils permettent notamment d'exporter le langage en une définition Coq, ou de générer automatiquement un interpréteur et un débogueur à partir de la sémantique.

Le cadre des sémantiques squelettiques a été progressivement enrichi, et ce document couvre deux versions différentes du métalangage Skel. La différence principale étant que la version utilisée en Partie I de ce document n'autorise pas les fonctions d'ordre supérieur.

Contributions

Il existe différents formats de sémantiques opérationnelles. Par exemple, on trouve des sémantiques à grand pas, reliant un programme à son résultat final, des sémantiques à petit pas, explicitant les calculs intermédiaires, et des machines abstraites, détaillant très précisément la gestion des différents éléments. Ces formats ont chacun leurs avantages et inconvénients. Les sémantiques à grand pas sont les plus intuitives et les plus répandues, mais ne peuvent pas rendre compte de tous les comportements observables avec d'autres types de sémantique. Selon les propriétés que l'on souhaite vérifier, il est parfois nécessaire de manipuler plusieurs version de la sémantique d'un même langage.

Cependant, formaliser plusieurs fois le même langage à la main n'est pas idéal. En plus de nécessiter davantage de travail, il faut s'assurer que les différentes définitions sont toutes équivalentes. Il est donc préférable de développer des méthodes systématiques pour générer toutes ces définitions à partir d'une seule, avec le plus de garanties possibles. Ce document approfondit ainsi les interdérivations de sémantiques opérationnelles, à la fois au niveau des langages utilisateurs et du métalangage Skel.

Transformation Automatique de Grand Pas en Petit Pas

La première partie de cette thèse présente une nouvelle méthode pour transformer automatiquement une sémantique squelettique à grand pas en une sémantique squelettique à petit pas équivalente.

La difficulté principale de cette transformation en petit pas est la création de nouveaux constructeurs. En effet, la syntaxe initiale n'est pas toujours suffisante pour exprimer les calculs intermédiaires qui peuvent apparaître durant une évaluation. Cette situation apparaît en général lorsqu'un argument a besoin d'être évalué plusieurs fois. Par exemple, pour le constructeur $\texttt{Plus}(e_1, e_2)$, la règle d'inférence commence par une prémisse $s, e_1 \Downarrow s_1, n_1$. En petit pas, si l'on

commence le calcul de e_1 avec une prémisse $s, e_1 \to s', e_1'$, on peut sans problèmes écraser la valeur initiale de e_1. Notre méthode génère donc la règle suivante.

$$\frac{s, e_1 \to s', e_1'}{s, \texttt{Plus}(e_1, e_2) \to s', \texttt{Plus}(e_1', e_2)}$$

Pour le constructeur `while` e `do` c en revanche, on ne peut pas avoir de règle similaire. Si on commence à évaluer e avec une prémisse $s, e \to s', e'$, on ne peut pas écraser la valeur initiale de e avec e', puisque l'on doit potentiellement réévaluer `while` e `do` c plus tard. La transformation étend alors la syntaxe, et introduit un nouveau constructeur (nommé `While1`) capable de stocker le résultat intermédiaire nécessaire.

Cette transformation se découpe en plusieurs étapes. Notamment, une phase d'analyse parcourt la sémantique pour déterminer quels points de programme nécessitent l'ajout de nouveaux constructeurs, et essaie de réutiliser les constructeurs existants autant que possible. Une dernière phase produit la sémantique à petit pas en utilisant ces nouveaux constructeurs.

Il est également important de vérifier que cet algorithme est correct, c'est à dire que la sémantique à petit pas en sortie est effectivement équivalente à la sémantique à grand pas fournie en entrée. Comme la phase d'analyse est complexe et sujette à des optimisations, la version complète de l'algorithme se prête mal à une certification directe. À la place, nous développons deux approches différentes et complémentaires pour s'assurer que la transformation produit le résultat escompté.

Tout d'abord, nous présentons une preuve papier que la transformation sans analyse est correcte. En pratique, cette version simplifiée produit une sémantique à petit pas moins optimisée avec plus de constructeurs que nécessaire. La preuve mathématique garantie que la stratégie fondamentale est valide sur tous les langages.

Ensuite, pour attester de l'implémentation de la version complète, nous générons également, pour chaque langage, un certificat Coq de l'équivalence entre les sémantiques d'entrée et de sortie. Necro permet d'obtenir les définitions Coq des deux sémantiques, et la stratégie connue de preuve d'équivalence ne dépend pas de ces sémantiques. Il nous est donc possible de générer un script de preuve Coq spécialisé pour chaque langage. Cette certification à posteriori permet à la fois de garantir l'équivalence directement sur le langage qui nous intéresse, mais aussi de ne pas craindre la présence de bugs dans l'implémentation puisque l'on ne vérifie que le résultat final.

Combiner cette transformation à l'environnement Necro nous permet également de générer automatiquement un interpréteur petit pas pour n'importe quel langage.

Nouvelles Sémantiques pour le Métalangage

Comme la sémantique squelettique d'un langage est écrite dans le métalangage Skel, elle n'a de sens qu'à travers la sémantique du métalangage lui-même. Skel est défini avec une sémantique à grand pas non déterministe, appelée *interprétation concrète*. Bien qu'utile pour de nombreuses preuves sur les langages et leurs programmes, cette interprétation concrète ne permet ni de raisonner sur des programmes qui ne terminent pas, ni d'exécuter ces programmes. Pour pallier cela, nous proposons deux nouvelles interprétations de Skel sous forme de machines abstraites déterministe et non déterministe.

Nous ne créons pas ces nouvelles sémantiques à la main, mais nous les générons en utilisant la technique de *correspondence fonctionnelle* [5] à partir de l'interprétation concrète. Il s'agit d'une méthode systématique pour déduire une machine abstraite à partir d'une sémantique à grand pas. Elle combine plusieurs transformation connues pour progressivement réécrire la sémantique en une forme interruptible.

Puisque l'interprétation concrète est non déterministe, lui appliquer la correspondance fonctionnelle nous permet d'obtenir une machine abstraite non déterministe équivalente. Cette nouvelle sémantique est intéressante, mais ne permet toujours pas d'exécuter un programme en pratique. Pour créer une machine abstraite déterministe, nous utilisons une version plus sophistiquée de la correspondance fonctionnelle. Cela consiste à introduire des points de contrôle dans la partie non déterministe de la sémantique, permettant à l'exécution de revenir en arrière en cas de problème. Nous obtenons ainsi une sémantique déterministe et exécutable pour le métalangage Skel. Cette machine abstraite est correcte par rapport à l'interprétation concrète : lorsque la machine obtient un résultat, il s'agit d'un résultat correcte. En revanche, tous les résultats corrects ne sont pas forcement accessible

Les deux machines abstraites (non déterministe et déterministe) ont été formalisées dans l'assistant de preuve Coq, et nous avons prouvé que ces deux nouvelles sémantiques sont effectivement respectivement équivalentes et correctes par rapport à l'interprétation concrète.

Nous utilisons également le système d'extraction de Coq pour obtenir une version OCaml exécutable de la machine abstraite déterministe, exécutant le métalangage Skel. En utilisant les outils de Necro, cette machine peut être automatiquement instancié pour n'importe quelle sémantique squelettique, fournissant ainsi un interpréteur certifié pour tout langage défini en Skel.

Plan du Manuscrit

La Section 2 de ce document présente les principales formes de sémantiques opérationnelles : sémantique à grand pas, sémantique à petit pas, sémantique à réduction, et machine abstraites.

La Partie I présente une stratégie pour transformer automatiquement une sémantique sque-

lettique à grand pas en une sémantique squelettique à petit pas. Cette transformation est implémentée dans une ancienne version de Necro pour les sémantiques squelettiques sans fonctions d'ordre supérieur. Nous montrons, par une preuve papier, qu'une version légèrement simplifiée de la transformation est correcte. Nous ne certifions pas la version complète de l'algorithme, mais pour chaque utilisation nous générons une garantie de résultat. L'implémentation produit donc automatiquement à la fois une sémantique à petit pas équivalente, et un script Coq vérifiant cette équivalence.

La Partie II de ce manuscrit se concentre sur le métalangage (Skel) des sémantiques squelettiques d'ordre supérieure actuelles. Nous utilisons des méthodes connues pour transformer l'interprétation concrète et ainsi générer des machines abstraites déterministe et non déterministe pour Skel. Nous montrons en Coq que la version non déterministe est équivalente à la sémantique de départ, et que la version déterministe est correcte par rapport à la sémantique à grand pas. Extraire la version déterministe nous permet d'obtenir un interpréteur certifié pour tout langage défini en Skel.

Table of Contents

Introduction

Chapter 1

Introduction

Software development uses *programming languages* as a communication tool between human programmers and our computers. Formal definitions of these languages are thus crucial to prevent translation issues between humans and electronic devices.

The *syntax* of a language expresses which objects are valid programs of the language. Programs can be considered mathematical objects, and are easily described with a grammar. The *semantics* of a language gives meaning to these programs and are essential to reason about them. Depending on the use case, this "meaning" could be a logical relation, or even another mathematical object. In this document, we limit ourselves to *operational semantics* where a program is interpreted as a sequence of computations. I.e., an operational semantics explains how we can run a program.

Formalizing operational semantics can be problematic. When able, as with small languages, it is preferable to use unambiguous inference rules. However, for more substantial languages, inference rules can be difficult to read, write, and maintain. As such, semantic behaviors are often stated in plain text, with varied levels of detail. For instance, the semantics of JavaScript is very precise and can be seen as an English translation of inference rules, while the Python documentation offers higher-level explanations of the expected behaviors.

Informal paper definitions are not adapted for the certification of language properties or the verification of algorithms or tools. To this end, it is preferable to mechanize the semantics on a computer, either in a generic proof assistant or in a specialized framework. Theorem provers are comprehensive tools that do not limit the range of expressible semantics, but they can require an effort of calibration for each language. Notable examples include JSCert [16], a Coq [52] formalization of the semantics of JavaScript, and CakeML [33], a language fully formalized in the Isabelle [39] proof assistant.

Specialized frameworks can ease the formalization process and automatically provide relevant tools, such as generating an executable interpreter from a semantics. For instance, Ott [50] and Lem [42] are lightweight tools providing a definition format—akin to inference rules—that can then be exported to multiple different formats, including LaTeX typesetting and several proof assistants. The predominant tool for language formalization is the $\mathbb{K}$ framework [48]. Semantics of languages are defined using small-step rewriting systems that can be applied under any

context. The authors also automatically provides an executable interpreter and access to their own verification tool. However, users are limited by the implementation of the framework and the rewriting semantics. Besides, there is no library to access the internal representation of the language, and no practical way to perform semantics manipulation.

In this document, we use skeletal semantics [17] where semantics of languages are expressed as objects of a meta-language called Skel. It is a convenient choice because it offers a straightforward access to the manipulation of user-defined semantics, and because definitions can be easily exported to Coq for further certifications. The use of a meta-language means semantics can easily be handled as data structures, and different semantics of the meta-language provide different interpretations of these objects. Since the skeletal semantics framework has been expanded during the PhD, this document actually covers two slightly different versions, the main difference being that the skeletal semantics used in the first half does not allow higher-order functions.

The framework is not the only choice to make to formally describe a semantics: the format of the semantics itself is also of importance. Within the domain of operational semantics, one could use for instance a big-step semantics to relate a program to its result, a small-step semantics to express intermediate computations, or an abstract machine to precisely describe the manipulation of the different arguments. Each format has benefits and drawbacks, e.g., big-step semantics are concise and intuitive, but cannot be used to express partial computations. Depending on the kind of properties we are interested in, it is sometimes necessary to manipulate several versions of the semantics of a language. For instance, the semantics of the high-level language of CompCert [34] was initially defined in big-step, and was later rewritten as small-step.

However, defining the same language multiple times by hand is far from optimal. Besides the additional formalization work, one has to make sure the different semantics are all equivalent. A better approach is to build generic transformations able to generate one format from another, and use them to automatically derive everything from a single main semantics.

Our work focuses on this idea of interderiving operational semantics formats. We apply known techniques to the skeletal semantics meta-language to expand its use cases. We also present a new interderivation strategy from big-step to small-step skeletal semantics.

1.1 Contributions

The first part of this document introduces a novel generic method to automatically transform a big-step skeletal semantics into an equivalent small-step skeletal semantics. It has also been implemented in OCaml, within the Necro [21] toolbox for the manipulation of skeletal semantics. Combined with other tools, we can also automatically generate an OCaml small-step interpreter for any language.

As the initial syntax is, in general, not sufficient to express partial computations, the trans-

formation needs to extend this syntax with new constructors. The main difficulty comes from determining which new constructors are actually necessary, as to not clutter the output small-step semantics, and is settled through a complex analysis phase.

Ensuring the correctness of this transformation is essential. I.e., we want to make sure that the output small-step semantics is equivalent to the input big-step semantics. Since the analysis phase is involved and contains optimizations, certifying the complete transformation would be challenging. Instead, we present two complementary results to ensure the correctness of the approach.

First, we produce a paper proof that the transformation without the analysis phase is correct. This version would produce a less optimized small-step semantics, with more constructors than necessary. This proof establishes that the fundamental strategy is valid and applicable to any language.

Second, to confirm the correctness of the implementation of the full transformation, we generate for each language an a posteriori Coq certificate of equivalence between input and output semantics. The Necro toolbox can export a semantics into a Coq definition file, and we make use of it to check the result of the transformation. This ensures the equivalence at the level of the user-defined language, and bypasses certifying the analysis optimizations and the implementation.

The second part of this document focuses on the semantics of the Skel meta-language. Its reference definition is a non-deterministic big-step semantics formalized with inference rules, called *concrete interpretation*. While useful to prove some properties of a language or of programs, this interpretation cannot reason about non-terminating programs and cannot run them.

We thus derive two new alternative semantics, in the form of non-deterministic and deterministic abstract machines. We use the known strategy of *functional correspondence* [5] to generate the new semantics. It is a generic interderivation technique, combining several known transformations, to progressively rewrite a big-step semantics into an equivalent abstract machine. Applying it on the concrete interpretation produces a non-deterministic abstract machine. The deterministic variant is obtained similarly, but we introduce checkpoints and backtracking by hand during the transformation to force a deterministic evaluation.

Both abstract machines are formalized in the Coq proof assistant, and we certify they behave as expected: the non-deterministic version is equivalent to the initial concrete interpretation; the determining version is sound (but not complete) with respect to the other two semantics.

We use the Coq extraction mechanism on the deterministic abstract machine to obtain a OCaml interpreter running the meta-language Skel. By instantiating it, we automatically generate a certified OCaml interpreter for any language defined in skeletal semantics.

As a summary, the main contributions are the following.

- A novel generic and automatic transformation from big-step to small-step skeletal semantics, implemented in OCaml;
- A paper proof of the correctness of the transformation, in a simplified case;
- An automatic a posteriori Coq proof script checking the equivalence between the input and output semantics;
- A non-deterministic abstract machine semantics for the meta-language Skel, proved equivalent to the base big-step semantics;
- A deterministic abstract machine semantics for Skel, proved sound with respect to the non-deterministic one;
- A generic certified OCaml interpreter for the meta-language Skel, that can be automatically instantiated with any skeletal semantics.

The transformation from big-step to small-step has been presented to the Principles and Practice of Declarative Programming (PPDP) 2022 conference under the title *Certified Derivation of Small-Step From Big-Step Skeletal Semantics* [10]. The abstract machine derivation and the certified interpreter have been presented to the Certified Programs and Proofs (CPP) 2022 conference under the title *Certified Abstract Machines for Skeletal Semantics* [6]. At the start of my PhD position, in the continuation of a previous internship, I also briefly worked on the formalization of HOπ in Coq. This work lead to a journal publication [7], but is not presented in this document.

1.2 Organization of the Document

Chapter 2 presents the different formats of operational semantics (big-step semantics, small-step semantics, reduction semantics, and abstract machines) and some known interderivation methods.

Part I introduces a novel strategy to automatically transform a big-step skeletal semantics into an equivalent small-step skeletal semantics. Chapter 3 presents the legacy skeletal semantics used in this first part. Chapter 4 displays and formalizes the transformation phases. Chapter 5 presents two complementary methods ensuring the correctness of the transformation. Chapter 6 discusses the implementation and the practical results, while Chapter 7 details the limits of the algorithm and its relation to the previous scientific literature.

Part II of this document focuses on the semantics of the higher-order skeletal semantics meta-language, defined in Chapter 8. We describe the known strategy of functional correspondence in Chapter 9 before applying it to the Skel meta-language. Chapter 10 presents the derivation of the non-deterministic abstract machine and the proof of equivalence, while Chapter 11 introduces the deterministic one and the proof of soundness. Finally, Chapter 12 describes how this deterministic abstract machine is used to automatically provide a certified interpreter for any language.

Chapter 2

Operational Semantics

Denotational semantics [49] is an approach where expressions are linked to mathematical objects, which represent their meaning. Programs might for instance be associated with game states, or mathematical functions. This approach is notably used for model-checking, to certify that a system respects a specification. Denotational semantics is defined by induction on the syntax of the language, which may simplify proofs at the cost of a more complex semantics to capture recursive constructs.

Axiomatic semantics is a different approach based on Hoare logic [30]. In it, we consider assertions on program states in first-order logic. The meaning of a computation is then the effect it has on these assertions as the program state is modified.

In this document, we exclusively focus on operational semantics, where a computation is associated to its result. The semantics simply describes how an evaluation is performed. Different flavors of operational semantics allow for different levels of detail.

In this chapter, we present the main four styles of operational semantics: big-step semantics (Section 2.2); small-step semantics (Section 2.3); reduction semantics (Section 2.4); and abstract machine (Section 2.5). As a running example, we use a very simple imperative programming language (IMP) with `while` loops, described in Section 2.1. Finally, Section 2.6 presents different known strategies for transforming a semantics from one style to another.

2.1 Syntax of IMP

We use b to denote booleans, i.e., elements of $\mathbb{B} \triangleq \{\top; \bot\}$. We also use the standard notion of boolean conjunction ($\wedge$) and negation ($\neg$), e.g., $\bot \wedge \top \triangleq \bot$ and $\neg\bot \triangleq \top$. We use n to denote integers in $\mathbb{Z}$, with addition ($+$) and equality test ($\stackrel{?}{=}$), e.g., $(3 \stackrel{?}{=} 4) \triangleq \bot$. We assume an infinite set $\mathcal{V}$ of variables, ranged over by x. We define arithmetic expressions (a), boolean expressions (e), and commands (c) as follows.

$$
\begin{aligned}
a &::= x \mid n \mid \texttt{Plus}(a_1, a_2) \\
e &::= b \mid \texttt{Equal}(a_1, a_2) \mid \texttt{And}(e_1, e_2) \mid \texttt{Neg}(e) \\
c &::= \texttt{skip} \mid \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \mid \texttt{while}\ e\ \texttt{do}\ c \mid c_1\ ;\ c_2 \mid x := a
\end{aligned}
$$

In the syntax above, $c_1\,;\,c_2$ is a sequence of computations; $x := a$ is an assignment operation; and `skip` can be seen as an empty or finished computation. For expressions, we use explicitly named constructors, e.g., $\texttt{Plus}(a_1, a_2)$, to tell them apart from the corresponding mathematical operation, e.g., $n_1 + n_2$.

To evaluate a program, we need a state (σ) mapping variables to integers. We note `AExp` (respectively `BExp`, `CExp`, and `State`) the set of all arithmetic expressions (resp. boolean expressions, commands, and states). States are partial maps with a finite domain. We note ϵ the empty state, and $\{x_1 \mapsto n_1; \ldots; x_m \mapsto n_m\}$ the state mapping each x_i to the corresponding n_i. For x in the domain of σ, $\sigma(x)$ is the associated integer. States σ can be extended with a new association $x_1 \mapsto n_1$. If the variable x is already part of the domain of σ, the extension overwrites the associated integer instead.

$$(\sigma[x \mapsto n])(y) \triangleq \begin{cases} n & \texttt{if } x = y \\ \sigma(y) & \texttt{otherwise} \end{cases}$$

We use the word *term* to refer to either a command or an arithmetic or boolean expression. The different operational semantics presented below evaluate configurations $\langle \sigma, c \rangle$ containing a state and a term.

2.2 Big-Step Semantics

Big-step operational semantics, also known as natural semantics, is the most common and intuitive form of operational semantics. It directly relates a computation to its final result. Most programming languages, such as JavaScript [29] or Standard ML [38], are defined in big-step.

For our IMP example, we create three evaluation predicates. They are defined in Figure 2.1 using inference rules.

- Arithmetic expressions evaluation ($\Downarrow_{\text{a}}$) takes as input a state and an expression and outputs an integer, so that $\Downarrow_{\text{a}} \subseteq (\texttt{State} \times \texttt{AExp}) \times \mathbb{Z}$.
- Boolean expressions evaluate to booleans: $\Downarrow_{\text{b}} \subseteq (\texttt{State} \times \texttt{BExp}) \times \mathbb{B}$.
- Commands return a final evaluation state: $\Downarrow \subseteq (\texttt{State} \times \texttt{CExp}) \times \texttt{State}$.

Integers and booleans evaluate to themselves. Variables are read from the state used for the evaluation. For arithmetic and boolean operations, we evaluate each subexpression before applying the appropriate mathematical operation. A `skip` command does nothing. An assignment leads to an extension of the state. A sequence computes both commands in order, using the output state of the first command to compute the second command. For conditional branchings and loops, we start by evaluating the boolean expression; depending on the resulting boolean, we can have two different behaviors.

$$\frac{}{\langle \sigma, x \rangle \Downarrow_{\mathrm{a}} \sigma(x)} \qquad \frac{}{\langle \sigma, n \rangle \Downarrow_{\mathrm{a}} n} \qquad \frac{\langle \sigma, a_1 \rangle \Downarrow_{\mathrm{a}} n_1 \qquad \langle \sigma, a_2 \rangle \Downarrow_{\mathrm{a}} n_2}{\langle \sigma, \texttt{Plus}(a_1, a_2) \rangle \Downarrow_{\mathrm{a}} n_1 + n_2}$$

$$\frac{}{\langle \sigma, b \rangle \Downarrow_{\mathrm{b}} b} \qquad \frac{\langle \sigma, e \rangle \Downarrow_{\mathrm{b}} b}{\langle \sigma, \texttt{Neg}(e) \rangle \Downarrow_{\mathrm{b}} \neg b}$$

$$\frac{\langle \sigma, a_1 \rangle \Downarrow_{\mathrm{a}} n_1 \qquad \langle \sigma, a_2 \rangle \Downarrow_{\mathrm{a}} n_2}{\langle \sigma, \texttt{Equal}(a_1, a_2) \rangle \Downarrow_{\mathrm{b}} (n_1 \stackrel{?}{=} n_2)} \qquad \frac{\langle \sigma, e_1 \rangle \Downarrow_{\mathrm{b}} b_1 \qquad \langle \sigma, e_2 \rangle \Downarrow_{\mathrm{b}} b_2}{\langle \sigma, \texttt{And}(e_1, e_2) \rangle \Downarrow_{\mathrm{b}} (b_1 \wedge b_2)}$$

$$\frac{}{\langle \sigma, \texttt{skip} \rangle \Downarrow \sigma} \qquad \frac{\langle \sigma, a \rangle \Downarrow_{\mathrm{a}} n}{\langle \sigma, x := a \rangle \Downarrow \sigma[x \mapsto n]} \qquad \frac{\langle \sigma, c_1 \rangle \Downarrow \sigma' \qquad \langle \sigma', c_2 \rangle \Downarrow \sigma''}{\langle \sigma, c_1 \,;\, c_2 \rangle \Downarrow \sigma''}$$

$$\frac{\langle \sigma, e \rangle \Downarrow_{\mathrm{b}} \top \qquad \langle \sigma, c_1 \rangle \Downarrow \sigma'}{\langle \sigma, \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \rangle \Downarrow \sigma'} \qquad \frac{\langle \sigma, e \rangle \Downarrow_{\mathrm{b}} \bot \qquad \langle \sigma, c_2 \rangle \Downarrow \sigma'}{\langle \sigma, \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \rangle \Downarrow \sigma'}$$

$$\frac{\langle \sigma, e \rangle \Downarrow_{\mathrm{b}} \top \qquad \langle \sigma, c \rangle \Downarrow \sigma' \qquad \langle \sigma', \texttt{while}\ e\ \texttt{do}\ c \rangle \Downarrow \sigma''}{\langle \sigma, \texttt{while}\ e\ \texttt{do}\ c \rangle \Downarrow \sigma''} \qquad \frac{\langle \sigma, e \rangle \Downarrow_{\mathrm{b}} \bot}{\langle \sigma, \texttt{while}\ e\ \texttt{do}\ c \rangle \Downarrow \sigma}$$

Figure 2.1: Big-Step Semantics for IMP

Proving that a program evaluates to a certain result is done by providing a derivation tree using the inference rules of the semantics.

Example 2.2.1. Consider the following program.

$$x := 1 \,;\, \texttt{if Equal}(x, 2)\ \texttt{then}\ (x := 3)\ \texttt{else}\ (x := 4)$$

Evaluating it results in a state where x is mapped to 4, as demonstrated with the following tree, where σ_1 is $\{x \mapsto 1\}$.

$$\dfrac{\dfrac{\dfrac{}{\langle \epsilon, 1 \rangle \Downarrow_{\mathrm{a}} 1}}{\langle \epsilon, x := 1 \rangle \Downarrow \sigma_1} \qquad \dfrac{\dfrac{\dfrac{}{\langle \sigma_1, x \rangle \Downarrow_{\mathrm{a}} 1} \qquad \dfrac{}{\langle \sigma_1, 2 \rangle \Downarrow_{\mathrm{a}} 2}}{\langle \sigma_1, \texttt{Equal}(x, 2) \rangle \Downarrow_{\mathrm{b}} \bot} \qquad \dfrac{\dfrac{}{\langle \sigma_1, 4 \rangle \Downarrow_{\mathrm{a}} 4}}{\langle \sigma_1, x := 4 \rangle \Downarrow \{x \mapsto 4\}}}{\langle \sigma_1, \texttt{if Equal}(x, 2)\ \texttt{then}\ (x := 3)\ \texttt{else}\ (x := 4) \rangle \Downarrow \{x \mapsto 4\}}}{\langle \epsilon, x := 1 \,;\, \texttt{if Equal}(x, 2)\ \texttt{then}\ (x := 3)\ \texttt{else}\ (x := 4) \rangle \Downarrow \{x \mapsto 4\}}$$

Unlike the other flavors of operational semantics presented later, a big-step derivation contains the whole evaluation of a program in a single object. This makes proofs by induction on this structure very simple and quite intuitive.

Big-step operational semantics also has important limitations. Notably, a wide variety of

behaviors are not captured by this definition. For instance, we cannot assign a meaning to stuck computations (like $\langle \epsilon, x := \texttt{Plus}(x, 1) \rangle$) or diverging computations (like $\langle \sigma, \texttt{while} \top \texttt{do skip} \rangle$), which means we cannot tell them apart. Some extensions of big-step semantics recover this distinction, usually at the cost of an explicit error constructor added to the language. Big-step is not suitable for concurrent semantics (see next section), as it does not provide a way to interrupt a computation. As such, it is not adapted to low-level programs such as in operating systems.

2.3 Small-Step Semantics

The next format we present is small-step operational semantics, also known as structural operational semantics. Instead of connecting a program to its final value, this semantics is a relation transition from computations to computations. It describes how a program can perform a small part of the computation (a step), and then re-express what is left to compute as a new program. For instance, while a big-step semantics would directly relate $(1 + 2) + (4 + 1) \Downarrow 8$, a small-step semantics allows for a more detailed sequence of reductions $(1 + 2) + (4 + 1) \to 3 + (4 + 1) \to 3 + 5 \to 8$, presenting the intermediate phases of the computation. This relation transition is syntax-oriented, which means that the behavior of a computation is expressed using the behavior if its subcomputations.

For our IMP example, we create three relations, defined in Figure 2.2 using inference rules.

- For arithmetic expressions: $\to_{\text{a}} \subseteq (\texttt{State} \times \texttt{AExp}) \times (\texttt{State} \times \texttt{AExp})$.
- For boolean expressions: $\to_{\text{b}} \subseteq (\texttt{State} \times \texttt{BExp}) \times (\texttt{State} \times \texttt{BExp})$.
- For commands: $\to \subseteq (\texttt{State} \times \texttt{CExp}) \times (\texttt{State} \times \texttt{CExp})$.

It is important to note that, for each relation, the input type is the same as the output type. Evaluating a configuration produces a new configuration. It allows us to create sequences of reductions, as we are mostly interested in the reflexive transitive closure ($\to^*$) of these relations. In the case of arithmetic and booleans expressions, the output state is not modified, so inference rules have premises of the form $\langle \sigma, a \rangle \to_{\text{a}} \langle \sigma, a' \rangle$, reusing σ—we do not need to introduce a different state variable.

For variables, we simply look up the associated value in the given state, as previously. For the `Plus` constructor, we need to write three inference rules. If the first subexpression can do a step, then the whole expression can do the same step. The symmetric situation is when the first subexpression is an integer (and thus cannot reduce further), and the second subexpression can reduce. Finally, an expression $\texttt{Plus}(n_1, n_2)$ where both subexpressions are integers can perform the addition and reduce to $n_1 + n_2$. We have similar cases for the other boolean operators, where we need two to three rules for each.

$$\frac{}{\langle \sigma, x\rangle \to_{\mathrm{a}} \langle \sigma, \sigma(x)\rangle} \qquad \frac{\langle \sigma, a_1\rangle \to_{\mathrm{a}} \langle \sigma, a_1'\rangle}{\langle \sigma, \texttt{Plus}(a_1, a_2)\rangle \to_{\mathrm{a}} \langle \sigma, \texttt{Plus}(a_1', a_2)\rangle}$$

$$\frac{\langle \sigma, a_2\rangle \to_{\mathrm{a}} \langle \sigma, a_2'\rangle}{\langle \sigma, \texttt{Plus}(n_1, a_2)\rangle \to_{\mathrm{a}} \langle \sigma, \texttt{Plus}(n_1, a_2')\rangle} \qquad \frac{}{\langle \sigma, \texttt{Plus}(n_1, n_2)\rangle \to_{\mathrm{a}} \langle \sigma, n_1 + n_2\rangle}$$

$$\frac{\langle \sigma, a_1\rangle \to_{\mathrm{a}} \langle \sigma, a_1'\rangle}{\langle \sigma, \texttt{Equal}(a_1, a_2)\rangle \to_{\mathrm{b}} \langle \sigma, \texttt{Equal}(a_1', a_2)\rangle} \qquad \frac{\langle \sigma, a_2\rangle \to_{\mathrm{a}} \langle \sigma, a_2'\rangle}{\langle \sigma, \texttt{Equal}(n_1, a_2)\rangle \to_{\mathrm{b}} \langle \sigma, \texttt{Equal}(n_1, a_2')\rangle}$$

$$\frac{}{\langle \sigma, \texttt{Equal}(n_1, n_2)\rangle \to_{\mathrm{b}} \langle \sigma, (n_1 \stackrel{?}{=} n_2)\rangle} \qquad \frac{\langle \sigma, e_1\rangle \to_{\mathrm{b}} \langle \sigma, e_1'\rangle}{\langle \sigma, \texttt{And}(e_1, e_2)\rangle \to_{\mathrm{b}} \langle \sigma, \texttt{And}(e_1', e_2)\rangle}$$

$$\frac{\langle \sigma, e_2\rangle \to_{\mathrm{b}} \langle \sigma, e_2'\rangle}{\langle \sigma, \texttt{And}(b_1, e_2)\rangle \to_{\mathrm{b}} \langle \sigma, \texttt{And}(b_1, e_2')\rangle} \qquad \frac{}{\langle \sigma, \texttt{And}(b_1, b_2)\rangle \to_{\mathrm{b}} \langle \sigma, (b_1 \wedge b_2)\rangle}$$

$$\frac{\langle \sigma, e\rangle \to_{\mathrm{b}} \langle \sigma, e'\rangle}{\langle \sigma, \texttt{Neg}(e)\rangle \to_{\mathrm{b}} \langle \sigma, \texttt{Neg}(e')\rangle} \qquad \frac{}{\langle \sigma, \texttt{Neg}(b)\rangle \to_{\mathrm{b}} \langle \sigma, \neg b\rangle}$$

$$\frac{\langle \sigma, a\rangle \to_{\mathrm{a}} \langle \sigma, a'\rangle}{\langle \sigma, x := a\rangle \to \langle \sigma, x := a'\rangle} \qquad \frac{}{\langle \sigma, x := n\rangle \to \langle \sigma[x \mapsto n], \texttt{skip}\rangle} \qquad \frac{\langle \sigma, c_1\rangle \to \langle \sigma', c_1'\rangle}{\langle \sigma, c_1 \,;\, c_2\rangle \to \langle \sigma', c_1' \,;\, c_2\rangle}$$

$$\frac{}{\langle \sigma, \texttt{skip} \,;\, c_2\rangle \to \langle \sigma, c_2\rangle} \qquad \frac{\langle \sigma, e\rangle \to_{\mathrm{b}} \langle \sigma, e'\rangle}{\langle \sigma, \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2\rangle \to \langle \sigma, \texttt{if}\ e'\ \texttt{then}\ c_1\ \texttt{else}\ c_2\rangle}$$

$$\frac{}{\langle \sigma, \texttt{if}\ \top\ \texttt{then}\ c_1\ \texttt{else}\ c_2\rangle \to \langle \sigma, c_1\rangle} \qquad \frac{}{\langle \sigma, \texttt{if}\ \bot\ \texttt{then}\ c_1\ \texttt{else}\ c_2\rangle \to \langle \sigma, c_2\rangle}$$

$$\frac{}{\langle \sigma, \texttt{while}\ e\ \texttt{do}\ c\rangle \to \langle \sigma, \texttt{if}\ e\ \texttt{then}\ (c \,;\, \texttt{while}\ e\ \texttt{do}\ c)\ \texttt{else}\ \texttt{skip}\rangle}$$

Figure 2.2: Small-Step Semantics for IMP

This shows the structure of small-step inference rules, as each rule has either zero or exactly one premise, where the reduction of a subterm takes place. In this document, we call the latter *congruence rules*: they state that if a subterm is able to reduce on its own, we can simply propagate the result to the whole term, keeping the other parts unchanged. We call inference rules without premises *contraction rules* [22]. They correspond to basic reduction steps, usually either the application of a basic operator or a simplification of the term.

For assignment commands, we have two rules: a congruence rule stating we can reduce the arithmetic expression, and a contraction rule to extend the state with an integer. In the second case, we return a `skip` command to represent a finished computation. For sequences, we reduce the first command if able, or drop a leading `skip` otherwise. For conditional branchings, a congruence rule allows us to reduce the boolean expression; when we have a boolean, two contraction rules then let us drop the part of the command we no longer need.

For loops, we bypass an important issue of small-step semantics with a peculiar rule. Normally, we would expect a congruence rule of the following form.

$$\frac{\langle \sigma, e \rangle \to_{\mathrm{b}} \langle \sigma, e' \rangle}{\langle \sigma, \texttt{while}\, e\, \texttt{do}\, c \rangle \to \langle \sigma, \texttt{while}\, e'\, \texttt{do}\, c \rangle}$$

However, such a rule would completely break the semantics. We need to evaluate e to check whether to enter the loop; if we do, i.e., e evaluates to $\top$, we need to loop back to $(\texttt{while}\, e\, \texttt{do}\, c)$ later on. Since the infinite loop $(\texttt{while}\, \top\, \texttt{do}\, c)$ clearly does not have the intended semantics, we cannot permanently overwrite e with e'.

One of the main difficulties of small-step semantics is that the syntax of the language is sometimes not expressive enough to represent all partial computations. In this overview, we use a common trick to sidestep the problem: notice that $\texttt{while}\, e\, \texttt{do}\, c$ has the same behavior as $\texttt{if}\, e\, \texttt{then}\, (c\,;\, \texttt{while}\, e\, \texttt{do}\, c)\, \texttt{else}\, \texttt{skip}$, so we simply add a contraction rule going from the first to the second. This bigger program carries two versions of e and c, and is able to express partial computations.

In the general case, we can systematically extend the language with new constructors in such problematic situations. For IMP, it would require two additional constructors `While1` and `While2`, the first of which remembers both the initial boolean expression e and the computation in progress e': see Figure 2.3. Similarly, if `skip` did not exist, we would need to create it to represent finished computations.

Note that there are no rules for the base elements. We do not want contraction rules of the form $\langle \sigma, b \rangle \to_{\mathrm{b}} \langle \sigma, b \rangle$, $\langle \sigma, n \rangle \to_{\mathrm{a}} \langle \sigma, n \rangle$, or $\langle \sigma, \texttt{skip} \rangle \to \langle \sigma, \texttt{skip} \rangle$, as they would yield infinite reduction sequences. A reduction sequence represents a computation, and we do not want a computation to stay locked onto a base element without progressing further.

Small-step semantics allows for a more precise control of the behavior of programs, and make

$$c ::= \ldots \mid \texttt{while}\ e\ \texttt{do}\ c \mid \texttt{While1}(e_0, e, c) \mid \texttt{While2}(c_0, e, c)$$

$$\frac{}{\langle s, \texttt{while}\ e\ \texttt{do}\ c\rangle \to \langle s, \texttt{While1}(e, e, c)\rangle} \qquad \frac{\langle s, e_0\rangle \to_{\mathrm{b}} \langle s, e_0'\rangle}{\langle s, \texttt{While1}(e_0, e, c)\rangle \to \langle s, \texttt{While1}(e_0', e, c)\rangle}$$

$$\frac{}{\langle s, \texttt{While1}(\bot, e, c)\rangle \to \langle s, \texttt{skip}\rangle} \qquad \frac{}{\langle s, \texttt{While1}(\top, e, c)\rangle \to \langle s, \texttt{While2}(c, e, c)\rangle}$$

$$\frac{\langle s, c_0\rangle \to \langle s', c_0'\rangle}{\langle s, \texttt{While2}(c_0, e, c)\rangle \to \langle s', \texttt{While2}(c_0', e, c)\rangle} \qquad \frac{}{\langle s, \texttt{While2}(s, \texttt{skip}, e, c)\rangle \to \langle s, \texttt{While}(e, c)\rangle}$$

Figure 2.3: Alternative Small-Step Syntax and Semantics for While Loop

up for the limitations of big-step semantics. For example, we can tell apart stuck computations that do not reduce, e.g., $\langle \epsilon, x := \texttt{Plus}(x, 1)\rangle$, from diverging computations, e.g.:

$$\begin{aligned}\langle \sigma, \texttt{while}\ \top\ \texttt{do skip}\rangle &\to \langle \sigma, \texttt{if}\ \top\ \texttt{then}\ (\texttt{skip}\ ;\ \texttt{while}\ \top\ \texttt{do skip})\ \texttt{else skip}\rangle\\ &\to \langle \sigma, \texttt{skip}\ ;\ \texttt{while}\ \top\ \texttt{do skip}\rangle\\ &\to \langle \sigma, \texttt{while}\ \top\ \texttt{do skip}\rangle\\ &\to \langle \sigma, \texttt{if}\ \top\ \texttt{then}\ (\texttt{skip}\ ;\ \texttt{while}\ \top\ \texttt{do skip})\ \texttt{else skip}\rangle\\ &\to \ldots\end{aligned}$$

Small-step semantics can be used to reason about infinite derivations, using the concepts of traces and prefix of execution. Small-step semantics can also easily capture notions of concurrency and interleaving.

Example 2.3.1. For instance, IMP can be extended with a parallel operator as follows.

$$c ::= \texttt{skip} \mid \ldots \mid c_1 \parallel c_2$$

Then, we simply need to extend the semantics with two congruence rules and a contraction rule.

$$\frac{\langle \sigma, c_1\rangle \to \langle \sigma', c_1'\rangle}{\langle \sigma, c_1 \parallel c_2\rangle \to \langle \sigma', c_1' \parallel c_2\rangle} \qquad \frac{\langle \sigma, c_2\rangle \to \langle \sigma', c_2'\rangle}{\langle \sigma, c_1 \parallel c_2\rangle \to \langle \sigma', c_1 \parallel c_2'\rangle} \qquad \frac{}{\langle \sigma, \texttt{skip} \parallel \texttt{skip}\rangle \to \langle \sigma, \texttt{skip}\rangle}$$

I.e., when we have two programs in parallel, if either is able to reduce on its own, then the whole computation can progress. This cannot be stated using big-step semantics, where inference rules can only fully evaluate subterms. Alternating between two computations require a notion of partial or interrupted computations.

Example 2.3.2. We show how to reduce the example program of Example 2.2.1. It requires 6

successive small steps to evaluate it fully. Once again, σ_1 represents the state $\{x \mapsto 1\}$.

$$1.\ \frac{\overline{\langle \epsilon, x := 1\rangle \to \langle \sigma_1, \texttt{skip}\rangle}}{\langle \epsilon, x := 1\ ;\ \texttt{if Equal}(x, 2)\ \texttt{then}\ (x := 3)\ \texttt{else}\ (x := 4)\rangle \to \langle \sigma_1, \texttt{skip}\ ;\ \texttt{if}\ \ldots\ \texttt{then}\ \ldots\ \texttt{else}\ \ldots\rangle}$$

$$2.\ \frac{}{\langle \sigma_1, \texttt{skip}\ ;\ \texttt{if}\ \ldots\ \texttt{then}\ \ldots\ \texttt{else}\ \ldots\rangle \to \langle \sigma_1, \texttt{if Equal}(x, 2)\ \texttt{then}\ (x := 3)\ \texttt{else}\ (x := 4)\rangle}$$

$$3.\ \frac{\dfrac{\overline{\langle \sigma_1, x\rangle \to \langle \sigma_1, 1\rangle}}{\langle \sigma_1, \texttt{Equal}(x, 2)\rangle \to \langle \sigma_1, \texttt{Equal}(1, 2)\rangle}}{\langle \sigma_1, \texttt{if Equal}(x, 2)\ \texttt{then}\ (x := 3)\ \texttt{else}\ (x := 4)\rangle \to \langle \sigma_1, \texttt{if Equal}(1, 2)\ \texttt{then}\ (x := 3)\ \texttt{else}\ (x := 4)\rangle}$$

$$4.\ \frac{\overline{\langle \sigma_1, \texttt{Equal}(1, 2)\rangle \to \langle \sigma_1, \bot\rangle}}{\langle \sigma_1, \texttt{if Equal}(1, 2)\ \texttt{then}\ (x := 3)\ \texttt{else}\ (x := 4)\rangle \to \langle \sigma_1, \texttt{if}\ \bot\ \texttt{then}\ (x := 3)\ \texttt{else}\ (x := 4)\rangle}$$

$$5.\ \frac{}{\langle \sigma_1, \texttt{if}\ \bot\ \texttt{then}\ (x := 3)\ \texttt{else}\ (x := 4)\rangle \to \langle \sigma_1, x := 4\rangle}$$

$$6.\ \frac{}{\langle \sigma_1, x := 4\rangle \to \langle \{x \mapsto 4\}, \texttt{skip}\rangle}$$

By joining the different small steps, we construct the following sequence of reductions.

$$\langle \epsilon, x := 1\ ;\ \texttt{if Equal}(x, 2)\ \texttt{then}\ (x := 3)\ \texttt{else}\ (x := 4)\rangle \to^* \langle \{x \mapsto 4\}, \texttt{skip}\rangle$$

In general, big-step and small-step semantics define the same behavior. A big-step evaluation should correspond to a sequence of small-step reductions. This is formalized in the following theorem.

Theorem 2.3.3. For all programs c and states σ and σ',

$$\langle \sigma, c\rangle \Downarrow \sigma' \iff \langle \sigma, c\rangle \to^* \langle \sigma', \texttt{skip}\rangle$$

Proof. The proof is quite standard, and can be done by induction on the big-step derivation and on the length of the small-step reduction sequence. See for instance [41]. □

The additional control offered by the small-step semantics rules can be useful for certifying some properties. Notably, it is a common representation for proving type safety of a language, by checking the properties of progress and preservation. However, small-step semantics is more verbose and less intuitive than big-step, as manipulating a sequence of reduction is harder than

a self-contained derivation tree. It also often requires an extended syntax, as would be the case for `while` loops without the above sidestep.

2.4 Reduction Semantics

Reduction semantics is a rephrasing of small-step semantics where we separate the basic reduction rules (i.e., contraction rules) from the location in the term where they can be applied. For IMP, we have the following contraction rules ($\rightsquigarrow$).

$$\begin{aligned}
\langle \sigma, x \rangle &\rightsquigarrow \langle \sigma, \sigma(x) \rangle \\
\langle \sigma, \texttt{Plus}(n_1, n_2) \rangle &\rightsquigarrow \langle \sigma, n_1 + n_2 \rangle \\
\langle \sigma, \texttt{Equal}(n_1, n_2) \rangle &\rightsquigarrow \langle \sigma, n_1 \stackrel{?}{=} n_2 \rangle \\
\langle \sigma, \texttt{And}(b_1, b_2) \rangle &\rightsquigarrow \langle \sigma, b_1 \wedge b_2 \rangle \\
\langle \sigma, \texttt{Neg}(b) \rangle &\rightsquigarrow \langle \sigma, \neg b \rangle \\
\langle \sigma, x := n \rangle &\rightsquigarrow \langle \sigma[x \mapsto n], \texttt{skip} \rangle \\
\langle \sigma, \texttt{skip} \ ; \ c_2 \rangle &\rightsquigarrow \langle \sigma, c_2 \rangle \\
\langle \sigma, \texttt{if} \ \top \ \texttt{then} \ c_1 \ \texttt{else} \ c_2 \rangle &\rightsquigarrow \langle \sigma, c_1 \rangle \\
\langle \sigma, \texttt{if} \ \bot \ \texttt{then} \ c_1 \ \texttt{else} \ c_2 \rangle &\rightsquigarrow \langle \sigma, c_2 \rangle \\
\langle \sigma, \texttt{while} \ e \ \texttt{do} \ c \rangle &\rightsquigarrow \langle \sigma, \texttt{if} \ e \ \texttt{then} \ (c \ ; \ \texttt{while} \ e \ \texttt{do} \ c) \ \texttt{else} \ \texttt{skip} \rangle
\end{aligned}$$

As seen in Figure 2.2, a large proportion of small-step inference rules are simply congruence rules, with very similar structure. In reduction semantics, we replace congruence rules with evaluation contexts, specifying where subcomputations are allowed. In the case of IMP, we have the following contexts.

$$\begin{aligned}
E ::= [\cdot] \mid \texttt{Plus}(E, a) \mid \texttt{Plus}(n, E) \mid \texttt{Equal}(E, a) \mid \texttt{Equal}(n, E) \mid \texttt{And}(E, e) \\
\mid \texttt{And}(b, E) \mid \texttt{Neg}(E) \mid \texttt{if} \ E \ \texttt{then} \ c_1 \ \texttt{else} \ c_2 \mid E \ ; \ c_2 \mid x := E
\end{aligned}$$

Contexts are terms with a single hole ($[\cdot]$) that can be filled with another term. For instance, if $E \triangleq \texttt{Equal}(n, \texttt{Plus}([\cdot], a))$, then $E[a'] \triangleq \texttt{Equal}(n, \texttt{Plus}(a', a))$. A more formal definition of reduction semantics would distinguish different kinds of contexts depending on the type (`AExp`/`BExp`/`CExp`) of the hole and the type of the produced term. In this overview, we simplify the notations by regrouping them in a single definition.

Then, we can apply the contraction rules ($\rightsquigarrow$) above at any depth in our language programs

with the following congruence rule[1], making use of evaluation contexts. For all terms t:

$$\frac{\langle \sigma, t\rangle \rightsquigarrow \langle \sigma', t'\rangle}{\langle \sigma, E[t]\rangle \to_{\mathrm{RS}} \langle \sigma', E[t']\rangle}$$

Evaluation contexts are defined such that each case corresponds exactly to a congruence rule of the small-step semantics (Figure 2.2). This approach leads to objects (evaluation contexts) representing where computations can happen, separated from the reduction rules of the different base cases.

Example 2.4.1. Evaluating the example program presented before (Example 2.3.2), we similarly have $\langle \epsilon, x := 1\ ;\ \texttt{if}\ \texttt{Equal}(x, 2)\ \texttt{then}\ (x := 3)\ \texttt{else}\ (x := 4)\rangle \to^*_{\mathrm{RS}} \langle \{x \mapsto 4\}, \texttt{skip}\rangle$ with the same 6 steps.

1. The first step reduces under the context $[\cdot]\ ;\ \texttt{if}\ \texttt{Equal}(x, 2)\ \texttt{then}\ (x := 3)\ \texttt{else}\ (x := 4)$ and uses the contraction rule $\langle \epsilon, x := 1\rangle \rightsquigarrow \langle \{x \mapsto 1\}, \texttt{skip}\rangle$.
2. The second step directly uses the contraction rule $\langle \{x \mapsto 1\}, \texttt{skip}; \texttt{if} \ldots \texttt{then} \ldots \texttt{else} \ldots\rangle \rightsquigarrow \langle \{x \mapsto 1\}, \texttt{if}\ \ldots\ \texttt{then}\ \ldots\ \texttt{else}\ \ldots\rangle$ (i.e., with the empty context $[\cdot]$).
3. The third step reduces under the context $\texttt{if}\ \texttt{Equal}([\cdot], 2)\ \texttt{then}\ (x := 3)\ \texttt{else}\ (x := 4)$ and uses the contraction rule $\langle \{x \mapsto 1\}, x\rangle \rightsquigarrow \langle \{x \mapsto 1\}, 1\rangle$.
4. The fourth step reduces under the context $\texttt{if}\ [\cdot]\ \texttt{then}\ (x := 3)\ \texttt{else}\ (x := 4)$ and uses the contraction rule $\langle \{x \mapsto 1\}, \texttt{Equal}(1, 2)\rangle \rightsquigarrow \langle \{x \mapsto 1\}, \bot\rangle$.
5. The fifth step directly uses the contraction rule $\langle \{x \mapsto 1\}, \texttt{if}\ \bot\ \texttt{then}\ (x := 3)\ \texttt{else}\ (x := 4)\rangle \rightsquigarrow \langle \{x \mapsto 1\}, x := 4\rangle$.
6. Finally, the last step directly uses the rule $\langle \{x \mapsto 1\}, x := 4\rangle \rightsquigarrow \langle \{x \mapsto 4\}, \texttt{skip}\rangle$.

As expected, we can easily show that the small-step semantics and reduction semantics define the exact same behavior.

Theorem 2.4.2. For all programs c and c' and states σ and σ',

$$\langle \sigma, c\rangle \to \langle \sigma', c'\rangle \iff \langle \sigma, c\rangle \to_{\mathrm{RS}} \langle \sigma', c'\rangle$$

Proof. By simply specifying the one-to-one correspondence between context constructors and congruence rules. □

Reduction semantics offers several quality-of-life improvements over small-step semantics. It separates the computational part, i.e., reducing a subterm, from the navigation necessary

1. If we take the more formal approach, we need one congruence rule per kind of context.

to reach the subterm. This eliminates the need for a tree structure, and produces a cleaner representation. Notably, reconstruction is simply defined as plugging the result into a context, instead of painfully traversing the derivation tree in reverse. Having access to the evaluation context as an independent object also allows for more refined manipulations. As such, reduction semantics proves to be a useful format for languages with control operators, such as the `callcc` operator in Scheme.

However, reduction semantics is still mostly a rephrasing of small-step semantics, and does not provide, in practice, a clear method for decomposing a term into an evaluation context and a reducible term.

2.5 Abstract Machine

The last format we present is abstract machines. It is similar to reduction semantics, in the sense that we have a reification of evaluation contexts, but it also goes further by providing an explicit strategy for manipulating them. An abstract machine is a reduction system between objects, called *machine states*, which represent the internal state of a virtual computer executing a program of the language. Machines states are usually configurations (e.g., $\langle \sigma, c \rangle$) extended with the additional information needed by the virtual computer (memory, evaluation modes, flags, ...).

In the case of IMP, the extra information we need are *continuations* (κ), objects representing what is left to compute after we finish executing the term under focus. For IMP, they are lists of elementary evaluation contexts and represent what is around the term we are evaluating.

$$\kappa \ ::= \ \bullet \ \mid \ \texttt{Plus}([\cdot], a) :: \kappa \ \mid \ \texttt{Plus}(n, [\cdot]) :: \kappa \ \mid \ \ldots \ \mid \ x := [\cdot] :: \kappa$$

The ellipsis hides the same cases as in the definition of evaluation contexts in reduction semantics, and $\bullet$ is the empty continuation signaling there is no more computation to perform.

