Extensible Transition System Semantics

Casper Bach Poulsen

Submitted to Swansea University in fulfilment
of the requirements for the Degree of Doctor of Philosophy

Swansea University
Prifysgol Abertawe

Department of Computer Science
Swansea University

2015

Declaration

This work has not been previously accepted in substance for any degree and is not being concurrently submitted in candidature for any degree.

Signed	....	(candidate)
Date	....	

Statement 1

This thesis is the result of my own investigations, except where otherwise stated. Other sources are acknowledged by footnotes giving explicit references. A bibliography is appended.

Signed	....	(candidate)
Date	....	

Statement 2

I hereby give my consent for my thesis, if accepted, to be available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

Signed	....	(candidate)
Date	....	

Abstract

Structural operational semantics (SOS) come in two main styles: big-step and small-step. Each style has its merits and drawbacks, and it is sometimes useful to maintain specifications in both styles. But it is both tedious and error-prone to maintain multiple specifications of the same language. Additionally, big-step SOS has poor support for language evolution, requires reformulation or introduction of new rules for existing constructs as a language is extended, and is sometimes regarded as inferior for type soundness proofs. This thesis addresses pragmatic shortcomings with giving and relating extensible small-step and big-step specifications, and with big-step type soundness proofs.

The thesis makes a number of contributions. First, we present *Extensible SOS* (XSOS), a simple but novel extension of Mosses' *Modular SOS* that supports concise and extensible specification of language features for both big-step and small-step semantics. Second, we internalise the well-known *refocusing* transformation in XSOS to provide a systematic transformation between extensible small-step and big-step specifications. Third, we consider types as abstract interpretations as a novel approach to big-step type soundness. Finally, we propose a novel type system for Hindley-Milner-Damas polymorphic type inference for a language with ML style references.

Acknowledgements

I sincerely thank Peter Mosses for his support, encouragement, and all that he has taught me during my years in Swansea. I would also like to thank Ulrich Berger for being my second supervisor, and my examiners, Graham Hutton and Anton Setzer, for interesting discussions during my viva, and for their valuable suggestions for improving this thesis. I am grateful to Fredrik Nordvall Forsberg, Neil Sculthorpe, Peter Mosses, Paolo Torrini, and Ulrich Berger for patient proofreading and feedback on earlier versions of this thesis. My thesis and I have also benefited from discussions with Martin Churchill, Ferdinand Vesely, Olivier Danvy, Jacob Johannsen, Jan Midtgaard, Andrew Tolmach, and Eelco Visser; thank you. I further thank Olivier Danvy for his inspiring lecture courses at Aarhus University, as well as his support in connection with applying for a Ph.D. in Swansea; and Eelco Visser for employing me after my Ph.D., and for his topical words about deadlines [Vis97, Acknowledgements, final paragraph].

Thanks are also due to all the people whose moral support and company I have enjoyed throughout my Ph.D. Thank you for your friendship, Phil and Emma James, Fredrik Nordvall Forsberg, Liam O'Reilly, Ferdinand Vesely and Olga Vesela, Dean Thomas, and Shannon Morrison. Thank you, Erling Midtgaard Hanssen and Dorete Bøving Larsen, for opening up your hearts and home whenever Margit and I visit home. Thank you to my family for always being there for me. A special thanks goes to my beloved Margit: thank you for agreeing to join me in moving to Swansea for our Welsh adventure, and for always being there for me, not least on the (sometimes strenuous) journey of my Ph.D.

This work was supported by an EPSRC grant (EP/I032495/1) to Swansea University in connection with the PLanCompS project (`www.plancomps.org`).

Table of Contents

List of Figures

List of Tables

1 Introduction

Contents

Formal semantics allow concise and precise specification of the meaning of programming languages, and provide valuable tools for reasoning about them. But harnessing the potential of formal semantics is a balancing act: a semantics must be abstract yet comprehensible; it should facilitate both formal reasoning and experimentation; and it must be practically maintainable as a language evolves. To date, there is no one framework which addresses all concerns in general. In the words of Cousot and Cousot [CC92]:

> The quest for a unique general-purpose semantics for programming languages has failed. A better approach is to establish correspondences between various semantics at different levels of abstraction.

While this observation was made more than two decades ago, the words still ring true today. Yet, we argue, there is a lack of formalisms that support giving and relating semantics at different levels of abstraction for evolving languages.

This thesis investigates a formalism for *operational semantics* that supports giving *extensible* programming language specifications at different levels of abstraction. The different levels of abstraction studied in this thesis are: *small-step* semantics that expose the intermediate states of an underlying transition system; *big-step* semantics that abstract from the intermediate states of the underlying transition system and relate programs directly to their outcomes; and *type systems*, which classify the outcomes of computations. Chapter 2 recalls state of the art techniques for giving and relating such specifications. In this chapter, we first motivate why we believe the framework and techniques that this thesis studies are of value, before presenting the thesis statement and contributions in Section 1.3.

1.1 Motivation: promises and problems with formal specification

Software is revolutionising our daily lives and society as a whole by providing the means for making machines work for us. Programming languages are the interface between human and machine that allows us to dictate our orders to the machine, by writing and executing programs. But how do we ensure that programs are correct? And, equally importantly, how do we ensure the correctness of the interpreters or compilers that make our programs executable on machines?

To answer such questions, we must specify what constitutes 'correct' behaviour. In some cases it is possible to exhaustively enumerate what the behaviour of a program should be for different inputs or stimuli. But for many interesting programs and applications such exhaustive enumeration is often either infeasible or downright impossible. The promise of formal semantics is that it provides a mathematical model that enables precise semantic specification and supports formal reasoning about programs and programming languages. But giving and maintaining formal specifications is challenging.

A witness to the challenges associated with formal specification is its lack of adoption in specification of mainstream programming languages: semantics of most mainstream programming languages (such as Java, C#, Haskell, F#, etc.) are specified informally, usually by means of informal prose descriptions of the semantics of constructs [GJS^{+}13, ECM06, Mar10, Sym12]. The problem with informal specification is that it is often ambiguous and incomplete. Informal specifications also make it hard to harness the potential of formal specifications, such as formal reasoning, or deriving executable interpreters or compilers from the specification.

The lack of adoption of formal specification for programming language semantics is in contrast to the state of affairs when it comes to syntax specifications, which are commonly given using formal BNF-style grammars. Programming language designers are not averse to this form of formal specification. Why, then, is it so rare to see semantics specified formally for mainstream programming languages?

The Definition of Standard ML [MTHM97] is one of the few mainstream programming languages whose semantics is specified formally. While The Definition of Standard ML holds an honourary place on the bookshelf of many programming language researchers, it has been subject to much constructive criticism and has motivated and inspired subsequent research, particularly in operational semantics. For example, Harper [HS00b], one of the authors of The Definition of Standard ML, criticises the use of the *big-step* style of semantic specification. This style has pragmatic challenges when it comes to proving type soundness [WF94, HS00b, LG09]. Another problem is that traditional approaches to extending the semantics with certain features, such as abrupt termination, divergence, control features, or concurrency, introduces many extra rules for all constructs in the language [MTHM97, Cha13, HS00b, HDM93, Uus13].