Machine states are then separated into two different *evaluation modes*, depending on the kind of computations we are performing[2]. $\langle \sigma, \ c \ \mid \ \kappa \rangle_{\mathrm{e}}$ is a state in computation mode, and can be seen as a computer currently evaluating $\langle \sigma, c \rangle$ with pending computations κ. Meanwhile, $\langle \kappa \ \mid \ \sigma, \ c \rangle_{\mathrm{k}}$ is in continuation mode, and corresponds to applying the continuation κ to a result c (here, c is either an integer, a boolean, or `skip`). We swap the order of arguments to visually contrast the two evaluation modes.

We cannot directly use evaluation contexts as continuations, as the abstract machine would need to modify the inside of a context when changing focus (see below). Instead, continuations can be seen as evaluation contexts in reverse. For instance, the context $\texttt{Equal}(\texttt{Plus}(n, [\cdot]), a)$

2. To be more formal, we would need 6 modes: 3 computation modes and 3 continuation modes, depending on the type of the term.

in Section 2.4 corresponds to the continuation $\texttt{Plus}(n, [\cdot]) :: \texttt{Equal}([\cdot], a) :: \bullet$ in an abstract machine state.

The steps of the abstract machine for IMP are given in Figure 2.4. For the sake of readability, we reuse the notation ($\rightarrow$), previously used for small-step semantics, for the reduction of abstract machines. There should be no confusion, as the objects being reduced are different.

For machine states in computation mode (e.g., $\langle \sigma,\ \texttt{Plus}(a_1, a_2) \mid \kappa \rangle_{\mathrm{e}}$), we do not need to look up the current continuation. If we need to evaluate a subterm, we focus on it and push the rest of the term on the continuation (e.g., $\langle \sigma,\ a_1 \mid \texttt{Plus}([\cdot], a_2) :: \kappa \rangle_{\mathrm{e}}$). If we are focused on a value (e.g., `skip`) we switch to continuation mode. In some cases (variable and while loop), we can also directly perform a contraction rule.

For machine states in continuation mode, we need to look at the beginning of the continuation, which corresponds to the area immediately around the result under focus. If we have a first elementary context, we apply the appropriate action, usually either focusing on another subterm or applying a contraction rule (e.g. $\langle \texttt{Neg}([\cdot]) :: \kappa \mid \sigma,\ b \rangle_{\mathrm{k}} \rightarrow \langle \kappa \mid \sigma,\ \neg b \rangle_{\mathrm{k}}$). There is no rule for the empty continuation, as in that case, we are done computing and the abstract machine stops.

In practice, to evaluate a configuration $\langle \sigma, c \rangle$, we start the machine with the empty continuation $\langle \sigma,\ c \mid \bullet \rangle_{\mathrm{e}}$ and repeatedly reduce. The machine stops when no rule is applicable, in a machine state of the form $\langle \bullet \mid \sigma',\ \texttt{skip} \rangle_{\mathrm{k}}$.

$$
\begin{aligned}
\langle \sigma,\ n \ \mid\ \kappa \rangle_{\mathrm{e}} &\to \langle \kappa \ \mid\ \sigma,\ n \rangle_{\mathrm{k}} \\
\langle \sigma,\ b \ \mid\ \kappa \rangle_{\mathrm{e}} &\to \langle \kappa \ \mid\ \sigma,\ b \rangle_{\mathrm{k}} \\
\langle \sigma,\ \texttt{skip} \ \mid\ \kappa \rangle_{\mathrm{e}} &\to \langle \kappa \ \mid\ \sigma,\ \texttt{skip} \rangle_{\mathrm{k}} \\
\langle \sigma,\ x \ \mid\ \kappa \rangle_{\mathrm{e}} &\to \langle \kappa \ \mid\ \sigma,\ \sigma(x) \rangle_{\mathrm{k}} \\
\langle \sigma,\ \texttt{Plus}(a_1, a_2) \ \mid\ \kappa \rangle_{\mathrm{e}} &\to \langle \sigma,\ a_1 \ \mid\ \texttt{Plus}([\cdot], a_2) :: \kappa \rangle_{\mathrm{e}} \\
\langle \sigma,\ \texttt{Equal}(a_1, a_2) \ \mid\ \kappa \rangle_{\mathrm{e}} &\to \langle \sigma,\ a_1 \ \mid\ \texttt{Equal}([\cdot], a_2) :: \kappa \rangle_{\mathrm{e}} \\
\langle \sigma,\ \texttt{And}(e_1, e_2) \ \mid\ \kappa \rangle_{\mathrm{e}} &\to \langle \sigma,\ e_1 \ \mid\ \texttt{And}([\cdot], e_2) :: \kappa \rangle_{\mathrm{e}} \\
\langle \sigma,\ \texttt{Neg}(e) \ \mid\ \kappa \rangle_{\mathrm{e}} &\to \langle \sigma,\ e \ \mid\ \texttt{Neg}([\cdot]) :: \kappa \rangle_{\mathrm{e}} \\
\langle \sigma,\ x := a \ \mid\ \kappa \rangle_{\mathrm{e}} &\to \langle \sigma,\ a \ \mid\ x := [\cdot] :: \kappa \rangle_{\mathrm{e}} \\
\langle \sigma,\ c_1\ ;\ c_2 \ \mid\ \kappa \rangle_{\mathrm{e}} &\to \langle \sigma,\ c_1 \ \mid\ [\cdot]\ ;\ c_2 :: \kappa \rangle_{\mathrm{e}} \\
\langle \sigma,\ \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \ \mid\ \kappa \rangle_{\mathrm{e}} &\to \langle \sigma,\ e \ \mid\ \texttt{if}\ [\cdot]\ \texttt{then}\ c_1\ \texttt{else}\ c_2 :: \kappa \rangle_{\mathrm{e}} \\
\langle \sigma,\ \texttt{while}\ e\ \texttt{do}\ c \ \mid\ \kappa \rangle_{\mathrm{e}} &\to \langle \sigma,\ \texttt{if}\ e\ \texttt{then}\ (c\ ;\ \texttt{while}\ e\ \texttt{do}\ c)\ \texttt{else}\ \texttt{skip} \ \mid\ \kappa \rangle_{\mathrm{e}}
\end{aligned}
$$

$$
\begin{aligned}
\langle \texttt{Plus}([\cdot], a_2) :: \kappa \ \mid\ \sigma,\ n_1 \rangle_{\mathrm{k}} &\to \langle \sigma,\ a_2 \ \mid\ \texttt{Plus}(n_1, [\cdot]) :: \kappa \rangle_{\mathrm{e}} \\
\langle \texttt{Plus}(n_1, [\cdot]) :: \kappa \ \mid\ \sigma,\ n_2 \rangle_{\mathrm{k}} &\to \langle \kappa \ \mid\ \sigma,\ n_1 + n_2 \rangle_{\mathrm{k}} \\
\langle \texttt{Equal}([\cdot], a_2) :: \kappa \ \mid\ \sigma,\ n_1 \rangle_{\mathrm{k}} &\to \langle \sigma,\ a_2 \ \mid\ \texttt{Equal}(n_1, [\cdot]) :: \kappa \rangle_{\mathrm{e}} \\
\langle \texttt{Equal}(n_1, [\cdot]) :: \kappa \ \mid\ \sigma,\ n_2 \rangle_{\mathrm{k}} &\to \langle \kappa \ \mid\ \sigma,\ n_1 \stackrel{?}{=} n_2 \rangle_{\mathrm{k}} \\
\langle \texttt{And}([\cdot], e_2) :: \kappa \ \mid\ \sigma,\ b_1 \rangle_{\mathrm{k}} &\to \langle \sigma,\ e_2 \ \mid\ \texttt{And}(b_1, [\cdot]) :: \kappa \rangle_{\mathrm{e}} \\
\langle \texttt{And}(b_1, [\cdot]) :: \kappa \ \mid\ \sigma,\ b_2 \rangle_{\mathrm{k}} &\to \langle \kappa \ \mid\ \sigma,\ b_1 \wedge b_2 \rangle_{\mathrm{k}} \\
\langle \texttt{Neg}([\cdot]) :: \kappa \ \mid\ \sigma,\ b \rangle_{\mathrm{k}} &\to \langle \kappa \ \mid\ \sigma,\ \neg b \rangle_{\mathrm{k}} \\
\langle x := [\cdot] :: \kappa \ \mid\ \sigma,\ n \rangle_{\mathrm{k}} &\to \langle \kappa \ \mid\ \sigma[x \mapsto n],\ \texttt{skip} \rangle_{\mathrm{k}} \\
\langle [\cdot]\ ;\ c_2 :: \kappa \ \mid\ \sigma,\ \texttt{skip} \rangle_{\mathrm{k}} &\to \langle \sigma,\ c_2 \ \mid\ \kappa \rangle_{\mathrm{e}} \\
\langle \texttt{if}\ [\cdot]\ \texttt{then}\ c_1\ \texttt{else}\ c_2 :: \kappa \ \mid\ \sigma,\ \top \rangle_{\mathrm{k}} &\to \langle \sigma,\ c_1 \ \mid\ \kappa \rangle_{\mathrm{e}} \\
\langle \texttt{if}\ [\cdot]\ \texttt{then}\ c_1\ \texttt{else}\ c_2 :: \kappa \ \mid\ \sigma,\ \bot \rangle_{\mathrm{k}} &\to \langle \sigma,\ c_2 \ \mid\ \kappa \rangle_{\mathrm{e}}
\end{aligned}
$$

Figure 2.4: Abstract Machine for IMP

Example 2.5.1. We describe the evaluation of the small program used in previous examples.

$$
\begin{aligned}
&\langle \epsilon,\ x := 1\ ;\ \texttt{if}\ \texttt{Equal}(x, 2)\ \texttt{then}\ (x := 3)\ \texttt{else}\ (x := 4)\ \mid\ \bullet \rangle_{\mathrm{e}} \\
\to\ &\langle \epsilon,\ x := 1\ \mid\ [\cdot]\ ;\ \texttt{if}\ \texttt{Equal}(x, 2)\ \texttt{then}\ (x := 3)\ \texttt{else}\ (x := 4)\ ::\ \bullet \rangle_{\mathrm{e}} \\
\to\ &\langle \epsilon,\ 1\ \mid\ x := [\cdot]\ ::\ [\cdot]\ ;\ \texttt{if}\ \texttt{Equal}(x, 2)\ \texttt{then}\ (x := 3)\ \texttt{else}\ (x := 4)\ ::\ \bullet \rangle_{\mathrm{e}} \\
\to\ &\langle x := [\cdot]\ ::\ [\cdot]\ ;\ \texttt{if}\ \texttt{Equal}(x, 2)\ \texttt{then}\ (x := 3)\ \texttt{else}\ (x := 4)\ ::\ \bullet\ \mid\ \epsilon,\ 1 \rangle_{\mathrm{k}} \\
\to\ &\langle [\cdot]\ ;\ \texttt{if}\ \texttt{Equal}(x, 2)\ \texttt{then}\ (x := 3)\ \texttt{else}\ (x := 4)\ ::\ \bullet\ \mid\ \{x \mapsto 1\},\ \texttt{skip} \rangle_{\mathrm{k}} \\
\to\ &\langle \{x \mapsto 1\},\ \texttt{if}\ \texttt{Equal}(x, 2)\ \texttt{then}\ (x := 3)\ \texttt{else}\ (x := 4)\ \mid\ \bullet \rangle_{\mathrm{e}} \\
\to\ &\langle \{x \mapsto 1\},\ \texttt{Equal}(x, 2)\ \mid\ \texttt{if}\ [\cdot]\ \texttt{then}\ (x := 3)\ \texttt{else}\ (x := 4)\ ::\ \bullet \rangle_{\mathrm{e}} \\
\to\ &\langle \{x \mapsto 1\},\ x\ \mid\ \texttt{Equal}([\cdot], 2)\ ::\ \texttt{if}\ [\cdot]\ \texttt{then}\ (x := 3)\ \texttt{else}\ (x := 4)\ ::\ \bullet \rangle_{\mathrm{e}} \\
\to\ &\langle \texttt{Equal}([\cdot], 2)\ ::\ \texttt{if}\ [\cdot]\ \texttt{then}\ (x := 3)\ \texttt{else}\ (x := 4)\ ::\ \bullet\ \mid\ \{x \mapsto 1\},\ 1 \rangle_{\mathrm{k}} \\
\to\ &\langle \{x \mapsto 1\},\ 2\ \mid\ \texttt{Equal}(1, [\cdot])\ ::\ \texttt{if}\ [\cdot]\ \texttt{then}\ (x := 3)\ \texttt{else}\ (x := 4)\ ::\ \bullet \rangle_{\mathrm{e}} \\
\to\ &\langle \texttt{Equal}(1, [\cdot])\ ::\ \texttt{if}\ [\cdot]\ \texttt{then}\ (x := 3)\ \texttt{else}\ (x := 4)\ ::\ \bullet\ \mid\ \{x \mapsto 1\},\ 2 \rangle_{\mathrm{k}} \\
\to\ &\langle \texttt{if}\ [\cdot]\ \texttt{then}\ (x := 3)\ \texttt{else}\ (x := 4)\ ::\ \bullet\ \mid\ \{x \mapsto 1\},\ \bot \rangle_{\mathrm{k}} \\
\to\ &\langle \{x \mapsto 1\},\ x := 4\ \mid\ \bullet \rangle_{\mathrm{e}} \\
\to\ &\langle \{x \mapsto 1\},\ 4\ \mid\ x := [\cdot]\ ::\ \bullet \rangle_{\mathrm{e}} \\
\to\ &\langle x := [\cdot]\ ::\ \bullet\ \mid\ \{x \mapsto 1\},\ 4 \rangle_{\mathrm{k}} \\
\to\ &\langle \bullet\ \mid\ \{x \mapsto 4\},\ \texttt{skip} \rangle_{\mathrm{k}}
\end{aligned}
$$

This abstract machine produces the same behavior as the previous forms of operational semantics.

Theorem 2.5.2. For all programs c and states σ and σ',

$$\langle \sigma,\ c\ \mid\ \bullet \rangle_{\mathrm{e}} \to^* \langle \bullet\ \mid\ \sigma',\ \texttt{skip} \rangle_{\mathrm{k}} \iff \langle \sigma, c \rangle \to^*_{\mathrm{RS}} \langle \sigma', \texttt{skip} \rangle$$

Proof Sketch. The proof uses an even stronger relation between reduction semantics and abstract machine. We show that machine states $\langle \sigma,\ c\ \mid\ \kappa \rangle_{\mathrm{e}}$ and $\langle \kappa\ \mid\ \sigma,\ c \rangle_{\mathrm{k}}$ are both equivalent to a configuration $\langle \sigma, E_\kappa[c] \rangle$, where E_κ is the context corresponding to the continuation κ.

Whenever the abstract machine takes a step, either it adjusts the term under focus by changing the continuation, as with the rule $\langle \sigma, \texttt{Plus}(a_1, a_2)\ \mid\ \kappa \rangle_{\mathrm{e}} \to \langle \sigma, a_1\ \mid\ \texttt{Plus}([\cdot], a_2)\ ::\ \kappa \rangle_{\mathrm{e}}$, and nothing happens in reduction semantics; or we apply a rule equivalent to a contraction rule in reduction semantics, e.g., $\langle x := [\cdot]\ ::\ \kappa\ \mid\ \sigma,\ n \rangle_{\mathrm{k}} \to \langle \kappa\ \mid\ \sigma[x \mapsto n],\ \texttt{skip} \rangle_{\mathrm{k}}$ is equivalent to using the contraction rule $\langle \sigma, x := n \rangle \rightsquigarrow \langle \sigma[x \mapsto n], \texttt{skip} \rangle$ under the context E_κ.

Once this is formalized, the theorem is only a special case, proved by a simple induction on the length of the sequence. □

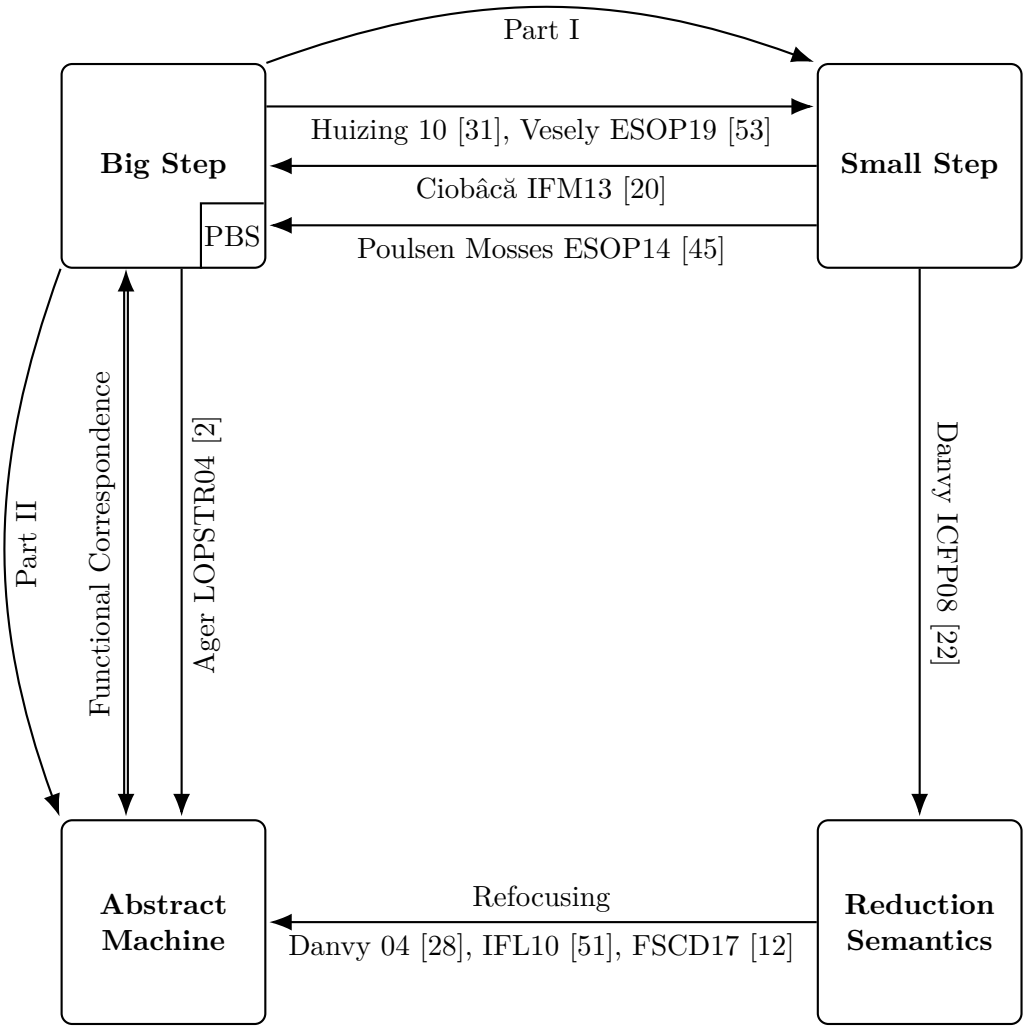

Figure 2.5: Related Work on Interderivation of Operational Semantics

The main advantage of abstract machines is that every decision, every change of focus, is made entirely explicit. After a computation step, we do not fully rebuilt a term, but navigate to the next subterm to be evaluated. This produces a semantics very close to an actual implementation, and it can easily be simulated and executed in practice. This fine-grain description of steps of executions is also sometimes used as reference for defining complexity classes. Notably, abstract machines can be used for modeling and measuring the complexity of algorithms [1].

2.6 Interderivation

Each of the operational semantics flavor presented above has its own strengths and drawbacks. Depending on the use case, it is sometimes necessary to manipulate several versions of a seman-

tics. To limit human errors and efforts when duplicating definitions, systematic interderivation strategies have been developed. Figure 2.5 displays some of them. However, the correctness of most strategies is not properly certified; they should be seen as time-tested guidelines on how to transform a semantics.

The most studied relation is the connection between big-step and abstract machines. The most widespread method is functional correspondence [5], a systematic approach for converting big-step semantics, written as interpreters[3], to abstract machines and back. It combines several known techniques, such as CPS translation [44] and defunctionalization [47], to progressively restructure the semantics of a language. It has been applied to languages with various features [13, 3, 43, 14, 11, 4, 25, 32], and has recently been implemented into a tool for the automatic translations of Racket interpreters [18]. The reverse direction is also possible. An abstract machine can be transformed into a direct-style big-step semantics by following the sequence of reverse transformations [5]. This requires the abstract machine to be fully defunctionalized (i.e., the continuations are only dispatch by a single mechanism), but a few ad-hoc techniques can often be used to tweak the abstract machine to meet this requirement [27, 15]. We use functional correspondence to transform a semantics in Part II of this document, and we describe this strategy in more details in Chapter 9. Ager [2] also proposes a translation from big-step to abstract machine by directly manipulating inference rules, creating continuations keeping track of the big-step premise being evaluated.

From small-step to big-step semantics, the main issue is that the small-step format is more expressive, and some behaviors cannot be defined in big-step, such as interleaving (Example 2.3.1). The different known approaches are partial and assume some structural properties to be applied. Notable transformations include the works of Ciobâcă [20] and Poulsen and Mosses [45], which manipulate small-step inference rules and assemble them. In the second paper, the output is in a format called pretty-big-step (PBS) semantics [19]. It is a subset of big-step where rules can only evaluate a single subterm, and then might need to perform a tail call for further computations. Terms are still related to their final result, but tree derivations are more vertical and there is no need to duplicate rules when extending the language to account for exceptions and divergence.

Transformation from big-step to small-step semantics has been less investigated. Huizing et al. [31] describe a very generic certified translation manipulating big-step inference rules, but the output small-step semantics is unusual and not intuitive: configurations are extended with a stack, and we guess starting states each time we focus on a new big-step premise. More recently, Vesely and Fisher [53] present a new approach for translating big-step interpreters. They combine numerous small known transformations—including CPS and direct-style translations and defunctionalization—into a complete pipeline for deriving a small-step semantics. In Part I

3. I.e., a semantics not written using inference rules, but in a form similar to a program, either in pseudo-code or in a meta-language—usually an extended λ-calculus or an ML-like language.

of this document, we present a novel systematic big-step to small-step translation, for languages written in a specific framework called skeletal semantics. We compare our translation to previous techniques in Chapter 7.

Small-step and reduction semantics are quite similar, since the difference is only the representation of evaluation contexts. These contexts can be made explicit by performing a CPS transformation and giving names to the continuations [22].

From there, the strategy for constructing a proper abstract machine is called refocusing [28]. In reduction semantics, after every step we plug the result in the evaluation context, forgetting about the location where the contraction rule was applied. Using this strategy for an abstract machine would be extremely inefficient. Refocusing corresponds to searching for a new evaluation context from the focus point of the last step. This removes the need to fully unzoom and rezoom at every application of a contraction rule. This refocusing method has been formalized, generalized, and certified [51, 12] in the Coq proof assistant[52]. The authors proved, independently from the target language, that the resulting abstract machine perfectly simulates the initial reduction semantics. To the best of our knowledge, this interderivation is the only related work of Figure 2.5 to be fully certified.

Part I

Object Language Transformation

Chapter 3

Skeletal Semantics

Skeletal semantics is a framework to formalize the operational semantics of programming languages. Users can describe the semantics they are interested in using using a small meta-language named Skel. The fundamental idea of the approach 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.

We present the formal definition of skeletal semantics we use to describe the transformation in Section 4.2 and state the equivalence results in Chapter 5. The framework has evolved since its first definition [17], and Part I uses an updated description [21]. The version presented in this part also slightly differs from the high-order skeletal semantics of Part II that allows function types and anonymous functions.

3.1 Example

A skeletal semantics is composed of types, filters, hooks, and rules. Types represent categories for the elements of the language. 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 3.1.1 (Types)**.** We write examples using the legacy Necro syntax [21], and use as a running example an imperative language called IMP. It is slightly different from the language presented in Chapter 2, as we merge arithmetic and boolean expressions into a single category. Types are defined with the keyword `type`; base types (`int`, `bool`, `ident`, `state`, and `value`) are only declared, while program types (`expr` and `stmt`) are declared alongside the signature of their constructors, like an algebraic datatype definition in OCaml.

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

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

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 functions called *filters*, which represent the basic operations of the language. We can reason on the semantics as a whole by assuming some properties on these filters. For instance, the final representation of `state` does not matter as long as we define read and write operations as filters.

Example 3.1.2 (Filters)**.** Filters are declared with the keyword `val`. They are explicitly typed, using the keyword `unit` in case of a missing input or output. We consider the following filters for IMP.

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

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

Example 3.1.3 (Hooks)**.** We define the hooks `hexpr` and `hstmt` for the evaluation of respectively expressions and statements in Figure 3.1; the matched term is declared with the keyword

```
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 v' = neg (v) in
  (s1, v')
```

```
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
```

Figure 3.1: Hooks in IMP

`matching`. Branchings are written `branch .. (or ..)* end`, while the other sorts of skelements (filter call, hook call, return) are not preceded by keywords, as we can easily tell them apart.

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

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

This version of skeletal semantics also supports basic parametric polymorphism. We do not present this feature, as it is not supported by neither the Coq extraction (Section 3.5) nor the small-step transformation (Chapter 4).

3.2 Syntax of Skel

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

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

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

The skeletal semantics of a language consists in:

- a set ($T = T_b \cup T_\mathrm{p}$) of types ranged over by s, combining base types T_b, and program types T_p ranged over by s_p;
- a set C of constructors, with a typing function $\texttt{ctype} : C \to \widetilde{T} \times T_\mathrm{p}$;
- a set F of filters, with a typing function $\texttt{ftype} : F \to \widetilde{T} \times \widetilde{T}$;
- a set H of hooks, with a typing function $\texttt{htype} : H \to (\widetilde{T} \times T_\mathrm{p}) \times \widetilde{T}$;
- a set R of rules of the form $h(\widetilde{y}, c(\widetilde{x})) := S$, assuming we have $\texttt{htype}(h) = ((\widetilde{s}, s_\mathrm{p}), \widetilde{s'})$, $\texttt{ctype}(c) = (\widetilde{s''}, s_\mathrm{p})$, $\|\widetilde{y}\| = \|\widetilde{s}\|$, 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 hook $\widetilde{T} \times T_\mathrm{p}$ is composed of auxiliary terms of type $\widetilde{T}$ and of the term being evaluated of program type T_p. We define three projections $\texttt{htype}_{\texttt{in}}$, $\texttt{htype}_{\texttt{p}}$, and $\texttt{htype}_{\texttt{out}}$ so that if $\texttt{htype}(h) = ((\widetilde{s}, s_\mathrm{p}), \widetilde{s'})$, then $\texttt{htype}_{\texttt{in}}(h) = \widetilde{s}$, $\texttt{htype}_{\texttt{p}}(h) = s_\mathrm{p}$, and $\texttt{htype}_{\texttt{out}}(h) = \widetilde{s'}$.

The Skel meta-language is strongly statically typed. As such, the variables v in terms and skeletons are associated with a base or program type. As hinted at in Section 3.1, the typing system is polymorphic but we ignore the details for the sake of simplicity.

A rule $h(\widetilde{y}, c(\widetilde{x})) := S$ defines the behavior of h on the constructor c by the skeleton S, which describes the sequence of reductions to perform using the input variables $\widetilde{x}$ and $\widetilde{y}$. We assume the variables $\widetilde{x}, \widetilde{y}$ to be pairwise distinct, and to contain the free variables of S. We suppose that

at most one rule handling c for h exists in R. The matching does not have to be exhaustive: a hook h without a rule for c simply cannot reduce terms with c as head constructor.

3.3 Concrete Interpretation

The dynamic of a skeletal semantics is given by the *concrete interpretation* of the rules defining its hooks [17], which corresponds to a big-step semantics. This interpretation computes the result of applying a hook to a term by inductively interpreting the skeleton of the rule of the corresponding constructor and hook, under some environment mapping skeleton variables to values.

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

Example 3.3.1. For IMP, we instantiate the base type `ident` with strings, `int` with integers ($\mathbb{Z}$), `bool` with Booleans ($\mathbb{B} = \{\top; \bot\}$), `value` with the disjoint union ($\mathbb{Z} \cup \mathbb{B}$), and `store` with partial functions from strings to values. The interpretations of the different filters are the following:

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

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

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

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

$$\frac{\Sigma(\widetilde{t}) \Downarrow_h \widetilde{b}}{\Sigma \vdash \texttt{Hook}\ h\ \widetilde{t} \Downarrow \widetilde{b}} \qquad \frac{\mathcal{R}_f(\widetilde{\Sigma(t)}) \Downarrow \widetilde{b}}{\Sigma \vdash \texttt{Filter}\ f\ \widetilde{t} \Downarrow \widetilde{b}} \qquad \frac{\widetilde{\Sigma(t)} = \widetilde{b}}{\Sigma \vdash \texttt{Return}\ \widetilde{t} \Downarrow \widetilde{b}} \qquad \frac{S_i \in \widetilde{S} \qquad \Sigma \vdash S_i \Downarrow \widetilde{b}}{\Sigma \vdash \texttt{Branching}\ \widetilde{S} \Downarrow \widetilde{b}}$$

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

Figure 3.2: Inference Rules for the (Inductive) Concrete Interpretation

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

The interpretation $\Sigma \vdash S \Downarrow \widetilde{b}$, defined in Figure 3.2, is a relation stating that S outputs the values $\widetilde{b}$ under the environment Σ; it assumes that the free variables of S are in the domain of Σ. A LetIn structure is evaluated sequentially, starting with K under Σ and then continuing with S under the environment updated with the outputs of K. The environment Σ is used when interpreting skelements to turn the input terms $\widetilde{t}$ into values. These values are simply returned in the case of a `Return` skelement. A filter call looks for a possible result related to these values in $\mathcal{R}_f$. A branching returns the result of one of its branches; it does not matter if some branches are stuck or non-terminating as long as one branch succeeds. To evaluate a hook call, we first compute the arguments of the hook, and find the rule corresponding to its constructor. We then interpret the skeleton of the rule under a new environment mapping the free variables of the skeleton to the appropriate values taken from Σ. Figure 3.2 uses an auxiliary judgment $\widetilde{a} \Downarrow_h \widetilde{b}$ saying that the hook h taking $\widetilde{a}$ as input can output $\widetilde{b}$. This allows for simpler definitions and is use extensively in the proofs (see Chapter 5).

Note that our approach differs from the one of [17]: they take the smallest fixpoint of a functional describing one step of the relation, whereas we directly define the semantics as an inductive definition. This interpretation is inherently big-step, as a judgment $\Sigma \vdash S \Downarrow \widetilde{b}$ computes the final value returned by S. It is also non-deterministic, as apparent in the rules for branching and filter call, since $\mathcal{R}_f$ may relate several results to the same input.

Example 3.3.2. We interpret the following statement `st` with the hook `hstmt`:

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

The program is a simple conditional: if `length` $\stackrel{?}{=}$ `width` $+ 2$, then `width` $:= 4$, otherwise `flag` $:= \bot$. We evaluate this statement in the state $a_0 = \{$`length` $\mapsto 4\,;$`width` $\mapsto 2\}$, so we expect the result to be $a_1 = \{$`length` $\mapsto 4\,;$`width` $\mapsto 4\}$. We remind that the rule is `hstmt(s, If(e1,t2,t3)) :=` $S_{\texttt{If}}$, where $S_{\texttt{If}}$ is the following skeleton (cf. Figure 3.1):

```
let (s1, v) = hexpr (s, e1) in
branch
  let () = isTrue (v) in          (* | S1  *)
  hstmt (s1, t2)                  (* |     *)
or
  let () = isFalse (v) in         (* || S2 *)
  hstmt (s1, t3)                  (* ||    *)
end
```

We interpret this skeleton in the initial environment:

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

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

We then evaluate the branching in the extended environment $\Sigma_1 = \Sigma_0 + \{\texttt{s1} \mapsto a_0; \texttt{v} \mapsto \top\}$. Only the branch guarded by `isTrue` can succeed; given our definition of $\mathcal{R}_{\texttt{isTrue}}$, we have $\mathcal{R}_{\texttt{isTrue}}(\Sigma_1(v)) \Downarrow ()$, meaning that we pass this filter call.

Finally, we compute the recursive call on the values $(a_0, \texttt{Assign("width",Iconst(4))})$. Once again we look for the corresponding rule in R and create a new environment for this evaluation.

$$\frac{\Sigma(\widetilde{t}) \Uparrow_h}{\Sigma \vdash \texttt{Hook}\ h\ \widetilde{t} \Uparrow} \qquad \frac{h(\widetilde{y}, c(\widetilde{x})) := S \in R \qquad \{\widetilde{y \mapsto a}\} + \{\widetilde{x \mapsto a'}\} \vdash S \Uparrow}{(\widetilde{a}, c(\widetilde{a'})) \Uparrow_h}$$

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

Figure 3.3: Coinductive Interpretation

As expected, this computation returns a_1 and it closes the derivation of $\Sigma_0 \vdash S_{\texttt{If}} \Downarrow a_1$.

$$\dfrac{\dfrac{\dfrac{\vdots}{\Sigma_2 \vdash S_{\texttt{Equal}} \Downarrow (a_0, \top)}}{\Sigma_0 \vdash \texttt{hexpr (s,e1)} \Downarrow (a_0, \top)} \qquad \dfrac{\dfrac{\mathcal{R}_{\texttt{isTrue}}(\top) \Downarrow ()}{\Sigma_1 \vdash \texttt{isTrue (v)} \Downarrow ()} \qquad \dfrac{\dfrac{\vdots}{\Sigma_3 \vdash S_{\texttt{Assign}} \Downarrow a_1}}{\Sigma_1 \vdash \texttt{hstmt (s1,t2)} \Downarrow a_1}}{\dfrac{\Sigma_1 \vdash S_1 \Downarrow a_1}{\Sigma_1 \vdash \texttt{Branching}(S_1, S_2) \Downarrow a_1}}}{\Sigma_0 \vdash S_{\texttt{If}} \Downarrow a_1}$$

3.4 Coinductive Interpretation

The concrete interpretation defined in Section 3.3 is a standard big-step semantics, and cannot be used to express infinite computations. As an additional tool, we define a coinductive semantics for the meta-language Skel. The rules are given in Figure 3.3, where the double lines mean the rules must be interpreted coinductively.

There is no rule for returns and filters, as they cannot diverge. A hook call diverges if the evaluation of the corresponding skeleton diverges. A branching diverges if one of the branches diverges. For LetIns, we have two different rules, as there is two opportunities for infinite behaviors. A sequence diverges if either the first skelement diverges, or the skelement succeeds (using the concrete interpretation) but the rest of the computation then diverges.

This coinductive interpretation is used in Chapter 5 to prove that the transformation of Chapter 4 preserves infinite behaviors.

3.5 Necro

Necro [21] is an OCaml implementation for the manipulation of skeletal semantics. Users can easily define programming languages via the readable syntax we use for our examples (e.g.,

Figure 3.1). Necro then provides tools to manipulate skeletal semantics. The transformation presented in Chapter 4 is implemented in this framework.

The main feature of Necro is its ability to create an OCaml interpreter from a skeletal semantics. The interpreter is parameterized by the types and functions representing the base types and filters. Once these have been instantiated, the interpreter provides a module containing an evaluation function for each hook of the skeletal semantics. The evaluation follows the approach of the concrete interpretation presented in Section 3.3, recursively calling the evaluation function each time a hook is encountered. We extend the tool to also generate a small-step interpreter (see Section 6.1).

More importantly, Necro is able to export a skeletal semantics into a formal Coq definition of the language. This tool outputs a deep embedding—i.e., a data structure containing the user definitions but without any semantic meaning—parameteric in the base types and filters. The concrete interpretation of Section 3.3 is independently formalized in Coq, and can be applied to give a semantics to user languages. We extensively use this tool to automatically provide equivalence certificates for the results of our transformation (see Section 5.2).

Chapter 4

Small-Step Transformation

Given a big-step skeletal semantics, we transform it to produce a skeletal semantics whose concrete interpretation behaves like a small-step interpretation of the initial semantics. We first present the main steps of the transformation on the IMP language (defined in Section 3.1), before defining the transformation formally in Section 4.2.

4.1 Overview on an Example

4.1.1 Coercions

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

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

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

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

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

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

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

The transitory semantics at this point of the transformation is available in Appendix B.2.

4.1.2 New Constructors

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

In some cases, however, we cannot reconstruct using the same constructor after a step. The different problematic situations are detailed in Section 4.2.2; here we only describe the main problem, namely that we cannot remember two versions of the same term.

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

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

- It is a tail-call, i.e., a final hook call forwarding its return values. Then there is nothing to do.
- The terms being evaluated are only used once. In this case, we can reuse the same constructor to reconstruct a term.
- Some of the evaluated terms are used elsewhere. The naive reconstruction does not work, and we need to create an additional constructor.

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

We illustrate our analysis on several IMP constructors. For `Plus`, we can reuse the constructor after each hook call:

```
| 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)
```

In both cases, the arguments of the calls—respectively `s`, `e1` and `s1`, `e2`—are not needed in the rest of the skeleton, so we can reconstruct at each program point reusing `Plus`.

For `Seq`, we have:

```
| Seq (t1, t2) ->
  let s1 = hstmt (s, t1) in              (* reuse *)
  hstmt (s1, t2)                         (* tail-call *)
```

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

The analysis gets more interesting for `While`:

```
| 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
```

The third hook 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 hook call since `t2` is needed in the tail-call.

For each of these two calls, we need to create a new constructor corresponding to their respective program point. The new constructors are built with two different kinds of arguments. Firstly, we create an argument for every term being evaluated at the analyzed hook call, namely `s`, `e1` for the first one, and `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 hook call—are duplicated. In the end, we extend the definition of IMP with the following constructors:

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

We also add a new rule for each constructor introduced. The new skeleton consists in resuming the computation from the corresponding analyzed hook call, updating the input of the call to make use of the new arguments of the constructor. The resulting rule for `While1` is almost the same as the one for `While`, while the skeleton of `While2` covers only its last two skelements. We do not modify the rules for `While` or the other constructors at this stage.

```
hook hstmt (s : state, t : stmt) matching t : state =
| ...
| While1 (s0, e0, e1, t2) ->
  let (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))
```

The added rules become useful when we change the rules to introduce calls to `While1` and `While2`, as part of the last step of the transformation. Note that these added rules already include the information from the analysis, see Section 4.2.2 for details.

The transitory semantics at this point of the transformation is provided in Appendix B.3.

4.1.3 Make the Skeletons Small-Step

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

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

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

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

More importantly, we need to update the skeletons themselves. We recall that skeletons are sequences of operations (skelements), which are mostly composed of filter calls and hook calls. For our transformation, we consider that only hook calls correspond to reduction steps. The reason is that filters represent simple atomic operations that are not meant to be interrupted,

while hook calls often correspond to the evaluation of a subterm. Thus, this last phase essentially focuses on transforming hook calls. We also need to take care of the result returned at the end of a skeleton, so that its type matches the updated output type of its hook.

We distinguish four cases, that we illustrate with rules from IMP. In each example, for readability, we provide small-step inference rules equivalent to the concrete interpretation of the output skeletal semantics rules.

1 If the last skelement of a skeleton is not a tail-call, then we need to wrap the results differently to match the new typing of the hook. The final result needs to be coerced to the input program type using one of the new constructors defined in the first phase (Section 4.1.1). For type checking, we also have to return the other arguments of the hook, even if they are not of any use. For instance, the output of the rule for the `Iconst` constructor in the initial big-step skeleton is `(s, v)` (cf. Figure 3.1). Using a coercion, we turn this pair into an expression `Ret_hexpr(s, v)`; if we could we would return this term only, but the output type of the `hexpr` hook is `(state * expr)`, so we also return a useless copy of s. The rule for `Iconst` is thus as follows.

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

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

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

```
| Plus (e1, e2) ->
  branch
    let (w1, w2) = hexpr (s, e1) in
    (w1, Plus (w2, e2))
  or
    let (s1, v1) = getRet_hexpr (e1) in
    ...
  end
```

$$\frac{s, e_1 \to w_1, w_2}{s, e_1 + e_2 \to w_1, w_2 + e_2}$$

$$\frac{e_1 = (s_1, v_1) \qquad \dots}{s, e_1 + e_2 \to \dots}$$

where the dots correspond to the transformation of the second hook call. In the first branch, we reconstruct as `(w1, Plus (w2, e2))`, overwriting the variables `s` and `e1` with the new terms resulting from the reduction step. In the second branch, we extract the coerced value using the hook defined alongside the constructor in Section 4.1.1. Even though we use a branching, the reduction is deterministic, as the definition of `getRet_hexpr` is restricted to coerced values,

while `hexpr` operates only on terms that are not coerced values.

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

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

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

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

```
| While1 (s0, e0, e1, t2) ->
  branch                                  (* |                          *)
    let (w1, w2) = hexpr (s0, e0) in      (* | first hook call          *)
    (s, While1 (w1, w2, e1, t2))          (* | transformed as           *)
  or                                      (* | previously               *)
    let (s1, v) = getRet_hexpr (e0) in    (* |                          *)
    branch                                (*   # initial structure      *)
      let () = isTrue (v) in              (*   #                        *)
      (s, While2 (s1, t2, e1, t2))        (*     || second hook call    *)
    or                                    (*   #                        *)
      let () = isFalse (v) in             (*   #                        *)
      (s, Ret_hstmt s1)                   (*    ## coerced return       *)
    end                                   (*   #                        *)
  end                                     (* |                          *)
```

$$\frac{s_0, e_0 \to w_1, w_2}{s, \texttt{While1}(s_0, e_0, e_1, t_2) \to s, \texttt{While1}(w_1, w_2, e_1, t_2)}$$

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

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.

4 Finally, we also cut the tail-calls to simplify the semantics. This creates administrative small-steps of the form $s, \texttt{If}(\texttt{True}, t_2, t_3) \to s, t_2$ where no subcomputation takes place. It generates

behaviors closer to usual pen-and-paper definitions. We can see this with the `Seq` constructor: the first hook call is transformed as previously, but the second is turned into a return.

```
| Seq (t1, t2) ->
  branch                                (* |                              *)
    let (w1, w2) = hstmt (s, t1) in     (* | first hook call               *)
    (w1, Seq (w2, t2))                  (* | transformed as previously     *)
  or                                    (* |                              *)
    let s1 = getRet_hstmt (t1) in       (* |                              *)
    (s1, t2)                            (*   || 2nd call becomes return   *)
  end                                   (* |                              *)
```

$$\frac{s, t_1 \to w_1, w_2}{s, t_1; t_2 \to w_1, w_2; t_2} \qquad \frac{t_1 = s'}{s, t_1; t_2 \to s', t_2}$$

This final phase produces a small-step skeletal semantics where each hook call reduces its arguments only once. It is equivalent to the initial big-step one, in the sense that evaluating a term with either semantics produces the same value. The result of the complete transformation on IMP, as well as the different intermediate semantics, can be found in Appendix B. We state the equivalence between the two semantics in Section 5.2.

4.2 Formal Transformation Phases

We define formally the different phases of our transformation using the notations introduced in Section 3.2. We start with a given skeletal semantics $(T_b, T_p, C_0, F, H_0, \mathrm{R_{BS}}, \texttt{ctype}, \texttt{ftype}, \texttt{htype})$. The elements describing types and filters (i.e., T_b, T_p, F, and `ftype`) are not modified by the transformation, and are implicitly carried over to all intermediate semantics. The sets of constructors (C_0) and hooks (H_0), as well as their typing functions `ctype` and `htype`, are expanded during the transformation. To simplify the notations, we do not change the names of the typing functions throughout the transformation. The main focus of the transformation is the set of rules defining the semantics of the user language. We start with a set $\mathrm{R_{BS}}$, and each phase of the translation expands or modifies it. Intermediate semantics are thus described by their sets of constructors and hooks, but most importantly by their set of rules.

4.2.1 Coercions

The first step is to add coercions for return values like in Section 4.1.1. For every hook we add a new constructor to pack its result as well as the corresponding hook to unpack it.

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

We remind that $\texttt{htype}_{\texttt{in}}$, $\texttt{htype}_{\texttt{p}}$, and $\texttt{htype}_{\texttt{out}}$ are the projections of the typing function $\texttt{htype} : H_0 \to (\widetilde{T} \times T_p) \times \widetilde{T}$, representing respectively the types of input states, the program type being reduced, and the output type of the hook. Each coercion turns the return values $\texttt{htype}_{\texttt{out}}(\texttt{h})$ of the corresponding hook into an executable program of type $\texttt{htype}_{\texttt{p}}(\texttt{h})$.

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

$$\mathrm{R}_{\mathrm{getRet}} = \{\texttt{getRet\_h}(\texttt{Ret\_h}\ \widetilde{v}) := \texttt{Return}\ \widetilde{v} \mid \texttt{h} \in H_0 \wedge \|\texttt{htype}_{\texttt{out}}(\texttt{h})\| = \|\widetilde{v}\|\}$$

Note that the rules $\mathrm{R}_{\mathrm{getRet}}$ do not need to be transformed further. The following phases of the transformation focus on changing the initial big-step rules. We finish this phase with the constructors C_1, hooks H_1, and rules $\mathrm{R}_{\mathrm{BS}} \cup \mathrm{R}_{\mathrm{getRet}}$.

4.2.2 New Constructors

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

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

We proceed in two steps: we first analyze the rules to annotate the hook calls, then we create the needed new constructors and their rules.

Analysis

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

- after a filter call;

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

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

Secondly, we need to take into account the non-determinism induced by a branching. In a skeleton like `let` $\widetilde{v}$ = `Branching(Hook` h_1 $\widetilde{t_1}$, `Hook` h_2 $\widetilde{t_2}$) `in Hook` h $\widetilde{t}$, two different reduction paths lead to the hook call after the branching. The premise of constructor reuse is that reevaluating the skeleton from the start should lead to the same evaluation context. However, reevaluating the skeleton in such a situation may take a different path and reach the last hook call with different values bound to the variables in $\widetilde{v}$. As such, we give up on reuse if the analyzed hook call is in the continuation of a branching.

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

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

L is a boolean indicating if we are at the toplevel of the LetIn structure of the main skeleton, used to detect tail-calls;

b is a boolean indicating if we are at a position allowing for reuse, i.e., indicating whether we are after a filter call or in the continuation of a branching;

V is the list of the variables that are only used once throughout the whole initial skeleton;

hr is the name of the hook corresponding to the rule being analyzed, also used to detect tail-calls, as detailed below.

The analysis is defined in Figure 4.1. Given a rule $h(\widetilde{y}, c(\widetilde{x})) := S$, we compute the set of variables that are used exactly once in S, written `SglUse`(S), and we fix V and h as respectively `SglUse`(S) and h. The parameter V and hr are constants while L and b are initialized at $\top$ and may change during the analysis.

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

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

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

Figure 4.1: Hook calls analysis

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

```
hook h2 (e : expr) matching e : value = ...
hook h1 (l : exprlist) matching l : value =
| Cons(e, l1) -> h2 (e)
```

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

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

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

A hook 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 hook 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. Hook 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, updating the set of rules as follows.

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

We now have the extended set of rules $\mathrm{R}_{\mathrm{lbl}} \cup \mathrm{R}_{\mathrm{getRet}}$, on constructors C_1 and hooks H_1.

Generation

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

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

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

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

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

The operation $[\![S]\!]^r_E$ inductively goes through S, building the continuation in its parameter E and returning the set of new rules.

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

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

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

the typing of c is $\texttt{ctype}(c) = ((\widetilde{u}, \widetilde{s}), \texttt{htype}_\texttt{p}(\texttt{hr}))$, since the new constructor should build the program type evaluated by the analyzed rule.

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

Example 4.2.1. We apply the generation process to the second hook call in the rule for `While` presented in Section 4.1.2. We reach this hook call K in a context E:

$$K := \texttt{ Hook (New While2) hstmt (s1, t2)}$$
$$E := <[\cdot] \mid \texttt{s2} \mid \texttt{Hook Tail hstmt (s2, While (e1, t2))}>$$

The new constructor needs the variables `(e1 : expr)` and `(t2 : stmt)` to evaluate E, but not `s2`, as it is defined by E. We get the input types $\widetilde{u} = (\texttt{state}, \texttt{stmt})$ of the hook call, for which we create the fresh variables $\widetilde{w} = (\texttt{s0}, \texttt{t0})$. From there we type the new constructor and create its rule:

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

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

It roughly corresponds to the following big-step inference rule, with annotations to help us transform it to small-step.

$$\frac{s_0, t_0 \Downarrow s_2 \qquad s_2, \texttt{While}(e_1, t_2) \Downarrow s_3}{s, \texttt{While2}(s_0, t_0, e_1, t_2) \Downarrow s_3}$$

Example 4.2.2 (Without constructor reuse)**.** If we do not aim at reusing initial constructors, the analysis simply creates a fresh constructor for each hook call, which are processed exactly as above. To evaluate terms of the form $\texttt{Plus}(e_1, e_2)$, we create `Plus1` for the evaluation of e_1 and `Plus2` for e_2.

```
hexpr(s, Plus1(s0, e0, e2)) := let (s1, v1) = Hook Reuse hexpr (s0, e0) in ...
hexpr(s, Plus2(s0, e0, v1)) := let (s2, v2) = Hook Reuse hexpr (s0, e0) in ...
```

They roughly corresponds to the following big-step inference rules.

$$\frac{s_0, e_0 \Downarrow s_1, v_1 \qquad s_1, e_2 \Downarrow s_2, v_2}{s, (s_0, e_0) +_1 e_2 \Downarrow s_2, (v_1 + v_2)} \qquad \frac{s_0, e_0 \Downarrow s_2, v_2}{s, (s_0, e_0) +_2 v_1 \Downarrow s_2, (v_1 + v_2)}$$

To sum up, the generation phase consists of running the scanning $[\![S_r]\!]^r_{[\cdot]}$ for every rule $r = (\mathtt{hr}(\widetilde{y}, c_r(\widetilde{x})) := S_r) \in \mathrm{R}_{\mathrm{lbl}}$, resulting in a new set $C_2 \supseteq C_1$ of constructors and a new set $\mathrm{R}_{\mathrm{gen}} \supseteq \mathrm{R}_{\mathrm{lbl}}$ of rules.

We now have a set of rules $\mathrm{R}_{\mathrm{gen}} \cup \mathrm{R}_{\mathrm{getRet}}$, defined on constructors C_2 and hooks H_1.

Optimization

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

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

Example 4.2.3. Consider the following hook and constructor:

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

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

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

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

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

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

4.2.3 Distribute Branchings

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

Reconstructing terms is problematic for hooks in nested computations. In the structure `let x = (let y = Hook eval t in S1) in S2`, a small-step transformation of `eval` may return a partially evaluated term which ends up stored in `x`, while `S2` may expect `x` to contain a value; for example `S2` may start by filtering `x` with `isTrue`. We avoid the issue by sequencing such nested computations as `let y = Hook eval t in let x = S1 in S2`, and the hook transformation of Section 4.2.4 ensures that `x` may only contain a value.

The grammar of skeletal semantics of Section 3.2 does not allow for nested `LetIn`, but the same issue is present for branchings inside `LetIn`. We therefore recursively transform a skeleton of the form $(\texttt{let}\ \widetilde{v} = \texttt{Branching}(S_1, S_2)\ \texttt{in}\ S)$ into $\texttt{Branching}(\texttt{let}\ \widetilde{v} = S_1\ \texttt{in}\ S, \texttt{let}\ \widetilde{v} = S_2\ \texttt{in}\ S)$, so that hook calls in S_1 and S_2 can be transformed in the final phase.

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

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

We apply this operation to every skeleton of our semantics:

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

Note that duplicating the continuations recursively will not exponentially grow the size of the final small-step semantics. We generate new constructors before this phase, hence the generated

constructors are shared between branches, avoiding any unwanted bloat. As an optimization, this distribution and duplication could also be skipped for branchings only containing filter calls and no hook calls.

The output of this phase is the set of rules $\mathrm{R}_{\mathrm{dist}} \cup \mathrm{R}_{\mathrm{getRet}}$, still defined on C_2 and H_1.