There are several alternatives to the big-step style. In fact, there is a proliferation of different approaches to formal semantics, ranging from more abstract and mathematical approaches, such as denotational semantics [SS71, Mos90, Sch86], categorical semantics [Mog91, TP97], game semantics [AJM00], or axiomatic semantics [Hoa69], to more operational approaches, such as Structural Operational Semantics (SOS) [Plo04], natural semantics [Kah87], or reduction semantics [Fel87]. This proliferation is in part motivated by the pragmatics of working with different frameworks in different settings, as Cousot and Cousot [CC92] observed in their quote in the introduction of this thesis. Winskel [Win93, Preface] echoes this observation; talking about denotational vs. axiomatic vs. operational semantics, he writes:

> It would be wrong to view these styles as in opposition to each other. They each have their uses.

Although the observations of the Cousots and Winskel were made more than two decades ago, there is little reason to believe that the state of affairs are different today. Cutting edge research in formal semantics sees a prolific use of different frameworks for different purposes. For example, CompCert [Ler06, LG09], a C-compiler which produces machine code that provably corresponds with the semantics of the C-program from which it is compiled, uses many different semantic styles, suitable for different purposes.

Specifications in each style are commonly specified and proven correct in relation to each other using modest means of automation. Giving and relating these is hard work. Hudak et al. [HHPJW07] cites this as one of the main reasons why Haskell was never given a formal specification:

> Nevertheless, we always found it a little hard to admit that a language as principled as Haskell aspires to be has no formal definition. But that is the fact of the matter, and it is not without its advantages. In particular, the absence of a formal language definition does allow the language to *evolve* more easily, because the costs of producing fully formal specifications of any proposed change are heavy, and by themselves discourage changes.

This thesis studies how to specify, relate, and work with operational semantics that support *language evolution* at different levels of abstraction.

1.2 Background: operational semantics

Operational semantics is an approach to formal specification that strikes a balance between power and simplicity. According to Winskel [Win93, Preface], "operational semantics describes the meaning of a programming language by specifying how it executes on an abstract machine." This makes them both accessible, amenable to formal reasoning, and amenable to deriving interpreters and compilers that faithfully implement the intended semantics. Operational semantics come in different styles and at different levels of abstraction. Each style has its merits and drawbacks.

1.2.1 Traditional styles of operational semantics

Operational semantics come in two styles: small-step and big-step. A small-step relation defines transitions between intermediate configurations in a transition system. In contrast, big-step relations abstract from intermediate transition steps by relating configurations directly to final outcomes.

Another important distinction between variants of operational semantics is how program contexts are represented. Operational semantics dates back to (at least) Landin's work on the Mechanical Evaluation of Expressions [Lan64]. Landin's SECD-machine consists of a transition function that operates on stacks. Years later, Plotkin in his famous Aarhus lecture notes remarked [Plo81, p. 18] on abstract machine semantics:

> They all have a tendency to pull the syntax to pieces or at any rate to wander around the syntax creating various complex symbolic structures which do not seem particularly forced by the demands of the language itself. Finally, they do not in general have any great claim to being syntax-directed in the sense of defining the semantics of compound phrases in terms of the semantics of their components, although the definition of the transition relation does fall into natural cases following the various syntactical possibilities.

Here, the "complex symbolic structures" that Plotkin refers to are the explicit stacks on which abstract machine semantics operate. Plotkin proposed Structural Operational Semantics (SOS) as an alternative. SOS transition relations are defined using conditional rules that rely on the structure of a composite term.

A big-step counterpart to SOS was proposed by Kahn [Kah87], who dubbed this variant *natural semantics*, due to its resemblance with natural deduction. Like SOS, rules in natural semantics are typically defined using conditional rules that rely on the structure of a composite term, and not on explicit representations of program context. Thus, *big-step SOS* is commonly used to denote natural semantics, while *small-step SOS* commonly refers to transition relation specifications defined using SOS.

Following Plotkin [Plo81], extending a language with features such as imperative state or throwing exceptions (*abrupt termination*) involves modifying and introducing new SOS rules for existing constructs in a language. This involves tedious and error-prone modification of rules and proofs. The problem is even more pronounced in natural semantics: for example, for each construct in The Definition of Standard ML [MTHM97] with n evaluable sub-phrases, there are n implicit rules for propagating an exception if it occurs.[1] This is known as the *duplication problem* in big-step semantics [Cha13].

Felleisen introduced *reduction semantics* as "a symbolic reasoning system for the core of expressive languages" [Fel87, p. 3]. Like abstract machines, reduction semantics operate with an explicit representation of program context, and reduces some of

[1]This is the so-called "exception convention" [MTHM97, p. 47].

the “wandering around the syntax”, as Plotkin calls it. Using reduction semantics, expressing features like imperative state, abrupt termination, and control constructs, such as Landin’s J operator [Lan65] or Scheme’s call/cc [SS75], does not involve changing or introducing new rules for existing constructs although it may give rise to changing the notion of substitution or the grammar of contexts [WF94].

Reduction semantics generally has good support for language evolution, in particular in connection with abrupt termination and control constructs. Recent developments show that the support for language evolution in SOS and natural semantics can be improved, too.

1.2.2 Recent developments

In order to add imperative stores to the SOS example language considered by Plotkin [Plo81], all existing rules are reformulated to mention and propagate the store. Similarly, when adding the possibility of throwing exceptions to a language, Plotkin introduces new rules for existing constructs. This provides poor support for language evolution, since such extensions require new rules to be introduced for every existing construct in the language.

Mosses [Mos04] proposed Modular SOS (MSOS) as a variant of SOS where such reformulation is not necessary. The central idea is that auxiliary entities are encoded in labels on transitions, and that propagating effects between consecutive transitions is handled by *composing* labels. The semantics of label composition is given by a category that can be extended with new components in such a way that these are implicitly propagated without mentioning them explicitly in rules that do not explicitly rely on them. This permits specifying abrupt termination and even control constructs [STM16] modularly, without any explicit syntactic representation of control context. In small-step MSOS, specifying and extending languages with these features does not give rise to modifying or introducing new rules for any existing constructs.

The technique that is used for propagating abrupt termination and control in a modular way does not straightforwardly scale to big-step semantics, however. Mosses remarks [Mos04, p. 218]:

> It seems unlikely that an analogous technique could be provided for use in big-step MSOS. Thus the small-step style has a distinct advantage for the specification of constructs which might involve abrupt termination.

Recently, Charguéraud [Cha13] proposed the novel *pretty-big-step* style of operational semantics. The style is big-step in that it relates configurations directly to final outcomes, but it uses less abstract rules than big-step semantics. The pretty-big-step style reduces the duplication problem with semantics involving abrupt termination or divergence, but still requires the introduction of boilerplate rules in order to propagate abrupt termination or divergence.

Danvy et al. [DN04, BD09, DJZ11, DM08, Dan08] uses transformations between functional representations of semantics to give and relate reduction semantics, structural operational semantics, and denotational (big-step) evaluators. This allows mechanical derivation of semantics at different levels of abstraction. Derived artifacts involving abrupt termination [Dan08] typically rely on either explicit program contexts, or on explicit propagation of exceptions that essentially corresponds to natural semantics suffering from the duplication problem. In functional programming languages, such duplication is expressible more concisely than in SOS by using pattern matching to propagate exceptions. However, it is not always obvious how to implement big-step functional evaluators in proof assistants such as Coq [BC04] or Agda [BDN09], where functions must satisfy strict syntactic criteria in order to guarantee that they can be reasoned about without compromising the consistency of the underlying logic.