4.2.4 Make the Skeletons Small-Step

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

$$\text{Forall } h \in H_0 \text{ with } \texttt{htype}(h) = ((\widetilde{s}, s_{\mathrm{p}}), \widetilde{u}), \text{ we redefine: } \texttt{htype}(h) = ((\widetilde{s}, s_{\mathrm{p}}), (\widetilde{s}, s_{\mathrm{p}}))$$

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

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

$$S ::= \texttt{let } \widetilde{v} = K \texttt{ in } S \mid \texttt{Branching } \widetilde{S} \mid \texttt{Hook Tail } h\ \widetilde{t} \mid \texttt{Return } \widetilde{t}$$
$$K ::= \texttt{Filter } f\ \widetilde{t} \mid \texttt{Hook Reuse } h\ \widetilde{w} \mid \texttt{Hook (New } c\texttt{) } h\ \widetilde{t} \mid \texttt{Return } \widetilde{t}$$

A skeleton $\texttt{let } \widetilde{v} = \texttt{Branching } \widetilde{S} \texttt{ in } S$ is impossible because of the distribution step of Section 4.2.3. A tail-call is necessarily a final hook, so a skeleton $\texttt{let } \widetilde{v} = \texttt{Hook Tail } h\ \widetilde{t} \texttt{ in } S$ is also not possible. Whenever a hook call is annotated `Reuse`, the analysis of Section 4.2.2 implies that its input terms are all variables $\widetilde{w}$. If a skeleton ends with K that is not a tail-call (i.e., a skelement $\texttt{Filter } f\ \widetilde{t}$ or $\texttt{Hook } a\ h\ \widetilde{t}$ with $a \neq \texttt{Tail}$), we can transform it into an equivalent skeleton following the grammar above by delaying the return. For this, we replace K with the skeleton $\texttt{let } \widetilde{z} = K \texttt{ in Return } \widetilde{z}$, where $\widetilde{z}$ are freshly created variables, in number corresponding to the output type of K. As such, it is sufficient to define our transformation process on skeletons respecting the simplified grammar.

The transformation relies on a substitution to remember how the initial arguments of the rule are changed through the different hook calls, as we show in Example 4.2.5. A substitution σ is a total mapping from variables to terms equal to the identity except on a finite set of variables called its domain. We write $x\sigma$ for the application of σ to x, ϵ for the identity substitution, and $[t/x]$ for the substitution whose domain is $\{x\}$ and such that $x\sigma = t$. We extend the notion to terms $t\sigma$ and tuples $\widetilde{t}\sigma$ as expected. Given two substitutions σ and σ', we define their sequence $\sigma\,;\sigma'$ so that $x(\sigma\,;\sigma') = (x\sigma)\sigma'$ for all x.

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

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

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

Figure 4.2: Transformation of a Skeleton

Given an extended skeleton S_r, a rule $r = (\texttt{hr}(\widetilde{y}, c_r(\widetilde{x})) := S_r)$, and a substitution σ representing the knowledge accumulated so far, the transformation $\| S_r \|^r_\sigma$ defined in Figure 4.2 results in a plain skeleton—without annotations. The first three rules are simple inductive cases, while the last four are the cases sketched in the overview (Section 4.1.3).

We coerce the results of a final `Return` skelement with the constructor `Ret_hr` defined in Section 4.2.1. We remind that we also return the environment variables $\widetilde{y}$ of the rule to respect the updated typing of the hook.

A tail-call is simply turned into a return, as the hook being called is identical to the one where the current rule is defined (see Figure 4.1).

As explained in Section 4.1.3, a hook call annotated (`New` c) is turned into a return with a term built with c, so that the rule created for c in Section 4.2.2 can later perform the expected small-step reduction. One might be surprised the hook call disappears, it is simply delegated to the rule for the new constructor c (see Section 4.2.2 right before Example 4.2.1). To compute the arguments of c, we distinguish in its rule $r_c = (\texttt{hr}(\widetilde{y}, c(\widetilde{w_c}, \widetilde{z_c})) := S_c)$ the variables $\widetilde{w_c}$ used as input of the analyzed call from the ones $\widetilde{z_c}$ necessary to compute S_c. The resulting skelement is then `Return` $(\widetilde{y}, c(\widetilde{t}, \widetilde{z_c}))$, where we replace $\widetilde{w_c}$ with the terms $\widetilde{t}$ being reduced. We know the variables $\widetilde{z_c}$ exist at the program point we are transforming, because they have been extracted from the same hook call in r during the analysis. Similarly, the variables $\widetilde{y}$ of r_c are the same as the environment variables of r by construction, so we can reuse them.

Example 4.2.4. The call for which we need to create `While2` is of the form (cf. Section 4.1.2):

```
hook hstmt (s : state, t : stmt) matching t : state =
...
| While1 (s0, e0, e1, t2) ->
  ...
  let (s1, v) =  ... in
  ...
  let s2 = hstmt (new While2) (s1, t2) in  ...
```

The rule created for `While2` in Example 4.2.1 is of the form `hstmt(s, While2(s0, t0, e1, t2)) := S`, where $\widetilde{w_c} = (\texttt{s0}, \texttt{t0})$ and $\widetilde{z_c} = (\texttt{e1}, \texttt{t2})$ are used to compute respectively the analyzed call and the rest of the skeleton. Replacing `(s0, t0)` with the arguments of the call `(s1, t2)`, the resulting skelement is `Return (s, While2 (s1, t2, e1, t2))`, and we see that `s`, `e1`, and `t2` are bound at the point we transform.

We change a hook call that can reuse its constructor into a branching representing its possible behaviors. The first branch begins by reducing the hook one step further `let` $\widetilde{z}$ = `Hook` h $\widetilde{w}$ `in` ..., storing the results in some fresh variables $\widetilde{z}$. We then reconstruct a configuration using the constructor c_r of the rule being processed. Starting from the initial input $(\widetilde{y}, c_r(\widetilde{x}))$, we apply σ before changing $\widetilde{w}$ 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 hook call, as we can see in the `Plus` example below. The second branch covers the case where the term represented by w is a coerced set of values. We extract the content of w into the variables $\widetilde{v}$ of the initial skeleton and continue transforming S, remembering that w is equal to `Ret_h`$(\widetilde{v})$.

Example 4.2.5. Consider the rule for `Plus`:

```
| 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)
```

The first call is turned into two branches, the first one stepping once `(s, e1)` into some fresh variables `(z1, z2)`. The reconstructed configuration is simply the input `(s, Plus (e1, e2))` where `s` and `e1` are replaced by `z1` and `z2`. In the second branch, we extract the content of `e1`, and then transform the rest of the skeleton, remembering that `e1 = Ret_hexpr(s1, v1)` in σ.

Transforming the second hook call illustrates why we need σ. As for the first call, the first branch steps once `(s1, e2)` into some fresh variables `(z3, z4)` and then 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_hexpr(s1, v1), e2))`, and now we can turn `(s1, e2)` into `(z3, z4)`, resulting in the configuration `(s, Plus (Ret_hexpr(z3, v1), z4))`. The second branch continues transforming the rest of the skeleton, where we no longer need the substitution. In the end, we obtain:

```
hook hexpr (s : state, e : expr) matching e : state * expr =
| Plus (e1, e2) ->
  branch                                    (* |                          *)
    let (z1, z2) = hexpr (s, e1) in         (* | first hook call          *)
    (z1, Plus (z2, e2))                     (* |                          *)
  or                                        (* |                          *)
    let (s1, v1) = getRet_hexpr (e1) in     (* |                          *)
    branch                                  (*  ||                        *)
      let (z3, z4) = hexpr (s1, e2) in      (*  || second hook call       *)
      (s, Plus (Ret_hexpr (z3, v1), z4))    (*  || <- need substitution   *)
    or                                      (*  ||                        *)
      let (s2, v2) = getRet_hexpr (e2) in   (*  ||                        *)
      let v = add (v1, v2) in               (*    # filter unchanged      *)
      (s, Ret_hexpr (s2, v))                (*     ## coerced return      *)
    end                                     (*  ||                        *)
  end                                       (* |                          *)
```

$$\frac{s, e_1 \to z_1, z_2}{s, e_1 + e_2 \to z_1, z_2 + e_2} \qquad \frac{e_1 = (s_1, v_1) \qquad s_1, e_2 \to z_3, z_4}{s, e_1 + e_2 \to s, (z_3, v_1) + z_4} \qquad \frac{e_1 = (s_1, v_1) \qquad e_2 = (s_2, v_2)}{s, e_1 + e_2 \to s, (s_2, v_1 + v_2)}$$

If we do not reuse the initial constructors, we do not need the substitution σ as we cannot have nested hook 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 4.2 become:

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

Example 4.2.6 (Without constructor reuse)**.** If we do not reuse `Plus`, the rules for `Plus`, `Plus1`, and `Plus2` behave as follows. Unlike `Plus` in Example 4.2.5, the new constructors takes three arguments by default (c.f., Example 4.2.2 and Appendix B.6).

$$\frac{}{s, e_1 + e_2 \to s, (s, e_1) +_1 e_2} \qquad \frac{s_0, e_0 \to z_1, z_2}{s, (s_0, e_0) +_1 e_2 \to s, (z_1, z_2) +_1 e_2}$$

$$\frac{}{s, (s_0, v_1) +_1 e_2 \to s, v_1 +_2 (s_0, e_2)} \qquad \frac{s_0, e_0 \to z_1, z_2}{s, v_1 +_2 (s_0, e_0) \to s, v_1 +_2 (z_1, z_2)}$$

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

Once again, the last phase of the transformation is applied only to the rules $\mathrm{R}_{\mathrm{dist}}$ defining

the hooks $\mathtt{h} \in H_0$. We also merge in the rules $\mathrm{R}_{\mathrm{getRet}}$ for the hooks `getRet_h` generated in Section 4.2.1 to create the final small-step rule set R_{SS}.

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

The set R_{SS}, defined on C_2 and H_1 (where `htype` has been updated), is the final result of the transformation. It is a set of rules where every $\mathtt{h} \in H_0$ makes a single step of computation.

Chapter 5

Certification of the Transformation

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 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 Section 5.2. This second approach does not attempt to formalize the whole transformation nor the reuse analysis, but it instead 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 3.3 (Figure 3.2). In summary, we write $\widetilde{a} \Downarrow_h \widetilde{b}$ to state that h takes the values $\widetilde{a}$ as input and output the values $\widetilde{b}$. As the transformation provides several intermediate semantics (i.e., set of rules), we extend the notation to include the rule set used, writing for instance $\widetilde{a} \Downarrow_h^{\mathrm{BS}} \widetilde{b}$ for the initial big-step semantics, and $\widetilde{a} \Downarrow_h^{\mathrm{SS}} \widetilde{b}$ for the generated small-step semantics of Section 4.2.4, without reuse in Section 5.1, and with reuse in Section 5.2.

We recall that the inductive interpretation is inherently big-step, as a judgment computes the whole skeleton, so we keep the formal notation $\Downarrow_h^{\mathrm{SS}}$ 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 do not certify the analysis of Section 4.2.2, as it would make the proof substantially more complex. We would need to justify the correctness of the analysis and that

the information is used properly, including for instance the substitution in Section 4.2.4. We only certify the base transformation without reuse, but extend the proof to both finite and diverging computations. The full proof is available in Appendix C.

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 `Ret_h` 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 hook h

$$\widetilde{a} \Downarrow_h^{\mathrm{BS}} \widetilde{b} \iff \exists \widetilde{a'},\ \widetilde{a}\ (\Downarrow_h^{\mathrm{SS}})^*\ (\widetilde{a'}, \texttt{Ret\_h}(\widetilde{b}))$$

where $\mathcal{R}^*$ is the reflexive transitive closure of a relation $\mathcal{R}$.

For diverging derivations, we define infinite small-step reductions coinductively with this single rule:

$$\frac{\widetilde{a} \Downarrow_h^{\mathrm{SS}} \widetilde{b} \qquad \widetilde{b} \overset{\infty}{\to}_h}{\widetilde{a} \overset{\infty}{\to}_h}$$

Big-step divergence (noted $\Uparrow_h$) is defined in Section 3.4 (Figure 3.3). We prove that the following equivalence holds for all initial terms $\widetilde{a}$ and hook h.

$$\widetilde{a} \Uparrow_h^{\mathrm{BS}} \iff \widetilde{a} \overset{\infty}{\to}_h$$

5.1.1 Proof Sketch

The interesting direction is to show that a sequence of small-step reductions implies a big-step evaluation, which can be done in two ways. A first technique [40] is to recognize subcomputations in the sequence of small steps which correspond to subtrees of the big-step derivation. For example, if $\texttt{Plus}(e_1, e_2)$ evaluates into v with small steps, it means that e_1 evaluates to some v_1, e_2 evaluates to some v_2, and $v = v_1 + v_2$. By induction, the two subcomputations for e_1 and e_2 can be turned into big-step derivations which are then combined to create the derivation for $\texttt{Plus}(e_1, e_2)$.

Another strategy [46, 35] relies on a concatenation lemma, stating that we can merge a small step into a big step: if e makes a small step to e' and e' evaluates in a big step to v, then e evaluates in a big step to v. We use this technique as it is easier to automatize for the Coq certification (see Section 5.2). It only works for finite sequences, however; we use a different strategy when dealing with divergence.

The downside of the approach based on the concatenation lemma is that the big-step and small-step semantics need to be defined on the same constructors. However, the initial big-step semantics is not defined on the newly created constructors, such as `While1`, `While2`, `Plus1`, or `Plus2`—remember that we do not reuse the initial constructors in the proof of this section. To bridge the gap between the initial big-step (BS) and the small-step (SS) semantics, we consider

an extended big-step semantics (EBS) defined on all constructors.

The rule set for EBS is noted $\mathrm{R}_{\mathrm{EBS}}$. It corresponds to the situation before going small-step (see Section 4.2.3), i.e., $\mathrm{R}_{\mathrm{dist}}$ (after delaying returns), but with additional custom rules so that the coercion constructors `Ret_h` can be evaluated.

$$\mathrm{R}_{\mathrm{EBS}} \triangleq \mathrm{R}_{\mathrm{dist}} \cup \{h(\widetilde{y}, \texttt{Ret\_h}(\widetilde{v})) := \texttt{Return}\ \widetilde{v} \mid h \in H_0\}$$

The additional rules allow derivations of the following form (see Lemma C.4.4).

$$(\widetilde{a}, \texttt{Ret\_h}(\widetilde{b})) \Downarrow_h^{\mathrm{EBS}} \widetilde{b}$$

As an example, the EBS semantics for IMP—with constructor reuse however—can be found in Appendix B.5.

We thus manipulate three semantics in this proof: the initial BS semantics written by the user, the EBS semantics generated in the middle of the transformation, and the SS semantics, result of the transformation. The proof 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 equivalence is straightforward since EBS has almost the same rules as BS for the initial constructors. Proving that SS implies EBS relies on the concatenation lemma in the finite case, and on the splitting strategy in the infinite one. The opposite direction is done by induction on the EBS derivation.

After some preliminary results, we detail each step in the following subsections.

5.1.2 Transformation Properties

Before stating the equivalence theorems, the proof introduces a few properties and certifies the behavior of the different phases of the transformation.

We define free (fv) and bound (bv) variables of terms and skeletons as expected (see Definitions 1 and 2). We also note NoRedef(S) the statement that a skeleton S does not bind multiple times the same variable name, defined as follows.

$$\begin{aligned}
\mathrm{NoRedef}(K) &\triangleq \top && \texttt{if}\ (K \neq \texttt{Branching}(\ldots)) \\
\mathrm{NoRedef}(\texttt{Branching}\ (S_1, \ldots, S_n)) &\triangleq \forall i,\ \mathrm{NoRedef}(S_i) \\
\mathrm{NoRedef}(\texttt{let}\ \widetilde{v} = K\ \texttt{in}\ S) &\triangleq \begin{cases} \{\widetilde{v}\}, \mathrm{bv}(K), \text{ and } \mathrm{bv}(S) \text{ are disjoint} \\ \mathrm{NoRedef}(K) \\ \mathrm{NoRedef}(S) \end{cases}
\end{aligned}$$

From this, we specify a Static Single Assignment form for skeletons, noted SSA(S) and defined

as follows[1].

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

It states that a skeleton S does not reuse variables names. The transformation is supposed to be applied to such skeletons. A consequent but straightforward part of the proof (Section C.3) ensures the different phases of the transformation preserve the SSA conditions, as to not inadvertently capture free variables.

Then, we prove a few results showing that the different transformation phases work as intended. For example, we verify that the distribution of branchings does not modify the semantics (Lemma C.4.1).

Lemma 5.1.1. For all skeleton S, state Σ, and terms $\widetilde{a}$ such that $\mathrm{SSA}(S)$,

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

This equivalence relies on a plain but important lemma (C.2.7), stating that, under some freshness conditions, the monadic bind on skeletons operates exactly like a LetIn structure:

$$\Sigma \vdash < S_1 \mid \widetilde{x} \mid S_2 > \Downarrow \widetilde{a} \iff \exists \widetilde{b}, \begin{cases} \Sigma \vdash S_1 \Downarrow \widetilde{b} \\ \Sigma + \widetilde{\{x \mapsto b\}} \vdash S_2 \Downarrow \widetilde{a} \end{cases}$$

I.e., $< S_1 \mid \widetilde{x} \mid S_2 >$ behaves like $\texttt{let } \widetilde{x} = \texttt{Branching}([S_1]) \texttt{ in } S_2$, which we could write, with an abuse of notation, as $\texttt{let } \widetilde{x} = S_1 \texttt{ in } S_2$. The proof of Lemma 5.1.1 is then a straightforward induction on S. Similarly, the distribution of branchings does not modify the coinductive semantics for diverging computations (see Lemma C.4.2).

The most important result at this point synthesizes the effects of both the generation of new constructors and the distribution of branchings to relate constructor names in labels and rules in the extended big-step semantics (Lemma C.4.13).

Lemma 5.1.2. For all rule $(h(\widetilde{y}, c(\widetilde{x})) := S)$ in $\mathrm{R_{EBS}}$, if $(\texttt{let } \widetilde{v} = \texttt{Hook } (\texttt{New } c_0)\ h_1\ \widetilde{t} \texttt{ in } S_0)$ is a subskeleton of S, then $\mathrm{R_{EBS}}$ contains a rule of the following form.

$$h(\widetilde{y}, c_0(\widetilde{w}, \widetilde{z})) := \texttt{let } \widetilde{v} = \texttt{Hook Reuse } h_1\ \widetilde{w} \texttt{ in } S_0$$

It states that if we encounter a hook call annotated with a fresh constructor c_0 in the extended big-step skeletal semantics, then we indeed previously generated the corresponding constructor c_0 and its skeleton. Remember that the last phase of the transformation (see Section 4.2.4) changes $(\texttt{let } \widetilde{v} = \texttt{Hook } (\texttt{New } c_0)\ h_1\ \widetilde{t} \texttt{ in } S_0)$ into $\texttt{Return}(\widetilde{y}, c_0(\widetilde{t}, \widetilde{z}))$. This lemma ensures indirectly that

1. The naming "SSA" is a slight 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.

the small-step behavior is not modified, as running h again still computes $h_1\ \widetilde{t}$ followed by (the transformed version of) S_0. The proof mainly checks that the contexts E of Section 4.2.2 used to define new skeletons interacts well with the distribution of branchings of Section 4.2.3.

With the effects of of the different transformation phases made explicit, we state and prove the equivalences between the different rule sets.

5.1.3 Initial and Extended Big-Step Semantics

The intermediate EBS semantics extends the initial BS one. It contains the user-defined rules for the initial constructors (e.g. `Plus`), and the rules created during the generation phase of Section 4.2.2 for the added constructors, like the rules for `While1` and `While2` in Section 4.2.2 or for `Plus1` and `Plus2` in Example 4.2.2.

The only significant transformation between the two semantics is the distribution of branchings (Section 4.2.3), but we prove it does not modify the semantics (Lemma 5.1.1). Thus, the equivalence holds on initial terms (Theorems C.5.1 and C.5.2). To state the theorems, we write $|t|$ for the the canonical injection of an initial term t into the extended semantics.

Theorem 5.1.3 (BS⇒EBS)**.** For all hook h and initial terms $\widetilde{a}$ and $\widetilde{b}$,

$$\widetilde{a} \Downarrow_h^{\mathrm{BS}} \widetilde{b} \implies \widetilde{|a|} \Downarrow_h^{\mathrm{EBS}} \widetilde{|b|}$$

Theorem 5.1.4 (EBS⇒BS)**.** For all hook h, initial terms $\widetilde{a}$, and extended terms $\widetilde{b'}$,

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

We follow the same strategy for diverging computations. Similarly, the corresponding lemma (C.4.2) assures us there is no problem with the distribution of branchings for coinduction, and we can deduce the equivalence between BS and EBS on initial terms (Theorem C.5.3).

Theorem 5.1.5 (Div: BS⇔EBS)**.** For all hook h and initial terms $\widetilde{a}$,

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

The equivalence between big-step and extended big-step is the only part manipulating both initial and extended terms. In the following subsections, all results manipulate extended terms only.

5.1.4 Small-Step Implies Extended Big-Step

We prove a sequence of small-step reductions can be turned into a big-step evaluation in the following sense (Theorem C.7.2).

Theorem 5.1.6 (SS⇒EBS)**.** For all hook h and terms $\widetilde{a}$, $\widetilde{a'}$, and $\widetilde{b}$,

$$\widetilde{a}\ (\Downarrow_h^{\mathrm{SS}})^*\ (\widetilde{a'}, \texttt{Ret\_h}(\widetilde{b})) \implies \widetilde{a} \Downarrow_h^{\mathrm{EBS}} \widetilde{b}$$

As explained previously, in the finite setting, we rely on a concatenation lemma (C.7.1), stating that we can merge a small step into a big step.

Lemma 5.1.7 (Concatenation)**.** For all hook h and terms $\widetilde{a}$, $\widetilde{a'}$, and $\widetilde{b}$,

$$\widetilde{a} \Downarrow_h^{\mathrm{SS}} \widetilde{a'} \wedge \widetilde{a'} \Downarrow_h^{\mathrm{EBS}} \widetilde{b} \implies \widetilde{a} \Downarrow_h^{\mathrm{EBS}} \widetilde{b}$$

The main theorem then follows from iterating this result, starting from $(\widetilde{a'}, \texttt{Ret\_h}(\widetilde{b})) \Downarrow_h^{\mathrm{EBS}} \widetilde{b}$ as a base case. We also use this strategy for the Coq certification, as generating automatically splitting lemmas would be more difficult (see Section 5.2).

For this concatenation result, because the evaluation of tuples ($\Downarrow_h$) is defined mutually with the evaluation of skeletons ($\cdot \vdash \cdot \Downarrow \cdot$), we need to prove in parallel a second property[2]:

$$\Sigma \vdash \| S \|^r \Downarrow^{\mathrm{SS}} \widetilde{a'} \wedge \widetilde{a'} \Downarrow_h^{\mathrm{EBS}} \widetilde{b} \implies \Sigma \vdash S \Downarrow^{\mathrm{EBS}} \widetilde{b}$$

We proceed by mutual induction on the derivations of $\widetilde{a} \Downarrow_h^{\mathrm{SS}} \widetilde{a'}$ and $\Sigma \vdash \| S \|^r \Downarrow^{\mathrm{SS}} \widetilde{a'}$. The main difficulty is to show that changing constructor in small-step corresponds to an EBS derivation where such a change does not occur. For this, we use Lemma 5.1.2 presented in Section 5.1.2. We discuss the issue in more details in the next section for the reverse implication, where we encounter the same difficulty.

For infinite behaviors, we prove the following theorem (C.7.4).

Theorem 5.1.8 (Div: SS⇒EBS)**.** For all hook h and terms $\widetilde{a}$,

$$\widetilde{a} \overset{\infty}{\to}_h \implies \widetilde{a} \Uparrow_h^{\mathrm{EBS}}$$

The concatenation lemma cannot be used here as we no longer have a base case to start its iteration. Instead, we show that $\widetilde{a} \overset{\infty}{\to}_h$ satisfies the big-step coinductive rules of Figure 3.3 by splitting the infinite sequence of small steps into subderivations.

Such a subderivation may occur when evaluating a constructor $c(\widetilde{x}, \widetilde{y})$ with a rule ($\texttt{let}\ \widetilde{v} = \texttt{Hook Reuse}\ h_1\ \widetilde{x}\ \texttt{in}\ S$) in $\mathrm{R}_{\mathrm{EBS}}$ (such rules are present for new constructors, see Example 4.2.2). We distinguish two cases: either the small-step divergence occurs when evaluating $\widetilde{x}$, which correspond to the hook call h_1, or when evaluating S. In the first case, by coinduction, we have a diverging EBS derivation for that skelement, and therefore the complete LetIn structure

2. The actual property has a couple additional hypothesis, stating notably that S is part of the semantics and thus has gone through the generation phase, see Lemma C.7.1

diverges. Otherwise, the evaluation of $\widetilde{x}$ terminates, and S diverges. Using Theorem 5.1.6 above for the finite case, we get a finite EBS derivation for $\widetilde{x}$, and by coinduction, a diverging EBS derivation for S, and the complete LetIn structures diverges as well.

5.1.5 Extended Big-Step implies Small-Step

Lastly, we prove we can transform a big-step evaluation into a sequence of small-step reductions (Theorem C.6.1).

Theorem 5.1.9 (EBS⇒SS)**.** For all hook h and terms $\widetilde{a}$ and $\widetilde{b}$,

$$\widetilde{a} \Downarrow_h^{\text{EBS}} \widetilde{b} \implies \exists \widetilde{a'},\ \widetilde{a}\ (\Downarrow_h^{\text{SS}})^*\ (\widetilde{a'}, \texttt{Ret\_h}(\widetilde{b}))$$

As before, we need to prove in parallel a similar result for skeletons[3]

$$\Sigma \vdash S \Downarrow^{\text{EBS}} \widetilde{b} \implies \exists \widetilde{a}, \widetilde{a'}.\ \Sigma \vdash \parallel S \parallel^r \Downarrow^{\text{SS}} \widetilde{a} \ \wedge\ \widetilde{a}\ (\Downarrow_h^{\text{SS}})^*\ (\widetilde{a'}, \texttt{Ret\_h}(\widetilde{b}))$$

and proceed by mutual induction on the derivations of $\widetilde{a} \Downarrow_h^{\text{EBS}} \widetilde{b}$ and $\Sigma \vdash S \Downarrow^{\text{EBS}} \widetilde{b}$. The proof is quite long because of the large number of cases.

The main difficulty is in changing constructors when needed. Indeed the EBS semantics does not make use of the newly defined constructors in the evaluation of the original constructors, unlike the small-step semantics, which for instance switches from `While` to `While1`. When we have $\Sigma \vdash S_{\texttt{While}} \Downarrow^{\text{EBS}} \widetilde{b}$, we need to argue that the skeleton for `While1` is almost the same as the one for `While` to be able to use the hypotheses we have at this point. This is where Lemma 5.1.2 becomes crucial: the first hook call in the rule for `While` is annotated with (`New While1`), and thus, by construction, we know the rule for `While1` uses (almost) the same skeleton and behaves similarly.

For infinite behaviors, we state the result as follows (Theorem C.6.4).

Theorem 5.1.10 (Div: EBS⇒SS)**.** For all hook h and terms $\widetilde{a}$,

$$\widetilde{a} \Uparrow_h^{\text{EBS}} \implies \widetilde{a} \xrightarrow{\infty}_h$$

By coinduction on the definition of $\widetilde{a} \xrightarrow{\infty}_h$, the proof needs to show that $\widetilde{a} \Uparrow_h^{\text{EBS}}$ satisfies the corresponding property (Lemma C.6.3).

Lemma 5.1.11. For all hook h and terms $\widetilde{a}$,

$$\widetilde{a} \Uparrow_h^{\text{EBS}} \implies \exists \widetilde{b},\ \widetilde{a} \Downarrow_h^{\text{SS}} \widetilde{b} \ \wedge\ \widetilde{b} \Uparrow_h^{\text{EBS}}$$

3. Once again, there is a couple additional hypothesis omitted here stating that S is a legitimate skeleton of the semantics, so we can leverage its properties (see Theorem C.6.1).

This result can be seen as the reverse of a concatenation lemma, i.e., a big-step divergence can be split into a small-step reduction and another big-step divergence. We proceed by induction on $\widetilde{a}$, and by case analysis on the definition of $\widetilde{a} \Uparrow_h^{\mathrm{EBS}}$. We conclude in each case with a proof very similar to the corresponding case in the proof of Theorem 5.1.9 for finite evaluations.

5.2 Coq Proof Script Generation

The paper proof of Section 5.1 comforts us in the correctness of (most of) the transformation. However, a mechanized proof is still valuable, as it protects us from human error in both the proof and the implementation.

A complete formalization of the transformation in a proof assistant seems out of reach because of its many intermediary steps. Instead, we follow the easier route of providing an a-posteriori certification alongside the resulting semantics. To this end, we rely on the possibility offered by Necro to export a skeletal semantics into a Coq representation. We automatically generate proof scripts showing that the produced small-step semantics corresponds to the initial big-step one. We explain how to generate the scripts using IMP as an example; the repository [8] contains examples of Coq scripts for other languages as well as the script generator.

5.2.1 Proof Sketch

Just like the paper proof of Section 5.1, we rely on the concrete interpretation judgment of Section 3.3 (see Figure 3.2).

In the case of IMP, the equivalence theorem between the semantics is as follows.

Theorem 5.2.1. For all $(s, s_0 : \texttt{state})$, $(e : \texttt{expr})$, $(t : \texttt{stmt})$, and $(v : \texttt{value})$,

$$(s, e) \Downarrow^{\mathrm{BS}}_{\texttt{hexpr}} (s_0, v) \iff \exists s',\ (s, e)\ (\Downarrow^{\mathrm{SS}}_{\texttt{hexpr}})^*\ (s', \texttt{Ret\_hexpr}(s_0, v))$$
$$(s, t) \Downarrow^{\mathrm{BS}}_{\texttt{hstmt}} s_0 \iff \exists s',\ (s, t)\ (\Downarrow^{\mathrm{SS}}_{\texttt{hstmt}})^*\ (s', \texttt{Ret\_hstmt}(s_0))$$

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

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

The challenging direction is to transform a sequence of small-step reductions into a big-step evaluation. The textbook proof method [40] relies on auxiliary lemmas to split the sequences of reductions in order to recreate big-step derivation trees, such as the following result for `Plus`.

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

This technique depends a lot on the semantics of the language, as each lemma must be derived from the constructor skeleton. We instead reuse the strategy of Section 5.1.4, with a concatenation lemma stating that we can merge a small step into a big step.

$$e \underset{\text{SS}}{\to} e' \Downarrow^{\text{BS}} v \quad \Longrightarrow \quad e \Downarrow^{\text{BS}} v$$

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

Since the concatenation lemma requires a common syntax, we generate the extended big-step semantics (EBS) we used in the paper proof (Section 5.1.1), defined on all constructors. The three semantics we need—the user-defined BS semantics, the EBS semantics generated in the middle of the transformation, and the resulting SS semantics—are transformed into Coq definitions using the export function of Necro. Necro also provides a Coq definition of the concrete interpretation, which itself depends on interpretations for the base 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 proof strategy is therefore similar to the paper proof: to show that BS and EBS are equivalent on initial terms, and then prove that EBS is equivalent to SS on all terms (including the extended ones). In the end, we get that BS is equivalent to SS on initial terms, as stated in Theorem 5.2.1.

5.2.2 Initial and Extended Big-Step

The intermediate EBS semantics extends the initial BS one on all constructors. It contains the (mostly unchanged) user-defined rules for the initial constructors (e.g. `Plus`), and the rules generated during the processing phase of Section 4.2.2 for the added constructors; see for instance the rules generated for `While1` and `While2` in Section 4.1.2. Finally, we add a rule for each return constructor to extract its resulting values, such as $\texttt{Ret}(v) \Downarrow^{\text{EBS}} v$. In the examples, we prefix the types of the extended semantics with a letter `e`, writing for instance `estate` for extended states, and we write $|t|$ for the the canonical injection of an initial term t into the extended semantics. The complete EBS generated for IMP can be found in Appendix B.5.

The first step of the certification checks that EBS and BS agree on the initial terms. It is easy to verify since EBS is a conservative extension of BS: we simply match every behavior—every applied rule—of each big-step semantics with exactly the same one on the other side. The distribution of branchings (Section 4.2.3) might reorganize skeletons a bit, but following a path would give out the same hypothesis, so Coq has no problem matching the two behaviors. The only difficulty is the manipulation of coercions back and forth between initial and extended

terms, but a few general lemmas are enough to automate the rewriting.

Theorem 5.2.2 (BS⇒EBS)**.** For all $(s, s_0 : \texttt{state})$, $(e : \texttt{expr})$, $(t : \texttt{stmt})$, and $(v : \texttt{value})$,

$$(s, e) \Downarrow^{\text{BS}}_{\texttt{hexpr}} (s_0, v) \implies (|s|, |e|) \Downarrow^{\text{EBS}}_{\texttt{hexpr}} (|s_0|, |v|)$$
$$(s, t) \Downarrow^{\text{BS}}_{\texttt{hstmt}} s_0 \implies (|s|, |t|) \Downarrow^{\text{EBS}}_{\texttt{hstmt}} |s_0|$$

Theorem 5.2.3 (EBS⇒BS)**.** For all $(s : \texttt{state})$, $(e : \texttt{expr})$, $(t : \texttt{stmt})$, $(s'_0 : \texttt{estate})$, and $(v' : \texttt{evalue})$,

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

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

5.2.3 Small-Step Implies Extended Big-Step

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

$$\texttt{Plus}(e_1, e_2) \underset{\text{SS}}{\to} \texttt{Plus}(e'_1, e_2) \Downarrow^{\text{EBS}} v \text{ from } e_1 \underset{\text{SS}}{\to} e'_1, e'_1 \Downarrow^{\text{EBS}} v_1, e_2 \Downarrow^{\text{EBS}} v_2, \text{ and } v = v_1 + v_2,$$

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

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

Lemma 5.2.4 (Concatenation)**.** For all $(s, s', s_0 : \texttt{estate})$, $(e, e' : \texttt{eexpr})$, $(t, t' : \texttt{estmt})$, $(v : \texttt{evalue})$,

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

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

Theorem 5.2.5 (SS⇒EBS)**.** For all $(s, s', s_0 : \texttt{estate})$, $(e : \texttt{eexpr})$, $(t : \texttt{estmt})$, and $(v : \texttt{evalue})$,

$$(s,e)\ (\Downarrow^{\text{SS}}_{\texttt{hexpr}})^*\ (s', \texttt{Ret\_hexpr}(s_0,v)) \implies (s,e)\ \Downarrow^{\text{EBS}}_{\texttt{hexpr}}\ (s_0,v)$$
$$(s,t)\ (\Downarrow^{\text{SS}}_{\texttt{hstmt}})^*\ (s', \texttt{Ret\_hstmt}(s_0)) \implies (s,t)\ \Downarrow^{\text{EBS}}_{\texttt{hstmt}}\ s_0$$

5.2.4 Extended Big-Step Implies Small-Step

Proving that a big-step evaluation of an extended term corresponds to a small-step sequence of reduction is done by induction on the derivation of the big-step evaluation. In each case, we need to build a complete sequence of small steps from the partial sequences we get from using the induction hypothesis. For instance, decomposing the evaluation $\texttt{Plus}(e_1, e_2) \Downarrow^{\text{EBS}} v$ 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 $\texttt{Plus}(e_1, e_2) \underset{\text{SS}}{\rightarrow}^* \texttt{Ret}(v)$.

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

Instead, we help Coq by generating congruence results about the small-step reduction. With them, we still might need backtracking to find the right combination of lemmas to apply, but all possible small-step reduction cases have been defined so we are not repeatedly losing exploration

time to regenerate them in each search. In the case of `Plus`, we need lemmas of the form:

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

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

Each lemma corresponds to a path in a small-step rule. In the case of `Plus` (whose skeleton is detailed in Example 4.2.5), following the different branches gives us three different paths:

```
- let (z1, z2) = hexpr (s, e1) in (z1, Plus (z2, e2))
- let (s1, v1) = getRet_hexpr (e1) in
  let (z3, z4) = hexpr (s1, e2) in (s, Plus (Ret_hexpr (z3, v1), z4))
- let (s1, v1) = getRet_hexpr (e1) in let (s2, v2) = getRet_hexpr (e2) in
  let v = add (v1, v2) in (s, Ret_hexpr (s2, v))
```

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

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

Theorem 5.2.6 (EBS⇒SS)**.** For all $(s, s_0 : \texttt{estate})$, $(e : \texttt{eexpr})$, $(t : \texttt{estmt})$, and $(v : \texttt{evalue})$,

$$\begin{aligned} (s, e) \Downarrow^{\text{EBS}}_{\texttt{hexpr}} (s_0, v) &\implies \exists s',\ (s, e)\ (\Downarrow^{\text{SS}}_{\texttt{hexpr}})^*\ (s', \texttt{Ret\_hexpr}(s_0, v)) \\ (s, t) \Downarrow^{\text{EBS}}_{\texttt{hstmt}} s_0 &\implies \exists s',\ (s, t)\ (\Downarrow^{\text{SS}}_{\texttt{hstmt}})^*\ (s', \texttt{Ret\_hstmt}(s_0)) \end{aligned}$$

Chapter 6

Implementation and Evaluation

The transformation and the self-certification have been implemented in Necro [21], with a number of options (Section 6.1.1) to tailor the transformation to specific needs. Using the existing Necro tools, we can generate a Coq version of the small-step semantics to prove properties on it—on top of the equivalence with the big-step one already automatically proved as presented in Section 5.2. We also generate an OCaml interpreter for the language allowing for small-step and big-step reductions (Section 6.1.2). In practice, we tested our transformation and automatic certification on a variety of user languages, with and without reuse. Section 6.2 presents our empirical results.

6.1 Implementation

6.1.1 Options and Optimization

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

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

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

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

No Reuse. Our transformation tries to reuse the user defined constructors as much as possible in order to reduce the number of additional constructors. If need be, an option forces the creation of new constructors for each hook call. This would lead to more terms, but also to

significantly simpler skeletons as every constructor would focus on a specific hook call. Depending on the purpose of the small-step semantics, it might be an interesting trade-off.

No Additional Steps. The transformation presented above makes additional administrative steps when changing constructor or refocusing, as it is common on paper to reduce the number of inference rules of the language. However, this is not necessary, and an option forces the transformation to keep tail-calls and perform a recursive call after changing constructor. This leads for instance to skeletons mimicking the rules:

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

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

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

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

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

However, this check is not sufficient for all cases, and `While2` still has 4 arguments as it evaluates `s1` which is not the main state. To recover the usual hand-written semantics, we would need to overwrite `s` with the content of `s1`. We are not certain overwriting states is always preferable, as it might conflict with other potential optimizations such as detecting read-only states (see Section 6.2).

6.1.2 Ocaml Interpreter

As presented in Section 3.5, Necro provides a tool to automatically generate an OCaml interpreter from a skeletal semantics. Using the transformation of this paper as an intermediate step, we are able to create an interpreter providing both big-step and small-step implementations

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

6.2 Evaluation

The transformation can be applied to any semantics written in skeletal semantics. The theorems of Chapter 5 hold even if the input semantics has an unusual shape. However, the result 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 [8]. In this section, we limit the evaluation to the transformation of big-step semantics.

We compare the sizes of the generated small-step semantics and equivalence proof scripts for various languages in Table 6.1. The examples include variants of the call-by-value (CBV) and call-by-name (CBN) λ-calculus implemented with closures, and evaluated in an environment mapping variables to closures; CBV has also been extended with non-determinism and exceptions. The examples written in an imperative style include a small language roughly corresponding to IMP expressions (Arithmetic), and extensions of IMP with local (IMP, LetIn) or global (IMP, write) state modification, and with exceptions and handlers (IMP, try/catch). Lastly, miniML is an ML-like language, extending the λ-calculus with arithmetic and boolean operations as well as constructs to define algebraic datatypes and perform pattern-matching on them. The generated small-step semantics and proofs for all the examples can be found in the source repository [8].

In the certification, about 500 lines of code are completely independent of the input language and contain definitions of skeletons or concrete interpretation. About 450 lines of code are templates which are filled with basic information about the syntax of the input semantics (the names of the hooks, constructors, and filters): these are general definitions, results, and tactics to manipulate concrete interpretation or coercions. They are part of the generated proof script, but are not counted in Table 6.1, where we lists the sizes of the language-dependent parts of the proof.

We see that the resulting small-step semantics are significantly longer than the initial big-step ones. This is because recursive computations are replaced with case disjunctions using branchings, quickly increasing the number of lines of the rules but not their complexity, as we can see with IMP. We also observe that the language-dependent part of the Coq proof scripts remains linear in the size of the small-step semantics. Indeed, it is composed mostly of the

Language	Lines of Code			Constructors		
	Big-Step	Small-Step	Coq	Big-Step	Small-Step	No Reuse
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 6.1: Size of the Generated Semantics and Proof Scripts

generated lemmas of the EBS implies SS part of the proof, which themselves depend on the number of different paths in the small-step rules, as explained in Section 5.2.4.

W.r.t. to the number of constructors, we see the impact of reusing them, especially for IMP where we go from two new constructors (`While1` and `While2`) with reuse to ten without.

The effectiveness of reusing constructors depends on how the input skeletal semantics is written. First, it depends on the ordering of the skelements in the initial rules. The process is more efficient when hook calls are grouped at the beginning of skeletons, since we do not reuse constructors after filter calls. For instance, the two following equivalent rules lead to respectively one and two new constructors.

```
hook eval (s : env, t : lambdaterm) matching t : clos =
| App (t1, t2) ->
  let c1 = eval (s, t1) in
  let c2 = eval (s, t2) in
  let (x, t3, s1) = getClos (c1) in
  let s2 = extEnv (s1, x, c2) in
  eval (s2, t3)
```

```
| App (t1, t2) ->
  let c1 = eval (s, t1) in
  let (x, t3, s1) = getClos (c1) in
  let c2 = eval (s, t2) in
  let s2 = extEnv (s1, x, c2) in
  eval (s2, t3)
```

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

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

The transformation would compute that the state `s` is used twice and thus create a new constructor with a copy of it to evaluate `e1`. A small-step operational semantics written by hand would rely on the fact that `hexpr` does not modify the state to avoid introducing a new constructor in that case. Such an analysis requires a global understanding of the semantics which goes beyond the local study of hook calls our transformation is based on. This discrepancy occurs in the miniML language, for which we create 10 new constructors (with reuse, cf. Table 6.1), when only 9 are strictly necessary.

Chapter 7

Conclusion of Part I

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. This allows users to benefit from the convenience of big-step definitions, while having access to the fine control of small-step semantics when necessary.

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. Equivalence proofs between big-step and small-step semantics are well-known [40, 46, 35] but repetitive for large languages, so they benefit greatly from the automation and trust of proof assistants. The transformation without reuse is also 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.

In practice, this automated transformation and certification have been successfully tested on a wide variety of languages. The transformation with reuse produces small-step skeletal semantics close to the SOS rules that one would write by hand. The current implementation still has some limitations discussed below (see Section 7.2), but we believe the core idea of the transformation can be expanded to handle these cases.

7.1 Related Work

While several approaches go from a small-step to a big-step setting by manipulating either inference rules [45, 20] or interpreters [26, 23], the other direction has been less pursued.

Vesely and Fisher [53] 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 hook calls in our case—and ignore other simple functions calls—filter calls—or focus changes. The input interpreter, defined in an ad-hoc language, may not be as expressive as skeletal semantics, in particular because only one evaluation function is possible. It is not clear whether 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 $\texttt{App}(t_1, t_2)$ with a subreduction $s, t_1 \to s', t_1'$, we would reconstruct a configuration as $s, \texttt{App}(t_1, t_2) \to s, \texttt{App1}(s', t_1', t_2)$ while they would reconstruct a term as $s \vdash \texttt{App}(t_1, t_2) \to \texttt{App}(C(s', t_1'), t_2)$. As a result, it is difficult to compare the outputs of the two approaches based on the number of additional constructors or rules, but the output semantics are very close to usual small-step definitions, with a minimal number of created constructors in both cases.

The strategy for simplifying the output semantics is also quite different. Our transformation performs an analysis first to avoid generating and using useless constructors, while they transform every `eval` call into a new constructors and rely on an a posteriori analysis to factorize constructors that behave the same. The analysis they present is lightweight and cannot regroup terms using other variables (e.g., states) differently. This does not seem as general as our approach, but it could probably be solved by embedding a stronger analysis if needed.

Finally, Vesely and Fisher [53] 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. [31] 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.

7.2 Limitations and Perspectives

Our current small-step transformation has a few limitations, and cannot be applied to every language expressible in the recent higher-order skeletal semantics. In this section, we informally present the two problematic cases, and discuss potential solutions.

7.2.1 Polymorphism

The version of skeletal semantics used in this first part supports polymorphism, even if we did not detail it in Chapter 3. However, our transformation do not apply to polymorphic hooks, as they present additional challenges. As an example, consider the following function.

```
type list<a> =
| Nil
| Cons of a * list<a>

(** truncates a list 'l' and keeps the 'n' first elements *)
hook firstn<a> (l:list<a>, n:nat) matching n : list<a> =
| Zero -> Nil<a>
| Succ (m) ->
  branch
    let Cons(x, l2) = l in
    let q = firstn l2 m in
    Cons(x, q)
  or
    let Nil = l in Nil<a>
  end
```

If we consider this hook to be a basic operation to be evaluated atomically (i.e., keep it in big-step form), our implementation can transform the rest of the semantics without issue[1]. If we instead want a small-step version of this operation, the transformation of Chapter 4 needs to be extended.

If all the polymorphic variables of the hook (here, `a`) are included in the polymorphic variables of the matched program type, our transformation should be applicable with very little modifications. A coercion can be added from the output type to the matched type. Intermediate program points can also be transformed into new constructors for the program type.

However, if the matched program type is not "polymorphic enough", then the procedure might fail. In the example above, the type `nat` is not polymorphic, so we cannot create a coercion

1. However, the legacy Coq export mechanism does not support polymorphic types, so we cannot generate an equivalence certificate.

from `list<a>` to `nat`[2]. A relatively simple solution would be to create a new (polymorphic) configuration program type with a single constructor, and modify the hook to take a single argument. In our example above, it would generate something along the lines of:

```
type config<a> =
| Conf of list<a> * nat

hook firstn<a> (c : config<a>) matching c : list<a> =
| Conf (l, n) ->
  branch
    let Zero = n in Nil<a>
  or
    let Succ m = n in
    branch
      let Cons(x, l2) = l in
      let q = firstn Conf(l2, m) in
      Cons(x, q)
    or
      let Nil = l in Nil<a>
    end
  end
```

Then, the transformation of Chapter 4 could apply. However, this approach has two limitations. First, it requires a new type and the modification of every call to the polymorphic function. Second, since hook calls to the polymorphic hook now explicitly use the new constructor (e.g., `firstn Conf(l2, m)`), partial results cannot be stored and the transformation systematically requires new constructors (see analysis of Section 4.2.2). User-provided constructors cannot be reused, which complicates the resulting small-step semantics.

7.2.2 Anonymous Functions

The legacy skeletal semantics presented in this first part does not allow the definition of anonymous functions. However, the recent version used in the second part of this document does (see Chapter 8). This creates new problematic cases, as it is unclear how to transform anonymous functions that might call upon evaluation functions.

A first solution would be to precede our transformation with a phase of defunctionalization (see Chapter 9), i.e., transform every anonymous function into object types before applying the strategy of Chapter 4. While it is a correct approach, semantics making heavy use of higher-order functions and anonymous functions would be heavily modified, and the resulting small-step semantics might be hard to follow.

2. Of course, in this toy example, the code could be re-written to pattern-match on the list instead of the integer. This would solve the issue. But in general, this problem cannot always be side-stepped.

Notably, the recent skeletal semantics supports the use of monads, allowing users to factorize common data manipulations. Since monads make use of anonymous functions, reshaping them properly becomes crucial. For instance, consider the following example using the syntax of the recent skeletal semantics (see Chapter 8).

```
type m<a> = ...
val ret<a> (v : a) : m<a> = ...
val bind<a,b> (w : m<a>) (f : a → m<b>) : m<b> = ...
binder @ := bind

type value =
  | Int int
  | ...
type term =
  | Plus (term, term)
  | ...

val add : int -> int -> int

val eval (t : term) : m<value> =
branch
  let Plus (t1, t2) = t in
  let Int n1 =@ eval t1 in
  let Int n2 =@ eval t2 in
  let n = add n1 n2 in
  ret<value> (Int n)
or ... end
```

Here, `let p =@ eval t in s` is syntactic sugar for `let temp = eval t in bind temp (λp.s)`.

Evaluating a term `Plus`(t_1, t_2) still consists into evaluating t_1 and t_2, recovering two integers, and adding them. The monad type `m` can hide a number of data structures and additional operations. It can for instance contain a state to evaluate variables, or be able to raise an exception if something goes wrong (e.g., division by zero). Eliminating anonymous functions before applying the small-step transformation would completely reshape the semantics and make the structure of the evaluation of `Plus`(t_1, t_2) difficult to follow.

A general strategy to transform arbitrary anonymous functions to small-step seems out of reach, however monads use them in a very controlled way, so a strategy with some restrictions might be feasible.

If we try to create, by hand, an equivalent small-step reduction function of type `term -> term`, we quickly notice that extracting the result from the monad type is not always possible, depending on the type of monad we want. However, an interesting opportunity appears if we preserve the monad around the small-step output type. For the example above, we could write

the following small-step function, independently from the choice of monad.

```
type term =
  | Plus (term, term)
  | ...
  | Value value

val evalss (t : term) : m<term> =
branch
  let Plus (t1, t2) = t in
  branch
    let t1' =@ evalss t1 in
    ret<term> (Plus(t1', t2))
  or
    let Value (Int n1) = t1 in
    branch
      let t2' =@ evalss t2 in
      ret<term> (Plus(Value (Int n1), t2'))
    or
      let Value (Int n2) = t2 in
      let n = add n1 n2 in
      ret<term> (Value (Int n))
    end
  end
or ... end
```

We recover a similar structure to the small-step evaluation of IMP in Example 4.2.5. If t_1 can be computed further, we do so and reconstruct `Plus`(t_1', t_2), and inject it into the monad type. Otherwise t_1 is an integer, and we can proceed similarly with t_2. Note that the monadic bind on t_1 can potentially shortcut the evaluation and return something else, just like it is possible in the big-step version of this function.

The main advantages are that we preserve the structure of the semantics while introducing a stepping function that should be equivalent to the initial evaluation function. However, the resulting small-step semantics is very unconventional, as the output type is not the same as the input type. We cannot chain the reduction by repeatedly applying `evalss`, but instead we should integrate the initial term in the monad and repeatedly apply (`λw. bind w evalss`).

$$t \xrightarrow{\texttt{ret}} w_0 \xrightarrow{\texttt{bind \_ evalss}} w_1 \xrightarrow{\texttt{bind \_ evalss}} w_2 \xrightarrow{\texttt{bind \_ evalss}} \dots$$

Assessing the correctness and feasibility of this approach is left as a future work.

Part II

Meta-Language Transformation

Chapter 8

Higher-Order Skeletal Semantics

The legacy skeletal semantics, used in Part I of this document, has two shortcomings that make expressing complex semantics tedious. They are resolved with the recent Skel meta-language.

First, associating a behavior to a constructor can only be done when defining a top-level evaluation function. For behaviors depending on the presence of several constructors, the user has to define auxiliary functions, resulting in tangled and unclear formalizations. For instance, the transformation of Section 4 needs to create auxiliary functions named `getRet` because there is no direct way to associate a behavior to specific terms such as $\texttt{Plus}(\texttt{Ret}(v_1), e_2)$. This issue also applies to functions that need to pattern-match on several arguments.

The problem is solved by strengthening the LetIn structure. We allow pattern recognition on the result of the first computation. For instance, the extraction of a value previously noted `let v1 = getRet e1 in ...` can now be performed with a skeleton `let Ret(v1) = e1 in ...`, without the need for an auxiliary function. If the term `e1` uses the constructor `Ret`, then the variable `v1` is bound as expected and the computation continues; else, there is no behavior and the evaluation of the skeleton fails.