1.2.3 Proof engineering and type soundness

An important application of operational semantics in programming language engineering is formal reasoning. A particularly active area of research that uses operational semantics for formal reasoning is type systems. According to Pierce [Pie02]:

> A type system is a tractable method for proving the absence of certain program behaviours by classifying phrases according to the kinds of values they compute.

In practice, operational semantics is commonly used to specify both dynamic semantics and type systems.[2] Proving type soundness is a matter of relating the set of programs that a type system accepts to a set of programs for which evaluation does not go “wrong”, following Milner’s famous slogan that “well-typed programs do not go wrong” [Mil78].

Either big-step or small-step operational semantics can be used for proving type soundness, but the small-step style is often favoured. In their influential paper on the Syntactic Approach to Type Soundness, Felleisen and Wright [WF94] recall big-step approaches to type soundness from the literature and remark:

> A seemingly minor extension to a language may require a complete restructuring of its denotational or structural operational semantics, and may therefore require a completely new approach to re-establish soundness.

They propose reduction semantics as a means of compartmentalising the type soundness proof into progress and preservation (or *subject reduction* [CF58]) lemmas. This compartmentalisation gives type soundness proofs that are more robust under language extensions. The progress/preservation style of proving type soundness works equally well for SOS.[3]

[2]Type system specifications can be thought of as instances of operational semantics (usually natural semantics), although operational semantics, in fact, borrows many of its ideas from early pioneering work in mathematical logic and type theory [Chu40, How80].

[3]For example, Pierce’s popular text-book on Types and Programming Languages uses SOS for dynamic semantics [Pie02].

Felleisen and Wright's observation that big-step semantics does not scale for type soundness proofs is echoed by Harper, one of the co-authors of The Definition of Standard ML [MTHM97] which was originally given in a purely structural big-step style, where by purely structural, we mean semantics that do not rely on an explicit syntactic representation of program context. Harper and Stone remark [HS00b, p. 350]:

> The [small-step] presentation avoids the need for implicit evaluation rules for handling exceptions, and supports a natural interpretation of type soundness that does not rely on artificial "wrong" transitions.

Here, the artificial "wrong" transitions that Harper refers to, is the traditional way of proving big-step type soundness, which involves adding rules for all cases where a big-step semantics can get stuck.

Another increasingly important aspect of using operational semantics for proof engineering is their amenability to mechanisation in proof assistants. Recent years have seen a growing interest in using proof assistants, such as Coq [BC04], Agda [BDN09], or Isabelle [NK14], for reasoning about operational semantics. The attraction of proof assistants is that they detect and reject proofs with typos, inconsistencies, or errors, and they provide support for automation. The rigorous nature of proof assistants make them particularly prone to tedious reformulation of semantics and re-proving propositions if the semantics of existing constructs is modified when specifications are extended.

1.2.4 Lack of support for language evolution at different levels of abstraction

Specifications using small-step and big-step SOS provide valuable tools for giving and reasoning about programming language semantics. Unfortunately, they suffer from a number of pragmatic issues:

- SOS rules are reformulated if new auxiliary entities are added to a language, and new rules for existing constructs are introduced when new sources of abrupt termination is introduced. This exhibits poor support for language evolution.

- Small-step and big-step semantics for the same language are often specified independently and proven equivalent by means of ad hoc proofs of correspondence [Nip06, LG09, NK14, PCG$^+$13, NH09]. Such proofs are routine, yet comprehensive, and are tedious and error-prone to maintain as a language evolves.

- Dealing with abrupt termination in traditional big-step semantics gives rise to annoying duplication in rules. While such rules can be hidden in specifications (such as the "exception convention" in Standard ML [MTHM97, p. 47]), such rules must be taken into account in induction principles, and thus proofs, which become polluted by annoying duplication.

- Big-step type soundness is traditionally proven by introducing so-called "wrong" rules. These are often added manually, which makes such proofs brittle since failure to add such "wrong" rules may compromise type soundness results.

These pragmatic issues make it expensive to give and maintain specifications of programming languages and type systems, and inhibits harnessing the potential of formal semantics for evolving languages.

Recent developments go some way towards resolving these issues: MSOS [Mos04] provides good support for language evolution in the small-step but not big-step style; pretty-big-step [Cha13] minimises the duplication problem but introduces additional syntax and relies on abort rules for all constructs to propagate abrupt termination and/or divergence. This clutters induction principles, and requires extra typing rules for the additional syntax [Cha13, Section 3.3]. Both Charguéraud and Leroy and Grall have proposals for dealing with type soundness without explicit wrong rules that we discuss later in Section 2.9.3. Danvy and Nielsen's refocusing transformation [DN04] provides a means of transforming small-step evaluation strategies into big-step evaluation strategies, for "refocus-ready" functional programs implementing reduction semantics [Dan08, Dan04, DJZ11, Zer13].

In this thesis we propose *extensible transition system semantics* as a framework that provides pragmatic support for the pragmatic issues identified in this section, by combining and extending many of the novel ideas from existing lines of research.

1.2.5 Scope: programming languages and their (component-based) semantics

The PLanCompS[4] project provides pragmatic approach to specifying programming language semantics. The approach is summarised by Mosses et al. [CMST15, p. 135] as follows:

> Its crucial novel feature is the introduction of an *open-ended* collection of so-called *fundamental constructs*, or *funcons*. Many of the funcons correspond closely to simplified language constructs. But in contrast to language constructs, each funcon has a fixed interpretation, which we specify, *once and for all*, using [small-step] MSOS.

In this thesis we consider techniques for making the semantics of funcons more widely and pragmatically applicable. Specifically, we investigate how to automatically derive modular big-step specifications that soundly abstract their modular small-step counterparts with fixed order of evaluation. Although some funcons are specified to allow interleaving order of evaluation, we restrict our attention to left-to-right order of evaluation, so as to more straightforwardly specify and reason about funcons using big-step semantics.

The original motivation for deriving big-step rules for funcons was to provide a basis for deriving more efficient prototype interpreters by a naive translation of MSOS

[4]`http://www.plancomps.org/`

rules into Horn-clauses [BPM14c]. As argued in this introduction, there is also use in maintaining both small-step and big-step semantics for specification and verification purposes. The techniques for addressing the pragmatic issues in Section 1.2.4 that we present in this thesis scale to (M)SOS specifications for funcons as well as traditional programming language semantics, and gives support for relating small-step and big-step semantics in a way that enables both deriving more efficient prototype interpreters, and provides a basis for formal verification.

1.3 Thesis

This thesis studies how to specify, relate, and work with operational semantics that support language evolution at different levels of abstraction, focusing particularly on purely structural operational semantics, i.e., semantics without any explicit syntactic representation of program context. We propose a simple but novel variant of Mosses' generalised transition systems [Mos04] that we call *extensible transition systems*. Our thesis is:

> *Extensible transition system semantics provides a basis for giving and relating extensible and purely structural semantics at different levels of abstraction.*

The different levels of abstraction that this thesis considers are small-step and big-step semantics, as well as type systems. The thesis investigates and contributes the following:

- We present (Chapter 3) Extensible SOS (XSOS) as a simple but novel extension of Mosses' Modular SOS. This provides a simple basis for giving concise and extensible specifications of various language features, including abrupt termination, in a way that scales to both small-step and big-step specifications.