Combining this pattern recognition with branchings, it is possible to simulate a usual pattern-matching. As such, we no longer enforce a mandatory pattern-matching when defining an evaluation function. For instance, Figure 8.1 contains a semantics for Call-by-Value λ-calculus using the recent Skel meta-language and Necro syntax. The skeleton for the `eval` function starts with a branching, and each branch enforces a specific constructor, allowing us to associate a behavior to each constructor.

Second, the legacy skeletal semantics does not support anonymous functions. There is no function type, and it is not possible to bind a variable to a function. While anonymous functions are not necessary to transcribe usual inference rules for small programming languages, they are frequently used in interpreters and complex languages. The legacy skeletal semantics is notably unpractical for languages making use of monads, such as JavaScript.

The recent skeletal semantics allows for function types and anonymous functions of the form $(\lambda \texttt{p} \rightarrow \texttt{s})$, where the input `p` is a pattern that may enforce a specific constructor, and `s` is a skeleton. This also enables the formalization of higher-order evaluation functions. For instance, we can define the standard `head` and `map` operations on lists as follows:

```
type ident

type lterm =
| Lam (ident, lterm)
| Var ident
| App (lterm, lterm)
type clos =
| Clos (ident, lterm, env)

type env

val extEnv : (env, ident, clos) → env
val getEnv : (ident, env) → clos
```

```
val eval (s : env) (l : lterm) : clos =
branch
  let Lam (x, t) = l in
  Clos (x, t, s)
or
  let Var x = l in
  getEnv (x, s)
or
  let App (t1, t2) = l in
  let Clos (x, t, s1) = eval s t1 in
  let w = eval s t2 in
  let s2 = extEnv (s1, x, w) in
  eval s2 t
end
```

Figure 8.1: Skeletal Semantics for Call-by-Value λ-calculus

```
type list<a> =
| Nil
| Cons (a, list<a>)

val head<a> : (list<a> → a) =
  λ Cons(x, _) → x

val map<a,b> (f : a → b) (l : list<a>) : list<b> =
branch
  let Nil = l in Nil<b>
or
  let Cons(x1, l1) = l in
  let x2 = f x1 in
  let l2 = map f l1 in
  Cons<b> (x2, l2)
end
```

The recent skeletal semantics also include several quality-of-life changes, including the followings. Skelements are merged with skeletons, allowing skeletons of the form `let p = s1 in s2`. Record types are supported. Filters and hooks are renamed to *unspecified* and *specified* evaluation functions, simplifying the notion of applying a function. Skeletons allow for existential quantification.

The Necro toolbox has also been expanded. Notably, it provides syntactic sugar for a convenient usage of monads, and it allows the import of other definition files. The most important feature is still the Coq extraction mechanism. Languages defined in skeletal semantics can automatically be exported into a Coq definition file. We remind that the tool provides a deep embedding, i.e., it generates a Coq data structure encompassing the definitions of types and

evaluation functions of the language. Combined with the formalized deterministic abstract machine of Chapter 12, we can use the OCaml extraction provided by Coq to obtain a certified OCaml interpreter for any user-defined language.

8.1 Syntax

We now introduce the new syntax of skeletal semantics. We write [] for an empty list and $(a :: l)$ for adding an element a to a list l. We note $[a_1; \ldots; a_n]$ for the list $(a_1 :: \ldots :: a_n :: [])$. We write $l_1 +\!\!+ l_2$ the concatenation of two lists, $[a_1; \ldots; a_n] +\!\!+ [b_1; \ldots; b_m] \triangleq [a_1; \ldots; a_n; b_1; \ldots; b_m]$.

The skeletal semantics of a language is composed of:

- a set of unspecified types;
- a set of specified types, with either typed constructors (for abstract datatypes) or typed field names (for record types);
- a set of typed unspecified skelterms;
- a set of typed specified skelterms.

We use the word "skelterm" for terms of the Skel meta-language to avoid confusion with terms of the embedded language (e.g., λ-terms). In the example of Figure 8.1, `ident` is an unspecified type; `lterm` a specified type with three constructors, `getEnv` is an unspecified skelterm, and `eval` a specified skelterm.

An *unspecified skelterm* is an identifier representing an abstract object/function not given when defining the language. To execute the language in practice, we would need to determine how to interpret this identifier (see Section 8.2). In the Necro syntax (see Figure 8.1), they are introduced with the keyword `val`.

We call *specified skelterms* identifiers associated to an explicit skelterm. We note `SpecDecl` the mapping from identifiers (i.e., strings) to their corresponding skelterm. They are also introduced with the keyword `val`, but require a definition.

We let c range over constructors; n and m over natural number; and x and d over identifiers (i.e., strings). For any entity e, we note l_e for lists of elements of this entity: for instance, l_c represents a list of constructors. The grammar of types (T), variables (v), patterns (p), skelterms (t), and skeletons (S) is defined as follows.

$$
\begin{aligned}
T &::= \texttt{TyVar}(x) \mid \texttt{TyBase}(x, l_T) \mid \texttt{TyProd}(l_T) \mid \texttt{TyArrow}(T_1, T_2) \\
v &::= \texttt{VLet}(x, T) \mid \texttt{VUnspec}(x, T, l_T) \mid \texttt{VSpec}(x, T, l_T) \\
p &::= \texttt{PWild}(T) \mid \texttt{PVar}(x, T) \mid \texttt{PConstr}(c, l_T, p) \mid \texttt{PTuple}(l_p) \\
&\quad \mid \texttt{PRec}([(d_1, l_{T_1}, p_1); \ldots; (d_m, l_{T_m}, p_m)]) \\
t &::= \texttt{TVar}(v) \mid \texttt{TConstr}(c, l_T, t) \mid \texttt{TTuple}(l_t) \mid \texttt{TNth}(t, l_T, n) \mid \texttt{TFunc}(p, S) \\
&\quad \mid \texttt{TRec}(t_{\texttt{opt}}, [(d_1, l_{T_1}, t_1); \ldots; (d_m, l_{T_m}, t_m)]) \mid \texttt{TField}(t, l_T, d) \\
t_{\texttt{opt}} &::= \texttt{Some}(t) \mid \texttt{None} \\
S &::= \texttt{Branching}(T, l_S) \mid \texttt{LetIn}(p, S_1, S_2) \mid \texttt{Apply}(t, l_t) \mid \texttt{Return}(t) \mid \texttt{Exists}(p, S, T)
\end{aligned}
$$

Just like the previous version, this new skeletal semantics is strongly and statically typed. Every piece of syntax is associated with a precise type. $\texttt{TyVar}(x)$ is a variable used to create polymorphic types. $\texttt{TyBase}(x, l_T)$ represents the basic types defined by the user, either specified or unspecified. The type of λ-terms in Figure 8.1 is $\texttt{TyBase}(\texttt{"lterm"}, [])$. The second argument is used for polymorphism; e.g., a polymorphic list would be typed as $\texttt{TyBase}(\texttt{"list"}, [\texttt{TyVar}(\texttt{"a"})])$, whereas a list of λ-terms would be typed $\texttt{TyBase}(\texttt{"list"}, [\texttt{TyBase}(\texttt{"lterm"}, [])])$. Then, $\texttt{TyProd}(l_T)$ is a product type corresponding to tuples. Notably, the unit type is $\texttt{TyProd}([])$, with the unit element being $\texttt{TTuple}([])$. Finally, $\texttt{TyArrow}(T_1, T_2)$ represents a function type taking an argument of type T_1 and returning a result of type T_2.

Constructors are associated with a triple indicating the polymorphic variables, the input types, and the name of the output base type. For instance in the example of Figure 8.1, the constructor `Lam` is associated with $([], \texttt{TyProd}([\texttt{TyBase}(\texttt{"ident"}, []); \texttt{TyBase}(\texttt{"lterm"}, [])]), \texttt{"lterm"})$, while `Cons` is mapped to $([\texttt{"a"}], \texttt{TyProd}([\texttt{TyVar}(\texttt{"a"}); \texttt{TyBase}(\texttt{"list"}, [\texttt{TyVar}(\texttt{"a"})])]), \texttt{"list"})$.

Skelterms are typed objects, usually representing completed computations. They are recursively composed of constructors, tuples, and records. Records are maps from datafields (d) to skelterms, and are implemented using lists. Base elements are either variables or functions. `TNth` can be used to extract part of a tuple based on positioning; `TField` to extract a field from a record (see Section 8.2). Lists of types are needed in some cases to instantiate polymorphism. Patterns are very similar to terms and used for pattern-matching, with a special constructor `PWild` to ignore part of a term.

A *skeleton* represents a computation to perform. It can either be a LetIn structure with pattern-matching ($\texttt{let}\ p = S_1\ \texttt{in}\ S_2$) chaining two computations, a non-deterministic choice among several computations ($\texttt{Branching}(T, l_S)$), an application of a function to a list of arguments ($\texttt{Apply}(t, l_t)$), or simply returning a skelterm. Branchings require an output type in case the list is empty. Skeletal semantics also allows for the existential quantification of a result we

do not know how to compute ($\texttt{Exists}(p, S, T)$). It can be interpreted as $\texttt{let}\ p = (\texttt{?} : T)\ \texttt{in}\ S$. It is used to transcribe programming languages where inference rules are not syntax-directed, such as transitivity rules for subtyping.

Skeletal typed variables v correspond to the different strings which may appear in a skelterm, pattern, or skeleton. A variable of the language is defined using $\texttt{PVar}(x, T)$ in a pattern and used with $\texttt{VLet}(x, T)$. A specified skelterm is referenced using $\texttt{VSpec}(x, T, l_T)$, an unspecified one using $\texttt{VUnspec}(x, T, l_T)$. Once again, lists of types are used to instantiate polymorphism.

As an example, the skelterm of the `eval` function of Figure 8.1 is of the following form—we only detail the second branch `let Var x = l in getEnv (x, s)`.

$$
\begin{aligned}
\texttt{SpecDecl}(\texttt{"eval"}) &= t \\
t &\triangleq \texttt{TFunc}(\texttt{PVar}(\texttt{"s"}, T_s), \texttt{Return}(\texttt{TFunc}(\texttt{PVar}(\texttt{"l"}, T_l), S))) \\
S &\triangleq \texttt{Branching}(T_c, [\ldots; \texttt{LetIn}(p, S_1, S_2); \ldots]) \\
p &\triangleq \texttt{PConstr}(\texttt{"Var"}, [], \texttt{PVar}(\texttt{"x"}, T_x)) \\
S_1 &\triangleq \texttt{Return}(\texttt{TVar}(\texttt{VLet}(\texttt{"l"}, T_l))) \\
S_2 &\triangleq \texttt{Apply}(\texttt{TVar}(\texttt{VUnspec}(\texttt{"getEnv"}, T_{\texttt{getEnv}}, [])), [t_{xs}]) \\
t_{xs} &\triangleq \texttt{TTuple}([\texttt{TVar}(\texttt{VLet}(\texttt{"x"}, T_x)); \texttt{TVar}(\texttt{VLet}(\texttt{"s"}, T_s))]) \\
T_s &\triangleq \texttt{TyBase}(\texttt{"env"}, []) \qquad T_l \triangleq \texttt{TyBase}(\texttt{"lterm"}, []) \\
T_c &\triangleq \texttt{TyBase}(\texttt{"clos"}, []) \qquad T_x \triangleq \texttt{TyBase}(\texttt{"ident"}, []) \\
T_{\texttt{getEnv}} &\triangleq \texttt{TyArrow}(\texttt{TyProd}([T_x; T_s]), T_c)
\end{aligned}
$$

8.2 Concrete Interpretation

We keep the name *concrete interpretation* for the standard big-step semantics of the meta-language Skel. In part I, it simply related input terms to (closed) output terms. Now that terms contain functions, the concrete interpretation can also output closures, i.e., a function paired with the environment it should be evaluated with. We create a new type for the results of this evaluation, called *concrete values*, or cvalues for short. The grammar of concrete values (r), and environments (Σ) is defined as follows.

$$
\begin{aligned}
r &::= \texttt{CVTuple}(l_r) \mid \texttt{CVConstr}(c, r) \mid \texttt{CVClos}(\Sigma, p, S) \mid \texttt{CVUnspec}(x, n, l_T, l_r) \\
&\quad \mid \texttt{CVRec}([(d_1, r_1); \ldots; (d_m, r_m)]) \mid \texttt{CVBase}(a) \\
\Sigma &::= [(x_1, r_1); \ldots; (x_m, r_m)]
\end{aligned}
$$

Environments are simply lists mapping identifiers to results. $\texttt{CVTuple}(l_r)$ represents a tuple of results, $\texttt{CVConstr}(c, r)$ a constructor packing another cvalue. $\texttt{CVClos}(\Sigma, p, S)$ is a clo-

sure, corresponding to the skelterm $\texttt{TFunc}(p, S)$ bundled with an evaluation environment Σ. $\texttt{CVUnspec}(x, n, l_T, l_r)$ represents a partially applied unspecified function x, with partial arguments l_r (see below for more details). $\texttt{CVRec}(l_{dr})$ is a record mapping fields to cvalues.

Once again, the argument l_T is used for polymorphism. From now on, we stop specifying the internal types of skeletal semantics, as they use a standard system and are not relevant to the work presented here. We also present the semantics of Skel ignoring types (see below and Figures 8.2 and 8.3), when the full version would have a few additional side-conditions and polymorphic instantiations.

Finally, $\texttt{CVBase}(a)$ is an injection from base values of unspecified types to cvalues. Indeed, the concrete interpretation assumes given an instantiation of unspecified types and unspecified skelterms. For each unspecified type, we expect a set representing its values (e.g., $\mathbb{N}$ for `"nat"`, or $\{\top; \bot\}$ for `"bool"`). In the grammar above, a represents an object in the interpretation of an unspecified type.

For each unspecified skelterm, we require an arity and a function producing a list of possible results. For notation purposes, we group these elements in two main auxiliary functions: `Arity` of type ($\texttt{string} \to \texttt{nat}$); and `UnspecDecl` of type ($\texttt{string} \to \texttt{cvalue list} \to \texttt{cvalue list}$). For an unspecified skelterm x, $\texttt{UnspecDecl}(x)$ takes as argument a list of size $\texttt{Arity}(x)$ and outputs a list of possible results.

Example 8.2.1. To simulate an If/Then/Else construction, we can provide unspecified skelterms `"isTrue"` and `"isFalse"` of type ($\texttt{"bool"} \to ()$) with the following definitions:

$$\begin{aligned} \texttt{Arity}(\texttt{"isTrue"}) &= 1 \\ \texttt{UnspecDecl}(\texttt{"isTrue"})[\texttt{CVBase}(\top)] &= [\texttt{CVTuple}([])] \\ \texttt{UnspecDecl}(\texttt{"isTrue"})[\texttt{CVBase}(\bot)] &= [] \end{aligned}$$

and similarly for `"isFalse"`. This shows that an empty list result can be interpreted as a failure. A conditional branching "If v then ... else ..." can be simulated with branchings, as with the legacy skeletal semantics (Section 3.1):

Example 8.2.2. A list of multiple results can also represent non-determinism. For instance, for choosing a number in an interval, we can use:

$$\begin{aligned} \texttt{Arity}(\texttt{"randInt"}) &= 2 \\ \texttt{UnspecDecl}(\texttt{"randInt"})[\texttt{CVBase}(5); \texttt{CVBase}(10)] &= [\texttt{CVBase}(5); \texttt{CVBase}(6); \ldots; \texttt{CVBase}(10)] \end{aligned}$$

The non-deterministic inductive inference rules of Figure 8.2 and 8.3 express how the different structures of skeletal semantics are evaluated. The important rules are those for the evaluation of skelterms ($\Sigma \vdash t \Downarrow_{\mathrm{t}} r$) and skeletons ($\Sigma \vdash S \Downarrow_{\mathrm{s}} r$) to concrete values.

Most of the rules are self-explanatory. For instance, evaluating a skeleton $\texttt{LetIn}(p, S_1, S_2)$ under Σ corresponds to evaluating S_1 under Σ to get r', performing a pattern-matching between p and r' to expand the environment to Σ', and finally evaluating S_2 under Σ'. Some premises might not be doable, for instance the pattern-matching between p and r' could fail. In this case, the rule does not apply and the skeleton cannot be evaluated.

For records, we usually simply evaluate each subterm. An optional record can be given to serve as a template, and it would then be extended with new fields, similarly to the OCaml notation `{rc with d1 = t1 ; ... ; dn = tn}`. Note that the new datafields and values are added to the beginning of the list, and thus overwrite previous definitions on the initial record.

Environment lookup ($\Sigma(x) = r$) corresponds to finding the first pair of the form (x, r) in Σ. Pattern-matching ($\Sigma + \{p \mapsto r\} \Downarrow_{\text{p}} \Sigma'$) corresponds to extending Σ into Σ' with pairs (x_i, r_i), and is only valid if p and r have (recursively) tuples of the same arity and use the same constructors. Patterns can also read fields from records.

The evaluation of unspecified skelterms is probably the least intuitive, as the concrete interpretation allows for partially applied unspecified functions. If we encounter an unspecified skelterm of arity 0, we immediately apply its interpretation. Otherwise, ($\texttt{Arity}(x) = n + 1$), we create a concrete value $\texttt{CVUnspec}(x, n, [])$, keeping track of the missing number of arguments $(n + 1)$ and the already given arguments (starting with an empty list).

The behavior of application for unspecified skelterms depends on the number of arguments. If there is none, we simplify as expected. Else we have $(m + 1)$ arguments. If we have enough new arguments ($m + 1 \geq n + 1$, simplified to $m \geq n$ in Figure 8.3) we apply the interpretation of the skelterm. Otherwise ($m < n$), we expand the list of partial arguments and diminish the counter of missing arguments.

A few rules of Figure 8.2 are non-deterministic, as we sometimes pick an element in a list. This happens when choosing a branch in a branching, which can be seen as an internal source of non-determinism; and when selecting a result from the interpretation of unspecified skelterms, which can be seen as external non-determinism. The use of the $\texttt{Exists}(p, S)$ constructor is also fundamentally non-deterministic, but a derivation for it can only be provided by hand.

Note that there is no rule for generating or consuming base values $\texttt{CVBase}(a)$ of unspecified types, as they can only be handled by unspecified skelterms.

Interpretation of Skelterms:

$$\frac{\Sigma(x) = r}{\Sigma \vdash \texttt{TVar}(\texttt{VLet}(x)) \Downarrow_\text{t} r} \qquad \frac{\texttt{Arity}(x) = 0 \qquad r \in \texttt{UnspecDecl}(x)[]}{\Sigma \vdash \texttt{TVar}(\texttt{VUnspec}(x)) \Downarrow_\text{t} r}$$

$$\frac{\texttt{Arity}(x) = n + 1}{\Sigma \vdash \texttt{TVar}(\texttt{VUnspec}(x)) \Downarrow_\text{t} \texttt{CVUnspec}(x, n, [])} \qquad \frac{\texttt{SpecDecl}(x) = t \qquad [] \vdash t \Downarrow_\text{t} r}{\Sigma \vdash \texttt{TVar}(\texttt{VSpec}(x)) \Downarrow_\text{t} r}$$

$$\frac{\Sigma \vdash t \Downarrow_\text{t} r}{\Sigma \vdash \texttt{TConstr}(c, t) \Downarrow_\text{t} \texttt{CVConstr}(c, r)} \qquad \frac{\forall i,\ \Sigma \vdash t_i \Downarrow_\text{t} r_i}{\Sigma \vdash \texttt{TTuple}([t_1; \ldots; t_n]) \Downarrow_\text{t} \texttt{CVTuple}([r_1; \ldots; r_n])}$$

$$\frac{\Sigma \vdash t \Downarrow_\text{t} \texttt{TTuple}([r_0; \ldots; r_m]) \qquad n \leq m}{\Sigma \vdash \texttt{TNth}(t, n) \Downarrow_\text{t} r_n} \qquad \frac{}{\Sigma \vdash \texttt{TFunc}(p, S) \Downarrow_\text{t} \texttt{CVClos}(\Sigma, p, S)}$$

$$\frac{\Sigma \vdash t \Downarrow_\text{t} \texttt{CVRec}(l_{dr}) \qquad l_{dr}(d) = r}{\Sigma \vdash \texttt{TField}(t, d) \Downarrow_\text{t} r}$$

$$\frac{\forall i,\ \Sigma \vdash t_i \Downarrow_\text{t} r_i}{\Sigma \vdash \texttt{TRec}(\texttt{None}, [(d_1, t_1); \ldots; (d_m, t_m)]) \Downarrow_\text{t} \texttt{CVRec}([(d_1, r_1); \ldots; (d_m, r_m)])}$$

$$\frac{\Sigma \vdash t \Downarrow_\text{t} \texttt{CVRec}(l_{dr}) \qquad \Sigma \vdash \texttt{TRec}(\texttt{None}, l'_{dt}) \Downarrow_\text{t} \texttt{CVRec}(l'_{dr})}{\Sigma \vdash \texttt{TRec}(\texttt{Some}(t), l'_{dt}) \Downarrow_\text{t} \texttt{CVRec}(l'_{dr} +\!\!+ l_{dr})}$$

Interpretation of Skeletons:

$$\frac{S \in l \qquad \Sigma \vdash S \Downarrow_\text{s} r}{\Sigma \vdash \texttt{Branching}(l) \Downarrow_\text{s} r} \qquad \frac{\Sigma \vdash t \Downarrow_\text{t} r}{\Sigma \vdash \texttt{Return}(t) \Downarrow_\text{s} r}$$

$$\frac{\Sigma \vdash t \Downarrow_\text{t} r_0 \qquad \forall i,\ \Sigma \vdash t_i \Downarrow_\text{t} r_i \qquad r_0\ \$\ [r_1; \ldots; r_n] \Downarrow_\text{a} r}{\Sigma \vdash \texttt{Apply}(t, [t_1; \ldots; t_n]) \Downarrow_\text{s} r}$$

$$\frac{\Sigma \vdash S_1 \Downarrow_\text{s} r' \qquad \Sigma + \{p \mapsto r'\} \Downarrow_\text{p} \Sigma' \qquad \Sigma' \vdash S_2 \Downarrow_\text{s} r}{\Sigma \vdash \texttt{LetIn}(p, S_1, S_2) \Downarrow_\text{s} r}$$

$$\frac{\Sigma + \{p \mapsto r'\} \Downarrow_\text{p} \Sigma' \qquad \Sigma' \vdash S \Downarrow_\text{s} r}{\Sigma \vdash \texttt{Exists}(p, S) \Downarrow_\text{s} r}\ \texttt{for any}\ r'$$

Figure 8.2: Concrete Interpretation (part 1)

Environment Lookup:

$$\frac{}{((x, r) :: \Sigma)(x) = r} \qquad \frac{x \neq y \qquad \Sigma(x) = r}{((y, r) :: \Sigma)(x) = r}$$

Interpretation of Application:

$$\frac{}{r \ \$ \ [] \Downarrow_{\mathrm{a}} r} \qquad \frac{\Sigma + \{p \mapsto r_0\} \Downarrow_{\mathrm{p}} \Sigma' \qquad \Sigma' \vdash S \Downarrow_{\mathrm{s}} r_1 \qquad r_1 \ \$ \ l_r \Downarrow_{\mathrm{a}} r}{\mathtt{CVClos}(\Sigma, p, S) \ \$ \ (r_0 :: l_r) \Downarrow_{\mathrm{a}} r}$$

$$\frac{m \geq n \qquad r' \in \mathtt{UnspecDecl}(x)(l +\!\!+ [r_0; \ldots; r_n]) \qquad r' \ \$ \ [r_{n+1}; \ldots; r_m] \Downarrow_{\mathrm{a}} r}{\mathtt{CVUnspec}(x, n, l) \ \$ \ [r_0; \ldots; r_m] \Downarrow_{\mathrm{a}} r}$$

$$\frac{m < n \qquad n' = n - (m + 1) \qquad l' = l +\!\!+ [r_0; \ldots; r_m]}{\mathtt{CVUnspec}(x, n, l) \ \$ \ [r_0; \ldots; r_m] \Downarrow_{\mathrm{a}} \mathtt{CVUnspec}(x, n', l')}$$

Pattern-Matching:

$$\frac{}{\Sigma + \{\mathtt{PWild} \mapsto r\} \Downarrow_{\mathrm{p}} \Sigma} \qquad \frac{}{\Sigma + \{\mathtt{PVar}(x) \mapsto r\} \Downarrow_{\mathrm{p}} (x, r) :: \Sigma}$$

$$\frac{\Sigma + \{p \mapsto r\} \Downarrow_{\mathrm{p}} \Sigma'}{\Sigma + \{\mathtt{PConstr}(c, p) \mapsto \mathtt{CVConstr}(c, r)\} \Downarrow_{\mathrm{p}} \Sigma'} \qquad \frac{}{\Sigma + \{\mathtt{PTuple}([]) \mapsto \mathtt{CVTuple}([])\} \Downarrow_{\mathrm{p}} \Sigma}$$

$$\frac{\Sigma + \{p \mapsto r\} \Downarrow_{\mathrm{p}} \Sigma' \qquad \Sigma' + \{\mathtt{PTuple}(l_p) \mapsto \mathtt{CVTuple}(l_r)\} \Downarrow_{\mathrm{p}} \Sigma''}{\Sigma + \{\mathtt{PTuple}(p :: l_p) \mapsto \mathtt{CVTuple}(r :: l_r)\} \Downarrow_{\mathrm{p}} \Sigma''}$$

$$\frac{}{\Sigma + \{\mathtt{PRec}([]) \mapsto \mathtt{CVRec}(l_{dr})\} \Downarrow_{\mathrm{p}} \Sigma}$$

$$\frac{l_{dr}(d) = r \qquad \Sigma + \{p \mapsto r\} \Downarrow_{\mathrm{p}} \Sigma' \qquad \Sigma' + \{\mathtt{PRec}(l_{dp}) \mapsto \mathtt{CVRec}(l_{dr})\} \Downarrow_{\mathrm{p}} \Sigma''}{\Sigma + \{\mathtt{PRec}((d, p) :: l_{dp}) \mapsto \mathtt{CVRec}(l_{dr})\} \Downarrow_{\mathrm{p}} \Sigma''}$$

Figure 8.3: Concrete Interpretation (part 2)

Chapter 9

Primer on Functional Correspondence

Functional correspondence [5] is a systematic approach for converting a big-step semantics into an abstract machine. It has been manually applied to many languages with different features [13, 3, 43, 14, 11, 4, 25, 32], showing its robustness and usefulness. In Chapters 10 and 11, we use it to rephrase the semantics of Skel—the meta-language of skeletal semantics— into non-deterministic and deterministic abstract machines.

Starting from an interpreter, the transformation combines several known techniques, notably CPS translation [44] (Section 9.2) and defunctionalization [47] (Section 9.3) to progressively generate an abstract machine (Section 9.4).

The strategy can also be reversed to transform a large range of abstract machines into big-step semantics, under some conditions [5, 27, 15]. In this overview, we focus only on the transformation from big-step semantics to abstract machines, as it is the only direction we use in this document.

Throughout this chapter, we use the arithmetic expressions from Chapter 2 as a small running example. For readability, the pseudo-code is written in an OCaml-like syntax. The full transformation on the complete IMP example is also available in Appendix A.

9.1 Rewrite the Semantics in Pseudo-Code

Functional correspondence was designed to bridge the gap between abstract machines and interpreters, i.e., semantics written either in pseudo-code or in a functional meta-language. If the input interpreter uses higher-order functions, the first step of functional correspondence would be to get rid of anonymous functions with free variables. This can be done by defunctionalization (as in Section 9.3 below).

In this document, the starting point is instead a big-step semantics defined with inference rules. We therefore rewrite the input semantics into an interpreter using pseudo-code. As an example, we transform the following toy language.

$$a ::= x \mid n \mid \texttt{Plus}(a_1, a_2)$$

$$\frac{}{\langle \sigma, x \rangle \Downarrow_{\mathrm{a}} \sigma(x)} \qquad \frac{}{\langle \sigma, n \rangle \Downarrow_{\mathrm{a}} n} \qquad \frac{\langle \sigma, a_1 \rangle \Downarrow_{\mathrm{a}} n_1 \qquad \langle \sigma, a_2 \rangle \Downarrow_{\mathrm{a}} n_2}{\langle \sigma, \mathtt{Plus}(a_1, a_2) \rangle \Downarrow_{\mathrm{a}} n_1 + n_2}$$

For each type defined in the syntax, we similarly define the same type in pseudo-code. Since we do not explicitly define variables (x), integers (n), and states (σ), we simply assume the corresponding types exists in the pseudo-code.

```
type aexp =
| Var of string
| Nat of nat
| Plus of aexp * aexp
```

Then, for each evaluation predicate (here, $\Downarrow_{\mathrm{a}}$), we create an evaluation function (here named `eval`). The input elements are the terms being evaluated; the output type is the result of the evaluation. The body of the evaluation function should merge the behaviors of the different inference rules. Notably, each premise assuming an evaluation predicate (e.g., $\langle \sigma, a_1 \rangle \Downarrow_{\mathrm{a}} n_1$) is translated into a call to the corresponding evaluation function (e.g., `let n1 = eval s a1 in ...`). When the semantics is deterministic, like here, the function can easily be defined using a pattern-matching on the term to evaluate. Otherwise, we also need to introduce a non-deterministic choice operator in the pseudo-code (see Section 10.1).

```
let eval (s : state) (a : aexp) : nat =
match a with
| Var(x) -> lookup s x
| Nat(n) -> n
| Plus(a1, a2) ->
  let n1 = eval s a1 in
  let n2 = eval s a2 in
  n1 + n2
```

Since we create temporary pseudo-code, only used as a transition between big-step and abstract machine, we can invent names for the different basic operations. For instance, for a state s, we note `lookup s x` for looking up the associated integer.

We obtain a pseudo-interpreter corresponding to our big-step semantics. Since the initial inference rules do not manipulate functions, the pseudo-code does not use higher-order functions, and we do not need to perform any additional manipulation.

9.2 CPS Transform

The next phase of functional correspondence is to transform all evaluation functions into Continuation Passing Style [44].

This first change makes the order of operations more explicit by creating tail-recursive function calls. At every program point, the rest of the computation is expressed as a function called a continuation. The CPS transformation modifies every evaluation function to take such a continuation as an additional argument. In practice, we apply two modifications throughout the pseudo-code.

First, at every return point of every evaluation function, instead of directly returning the obtained result, we call the current continuation with the result as argument. For instance, instead of returning an integer `n`, we make the function call `k n`.

Second, we make use of continuations to prevent nested computations. Every function call of the form `let n = eval .. in ..` is changed into a final call `eval .. (fun n -> ..)`. The chaining in the computation is replaced by a continuation, containing what is computed after the `in` keyword.

For our running example, we obtain the following pseudo-code.

```
let eval s a (k : nat -> nat) : nat =
match a with
| Var(x) -> k (lookup s x)
| Nat(n) -> k n
| Plus(a1, a2) ->
  eval s a1 (fun n1 ->
  eval s a2 (fun n2 ->
  k (n1+n2)))
```

In the general case, the typing should be `let eval s a (k : nat -> 't) : 't = ...`, i.e., continuations can be made polymorphic. Since we do not need it, we output a `nat` for simplicity.

This creates a very directed semantics where every behavior—ignoring basic operations like `lookup` and addition—is simply a final call to another function, either a continuation or an evaluation function.

9.3 Defunctionalization

The next main phase of functional correspondence is to defunctionalize the continuations generated previously. Abstract machines manipulate explicit objects, and not functions nor closures. As such, we need to reify—i.e., transform into objects—the anonymous functions used as continuations.

For each function type among the anonymous functions in the pseudo-code, we create a new associated object type. For our example, we only have a single function type used by all continuations: `nat -> nat`. We create a corresponding object type named `kt`.

Then, we create a new constructor (of the associated type) for each anonymous function. The arguments of the constructor are the free variables of the anonymous function. In our example,

we have two anonymous functions, the second being `(fun n2 -> k (n1+n2))`. Since its free variables are `n1` and `k`, we create an associated constructor `KPlus2 of nat * kt`. Similarly, the first anonymous function has three free variables (`s`, `a2`, and `k`), and the corresponding constructor has three arguments.

The goal is to replace each anonymous function with a corresponding object. However, continuations can be applied (e.g., `k n`) while objects cannot. For each new object type (here, `kt`), we create what we call a *dispatch* function (here named `disp`) that will trigger the evaluation of the corresponding anonymous function. For an anonymous function of type `t1 -> t2` for which we created a new object of type `kt`, we also create a dispatch function of type `kt -> t1 -> t2`. The dispatch function associates to each new constructor the code of its corresponding anonymous function.

Then, we can replace every anonymous function with an object using the corresponding new constructor. At the same time, we change every continuation call to use the dispatch function, replacing for instance `k n` with `disp k n`.

```
type kt =
| KID
| KPlus1 of state * aexp * kt
| KPlus2 of nat * kt

let disp (k : kt) (n : nat) : nat =
match k with
| KPlus1(s, a2, k') ->
  eval s a2 KPlus2(n, k')
| KPlus2(n1, k') ->
  disp k' (n1+n)

let eval s a (k : kt) : nat =
match a with
| Var(x) -> disp k (lookup s x)
| Nat(n) -> disp k n
| Plus(a1, a2) ->
  eval s a1 KPlus1(s, a2, k)
```

We obtain an equivalent semantics without anonymous functions. Continuations are now objects, even if they still indirectly contain the rest of the computation to perform. Applying a continuation now requires an explicit call to the corresponding dispatch function.

Additionally, we create a continuation constructor `KID` without arguments. It can be seen as an empty continuation, or the identity function ($\lambda n.n$), and is used to start and stop a computation (see next section). In `disp`, there is no dispatch rule for `KID`, as there is no computation left to perform and we expect the abstract machine to stop.

9.4 Abstract Machine

The pseudo-code at the end of the previous section is almost an abstract machine. The only difference is that we want to stage a computation into several tiny steps, so we need to introduce a pause at every function call. We also stop using pseudo-code, and introduce more standard notations.

Types can easily be transformed into usual syntactic definitions.

$$a ::= x \mid n \mid \texttt{Plus}(a_1, a_2)$$
$$k ::= \texttt{K}_{\text{id}} \mid \texttt{KPlus1}(\sigma, a, k) \mid \texttt{KPlus2}(n_1, k)$$

For each function of the pseudo-code, we create an evaluation mode for our new abstract machine. For each evaluation mode, the arguments of the machine states are simply the arguments of the corresponding function. For our example, we have machine states of the form $\langle \sigma, a, k \rangle_{\text{e}}$ for the function `eval`, and of the form $\langle k, n \rangle_{\text{k}}$ for the function `disp`. I.e., a machine state corresponds to the computation of one of the functions of the pseudo-code.

Then, the steps of the abstract machine follow the different cases of the pseudo-code. For each behavior of the pseudo-code, we create a step: the left-hand side state corresponds to the evaluation of the input arguments; the right-hand side state corresponds to the final function call of the behavior. For instance, running the `eval` function with arguments `s Nat(n) k` makes a function call to `disp k n`; this is translated into a step $\langle \sigma, n, k \rangle_{\text{e}} \to \langle k, n \rangle_{\text{k}}$.

For arithmetic expressions, we end up with five steps.

$$\langle \sigma, x, k \rangle_{\text{e}} \to \langle k, \sigma(x) \rangle_{\text{k}}$$
$$\langle \sigma, n, k \rangle_{\text{e}} \to \langle k, n \rangle_{\text{k}}$$
$$\langle \sigma, \texttt{Plus}(a_1, a_2), k \rangle_{\text{e}} \to \langle \sigma, a_1, \texttt{KPlus1}(\sigma, a_2, k) \rangle_{\text{e}}$$

$$\langle \texttt{KPlus1}(\sigma, a, k), n \rangle_{\text{k}} \to \langle \sigma, a, \texttt{KPlus2}(n, k) \rangle_{\text{e}}$$
$$\langle \texttt{KPlus2}(n_1, k), n \rangle_{\text{k}} \to \langle k, (n_1 + n) \rangle_{\text{k}}$$

Note that there is no step for the evaluation of the basic continuation $\texttt{K}_{\text{id}}$, as there is no corresponding behavior in the pseudo-code. This continuation is used as a base for starting and stopping a computation, like the continuation $\bullet$ in the example of Section 2.5. To evaluate a configuration $\langle s, a \rangle$ with this abstract machine, we would create an initial machine state $\langle s, a, \texttt{K}_{\text{id}} \rangle_{\text{e}}$ and repeatedly reduce. The machine would stop at a state of the form $\langle \texttt{K}_{\text{id}}, n \rangle_{\text{k}}$, indicating that the result is n.

The derivation and the final abstract machine for the complete IMP example are available in Appendix A.

Chapter 10

Non-Deterministic Abstract Machine for Skeletal Semantics

The first step towards a certified interpreter is to transform the big-step interpretation of skeletons into an abstract machine while keeping the non-determinism of the concrete interpretation. We thus derive a Non-Deterministic Abstract Machine (NDAM) from the concrete interpretation. While the NDAM obtained in this chapter is not executable, the transformation we present serves as a basis to generate a deterministic abstract machine in Chapter 11. Also, we use the NDAM in Coq as an intermediate semantics to certify the correctness of the final deterministic abstract machine with respect to the initial concrete interpretation.

We follow the textbook strategy of functional correspondence presented in Chapter 9. After rewriting the concrete interpretation from inference rules to pseudo-code (Section 10.1), we perform a CPS-transformation (Section 10.2), a phase of defunctionalization (Section 10.3), and create the abstract machine proper and its evaluation modes (Section 10.4). Finally, Section 10.5 presents the Coq equivalence result between the resulting non-deterministic abstract machine and the rules of Figure 8.2.

In each code snippet illustrating the different phases of the transformation, we use a gray background to indicate changes relative to previous sections. All intermediate phases of this derivation are available in Appendix D.1.

10.1 Pseudo-interpreter

We translate the concrete interpretation into pseudo-code, using an OCaml-like syntax [36] where natural numbers have constructors `Z` and `S`. We translate each predicate of Figure 8.2 into evaluation functions, resulting into the following pseudo-code for skeletons:

```
let eval_sk (sk : skeleton) (e : env) : cvalue =
match sk with
| Branching (skl) ->
  (* oracle picking the correct skeleton *)
  let sk' = pick skl in
  eval_sk sk' e
```

```
| ...
| LetIn (p, sk1, sk2) ->
  let r = eval_sk sk1 e in
  let e2 = eval_pat p r e in
  eval_sk sk2 e2
```

The non-determinism of the evaluation of a branching is reflected by the `pick` function which behaves like an oracle. It is able to choose, within a list, an appropriate element that will make the whole evaluation succeed and output a result.

Similarly, we use the same function for unspecified skelterms. As presented before, the evaluation of such a skelterm depends on its arity. If it does not require arguments, we immediately call its interpretation, and then arbitrarily select a result from the obtained list, using `pick`. Otherwise, we create a partially applied skelterm initialized with zero arguments.

```
let eval_trm (t : skelterm) (e : env) : cvalue =
match t with
| TVar (v) -> match v with
  | VLet (x) -> lookup e x
  | VSpec (h) -> eval_trm (specdecl h) []
  | VUnspec (f) -> match arity f with
    | Z -> let rl = unspecdecl f [] in
           (* oracle picking the correct cvalue *)
           let r = pick rl in
           r
    | S m -> CVUnspec (f, m, [])
| TConstr (c, t') ->
  let r = eval_trm t' e in
  CVConstr (c, r)
| ...
```

We also define evaluation functions for lists of skelterms and lists of pattern-matchings. The former corresponds to mapping the evaluation function for skelterms over the list, while the latter behaves like a fold over the list, in accordance with the rules of Figure 8.2.

The full initial pseudo-code is available in Appendix D.1.1.

10.2 CPS-Transform

We apply a CPS transformation to the main evaluation functions: `eval_trm`, `eval_sk`, `eval_pat`, `lookup`, etc. We do not modify the parametric functions (`pick` / `specdecl` / `arity` / `unspecdecl`) nor the basic functions such as the concatenation of two lists.

```
let eval_trm t e (k : cvalue -> cvalue) : cvalue =
match t with
```

```
| TVar (v) -> match v with
  | VLet (x) -> lookup e x k
  | VSpec (h) -> eval_trm (specdecl h) [] k
  | VUnspec (f) -> match arity f with
    | Z -> let r = pick (unspecdecl f []) in
           k r
    | S m -> k (CVUnspec (f, m, []))
| TConstr (c, t') ->
  eval_trm t' e (fun r -> k (CVConstr (c, r)))
| ...

let eval_sk sk e (k : cvalue -> cvalue) : cvalue =
match sk with
| Branching (skl) ->
  let sk' = pick skl in
  eval_sk sk' e k
| ...
| LetIn (p, sk1, sk2) ->
  eval_sk sk1 e (fun r ->
  eval_pat p r e (fun e2 ->
  eval_sk sk2 e2 k))
```

The arity zero case is an example where the result is passed to the continuation. Calls to unmodified functions, such as `pick`, are left unchanged. A call to an evaluation function f is changed so that we create a continuation for the rest of the code and make a single tail-call to f with it. When several of such calls are chained, we build nested continuations, like for the constructor `LetIn`: the continuation after evaluating `sk1` consists of performing the pattern-matching with a continuation evaluating `sk2`.

The full pseudo-code at this point is available in Appendix D.1.2.

10.3 Defunctionalization

For each anonymous function generated in Section 10.2, we create a fresh constructor, whose arguments are the free variables of the function. In the code, we replace each anonymous continuation by its new corresponding constructor.

```
type krt =
| KRID
| KRLet of pattern * skeleton * env * krt
| ...

let eval_sk sk e (k : krt) : cvalue =
match sk with
```

```
| Branching (skl) ->
  let sk' = pick skl in
  eval_sk sk' e k
| ...
| LetIn (p, sk1, sk2) ->
  eval_sk sk1 e (KRLet (p,sk2,e,k))
```

We create the type `krt` (and its corresponding dispatch function `disp_kr`) to represent continuations expecting a result, i.e., a concrete value. For example, the continuation of the evaluation of `sk1` in the LetIn case of Section 10.2 is replaced by the constructor `KRLet`, whose arguments are the free variables of the continuation. We also replace any application of the continuation `k r` by a call `disp_kr k r`, like in the arity zero case. The extra constructor kr_{id} represents the identity continuation.

```
let disp_kr (k : krt) (r : cvalue) : cvalue =
match k with
| KRLet (p,sk,e,k') -> eval_pat p r e (KELet (sk,k'))
| ...
```

```
let eval_trm t e (k : krt) : cvalue =
match t with
| TVar (v) -> match v with
  | VLet (x) -> lookup e x k
  | VSpec (h) -> eval_trm (specdecl h) [] k
  | VUnspec (f) -> match arity f with
    | Z -> let r = pick (unspecdecl f []) in
           disp_kr k r
    | S m -> disp_kr k (CVUnspec (f, m, []))
| ...
```

Continuations expecting concrete values are not the only possibility in our CPS-transformed evaluator: some expect either a list of concrete values, an environment, or the content of a record. Therefore, we also create new types `klt`, `ket`, and `kdt` corresponding to continuations expecting respectively lists of cvalues, environments, and record contents. These three types are also given their own identity continuation: kl_{id}, ke_{id}, and kd_{id}.

```
type ket =
| KEID
| KELet of skeleton * krt
| ...
let disp_ke (k : ket) (e : env) : cvalue =
match k with
| KELet (sk, k') -> eval_sk sk e k'
| ...
```

$$
\begin{aligned}
\langle \texttt{Branching}(l), \Sigma, k\rangle_{\text{sk}} &\to \langle S, \Sigma, k\rangle_{\text{sk}} \quad \texttt{for } (S \in l) \\
\langle \texttt{LetIn}(p, S_1, S_2), \Sigma, k\rangle_{\text{sk}} &\to \langle S_1, \Sigma, \texttt{KRLet}(p, S_2, \Sigma, k)\rangle_{\text{sk}} \\
\cdots &\to \cdots \\
\langle \text{kr}_{\text{id}}, r\rangle_{\text{kr}} &\not\to \texttt{(* end of computation *)} \\
\langle \texttt{KRLet}(p, S, \Sigma, k), r\rangle_{\text{kr}} &\to \langle p, r, \Sigma, \texttt{KELet}(S, k)\rangle_{\text{pat}} \\
\cdots &\to \cdots \\
\langle \texttt{KELet}(S, k), \Sigma\rangle_{\text{ke}} &\to \langle S, \Sigma, k\rangle_{\text{sk}} \\
\cdots &\to \cdots
\end{aligned}
$$

Figure 10.1: Non-Deterministic Abstract Machine

We can see `KELet` being used when we dispatch the continuation `KRLet` in `disp_kr`: it corresponds to the continuation of the evaluation of the pattern `p` in the LetIn case of Section 10.2.

The full pseudo-code at this point is available in Appendix D.1.3.

10.4 Abstract Machine

We can now generate the non-deterministic abstract machine. Each evaluation and dispatch function of the pseudo-interpreter becomes a mode of the abstract machine, with the same arguments as the ones of the function. For instance, the function `eval_sk S Σ k` is turned into the state $\langle S, \Sigma, k\rangle_{\text{sk}}$, while `disp_kr k r` corresponds to $\langle k, r\rangle_{\text{kr}}$.

Each path in the code of Section 10.3 produces a step of the abstract machine, where the resulting state corresponds to the tail-call. The parametric functions (such as `specdecl` and `pick`), can be translated as guarding conditions of the abstract machine. For instance, the LetIn case of the pseudo-code generates the following step.

$$\langle \texttt{LetIn}(p, S_1, S_2), \Sigma, k\rangle_{\text{sk}} \to \langle S_1, \Sigma, \texttt{KRLet}(p, S_2, \Sigma, k)\rangle_{\text{sk}}$$

This correctly produces a stepping relation. Some of the steps of the NDAM are given in Figure 10.1. The full abstract machine is available in Appendix D.1.4.

The initial states of the NDAM correspond to injections $\langle t, [], \text{kr}_{\text{id}}\rangle_{\text{trm}}$ and $\langle S, [], \text{kr}_{\text{id}}\rangle_{\text{sk}}$ for evaluating respectively a skelterm t and a skeleton S. Running this abstract machine in practice would require an oracle, as the rule $\langle \texttt{Branching}([S_1, \ldots, S_n]), \Sigma, k\rangle_{\text{sk}} \to \langle S_i, \Sigma, k\rangle_{\text{sk}}$ for branchings is non-deterministic. If a wrong branch is selected, the abstract machine might fail to produce a result. If the NDAM reaches a final state of the form $\langle \text{kr}_{\text{id}}, r\rangle_{\text{kr}}$, then r is a valid result of the evaluation.

For all states a and b, we write $a \to^* b$ for the reflexive and transitive closure of the stepping relation of the abstract machine, and $a \to^n b$ for a sequence of n steps with $n \in \mathbb{N}$.

10.5 Certification

The NDAM is formalized in Coq in the file `Concrete_ndam.v` (see the implementation [9]), and we prove this NDAM to be sound and complete with respect to the concrete interpretation. The Coq proof is done at the meta level (parametric in the skeletal semantics), as such it is valid independently from the language we are interested in (e.g., λ-calculus).

Intuitively, each big-step relation of the concrete interpretation (see Figure 8.2) corresponds to an evaluation mode of the NDAM. For instance, we have:

$$\Sigma \vdash S \Downarrow_{\mathrm{s}} r \;\text{ iff }\; \langle S, \Sigma, \mathrm{kr_{id}} \rangle_{\mathrm{sk}} \to^* \langle \mathrm{kr_{id}}, r \rangle_{\mathrm{kr}}$$

We call $\mathrm{kr_{id}}$, $\mathrm{ke_{id}}$, $\mathrm{kl_{id}}$, and $\mathrm{kd_{id}}$ the *basic continuations* of the abstract machine. States stuck evaluating a basic continuation, e.g., $\langle \mathrm{kr_{id}}, r \rangle_{\mathrm{kr}}$, are called *final states*.

Since the NDAM also manipulates reified continuations, we need a few results to reshape them when appropriate. We first define *continuation composition* $k[k']$. Given two continuations k and k', we write $k[k']$ for the substitution of the basic continuation inside k by k', defined as follows:

$$\begin{aligned} \mathrm{kr_{id}}[k'] &\triangleq k' \\ \texttt{KRLet}(p, S, \Sigma, k)[k'] &\triangleq \texttt{KRLet}(p, S, \Sigma, k[k']) \\ \ldots &\triangleq \ldots \end{aligned}$$

This continuation composition can be naturally extended to machine states, e.g., $\langle S, \Sigma, k \rangle_{\mathrm{sk}}[k'] \triangleq \langle S, \Sigma, k[k'] \rangle_{\mathrm{sk}}$.

Because the NDAM never pattern-matches on basic continuations, machine steps are preserved by continuation composition.

Lemma 10.5.1. For all a, b, and k, $a \to b$ implies $a[k] \to b[k]$.

This lemma is sufficient to prove the completeness part. Henceforth, we state lemmas and theorems for the skeleton mode of the machine, but they can be stated similarly for the other modes.

Theorem 10.5.2. For all S, Σ, and r, if $\Sigma \vdash S \Downarrow_{\mathrm{s}} r$, then $\langle S, \Sigma, \mathrm{kr_{id}} \rangle_{\mathrm{sk}} \to^* \langle \mathrm{kr_{id}}, r \rangle_{\mathrm{kr}}$.

From the concrete interpretation to the NDAM, we proceed by structural induction on the inductive properties of the concrete interpretation (Figure 8.2), and simply apply the induction hypothesis and merge sequences, using Lemma 10.5.1.

Sketch. For instance, for the constructor LetIn:

$$\frac{\Sigma \vdash S_1 \Downarrow_{\mathrm{s}} r' \qquad \Sigma + \{p \mapsto r'\} \Downarrow_{\mathrm{p}} \Sigma' \qquad \Sigma' \vdash S_2 \Downarrow_{\mathrm{s}} r}{\Sigma \vdash \texttt{LetIn}(p, S_1, S_2) \Downarrow_{\mathrm{s}} r}$$

Applying the induction hypothesis on each premise gives us:

(1) $\langle S_1, \Sigma, \mathrm{kr_{id}} \rangle_{\mathrm{sk}} \to^* \langle \mathrm{kr_{id}}, r' \rangle_{\mathrm{kr}}$

(2) $\langle p, r', \Sigma, \mathrm{ke_{id}} \rangle_{\mathrm{pat}} \to^* \langle \mathrm{ke_{id}}, \Sigma' \rangle_{\mathrm{ke}}$

(3) $\langle S_2, \Sigma', \mathrm{kr_{id}} \rangle_{\mathrm{sk}} \to^* \langle \mathrm{kr_{id}}, r \rangle_{\mathrm{kr}}$

We then merge them to reconstruct a sequence, held by a few additional steps. The identity continuations are lifted using Lemma 10.5.1 when necessary.

$$\begin{aligned}
& \langle \texttt{LetIn}(p, S_1, S_2), \Sigma, \mathrm{kr_{id}} \rangle_{\mathrm{sk}} & \\
\to\ & \langle S_1, \Sigma, \texttt{KRLet}(p, S_2, \Sigma, \mathrm{kr_{id}}) \rangle_{\mathrm{sk}} & \texttt{Definition} \\
\to^*\ & \langle \texttt{KRLet}(p, S_2, \Sigma, \mathrm{kr_{id}}), r' \rangle_{\mathrm{kr}} & (1) + \texttt{Lemma} \\
\to\ & \langle p, r', \Sigma, \texttt{KELet}(S_2, \mathrm{kr_{id}}) \rangle_{\mathrm{pat}} & \texttt{Definition} \\
\to^*\ & \langle \texttt{KELet}(S_2, \mathrm{kr_{id}}), \Sigma' \rangle_{\mathrm{ke}} & (2) + \texttt{Lemma} \\
\to\ & \langle S_2, \Sigma', \mathrm{kr_{id}} \rangle_{\mathrm{sk}} & \texttt{Definition} \\
\to^*\ & \langle \mathrm{kr_{id}}, r \rangle_{\mathrm{kr}} & (3)
\end{aligned}$$

□

For the reverse implication, we make explicit where continuations are needed by splitting sequences of steps at the point where the continuation is actually focused on. Up to that point, the continuation could be replaced by a basic continuation, as stated in the following result. If the sequence does not use the continuation (e.g., $\langle \texttt{LetIn}(p, S_1, S_2), \Sigma, k \rangle_{\mathrm{sk}} \to \langle S_1, \Sigma, \texttt{KRLet}(p, S_2, \Sigma, k) \rangle_{\mathrm{sk}}$), there is no splitting point for us to exploit. To avoid this case, the lemma assumes the sequence leads to a final state, ensuring the continuation is used.

Lemma 10.5.3. For all S, Σ, k, n, and final state b, if we have $\langle S, \Sigma, k \rangle_{\mathrm{sk}} \to^n b$ then there exist n_1, n_2, and r such that $\langle S, \Sigma, \mathrm{kr_{id}} \rangle_{\mathrm{sk}} \to^{n_1} \langle \mathrm{kr_{id}}, r \rangle_{\mathrm{kr}}$, $\langle k, r \rangle_{\mathrm{kr}} \to^{n_2} b$, and $n = n_1 + n_2$.

The proof is by strong induction on the length of the sequence of steps. Intuitively, it holds because we stop at the first step pattern-matching the initial continuation; the steps before the cut are still valid after changing the unused continuation.

With this lemma, we can tackle the soundness part of the certification.

Theorem 10.5.4. For all S, Σ, and r, if $\langle S, \Sigma, \mathrm{kr_{id}} \rangle_{\mathrm{sk}} \to^* \langle \mathrm{kr_{id}}, r \rangle_{\mathrm{kr}}$, then $\Sigma \vdash S \Downarrow_{\mathrm{s}} r$.

We proceed by strong induction on the length of the sequence of steps, and make use of Lemma 10.5.3.

Sketch. For instance, for the constructor LetIn, we start with:

$$\langle \texttt{LetIn}(p, S_1, S_2), \Sigma, \mathrm{kr_{id}} \rangle_{\mathrm{sk}} \to^* \langle \mathrm{kr_{id}}, r \rangle_{\mathrm{kr}}.$$

Using the definition of the NDAM (Figure 10.1) and Lemma 10.5.3 several times, we can cut the sequence into the following pieces:

- $\langle S_1, \Sigma, \mathrm{kr_{id}} \rangle_{\mathrm{sk}} \to^* \langle \mathrm{kr_{id}}, r' \rangle_{\mathrm{kr}}$
- $\langle p, r', \Sigma, \mathrm{ke_{id}} \rangle_{\mathrm{pat}} \to^* \langle \mathrm{ke_{id}}, \Sigma' \rangle_{\mathrm{ke}}$
- $\langle S_2, \Sigma', \mathrm{kr_{id}} \rangle_{\mathrm{sk}} \to^* \langle \mathrm{kr_{id}}, r \rangle_{\mathrm{kr}}$

Each piece is strictly shorter than the initial sequence, so we can apply the induction hypothesis on each of them. We obtain the premises of the LetIn concrete interpretation (cf. Figure 8.2) so we can conclude. The other cases are similar. □

Chapter 11

Deterministic Abstract Machine for Skeletal Semantics

The abstract machine of Chapter 10 is non-deterministic, and thus does not offer a computable semantics for the empirical evaluation of skelterms and skeletons. In this chapter, we show how to obtain a deterministic machine using explicit backtracking. The deterministic machine returns at most one of the possible results; computing all of them is impossible in presence of non-terminating executions. We choose to try branches in order and to backtrack if necessary.

To this end, we consider a more complex CPS transformation with a failure continuation (Section 11.1). We then derive the resulting deterministic abstract machine through defunctionalization in Section 11.2, and present the Coq certification of its soundness in Section 11.3.

11.1 CPS-Transform

Starting from the big-step evaluator of Section 10.1, we modify every evaluation function to take two continuations [24]:

- a success continuation `k` as in Section 10.2, indicating what to do if the computation succeeds, and
- a failure continuation `fk`, remembering the last checkpoint to backtrack to in case of failure.

When the evaluator of Section 10.1 chooses between several results using `pick`, we instead decide to always evaluate the first of the possible choices and create a checkpoint for the other possibilities. For branchings, it means evaluating the first branch, and remembering the others in the failure continuation. When we reach an empty branching, it means that all the evaluations of all the branches failed, and we need to backtrack by calling `fk`. Once again, we use a gray background to indicate changes or new functions, here relative to the initial pseudo-code of Section 10.1.

```
let eval_sk sk e
  (k : cvalue -> (() -> cvalue) -> cvalue)
  (fk : () -> cvalue) : cvalue =
```

```
match sk with
| Branching (skl) -> match skl with
  | [] -> fk ()
  | sk'::l ->
    eval_sk sk' e k
     (fun _ -> eval_sk (Branching(l)) e k fk)
| ...
| LetIn (p, sk1, sk2) ->
  eval_sk sk1 e (fun r fk2 ->
  eval_pat p r e (fun e2 fk3 ->
  eval_sk sk2 e2 k fk3) fk2) fk
```

The other source of non-determinism is the evaluation of unspecified skelterms, which returns a list of possible results (see Figure 8.2). In this case, we create a new evaluation function `select_list` which follows the same principle as for branchings: it sequentially tries the elements of the list, and otherwise calls the failure continuation.

```
let select_list rl
  (k : cvalue -> (() -> cvalue) -> cvalue)
  (fk : () -> cvalue) : cvalue =
match rl with
| [] -> fk ()
| r::l -> k r (fun _ -> select_list l k fk)
```

Also, we complete the pattern-matchings of the pseudo-code to make them total, and backtrack in the problematic cases. Initially, the pseudo-code follows the rules of Figure 8.2 and does not cover cases that cannot evaluate. For instance, the rules for looking up a variable in an environment assumes the environment to have at least one entry:

```
def lookup e x : cvalue = match e with
| (y,r)::e2 -> if x=y then r
               else lookup e2 x
```

Here, to cover all possible behaviors, we add a case for when the environment is empty. We add similar backtracking cases at several points throughout the pseudo-code.

```
def lookup e x k fk : cvalue = match e with
| [] -> fk ()
| (y,r)::e2 -> if x=y then k r fk
               else lookup e2 x k fk
```

Lastly, we need to pass the current failure continuation as an argument of the success continuation `k`. The reason is that a computation can seemingly succeed at first and fail later on; we would then need to backtrack to checkpoints unknown to `k`. This is obvious for functions

such as `select_list`: it succeeds in selecting an element of the list, but the continuation might fail later.

The extra failure continuation does not fundamentally change how deterministic constructors are CPS-transformed, as we can see with the resulting code for `LetIn` in the above `eval_sk` function: we just need to pass along the failure continuation, and be mindful of the new type of success continuations.

The full pseudo-code after CPS transformation is available in Appendix D.2.1.

11.2 Defunctionalization and Abstract Machine

On top of the types `krt`, `klt`, `ket`, and `kdt` of Section 10.3, this defunctionalization phase generates a new type `fkt` and its dispatch function `disp_fk` for failure continuations.

```
type fkt =
| FEmpty
| FSK of skeleton * env * krt * fkt
| FList of (cvalue list) * krt * fkt
let disp_fk fk = match fk with
| FSK(sk, e, k, fk') -> eval_sk sk e k fk'
| FList(rl, k, fk') -> select_list rl k fk'
```

The constructors `FSK` and `FList` correspond to the anonymous functions of Section 11.1 where we construct backtracking checkpoints for trying respectively the arguments of a branching or the list of possible interpretations of an unspecified skelterm. We also create a constructor `FEmpty` representing an empty failure continuation, used to start a computation. It has no rule in the dispatch function, since no backtrack is possible.

```
let select_list rl (k : krt) (fk : fkt): cvalue =
match rl with
| [] -> disp_fk fk
| r::l -> k r (FList (l, k, fk))

let eval_sk sk e (k : krt) (fk : fkt): cvalue =
match sk with
| Branching (skl) -> match skl with
  | [] -> disp_fk fk
  | sk'::l -> eval_sk sk' e k
              (FSK (Branching(l), e, k, fk))
| ...
```

As previously, the new constructors replace the anonymous functions, and we add a call to the dispatch function at every backtrack point.

$$
\begin{aligned}
\langle \texttt{Branching}([]), \Sigma, k, f\rangle_{\text{sk}} &\to \langle f\rangle_{\text{fk}} \\
\langle \texttt{Branching}(S :: l), \Sigma, k, f\rangle_{\text{sk}} &\to \langle S, \Sigma, k, \texttt{FSK}(\texttt{Branching}(l), \Sigma, k, f)\rangle_{\text{sk}} \\
\langle \texttt{LetIn}(p, S_1, S_2), \Sigma, k, f\rangle_{\text{sk}} &\to \langle S_1, \Sigma, \texttt{KRLet}(p, S_2, \Sigma, k), f\rangle_{\text{sk}} \\
\cdots &\to \cdots \\
\langle [], k, f\rangle_{\text{lst}} &\to \langle f\rangle_{\text{fk}} \\
\langle r :: l, k, f\rangle_{\text{lst}} &\to \langle k, r, \texttt{FList}(l, k, f)\rangle_{\text{kr}} \\
\cdots &\to \cdots \\
\langle \text{kr}_{\text{id}}, r, f\rangle_{\text{kr}} &\not\to \texttt{(* end of computation *)} \\
\langle \texttt{KRLet}(p, S, \Sigma, k), r, f\rangle_{\text{kr}} &\to \langle p, r, \Sigma, \texttt{KELet}(S, k), f\rangle_{\text{pat}} \\
\cdots &\to \cdots \\
\langle \texttt{KELet}(S, k), \Sigma, f\rangle_{\text{ke}} &\to \langle S, \Sigma, k, f\rangle_{\text{sk}} \\
\cdots &\to \cdots \\
\langle \texttt{FEmpty}\rangle_{\text{fk}} &\not\to \texttt{(* failure *)} \\
\langle \texttt{FSK}(S, \Sigma, k, f)\rangle_{\text{fk}} &\to \langle S, \Sigma, k, f\rangle_{\text{sk}} \\
\langle \texttt{FList}(l, k, f)\rangle_{\text{fk}} &\to \langle l, k, f\rangle_{\text{lst}}
\end{aligned}
$$

Figure 11.1: Deterministic Abstract Machine

The full pseudo-code at this point is available in Appendix D.2.2.

Transforming the defunctionalized evaluator produces a deterministic abstract machine, as each machine state is either stuck or reduces to exactly one machine state. Compared to the non-deterministic machine of Section 10.4, the states carry an extra argument corresponding to the failure continuation.

The deterministic machine has two additional modes `lst` and `fk`, which correspond to the functions (`select_list` and `disp_fk`): we present their steps in Figure 11.1. As expected, the `lst` mode tries the continuation on each element of the list, and triggers the backtracking mode `fk` on the empty list; it is also invoked on an empty branching. The `fk` mode then restores the backtracking checkpoint, unless the failure continuation is empty, in which case the machine stops.

We initialize the machine with either $\langle t, [], \text{kr}_{\text{id}}, \texttt{FEmpty}\rangle_{\text{trm}}$ to evaluate a skelterm t or $\langle S, [], \text{kr}_{\text{id}}, \texttt{FEmpty}\rangle_{\text{sk}}$ for a skeleton S. There are three possible outcomes:

- the machine gets stuck at $\langle \text{kr}_{\text{id}}, r, f\rangle_{\text{kr}}$, which means r is a result (there might be other correct results, but we stop at the first);
- it gets stuck at $\langle \texttt{FEmpty}\rangle_{\text{fk}}$, which means all possible branches have been tried and there is no result;

- the evaluation does not terminate, as the abstract machine can loop on an infinite branch (there might be correct results on other branches).

The complete deterministic abstract machine is available in Appendix D.2.3.

11.3 Certification

The deterministic Abstract Machine (AM) is formalized in Coq in the file `Concrete_am.v` (see the implementation [9]), and once again the certification is independent from the skeletal semantics (language) we are interested in.

We prove in Coq that the AM is sound with respect to the NDAM: if the AM finds a result, then the NDAM can also find the same result. Because the NDAM is sound with respect to the concrete interpretation, so is the AM. However it is not complete: the AM can find at most one of the results, and may loop in an infinite branch even if there is a valid result elsewhere.

To avoid confusion, the stepping relations of the abstract machines are written ($\rightarrow_{\text{AM}}$) and ($\rightarrow_{\text{ND}}$) in this section. We note $\widetilde{x}$ for a sequence $(x_1, \ldots, x_n)$, $\langle \widetilde{x} \rangle_{\text{m}}$ for a NDAM state of mode m, and $\langle \widetilde{x}, f \rangle_{\text{m}}$ for an AM state of mode m—failure continuations are always the last argument in the deterministic case.

Unlike in Section 10.5, both semantics use success continuations so there is no need for lemmas to manipulate them. However, only the AM uses failure continuations, so we do need a few results to handle them.

Firstly, we define *failure continuation composition* $f[f']$ which replaces the empty failure continuation in f by f'.

$$\begin{aligned} \texttt{FEmpty}[f'] &\triangleq f' \\ \texttt{FSK}(S, \Sigma, k, f)[f'] &\triangleq \texttt{FSK}(S, \Sigma, k, f[f']) \\ \texttt{FList}(l, k, f)[f'] &\triangleq \texttt{FList}(l, k, f[f']) \end{aligned}$$

We extend it to AM states so that $\langle \widetilde{x}, f \rangle_{\text{m}}[f'] \triangleq \langle \widetilde{x}, f[f'] \rangle_{\text{m}}$.

As previously, steps hold after composition:

Lemma 11.3.1. For all a, b, and f, $a \rightarrow_{\text{AM}} b$ implies $a[f] \rightarrow_{\text{AM}} b[f]$.

Secondly, we prove another lemma to discard failure continuations. If a sequence never uses it, we can remove it from both the head and tail states. Otherwise, we can split this sequence at the point where it is called, and the first part can be written without using the failure continuation.

Lemma 11.3.2. For all n, AM mode m, and states $\langle \widetilde{x}, f \rangle_{\text{m}}$ and b, if $\langle \widetilde{x}, f \rangle_{\text{m}} \rightarrow^n_{\text{AM}} b$ then either:

- there exists b' such that $\langle \widetilde{x}, \texttt{FEmpty} \rangle_{\text{m}} \rightarrow^n_{\text{AM}} b'$ and $b = b'[f]$, or

- there exist n_1 and n_2 such that $\langle \widetilde{x}, \texttt{FEmpty} \rangle_{\mathrm{m}} \to_{\mathrm{AM}}^{n_1} \langle \texttt{FEmpty} \rangle_{\mathrm{fk}}$, $\langle f \rangle_{\mathrm{fk}} \to_{\mathrm{AM}}^{n_2} b$, and $n = n_1 + n_2$.

The proof is done by strong induction on the length of the sequence of steps, and uses a few basic results about composition, such as associativity.

Lastly, we can pose our main theorem. It states that the AM is sound with respect to the NDAM.

Theorem 11.3.3. For all l, k, r, and f, if $\langle l, k, \texttt{FEmpty} \rangle_{\mathrm{lst}} \to_{\mathrm{AM}}^{*} \langle \mathrm{kr}_{\mathrm{id}}, r, f \rangle_{\mathrm{kr}}$ then there exists $r' \in l$ such that $\langle k, r' \rangle_{\mathrm{kr}} \to_{\mathrm{ND}}^{*} \langle \mathrm{kr}_{\mathrm{id}}, r \rangle_{\mathrm{kr}}$.

For all NDAM mode m (i.e., other than `lst` and `fk`), for all $\langle \widetilde{x}, \texttt{FEmpty} \rangle_{\mathrm{m}}$, r, and f, if $\langle \widetilde{x}, \texttt{FEmpty} \rangle_{\mathrm{m}} \to_{\mathrm{AM}}^{*} \langle \mathrm{kr}_{\mathrm{id}}, r, f \rangle_{\mathrm{kr}}$ then $\langle \widetilde{x} \rangle_{\mathrm{m}} \to_{\mathrm{ND}}^{*} \langle \mathrm{kr}_{\mathrm{id}}, r \rangle_{\mathrm{kr}}$.

In particular, for the mode evaluating skeletons `sk`, the following holds.

Corollary 11.3.4. For all S, Σ, r, and f, if

$$\langle S, \Sigma, \mathrm{kr}_{\mathrm{id}}, \texttt{FEmpty} \rangle_{\mathrm{sk}} \to_{\mathrm{AM}}^{*} \langle \mathrm{kr}_{\mathrm{id}}, r, f \rangle_{\mathrm{kr}},$$

then

$$\langle S, \Sigma, \mathrm{kr}_{\mathrm{id}} \rangle_{\mathrm{sk}} \to_{\mathrm{ND}}^{*} \langle \mathrm{kr}_{\mathrm{id}}, r \rangle_{\mathrm{kr}}.$$

The mode `lst` is treated differently in the theorem because there is no corresponding mode in the NDAM. It amounts to choosing the first cvalue $r' \in l$ that would produce a result. In the NDAM, we would have an oracle picking the right element of l to evaluate.

The proof is done by strong induction on the sequence of the AM. We follow the AM sequence, and most steps correspond to similar steps in the NDAM. We restrict ourselves to sequences with an empty failure continuation to make use of the induction hypothesis. When the AM would create a checkpoint, we use Lemma 11.3.2; a case disjunction on the result allows us to pick the correct branch for the NDAM and keep an empty failure continuation.

Remark. In the proof of Theorem 11.3.3, an AM state with an empty failure continuation and the corresponding NDAM state have exactly the same success continuation. This is why we no longer need the continuation composition of Section 10.5 (and the associated lemmas) to handle them.

Finally, since the NDAM is sound w.r.t. the concrete interpretation, so is the AM.

Theorem 11.3.5. For all S, Σ, r, and f, if

$$\langle S, \Sigma, \mathrm{kr}_{\mathrm{id}}, \texttt{FEmpty} \rangle_{\mathrm{sk}} \to_{\mathrm{AM}}^{*} \langle \mathrm{kr}_{\mathrm{id}}, r, f \rangle_{\mathrm{kr}},$$

then

$$\Sigma \vdash S \Downarrow_{\mathrm{s}} r.$$

Proof. From Theorems 11.3.3 and 10.5.4. □

Chapter 12

Certified Interpreter

We show how to use the deterministic abstract machine of Section 11 to automatically generate a certified OCaml interpreter from a skeletal description of a language.

As stated before, the two abstract machines have been implemented and certified in Coq. Since the NDAM is not computable, it is coded as a relation:

```
Inductive step : state -> state -> Prop
```

However, the AM is computable and coded as a partial function. The only states not being mapped are the final states (i.e., blocked on $\mathrm{kr}_{\mathrm{id}}/\mathrm{kl}_{\mathrm{id}}/\mathrm{ke}_{\mathrm{id}}$ or `FEmpty`):

```
Definition step (a : state) : option state
```

This stepping function can be repeated to form a partial evaluation function. Since Coq only accepts terminating functions, we need a fuel parameter, bounding the number of steps the function can do.

```
Fixpoint evalfuel (n : nat) (a : state) : option cvalue
```

This function extracts the result r of a final state $\langle \mathrm{kr}_{\mathrm{id}}, r, f\rangle_{\mathrm{kr}}$, it fails on $n = 0$ or $a = \langle \texttt{FEmpty}\rangle_{\mathrm{fk}}$, and it steps then recurses with one less fuel otherwise.

From these definitions, using Coq extraction to OCaml [37], we generate an executable version of the deterministic abstract machine. This extraction only has do be done once and it is independent of the user-defined skeletal semantics. The output has been slightly reorganized to make better use of OCaml modules.

At this point, the generated OCaml functions can simulate the evaluation of skeletons. This generic interpreter still needs to be instantiated with an actual OCaml representation of a skeletal semantics. To this end, we use the Necro tool to deeply embed the semantics into Coq, which we then extract to OCaml. I.e., the user definition is translated successively into a Coq datatype and an OCaml datatype. A small script is provided to perform this translation and automatically generate an OCaml module to instantiate the generic abstract machine.

As explained before, the Coq deep embedding—and thus the OCaml module—is parametric in the representation of unspecified types and skelterms (e.g., `ident`, `env`, `extEnv`, and `getEnv` for the λ-calculus of Chapter 8 and Figure 12.1 below). To exploit this module, a user then

needs to provide OCaml types and functions for them before having access to the AM functions, notably `evalfuel` to launch an evaluation.

The soundness proofs of Sections 10.5 and 11.3 are parameterized by the used skeletal semantics. As such, they are valid for any language. So this OCaml interpreter, extracted from the AM and specialized with a language, is sound with respect to the initial concrete interpretation of Section 8.2. If an execution of `evalfuel` produces a result, then this result is a correct behavior of the skeletal semantics. Once again, if the interpreter does not produce a result, we have no guarantees: the interpreter might need more fuel, it might be stuck in an infinite loop, or there might be no correct result at all.

The advantage of working at the meta-level, i.e., proving correction once and for all languages, has a drawback: the execution happens in the meta-language, namely Skel, while a user may prefer working at the level of the language, e.g., λ-terms. This deep embedding requires the user to understand the Skel meta-language. For instance, the user-defined OCaml functions representing unspecified skelterms (filters) manipulate concrete values. Using OCaml macros can alleviate notations, but cannot resolve fully this gap. Furthermore, an abstract machine for the language itself would be more efficient than the abstract machine for the meta-language.

As examples, we instantiated this meta-interpreter with different languages. The main ones are an imperative language with mutable state, and an extended lambda-calculus with features (pairs, fix-point recursion, etc.). The specialized interpreters, as well as the rest of this work, are available online [9].

As an example, we detail the process to create an interpreter for the λ-calculus language defined in Chapter 8, copied below.

```
type ident

type lterm =
| Lam (ident, lterm)
| Var ident
| App (lterm, lterm)
type clos =
| Clos (ident, lterm, env)

type env

val extEnv : (env, ident, clos) → env
val getEnv : (ident, env) → clos
```

```
val eval (s : env) (l : lterm) : clos =
branch
  let Lam (x, t) = l in
  Clos (x, t, s)
or
  let Var x = l in
  getEnv (x, s)
or
  let App (t1, t2) = l in
  let Clos (x, t, s1) = eval s t1 in
  let w = eval s t2 in
  let s2 = extEnv (s1, x, w) in
  eval s2 t
end
```

Figure 12.1: Skeletal Semantics for Call-by-Value λ-calculus

The first step for the user is to write its own language, e.g., the λ-calculus semantics above

(Figure 12.1). Once the language is formalized in skeletal semantics, our tool can automatically generate an OCaml module. There is no need to open it as the content is equivalent to the Skel definition, and contains for instance the following lines.

```
let base_type = "clos" :: ("env" :: ("ident" :: ("lterm" :: [])))
let constructor = "App" :: ("Lam" :: ("Var" :: ("Clos" :: [])))
let unspec_term = "extEnv" :: ("getEnv" :: [])
let spec_term = "eval" :: []
...
```

As can be seen, this file is a crude translation from Skel to Coq to OCaml, and as such is not meant to be easily readable by humans. Every piece of the language is transformed into an OCaml value, including types, rules, skeletons, etc.

This module is almost enough to instantiate the generic deterministic abstract machine presented in Section 11.2. Now, we need to create a new OCaml file to provide the missing pieces. For unspecified types, we create an OCaml type representing basic concrete values (see Section 8.2). For the λ-calculus, we need to cover the two unspecified types `ident` and `env`.

```
type basevalue =
  | Vident of string
  | Venv of (string * cvalue) list
```

We choose to represent variables using OCaml strings, and environments using OCaml lists.

For unspecified skelterms (`extEnv` and `getEnv`), we define OCaml functions implementing the desired behavior.

```
let function_extenv = function
  | Cval_tuple ([Cval_base (Venv st); Cval_base (Vident x); v])::[] ->
      [Cval_base (Venv ((x,v)::st))]
  | _ -> []
```

Here, for `extEnv`, we check that the function is given a single argument, which is a triple composed of an environment, a variable name, and a cvalue to write. If something is off, the implementation fails by returning an empty list, which would cause the abstract machine to backtrack. Else, we return an extended environment, by adding the pair (x,v) to the beginning of the list. We also write a similar function for `getEnv`.

Now we have all the definitions required to instantiate the abstract machine. It can then be used to execute programs. We can for instance write the following OCaml code.

```
(* shortcuts for readability *)
let xlam x t = Cval_constructor ("Lam", Cval_tuple [Cval_base (Vident x); t])
let xapp t1 t2 = Cval_constructor ("App", Cval_tuple [t1; t2])
let xvar x = Cval_constructor ("Var", Cval_base (Vident x))
let emptyenv = Cval_base (Venv [])
```

```
let _ =
  (* (λx. x x) (λy. y) *)
  let lambdaterm = xapp (xlam "x" (xapp (xvar "x") (xvar "x")))
                        (xlam "y" (xvar "y"))
  let sk = ... in
  let amstate = inject_skel sk  [("s", emptyenv); ("t", lambdaterm)] in
  let r = evalfuel 614 amstate in
  ...
```

In this example, `sk` is the skeleton `Apply("eval", [`s`;`t`])`, with some required type annotations.

The functions `inject_skel` and `evalfuel` are already defined in the abstract machine module: the former creates an AM state from a skeleton and an environment, while the latter launches the computation on a machine state. In the rest of the program, `r` contains the expected result, i.e., an option, containing the closure $\lambda y.y$ in an empty environment.

In this example, 614 is the smallest amount of fuel needed to perform two beta-reductions and return a result. The computation is noticeably inefficient as we simulate the evaluation of the meta-language Skel. For instance, even evaluating a λ-abstraction into a λ-closure, by following the skeleton `let Lam (x, t) = l in Clos (x, t, s)`, takes dozens of steps. The substitutions to perform (`l` at first, and `(x,t,s)` later) trigger the abstract machine to go through the environment several times. Also, the pattern-matching needs to recursively scan the term and bind the variables `x` and `t` one by one, which leads to the creation of multiple continuations and several changes of focus.

Chapter 13

Conclusion of Part II

We present two new semantics for the meta-language Skel in the form of a non-deterministic and a deterministic abstract machine. They are derived from the initial big-step semantics using a known transformation strategy called functional correspondence. A novelty of our approach is to use these classic tools (CPS transformation and defunctionalization) at the meta-level. This yields a generic abstract machine than can be proved sound once and for all, independently of the input language.

We implement the machines in the Coq proof assistant to certify their soundness. Using previous tools and the Coq extraction mechanism, we can automatically generate a certified OCaml interpreter for the deterministic abstract machine specialized for any skeletal semantics. This can be used as a certified interpreter for any language written using skeletal semantics. We summarize the transformations needed to reach our goal in Figure 13.1.

As a future work, we would like to create a deterministic abstract machine in the form of a breadth-first search of all possible behaviors. Unlike the one presented in this document, it would not risk being stuck in a loop, ensuring it to eventually find a result if there exists one. Also, the certified interpreter could be transformed into a debugger, to make use of the certification of the evaluation in a setting where the drawback of inefficiency is not decisive.

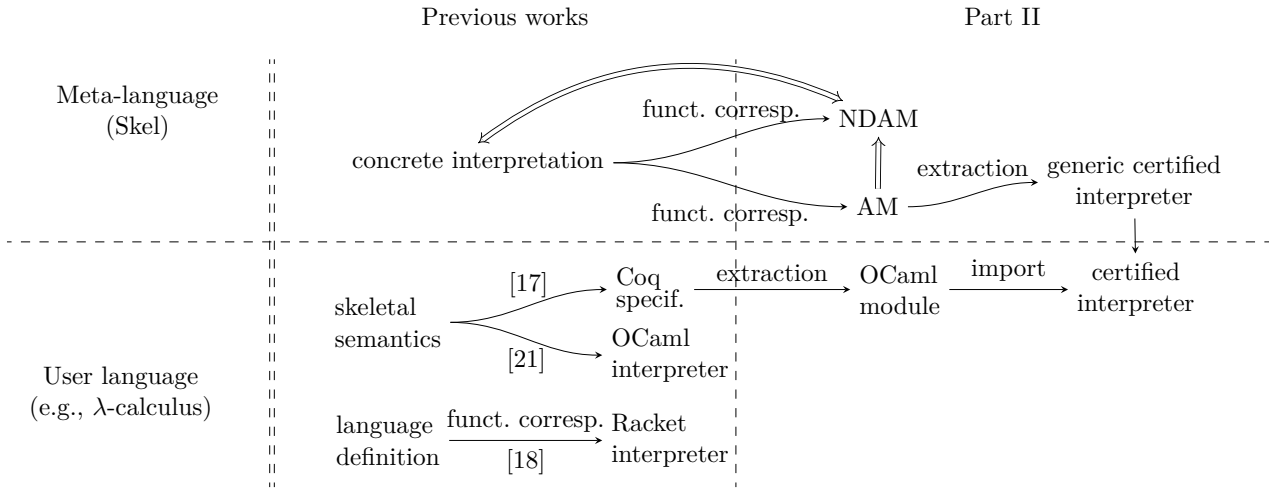

Figure 13.1: Summary of Part II and comparison with related work

Related Work

The technique of functional correspondence has been applied by hand to many languages with many different features. Automatic application of the technique is only very recent. Buszka and Biernacki [18] present an algorithm and a tool to automatically generate abstract machines from evaluators, based on the same transformations with a more complex static analysis. We probably could have used their tool to create the NDAM, but the AM would have required the introduction of backtracking by hand. Their approach is general, in the sense that they can automatically generate an abstract machine for any language. On our side, we created a single AM for the meta-language Skel, that can then be automatically specialized for any language. They did not certify their tool, while we prove the soundness of our AM in Coq.

In [21], the authors propose a tool to automatically generate a shallow embedding of a skeletal semantics into an OCaml interpreter. Skel types are directly translated into OCaml types, and skelterms into OCaml functions. Because the meta-language Skel is completely transparent, the interpreter is at the level of the language we are interested in, and more intuitive to use. However, unlike our approach, their ad-hoc translation is not certified and offers no guarantees on the behavior of the interpreter.

Conclusion

Conclusion

As presented before, different operational semantics styles have different benefits. For instance, the simplicity and readability of big-step semantics make it the preferred style for defining programming languages, but small-step semantics are better suited to reason about non-terminating or stuck computations. Formalizing the same language multiple times by hand is not a reasonable solution, as it requires a great deal of work and cross-reviewing. We address this issue with the development of interderivation techniques, allowing the use of a central definition and the systematic derivation of other formats when needed. Skeletal semantics is a suitable framework for defining and implementing such interderivations, as it offers an easy manipulation of language semantics as data structures. Regrouping different transformation techniques under the same setting also allows us to compare them and, hopefully, compose them into more complex translations.

Our first contribution is a novel generic and automatic translation of a big-step skeletal semantics into an equivalent small-step skeletal semantics. The complexity of the transformation lies in the creation of new constructors to allow the reconstruction of partially evaluated terms. This method is implemented in the Necro toolbox for the manipulation of skeletal semantics. We prove on paper that the core of the transformation (without constructor reuse) is correct. We also ensure, in practice, that the resulting small-step semantics corresponds to the big-step one by producing for any input language an equivalence certificate verifiable in Coq. This generic automatic transformation from big-step to small-step, with the corresponding proofs, is submitted for publication.

Our second contribution is the application of known interderivation techniques to the meta-language Skel itself. We use functional correspondence to obtain an equivalent non-deterministic abstract machine, and a slightly modified approach to create a deterministic version. We certify in Coq that this last semantics is indeed correct with respect to the initial (big-step) concrete interpretation of Skel. While the translation techniques used are already established, applying them to the meta-level indirectly benefits any language written in skeletal semantics. Similarly, the Coq correctness proof only has to be done once. We extract an executable OCaml interpreter for Skel from the abstract machine, which can be instantiated with any language. These abstract machines, their derivations, as well as the certified interpreter, have been presented to the CPP 2022 conference [6].

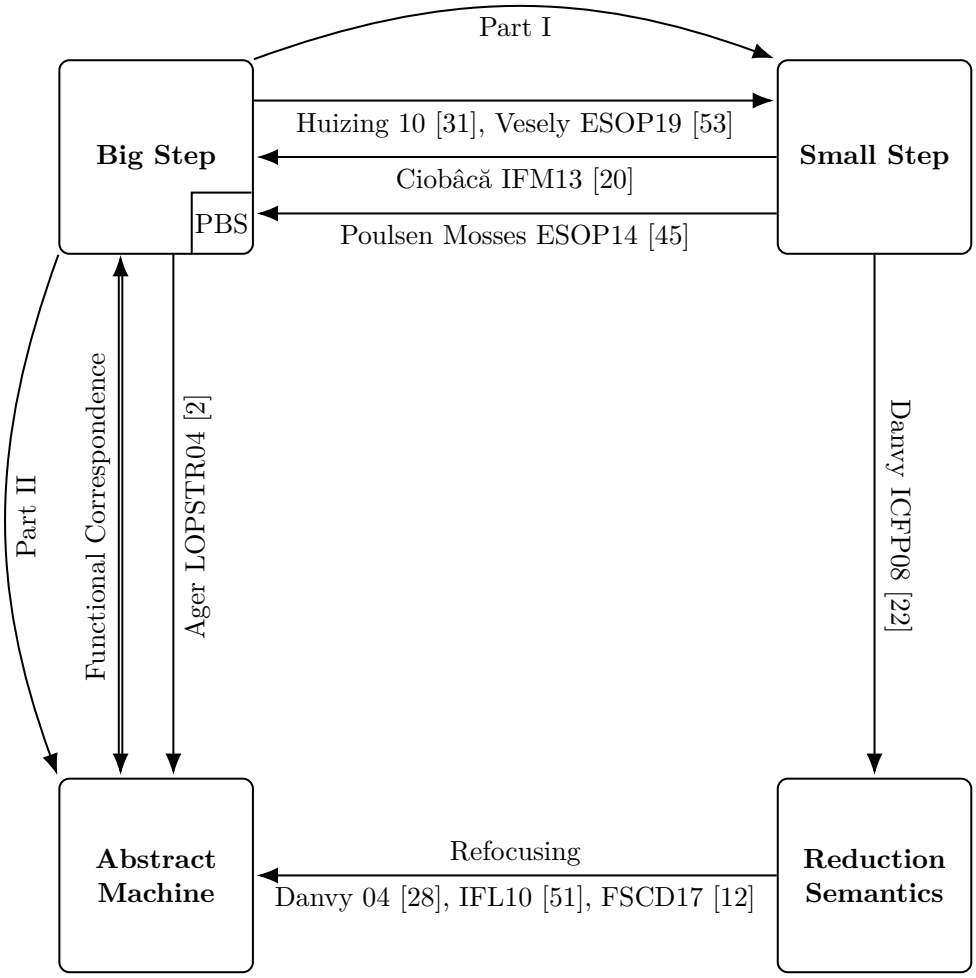

Figure 13.2: Related Work on Interderivation of Operational Semantics

Future Work

As discussed in Chapter 7 and 13, the contributions presented in this document can also be expanded. For the big-step to small-step transformation, we would like to generalize the translation to polymorphic functions, and support the use of monads by generating, when able, a monadic small-step skeletal semantics. For the deterministic abstract machine, we could generate a breadth-first version that would not fail on infinite loops. Also, it would be interesting to combine the certified interpreter with other tools to generate trustworthy (albeit slow) debuggers.

Our main objective is to expand and develop the interderivation of different semantic formats, both for the meta-language Skel and to provide as tools to user-defined languages. Notably, we plan to cover the transformations missing from Figure 13.2, but also the own already known,

which are defined so far at various degrees of precision. One is presented on an example only [22], others are defined on a meta-language—either previously known [45] or somewhat ad-hoc [2, 20, 12, 18]—, and only one has been formalized in a proof assistant [12].

Using the same framework to express them all would allow to combine them, and see if different translation paths generate similar results. The restrictions on the input semantics would also be expressed in the same framework. For example, the functional correspondence transformation from abstract machines to big-step assumes the input machine to be in defunctionalized form, a criterion we could try to characterize at the level of the input skeletal semantics.

It would also allow for comparisons between transformations. For instance, the current transformations from small-step to big-step [20, 45] have restrictions expressed w.r.t. their respective meta-language. It is not clear if one is more expressive than the other, or how they rule out the small-step semantics they cannot cover, such as the ones with interleaving. A unified framework would make such expressiveness comparisons possible.

BIBLIOGRAPHY

[1] Beniamino Accattoli, "The Complexity of Abstract Machines", *in*: *Proceedings Third International Workshop on Rewriting Techniques for Program Transformations and Evaluation, WPTE@FSCD 2016, Porto, Portugal, 23rd June 2016*, ed. by Horatiu Cirstea and Santiago Escobar, vol. 235, EPTCS, 2016, pp. 1–15, URL: https://doi.org/10.4204/EPTCS.235.1.

[2] Mads Sig Ager, "From Natural Semantics to Abstract Machines", *in*: *Logic Based Program Synthesis and Transformation, 14th International Symposium, LOPSTR 2004, Verona, Italy, August 26-28, 2004, Revised Selected Papers*, ed. by Sandro Etalle, vol. 3573, Lecture Notes in Computer Science, Springer, 2004, pp. 245–261, ISBN: 3-540-26655-0, URL: https://doi.org/10.1007/11506676_16.

[3] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard, "A functional correspondence between call-by-need evaluators and lazy abstract machines", *in*: *Inf. Process. Lett.* 90.*5* (2004), pp. 223–232, URL: https://doi.org/10.1016/j.ipl.2004.02.012.

[4] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard, "A functional correspondence between monadic evaluators and abstract machines for languages with computational effects", *in*: *Theor. Comput. Sci.* 342.*1* (2005), pp. 149–172, URL: https://doi.org/10.1016/j.tcs.2005.06.008.

[5] Mads Sig Ager et al., "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, 2003, pp. 8–19, URL: https://doi.org/10.1145/888251.888254.

[6] Guillaume Ambal, Sergueï Lenglet, and Alan Schmitt, "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*, ed. by Andrei Popescu and Steve Zdancewic, ACM, 2022, pp. 55–67, URL: https://doi.org/10.1145/3497775.3503676.

[7] Guillaume Ambal, Sergueï Lenglet, and Alan Schmitt, "HOπ in Coq", *in*: *J. Autom. Reason.* 65.*1* (2021), pp. 75–124, URL: https://doi.org/10.1007/s10817-020-09553-0.

[8] Guillaume Ambal, Sergueï Lenglet, and Alan Schmitt, *Implementation of "Automatic Transformation of a Big-Step Skeletal Semantics into Small-Step"*, 2020, URL: https://gitlab.inria.fr/skeletons/necro/-/tree/RRsmallstep.

[9] Guillaume Ambal, Sergueï Lenglet, and Alan Schmitt, *Implementation of "Certified Abstract Machines for Skeletal Semantics"*, 2021, URL: https://gitlab.inria.fr/skeletons/necro-coq/-/tree/CPP2022.

[10] Guillaume Ambal et al., "Certified Derivation of Small-Step From Big-Step Skeletal Semantics", *in*: *PPDP 2022: 24th International Symposium on Principles and Practice of Declarative Programming, Tbilisi, Georgia, September 20 - 22, 2022*, ACM, 2022, 11:1–11:48, URL: https://doi.org/10.1145/3551357.3551384.

[11] Malgorzata Biernacka, Dariusz Biernacki, and Olivier Danvy, "An Operational Foundation for Delimited Continuations in the CPS Hierarchy", *in*: *Log. Methods Comput. Sci.* 1.*2* (2005), URL: https://doi.org/10.2168/LMCS-1(2:5)2005.

[12] Malgorzata Biernacka, Witold Charatonik, and Klara Zielinska, "Generalized Refocusing: From Hybrid Strategies to Abstract Machines", *in*: *2nd International Conference on Formal Structures for Computation and Deduction, FSCD 2017, September 3-9, 2017, Oxford, UK*, ed. by Dale Miller, vol. 84, LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017, 10:1–10:17, ISBN: 978-3-95977-047-7, URL: https://doi.org/10.4230/LIPIcs.FSCD.2017.10.

[13] Malgorzata Biernacka et al., "An Abstract Machine for Strong Call by Value", *in*: *Programming Languages and Systems - 18th Asian Symposium, APLAS 2020, Fukuoka, Japan, November 30 - December 2, 2020, Proceedings*, ed. by Bruno C. d. S. Oliveira, vol. 12470, Lecture Notes in Computer Science, Springer, 2020, pp. 147–166, URL: https://doi.org/10.1007/978-3-030-64437-6_8.

[14] Dariusz Biernacki and Olivier Danvy, "From Interpreter to Logic Engine by Defunctionalization", *in*: *Logic Based Program Synthesis and Transformation, 13th International Symposium LOPSTR 2003, Uppsala, Sweden, August 25-27, 2003, Revised Selected Papers*, ed. by Maurice Bruynooghe, vol. 3018, Lecture Notes in Computer Science, Springer, 2003, pp. 143–159, URL: https://doi.org/10.1007/978-3-540-25938-1_13.

[15] Dariusz Biernacki, Olivier Danvy, and Kevin Millikin, "A Dynamic Continuation-Passing Style for Dynamic Delimited Continuations", *in*: *ACM Trans. Program. Lang. Syst.* 38.*1* (2015), 2:1–2:25, URL: https://doi.org/10.1145/2794078.

[16] Martin Bodin et al., "A trusted mechanised JavaScript specification", *in*: *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, ed. by Suresh Jagannathan and Peter Sewell, ACM, 2014, pp. 87–100, URL: https://doi.org/10.1145/2535838.2535876.

[17] Martin Bodin et al., "Skeletal semantics and their interpretations", *in*: *Proc. ACM Program. Lang.* 3.*POPL* (2019), 44:1–44:31, URL: https://doi.org/10.1145/3290357.

[18] Maciej Buszka and Dariusz Biernacki, "Automating the Functional Correspondence between Higher-Order Evaluators and Abstract Machines", *in*: *LOPSTR 2021*, vol. abs/2108.07132, 2021, arXiv: 2108.07132, URL: https://arxiv.org/abs/2108.07132.

[19] Arthur Charguéraud, "Pretty-Big-Step Semantics", *in*: *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, ed. by Matthias Felleisen and Philippa Gardner, vol. 7792, Lecture Notes in Computer Science, Springer, 2013, pp. 41–60, ISBN: 978-3-642-37035-9, URL: https://doi.org/10.1007/978-3-642-37036-6_3.

[20] Ştefan Ciobâcă, "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*, ed. by Einar Broch Johnsen and Luigia Petre, vol. 7940, Lecture Notes in Computer Science, Springer, 2013, pp. 347–361, ISBN: 978-3-642-38612-1, URL: https://doi.org/10.1007/978-3-642-38613-8_24.

[21] Nathanaël Courant, Enzo Crance, and Alan Schmitt, "Necro: Animating Skeletons", *in*: *ML 2019*, Berlin, Germany, Aug. 2019.

[22] Olivier Danvy, "Defunctionalized interpreters for programming languages", *in*: *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, ed. by James Hook and Peter Thiemann, ACM, 2008, pp. 131–142, ISBN: 978-1-59593-919-7, URL: https://doi.org/10.1145/1411204.1411206.

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

[24] Olivier Danvy and Andrzej Filinski, "Abstracting Control", *in*: *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, LFP 1990, Nice, France, 27-29 June 1990*, ACM, 1990, pp. 151–160, URL: https://doi.org/10.1145/91556.91622.

[25] Olivier Danvy and Jacob Johannsen, "Inter-deriving semantic artifacts for object-oriented programming", *in*: *J. Comput. Syst. Sci.* 76.*5* (2010), pp. 302–323, URL: https://doi.org/10.1016/j.jcss.2009.10.004.

[26] Olivier Danvy, Jacob Johannsen, and Ian Zerny, "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*, ed. by Siau-Cheng Khoo and Jeremy G. Siek, ACM, 2011, pp. 1–12, URL: https://doi.org/10.1145/1929501.1929503.

[27] Olivier Danvy and Kevin Millikin, "A Rational Deconstruction of Landin's SECD Machine with the J Operator", *in*: *Log. Methods Comput. Sci.* 4.*4* (2008), URL: https://doi.org/10.2168/LMCS-4(4:12)2008.

[28] Olivier Danvy and Lasse R. Nielsen, "Refocusing in Reduction Semantics", *in*: vol. 11, 2004, URL: https://doi.org/10.7146/brics.v11i26.21851.

[29] ECMA International, ed., *ECMAScript language specification. Standard ECMA-262*, 2021, URL: https://262.ecma-international.org/.

[30] C. A. R. Hoare, "An Axiomatic Basis for Computer Programming", *in*: *Commun. ACM* 12.*10* (1969), pp. 576–580, URL: https://doi.org/10.1145/363235.363259.

[31] Cornelis Huizing, Ron Koymans, and Ruurd Kuiper, "A Small Step for Mankind", *in*: *Concurrency, Compositionality, and Correctness, Essays in Honor of Willem-Paul de Roever*, ed. by Dennis Dams, Ulrich Hannemann, and Martin Steffen, vol. 5930, Lecture Notes in Computer Science, Springer, 2010, pp. 66–73, ISBN: 978-3-642-11511-0, URL: https://doi.org/10.1007/978-3-642-11512-7_5.

[32] Wojciech Jedynak, Malgorzata Biernacka, and Dariusz Biernacki, "An operational foundation for the tactic language of Coq", *in*: *15th International Symposium on Principles and Practice of Declarative Programming, PPDP '13, Madrid, Spain, September 16-18, 2013*, ed. by Ricardo Peña and Tom Schrijvers, ACM, 2013, pp. 25–36, URL: https://doi.org/10.1145/2505879.2505890.

[33] Ramana Kumar et al., "CakeML: a verified implementation of ML", *in*: *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, ed. by Suresh Jagannathan and Peter Sewell, ACM, 2014, pp. 179–192, URL: https://doi.org/10.1145/2535838.2535841.

[34] Xavier Leroy, "Formal verification of a realistic compiler", *in*: *Commun. ACM* 52.*7* (2009), pp. 107–115, URL: https://doi.org/10.1145/1538788.1538814.

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

[36] Xavier Leroy et al., *The OCaml system: Documentation and user's manual*, version 4.12, INRIA, 2021, URL: https://ocaml.org/manual/.

[37] Pierre Letouzey, "A New Extraction for Coq", *in*: *Types for Proofs and Programs, Second International Workshop, TYPES 2002, Berg en Dal, The Netherlands, April 24-28, 2002, Selected Papers*, ed. by Herman Geuvers and Freek Wiedijk, vol. 2646, Lecture Notes in Computer Science, Springer, 2002, pp. 200–219, URL: https://doi.org/10.1007/3-540-39185-1_12.

[38] Robin Milner et al., *The Definition of Standard ML, Revised Edition*, May 1997, ISBN: 978-0-2626-3181-5.

[39] Wolfgang Naraschewski and Tobias Nipkow, *Isabelle/HOL*, 2020, URL: http://www.cl.cam.ac.uk/research/hvg/Isabelle/.

[40] Hanne Riis Nielson and Flemming Nielson, *Semantics with applications - a formal introduction*, Wiley professional computing, Wiley, 1992, ISBN: 978-0-471-92980-2.

[41] Hanne Riis Nielson and Flemming Nielson, *Semantics with Applications: An Appetizer*, Undergraduate Topics in Computer Science, Springer, 2007, ISBN: 978-1-84628-691-9, URL: https://doi.org/10.1007/978-1-84628-692-6.

[42] Scott Owens et al., "Lem: A Lightweight Tool for Heavyweight Semantics", *in*: *Interactive Theorem Proving - Second International Conference, ITP 2011, Berg en Dal, The Netherlands, August 22-25, 2011. Proceedings*, ed. by Marko C. J. D. van Eekelen et al., vol. 6898, Lecture Notes in Computer Science, Springer, 2011, pp. 363–369, URL: https://doi.org/10.1007/978-3-642-22863-6%5C_27.

[43] Maciej Piróg and Dariusz Biernacki, "A systematic derivation of the STG machine verified in Coq", *in*: *Proceedings of the 3rd ACM SIGPLAN Symposium on Haskell, Haskell 2010, Baltimore, MD, USA, 30 September 2010*, ed. by Jeremy Gibbons, ACM, 2010, pp. 25–36, URL: https://doi.org/10.1145/1863523.1863528.

[44] Gordon D. Plotkin, "Call-by-Name, Call-by-Value and the lambda-Calculus", *in*: *Theor. Comput. Sci.* 1.*2* (1975), pp. 125–159, URL: https://doi.org/10.1016/0304-3975(75)90017-1.

[45] Casper Bach Poulsen and Peter D. Mosses, "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*, ed. by Zhong Shao, vol. 8410, Lecture Notes in Computer Science, Springer, 2014, pp. 270–289, ISBN: 978-3-642-54832-1, URL: https://doi.org/10.1007/978-3-642-54833-8_15.

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

[47] John C. Reynolds, "Definitional interpreters for higher-order programming languages", *in*: *Proceedings of the ACM annual conference, ACM 1972, 1972, Volume 2*, ed. by John J. Donovan and Rosemary Shields, ACM, 1972, pp. 717–740, URL: https://doi.org/10.1145/800194.805852.

[48] Grigore Rosu and Traian-Florin Serbanuta, "An overview of the K semantic framework", *in*: *J. Log. Algebraic Methods Program.* 79.*6* (2010), pp. 397–434, URL: https://doi.org/10.1016/j.jlap.2010.03.012.

[49] Dana S. Scott and Christopher S. Strachey, "Toward a mathematical semantics for computer languages", *in*: 1971.

[50] Peter Sewell et al., "Ott: Effective tool support for the working semanticist", *in*: *J. Funct. Program.* 20.*1* (2010), pp. 71–122, URL: https://doi.org/10.1017/S0956796809990293.

[51] Filip Sieczkowski, Malgorzata Biernacka, and Dariusz Biernacki, "Automating Derivations of Abstract Machines from Reduction Semantics: - A Generic Formalization of Refocusing in Coq", *in*: *Implementation and Application of Functional Languages - 22nd International Symposium, IFL 2010, Alphen aan den Rijn, The Netherlands, September 1-3, 2010, Revised Selected Papers*, ed. by Jurriaan Hage and Marco T. Morazán, vol. 6647, Lecture Notes in Computer Science, Springer, 2010, pp. 72–88, URL: https://doi.org/10.1007/978-3-642-24276-2_5.

[52] The Coq Development Team, *The Coq Proof Assistant*, version 8.13, Jan. 2021, URL: https://doi.org/10.5281/zenodo.4501022.

[53] Ferdinand Vesely and Kathleen Fisher, "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*, ed. by Luís Caires, vol. 11423, Lecture Notes in Computer Science, Springer, 2019, pp. 205–231, ISBN: 978-3-030-17183-4, URL: https://doi.org/10.1007/978-3-030-17184-1_8.