- We provide an internalisation of Danvy and Nielsen's *refocusing* technique in the framework of XSOS, which gives a direct means of relating small-step and (pretty-)big-step XSOS specifications (Chapter 4) as well as their coinductive counterparts (Chapter 5);

- A common criticism of big-step SOS is that it is error-prone to work with in connection with type soundness proofs: type soundness typically relies on a manual instrumentation of the semantics by adding all the cases where the semantics can get stuck. Failure to add all cases potentially allows us to prove type soundness of unsound type systems. We recall and investigate a type soundness proof method based on Cousot's work on Types as Abstract Interpretations [Cou97]. This avoids adding artificial "wrong" transitions in order to prove type soundness using big-step semantics (Chapter 6) and provides a foundation for proving simultaneous type soundness and strong normalisation;

- We propose that the types as abstract interpretations approach is applicable to extensible specifications of dynamic semantics and type systems. We investigate the extent to which this allows us to alleviate the drawback of big-step type soundness proofs that seemingly minor extensions of big-step semantics may require restructuring both semantics and proof (Chapter 7), and discuss the pros and cons of the types as abstract interpretations approach.

- To further test the extensibility and pragmatic properties of the type soundness proof method, we propose a novel *type and effect system* for Hindley/Milner polymorphic type inference in the presence of imperative references (Chapter 8). The system records a store at the type level to inhibit polymorphic generalisation over types in the store. We investigate how the types as abstract interpretations proof method applies to prove the system correct.

These contributions address the pragmatic issues outlined in Section 1.2.4, and improve the state-of-the-art in supporting language evolution in operational semantics and in harnessing the potential of formal specification.

1.4 Relationship with previous publications

Several of the ideas presented in this thesis have previously appeared in the following papers, all of which have the author of this thesis as the main author:

1. Casper Bach Poulsen and Peter D. Mosses [BPM14c]. Generating specialized interpreters for Modular Structural Operational Semantics. In *LOPSTR'13*, volume 8901 of *LNCS*, pages 220–236. Springer, 2014.

2. Casper Bach Poulsen and Peter D. Mosses [BPM14a]. Deriving pretty-big-step semantics from small-step semantics. In *ESOP'14*, volume 8410 of *LNCS*, pages 270–289. Springer, 2014.

3. Casper Bach Poulsen, Peter D. Mosses, and Paolo Torrini [BPMT15]. Imperative polymorphism by store-based types as abstract interpretations. In *PEPM'15*, pages 3–8. ACM, 2015.

4. Casper Bach Poulsen and Peter D. Mosses [BPM16]. Flag-based big-step semantics. To appear in *The Journal of Logical and Algebraic Methods in Programming*. `http://plancomps.org/flag-based-big-step`, 2016.

The first of these publications [BPM14c], co-authored with Mosses, presents the refocusing transformation for MSOS rules, and a study of comparing the performance of interpreters based on, respectively, small-step and refocused small-step MSOS rules implemented in Prolog. The study relies on an internalisation of Danvy and Nielsen's refocusing technique in MSOS, which our ESOP'14 paper [BPM14a] follows up on. The ideas in our LOPSTR'13 paper [BPM14c] are mainly those of the author of this thesis.

In our subsequent ESOP'14 paper [BPM14a], also co-authored with Mosses, we observe the correspondence between what we call 'refocused' rules in LOPSTR, and the pretty-big-step style introduced by Charguéraud [Cha13] at ESOP'13. We also present an idea for representing abrupt termination in big-step semantics using a flag-based approach; an adaptation of a similar technique attributed to Klin by Mosses [Mos04]. Chapter 3 of this thesis also describes this idea, and how it is incorporated in *extensible transition systems*, which alleviate the need for the somewhat ad hoc syntactic conventions for writing labels that is used in our ESOP'14 paper [BPM14a, Section 3.2]. The notion of extensible transition system semantics is a novelty of this thesis, and was not a part of our ESOP paper. The ESOP paper provides proofs of correctness for only the languages it contains. Chapters 4 and 5 in this thesis generalise the correctness proofs of the ESOP paper by proving refocusing correct for any rules that satisfy a lax format of *Extensible SOS* rules.

Our PEPM'15 paper [BPMT15], co-authored with Mosses and Torrini, presents an adaptation of types as abstract interpretations to SOS. Mosses and the author wrote Section 2 together, which describes work that is superseded by [BPM16]. The author collaborated on adapting the types as abstract interpretations approach to SOS with Torrini who came up with the idea that lead to the strengthened induction hypothesis for the type soundness proof technique described in Section 3 of [BPMT15], which he also co-authored. Chapter 6 of this thesis recalls this technique, and observes some further implications: the strong normalisation corollary and how to implement it in the Coq proof assistant. Chapter 7 adapts the technique to XSOS, and applies it to a language with let-polymorphism and ML-style references. Chapter 8 of this thesis provides an adapted version of the store-based type system from [BPMT15].

Chapter 5 of this thesis uses ideas from [BPM16], which is an extended version of an abstract [BPM14b] presented at NWPT'14. The idea in [BPM16, BPM14b] is to use a flag for distinguishing divergence in the coinductive interpretation of big-step derivation trees. In contrast to the paper [BPM16], which considers how to apply the technique to traditional big-step rules, Chapter 5 of this thesis focuses on how to apply the technique to pretty-big-step rules.

2 Operational Semantics in Theory and Practice

Contents

We give an overview of the state of the art approaches to purely structural operational semantics, and their (lack of) support for language evolution and reasoning about programs that diverge. We also recall traditional approaches to proving type soundness of big-step semantics, and suggest that interpreting types as abstract interpretations holds potential for simplifying proofs by avoiding what Harper and Stone refer to as artificial "wrong" transitions [HS00b].

In addition to presenting state of the art approaches, this section also introduces notation and conventions that will be used throughout the thesis.

2.1 Prerequisites and conventions

The reader is assumed to have some familiarity with set theory, inductive definitions, and inductive proofs. For a gentle introduction to these concepts see, e.g., textbooks by Pierce [Pie02, Chapters 1-3], Sangiorgi [San11], or Winskel [Win93]. We also assume some basic familiarity with coinductive definitions and proofs, where we mainly use *guarded* coinduction. For a gentle introduction to the concepts of coinduction see, e.g.,

	Symbol	**Interpretation**
Logical connectives:	$\wedge, \vee$	conjunction, disjunction
	$\Longrightarrow, \Longleftrightarrow$	logical implication, if-and-only-if
	$\neg, \forall, \exists$	not, for all, there exists
Set-based operations:	$\cap, \cup, \setminus$	intersection, union, difference
	$\subseteq, \subset, \in$	subset, proper subset, membership
	$\times, ^*$	Cartesian product, Kleene star

Table 2.1: Summary of notation

textbooks by Pierce [Pie02, Chapter 21] or Sangiorgi [San11]. For a gentle introduction to proofs by guarded coinduction see, e.g., Leroy and Grall's paper on Coinductive Big-Step Operational Semantics [LG09, p. 285-287]. Table 2.1 summarises the notation we will be using.

2.2 Structural operational semantics

In Plotkin's famous Aarhus lecture notes [Plo81, Plo04] that introduced SOS he writes:

> It is the purpose of these notes to develop a simple and direct method for specifying the semantics of programming languages. Very little is required in the way of mathematical background; all that will be involved is "symbol-pushing" of one kind or another of the sort which will already be familiar to readers with experience of either the non-numerical aspects of programming languages or else formal deductive systems of the kind employed in mathematical logic.

The means with which Plotkin achieves this end, is by formalising semantics as a transition system, where the transition relation is specified using *structural* inference rules, i.e., rules where the meaning of composite program phrases is given by the meaning of its structural sub-components.

This section recalls the theory behind SOS, and illustrates its usage on a small example language that we use throughout this thesis. We specify the language by gradually adding more features, illustrating some of the strengths (simple rules that support interleaving) and weaknesses (lack of modularity) of the approach.

To minimise the syntactic clutter in formulas, we rely on the convention that any variables that occur free in a statement are implicitly quantified at the top-level. For example, Definition 2.1 uses this convention to define what constitutes a deterministic transition relation for some set of configurations $\gamma \in \Gamma$ and labels $l \in L$.[1]

[1]In process algebras (e.g., [San11, p. 16]), one often sees deterministic relations defined as relations which, if they emit the same output have the same resulting configuration; i.e., for any $\gamma, \gamma', \gamma'', l$, it holds

Definition 2.1 (Deterministic relation) A relation $\rightsquigarrow \subseteq \Gamma \times L \times \Gamma$ is deterministic iff:

$$\begin{array}{l} \gamma \overset{l}{\rightsquigarrow} \gamma' \implies \\ \gamma \overset{l'}{\rightsquigarrow} \gamma'' \implies \\ l = l' \wedge \gamma' = \gamma'' \end{array}$$

The formula in Definition 2.1 is supposed to read:

> for all configurations $\gamma, \gamma', \gamma''$ and labels l, l', if $\gamma \overset{l}{\rightsquigarrow} \gamma'$
> and if $\gamma \overset{l'}{\rightsquigarrow} \gamma''$
> then $l = l'$ and $\gamma' = \gamma''$

In this formula, we call $\gamma \xrightarrow{l} \gamma'$ and $\gamma \xrightarrow{l'} \gamma''$ *premises* or *hypotheses*, and say that $l = l' \wedge \gamma' = \gamma''$ is the *conclusion* or the *goal* of the formula.

2.2.1 Transition system semantics

Each step in a transition system iterates a program and its state (which we call a *configuration*) towards a final configuration from which no further transitions can be made. Each intermediate step may emit some observable output, which is recorded in the label on the transition. Definition 2.2 is due to Plotkin and defines transition systems formally.

Definition 2.2 (Labelled terminal transition system) A labelled terminal transition system (LTTS) is a quadruple $\langle \Gamma, L, \rightarrow, T \rangle$ consisting of a set Γ of configurations γ, a set L of labels l, a ternary relation $\xrightarrow{l} \subseteq \Gamma \times L \times \Gamma$ of labelled transitions ($\langle \gamma, l, \gamma \rangle \in \rightarrow$ is written $\gamma \xrightarrow{l} \gamma$), and a set $T \subseteq \Gamma$ of terminal configurations, such that $\gamma \xrightarrow{l} \gamma'$ implies $\gamma \notin T$.

A computation in an LTTS (from γ_0) is a finite or infinite sequence of successive transitions $\gamma_i \xrightarrow{l_i} \gamma_{i+1}$ written $\gamma_0 \xrightarrow{l_1} \gamma_1 \xrightarrow{l_2} \ldots$, such that if the sequence terminates with γ_n we have $\gamma_n \in T$.

In SOS, the set of possible transitions are expressed using logical inference rules. Inference rules have the form:

$$\frac{\text{premises}}{\text{conclusion}} \qquad \text{(Rule name)}$$

For a transition $\gamma \xrightarrow{l} \gamma'$, we say that the configuration γ is the *source* of the transition, and γ' is its *target*. Configurations typically consist of programs and auxiliary entities that record the state of the program being executed.

that $\gamma \overset{l}{\rightsquigarrow} \gamma' \implies \gamma \overset{l'}{\rightsquigarrow} \gamma'' \implies l = l' \implies \gamma' = \gamma''$. We opt for a slightly different definition of determinism here, since it is a more natural property to consider if labels have more structure (e.g., in MSOS [Mos04] they may contain auxiliary entities such as stores).

A transition system makes a transition $\gamma \xrightarrow{l} \gamma'$ exactly when we can construct a proof of the validity of the transition using the inference rules. Proofs consist of finite, upwardly branching derivation tree whose nodes are inferences where each inference is justified by an inference rule. The root (conclusion) of a derivation tree is a transition $\gamma \xrightarrow{l} \gamma'$, and the leaves of the derivation tree are *simple rules*, i.e., rules without premises. The set of transitions in a transition system is the set of all $\gamma \xrightarrow{l} \gamma'$ for which we can construct such finite derivation trees.

Transition system semantics are widely used for formalising and studying process algebras and concurrency theory [Mil75, Hoa78], and rules come in different flavours and variants with different meta-theoretical properties. In their overview of SOS rule formats and meta-theory, Mousavi et al. [MRG07] recall:

> If there are negative premises in the semantical rules it is not self-evident anymore whether the rules define a transition relation in an unambiguous way.

The meaning of such rules has been a subject of study in its own right [Gro93, vG04, CMM13]. We note, however, that Plotkin's original lecture notes on SOS did not rely on negative premises, yet showed how to give semantics for a wide range of interesting constructs, including imperative and applicative constructs. In this thesis we will be restricting our attention to rules without negative premises, i.e., premises asserting that some transition is impossible to make.

Another application area for small-step SOS is concurrency [Mil82]. It is challenging to give denotational semantics for concurrency. Pierce [Pie02, page 33] writes:

> The bête noire of denotational semantics turned out to be the treatment of nondeterminism and concurrency.

It is similarly challenging to give big-step rules for concurrency. We discuss this issue further in Section 2.4. In this thesis, we restrict our attention to programming languages with limited non-determinism (the limitation being that non-determinism may only occur in leaves of derivation trees – in reduction semantics, this is known as *unique decomposition* [XSA01, DN04]), interleaving, and concurrency.

We stress that the class of semantics with these restrictions (rules without negative premises and limited non-determinism) still exhibit poor support for language evolution at different levels of abstraction. The remainder of this section recalls the basics of giving small-step SOS specifications, and its (lack of) support for language evolution.

2.2.2 SOS for simple arithmetic expressions

Let us consider a simple arithmetic language.

Abstract syntax. Figure 2.1 defines the *abstract syntax* of the simple arithmetic expression language. We say that it is *abstract*, because it abstracts from the actual parsed token strings of a language. Abstract syntax introduces symbols for syntactic

$\mathit{Expr} \ni e ::= \texttt{plus}(e,e) \mid v$	Expressions
$\mathit{Val} \ni v ::= n$	Values
$n \in \mathbb{N} \triangleq \{0,1,2,\ldots\}$	Natural numbers

Figure 2.1: Abstract syntax for simple arithmetic expressions

$$\frac{e_1 \to e_1'}{\texttt{plus}(e_1,e_2) \to \texttt{plus}(e_1',e_2)} \quad \text{(SOS-Plus1)}$$

$$\frac{e_2 \to e_2'}{\texttt{plus}(n_1,e_2) \to \texttt{plus}(n_1,e_2')} \quad \text{(SOS-Plus2)}$$

$$\frac{}{\texttt{plus}(n_1,n_2) \to n_1 + n_2} \quad \text{(SOS-Plus)}$$

Figure 2.2: SOS for simple arithmetic expressions

sets and meta-variables ranging over these, where meta-variables are distinguished using primes and subscripts.