Appendix A

Functional Correspondence on IMP

A.1 Syntax and Big-Step Semantics

$$a ::= x \mid n \mid \texttt{Plus}(a_1, a_2)$$

$$e ::= b \mid \texttt{Equal}(a_1, a_2) \mid \texttt{And}(e_1, e_2) \mid \texttt{Neg}(e)$$

$$c ::= \texttt{skip} \mid \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \mid \texttt{while}\ e\ \texttt{do}\ c \mid c_1\ ;\ c_2 \mid x := a$$

```
let eval_a (s : state) (a : aexp) : nat =
match a with
| Var(x) -> lookup s x
| Nat(n) -> n
| Plus(a1, a2) ->
  let n1 = eval_a s a1 in
  let n2 = eval_a s a2 in
  n1 + n2

let eval_b (s : state) (e : bexp) : bool =
match e with
| Bool(b) -> b
| Equal(a1, a2) ->
  let n1 = eval_a s a1 in
  let n2 = eval_a s a2 in
  n1 == n2
| And(e1, e2) ->
  let b1 = eval_b s e1 in
  let b2 = eval_b s e2 in
  b1 && b2
| Neg(e') ->
  let b = eval_b s e' in
  not b

let eval_c (s : state) (c : cexp) : state =
match c with
```

```
| Skip -> s
| If(e, c1, c2) ->
  let b = eval_b s e in
  match b with
  | true -> eval_c s c1
  | false -> eval_c s c2
| While(e, c) ->
  let b = eval_b s e in
  match b with
  | true ->
    let s' = eval_c s c in
    eval_c s' While(e, c)
  | false -> s
| Seq(c1, c2) ->
  let s' = eval_c s c1 in
  eval_c s' c2
| Assign(x, a) ->
  let n = eval_a s a in
  extend s x n
```

A.2 CPS Transform

```
let eval_a s a (k : nat -> state) : state =
match a with
| Var(x) -> k (lookup s x)
| Nat(n) -> k n
| Plus(a1, a2) ->
  eval_a s a1 (fun n1 ->
  eval_a s a2 (fun n2 ->
  k (n1+n2)))

let eval_b s e (k : bool -> state) : state =
match e with
| Bool(b) -> k b
| Equal(a1, a2) ->
  eval_a s a1 (fun n1 ->
  eval_a s a2 (fun n2 ->
  k (n1 == n2)))
| And(e1, e2) ->
  eval_b s e1 (fun b1 ->
  eval_b s e2 (fun b2 ->
  k (b1 && b2)))
| Neg(e') ->
  eval_b s e' (fun b ->
  k (not b))
```

```
let eval_c s c (k : state -> state) : state =
match c with
| Skip -> k s
| If(e, c1, c2) ->
  eval_b s e (fun b ->
  match b with
  | true -> eval_c s c1 k
  | false -> eval_c s c2 k)
| While(e, c) ->
  eval_b s e (fun b ->
  match b with
  | true ->
    eval_c s c (fun s' ->
    eval_c s' While(e, c) k)
  | false -> k s)
| Seq(c1, c2) ->
  eval_c s c1 (fun s' ->
  eval_c s' c2 k)
| Assign(x, a) ->
  eval_a s a (fun n ->
  k (extend s x n))
```

A.3 Defunctionalization

```
type kat =
| KAID
| KAPlus1 of state * aexp * kat
| KAPlus2 of nat * kat
| KAEqual1 of state * aexp * kbt
| KAEqual2 of nat * kbt
| KAAssign of state * var * kct

let disp_ka (k : kat) (n : nat) : state =
match k with
| KAPlus1(s, a2, k') ->
  eval_a s a2 KAPlus2(n, k')
| KAPlus2(n1, k') ->
  disp_ka k' (n1+n)
| KAEqual1(s, a2, k') ->
  eval_a s a2 KAEqual2(n, k')
| KAEqual2(n1, k') ->
  disp_kb k' (n1 == n)
| KAAssign(s, x, k') ->
  disp_kc k' (extend s x n)
```

```
let eval_a s a (k : kat) : state =
match a with
| Var(x) -> disp_ka k (lookup s x)
| Nat(n) -> disp_ka k n
| Plus(a1, a2) ->
  eval_a s a1 KAPlus1(s, a2, k)

type kbt =
| KBID
| KBAnd1 of state * bexp * kbt
| KBAnd2 of bool * kbt
| KBNeg of kbt
| KBIf of state * cexp * cexp * kct
| KBWhile of state * cexp * bexp * kct

let disp_kb (k : kbt) (b : bool) : state =
match k with
| KBAnd1(s, e2, k') ->
  eval_b s e2 KBAnd2(b, k')
| KBAnd2(b1, k') ->
  disp_kb k' (b1 && b)
| KBNeg(k') ->
  disp_kb k' (not b)
| KBIf(s, c1, c2, k') ->
  match b with
  | true -> eval_c s c1 k'
  | false -> eval_c s c2 k'
| KBWhile(s, c, e, k') ->
  match b with
  | true ->
    eval_c s c KCWhile(e, c, k')
  | false -> disp_kb k' s

let eval_b s e (k : kbt) : state =
match e with
| Bool(b) -> disp_kb k b
| Equal(a1, a2) ->
  eval_a s a1 KAEqual1(s, a2, k)
| And(e1, e2) ->
  eval_b s e1 KBAnd1(s, e2, k)
| Neg(e') ->
  eval_b s e' KBNeg(k)

type kct =
| KCID
| KCWhile of bexp * cexp * kct
```

```
| KCSeq of cexp * kct

let disp_kc (k : kct) (s : state) : state =
match k with
| KCWhile(e, c, k') ->
  eval_c s While(e, c) k'
| KCSeq(c, k') ->
  eval_c s c k'

let eval_c s c (k : kct) : state =
match c with
| Skip -> disp_kc k s
| If(e, c1, c2) ->
  eval_b s e KBIf(s, c1, c2, k)
| While(e, c) ->
  eval_b s e KBWhile(s, c, e, k)
| Seq(c1, c2) ->
  eval_c s c1 KCSeq(c2, k)
| Assign(x, a) ->
  eval_a s a KAAssign(s, x, k)
```

Yes, `KCWhile(e,c,k)` is equivalent to `KCSeq(While(e,c),k)`. The final semantics could be simplified.

A.4 Abstract Machine

Mode **ka**:

$$\langle \mathrm{ka}_{\mathrm{id}}, n\rangle_{\mathrm{ka}} \not\to \texttt{(* end of computation *)}$$
$$\langle \texttt{KAPlus1}(\sigma, a, k), n\rangle_{\mathrm{ka}} \to \langle \sigma, a, \texttt{KAPlus2}(n, k)\rangle_{\mathrm{a}}$$
$$\langle \texttt{KAPlus2}(n_1, k), n\rangle_{\mathrm{ka}} \to \langle k, (n_1 + n)\rangle_{\mathrm{ka}}$$
$$\langle \texttt{KAEqual1}(\sigma, a, k), n\rangle_{\mathrm{ka}} \to \langle \sigma, a, \texttt{KAEqual2}(n, k)\rangle_{\mathrm{a}}$$
$$\langle \texttt{KAEqual2}(n_1, k), n\rangle_{\mathrm{ka}} \to \langle k, (n_1 \stackrel{?}{=} n)\rangle_{\mathrm{kb}}$$
$$\langle \texttt{KAAssign}(\sigma, x, k), n\rangle_{\mathrm{ka}} \to \langle k, \sigma[x \mapsto n]\rangle_{\mathrm{kc}}$$

Mode **a**:

$$\langle \sigma, x, k\rangle_{\mathrm{a}} \to \langle k, \sigma(x)\rangle_{\mathrm{ka}}$$
$$\langle \sigma, n, k\rangle_{\mathrm{a}} \to \langle k, n\rangle_{\mathrm{ka}}$$
$$\langle \sigma, \texttt{Plus}(a_1, a_2), k\rangle_{\mathrm{a}} \to \langle \sigma, a_1, \texttt{KAPlus1}(\sigma, a_2, k)\rangle_{\mathrm{a}}$$

Mode **kb**:

$$\langle \mathrm{kb}_{\mathrm{id}}, b\rangle_{\mathrm{kb}} \not\to \texttt{(* end of computation *)}$$
$$\langle \texttt{KBAnd1}(\sigma, e, k), b\rangle_{\mathrm{kb}} \to \langle \sigma, e, \texttt{KBAnd2}(b, k)\rangle_{\mathrm{b}}$$
$$\langle \texttt{KBAnd2}(b_1, k), b\rangle_{\mathrm{kb}} \to \langle k, (b_1 \wedge b)\rangle_{\mathrm{kb}}$$
$$\langle \texttt{KBNeg}(k), b\rangle_{\mathrm{kb}} \to \langle k, \neg b\rangle_{\mathrm{kb}}$$
$$\langle \texttt{KBIf}(\sigma, c_1, c_2, k), \top\rangle_{\mathrm{kb}} \to \langle \sigma, c_1, k\rangle_{\mathrm{c}}$$
$$\langle \texttt{KBIf}(\sigma, c_1, c_2, k), \bot\rangle_{\mathrm{kb}} \to \langle \sigma, c_2, k\rangle_{\mathrm{c}}$$
$$\langle \texttt{KBWhile}(\sigma, c, e, k), \top\rangle_{\mathrm{kb}} \to \langle \sigma, c, \texttt{KCWhile}(e, c, k)\rangle_{\mathrm{c}}$$
$$\langle \texttt{KBWhile}(\sigma, c, e, k), \bot\rangle_{\mathrm{kb}} \to \langle k, \sigma\rangle_{\mathrm{kc}}$$

Mode **b**:

$$\langle \sigma, b, k\rangle_{\mathrm{b}} \to \langle k, b\rangle_{\mathrm{kb}}$$
$$\langle \sigma, \texttt{Equal}(a_1, a_2), k\rangle_{\mathrm{b}} \to \langle \sigma, a_1, \texttt{KAEqual1}(\sigma, a_2, k)\rangle_{\mathrm{a}}$$
$$\langle \sigma, \texttt{And}(e_1, e_2), k\rangle_{\mathrm{b}} \to \langle \sigma, e_1, \texttt{KBAnd1}(\sigma, e_2, k)\rangle_{\mathrm{b}}$$
$$\langle \sigma, \texttt{Neg}(e), k\rangle_{\mathrm{b}} \to \langle \sigma, e, \texttt{KBNeg}(k)\rangle_{\mathrm{b}}$$

Mode **kc**:

$$\langle \mathrm{kc}_{\mathrm{id}}, \sigma\rangle_{\mathrm{kc}} \not\to \texttt{(* end of computation *)}$$
$$\langle \texttt{KCWhile}(e, c, k), \sigma\rangle_{\mathrm{kc}} \to \langle \sigma, \texttt{while}\ e\ \texttt{do}\ c, k\rangle_{\mathrm{c}}$$
$$\langle \texttt{KCSeq}(c, k), \sigma\rangle_{\mathrm{kc}} \to \langle \sigma, c, k\rangle_{\mathrm{c}}$$

Mode **c**:

$$\langle \sigma, \texttt{skip}, k\rangle_{\mathrm{c}} \to \langle k, \sigma\rangle_{\mathrm{kc}}$$
$$\langle \sigma, \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2, k\rangle_{\mathrm{c}} \to \langle \sigma, e, \texttt{KBIf}(\sigma, c_1, c_2, k)\rangle_{\mathrm{b}}$$
$$\langle \sigma, \texttt{while}\ e\ \texttt{do}\ c, k\rangle_{\mathrm{c}} \to \langle \sigma, e, \texttt{KBWhile}(\sigma, c, e, k)\rangle_{\mathrm{b}}$$
$$\langle \sigma, c_1\ ;\ c_2, k\rangle_{\mathrm{c}} \to \langle \sigma, c_1, \texttt{KCSeq}(c_2, k)\rangle_{\mathrm{c}}$$
$$\langle \sigma, x := a, k\rangle_{\mathrm{c}} \to \langle \sigma, a, \texttt{KAAssign}(\sigma, x, k)\rangle_{\mathrm{a}}$$

Appendix B

Successive Transformations of IMP in Skeletal Semantics

B.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
```

B.2 After Adding Coercions

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

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

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

hook getRet_hexpr (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
```

B.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 *)
```

B.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
```

$$\frac{}{s, b \to s, (s, \texttt{boolToVal}(b))}$$

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

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

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

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

$$\frac{s, e_1 \to z_1, z_2}{s, \texttt{Not}(e_1) \to z_1, \texttt{Not}(z_2)}$$

$$\frac{e_1 = (s_1, v)}{s, \texttt{Not}(e_1) \to s, (s_1, \neg v)}$$

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

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

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

```
| Var x ->
  let v = read (x, s) in
  (s, Ret_hexpr (s, v))
```

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

```
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)))
```

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

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

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

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

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

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

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

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

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

```
  | 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
```

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

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

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

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

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

B.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
```

B.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))
| 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
```

$$\frac{}{s, \texttt{Not}(e_1) \to s, \texttt{Not1}(s, e_1)}$$

$$\frac{s_0, e_0 \to z_1, z_2}{s, \texttt{Not1}(s_0, e_0) \to s, \texttt{Not1}(z_1, z_2)}$$

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

$$\frac{}{s, b \to s, (s, \texttt{boolToVal}(b))}$$

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

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

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

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

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

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

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

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

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

```
| 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))
```

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

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

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

```
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
| 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
```

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

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

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

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

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

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

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

```
| 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
```

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

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

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

$$\frac{}{s, \texttt{Skip} \to s, (s)}$$

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

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

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

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

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

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

Appendix C

Proof of the Transformation

We present a pen-and-paper proof that the transformation of Chapter 4 is correct, in the simplified case where we do not perform the analysis of Section 4.2.2 and systematically create new constructors.

C.1 Definitions and Proof Structure

We recall the rules for the inductive (Figure C.1) and coinductive (Figure C.2) interpretation of skeletal semantics. The main differences with is that, for coinduction, there is no rule for returns or filters—as we do not allow filters to diverge—and there are two rules for the `LetIn` construct. Indeed, $\texttt{let}\ \widetilde{v} = K\ \texttt{in}\ S$ diverges because either K or S diverges.

The skeletal semantics (Chapter 3) as well as the different phases of the transformation (Section 4.2) are presented earlier in this document. We only redefine the last phase of the transformation (Figure C.3), as it can be simplified to not use substitutions.

We start with a few necessary definitions used in the different proofs.

$$\frac{\Sigma(\widetilde{t}) \Downarrow_h \widetilde{b}}{\Sigma \vdash \texttt{Hook}\ h\ \widetilde{t} \Downarrow \widetilde{b}} \qquad \frac{\mathcal{R}_f(\widetilde{\Sigma(t)}) \Downarrow \widetilde{b}}{\Sigma \vdash \texttt{Filter}\ f\ \widetilde{t} \Downarrow \widetilde{b}} \qquad \frac{\widetilde{\Sigma(t)} = \widetilde{b}}{\Sigma \vdash \texttt{Return}\ \widetilde{t} \Downarrow \widetilde{b}} \qquad \frac{S_i \in \widetilde{S} \qquad \Sigma \vdash S_i \Downarrow \widetilde{b}}{\Sigma \vdash \texttt{Branching}\ \widetilde{S} \Downarrow \widetilde{b}}$$

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

Figure C.1: Inductive Interpretation

$$\frac{\Sigma(\widetilde{t}) \Uparrow_h}{\Sigma \vdash \texttt{Hook}\ h\ \widetilde{t} \Uparrow} \textsc{Div-Pc} \qquad \frac{S_i \in \widetilde{S} \qquad \Sigma \vdash S_i \Uparrow}{\Sigma \vdash \texttt{Branching}\ \widetilde{S} \Uparrow} \textsc{Div-Br} \qquad \frac{\Sigma \vdash K \Uparrow}{\Sigma \vdash \texttt{let}\ \widetilde{v} = K\ \texttt{in}\ S \Uparrow} \textsc{Div-LetL}$$

$$\frac{h(\widetilde{y}, c(\widetilde{x})) := S \in R \qquad \{\widetilde{y \mapsto a}\} + \{\widetilde{x \mapsto a'}\} \vdash S \Uparrow}{(\widetilde{a}, c(\widetilde{a'})) \Uparrow_h} \textsc{Div-tuple}$$

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

Figure C.2: Coinductive Interpretation

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

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

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

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

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.

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

$$
\begin{aligned}
\| \texttt{Branching}\ (S_1, \ldots, S_n) \|^r &\triangleq \texttt{Branching}\ (\| S_1 \|^r, \ldots, \| S_n \|^r) \\
\| \texttt{let}\ \widetilde{v} = \texttt{Return}\ \widetilde{t}\ \texttt{in}\ S \|^r &\triangleq \texttt{let}\ \widetilde{v} = \texttt{Return}\ \widetilde{t}\ \texttt{in}\ \| S \|^r \\
\| \texttt{let}\ \widetilde{v} = \texttt{Filter}\ f\ \widetilde{t}\ \texttt{in}\ S \|^r &\triangleq \texttt{let}\ \widetilde{v} = \texttt{Filter}\ f\ \widetilde{t}\ \texttt{in}\ \| S \|^r \\
\| \texttt{Return}\ \widetilde{t} \|^r &\triangleq \texttt{Return}\ (\widetilde{y}, \texttt{Ret\_hr}(\widetilde{t})) \\
\| \texttt{let}\ \widetilde{v} = \texttt{Hook}\ (\texttt{New}\ c)\ h\ \widetilde{t}\ \texttt{in}\ S \|^r &\triangleq \texttt{Return}\ (\widetilde{y}, c(\widetilde{t}, \widetilde{z_c})) \\
&\quad \text{where}\ (\texttt{hr}(\widetilde{y}, c(\widetilde{w_c}, \widetilde{z_c})) := S_c) \in \mathrm{R}_{\mathrm{dist}} \\
\| \texttt{let}\ \widetilde{v} = \texttt{Hook Reuse}\ h\ (\widetilde{w'}, w)\ \texttt{in}\ S \|^r &\triangleq \texttt{Branching}(S_1, S_2) \quad \text{where}
\end{aligned}
$$

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

Figure C.3: Small-Step Transformation of a Skeleton without Analysis

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

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

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.

$$K \in \texttt{let}\ \widetilde{v} = K\ \texttt{in}\ S$$
$$S \in \texttt{let}\ \widetilde{v} = K\ \texttt{in}\ S$$
$$S_i \in \texttt{Branching}\ (S_1, \ldots, S_n)$$

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.

$$S \triangleleft \texttt{let}\ \widetilde{v} = K\ \texttt{in}\ S$$
$$S_i \triangleleft \texttt{Branching}\ (S_1, \ldots, S_n)$$

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

The proof uses the sets of rules at the different phases of the transformation to state and certify results. As explained in Chapter 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_h` constructors;

$$\mathrm{R_{EBS}} \triangleq \mathrm{R_{dist}} \cup \{h(\widetilde{y}, \texttt{Ret\_h}(\widetilde{v})) := \texttt{Return}\ \widetilde{v} \mid h \in H_0\}$$

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

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 $\widetilde{a} \Downarrow_h^{\mathrm{R_{BS}}} \widetilde{b}$ for the concrete evaluation, and $\Sigma \vdash S \Uparrow^{\mathrm{R_{BS}}}$ and $\widetilde{a} \Uparrow_h^{\mathrm{R_{BS}}}$ for the divergence of the initial big-step semantics.

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

$$\frac{\widetilde{a} \Downarrow_h^{\mathrm{R_{SS}}} \widetilde{b} \qquad \widetilde{b} \overset{\infty}{\to}_h}{\widetilde{a} \overset{\infty}{\to}_h}$$

Other equivalent definitions include:

- $\widetilde{a} \overset{\infty}{\to}_h$ is the maximal predicate Q satisfying the following property.

$$Q(\widetilde{a}, h) \quad \Longrightarrow \quad \exists \widetilde{b},\ \widetilde{a} \Downarrow_h^{\text{R}_{\text{SS}}} \widetilde{b}\ \wedge\ Q(\widetilde{b}, h)$$

- Let $F(X) = \{(\widetilde{a}, h) \mid \exists \widetilde{b},\ \widetilde{a} \Downarrow_h^{\text{R}_{\text{SS}}} \widetilde{b} \wedge (\widetilde{b}, h) \in X\}$, we have:

$$\overset{\infty}{\to} = \{(\widetilde{a}, h) \mid \widetilde{a} \overset{\infty}{\to}_h\} \quad \triangleq \quad \nu X.F(X)$$

We also recall notations from the transformation phases (Section 4.2). 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 "`New` c" annotation to every hook call);
- $\lceil S \rfloor$ for the distribution of branchings;
- $\parallel S \parallel^r$ for the final small-step transformation, redefined with simpler rules in Figure C.3.

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 C.2.1. 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 C.2.2. For all Σ_1, Σ_2, and Σ_3,

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

Lemma C.2.3. For all Σ_1, Σ_2, and Σ_3,

- $\forall x, \quad x \notin (\text{dom}(\Sigma_2) \setminus \text{dom}(\Sigma_3)) \implies (\Sigma_1 + \Sigma_3)(x) = (\Sigma_1 + \Sigma_2 + \Sigma_3)(x)$

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

Lemma C.2.4. 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 C.2.3 (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 $\texttt{Hook}\ h\ (\widetilde{t}, t)$, similarly
- If $S = \texttt{Branching}\ (\widetilde{S})$
  $\Sigma_1 + \Sigma_3 \vdash \texttt{Branching}\ (\widetilde{S}) \Downarrow \widetilde{a}$
  $\iff \exists S_i \in \widetilde{S}, \Sigma_1 + \Sigma_3 \vdash S_i \Downarrow \widetilde{a}$
  $\iff \exists S_i \in \widetilde{S}, \Sigma_1 + \Sigma_2 + \Sigma_3 \vdash S_i \Downarrow \widetilde{a}$ using induction hypothesis
  $\iff \Sigma_1 + \Sigma_2 + \Sigma_3 \vdash \texttt{Branching}\ (\widetilde{S}) \Downarrow \widetilde{a}$
- If $S = (\texttt{let}\ \widetilde{v} = K\ \texttt{in}\ S')$
  $\Sigma_1 + \Sigma_3 \vdash \texttt{let}\ \widetilde{v} = K\ \texttt{in}\ S' \Downarrow \widetilde{a}$
  $\iff \exists \widetilde{b}, \begin{cases} \Sigma_1 + \Sigma_3 \vdash K \Downarrow \widetilde{b} \\ \Sigma_1 + \Sigma_3 + \widetilde{\{v \mapsto b\}} \vdash S' \Downarrow \widetilde{a} \end{cases}$
  $\iff \exists \widetilde{b}, \begin{cases} \Sigma_1 + \Sigma_2 + \Sigma_3 \vdash K \Downarrow \widetilde{b} \\ \Sigma_1 + \Sigma_2 + \widetilde{(\Sigma_3 + \{v \mapsto b\})} \vdash S' \Downarrow \widetilde{a} \end{cases}$ using IH twice and Lemma C.2.2.
  $\mathrm{fv}(S') \subset \mathrm{fv}(S) \cup \{\widetilde{v}\}$, 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 C.2.5. 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

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

Proof. This lemma is similar to Lemma C.2.4, but for divergence. See Figure C.2 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 C.2.4 to go from one environment to the other.

□

Lemma C.2.6. For all S_1, x, S_2, and Σ, if $(\mathrm{bv}(S_1) \setminus \{\widetilde{x}\}) \cap \mathrm{fv}(S_2) = \emptyset$, then

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

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 C.2.7, 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 C.2).

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:

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

$$
\begin{aligned}
&\Sigma \vdash S_1 \Uparrow \\
&\quad \text{OR} \\
\iff & \exists \widetilde{b}, \begin{cases} \Sigma \vdash S_1 \Downarrow \widetilde{b} \\ \Sigma + \widetilde{\{x \mapsto b\}} \vdash S_2 \Uparrow \end{cases} \quad \text{by Lemma C.2.5}
\end{aligned}
$$

$\{\widetilde{v}\} \subset \text{bv}(S_1)$, so our hypothesis implies $(\{\widetilde{v}\} \setminus \{\widetilde{x}\}) \cap \text{fv}(S_2) = \emptyset$, enough to use Lemma C.2.5. □

Lemma C.2.7. For all S_1, x, S_2, Σ, and $\widetilde{a}$, if $(\text{bv}(S_1) \setminus \{\widetilde{x}\}) \cap \text{fv}(S_2) = \emptyset$, then

$$
\Sigma \vdash < S_1 \mid \widetilde{x} \mid S_2 > \Downarrow \widetilde{a} \iff \exists \widetilde{b}, \begin{cases} \Sigma \vdash S_1 \Downarrow \widetilde{b} \\ \Sigma + \widetilde{\{x \mapsto b\}} \vdash S_2 \Downarrow \widetilde{a} \end{cases}
$$

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 C.1).

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.

$$
\begin{aligned}
&\Sigma \vdash < S_1 \mid \widetilde{x} \mid S_2 > \Downarrow \widetilde{a} \\
\iff & \exists \widetilde{c}, \begin{cases} \Sigma \vdash K \Downarrow \widetilde{c} \\ \Sigma + \widetilde{\{v \mapsto c\}} \vdash < S' \mid \widetilde{x} \mid S_2 > \Downarrow \widetilde{a} \end{cases} \text{ by definition} \\
\iff & \exists \widetilde{c}, \begin{cases} \Sigma \vdash K \Downarrow \widetilde{c} \\ \exists \widetilde{b}, \begin{cases} \Sigma + \widetilde{\{v \mapsto c\}} \vdash S' \Downarrow \widetilde{b} \\ \Sigma + \widetilde{\{v \mapsto c\}} + \widetilde{\{x \mapsto b\}} \vdash S_2 \Downarrow \widetilde{a} \end{cases} \end{cases} \text{ by induction hypothesis} \\
\iff & \exists \widetilde{c}, \widetilde{b}, \begin{cases} \Sigma \vdash K \Downarrow \widetilde{c} \\ \Sigma + \widetilde{\{v \mapsto c\}} \vdash S' \Downarrow \widetilde{b} \\ \Sigma + \widetilde{\{x \mapsto b\}} \vdash S_2 \Downarrow \widetilde{a} \end{cases} \text{ by Lemma C.2.4}
\end{aligned}
$$

$\{\widetilde{v}\} \subset \text{bv}(S_1)$, 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 C.2.8. 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 $S_1 = \texttt{Branching}\ (S'_1, \ldots, S'_n)$, then $\lceil < S_1 \mid \widetilde{v} \mid S_2 > \rfloor = \texttt{Branching}\ (\lceil < S'_1 \mid \widetilde{v} \mid S_2 > \rfloor, \ldots.)$ 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 $S_1 = (\texttt{let}\ \widetilde{w} = \texttt{Branching}\ (S'_1, \ldots, S'_n)\ \texttt{in}\ S'_0)$, then $\lceil< S_1 \mid \widetilde{v} \mid S_2 >\rfloor = \lceil\texttt{let}\ \widetilde{w} = \texttt{Branching}\ (S'_1, \ldots, S'_n)\ \texttt{in}\ < S'_0 \mid \widetilde{v} \mid S_2 >\rfloor = \texttt{Branching}\ (\lceil< S'_1 \mid \widetilde{w} \mid< S'_0 \mid \widetilde{v} \mid S_2 >>\rfloor, \ldots)$.

  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 C.2.9. Forall S_1, $\widetilde{v}$, S_2, E, and K_H, where K_H is not a branching, if $E[K_H] \triangleleft \lceil< S_1 \mid \widetilde{v} \mid S_2 >\rfloor$ and $K_H \notin S_1$, then $E[K_H] \triangleleft \lceil S_2 \rfloor$.

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

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

□

C.3 SSA

Lemma C.3.1. Forall $S' \in S$, if SSA(S) then SSA(S').

Proof. By simply following Definitions 3 and 4. □

Lemma C.3.2. For all S_1, S_2, $\widetilde{v}$, if SSA($\texttt{let}\ \widetilde{v} = \texttt{Branching}(S_1)\ \texttt{in}\ S_2$), then:

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

Remark. 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:

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

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

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

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

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

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

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

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

□

Lemma C.3.3. SSA form is preserved during the generation phase creating new skeletons. I.e.:

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

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

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

The third case is trivial, the other two are solved using Lemma C.3.1.

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

- If SSA($\texttt{let}\ \widetilde{w} = \texttt{Branching}\ (S_1, \ldots, S_n)\ \texttt{in}\ S_E$) then SSA($< S_i \mid \widetilde{w} \mid S_E >$).
- If SSA($\texttt{let}\ \widetilde{v} = K\ \texttt{in}\ < S \mid \widetilde{w} \mid S_E >$) then SSA($< S \mid \widetilde{w} \mid S_E >$).
- If SSA($\texttt{let}\ \widetilde{v} = K\ \texttt{in}\ < S \mid \widetilde{w} \mid S_E >$) then 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 C.3.1. The remaining (first) point can be cut in two halves, by proving the intermediate result SSA($\texttt{let}\ \widetilde{w} = \texttt{Branching}\ (S_i)\ \texttt{in}\ S_E$). The first half can be checked directly by following Definition 3. The second half is proved separately (Lemma C.3.2). □

Lemma C.3.4. The distribution of branchings preserves SSA form.

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

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

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

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

First, we check by following Definition 3 that we have SSA($\texttt{let}\ \widetilde{v} = \texttt{Branching}(S_i)\ \texttt{in}\ S'$) for each i. Then, using Lemma C.3.2, we get:

- 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:

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

□

Lemma C.3.5. 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 $(h(\widetilde{y}, c(\widetilde{z})) := S) \in \mathrm{R_{BS}}$ we initially have SSA(S), then for all rule $(h(\widetilde{y}, c(\widetilde{z})) := S) \in \mathrm{R_{gen}}$ we also have SSA(S).

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

For this, we check we have 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 SSA(S)) and an environment $E = [\cdot]$, so it holds. Then, the preservation of the property comes from Lemma C.3.3. □

C.4 Properties of the Transformation Phases

Lemma C.4.1. For all environment Σ, terms $\widetilde{a}$, and skeleton S such that 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 4.2.3 for the definition). The only interesting case is when S is of the form $\texttt{let}\ \widetilde{v} = \texttt{Branching}(S_1, \ldots, S_n)\ \texttt{in}\ S'$. Then, by definition, $\lceil S \rfloor \triangleq \texttt{Branching}(\lceil < S_1 \mid \widetilde{v} \mid S' > \rfloor, \ldots, \lceil < S_n \mid \widetilde{v} \mid S' > \rfloor)$.

$$
\begin{aligned}
&\Sigma \vdash S \Downarrow \widetilde{a} \\
\iff &\exists \widetilde{b}, S_i, \begin{cases} \Sigma \vdash S_i \Downarrow \widetilde{b} \\ \Sigma + \widetilde{\{v \mapsto b\}} \vdash S' \Downarrow \widetilde{a} \end{cases} \text{ by definition (using two rules)} \\
\iff &\exists S_i, \Sigma \vdash < S_i \mid \widetilde{v} \mid S' > \Downarrow \widetilde{a} \quad \text{using Lemma C.2.7} \\
\text{We have } &(\mathrm{bv}(S_i) \setminus \{\widetilde{v}\}) \cap \mathrm{fv}(S') = \emptyset \text{ from } \begin{cases} \mathrm{bv}(S) \cap \mathrm{fv}(S) = \emptyset \text{ from } \mathrm{SSA}(S) \\ \mathrm{fv}(S') \subset \mathrm{fv}(S) \cup \{\widetilde{v}\} \\ \mathrm{bv}(S_i) \subset \mathrm{bv}(S) \end{cases} \\
\iff &\exists S_i, \Sigma \vdash \lceil < S_i \mid \widetilde{v} \mid S' > \rfloor \Downarrow \widetilde{a} \quad \text{using IH, since } < S_i \mid \widetilde{v} \mid S' > \text{ is smaller than } S \\
\iff &\Sigma \vdash \lceil S \rfloor \Downarrow \widetilde{a}
\end{aligned}
$$

□

Lemma C.4.2. For all environment Σ and skeleton S such that $\mathrm{SSA}(S)$,

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

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

Proof. This lemma is similar to Lemma C.4.1, and so is the proof. This is done by a straightforward induction on S. Instead of using Lemma C.2.7, we use the similar Lemma C.2.6 for divergence. Also, we directly use Lemma C.4.1 instead of the induction hypothesis for finite subevaluations. □

Lemma C.4.3. Delaying returns does not impact the inductive interpretation. Thus, the rules common to $\mathrm{R}_{\mathrm{dist}}$ and $\mathrm{R}_{\mathrm{EBS}}$ 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 hooks, with the same reasoning). This comes from the two following derivation trees having the same leaf $\mathcal{R}_f(\widetilde{\Sigma(t)}) \Downarrow \widetilde{a}$.

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

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

□

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

Lemma C.4.4. For all hook h and terms $\widetilde{a}$, $\widetilde{b}$, and $\widetilde{d}$,

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

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 C.1.

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

□

Lemma C.4.5. For all hook h and terms $\widetilde{b}$ and $\widetilde{d}$,

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

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 C.1.

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

□

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

Lemma C.4.6. If $(h(\widetilde{y}, c(\widetilde{w}, \widetilde{z})) := \texttt{let}\ \widetilde{v} = \texttt{Hook Reuse}\ h_1\ \widetilde{w}\ \texttt{in}\ S) \in \mathrm{R_{EBS}}$, then

$$(\widetilde{e}, e_0) \Downarrow_{h_1}^{\mathrm{R_{SS}}} (\widetilde{e}, e_0') \implies (\widetilde{a}, c((\widetilde{e}, e_0), \widetilde{a'})) \Downarrow_h^{\mathrm{R_{SS}}} (\widetilde{a}, c((\widetilde{e}, e_0'), \widetilde{a'}))$$

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

$$\frac{(\ldots := \| \ldots \|^r) \in \mathrm{R_{SS}} \qquad \dfrac{(\widetilde{e}, e_0) \Downarrow_{h_1}^{\mathrm{R_{SS}}} (\widetilde{e}, e_0') \qquad \dfrac{\Sigma'(\widetilde{y}, c(\widetilde{u}, \widetilde{z})) = (\widetilde{a}, c((\widetilde{e}, e_0'), \widetilde{a'}))}{\Sigma' \vdash \texttt{Return}\ (\widetilde{y}, c(\widetilde{u}, \widetilde{z})) \Downarrow^{\mathrm{R_{SS}}} (\widetilde{a}, c((\widetilde{e}, e_0'), \widetilde{a'}))}}{\Sigma \vdash \texttt{let}\ \widetilde{u} = \texttt{Hook}\ h_1\ \widetilde{w}\ \texttt{in Return}\ (\widetilde{y}, c(\widetilde{u}, \widetilde{z})) \Downarrow^{\mathrm{R_{SS}}} (\widetilde{a}, c((\widetilde{e}, e_0'), \widetilde{a'}))}}{\Sigma \vdash \| \texttt{let}\ \widetilde{v} = \texttt{Hook Reuse}\ h_1\ \widetilde{w}\ \texttt{in}\ S \|^r \Downarrow^{\mathrm{R_{SS}}} (\widetilde{a}, c((\widetilde{e}, e_0'), \widetilde{a'}))}}{(\widetilde{a}, c((\widetilde{e}, e_0), \widetilde{a'})) \Downarrow_h^{\mathrm{R_{SS}}} (\widetilde{a}, c((\widetilde{e}, e_0'), \widetilde{a'}))}$$

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 C.4.7. If $(h(\widetilde{y}, c(\widetilde{w}, \widetilde{z})) := \texttt{let}\ \widetilde{v} = \texttt{Hook Reuse}\ h_1\ \widetilde{w}\ \texttt{in}\ S) \in \mathrm{R_{EBS}}$, and e_0 is not of the form `Ret_h1`(...), then

$$(\widetilde{a}, c((\widetilde{e}, e_0), \widetilde{a'}))\ \Downarrow_h^{\mathrm{R_{SS}}}\ \widetilde{d} \implies \exists e_0', \begin{cases} \widetilde{d} = (\widetilde{a}, c((\widetilde{e}, e_0'), \widetilde{a'})) \\ \widetilde{e}, e_0\ \Downarrow_{h_1}^{\mathrm{R_{SS}}}\ (\widetilde{e}, e_0') \end{cases}$$

This Lemma is the converse of Lemma C.4.6 above.

Proof. The property holds because the only possible evaluation tree is the one presented in the proof of Lemma C.4.6 above. Indeed $[\Sigma \vdash \texttt{let}\ \widetilde{v} = \texttt{Hook getRet\_h1}\ (w)\ \texttt{in}\ \|\ S\ \|^r\ \Downarrow^{\mathrm{R_{SS}}}\ \ldots]$, with $\Sigma(w) = e_0$, is not possible since `getRet_h1` only has a rule for the constructor `Ret_h1`. □

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

Lemma C.4.8. If $(h(\widetilde{y}, c((\widetilde{w}, w), \widetilde{z})) := \texttt{let}\ \widetilde{v} = \texttt{Hook Reuse}\ h_1\ (\widetilde{w}, w)\ \texttt{in}\ S) \in \mathrm{R_{EBS}}$, then

$$\{\widetilde{y \mapsto a}\} + \{\widetilde{z \mapsto a'}\} + \{\widetilde{v \mapsto b}\} \vdash \|\ S\ \|^r\ \Downarrow^{\mathrm{R_{SS}}}\ \widetilde{d} \implies (\widetilde{a}, c((\widetilde{e}, \texttt{Ret\_h1}(\widetilde{b})), \widetilde{a'}))\ \Downarrow_h^{\mathrm{R_{SS}}}\ \widetilde{d}$$

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

$$\dfrac{(\ldots := \|\ \ldots\ \|^r) \in \mathrm{R_{SS}} \qquad \dfrac{\dfrac{\texttt{Ret\_h1}(\widetilde{b})\ \Downarrow_{\texttt{getRet\_h1}}^{\mathrm{R_{SS}}}\ \widetilde{b} \qquad \overline{\Sigma' \vdash \|\ S\ \|^r\ \Downarrow^{\mathrm{R_{SS}}}\ \widetilde{d}}}{\Sigma \vdash \texttt{let}\ \widetilde{v} = \texttt{Hook getRet\_h1}\ (w)\ \texttt{in}\ \|\ S\ \|^r\ \Downarrow^{\mathrm{R_{SS}}}\ \widetilde{d}}}{\Sigma \vdash \|\ \texttt{let}\ \widetilde{v} = \texttt{Hook Reuse}\ h_1\ (\widetilde{w}, w)\ \texttt{in}\ S\ \|^r\ \Downarrow^{\mathrm{R_{SS}}}\ \widetilde{d}}}{(\widetilde{a}, c((\widetilde{e}, \texttt{Ret\_h1}(\widetilde{b})), \widetilde{a'}))\ \Downarrow_h^{\mathrm{R_{SS}}}\ \widetilde{d}}$$

With $\Sigma = \{\widetilde{y \mapsto a}\} + \{\widetilde{w \mapsto e}\} + \{w \mapsto \texttt{Ret\_h1}(\widetilde{b})\} + \{\widetilde{z \mapsto a'}\}$ and $\Sigma' = \Sigma + \{\widetilde{v \mapsto b}\}$. The first leaf uses Lemma C.4.5. The second leaf uses Lemma C.2.4 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 C.4.9. If $(h(\widetilde{y}, c((\widetilde{w}, w), \widetilde{z})) := \texttt{let}\ \widetilde{v} = \texttt{Hook Reuse}\ h_1\ (\widetilde{w}, w)\ \texttt{in}\ S) \in \mathrm{R_{EBS}}$, then

$$(\widetilde{a}, c((\widetilde{e}, \texttt{Ret\_h1}(\widetilde{b})), \widetilde{a'}))\ \Downarrow_h^{\mathrm{R_{SS}}}\ \widetilde{d} \implies \{\widetilde{y \mapsto a}\} + \{\widetilde{z \mapsto a'}\} + \{\widetilde{v \mapsto b}\} \vdash \|\ S\ \|^r\ \Downarrow^{\mathrm{R_{SS}}}\ \widetilde{d}$$

This Lemma is the converse of Lemma C.4.8 above.

Proof. The property holds because the only possible evaluation tree is the one presented in the proof of Lemma C.4.8 above. Indeed $[\Sigma \vdash \texttt{let}\ \widetilde{u} = \texttt{Hook}\ h_1\ (\widetilde{w}, w)\ \texttt{in Return}\ (\widetilde{y}, c(\widetilde{u}, \widetilde{z}))\ \Downarrow^{\mathrm{R_{SS}}}$

...], with $\Sigma(w) = \texttt{Ret\_h1}(\widetilde{b})$, is not possible since, in $\mathrm{R_{SS}}$, the hook h_1 does not have a rule for the constructor `Ret_h1`. We also use Lemma C.2.4 to simplify the environment. □

Lemma C.4.10. For every rule $r = (h(\widetilde{y}, c(\widetilde{x})) := S) \in \mathrm{R_{BS}}$, 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 $S_0 = \texttt{Branching}\ (S_1, \ldots, S_n)$, 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 $\lceil E[S_0] \rfloor = \texttt{Branching}\ (\lceil E[S_1] \rfloor, \ldots, \lceil E[S_n] \rfloor)$. If $E = [\cdot]$, this is trivial. If E is of the form $< [\cdot] \mid \widetilde{v} \mid S' >$, we have:

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

  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 $K = \texttt{Branching}\ (S'_1, \ldots, S'_n)$, then $\lceil E[S_0] \rfloor = \texttt{Branching}\ (\ldots, \lceil < S'_i \mid \widetilde{v} \mid E[S_1] > \rfloor, \ldots)$. Using Lemma C.2.8, 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 $E[K_H]$ (where $K_H = \texttt{Hook}\ (\texttt{New}\ c_0)\ h_1\ \widetilde{t}$) as its tail, then it comes from the distribution of a skeleton of the form $E'[K_H]$. The new rule for c_0 is the distribution of $E'[K_H]$, which results in $E[K_H]$ as well.

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

Lemma C.4.11. Forall S_0, E, E_0, r, c_0, h_1, $\widetilde{t}$, let $K_H \triangleq \texttt{Hook}\ (\texttt{New}\ c_0)\ h_1\ \widetilde{t}$.

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

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

- If $S_0 = \texttt{Branching}\ (S'_1, \ldots, S'_n)$, then there is exactly one S'_i that contains K_H and we generate $[\![S'_i]\!]^r_{E_0}$. Also $\lceil E_0[S_0] \rfloor = \texttt{Branching}\ (\lceil E_0[S'_1] \rfloor, \ldots, \lceil E_0[S'_n] \rfloor)$, so we can deduce that $E[K_H] \triangleleft \lceil E_0[S'_i] \rfloor$. By using our induction hypothesis with S'_i, we immediately get our desired result.
- If $S_0 = K$ is not a branching, then $K_H \in S_0$ implies that $S_0 = K_H$. We take $E' = E_0$ and need to check $\lceil E_0[K_H] \rfloor = E[K_H]$. We know that $E[K_H] \triangleleft \lceil E_0[K_H] \rfloor$. If $E_0 = [\cdot]$, then $E = [\cdot]$ and we have the equality. Else $\lceil E_0[K_H] \rfloor$ is of the form $\texttt{let}\ \ldots = K_H\ \texttt{in}\ \lceil \ldots \rfloor$, and since $K_H \notin E_0$, it does not appear in the "..." parts. Necessarily (from Definition 5), we have $E[K_H] = \lceil E_0[K_H] \rfloor$.
- If $S_0 = (\texttt{let}\ \widetilde{v} = \texttt{Branching}\ (S'_1, \ldots, S'_n)\ \texttt{in}\ S')$ and there is exactly one S'_i that contains K_H. Let $E_1 =< [\cdot] \mid \widetilde{v} \mid E_0[S'] >$, we generate $[\![S'_i]\!]^r_{E_1}$. Also $\lceil E_0[S_0] \rfloor = \texttt{Branching}\ (\lceil < S'_1 \mid \widetilde{v} \mid E_0[S'] > \rfloor, \ldots, \lceil < S'_n \mid \widetilde{v} \mid E_0[S'] > \rfloor)$, so we can deduce that $E[K_H] \triangleleft \lceil < S'_i \mid \widetilde{v} \mid E_0[S'] > \rfloor = \lceil E_1[S'_i] \rfloor$. We can use our induction hypothesis with E_1 and S'_i to immediately get our desired result.
- If $S_0 = (\texttt{let}\ \widetilde{v} = \texttt{Branching}\ (S'_1, \ldots, S'_n)\ \texttt{in}\ S')$ and $K_H \in S'$. Then K_H does not appear in any S'_i and we generate $[\![S']\!]^r_{E_0}$. Also $\lceil E_0[S_0] \rfloor = \texttt{Branching}\ (\lceil < S'_1 \mid \widetilde{v} \mid E_0[S'] > \rfloor, \ldots, \lceil < S'_n \mid \widetilde{v} \mid E_0[S'] > \rfloor)$. Since $E[K_H] \triangleleft \lceil E_0[S_0] \rfloor$, there is an S'_i such that $E[K_H] \triangleleft \lceil < S'_1 \mid \widetilde{v} \mid E_0[S'] > \rfloor$. We use Lemma C.2.9 to get more precisely that $E[K_H] \triangleleft \lceil E_0[S'] \rfloor$. Then we can use our induction hypothesis on E_0 and S' to get our desired result.
- If $S_0 = (\texttt{let}\ \widetilde{v} = K_H\ \texttt{in}\ S')$. let $E_1 =< [\cdot] \mid \widetilde{v} \mid E_0[S'] >$, we generate $[\![K_H]\!]^r_{E_1}$. We take $E' = E_1$, and need to show that $E[K_H]$ is equal to $\lceil E_1[K_H] \rfloor = (\texttt{let}\ \widetilde{v} = K_H\ \texttt{in}\ \lceil E_0[S'] \rfloor)$. Once again, $E[K_H] \triangleleft \lceil E_1[K_H] \rfloor$ and K_H does not appear in $\lceil E_0[S'] \rfloor$, so $E[K_H] \not\triangleleft \lceil E_0[S'] \rfloor$ 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 $K \neq K_H$ is not a branching, then $K_H \in S'$ 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 $E[K_H] \triangleleft \lceil E_0[S'] \rfloor$ and we have our desired result by using our induction hypothesis on E_0 and S'.

□

Lemma C.4.12. For all rule $(h(\widetilde{y}, c(\widetilde{x})) := \lceil S \rfloor) \in \mathrm{R}_{\mathrm{dist}}$, if $E[\texttt{Hook}\ (\texttt{New}\ c_0)\ h_1\ \widetilde{t}] \triangleleft \lceil S \rfloor$, then $\mathrm{R}_{\mathrm{dist}}$ contains a rule of the form:

$$h(\widetilde{y}, c_0(\widetilde{w}, \widetilde{z})) := E[\texttt{Hook Reuse}\ h_1\ \widetilde{w}]$$

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

$$h(\widetilde{y}, c(\widetilde{x_1}, \widetilde{x_2})) := E_S[\texttt{Hook Reuse}\ h_2\ \widetilde{x_1}]\ \in \mathrm{R}_{\mathrm{gen}}$$

with $\widetilde{x} = \widetilde{x_1}, \widetilde{x_2}$. So it has been created from a generation operation $[\![\texttt{Hook}\ (\texttt{New}\ c)\ h_2\ \widetilde{t'}]\!]^r_{E_S}$, with $r = (h(\widetilde{y}, c'(\widetilde{v})) := S_0) \in \mathrm{R}_{\mathrm{BS}}$ where c' is an initial constructor. Using Lemma C.4.10, we have $\lceil S \rfloor = \lceil E_S[\texttt{Hook Reuse}\ h_2\ \widetilde{x_1}] \rfloor \triangleleft \lceil S_0 \rfloor$. So by transitivity we recover the following hypothesis: $(h(\widetilde{y}, c'(\widetilde{v})) := \lceil S_0 \rfloor) \in \mathrm{R}_{\mathrm{dist}}$ where c' is an initial constructor and $E[\texttt{Hook}\ (\texttt{New}\ c_0)\ h_1\ \widetilde{t}] \triangleleft \lceil S_0 \rfloor$.

Now, we can assume without loss of generality that c is an initial constructor, and so that we have $r = (h(\widetilde{y}, c(\widetilde{x})) := S) \in \mathrm{R}_{\mathrm{BS}}$. We use Lemma C.4.11 with $[\![S]\!]^r_{[\cdot]}$ to get E' such that we generate $[\![\texttt{Hook}\ (\texttt{New}\ c_0)\ h_1\ \widetilde{t}]\!]^r_{E'}$ and have $\lceil E'[\texttt{Hook}\ (\texttt{New}\ c_0)\ h_1\ \widetilde{t}] \rfloor = E[\texttt{Hook}\ (\texttt{New}\ c_0)\ h_1\ \widetilde{t}]$. Thus the creation of new skeletons produces a rule $(h(\widetilde{y}, c_0(\widetilde{w}, \widetilde{z})) := E'[\texttt{Hook Reuse}\ h_1\ \widetilde{w}]) \in \mathrm{R}_{\mathrm{gen}}$ with fresh variables $\widetilde{w}$. And after distributing branchings we have

$$(h(\widetilde{y}, c_0(\widetilde{w}, \widetilde{z})) := E[\texttt{Hook Reuse}\ h_1\ \widetilde{w}]) \in \mathrm{R}_{\mathrm{dist}}$$

□

Lemma C.4.13. $\forall (h(\widetilde{y}, c(\widetilde{x})) := S) \in \mathrm{R}_{\mathrm{EBS}}$, if $(\texttt{let}\ \widetilde{v} = \texttt{Hook}\ (\texttt{New}\ c_0)\ h_1\ \widetilde{t}\ \texttt{in}\ S_0) \triangleleft S$, then $\mathrm{R}_{\mathrm{EBS}}$ contains a rule of the form:

$$h(\widetilde{y}, c_0(\widetilde{w}, \widetilde{z})) := \texttt{let}\ \widetilde{v} = \texttt{Hook Reuse}\ h_1\ \widetilde{w}\ \texttt{in}\ S_0$$

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

Proof. Directly from Lemma C.4.12, 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 Section 4.2), and not newly created constructors (e.g., `While1` or `Ret_h`).

Theorem C.5.1 (BS⇒EBS)**.** For all h and initial terms $\widetilde{a}$, and $\widetilde{b}$,

$$\widetilde{a} \Downarrow_h^{\mathrm{R_{BS}}} \widetilde{b} \implies \widetilde{|a|} \Downarrow_h^{\mathrm{R_{EBS}}} \widetilde{|b|}$$

Proof. Our hypothesis comes from a rule of the form $h(\widetilde{y}, c(\widetilde{x})) := S \in \mathrm{R_{BS}} \subseteq \mathrm{R_{gen}}$ 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, $h(\widetilde{y}, c(\widetilde{x})) := \lceil S \rfloor$ is a rule of $\mathrm{R_{dist}}$. And from Lemma C.4.1, we have $\widetilde{\{y \mapsto d\}} + \widetilde{\{x \mapsto d'\}} \vdash \lceil S \rfloor \Downarrow \widetilde{b}$ and $\widetilde{|a|} \Downarrow_h^{\mathrm{R_{dist}}} \widetilde{|b|}$. Since delaying returns does not change the behavior (Lemma C.4.3), we also have $\widetilde{|a|} \Downarrow_h^{\mathrm{R_{EBS}}} \widetilde{|b|}$. □

Theorem C.5.2 (EBS⇒BS)**.** For all h, initial terms $\widetilde{a}$ and extended terms $\widetilde{b'}$,

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

Proof. Since $\widetilde{a}$ are initial terms and do not use coercion constructors of the form `Ret_h`, we also have $\widetilde{|a|} \Downarrow_h^{\mathrm{R_{dist}}} \widetilde{b'}$ from Lemma C.4.3.

This comes from a rule of the form $h(\widetilde{y}, c(\widetilde{x})) := \lceil S \rfloor \in \mathrm{R_{dist}}$ 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, $h(\widetilde{y}, c(\widetilde{x})) := S$ is a rule of $\mathrm{R_{gen}}$, and from Lemma C.4.1, we have $\widetilde{|a|} \Downarrow_h^{\mathrm{R_{gen}}} \widetilde{b'}$.

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

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

□

Theorem C.5.3 (Div: BS⇔EBS)**.** For all h and initial terms $\widetilde{a}$,

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

Proof. Similarly to the two previous theorems. Since we only focus on initial terms (no new generated constructors), $\widetilde{a} \Uparrow_h^{\mathrm{R_{BS}}} \iff \widetilde{|a|} \Uparrow_h^{\mathrm{R_{gen}}}$. From the definition of $\mathrm{R_{dist}}$ and Lemma C.4.2,

$|\widetilde{a}| \Uparrow_h^{\mathrm{R_{gen}}} \iff |\widetilde{a}| \Uparrow_h^{\mathrm{R_{dist}}}$. Since we only focus on initial terms (no coerced returns) and delaying returns is also equivalent for coinduction, $|\widetilde{a}| \Uparrow_h^{\mathrm{R_{dist}}} \iff |\widetilde{a}| \Uparrow_h^{\mathrm{R_{EBS}}}$. □

C.6 Extended Big-Step Implies Small-Step

We first certify the finite case with the following theorem.

Theorem C.6.1 (EBS ⇒ SS)**.** For all $\widetilde{a}$, c, $\widetilde{a'}$, $\widetilde{b}$, and h,

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

Proof. Since $(\widetilde{a}, c(\widetilde{a'})) \Downarrow_h^{\mathrm{R_{EBS}}} \widetilde{b}$ 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 $\widetilde{a}, c, \widetilde{a'}, \widetilde{b}, h, \widetilde{y}, \widetilde{x}, S_0, \Sigma, S$, we have:

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

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 $(\widetilde{a}, \texttt{Ret\_h}(\widetilde{b}))$.