The syntactic sets *Expr* and *Val* are defined in Figure 2.1 using a context-free grammar specified in a style reminiscent of BNF. The figure also specifies that we use n as a meta-variable to range over the set of natural numbers, where the set of natural numbers is defined using definitional equality '$\triangleq$'. Here, *Val* is a distinct syntactic sort, even though it only has a single production. The motivation for this distinction is to allow extending *Val* with more values later.

Transition relation. The rules in Figure 2.2 define a transition relation. In the rule (SOS-Plus), '$+$' is a primitive operation that adds two natural numbers.

We say that (SOS-Plus) is a *simple* rule, since it has no premises. Each of the rules (SOS-Plus1) and (SOS-Plus2) have a single premise that makes a transition in a sub-expression of `plus`. The conclusion target plugs the result from doing the transition into the configuration from the conclusion source, such that the structure of the rest of that configuration is preserved. Such rules are called *congruence* rules.

Comparing the rules with the definition of labelled terminal transition systems (Definition 2.2), there is an apparent mismatch between this transition relation and the definition of LTTS: the transition relation in Figure 2.2 does not make explicit use of labels, whereas transitions in an LTTS do. This mismatch is, however, superficial: we can safely assume that each arrow is annotated by an implicit unit label in the singleton set $\mathbb{1} \triangleq \{\texttt{unit}\}$. Thus, the set of transitions is $\to\, \subseteq \mathit{Expr} \times \mathbb{1} \times \mathit{Expr}$.

$$\frac{e_1 \to e_1'}{\mathtt{plus}(e_1, e_2) \to \mathtt{plus}(e_1', e_2)} \qquad \text{(SOS-Plus1)}$$

$$\frac{e_2 \to e_2'}{\mathtt{plus}(e_1, e_2) \to \mathtt{plus}(e_1, e_2')} \qquad \text{(SOS-Plus2)}$$

$$\frac{}{\mathtt{plus}(n_1, n_2) \to n_1 + n_2} \qquad \text{(SOS-Plus)}$$

Figure 2.3: SOS for simple arithmetic expressions with interleaving

The transition $\mathtt{plus}(1, \mathtt{plus}(2, \mathtt{plus}(3,4))) \to \mathtt{plus}(1, \mathtt{plus}(2,7))$ is in the set of transitions for $\to$, since we can construct the upwardly branching derivation tree:

$$\text{(SOS-Plus2)}\ \dfrac{\text{(SOS-Plus2)}\ \dfrac{\text{(SOS-Plus)}\ \dfrac{}{\mathtt{plus}(3,4) \to 7}}{\mathtt{plus}(2, \mathtt{plus}(3,4)) \to \mathtt{plus}(2,7)}}{\mathtt{plus}(1, \mathtt{plus}(2, \mathtt{plus}(3,4))) \to \mathtt{plus}(1, \mathtt{plus}(2,7))}$$

Here, inferences are tagged on the left-hand side to indicate which rules they are an instance of. In the rest of this thesis we omit such tags when obvious.

LTTS. The LTTS $\langle \Gamma, L, \to, T \rangle$ gives a semantics for simple arithmetic expressions, where $\Gamma \triangleq \mathit{Expr}$, $L \triangleq \mathbb{1}$, $T \triangleq \mathit{Val}$, and $\to$ is the transition relation defined by the rules in Figure 2.2. Using our transition relation we can compute the result of the expression $\mathtt{plus}(1, \mathtt{plus}(2, \mathtt{plus}(3,4)))$. The computation is given by the following trace in the LTTS:

$$\begin{aligned} & \mathtt{plus}(1, \mathtt{plus}(2, \mathtt{plus}(3,4))) \\ \to\ & \mathtt{plus}(1, \mathtt{plus}(2,7)) \\ \to\ & \mathtt{plus}(1,9) \\ \to\ & 10 \end{aligned}$$

2.2.3 SOS for simple arithmetic expressions with interleaving

One of the merits of SOS is that it allows straightforward specification of interleaving order of evaluation. Figure 2.3 specifies a transition relation for simple arithmetic expressions where the order of evaluation is interleaving. E.g., the following trace is a computation in the LTTS with $\to$ as defined in Figure 2.3:

$$\begin{aligned} & \mathtt{plus}(\mathtt{plus}(1,2), \mathtt{plus}(3,5)) \\ \to\ & \mathtt{plus}(\mathtt{plus}(1,2), 8) \\ \to\ & \mathtt{plus}(3,8) \\ \to\ & 11 \end{aligned}$$

$$\frac{}{\gamma \xrightarrow{\epsilon}^{*} \gamma} \quad \text{(Refl)}$$

$$\frac{\gamma \xrightarrow{l} \gamma' \qquad \gamma' \xrightarrow{l'}^{*} \gamma''}{\gamma \xrightarrow{l \cdot l'}^{*} \gamma''} \quad \text{(Trans)}$$

Figure 2.4: Reflexive-transitive closure of $\rightarrow \subseteq \Gamma \times L \times \Gamma$

For our simple arithmetic language, the choice of evaluation order is not important: the same expression always evaluates to the same result regardless of the order of evaluation. We can prove this by reasoning formally about sequences of transition steps. To this end, one commonly takes the *reflexive-transitive closure* of the transition relation. Figure 2.4 summarises rules that implement the reflexive transitive closure of a transition relation, where $\cdot \in L \rightarrow L \rightarrow L$ is an associative infix operator that returns the label resulting from concatenating two labels, and $\epsilon \in L$ is the unit of concatenation; i.e., it satisfies the following laws:

$$(l_1 \cdot l_2) \cdot l_3 = l_1 \cdot (l_2 \cdot l_3)$$
$$\epsilon \cdot l = l$$
$$l \cdot \epsilon = l$$

Using the reflexive-transitive closure of the transition relation in Figure 2.3, we can prove that simple arithmetic expressions with interleaving are *confluent*.

Informally, a relation is confluent if it, regardless of non-deterministic behaviour, always yields the same configuration and state. We can prove that this holds for the transitive closure of the interleaving transition relation in Figure 2.3. The property can be proven using the Tait-Martin Löf proof method using so-called “strip lemmas”. The proof method is described in texts such as [Bar84, Pol95].

Confluence gives us that the order of evaluation is not important for simple arithmetic expressions. However, if we were to have imperative features in our language, the order of evaluation could affect the results of evaluation. We return to this point in Section 2.2.6.

2.2.4 SOS for λ_{cbv}

Let us now consider an extension of our simple arithmetic language to include more interesting features. We extend our language with the call-by-value λ-calculus, and refer to the resulting language as λ_{cbv}.

Abstract syntax. Figure 2.5 summarises the abstract syntax for λ_{cbv}. Here, we follow Plotkin [Plo04, p. 91] and use closures to represent statically-scoped λ-abstractions,

$$\begin{aligned}
\mathit{Expr} \ni e &::= \ldots \mid \lambda x.e \mid e\ e \mid x && \text{Expressions}\\
\mathit{Val} \ni v &::= \ldots \mid \langle x, e, \rho \rangle && \text{Values}\\
x, y \in \mathit{Var} &\triangleq \{\mathrm{x}, \mathrm{y}, \ldots\} && \text{Variables}\\
\rho \in \mathit{Env} &\triangleq \mathit{Var} \xrightarrow{\text{fin}} \mathit{Val} && \text{Environments}
\end{aligned}$$

Figure 2.5: Abstract syntax for λ_{cbv} (extends Figure 2.1)

where a closure $\langle x, e, \rho \rangle$ records the variable x that the abstraction binds; the body of the abstraction e; and an environment, ρ, that records a set of bindings for statically scoped variables. Here, an environment is a finite map from variables to values. For looking up the value bound to variable x in a map, we write $\rho(x)$. If x is not in the domain of ρ, the operation is undefined. For environment updates, we write $\rho[x \mapsto v]$ for the update of ρ with v at x.