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

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

- If $c = \texttt{Ret\_h}$, using Lemma C.4.5 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

$$\dfrac{h(\widetilde{y}, c(\widetilde{w}, \widetilde{z})) := \ldots \qquad \dfrac{\overline{\Sigma_0(\widetilde{w}) \Downarrow_{h_1}^{\mathrm{R_{EBS}}} \widetilde{b'}} \qquad \overline{\Sigma_0 + \{\widetilde{v \mapsto b'}\} \vdash S \Downarrow^{\mathrm{R_{EBS}}} \widetilde{b}}}{\Sigma_0 \vdash \texttt{let } \widetilde{v} = \texttt{Hook Reuse } h_1 \ \widetilde{w} \ \texttt{ in } \ S \Downarrow^{\mathrm{R_{EBS}}} \widetilde{b}}}{(\widetilde{a}, c((\widetilde{a'_0}, a'_1), \widetilde{a'_2})) \Downarrow_h^{\mathrm{R_{EBS}}} \widetilde{b}}$$

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 $h(\widetilde{y}, c(\widetilde{w}, \widetilde{z})) := \texttt{let } \widetilde{v} = \texttt{Hook Reuse } h_1 \ \widetilde{w} \ \texttt{ in } \ S$.

Using our induction hypothesis on the first leaf we get $(\widetilde{a'_0}, a'_1)\ (\Downarrow_{h_1}^{\mathrm{R_{SS}}})^*\ (\widetilde{a'_0}, \texttt{Ret\_h1}(\widetilde{b'}))$. By Lemma C.4.6, we then have $(\widetilde{a}, c((\widetilde{a'_0}, a'_1), \widetilde{a'_2}))\ (\Downarrow_h^{\mathrm{R_{SS}}})^*\ (\widetilde{a}, c((\widetilde{a'_0}, \texttt{Ret\_h1}(\widetilde{b'})), \widetilde{a'_2}))$.

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

Finally, using Lemma C.4.8 (and Lemma C.2.4 to rewrite the environment), we have $(\widetilde{a}, c((\widetilde{a'_0}, \texttt{Ret\_h1}(\widetilde{b'})), \widetilde{a'_2}))\ \Downarrow_h^{\mathrm{R_{SS}}}\ (\widetilde{a}, d)$, 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:

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

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 $\Sigma_0 \vdash \parallel S \parallel^r \Downarrow^{\mathrm{R_{SS}}} (\widetilde{a}, d)$ and $(\widetilde{a}, d)\ (\Downarrow_h^{\mathrm{R_{SS}}})^*\ (\widetilde{a}, \texttt{Ret\_h}(\widetilde{b}))$. Now we can complete our goal as such:

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

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

Note that we have $\mathrm{SSA}(S_0)$, either from assumption for the initial skeletons, or by using Lemmas C.3.5 and C.3.4 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 $d = \texttt{Ret\_h}(\widetilde{b})$:

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

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

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

- 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:

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

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

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

- If S is of the form $\texttt{let}\ \widetilde{v} = \texttt{Hook}\ (\texttt{New}\ c_0)\ h_1\ \widetilde{t}\ \texttt{in}\ S_0$, we have:

$$\frac{\dfrac{\Sigma(\widetilde{t}) \Downarrow_{h_1}^{\mathrm{R_{EBS}}} \widetilde{b'}}{\Sigma \vdash \texttt{Hook}\ (\texttt{New}\ c_0)\ h_1\ \widetilde{t} \Downarrow^{\mathrm{R_{EBS}}} \widetilde{b'}} \qquad \overline{\Sigma + \{\widetilde{v \mapsto b'}\} \vdash S_0 \Downarrow^{\mathrm{R_{EBS}}} \widetilde{b}}}{\Sigma \vdash \texttt{let}\ \widetilde{v} = \texttt{Hook}\ (\texttt{New}\ c_0)\ h_1\ \widetilde{t}\ \texttt{in}\ S_0 \Downarrow^{\mathrm{R_{EBS}}} \widetilde{b}}$$

Let us name $\Sigma(\widetilde{t}) = (\widetilde{e}, e_t)$. We use our induction hypothesis on the first leaf and get $(\widetilde{e}, e_t)\ (\Downarrow_{h_1}^{\mathrm{R_{SS}}})^*\ (\widetilde{e}, \texttt{Ret\_h1}(\widetilde{b'}))$. 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 $(\widetilde{a}, d_0)\ (\Downarrow_h^{\mathrm{R_{SS}}})^*\ (\widetilde{a}, \texttt{Ret\_h}(\widetilde{b}))$.

From Lemma C.4.13, $\mathrm{R_{EBS}}$ contains a rule of the form

$h(\widetilde{y}, c_0(\widetilde{w}, \widetilde{z})) := \texttt{let}\ \widetilde{v} = \texttt{Hook Reuse}\ h_1\ \widetilde{w}\ \texttt{in}\ S_0$.

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 $(\widetilde{a}, d)\ (\Downarrow_h^{\mathrm{R_{SS}}})^*\ (\widetilde{a}, d_0)$ which can be cut into two pieces:

$$\frac{\dfrac{\cdots}{(\widetilde{a}, d)\ (\Downarrow_h^{\mathrm{R_{SS}}})^*\ (\widetilde{a}, c_0(\widetilde{e}, \texttt{Ret\_h1}(\widetilde{b'}), \Sigma(\widetilde{z})))} \qquad \dfrac{\cdots}{(\widetilde{a}, c_0(\widetilde{e}, \texttt{Ret\_h1}(\widetilde{b'}), \Sigma(\widetilde{z}))) \Downarrow_h^{\mathrm{R_{SS}}} (\widetilde{a}, d_0)}}{(\widetilde{a}, d)\ (\Downarrow_h^{\mathrm{R_{SS}}})^*\ (\widetilde{a}, d_0)}$$

The first part comes directly from iterating Lemma C.4.6. Also, Lemma C.4.8 gives us:

$$\frac{\{\widetilde{y \mapsto a}\} + \{z \mapsto \widetilde{\Sigma(z)}\} + \{\widetilde{v \mapsto b'}\} \vdash \parallel S_0 \parallel^r \Downarrow^{\mathrm{R_{SS}}} (\widetilde{a}, d_0)}{(\widetilde{a}, c_0(\widetilde{e}, \texttt{Ret\_h1}(\widetilde{b'}), \Sigma(\widetilde{z}))) \Downarrow_h^{\mathrm{R_{SS}}} (\widetilde{a}, d_0)}$$

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 C.2.4. □

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

Lemma C.6.2. For all h, $\widetilde{y}$, c, $\widetilde{x}$, S_0, Σ, S, and $\widetilde{a}$,

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

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 $\texttt{Branching}\ (S_1, \ldots, S_n)$, using the induction hypothesis with S_i:

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

▶ 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:

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

▶ If S is of the form $\texttt{let}\ \widetilde{v} = \texttt{Hook}\ (\texttt{New}\ c_0)\ h_1\ \widetilde{t}\ \texttt{in}\ S_0$, we have:
From Lemma C.4.13, $\mathrm{R_{EBS}}$ contains a rule of the form

$$h(\widetilde{y}, c_0(\widetilde{w}, \widetilde{z})) := \texttt{let}\ \widetilde{v} = \texttt{Hook Reuse}\ h_1\ \widetilde{w}\ \texttt{in}\ S_0$$

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

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

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

$$\frac{h(\widetilde{y}, c_0(\widetilde{w}, \widetilde{z})) := \ldots \in \mathrm{R_{EBS}} \qquad \dfrac{}{\Sigma' \vdash \texttt{let}\ \widetilde{v} = \texttt{Hook Reuse}\ h_1\ \widetilde{w}\ \texttt{in}\ S_0 \Uparrow^{\mathrm{R_{EBS}}}}}{(\widetilde{a}, d) \Uparrow_h^{\mathrm{R_{EBS}}}}\ \textsc{Div-tuple}$$

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 hook call $(\texttt{Hook}\ (\texttt{New}\ c_0)\ h_1\ \widetilde{t})$ diverges, this is straightforward as $\Sigma(\widetilde{t}) = \Sigma'(\widetilde{w})$. If evaluating S_0 diverges, we go from Σ to Σ' using Lemma C.2.5.

□

Lemma C.6.3. For all $\widetilde{a}$, c, $\widetilde{a'}$, and h,

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

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

- If it comes from a tree

$$\dfrac{h(\widetilde{y}, c(\widetilde{w}, \widetilde{z})) := \ldots \in \mathrm{R_{EBS}} \qquad \dfrac{\dfrac{\widetilde{a_1} \Uparrow_{h_1}^{\mathrm{R_{EBS}}}}{\Sigma \vdash \texttt{Hook}\ h_1\ \widetilde{w} \Uparrow^{\mathrm{R_{EBS}}}}\ \textsc{Div-Pc}}{\Sigma \vdash \texttt{let}\ \widetilde{v} = \texttt{Hook Reuse}\ h_1\ \widetilde{w}\ \texttt{in}\ S \Uparrow^{\mathrm{R_{EBS}}}}\ \textsc{Div-LetL}}{(\widetilde{a}, c(\widetilde{a_1}, \widetilde{a_3})) \Uparrow_h^{\mathrm{R_{EBS}}}}\ \textsc{Div-tuple}$$

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 $\widetilde{a_1} \Downarrow_{h_1}^{\mathrm{R_{SS}}} \widetilde{b'}$ and $\widetilde{b'} \Uparrow_{h_1}^{\mathrm{R_{EBS}}}$. We choose $\widetilde{b} = (\widetilde{a}, c(\widetilde{b'}, \widetilde{a_3}))$ and we have $(\widetilde{a}, c(\widetilde{a_1}, \widetilde{a_3}))) \Downarrow_h^{\mathrm{R_{SS}}} \widetilde{b}$ from Lemma C.4.6 and $\widetilde{b} \Uparrow_h^{\mathrm{R_{EBS}}}$ from a tree similar to the one above (replacing $\widetilde{a_1}$ by $\widetilde{b'}$).

- If it comes from a tree

$$\dfrac{h(\widetilde{y}, c(\widetilde{w}, \widetilde{z})) := \ldots \in \mathrm{R_{EBS}} \qquad \dfrac{\dfrac{\widetilde{a_1}, a_2 \Downarrow_{h_1}^{\mathrm{R_{EBS}}} \widetilde{d}}{\Sigma \vdash \texttt{Hook}\ h_1\ \widetilde{w} \Downarrow^{\mathrm{R_{EBS}}} \widetilde{d}} \qquad \dfrac{}{\Sigma' \vdash S \Uparrow^{\mathrm{R_{EBS}}}}}{\Sigma \vdash \texttt{let}\ \widetilde{v} = \texttt{Hook Reuse}\ h_1\ \widetilde{w}\ \texttt{in}\ S \Uparrow^{\mathrm{R_{EBS}}}}\ \textsc{Div-LetR}}{(\widetilde{a}, c(\widetilde{a_1}, a_2, \widetilde{a_3})) \Uparrow_h^{\mathrm{R_{EBS}}}}\ \textsc{Div-tuple}$$

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 `Ret_h1`(..), then Theorem C.6.1 gives us $\widetilde{a_1}, a_2\ (\Downarrow_{h_1}^{\mathrm{R_{SS}}})^*\ \widetilde{a_1}, \texttt{Ret\_h1}(\widetilde{d})$ which implies there is a_2' such that $\widetilde{a_1}, a_2 \Downarrow_{h_1}^{\mathrm{R_{SS}}} \widetilde{a_1}, a_2'\ \ (\Downarrow_{h_1}^{\mathrm{R_{SS}}})^*\ \widetilde{a_1}, \texttt{Ret\_h1}(\widetilde{d})$. We take $\widetilde{b} = (\widetilde{a}, c(\widetilde{a_1}, a_2', \widetilde{a_3}))$, we have $(\widetilde{a}, c(\widetilde{a_1}, a_2, \widetilde{a_3}))) \Downarrow_h^{\mathrm{R_{SS}}} \widetilde{b}$ from Lemma C.4.6 and $\widetilde{b} \Uparrow_h^{\mathrm{R_{EBS}}}$ from a tree similar to the one above (replacing a_2 by a_2').

Otherwise we have $a_2 = \texttt{Ret\_h1}(\widetilde{d})$ from Lemma C.4.4. Using Lemma C.6.2 above, there is d_0 such that $\Sigma' \vdash \| S \|^r \Downarrow^{\mathrm{R_{SS}}} (\widetilde{a}, d_0)$ and $(\widetilde{a}, d_0) \Uparrow_h^{\mathrm{R_{EBS}}}$, where r is the rule on the tree above. We choose $\widetilde{b} = (\widetilde{a}, d_0)$ and can conclude $(\widetilde{a}, c(\widetilde{a'})) \Downarrow_h^{\mathrm{R_{SS}}} \widetilde{b}$ using Lemma C.4.8.

- Lastly, if it comes from a tree

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

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 C.6.2 and there is d such that $\Sigma_0 \vdash \parallel S \parallel^r \Downarrow^{\mathrm{R_{SS}}} (\widetilde{a}, d)$ and $(\widetilde{a}, d) \Uparrow_h^{\mathrm{R_{EBS}}}$, where $r = (h(\widetilde{y}, c(\widetilde{x})) := S)$. Then we complete our goal as such:

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

□

Theorem C.6.4 (Div: EBS ⇒ SS)**.** For all hook h and terms $\widetilde{a}$,

$$\widetilde{a} \Uparrow_h^{\mathrm{R_{EBS}}} \implies \widetilde{a} \overset{\infty}{\to}_h$$

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

$$\forall \widetilde{a}, h, \quad \widetilde{a} \Uparrow_h^{\mathrm{R_{EBS}}} \quad \implies \quad \exists \widetilde{b},\ \widetilde{a} \Downarrow_h^{\mathrm{R_{SS}}} \widetilde{b} \ \wedge\ \widetilde{b} \Uparrow_h^{\mathrm{R_{EBS}}}$$

We prove this separately in Lemma C.6.3 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 C.7.1 (Concatenation)**.** For all $\widetilde{a}$, c, $\widetilde{a'}$, d, $\widetilde{b}$, and h,

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

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

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

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

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 $(\widetilde{a}, c(\widetilde{a'})) \Downarrow_h^{\mathrm{R_{SS}}} \widetilde{b}$ and on S.

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

- If c is a new constructor, the main rule $r \in \mathrm{R_{EBS}}$ is of the form $h(\widetilde{y}, c(\widetilde{w}, \widetilde{z})) := \texttt{let}\ \widetilde{v} = \texttt{Hook Reuse}\ h_1\ \widetilde{w}\ \texttt{in}\ S_0.$ and we have $\widetilde{a'} = ((\widetilde{a'_0}, a'_1), \widetilde{a'_2})$.

  ⊛ If a'_1 is not of the form $\texttt{Ret\_h1}(\ldots)$, then the small-step reduction uses the first branch and our small-step hypothesis corresponds to the following tree.

$$\dfrac{(\ldots := \| \ldots \|^r) \in \mathrm{R_{SS}} \qquad \dfrac{\dfrac{(\widetilde{a'_0}, a'_1) \Downarrow_{h_1}^{\mathrm{R_{SS}}} (\widetilde{a'_0}, a''_1) \qquad \dfrac{\Sigma'(\widetilde{y}, c(\widetilde{u}, \widetilde{z})) = (\widetilde{a}, d)}{\Sigma' \vdash \texttt{Return}\ (\widetilde{y}, c(\widetilde{u}, \widetilde{z})) \Downarrow^{\mathrm{R_{SS}}} (\widetilde{a}, d)}}{\Sigma_0 \vdash \texttt{let}\ \widetilde{u} = \texttt{Hook}\ h_1\ \widetilde{w}\ \texttt{in Return}\ (\widetilde{y}, c(\widetilde{u}, \widetilde{z})) \Downarrow^{\mathrm{R_{SS}}} (\widetilde{a}, d)}}{\Sigma_0 \vdash \| \texttt{let}\ \widetilde{v} = \texttt{Hook Reuse}\ h_1\ \widetilde{w}\ \texttt{in}\ S_0 \|^r \Downarrow^{\mathrm{R_{SS}}} (\widetilde{a}, d)}}{(\widetilde{a}, c((\widetilde{a'_0}, a'_1), \widetilde{a'_2})) \Downarrow_h^{\mathrm{R_{SS}}} (\widetilde{a}, d)}$$

  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 $(\widetilde{a}, c((\widetilde{a'_0}, a''_1), \widetilde{a'_2}))) \Downarrow_h^{\mathrm{R_{EBS}}} \widetilde{b}$, corresponding to the tree:

$$\dfrac{(\ldots := \ldots) \in \mathrm{R_{EBS}} \qquad \dfrac{(\widetilde{a'_0}, a''_1) \Downarrow_{h_1}^{\mathrm{R_{EBS}}} \widetilde{b'} \qquad \Sigma_1 + \{\widetilde{u \mapsto b'}\} \vdash S_0 \Downarrow^{\mathrm{R_{EBS}}} \widetilde{b}}{\Sigma_1 \vdash \texttt{let}\ \widetilde{v} = \texttt{Hook Reuse}\ h_1\ \widetilde{w}\ \texttt{in}\ S_0 \Downarrow^{\mathrm{R_{EBS}}} \widetilde{b}}}{(\widetilde{a}, c((\widetilde{a'_0}, a''_1), \widetilde{a'_2})) \Downarrow_h^{\mathrm{R_{EBS}}} \widetilde{b}}$$

  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 $(\widetilde{a'_0}, a'_1) \Downarrow_{h_1}^{\mathrm{R_{SS}}} (\widetilde{a'_0}, a''_1)$ and

$(\widetilde{a'_0}, a''_1) \Downarrow_{h_1}^{\mathrm{R_{EBS}}} \widetilde{b'}$ and produce our goal, i.e. $(\widetilde{a}, c(\widetilde{a'})) \Downarrow_h^{\mathrm{R_{EBS}}} \widetilde{b}$:

$$\dfrac{(\ldots := \ldots) \in \mathrm{R_{EBS}} \qquad \dfrac{(\widetilde{a'_0}, a'_1) \Downarrow_{h_1}^{\mathrm{R_{EBS}}} \widetilde{b'} \qquad \Sigma_0 + \widetilde{\{u \mapsto b'\}} \vdash S_0 \Downarrow^{\mathrm{R_{EBS}}} \widetilde{b}}{\Sigma_0 \vdash \texttt{let}\ \widetilde{v} = \texttt{Hook Reuse}\ h_1\ \widetilde{w}\ \texttt{in}\ S_0 \Downarrow^{\mathrm{R_{EBS}}} \widetilde{b}}}{(\widetilde{a}, c((\widetilde{a'_0}, a'_1), \widetilde{a'_2})) \Downarrow_h^{\mathrm{R_{EBS}}} \widetilde{b}}$$

where we go from Σ_1 to Σ_0 with Lemma C.2.4, since $\widetilde{w}$ does not appear in S_0.

⊛ If $a'_1 = \texttt{Ret\_h1}(\widetilde{b'})$, then the small-step reduction uses the second branch. We can single out the last variable of the hook call to simplify notations:
$h(\widetilde{y}, c((\widetilde{w}, w), \widetilde{z})) := \texttt{let}\ \widetilde{v} = \texttt{Hook Reuse}\ h_1\ (\widetilde{w}, w)\ \texttt{in}\ S_0$, with a state
$\Sigma_0 = \widetilde{\{y \mapsto a\}} + \widetilde{\{w \mapsto a'_0\}} + \{w \mapsto \texttt{Ret\_h1}(\widetilde{b'})\} + \widetilde{\{z \mapsto a'_2\}}$. Our first hypothesis then corresponds to the following tree.

$$\dfrac{(\ldots := \| \ldots \|^r) \in \mathrm{R_{SS}} \qquad \dfrac{\dfrac{\texttt{Ret\_h1}(\widetilde{b'}) \Downarrow_{\texttt{getRet\_h1}}^{\mathrm{R_{SS}}} \widetilde{b'} \qquad \Sigma_0 + \widetilde{\{v \mapsto b'\}} \vdash \| S_0 \|^r \Downarrow^{\mathrm{R_{SS}}} (\widetilde{a}, d)}{\Sigma_0 \vdash \texttt{let}\ \widetilde{v} = \texttt{Hook getRet\_h1}\ (w)\ \texttt{in}\ \| S_0 \|^r \Downarrow^{\mathrm{R_{SS}}} (\widetilde{a}, d)}}{\Sigma_0 \vdash \| \texttt{let}\ \widetilde{v} = \texttt{Hook Reuse}\ h_1\ (\widetilde{w}, w)\ \texttt{in}\ S_0 \|^r \Downarrow^{\mathrm{R_{SS}}} (\widetilde{a}, d)}}{(\widetilde{a}, c((\widetilde{a'_0}, \texttt{Ret\_h1}(\widetilde{b'})), \widetilde{a'_2})) \Downarrow_h^{\mathrm{R_{SS}}} (\widetilde{a}, d)}$$

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

$$\dfrac{h(\widetilde{y}, c((\widetilde{w}, w), \widetilde{z})) := \ldots \qquad \dfrac{\overline{(\widetilde{a'_0}, \texttt{Ret\_h1}(\widetilde{b'})) \Downarrow_{h_1}^{\mathrm{R_{EBS}}} \widetilde{b'}} \qquad \overline{\Sigma_0 + \widetilde{\{v \mapsto b'\}} \vdash S_0 \Downarrow^{\mathrm{R_{EBS}}} \widetilde{b}}}{\Sigma_0 \vdash \texttt{let}\ \widetilde{v} = \texttt{Hook Reuse}\ h_1\ (\widetilde{w}, w)\ \texttt{in}\ S_0 \Downarrow^{\mathrm{R_{EBS}}} \widetilde{b}}}{(\widetilde{a}, c((\widetilde{a'_0}, \texttt{Ret\_h1}(\widetilde{b'})), \widetilde{a'_2})) \Downarrow_h^{\mathrm{R_{EBS}}} \widetilde{b}}$$

where the other leaf comes from Lemma C.4.4.

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

$$\dfrac{h(\widetilde{y}, c(\widetilde{x})) := \| S \|^r \in \mathrm{R_{SS}} \qquad \dfrac{\ldots}{\Sigma_0 \vdash \| S \|^r \Downarrow^{\mathrm{R_{SS}}} (\widetilde{a}, d)}}{(\widetilde{a}, c(\widetilde{a'})) \Downarrow_h^{\mathrm{R_{SS}}} (\widetilde{a}, d)}$$

Where $r = (\widetilde{h(\widetilde{y}, c(\widetilde{x}))} := S) \in \mathrm{R_{EBS}}$, S does not start with a `Reuse` label, and $\Sigma_0 = \widetilde{\{y \mapsto a\}} + \{x \mapsto a'\}$.

We use our induction hypothesis on the leaf, and we complete our goal as such:

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

▶ Second result, with hypotheses $\begin{cases} r = (h(\widetilde{y}, c(\widetilde{x})) := S_0) \in \mathrm{R_{EBS}} \\ S \triangleleft S_0 \\ \Sigma(\widetilde{y}) = \widetilde{a} \\ \Sigma \vdash \parallel S \parallel^r \Downarrow^{\mathrm{R_{SS}}} (\widetilde{a}, d) \\ (\widetilde{a}, d) \Downarrow_h^{\mathrm{R_{EBS}}} \widetilde{b} \\ S \text{ has no } \texttt{Reuse} \text{ label} \end{cases}$

We perform an induction on S.

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

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

implies, from Lemma C.4.4, that we have $d = \texttt{Ret\_h}(\widetilde{b})$ 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 $\texttt{Branching}\ (S_1, \ldots, S_n)$, using the induction hypothesis with S_i:

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

• 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:

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

• If S is of the form $\texttt{let}\ \widetilde{v} = \texttt{Hook}\ (\texttt{New}\ c_0)\ h_1\ \widetilde{t}\ \texttt{in}\ S_0$, then from Lemma C.4.13, $\mathrm{R_{EBS}}$ contains a rule of the form $h(\widetilde{y}, c_0(\widetilde{w}, \widetilde{z})) := \texttt{let}\ \widetilde{v} = \texttt{Hook Reuse}\ h_1\ \widetilde{w}\ \texttt{in}\ S_0$. We have:

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

From which we can deduce $d = \widetilde{c_0(\Sigma(\widetilde{t}), \Sigma(\widetilde{z}))}$, 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:

$$\frac{h(\widetilde{y}, c_0(\widetilde{w}, \widetilde{z})) := \ldots \in \mathrm{R_{EBS}} \qquad \dfrac{\dfrac{\Sigma'(\widetilde{w}) \Downarrow_{h_1}^{\mathrm{R_{EBS}}} \widetilde{b'}}{\Sigma' \vdash \texttt{Hook Reuse}\ h_1\ \widetilde{w} \Downarrow \widetilde{b'}} \qquad \Sigma' + \{\widetilde{v \mapsto b'}\} \vdash S_0 \Downarrow^{\mathrm{R_{EBS}}} \widetilde{b}}{\Sigma' \vdash \texttt{let}\ \widetilde{v} = \texttt{Hook Reuse}\ h_1\ \widetilde{w}\ \texttt{in}\ S_0 \Downarrow^{\mathrm{R_{EBS}}} \widetilde{b}}}{(\widetilde{a}, d) \Downarrow_h^{\mathrm{R_{EBS}}} \widetilde{b}}$$

Note that $\Sigma'(\widetilde{w}) = \Sigma(\widetilde{t})$. Also, using Lemma C.2.4, we can deduce $\Sigma + \{\widetilde{v \mapsto b'}\} \vdash S_0 \Downarrow^{\mathrm{R_{EBS}}} \widetilde{b}$ and create our goal derivation:

$$\frac{\dfrac{\Sigma(\widetilde{t}) \Downarrow_{h_1}^{\mathrm{R_{EBS}}} \widetilde{b'}}{\Sigma \vdash \texttt{Hook}\ (\texttt{New}\ c_0)\ h_1\ \widetilde{t} \Downarrow \widetilde{b'}} \qquad \Sigma + \{\widetilde{v \mapsto b'}\} \vdash S_0 \Downarrow^{\mathrm{R_{EBS}}} \widetilde{b}}{\Sigma \vdash \texttt{let}\ \widetilde{v} = \texttt{Hook}\ (\texttt{New}\ c_0)\ h_1\ \widetilde{t}\ \texttt{in}\ S_0 \Downarrow^{\mathrm{R_{EBS}}} \widetilde{b}}$$

□

Theorem C.7.2 (SS ⇒ EBS)**.**

$$\forall \widetilde{a}, c, \widetilde{a'}, \widetilde{b}, h, \quad (\widetilde{a}, c(\widetilde{a'}))\ (\Downarrow_h^{\mathrm{R_{SS}}})^*\ (\widetilde{a}, \texttt{Ret\_h}(\widetilde{b})) \implies (\widetilde{a}, c(\widetilde{a'})) \Downarrow_h^{\mathrm{R_{EBS}}} \widetilde{b}$$

Proof. We prove it by induction on the number of small-step reductions. If $c(\widetilde{a'}) = \texttt{Ret\_h}(\widetilde{b})$, i.e. there is zero small steps, the result holds from Lemma C.4.4. Otherwise, the induction case

comes directly from Lemma C.7.1 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 C.7.3. Assuming $(h(\widetilde{y}, c((\widetilde{w}, w), \widetilde{z})) := \texttt{let}\ \widetilde{v} = \texttt{Hook Reuse}\ h_1\ (\widetilde{w}, w)\ \texttt{in}\ S) \in \mathrm{R_{EBS}}$, if we have a small-step divergence

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

then either the first hook call diverges

$$(\widetilde{a}, a_0) \overset{\infty}{\to}_{h_1}$$

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

$$\exists \widetilde{a'}, \begin{cases} (\widetilde{a}, a_0)\ (\Downarrow_{h_1}^{\mathrm{R_{SS}}})^*\ (\widetilde{a}, \texttt{Ret\_h1}(\widetilde{a'})) \\ (\widetilde{b}, c((\widetilde{a}, \texttt{Ret\_h1}(\widetilde{a'})), \widetilde{b'})) \overset{\infty}{\to}_h \end{cases}$$

Proof. We use the excluded middle axiom, and perform a case disjunction on whether there is a tuple of the form $(\widetilde{b}, c((\widetilde{a}, \texttt{Ret\_h1}(\widetilde{a'})), \widetilde{b'}))$ in the infinite sequence of reduction.

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

$$(\widetilde{b}, c((\widetilde{a}, a_0), \widetilde{b'}))\ (\Downarrow_h^{\mathrm{R_{SS}}})^*\ (\widetilde{b}, c((\widetilde{a}, \texttt{Ret\_h1}(\widetilde{a'})), \widetilde{b'})) \overset{\infty}{\to}_h$$

Now we show by induction on $(\widetilde{b}, c((\widetilde{a}, a_0), \widetilde{b'}))\ (\Downarrow_h^{\mathrm{R_{SS}}})^*\ (\widetilde{b}, c((\widetilde{a}, \texttt{Ret\_h1}(\widetilde{a'})), \widetilde{b'}))$ that we have $(\widetilde{a}, a_0)\ (\Downarrow_{h_1}^{\mathrm{R_{SS}}})^*\ (\widetilde{a}, \texttt{Ret\_h1}(\widetilde{a'}))$. The base case (zero steps) is trivial. For the induction case, note that a_0 cannot be of the form $\texttt{Ret\_h1}(\ldots)$ as we chose $(\widetilde{b}, c((\widetilde{a}, \texttt{Ret\_h1}(\widetilde{a'})), \widetilde{b'}))$ to be the first of this form. We then use Lemma C.4.7 to transform the first step, and the induction hypothesis to transform the rest.

• If there is no such tuple, we show $(\widetilde{a}, a_0) \overset{\infty}{\to}_{h_1}$ by coinduction. We cut our hypothesis and there is $\widetilde{t}$ such that $(\widetilde{b}, c((\widetilde{a}, a_0), \widetilde{b'})) \Downarrow_h^{\mathrm{R_{SS}}} \widetilde{t} \overset{\infty}{\to}_h$. From Lemma C.4.7, we have $\widetilde{t} = (\widetilde{b}, c((\widetilde{a}, a_0'), \widetilde{b'}))$ with $(\widetilde{a}, a_0) \Downarrow_{h_1}^{\mathrm{R_{SS}}} (\widetilde{a}, a_0')$. Also, $(\widetilde{b}, c((\widetilde{a}, a_0'), \widetilde{b'})) \overset{\infty}{\to}_h$ still has the property of not having any intermediate tuple of the form $(\widetilde{b}, c((\widetilde{a}, \texttt{Ret\_h1}(\widetilde{a'})), \widetilde{b'}))$, so we can use our induction hypothesis with it to conclude.

□

Theorem C.7.4 (Div: SS ⇒ EBS)**.** For all hook h and terms $\widetilde{a}$,

$$\widetilde{a} \overset{\infty}{\to}_h \quad \Longrightarrow \quad \widetilde{a} \Uparrow_h^{\mathrm{R_{EBS}}}$$

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

- $\forall \widetilde{a}, h, \quad \widetilde{a} \overset{\infty}{\to}_h \quad \Longrightarrow \quad \widetilde{a} \Uparrow_h^{\mathrm{R_{EBS}}}$
- $\forall h, \widetilde{y}, c, \widetilde{x}, S_0, \Sigma, S, \widetilde{b}, \quad \begin{cases} r = (h(\widetilde{y}, c(\widetilde{x})) := S_0) \in \mathrm{R_{EBS}} \\ S \triangleleft S_0 \\ \Sigma \vdash \parallel S \parallel^r \Downarrow^{\mathrm{R_{SS}}} \widetilde{b} \\ \widetilde{b} \overset{\infty}{\to}_h \\ S \text{ has no } \texttt{Reuse} \text{ label} \end{cases} \Longrightarrow \quad \Sigma \vdash S \Uparrow^{\mathrm{R_{EBS}}}$

We note the predicates

$$Q = \{(\widetilde{a}, h) \mid \widetilde{a} \overset{\infty}{\to}_h\}$$

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

To prove both these result by coinduction, we show that the two predicates satisfy the properties corresponding to the rules of Figure C.2. 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 C.2 and using elements of Q and Q' as leaves.

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

• If c is an initial constructor, then $\mathrm{R_{EBS}}$ contains a rule r of the form $h(\widetilde{y}, c(\widetilde{x})) := S_0$ 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 $\Sigma \vdash \parallel S_0 \parallel^r \Downarrow^{\mathrm{R_{SS}}} \widetilde{b}$ and $\widetilde{b} \overset{\infty}{\to}_h$. 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.

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

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

$$h(\widetilde{y}, c(\widetilde{w}, \widetilde{z})) := \texttt{let } \widetilde{v} = \texttt{Hook Reuse } h_1 \ \widetilde{w} \ \texttt{ in } S_0$$

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 C.7.3 to perform a case disjunction on the behavior of $\widetilde{a}, c(\widetilde{a_1}, \widetilde{a_2}) \overset{\infty}{\to}_h$.

⊛ If $\widetilde{a_1} \overset{\infty}{\to}_{h_1}$, then we can start constructing our goal as such:

$$\dfrac{h(\widetilde{y}, c(\widetilde{w}, \widetilde{z})) := \ldots \in \mathrm{R_{EBS}} \qquad \dfrac{\dfrac{(\widetilde{a_1}, h_1) \in Q}{\Sigma \vdash \texttt{Hook}\ h_1\ \widetilde{w} \Uparrow^{\mathrm{R_{EBS}}}}\ \textsc{Div-Pc} \qquad \Sigma \vdash \texttt{let}\ \widetilde{v} = \texttt{Hook Reuse}\ h_1\ \widetilde{w}\ \texttt{in}\ S_0 \Uparrow^{\mathrm{R_{EBS}}}}{\widetilde{a}, c(\widetilde{a_1}, \widetilde{a_2}) \Uparrow_h^{\mathrm{R_{EBS}}}}\ \textsc{Div-tuple}}{}\ \textsc{Div-LetL}$$

⊛ Else there is $\widetilde{a'_1}$ such that $\widetilde{a_1}\ (\Downarrow_{h_1}^{\mathrm{R_{SS}}})^*\ (\ldots, \texttt{Ret\_h1}(\widetilde{a'_1}))$ and $(\widetilde{a}, c(\ldots, \texttt{Ret\_h1}(\widetilde{a'_1}), \widetilde{a_2})) \overset{\infty}{\to}_h$. Using Theorem C.7.2 on the first point, we get $\widetilde{a_1} \Downarrow_{h_1}^{\mathrm{R_{EBS}}} \widetilde{a'_1}$. Using Lemma C.4.9 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 $\widetilde{b} \overset{\infty}{\to}_h$, 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 C.2.4, and so $(\Sigma + \widetilde{\{v \mapsto a'_1\}}, S_0) \in Q'$ using rule r, and we can construct our goal as such:

$$\dfrac{h(\widetilde{y}, c(\widetilde{w}, \widetilde{z})) := \ldots \in \mathrm{R_{EBS}} \qquad \dfrac{\dfrac{\widetilde{a_1} \Downarrow_{h_1}^{\mathrm{R_{EBS}}} \widetilde{a'_1}}{\Sigma \vdash \texttt{Hook}\ h_1\ \widetilde{w} \Downarrow^{\mathrm{R_{EBS}}} \widetilde{a'_1}} \qquad (\Sigma + \widetilde{\{v \mapsto a'_1\}}, S_0) \in Q'}{\Sigma \vdash \texttt{let}\ \widetilde{v} = \texttt{Hook Reuse}\ h_1\ \widetilde{w}\ \texttt{in}\ S_0 \Uparrow^{\mathrm{R_{EBS}}}}\ \textsc{Div-LetR}}{\widetilde{a}, c(\widetilde{a_1}, \widetilde{a_2}) \Uparrow_h^{\mathrm{R_{EBS}}}}\ \textsc{Div-tuple}$$

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

First, we can see certain cases for S are not possible. S cannot be a hook 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 $(\ldots, \texttt{Ret\_h}(\ldots))$ and $\widetilde{b} \overset{\infty}{\to}_h$ contradicts this.

• If $S = \texttt{Branching}\ (S_1, \ldots, S_n)$, then $\Sigma \vdash \texttt{Branching}\ (\parallel S_1 \parallel^r, \ldots, \parallel S_n \parallel^r) \Downarrow^{\mathrm{R_{SS}}} \widetilde{b}$ 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.

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

• If $S = (\texttt{let}\ \widetilde{v} = K\ \texttt{in}\ S')$ 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.

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

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 $S = (\texttt{let}\ \widetilde{v} = \texttt{Hook}\ (\texttt{New}\ c_0)\ h_1\ \widetilde{t}\ \texttt{in}\ S')$ then, from Lemma C.4.13, $\mathrm{R_{EBS}}$ contains a rule r' of the form

$$h(\widetilde{y}, c_0(\widetilde{w}, \widetilde{z})) := \texttt{let}\ \widetilde{v} = \texttt{Hook Reuse}\ h_1\ \widetilde{w}\ \texttt{in}\ S'$$

Let $\widetilde{a_0} = \Sigma(\widetilde{y})$, $\widetilde{a_1} = \Sigma(\widetilde{t})$, and $\widetilde{a_2} = \Sigma(\widetilde{z})$. We have $\| S \|^r = \texttt{Return}\ (\widetilde{y}, c_0(\widetilde{t}, \widetilde{z}))$ and so $\widetilde{b} = (\widetilde{a_0}, c_0(\widetilde{a_1}, \widetilde{a_2}))$. We use Lemma C.7.3 to perform a case disjunction on the behavior of $\widetilde{a_0}, c_0(\widetilde{a_1}, \widetilde{a_2}) \overset{\infty}{\to}_h$.

⊛ If $\widetilde{a_1} \overset{\infty}{\to}_{h_1}$, then we can start constructing our goal as such:

$$\frac{\dfrac{(\widetilde{a_1}, h_1) \in Q}{\Sigma \vdash \texttt{Hook}\ h_1\ \widetilde{t} \Uparrow^{\mathrm{R_{EBS}}}} \textsc{Div-Pc}}{\Sigma \vdash \texttt{let}\ \widetilde{v} = \texttt{Hook}\ (\texttt{New}\ c_0)\ h_1\ \widetilde{t}\ \texttt{in}\ S' \Uparrow^{\mathrm{R_{EBS}}}} \textsc{Div-LetL}$$

⊛ Else there is $\widetilde{a_1'}$ such that $\widetilde{a_1}\ (\Downarrow_{h_1}^{\mathrm{R_{SS}}})^*\ (\ldots, \texttt{Ret\_h1}(\widetilde{a_1'}))$ and $(\widetilde{a_0}, c_0(\ldots, \texttt{Ret\_h1}(\widetilde{a_1'}), \widetilde{a_2})) \overset{\infty}{\to}_h$. Using Theorem C.7.2 on the first point, we get $\widetilde{a_1} \Downarrow_{h_1}^{\mathrm{R_{EBS}}} \widetilde{a_1'}$. Using Lemma C.4.9 on the second point (after splitting a step), we get there is $\widetilde{b'}$ such that $\Sigma' \vdash \| S' \|^{r'} \Downarrow^{\mathrm{R_{SS}}} \widetilde{b'}$ and $\widetilde{b'} \overset{\infty}{\to}_h$, 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 C.2.4, and so $(\Sigma + \{\widetilde{v \mapsto a_1'}\}, S') \in Q'$ using rule r', and we can construct our goal as such:

$$\frac{\dfrac{\widetilde{a_1} \Downarrow_{h_1}^{\mathrm{R_{EBS}}} \widetilde{a_1'}}{\Sigma \vdash \texttt{Hook}\ h_1\ \widetilde{t} \Downarrow^{\mathrm{R_{EBS}}} \widetilde{a_1'}} \qquad (\Sigma + \{\widetilde{v \mapsto a_1'}\}, S') \in Q'}{\Sigma \vdash \texttt{let}\ \widetilde{v} = \texttt{Hook}\ (\texttt{New}\ c_0)\ h_1\ \widetilde{t}\ \texttt{in}\ S' \Uparrow^{\mathrm{R_{EBS}}}} \textsc{Div-LetR}$$

□

Appendix D

Complete Derivation of the NDAM and AM

D.1 Successive Phases of the NDAM Pseudo-code

D.1.1 Initial Pseudo-Code

```
def eval_trm t e : cvalue = match t with
| TVar (v) -> match v with
  | VLet (x) -> lookup e x
  | VSpec (h) -> eval_trm (specdecl h) []
  | VUnspec (f) -> match arity f with
    | Z -> let rl = unspecdecl f [] in
           (* oracle picking the correct cvalue *)
           let r = pick rl in
           r
    | S m -> CVUnspec(f, m, [])
| TConstr (c, t') ->
  let r = eval_trm t' e in
  CVConstr (c, r)
| TTuple (tl) ->
  let rl = eval_trmlst tl e in
  CVTuple (rl)
| TFunc (p, sk) -> CVClos (e, p, sk)
| TField (t', d) ->
  let r = eval_trm t' e in
  getfield r d
| TNth (t', n) ->
  let r = eval_trm t' e in
  getnth r n
| TRec (topt, ldt) -> match topt with
  | Some t' -> let r = eval_trm t' e in
               uprec r ldt e
  | None -> let ldr = eval_ldt ldt e in
            CVRec (ldr)
```

```
def uprec r ldt e : cvalue = match r with
| CVRec ldr1 ->
  let ldr2 = eval_ldt ldt e in
  CVRec (ldr2++ldr1)

def getfield r d : cvalue = match r with
| CVRec ldr -> lookup ldr d

def getnth r n : cvalue = match r with
| TTuple rl ->
  let ropt = nth rl n in
  getsome ropt

def getsome ropt : cvalue = match ropt with
| Some r -> r

def eval_ldt ldt e : (string * cvalue) list = match ldt with
| [] -> []
| (d, t)::ldt' ->
  let r = eval_trm t e in
  let ldr = eval_ldt ldt' e in
  (d, r)::ldr

def lookup e x : cvalue = match e with
| (y, r)::e2 ->
  if x=y then r
         else lookup e2 x

def eval_trmlst tl e : cvalue list = match tl with
| [] -> []
| t::l ->
  let r = eval_trm t e in
  let rl = eval_trmlst l e in
  r::rl

def eval_sk sk e : cvalue = match sk with
| Branching (skl) ->
  (* oracle picking the correct skeleton *)
  let sk' = pick skl in
  eval_sk sk' e
| LetIn (p, sk1, sk2) ->
  let r = eval_sk sk1 e in
  let e2 = eval_pat p r e in
  eval_sk sk2 e2
| Return t -> eval_trm t e
```

```
| Apply (t, arglist) ->
  let r = eval_trm t e in
  let rl = eval_trmlst arglist e in
  apply_res r rl
| Exists (p, sk) ->
  (* different oracle for the existential *)
  let r = magic () in
  let e2 = eval_pat p r e in
  eval_sk sk e2

def apply_res r arglist = match arglist with
| [] -> r
| arg::al -> match r with
  | CVUnspec (f, Z, parg) ->
    let rl = unspecdecl f (parg++[arg]) in
    (* oracle picking the correct cvalue *)
    let r' = pick rl in
    apply_res r' al
  | CVUnspec (f, S n, parg) ->
    apply_res (CVUnspec (f, n, parg++[arg])) al
  | CVClos (e, p, sk) ->
    let e' = eval_pat p arg e in
    let r' = eval_sk sk e' in
    apply_res r' al

def eval_pat p r e : env = match p, r with
| PWild, _ -> e
| PVar (x), _ -> (x, r)::e
| PConstr (c, p2), CVConstr (c2, r2) when (c=c2) ->
  eval_pat p2 r2 e
| PTuple (pl), CVTuple (rl) ->
  eval_patlst pl rl e
| PRec (ldp), CVRec (ldr) ->
  eval_patldp ldp ldr e

def eval_patldp ldp ldr e : env = match ldp with
| [] -> e
| (d, p)::ldp' ->
  let r = lookup ldr d in
  let e2 = eval_pat p r e in
  eval_patldp ldp' ldr e2

def eval_patlst pl rl e : env = match pl, rl with
| [], [] -> e
| p::l1, r::l2 ->
```

```
  let e2 = eval_pat p r e in
  eval_patlst l1 l2 e2
```

D.1.2 CPS Transform

```
def eval_trm (t : skelterm) (e : env) (k : cvalue -> cvalue) : cvalue =
match t with
| TVar (v) -> match v with
  | VLet (x) -> lookup e x k
  | VSpec (h) -> eval_trm (specdecl h) [] k
  | VUnspec (f) -> match arity f with
    | Z -> let r = pick (unspecdecl f []) in
           k r
    | S m -> k (CVUnspec (f, m, []))
| TConstr (c, t') ->
  eval_trm t' e (fun r ->
  k (CVConstr (c, r)))
| TTuple (tl) ->
  eval_trmlst tl e (fun rl ->
  k (CVTuple (rl)))
| TFunc (p, sk) -> k (CVClos (e, p, sk))
| TField (t', d) ->
  eval_trm t' e (fun r ->
  getfield r d k)
| TNth (t', n) ->
  eval_trm t' e (fun r ->
  getnth r n k)
| TRec (topt, ldt) -> match topt with
  | Some t' -> eval_trm t' e (fun r ->
               uprec r ldt e k)
  | None -> eval_ldt ldt e (fun ldr ->
            k (CVRec (ldr)))

def uprec (r : cvalue) (ldt : (string * skelterm) list)
  (e : env) (k : cvalue -> cvalue) : cvalue = match r with
| CVRec ldr1 ->
  eval_ldt ldt e (fun ldr2 ->
  k (CVRec (ldr2++ldr1)))

def getfield (r : cvalue) (d : string) (k : cvalue -> cvalue) : cvalue =
match r with
| CVRec ldr -> lookup ldr d k

def getnth (r : cvalue) (n : int) (k : cvalue -> cvalue) : cvalue =
match r with
```

```
| TTuple rl ->
  getsome (nth rl n) k

def getsome (ropt : cvalue option) (k : cvalue -> cvalue) : cvalue =
match ropt with
| Some r -> k r

def eval_ldt (ldt : (string * skelterm) list) (e : env)
  (k : (string * cvalue) list -> cvalue) : cvalue = match ldt with
| [] -> k []
| (d, t)::ldt' ->
  eval_trm t e (fun r ->
  eval_ldt ldt' e (fun ldr ->
  k ((d, r)::ldr)))

def lookup (e : env) (x : string) (k : cvalue -> cvalue) : cvalue =
match e with
| (y, r)::e2 ->
  if x=y then k r
         else lookup e2 x k

def eval_trmlst (tl : skelterm list) (e : env)
  (k : cvalue list -> cvalue) : cvalue = match tl with
| [] -> k []
| t::l ->
  eval_trm t e (fun r ->
  eval_trmlst l e (fun rl ->
  k (r::rl)))

def eval_sk (sk : skeleton) (e : env) (k : cvalue -> cvalue) : cvalue =
match sk with
| Branching (skl) ->
  let sk' = pick skl in
  eval_sk sk' e k
| LetIn (p, sk1, sk2) ->
  eval_sk sk1 e (fun r ->
  eval_pat p r e (fun e2 ->
  eval_sk sk2 e2 k))
| Return t -> eval_trm t e k
| Apply (t, arglist) ->
  eval_trm t e (fun r ->
  eval_trmlst arglist e (fun rl ->
  apply_res r rl k))
| Exists (p, sk) ->
  let r = magic () in
```

```
  eval_pat p r e (fun e2 ->
  eval_sk sk e2 k)

def apply_res (r : cvalue) (arglist : cvalue list)
  (k : cvalue -> cvalue) : cvalue = match arglist with
| [] -> k r
| arg::al -> match r with
  | CVUnspec (f, Z, parg) ->
    let r' = pick (unspecdecl f (parg++[arg])) in
    apply_res r' al k
  | CVUnspec (f, S n, parg) ->
    apply_res (CVUnspec (f, n, parg++[arg])) al k
  | CVClos (e, p, sk) ->
    eval_pat p arg e (fun e' ->
    eval_sk sk e' (fun r' ->
    apply_res r' al k))

def eval_pat (p : pattern) (r : cvalue) (e : env)
  (k : env -> cvalue) : cvalue  = match p, r with
| PWild, _ -> k e
| PVar (x), _ -> k ((x, r)::e)
| PConstr (c, p2), CVConstr (c2, r2) when (c=c2) ->
  eval_pat p2 r2 e k
| PTuple (pl), CVTuple (rl) ->
  eval_patlst pl rl e k
| PRec (ldp), CVRec (ldr) ->
  eval_patldp ldp ldr e k

def eval_patldp (ldp : (string * pattern) list)
  (ldr : (string * cvalue) list) (e : env) (k : env -> cvalue) : cvalue =
match ldp with
| [] -> k e
| (d, p)::ldp' ->
  lookup ldr d (fun r ->
  eval_pat p r e (fun e2 ->
  eval_patldp ldp' ldr e2 k))

def eval_patlst (pl : pattern list) (rl : cvalue list) (e : env)
  (k : env -> cvalue) : cvalue = match pl, rl with
| [], [] -> k e
| p::l1, r::l2 ->
  eval_pat p r e (fun e2 ->
  eval_patlst l1 l2 e2 k)
```