Transition relation. In order to capture the semantics of substituting variables with values, we use environments to record substitutions in an *auxiliary entity* in the configuration of the underlying transition system. Thus, as opposed to simple arithmetic expressions, where configurations consisted only of expressions, configurations are now pairs of environments and expressions, i.e., $\Gamma \triangleq \mathit{Env} \times \mathit{Expr}$ and $\rightarrow \subseteq \Gamma \times \mathbb{1} \times \Gamma$. Environments record a fixed set of bindings, and for any transition $(\rho, e) \rightarrow (\rho', e')$ it holds that $\rho = \rho'$. We follow Plotkin [Plo81] and write such transitions as a judgment $\rho \vdash e \rightarrow e'$.

Environments must be propagated in all rules, and so we must modify the rules for simple arithmetic expressions in order to extend the language. Figure 2.6 summarises the rules that define the transition relation for λ_{cbv}. The rule (SOS-λ_{cbv}-Var) has a premise, but we still say that it is a simple rule, since its premise is a *side-condition*, i.e., a constraint that does not involve the transition relation.

The rules for λ_{cbv} implement deterministic left-to-right order of evaluation. We could have chosen to give an interleaving semantics instead. Such a semantics is summarised in Figure 2.7. It is well-known (e.g., [Pol95]) that such a semantics for λ-calculus is confluent (known as the Church-Rosser theorem [CR36]).

Divergence. The untyped call-by-value λ-calculus has terms whose evaluation diverge. We can use *coinduction* to formalise and reason about these terms. A common approach to formalising the set of terms that diverge is to give a coinductively defined rule, like the one in Figure 2.8. Using this rule, we can only construct infinite derivation trees: there are no simple rules for $\overset{\infty}{\rightarrow}$. Thus, we cannot use induction to reason about the relation. Instead, we use coinduction.

Coinduction allows us to reason about possibly-infinite structures. But how does one construct and reason about infinite structures? In this thesis we follow the ap-

$$\frac{\rho \vdash e_1 \to e_1'}{\rho \vdash \mathtt{plus}(e_1, e_2) \to \mathtt{plus}(e_1', e_2)} \qquad \text{(SOS-}\lambda_{\mathrm{cbv}}\text{-Plus1)}$$

$$\frac{\rho \vdash e_2 \to e_2'}{\rho \vdash \mathtt{plus}(n_1, e_2) \to \mathtt{plus}(n_1, e_2')} \qquad \text{(SOS-}\lambda_{\mathrm{cbv}}\text{-Plus2)}$$

$$\frac{}{\rho \vdash \mathtt{plus}(n_1, n_2) \to n_1 + n_2} \qquad \text{(SOS-}\lambda_{\mathrm{cbv}}\text{-Plus)}$$

$$\frac{}{\rho \vdash \lambda x.e \to \langle x, e, \rho \rangle} \qquad \text{(SOS-}\lambda_{\mathrm{cbv}}\text{-Lam)}$$

$$\frac{\rho \vdash e_1 \to e_1'}{\rho \vdash e_1\ e_2 \to e_1'\ e_2} \qquad \text{(SOS-}\lambda_{\mathrm{cbv}}\text{-App1)}$$

$$\frac{\rho \vdash e_2 \to e_2'}{\rho \vdash \langle x, e, \rho' \rangle\ e_2 \to \langle x, e, \rho' \rangle\ e_2'} \qquad \text{(SOS-}\lambda_{\mathrm{cbv}}\text{-App2)}$$

$$\frac{\rho'[x \mapsto v_2] \vdash e \to e'}{\rho \vdash \langle x, e, \rho' \rangle\ v_2 \to \langle x, e', \rho' \rangle\ v_2} \qquad \text{(SOS-}\lambda_{\mathrm{cbv}}\text{-AppC)}$$

$$\frac{}{\rho \vdash \langle x, v, \rho' \rangle\ v_2 \to v} \qquad \text{(SOS-}\lambda_{\mathrm{cbv}}\text{-App)}$$

$$\frac{x \in \mathrm{dom}(\rho)}{\rho \vdash x \to \rho(x)} \qquad \text{(SOS-}\lambda_{\mathrm{cbv}}\text{-Var)}$$

Figure 2.6: SOS for λ_{cbv}

proach implemented in the Coq proof assistant. In Coq, infinite structures can be constructed and reasoned about using *co-recursive* functions that are *productive*. Productive co-recursive functions are the dual to structurally decreasing recursive functions. Recall that recursive functions *consume* data of inductive types, and that a function is structurally decreasing when each recursive call is on something that is (eventually) structurally smaller than the argument of the current call. For example, the following `boring` recursive Coq function iterates through a list of natural numbers and returns the empty list:

```
Fixpoint boring (l : list nat) : list nat :=
  match l with
    | n :: l' => boring l'
    | [] => []
  end.
```

The `boring` function is structurally decreasing since each recursive call of it is on a list that is smaller than the one in the previous call. Compare with the following productive co-recursive function defined over the datatype `stream`:

$$\frac{\rho \vdash e_1 \rightarrow e_1'}{\rho \vdash \mathtt{plus}(e_1, e_2) \rightarrow \mathtt{plus}(e_1', e_2)} \quad \text{(SOS-}\lambda_{\text{cbv}}'\text{-Plus1)}$$

$$\frac{\rho \vdash e_2 \rightarrow e_2'}{\rho \vdash \mathtt{plus}(e_1, e_2) \rightarrow \mathtt{plus}(e_1, e_2')} \quad \text{(SOS-}\lambda_{\text{cbv}}'\text{-Plus2)}$$

$$\frac{}{\rho \vdash \mathtt{plus}(n_1, n_2) \rightarrow n_1 + n_2} \quad \text{(SOS-}\lambda_{\text{cbv}}'\text{-Plus)}$$

$$\frac{}{\rho \vdash \lambda x.e \rightarrow \langle x, e, \rho \rangle} \quad \text{(SOS-}\lambda_{\text{cbv}}'\text{-Lam)}$$

$$\frac{\rho \vdash e_1 \rightarrow e_1'}{\rho \vdash e_1\ e_2 \rightarrow e_1'\ e_2} \quad \text{(SOS-}\lambda_{\text{cbv}}'\text{-App1)}$$

$$\frac{\rho \vdash e_2 \rightarrow e_2'}{\rho \vdash e_1\ e_2 \rightarrow e_1\ e_2'} \quad \text{(SOS-}\lambda_{\text{cbv}}'\text{-App2)}$$

$$\frac{\rho'[x \mapsto v_2] \vdash e \rightarrow e'}{\rho \vdash \langle x, e, \rho' \rangle\ v_2 \rightarrow \langle x, e', \rho' \rangle\ v_2} \quad \text{(SOS-}\lambda_{\text{cbv}}'\text{-AppC)}$$

$$\frac{}{\rho \vdash \langle x, v, \rho' \rangle\ v_2 \rightarrow v} \quad \text{(SOS-}\lambda_{\text{cbv}}'\text{-App)}$$

$$\frac{x \in \text{dom}(\rho)}{\rho \vdash x \rightarrow \rho(x)} \quad \text{(SOS-}\lambda_{\text{cbv}}'\text{-Var)}$$