D.1.3 Defunctionalization

```
(* corresponds to functions (cvalue -> cvalue) *)
type krt =
| KRID
| KRConstr of string * krt
| KRList of (skelterm list) * env * klt
| KRLet of pattern * skeleton * env * krt
| KRApp1 of (skelterm list) * env * krt
| KRApp3 of (cvalue list) * krt
| KRField of string * krt
| KRNth of int * krt
| KRUp of ((string * skelterm) list) * env * krt
| KRLdt of ((string * skelterm) list) * env * string * kdt
| KRLdp of pattern * env * ((string * pattern) list)
          * ((string * cvalue) list) * ket

def disp_kr (k : krt) (r : cvalue) = match k with
| KRConstr (c, k') -> disp_kr k' (CVConstr (c, r))
| KRList (l, e, k') -> eval_trmlst l e KLList(k', r)
| KRLet (p, sk, e, k') -> eval_pat p r e KELet(sk, k')
| KRApp1 (al, e, k') -> eval_trmlst al e KLApp2(r, k')
| KRApp3 (al, k') -> apply_res r al k'
| KRField (d, k') -> getfield r d k'
| KRNth (n, k') -> getnth r n k'
| KRUp (ldt, e, k') -> uprec r ldt e k'
| KRLdt (ldt, e, d, k') -> eval_ldt ldt e KDCons(d, r, k')
| KRLdp (p, e, ldp, ldr, k') -> eval_pat p r e KELdp(ldp, ldr, k')

(* corresponds to functions ((string * cvalue) list -> cvalue) *)
type kdt =
| KDID
| KDRec of krt
| KDUp of ((string * cvalue) list) * krt
| KDCons of string * cvalue * kdt

def disp_kd (k : kdt) (ldr : (string * cvalue) list) = match k with
| KDRec (k') -> disp_kr k' (CVRec (ldr))
| KDUp (ldr1, k') -> disp_kr k' (CVRec (ldr++ldr1))
| KDCons (d, r, k') -> disp_kd k' ((d, r)::ldr)

(* corresponds to functions (cvalue list -> cvalue) *)
type klt =
| KLID
| KLTuple of krt
| KLList of klt * cvalue
```

```
| KLApp2 of cvalue * krt

def disp_kl (k : klt) (rl : cvalue list) = match k with
| KLTuple k' -> disp_kr k' (CVTuple (rl))
| KLList (k', r) -> disp_kl k' (r::rl)
| KLApp2 (r, k') -> apply_res r rl k'

(* corresponds to functions (env -> cvalue) *)
type ket =
| KEID
| KELet of skeleton * krt
| KEPat of (pattern list) * (cvalue list) * ket
| KEApp of skeleton * (cvalue list) * krt
| KELdp of ((string * pattern) list) * ((string * cvalue) list) * ket

def disp_ke (k : ket) (e : env) = match k with
| KELet (sk, k') -> eval_sk sk e k'
| KEPat (l, rl, k') -> eval_patlst l rl e k'
| KEApp (sk, al, k') -> eval_sk sk e KRApp3(al, k')
| KELdp (ldp, ldr, k') -> eval_patldp ldp ldr e k'

def eval_trm (t : skelterm) (e : env) (k : krt) : cvalue =
match t with
| TVar (v) -> match v with
  | VLet (x) -> lookup e x k
  | VSpec (h) -> eval_trm (specdecl h) [] k
  | VUnspec (f) -> match arity f with
    | Z -> let r = (unspecdecl f []) in
           disp_kr k r
    | S m -> disp_kr k (CVUnspec (f, m, []))
| TConstr (c, t') ->
  eval_trm t' e KRConstr(c, k)
| TTuple (tl) ->
  eval_trmlst tl e KLTuple(k)
| TFunc (p, sk) -> disp_kr k (CVClos (e, p, sk))
| TField (t', d) ->
  eval_trm t' e KRField(d, k)
| TNth (t', n) ->
  eval_trm t' e KRNth(n, k)
| TRec (topt, ldt) -> match topt with
  | Some t' -> eval_trm t' e KRUp(ldt, e, k)
  | None -> eval_ldt ldt e KDRec(k)

def uprec (r : cvalue) (ldt : (string * skelterm) list)
  (e : env) (k : krt) : cvalue = match r with
```

```
| CVRec ldr1 ->
  eval_ldt ldt e KDUp(ldr1, k)

def getfield (r : cvalue) (d : string) (k : krt) : cvalue =
match r with
| CVRec ldr -> lookup ldr d k

def getnth (r : cvalue) (n : int) (k : krt) : cvalue =
match r with
| TTuple rl ->
  getsome (nth rl n) k

def getsome (ropt : cvalue option) (k : krt) : cvalue =
match ropt with
| Some r -> disp_kr k r

def eval_ldt (ldt : (string * skelterm) list)
  (e : env) (k : kdt) : cvalue = match ldt with
| [] -> disp_kd k []
| (d, t)::ldt' ->
  eval_trm t e KRLdt(ldt', e, d, k)

def lookup (e : env) (x : string) (k : krt) : cvalue =
match e with
| (y, r)::e2 ->
  if x=y then disp_kr k r
         else lookup e2 x k

def eval_trmlst (tl : skelterm list) (e : env) (k : klt) : cvalue =
match tl with
| [] -> disp_kl k []
| t::l -> eval_trm t e KRList(l, e, k)

def eval_sk (sk : skeleton) (e : env) (k : krt) : cvalue =
match sk with
| Branching (skl) ->
  let sk' = pick skl in
  eval_sk sk' e k
| LetIn (p, sk1, sk2) ->
  eval_sk sk1 e KRLet(p, sk2, e, k)
| Return t -> eval_trm t e k
| Apply (t, arglist) ->
  eval_trm t e KRApp1(arglist, e, k)
| Exists (p, sk) ->
  let r = magic () in
```

```
  eval_pat p r e KELet(sk, k)

def apply_res (r : cvalue) (argl : cvalue list) (k : krt) : cvalue =
match argl with
| [] -> disp_kr k r
| arg::al -> match r with
  | CVUnspec (f, Z, parg) ->
    let r' = pick (unspecdecl f (parg++[arg])) in
    apply_res r' al k
  | CVUnspec (f, S n, parg) ->
    apply_res (CVUnspec (f, n, parg++[arg])) al k
  | CVClos (e, p, sk) ->
    eval_pat p arg e KEApp(sk, al, k)

def eval_pat (p : pattern) (r : cvalue) (e : env) (k : ket) : cvalue =
match p, r with
| PWild, _ -> disp_ke k e
| PVar (x), _ -> disp_ke k ((x, r)::e)
| PConstr (c, p2), CVConstr (c2, r2) when (c=c2) ->
  eval_pat p2 r2 e k
| PTuple (pl), CVTuple (rl) ->
  eval_patlst pl rl e k
| PRec (ldp), CVRec (ldr) ->
  eval_patldp ldp ldr e k

def eval_patldp (ldp : (string * pattern) list)
  (ldr : (string * cvalue) list) (e : env) (k : ket) : cvalue =
match ldp with
| [] -> disp_ke k e
| (d, p)::ldp' ->
  lookup ldr d KRLdp(p, e, ldp', ldr, k)

def eval_patlst (pl : pattern list) (rl : cvalue list) (e : env)
  (k : ket) : cvalue = match pl, rl with
| [], [] -> disp_ke k e
| p::l1, r::l2 ->
  eval_pat p r e KEPat(l1, l2, k)
```

D.1.4 Non-Deterministic Abstract Machine

Mode **kr**:

$$\begin{aligned}
\langle \mathrm{kr}_{\mathrm{id}}, r\rangle_{\mathrm{kr}} &\not\to \texttt{(* end of computation *)}\\
\langle \texttt{KRConstr}(c,k), r\rangle_{\mathrm{kr}} &\to \langle k, \texttt{CVConstr}(c,r)\rangle_{\mathrm{kr}}\\
\langle \texttt{KRList}(l,\Sigma,k), r\rangle_{\mathrm{kr}} &\to \langle l, \Sigma, \texttt{KLList}(k,r)\rangle_{\mathrm{trml}}\\
\langle \texttt{KRLet}(p,S,\Sigma,k), r\rangle_{\mathrm{kr}} &\to \langle p, r, \Sigma, \texttt{KELet}(S,k)\rangle_{\mathrm{pat}}\\
\langle \texttt{KRApp1}(l,\Sigma,k), r\rangle_{\mathrm{kr}} &\to \langle l, \Sigma, \texttt{KLApp2}(r,k)\rangle_{\mathrm{trml}}\\
\langle \texttt{KRApp3}(l,k), r\rangle_{\mathrm{kr}} &\to \langle l, r, k\rangle_{\mathrm{app}}\\
\langle \texttt{KRField}(d,k), r\rangle_{\mathrm{kr}} &\to \langle r, d, k\rangle_{\mathrm{getf}}\\
\langle \texttt{KRNth}(n,k), r\rangle_{\mathrm{kr}} &\to \langle r, n, k\rangle_{\mathrm{getn}}\\
\langle \texttt{KRUp}(l_{dt},\Sigma,k), r\rangle_{\mathrm{kr}} &\to \langle r, l_{dt}, \Sigma, k\rangle_{\mathrm{uprec}}\\
\langle \texttt{KRLdt}(l_{dt},\Sigma,d,k), r\rangle_{\mathrm{kr}} &\to \langle l_{dt}, \Sigma, \texttt{KDCons}(d,r,k)\rangle_{\mathrm{ldt}}\\
\langle \texttt{KRLdt}(p,\Sigma,l_{dp},l_{dr},k), r\rangle_{\mathrm{kr}} &\to \langle p, r, \Sigma, \texttt{KELdp}(l_{dp},l_{dr},k)\rangle_{\mathrm{pat}}
\end{aligned}$$

Mode **kd**:

$$\begin{aligned}
\langle \mathrm{kd}_{\mathrm{id}}, l_{dr}\rangle_{\mathrm{kd}} &\not\to \texttt{(* end of computation *)}\\
\langle \texttt{KDRec}(k), l_{dr}\rangle_{\mathrm{kd}} &\to \langle k, \texttt{CVRec}(l_{dr})\rangle_{\mathrm{kr}}\\
\langle \texttt{KDUp}(l'_{dr}, k), l_{dr}\rangle_{\mathrm{kd}} &\to \langle k, \texttt{CVRec}(l_{dr} +\!\!+ l'_{dr})\rangle_{\mathrm{kr}}\\
\langle \texttt{KDCons}(d,r,k), l_{dr}\rangle_{\mathrm{kd}} &\to \langle k, ((d,r) :: l_{dr})\rangle_{\mathrm{kd}}
\end{aligned}$$

Mode **kl**:

$$\begin{aligned}
\langle \mathrm{kl}_{\mathrm{id}}, l\rangle_{\mathrm{kl}} &\not\to \texttt{(* end of computation *)}\\
\langle \texttt{KLTuple}(k), l\rangle_{\mathrm{kl}} &\to \langle k, \texttt{CVTuple}(l)\rangle_{\mathrm{kr}}\\
\langle \texttt{KLList}(k,r), l\rangle_{\mathrm{kl}} &\to \langle k, (r :: l)\rangle_{\mathrm{kl}}\\
\langle \texttt{KLApp2}(r,k), l\rangle_{\mathrm{kl}} &\to \langle l, r, k\rangle_{\mathrm{app}}
\end{aligned}$$

Mode **ke**:

$$\begin{aligned}
\langle \mathrm{ke}_{\mathrm{id}}, \Sigma\rangle_{\mathrm{ke}} &\not\to \texttt{(* end of computation *)} \\
\langle \texttt{KELet}(S,k), \Sigma\rangle_{\mathrm{ke}} &\to \langle S, \Sigma, k\rangle_{\mathrm{sk}} \\
\langle \texttt{KEPat}(l_p, l_r, k), \Sigma\rangle_{\mathrm{ke}} &\to \langle l_p, l_r, \Sigma, k\rangle_{\mathrm{patl}} \\
\langle \texttt{KEApp}(S, l_a, k), \Sigma\rangle_{\mathrm{ke}} &\to \langle S, \Sigma, \texttt{KRApp3}(l_a, k)\rangle_{\mathrm{sk}} \\
\langle \texttt{KELdp}(l_{dp}, l_{dr}, k), \Sigma\rangle_{\mathrm{ke}} &\to \langle l_{dp}, l_{dr}, \Sigma, k\rangle_{\mathrm{patldp}}
\end{aligned}$$

Mode **trm**:

$$\begin{aligned}
\langle \texttt{TVar}(\texttt{VLet}(x)), \Sigma, k\rangle_{\mathrm{trm}} &\to \langle \Sigma, x, k\rangle_{\mathrm{lkup}} \\
\langle \texttt{TVar}(\texttt{VSpec}(x)), \Sigma, k\rangle_{\mathrm{trm}} &\to \langle \texttt{SpecDecl}(x), [], k\rangle_{\mathrm{trm}} \\
\langle \texttt{TVar}(\texttt{VUnspec}(x)), \Sigma, k\rangle_{\mathrm{trm}} &\to \langle k, r\rangle_{\mathrm{kr}} \qquad \texttt{if Arity}(x) = 0 \wedge r \in \texttt{UnspecDecl}(x)[] \\
\langle \texttt{TVar}(\texttt{VUnspec}(x)), \Sigma, k\rangle_{\mathrm{trm}} &\to \langle k, \texttt{CVUnspec}(x, m, [])\rangle_{\mathrm{kr}} \qquad \texttt{if Arity}(x) = m+1 \\
\langle \texttt{TConstr}(c,t), \Sigma, k\rangle_{\mathrm{trm}} &\to \langle t, \Sigma, \texttt{KRConstr}(c,k)\rangle_{\mathrm{trm}} \\
\langle \texttt{TTuple}(l), \Sigma, k\rangle_{\mathrm{trm}} &\to \langle l, \Sigma, \texttt{KLTuple}(k)\rangle_{\mathrm{trml}} \\
\langle \texttt{TFunc}(p,S), \Sigma, k\rangle_{\mathrm{trm}} &\to \langle k, \texttt{CVClos}(\Sigma, p, S)\rangle_{\mathrm{kr}} \\
\langle \texttt{TField}(t,d), \Sigma, k\rangle_{\mathrm{trm}} &\to \langle t, \Sigma, \texttt{KRField}(d,k)\rangle_{\mathrm{trm}} \\
\langle \texttt{TNth}(t,n), \Sigma, k\rangle_{\mathrm{trm}} &\to \langle t, \Sigma, \texttt{KRNth}(n,k)\rangle_{\mathrm{trm}} \\
\langle \texttt{TRec}(\texttt{Some}(t), l_{dt}), \Sigma, k\rangle_{\mathrm{trm}} &\to \langle t, \Sigma, \texttt{KRUp}(l_{dt}, \Sigma, k)\rangle_{\mathrm{trm}} \\
\langle \texttt{TRec}(\texttt{None}, l_{dt}), \Sigma, k\rangle_{\mathrm{trm}} &\to \langle l_{dt}, \Sigma, \texttt{KDRec}(k)\rangle_{\mathrm{ldt}}
\end{aligned}$$

Modes **uprec**, **getf**, **getn**, and **gets**:

$$\begin{aligned}
\langle \texttt{CVRec}(l_{dr}), l_{dt}, \Sigma, k\rangle_{\mathrm{uprec}} &\to \langle l_{dt}, \Sigma, \texttt{KDUp}(l_{dr}, k)\rangle_{\mathrm{ldt}} \\
\langle \texttt{CVRec}(l_{dr}), d, k\rangle_{\mathrm{getf}} &\to \langle l_{dr}, d, k\rangle_{\mathrm{lkup}} \\
\langle \texttt{CVTuple}(l_r), n, k\rangle_{\mathrm{getn}} &\to \langle (\texttt{nth } l_r\ n), k\rangle_{\mathrm{gets}} \\
\langle \texttt{Some}(r), k\rangle_{\mathrm{gets}} &\to \langle k, r\rangle_{\mathrm{kr}}
\end{aligned}$$

Note: `nth` is defined as:

$$\begin{aligned}
\texttt{nth}\ [x_0; \ldots; x_m]\ n &\triangleq \texttt{Some}(x_n) \qquad \texttt{if } n \le m \\
\texttt{nth}\ [x_0; \ldots; x_m]\ n &\triangleq \texttt{None} \qquad \texttt{if } n > m
\end{aligned}$$

Mode **ldt**:

$$\langle [], \Sigma, k\rangle_{\text{ldt}} \to \langle k, []\rangle_{\text{kd}}$$
$$\langle (d,t) :: l_{dt}, \Sigma, k\rangle_{\text{ldt}} \to \langle t, \Sigma, \texttt{KRLdt}(l_{dt}, \Sigma, d, k)\rangle_{\text{trm}}$$

Mode **lkup**:

$$\langle (y,r) :: l, x, k\rangle_{\text{lkup}} \to \langle k, r\rangle_{\text{kr}} \qquad \texttt{if } x = y$$
$$\langle (y,r) :: l, x, k\rangle_{\text{lkup}} \to \langle l, x, k\rangle_{\text{lkup}} \qquad \texttt{if } x \neq y$$

Mode **trml**:

$$\langle [], \Sigma, k\rangle_{\text{trml}} \to \langle k, []\rangle_{\text{kl}}$$
$$\langle t :: l, \Sigma, k\rangle_{\text{trml}} \to \langle t, \Sigma, \texttt{KRList}(l, \Sigma, k)\rangle_{\text{trm}}$$

Mode **sk**:

$$\langle \texttt{Branching}(l_S), \Sigma, k\rangle_{\text{sk}} \to \langle S, \Sigma, k\rangle_{\text{sk}} \qquad \texttt{if } S \in l_S$$
$$\langle \texttt{LetIn}(p, S_1, S_2), \Sigma, k\rangle_{\text{sk}} \to \langle S_1, \Sigma, \texttt{KRLet}(p, S_2, \Sigma, k)\rangle_{\text{sk}}$$
$$\langle \texttt{Return}(t), \Sigma, k\rangle_{\text{sk}} \to \langle t, \Sigma, k\rangle_{\text{trm}}$$
$$\langle \texttt{Apply}(t, l), \Sigma, k\rangle_{\text{sk}} \to \langle t, \Sigma, \texttt{KRApp1}(l, \Sigma, k)\rangle_{\text{trm}}$$
$$\langle \texttt{Exists}(p, S), \Sigma, k\rangle_{\text{sk}} \to \langle p, r, \Sigma, \texttt{KELet}(S, k)\rangle_{\text{pat}} \qquad \texttt{for any } r$$

Mode **app**:
(note: arguments swapped to pattern-match the first)

$$\langle [], r, k\rangle_{\text{app}} \to \langle k, r\rangle_{\text{kr}}$$
$$\langle a :: l_a, \texttt{CVUnspec}(x, 0, l), k\rangle_{\text{app}} \to \langle l_a, r, k\rangle_{\text{app}} \qquad \texttt{if } r \in \texttt{UnspecDecl}(x)(l +\!\!+ [a])$$
$$\langle a :: l_a, \texttt{CVUnspec}(x, m+1, l), k\rangle_{\text{app}} \to \langle l_a, \texttt{CVUnspec}(x, m, l +\!\!+ [a]), k\rangle_{\text{app}}$$
$$\langle a :: l_a, \texttt{CVClos}(\Sigma, p, S), k\rangle_{\text{app}} \to \langle p, a, \Sigma, \texttt{KEApp}(S, l_a, k)\rangle_{\text{pat}}$$

Mode **pat**:

$$\begin{aligned}
\langle \texttt{PWild}, r, \Sigma, k\rangle_{\text{pat}} &\to \langle k, \Sigma\rangle_{\text{ke}} \\
\langle \texttt{PVar}(x), r, \Sigma, k\rangle_{\text{pat}} &\to \langle k, (x, r) :: \Sigma\rangle_{\text{ke}} \\
\langle \texttt{PConstr}(c, p), \texttt{CVConstr}(c_2, r), \Sigma, k\rangle_{\text{pat}} &\to \langle p, r, \Sigma, k\rangle_{\text{pat}} \qquad \texttt{if } c = c_2 \\
\langle \texttt{PTuple}(l_p), \texttt{CVTuple}(l_r), \Sigma, k\rangle_{\text{pat}} &\to \langle l_p, l_r, \Sigma, k\rangle_{\text{patl}} \\
\langle \texttt{PRec}(l_{dp}), \texttt{CVRec}(l_{dr}), \Sigma, k\rangle_{\text{pat}} &\to \langle l_{dp}, l_{dr}, \Sigma, k\rangle_{\text{patldp}}
\end{aligned}$$

Mode **patldp**:

$$\begin{aligned}
\langle [], l_{dr}, \Sigma, k\rangle_{\text{patldp}} &\to \langle k, \Sigma\rangle_{\text{ke}} \\
\langle (d, p) :: l_{dp}, l_{dr}, \Sigma, k\rangle_{\text{patldp}} &\to \langle l_{dr}, d, \texttt{KRLdt}(p, \Sigma, l_{dp}, l_{dr}, k)\rangle_{\text{lkup}}
\end{aligned}$$

Mode **patl**:

$$\begin{aligned}
\langle [], [], \Sigma, k\rangle_{\text{patl}} &\to \langle k, \Sigma\rangle_{\text{ke}} \\
\langle p :: l_p, r :: l_r, \Sigma, k\rangle_{\text{patl}} &\to \langle p, r, \Sigma, \texttt{KEPat}(l_p, l_r, k)\rangle_{\text{pat}}
\end{aligned}$$

D.2 Successive Phases of the AM Pseudo-code

We start over from the pseudo-code of Section D.1.1. However, we ignore the `Exists` constructor as we cannot propose an evaluation for it.

D.2.1 CPS Transform

```
def eval_trm (t : skelterm) (e : env)
  (k : cvalue -> (() -> cvalue) -> cvalue)
  (fk : () -> cvalue) : cvalue = match t with
| TVar (v) -> match v with
  | VLet (x) -> lookup e x k fk
  | VSpec (h) -> eval_trm (specdecl h) [] k fk
  | VUnspec (f) -> match arity f with
    | Z -> select_list (unspecdecl f []) k fk
    | S m -> k (CVUnspec (f, m, [])) fk
| TConstr (c, t') ->
  eval_trm t' e (fun r fk2 ->
  k (CVConstr (c, r)) fk2) fk
| TTuple (tl) ->
  eval_trmlst tl e (fun rl fk2 ->
  k (CVTuple (rl)) fk2) fk
| TFunc (p, sk) -> k (CVClos (e, p, sk)) fk
```

```
| TField (t', d) ->
  eval_trm t' e (fun r fk2 ->
  getfield r d k fk2) fk
| TNth (t', n) ->
  eval_trm t' e (fun r fk2 ->
  getnth r n k fk2) fk
| TRec (topt, ldt) -> match topt with
  | Some t' -> eval_trm t' e (fun r fk2 ->
               uprec r ldt e k fk2) fk
  | None -> eval_ldt ldt e (fun ldr fk2 ->
            k (CVRec (ldr)) fk2) fk

def uprec (r : cvalue) (ldt : (string * skelterm) list) (e : env)
  (k : cvalue -> (() -> cvalue) -> cvalue)
  (fk : () -> cvalue) : cvalue = match r with
| CVRec ldr1 ->
  eval_ldt ldt e (fun ldr2 fk2 ->
  k (CVRec (ldr2++ldr1)) fk2) fk
| _ -> fk ()

def getfield (r : cvalue) (d : string)
  (k : cvalue -> (() -> cvalue) -> cvalue)
  (fk : () -> cvalue) : cvalue = match r with
| CVRec ldr -> lookup ldr d k fk
| _ -> fk ()

def getnth (r : cvalue) (n : int)
  (k : cvalue -> (() -> cvalue) -> cvalue)
  (fk : () -> cvalue) : cvalue = match r with
| TTuple rl ->
  getsome (nth rl n) k fk
| _ -> fk ()

def getsome (ropt : cvalue option)
  (k : cvalue -> (() -> cvalue) -> cvalue)
  (fk : () -> cvalue) : cvalue = match ropt with
| Some r -> k r fk
| None -> fk ()

def eval_ldt (ldt : (string * skelterm) list) (e : env)
  (k : (string * cvalue) list -> (() -> cvalue) -> cvalue)
  (fk : () -> cvalue) : cvalue = match ldt with
| [] -> k [] fk
| (d, t)::ldt' ->
  eval_trm t e (fun r fk2 ->
```

```
  eval_ldt ldt' e (fun ldr fk3 ->
  k ((d, r)::ldr) fk3) fk2) fk

def lookup (e : env) (x : string)
  (k : cvalue -> (() -> cvalue) -> cvalue)
  (fk : () -> cvalue) : cvalue = match e with
| [] -> fk ()
| (y, r)::e2 ->
  if x=y then k r fk
         else lookup e2 x k fk

def eval_trmlst (tl : skelterm list) (e : env)
  (k : cvalue list -> (() -> cvalue) -> cvalue)
  (fk : () -> cvalue) : cvalue = match tl with
| [] -> k [] fk
| t::l ->
  eval_trm t e (fun r fk2 ->
  eval_trmlst l e (fun rl fk3 ->
  k (r::rl) fk3) fk2) fk

(* Note: there is no rule for Exists as we cannot compute it *)
def eval_sk (sk : skeleton) (e : env)
  (k : cvalue -> (() -> cvalue) -> cvalue)
  (fk : () -> cvalue) : cvalue = match sk with
| Branching (skl) -> match skl with
  | [] -> fk ()
  | sk'::l -> eval_sk sk' e k
              (fun _ -> eval_sk (Branching l) e k fk)
| LetIn (p, sk1, sk2) ->
  eval_sk sk1 e (fun r fk2 ->
  eval_pat p r e (fun e2 fk3 ->
  eval_sk sk2 e2 k fk3) fk2) fk
| Return t -> eval_trm t e k fk
| Apply (t, arglist) ->
  eval_trm t e (fun r fk2 ->
  eval_trmlst arglist e (fun rl fk3 ->
  apply_res r rl k fk3) fk2) fk

def apply_res (r : cvalue) (arglist : cvalue list)
  (k : cvalue -> (() -> cvalue) -> cvalue)
  (fk : () -> cvalue) : cvalue = match arglist with
| [] -> k r fk
| arg::al -> match r with
  | CVUnspec (f, Z, parg) ->
    let rl = unspecdecl f (parg++[arg]) in
```

```
    select_list rl (fun r' fk2 ->
    apply_res r' al k fk2) fk
  | CVUnspec (f, S n, parg) ->
    apply_res (CVUnspec (f, n, parg++[arg])) al k fk
  | CVClos (e, p, sk) ->
    eval_pat p arg e (fun e' fk2 ->
    eval_sk sk e' (fun r' fk3 ->
    apply_res r' al k fk3) fk2) fk
  | _ -> fk ()

def select_list (rl : cvalue list)
  (k : cvalue -> (() -> cvalue) -> cvalue)
  (fk : () -> cvalue) : cvalue = match rl with
| [] -> fk ()
| r::l -> k r (fun _ -> select_list l k fk)

def eval_pat (p : pattern) (r : cvalue) (e : env)
  (k : env -> (() -> cvalue) -> cvalue)
  (fk : () -> cvalue) : cvalue = match p with
| PWild -> k e fk
| PVar (x) -> k ((x, r)::e) fk
| PConstr (c, p2) -> match r with
  | CVConstr (c2, r2) ->
    if c=c2 then eval_pat p2 r2 e k fk
            else fk ()
  | _ -> fk ()
| PTuple (pl) -> match r with
  | CVTuple (rl) -> eval_patlst pl rl e k fk
  | _ -> fk ()
| PRec (ldp) -> match r with
  | CVRec (ldr) -> eval_patldp ldp ldr e k fk
  | _ -> fk ()

def eval_patldp (ldp : (string * pattern) list)
  (ldr : (string * cvalue) list) (e : env)
  (k : env -> (() -> cvalue) -> cvalue)
  (fk : () -> cvalue) : cvalue  = match ldp with
| [] -> k e fk
| (d, p)::ldp' ->
  lookup ldr d (fun r fk2 ->
  eval_pat p r e (fun e2 fk3 ->
  eval_patldp ldp' ldr e2 k fk3) fk2) fk

def eval_patlst (pl : pattern list) (rl : cvalue list) (e : env)
  (k : env -> (() -> cvalue) -> cvalue)
```

```
  (fk : () -> cvalue) : cvalue = match pl with
| [] -> match rl with
  | [] -> k e fk
  | _ -> fk ()
| p::l1 -> match rl with
  | [] -> fk ()
  | r::l2 ->
  eval_pat p r e (fun e2 fk2 ->
  eval_patlst l1 l2 e2 k fk2) fk
```

D.2.2 Defunctionalization

```
(* corresponds to functions (cvalue -> (() -> cvalue) -> cvalue) *)
type krt =
| KRID
| KRConstr of string * krt
| KRList of (skelterm list) * env * klt
| KRLet of pattern * skeleton * env * krt
| KRApp1 of (skelterm list) * env * krt
| KRApp3 of (cvalue list) * krt
| KRField of string * krt
| KRNth of int * krt
| KRUp of ((string * skelterm) list) * env * krt
| KRLdt of ((string * skelterm) list) * env * string * kdt
| KRLdp of pattern * env * ((string * pattern) list)
           * ((string * cvalue) list) * ket

def disp_kr (k : krt) (r : cvalue) (fk : fkt) =
match k with
| KRConstr (c, k') -> disp_kr k' (CVConstr (c, r)) fk
| KRList (l, e, k') -> eval_trmlst l e KLList(k', r) fk
| KRLet (p, sk, e, k') -> eval_pat p r e KELet(sk, k') fk
| KRApp1 (al, e, k') -> eval_trmlst al e KLApp2(r, k') fk
| KRApp3 (al, k') -> apply_res r al k' fk
| KRField (d, k') -> getfield r d k' fk
| KRNth (n, k') -> getnth r n k' fk
| KRUp (ldt, e, k') -> uprec r ldt e k' fk
| KRLdt (ldt, e, d, k') -> eval_ldt ldt e KDCons(d, r, k') fk
| KRLdp (p, e, ldp, ldr, k') -> eval_pat p r e KELdp(ldp, ldr, k') fk

(* corresponds to functions of type
   ((string * cvalue) list -> (() -> cvalue) -> cvalue) *)
type kdt =
| KDID
| KDRec of krt
```

```
| KDUp of ((string * cvalue) list) * krt
| KDCons of string * cvalue * kdt

def disp_kd (k : kdt) (ldr : (string * cvalue) list) (fk : fkt) =
match k with
| KDRec (k') -> disp_kr k' (CVRec (ldr)) fk
| KDUp (ldr1, k') -> disp_kr k' (CVRec (ldr++ldr1)) fk
| KDCons (d, r, k') -> disp_kd k' ((d, r)::ldr) fk

(* corresponds to functions (cvalue list -> (() -> cvalue) -> cvalue) *)
type klt =
| KLID
| KLTuple of krt
| KLList of klt * cvalue
| KLApp2 of cvalue * krt

def disp_kl (k : klt) (rl : cvalue list) (fk : fkt) =
match k with
| KLTuple k' -> disp_kr k' (CVTuple (rl)) fk
| KLList (k', r) -> disp_kl k' (r::rl) fk
| KLApp2 (r, k') -> apply_res r rl k' fk

(* corresponds to functions (env -> (() -> cvalue) -> cvalue) *)
type ket =
| KEID
| KELet of skeleton * krt
| KEPat of (pattern list) * (cvalue list) * ket
| KEApp of skeleton * (cvalue list) * krt
| KELdp of ((string * pattern) list) * ((string * cvalue) list) * ket

def disp_ke (k : ket) (e : env) (fk : fkt) =
match k with
| KELet (sk, k') -> eval_sk sk e k' fk
| KEPat (l, rl, k') -> eval_patlst l rl e k' fk
| KEApp (sk, al, k') -> eval_sk sk e KRApp3(al, k') fk
| KELdp (ldp, ldr, k') -> eval_patldp ldp ldr e k' fk

(* corresponds to functions (() -> cvalue) *)
type fkt =
| FEmpty
| FSK of skeleton * env * krt * fkt
| FList of (cvalue list) * krt * fkt

def disp_fk (fk : fkt) = match fk with
| FSK(sk, e, k, fk') -> eval_sk sk e k fk'
```

```
| FList(rl, k, fk') -> select_list rl k fk'

def eval_trm (t : skelterm) (e : env)
  (k : krt) (fk : fkt) : cvalue = match t with
| TVar (v) -> match v with
  | VLet (x) -> lookup e x k fk
  | VSpec (h) -> eval_trm (specdecl h) [] k fk
  | VUnspec (f) -> match arity f with
    | Z -> select_list (unspecdecl f []) k fk
    | S m -> disp_kr k (CVUnspec (f, m, [])) fk
| TConstr (c, t') ->
  eval_trm t' e KRConstr(c, k) fk
| TTuple (tl) ->
  eval_trmlst tl e KLTuple(k) fk
| TFunc (p, sk) -> disp_kr k (CVClos (e, p, sk)) fk
| TField (t', d) ->
  eval_trm t' e KRField(d, k) fk
| TNth (t', n) ->
  eval_trm t' e KRNth(n, k) fk
| TRec (topt, ldt) -> match topt with
  | Some t' -> eval_trm t' e KRUp(ldt, e, k) fk
  | None -> eval_ldt ldt e KDRec(k) fk

def uprec (r : cvalue) (ldt : (string * skelterm) list) (e : env)
  (k : krt) (fk : fkt) : cvalue = match r with
| CVRec ldr1 ->
  eval_ldt ldt e KDUp(ldr1, k) fk
| _ -> disp_fk fk

def getfield (r : cvalue) (d : string)
  (k : krt) (fk : fkt) : cvalue = match r with
| CVRec ldr -> lookup ldr d k fk
| _ -> disp_fk fk

def getnth (r : cvalue) (n : int)
  (k : krt) (fk : fkt) : cvalue = match r with
| TTuple rl ->
  getsome (nth rl n) k fk
| _ -> disp_fk fk

def getsome (ropt : cvalue option)
  (k : krt) (fk : fkt) : cvalue = match ropt with
| Some r -> disp_kr k r fk
| _ -> disp_fk fk
```

```
def eval_ldt (ldt : (string * skelterm) list) (e : env)
  (k : kdt) (fk : fkt) : cvalue = match ldt with
| [] -> disp_kd k [] fk
| (d, t)::ldt' ->
  eval_trm t e KRLdt(ldt', e, d, k) fk

def lookup (e : env) (x : string)
  (k : krt) (fk : fkt) : cvalue = match e with
| [] -> disp_fk fk
| (y, r)::e2 ->
  if x=y then disp_kr k r fk
         else lookup e2 x k fk

def eval_trmlst (tl : skelterm list) (e : env)
  (k : klt) (fk : fkt) : cvalue = match tl with
| [] -> disp_kl k [] fk
| t::l -> eval_trm t e KRList(l, e, k) fk

def eval_sk (sk : skeleton) (e : env)
  (k : krt) (fk : fkt) : cvalue = match sk with
| Branching (skl) -> match skl with
  | [] -> disp_fk fk
  | sk'::l -> eval_sk sk' e k FSK(Branching l, e, k, fk)
| LetIn (p, sk1, sk2) ->
  eval_sk sk1 e KRLet(p, sk2, e, k) fk
| Return t -> eval_trm t e k fk
| Apply (t, arglist) ->
  eval_trm t e KRApp1(arglist, e, k) fk

def apply_res (r : cvalue) (argl : cvalue list)
  (k : krt) (fk : fkt) : cvalue = match argl with
| [] -> disp_kr k r fk
| arg::al -> match r with
  | CVUnspec (f, Z, parg) ->
    let rl = unspecdecl f (parg++[arg]) in
    select_list rl KRApp3(al, k) fk
  | CVUnspec (f, S n, parg) ->
    apply_res (CVUnspec (f, n, parg++[arg])) al k fk
  | CVClos (e, p, sk) ->
    eval_pat p arg e KEApp(sk, al, k) fk
  | _ -> disp_fk fk

def select_list (rl : cvalue list)
  (k : krt) (fk : fkt) : cvalue = match rl with
| [] -> disp_fk fk
```

```
| r::l -> k r FList(l, k, fk)

def eval_pat (p : pattern) (r : cvalue) (e : env)
  (k : ket) (fk : fkt) : cvalue = match p with
| PWild -> disp_ke k e fk
| PVar (x) -> disp_ke k ((x, r)::e) fk
| PConstr (c, p2) -> match r with
  | CVConstr (c2, r2) ->
    if c=c2 then eval_pat p2 r2 e k fk
            else disp_fk fk
  | _ -> disp_fk fk
| PTuple (pl) -> match r with
  | CVTuple (rl) -> eval_patlst pl rl e k fk
  | _ -> disp_fk fk
| PRec (ldp) -> match r with
  | CVRec (ldr) -> eval_patldp ldp ldr e k fk
  | _ -> disp_fk fk

def eval_patldp (ldp : (string * pattern) list)
  (ldr : (string * cvalue) list) (e : env)
  (k : ket) (fk : fkt) : cvalue = match ldp with
| [] -> disp_ke k e fk
| (d, p)::ldp' ->
  lookup ldr d KRLdp(p, e, ldp', ldr, k) fk

def eval_patlst (pl : pattern list) (rl : cvalue list) (e : env)
  (k : ket) (fk : fkt) : cvalue = match pl with
| [] -> match rl with
  | [] -> disp_ke k e fk
  | _ -> disp_fk fk
| p::l1 -> match rl with
  | [] -> disp_fk fk
  | r::l2 -> eval_pat p r e KEPat(l1, l2, k) fk
```

D.2.3 Deterministic Abstract Machine

Mode **kr**:

$$\langle \mathrm{kr}_{\mathrm{id}}, r, f\rangle_{\mathrm{kr}} \not\to \texttt{(* end of computation *)}$$
$$\langle \texttt{KRConstr}(c,k), r, f\rangle_{\mathrm{kr}} \to \langle k, \texttt{CVConstr}(c,r), f\rangle_{\mathrm{kr}}$$
$$\langle \texttt{KRList}(l,e,k), r, f\rangle_{\mathrm{kr}} \to \langle l, e, \texttt{KLList}(k,r), f\rangle_{\mathrm{trml}}$$
$$\langle \texttt{KRLet}(p,S,\Sigma,k), r, f\rangle_{\mathrm{kr}} \to \langle p, r, \Sigma, \texttt{KELet}(S,k), f\rangle_{\mathrm{pat}}$$
$$\langle \texttt{KRApp1}(l,e,k), r, f\rangle_{\mathrm{kr}} \to \langle l, e, \texttt{KLApp2}(r,k), f\rangle_{\mathrm{trml}}$$
$$\langle \texttt{KRApp3}(l,k), r, f\rangle_{\mathrm{kr}} \to \langle l, r, k, f\rangle_{\mathrm{app}}$$
$$\langle \texttt{KRField}(d,k), r, f\rangle_{\mathrm{kr}} \to \langle r, d, k, f\rangle_{\mathrm{getf}}$$
$$\langle \texttt{KRNth}(n,k), r, f\rangle_{\mathrm{kr}} \to \langle r, n, k, f\rangle_{\mathrm{getn}}$$
$$\langle \texttt{KRUp}(l_{dt},\Sigma,k), r, f\rangle_{\mathrm{kr}} \to \langle r, l_{dt}, \Sigma, k, f\rangle_{\mathrm{uprec}}$$
$$\langle \texttt{KRLdt}(l_{dt},\Sigma,d,k), r, f\rangle_{\mathrm{kr}} \to \langle l_{dt}, \Sigma, \texttt{KDCons}(d,r,k), f\rangle_{\mathrm{ldt}}$$
$$\langle \texttt{KRLdt}(p,\Sigma,l_{dp},l_{dr},k), r, f\rangle_{\mathrm{kr}} \to \langle p, r, \Sigma, \texttt{KELdp}(l_{dp},l_{dr},k), f\rangle_{\mathrm{pat}}$$

Mode **kd**:

$$\langle \mathrm{kd}_{\mathrm{id}}, l_{dr}, f\rangle_{\mathrm{kd}} \not\to \texttt{(* end of computation *)}$$
$$\langle \texttt{KDRec}(k), l_{dr}, f\rangle_{\mathrm{kd}} \to \langle k, \texttt{CVRec}(l_{dr}), f\rangle_{\mathrm{kr}}$$
$$\langle \texttt{KDUp}(l'_{dr},k), l_{dr}, f\rangle_{\mathrm{kd}} \to \langle k, \texttt{CVRec}(l_{dr} +\!\!+ l'_{dr}), f\rangle_{\mathrm{kr}}$$
$$\langle \texttt{KDCons}(d,r,k), l_{dr}, f\rangle_{\mathrm{kd}} \to \langle k, ((d,r) :: l_{dr}), f\rangle_{\mathrm{kd}}$$

Mode **kl**:

$$\langle \mathrm{kl}_{\mathrm{id}}, l, f\rangle_{\mathrm{kl}} \not\to \texttt{(* end of computation *)}$$
$$\langle \texttt{KLTuple}(k), l, f\rangle_{\mathrm{kl}} \to \langle k, \texttt{CVTuple}(l), f\rangle_{\mathrm{kr}}$$
$$\langle \texttt{KLList}(k,r), l, f\rangle_{\mathrm{kl}} \to \langle k, r :: l, f\rangle_{\mathrm{kl}}$$
$$\langle \texttt{KLApp2}(r,k), l, f\rangle_{\mathrm{kl}} \to \langle l, r, k, f\rangle_{\mathrm{app}}$$

Mode **ke**:

$$\langle \mathrm{ke}_{\mathrm{id}}, \Sigma, f\rangle_{\mathrm{ke}} \not\to \texttt{(* end of computation *)}$$
$$\langle \texttt{KELet}(S, k), \Sigma, f\rangle_{\mathrm{ke}} \to \langle S, \Sigma, k, f\rangle_{\mathrm{sk}}$$
$$\langle \texttt{KEPat}(l_p, l_r, k), \Sigma, f\rangle_{\mathrm{ke}} \to \langle l_p, l_r, \Sigma, k, f\rangle_{\mathrm{patl}}$$
$$\langle \texttt{KEApp}(S, l_a, k), \Sigma, f\rangle_{\mathrm{ke}} \to \langle S, \Sigma, \texttt{KRApp3}(l_a, k), f\rangle_{\mathrm{sk}}$$
$$\langle \texttt{KELdp}(l_{dp}, l_{dr}, k), \Sigma, f\rangle_{\mathrm{ke}} \to \langle l_{dp}, l_{dr}, \Sigma, k, f\rangle_{\mathrm{patldp}}$$

Mode **fk**:

$$\langle \texttt{FEmpty}\rangle_{\mathrm{fk}} \not\to \texttt{(* fail *)}$$
$$\langle \texttt{FSK}(S, \Sigma, k, f)\rangle_{\mathrm{fk}} \to \langle S, \Sigma, k, f\rangle_{\mathrm{sk}}$$
$$\langle \texttt{FList}(l, k, f)\rangle_{\mathrm{fk}} \to \langle l, k, f\rangle_{\mathrm{lst}}$$

Mode **trm**:

$$\langle \texttt{TVar}(\texttt{VLet}(x)), \Sigma, k, f\rangle_{\mathrm{trm}} \to \langle \Sigma, x, k, f\rangle_{\mathrm{lkup}}$$
$$\langle \texttt{TVar}(\texttt{VSpec}(x)), \Sigma, k, f\rangle_{\mathrm{trm}} \to \langle \texttt{SpecDecl}(x), [], k, f\rangle_{\mathrm{trm}}$$
$$\langle \texttt{TVar}(\texttt{VUnspec}(x)), \Sigma, k, f\rangle_{\mathrm{trm}} \to \langle \texttt{UnspecDecl}(x)[], k, f\rangle_{\mathrm{lst}} \qquad \texttt{if Arity}(x) = 0$$
$$\langle \texttt{TVar}(\texttt{VUnspec}(x)), \Sigma, k, f\rangle_{\mathrm{trm}} \to \langle k, \texttt{CVUnspec}(x, m, []), f\rangle_{\mathrm{kr}} \qquad \texttt{if Arity}(x) = m + 1$$
$$\langle \texttt{TConstr}(c, t), \Sigma, k, f\rangle_{\mathrm{trm}} \to \langle t, \Sigma, \texttt{KRConstr}(c, k), f\rangle_{\mathrm{trm}}$$
$$\langle \texttt{TTuple}(l), \Sigma, k, f\rangle_{\mathrm{trm}} \to \langle l, \Sigma, \texttt{KLTuple}(k), f\rangle_{\mathrm{trml}}$$
$$\langle \texttt{TFunc}(p, S), \Sigma, k, f\rangle_{\mathrm{trm}} \to \langle k, \texttt{CVClos}(\Sigma, p, S), f\rangle_{\mathrm{kr}}$$
$$\langle \texttt{TField}(t, d), \Sigma, k, f\rangle_{\mathrm{trm}} \to \langle t, \Sigma, \texttt{KRField}(d, k), f\rangle_{\mathrm{trm}}$$
$$\langle \texttt{TNth}(t, n), \Sigma, k, f\rangle_{\mathrm{trm}} \to \langle t, \Sigma, \texttt{KRNth}(n, k), f\rangle_{\mathrm{trm}}$$
$$\langle \texttt{TRec}(\texttt{Some}(t), l_{dt}), \Sigma, k, f\rangle_{\mathrm{trm}} \to \langle t, \Sigma, \texttt{KRUp}(l_{dt}, \Sigma, k), f\rangle_{\mathrm{trm}}$$
$$\langle \texttt{TRec}(\texttt{None}, l_{dt}), \Sigma, k, f\rangle_{\mathrm{trm}} \to \langle l_{dt}, \Sigma, \texttt{KDRec}(k), f\rangle_{\mathrm{ldt}}$$

Modes **uprec**, **getf**, **getn**, and **gets**:

$$\begin{aligned}
\langle \texttt{CVRec}(l_{dr}), l_{dt}, \Sigma, k, f\rangle_{\text{uprec}} &\to \langle l_{dt}, \Sigma, \texttt{KDUp}(l_{dr}, k), f\rangle_{\text{ldt}} \\
\langle _, l_{dt}, \Sigma, k, f\rangle_{\text{uprec}} &\to \langle f\rangle_{\text{fk}} && \texttt{otherwise} \\
\langle \texttt{CVRec}(l_{dr}), d, k, f\rangle_{\text{getf}} &\to \langle l_{dr}, d, k, f\rangle_{\text{lkup}} \\
\langle _, d, k, f\rangle_{\text{getf}} &\to \langle f\rangle_{\text{fk}} && \texttt{otherwise} \\
\langle \texttt{CVTuple}(l_r), n, k, f\rangle_{\text{getn}} &\to \langle (\texttt{nth}\ l_r\ n), k, f\rangle_{\text{gets}} \\
\langle _, n, k, f\rangle_{\text{getn}} &\to \langle f\rangle_{\text{fk}} && \texttt{otherwise} \\
\langle \texttt{Some}(r), k, f\rangle_{\text{gets}} &\to \langle k, r, f\rangle_{\text{kr}} \\
\langle \texttt{None}, k, f\rangle_{\text{gets}} &\to \langle f\rangle_{\text{fk}}
\end{aligned}$$

Mode **ldt**:

$$\begin{aligned}
\langle [], \Sigma, k, f\rangle_{\text{ldt}} &\to \langle k, [], f\rangle_{\text{kd}} \\
\langle (d, t) :: l_{dt}, \Sigma, k, f\rangle_{\text{ldt}} &\to \langle t, \Sigma, \texttt{KRLdt}(l_{dt}, \Sigma, d, k), f\rangle_{\text{trm}}
\end{aligned}$$

Mode **lkup**:

$$\begin{aligned}
\langle [], x, k, f\rangle_{\text{lkup}} &\to \langle f\rangle_{\text{fk}} \\
\langle (y, r) :: l, x, k, f\rangle_{\text{lkup}} &\to \langle k, r, f\rangle_{\text{kr}} && \texttt{if}\ x = y \\
\langle (y, r) :: l, x, k, f\rangle_{\text{lkup}} &\to \langle l, x, k, f\rangle_{\text{lkup}} && \texttt{if}\ x \neq y
\end{aligned}$$

Mode **trml**:

$$\begin{aligned}
\langle [], \Sigma, k, f\rangle_{\text{trml}} &\to \langle k, [], f\rangle_{\text{kl}} \\
\langle t :: l, \Sigma, k, f\rangle_{\text{trml}} &\to \langle t, \Sigma, \texttt{KRList}(l, \Sigma, k), f\rangle_{\text{trm}}
\end{aligned}$$

Mode **sk**:

$$\begin{aligned}
\langle \texttt{Branching}([]), \Sigma, k, f\rangle_{\text{sk}} &\to \langle f\rangle_{\text{fk}} \\
\langle \texttt{Branching}(S :: l), \Sigma, k, f\rangle_{\text{sk}} &\to \langle S, \Sigma, k, \texttt{FSK}(\texttt{Branching}(l), \Sigma, k, f)\rangle_{\text{sk}} \\
\langle \texttt{LetIn}(p, S_1, S_2), \Sigma, k, f\rangle_{\text{sk}} &\to \langle S_1, \Sigma, \texttt{KRLet}(p, S_2, \Sigma, k), f\rangle_{\text{sk}} \\
\langle \texttt{Return}(t), \Sigma, k, f\rangle_{\text{sk}} &\to \langle t, \Sigma, k, f\rangle_{\text{trm}} \\
\langle \texttt{Apply}(t, l), \Sigma, k, f\rangle_{\text{sk}} &\to \langle t, \Sigma, \texttt{KRApp1}(l, \Sigma, k), f\rangle_{\text{trm}} \\
\langle \texttt{Exists}(p, S), \Sigma, k, f\rangle_{\text{sk}} &\to \langle f\rangle_{\text{fk}}
\end{aligned}$$

Mode **app**:
(note: arguments swapped to pattern-match the first)

$$\langle [], r, k, f\rangle_{\text{app}} \to \langle k, r, f\rangle_{\text{kr}}$$
$$\langle a :: l_a, \texttt{CVUnspec}(x, 0, l), k, f\rangle_{\text{app}} \to \langle \texttt{UnspecDecl}(x)(l +\!\!+ [a]), \texttt{KRApp3}(l_a, k), f\rangle_{\text{lst}}$$
$$\langle a :: l_a, \texttt{CVUnspec}(x, m+1, l), k, f\rangle_{\text{app}} \to \langle l_a, \texttt{CVUnspec}(x, m, l +\!\!+ [a]), k, f\rangle_{\text{app}}$$
$$\langle a :: l_a, \texttt{CVClos}(\Sigma, p, S), k, f\rangle_{\text{app}} \to \langle p, a, \Sigma, \texttt{KEApp}(S, l_a, k), f\rangle_{\text{pat}}$$
$$\langle a :: l_a, \_, k, f\rangle_{\text{app}} \to \langle f\rangle_{\text{fk}} \qquad \texttt{otherwise}$$

Mode **lst**:

$$\langle [], k, f\rangle_{\text{lst}} \to \langle f\rangle_{\text{fk}}$$
$$\langle r :: l, k, f\rangle_{\text{lst}} \to \langle k, r, \texttt{FList}(l, k, f)\rangle_{\text{kr}}$$

Mode **pat**:

$$\langle \texttt{PWild}, r, \Sigma, k, f\rangle_{\text{pat}} \to \langle k, \Sigma, f\rangle_{\text{ke}}$$
$$\langle \texttt{PVar}(x), r, \Sigma, k, f\rangle_{\text{pat}} \to \langle k, (x, r) :: \Sigma, f\rangle_{\text{ke}}$$
$$\langle \texttt{PConstr}(c, p), \texttt{CVConstr}(c_2, r), \Sigma, k, f\rangle_{\text{pat}} \to \langle p, r, \Sigma, k, f\rangle_{\text{pat}} \qquad \texttt{if } c = c_2$$
$$\langle \texttt{PTuple}(l_p), \texttt{CVTuple}(l_r), \Sigma, k, f\rangle_{\text{pat}} \to \langle l_p, l_r, \Sigma, k, f\rangle_{\text{patl}}$$
$$\langle \texttt{PRec}(l_{dp}), \texttt{CVRec}(l_{dr}), \Sigma, k, f\rangle_{\text{pat}} \to \langle l_{dp}, l_{dr}, \Sigma, k, f\rangle_{\text{patldp}}$$
$$\langle \_, \_, \Sigma, k, f\rangle_{\text{pat}} \to \langle f\rangle_{\text{fk}} \qquad \texttt{otherwise}$$

Mode **patldp**:

$$\langle [], l_{dr}, \Sigma, k, f\rangle_{\text{patldp}} \to \langle k, \Sigma, f\rangle_{\text{ke}}$$
$$\langle (d, p) :: l_{dp}, l_{dr}, \Sigma, k, f\rangle_{\text{patldp}} \to \langle l_{dr}, d, \texttt{KRLdt}(p, \Sigma, l_{dp}, l_{dr}, k), f\rangle_{\text{lkup}}$$

Mode **patl**:

$$\langle [], [], \Sigma, k, f\rangle_{\text{patl}} \to \langle k, \Sigma, f\rangle_{\text{ke}}$$
$$\langle p :: l_p, r :: l_r, \Sigma, k, f\rangle_{\text{patl}} \to \langle p, r, \Sigma, \texttt{KEPat}(l_p, l_r, k), f\rangle_{\text{pat}}$$
$$\langle \_, \_, \Sigma, k, f\rangle_{\text{patl}} \to \langle f\rangle_{\text{fk}} \qquad \texttt{otherwise}$$

UNIVERSITÉ DE RENNES 1

Titre : Transformations de Sémantiques Squelettiques

Mots-clés : Sémantiques Squelettiques, Sémantiques Opérationnelles, Grand-Pas, Petit-Pas, Machines Abstraites, Interprétation Certifiée

Résumé : Les sémantiques squelettiques sont un cadre logique pour décrire les sémantiques opérationnelles.

Tout d'abord, nous présentons une transformation automatique d'une sémantique squelettique écrite en style grand-pas vers une sémantique équivalente en style petit-pas. Cette transformation est implémentée dans l'outil Necro, ce qui nous permet de générer automatiquement un interpréteur OCaml pour la sémantique petit-pas ainsi qu'une formalisation Coq des deux sémantiques. Nous certifions la transformation de deux manières : nous donnons une preuve papier du cœur de la transformation, et nous générons des scripts de preuve Coq spécialisés durant la transformation.

Nous proposons également une méthode automatique pour générer un interpréteur OCaml certifié pour n'importe quel langage défini en sémantiques squelettiques. Pour cela, nous présentons deux nouvelles interprétations des sémantiques squelettiques, sous la forme de machines abstraites déterministe et non-déterministe. Ces machines sont obtenues à partir de l'interprétation grand-pas principale en utilisant la correspondence fonctionnelle, une méthode connue pour transformer un évaluateur en machine abstraite. Ces nouvelles interprétations sont formalisées en Coq, et nous vérifions leur correction. Enfin, nous utilisons le système d'extraction de Coq vers OCaml pour obtenir un interpréteur certifié.

Title: Skeletal Semantics Transformations

Keywords: Skeletal Semantics, Operational Semantics, Big-Step, Small-Step, Abstract Machines, Certified Interpretation

Abstract: Skeletal semantics is a framework to describe the operational semantics of programming languages.

We first 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 transformation correct in two ways: we provide a paper proof of the core of the translation, and we generate Coq certification scripts alongside the transformation.

We also propose an automatic generation of a certified OCaml interpreter for any language written in skeletal semantics. To this end, we introduce two new interpretations, i.e., formal meanings, of skeletal semantics, in the form of non-deterministic and deterministic abstract machines. These machines are derived from the usual big-step interpretation of skeletal semantics using functional correspondence, a standard transformation from big-step evaluators to abstract machines. All these interpretations are formalized in the Coq proof assistant and we certify their soundness. We finally use the extraction from Coq to OCaml to obtain the certified interpreter.