Figure 2.7: Interleaving SOS for λ_{cbv}

$$\frac{\gamma \xrightarrow{l} \gamma' \quad \gamma' \overset{\infty}{\rightarrow}}{\gamma \overset{\infty}{\rightarrow}} \quad \text{(TransInf)}$$

Figure 2.8: Coinductive infinite closure of $\rightarrow$

```
CoInductive stream (A : Type) := Cons (a:A) (s:stream A).

CoFixpoint add_one (s : stream nat) : stream nat :=
  match s with
    | Cons n s’ => Cons (n+1) (add_one s’)
  end.
```

In this example, the function `add_one` takes an infinite stream of natural numbers and adds one to each natural number of the stream, such that each co-recursive call produces an element of the result stream before making the recursive call. Co-recursive functions can be called lazily to give finite approximations of (possibly-)infinite structures.

Coq automatically checks that a function is productive by a so-called *guardedness check* (see, e.g., [BC04, LG09, Chl11] for more details). In this thesis, by coinductive proofs we understand guarded coinductive proofs that can be implemented in and accepted by Coq. For more details on coinduction, we refer the reader to introductory textbooks, such as Pierce's [Pie02, Chapter 21] or Sangiorgi's [San11].

We can use guarded coinduction to prove that self-applicative expressions, like $\omega \triangleq (\lambda x.x\ x)\ (\lambda x.x\ x)$, diverge.

Proposition 2.3 (ω diverges) The expression ω diverges, i.e.:

$$\rho \vdash \omega \overset{\infty}{\rightarrow}$$

Proof. The proof is surprisingly tricky when reasoning by guarded coinduction on the structure of rules. E.g., applying (TransInf) twice, and each of (SOS-λ_{cbv}-App1) and (SOS-λ_{cbv}-App2) once, we get the goal:

$$\rho \vdash \langle x, x\ x, \rho\rangle\ \langle x, x\ x, \rho\rangle \overset{\infty}{\rightarrow} \tag{Goal}$$

It seems sensible to use this goal as our coinduction hypothesis, i.e.:

$$\rho \vdash \langle x, x\ x, \rho\rangle\ \langle x, x\ x, \rho\rangle \overset{\infty}{\rightarrow} \tag{CIH}$$

By applying (TransInf) twice, and (SOS-λ_{cbv}-App) and (SOS-λ_{cbv}-Var) twice, the goal becomes:

$$\rho \vdash \langle x, \langle x, x\ x, \rho\rangle\ \langle x, x\ x, \rho\rangle, \rho\rangle\ \langle x, x\ x, \rho\rangle \overset{\infty}{\rightarrow} \tag{Goal}$$

Although the term in this goal is equivalent (as we shortly argue) to the coinduction hypothesis, (CIH) does trivially match the goal. The problem is only aggravated if we continue to apply (TransInf) and the matching small-step rules to continue the derivation: the catch is that the expression grows in each step, so the coinduction hypothesis cannot be trivially applied.

The issue arises due to the way closures are formalised, and does not apply to substitution-based approaches [LG09]. The issue is resolvable without substitution by: introducing a rewriting relation that allows us to simplify a term to prevent its growth after each step; using the rewriting relation to define a relaxed infinite closure; proving that the relaxed closure is sound; and using it to show that ω diverges. The proof is given in the Coq formalisation accompanying this thesis (see `omg_div` in `sos/lamcbvlet/small_lamcbvlet.v`).[2]

In Proposition 2.8 we prove that ω diverges using natural semantics with closures, where the property follows by guarded coinduction alone, without any rewriting. □

We have recalled how to extend an SOS for a simple arithmetic language with the call-by-value λ-calculus. The extension required us to modify all rules in our language.

[2]Available on: `http://cs.swansea.ac.uk/~cscbp/xtss.zip`

$$\mathit{Expr} \ni e ::= \ \ldots \mid \texttt{throw}(e) \mid \texttt{catch}(e,e) \mid \texttt{EXC}(v)$$

Figure 2.9: Abstract syntax for $\lambda^{\bullet}_{\text{cbv}}$ (extends Figure 2.5)

$$\frac{\rho \vdash e \to e'}{\rho \vdash \texttt{throw}(e) \to \texttt{throw}(e')} \quad \text{(SOS-Throw1)}$$

$$\frac{}{\rho \vdash \texttt{throw}(v) \to \texttt{EXC}(v)} \quad \text{(SOS-Throw)}$$

$$\frac{\rho \vdash e_1 \to e_1'}{\rho \vdash \texttt{catch}(e_1,e_2) \to \texttt{catch}(e_1',e_2)} \quad \text{(SOS-Catch1)}$$

$$\frac{}{\rho \vdash \texttt{catch}(v_1,e_2) \to v_1} \quad \text{(SOS-CatchV)}$$

$$\frac{}{\rho \vdash \texttt{catch}(\texttt{EXC}(v),e_2) \to e_2\ v} \quad \text{(SOS-CatchE)}$$

Figure 2.10: SOS for exception handling (extends Figure 2.6)

2.2.5 SOS for $\lambda^{\bullet}_{\text{cbv}}$

Now, suppose we want to extend λ_{cbv} with abrupt termination, namely throwing and handling exceptions.

Abstract syntax. The abstract syntax for our language is modified as specified in Figure 2.9. The new constructs introduced are a `throw` construct for throwing exceptions, a `catch` construct for catching and handling them, and an $\texttt{EXC}(v)$ expression that represents a thrown exception.

Transition relation. The rules for the newly introduced constructs are summarised in Figure 2.10. In addition to those rules, we need rules for propagating exceptions. The traditional approach to propagating exceptions is to add explicit propagation rules. Figure 2.11 gives such rules. The approach inherent to these rules differs slightly from Plotkin's exposition [Plo81], but achieve the same effect of propagating abrupt termination using slightly more concise rules. Plotkin's approach to propagating abrupt termination is recalled in Appendix C.2.

As we extend our language with new constructs, rules for propagating exceptions must be added for those constructs too. Such rules are tedious and error-prone to write and read.

$$\frac{}{\rho \vdash \texttt{plus}(\text{EXC}(v), e) \to \text{EXC}(v)} \quad \text{(SOS-}\lambda^{\bullet}_{\text{cbv}}\text{-Plus1-Exc)}$$

$$\frac{}{\rho \vdash \texttt{plus}(e, \text{EXC}(v)) \to \text{EXC}(v)} \quad \text{(SOS-}\lambda^{\bullet}_{\text{cbv}}\text{-Plus2-Exc)}$$

$$\frac{}{\rho \vdash \text{EXC}(v)\ e \to \text{EXC}(v)} \quad \text{(SOS-}\lambda^{\bullet}_{\text{cbv}}\text{-App1-Exc)}$$

$$\frac{}{\rho \vdash e\ \text{EXC}(v) \to \text{EXC}(v)} \quad \text{(SOS-}\lambda^{\bullet}_{\text{cbv}}\text{-App2-Exc)}$$

$$\frac{}{\rho \vdash \texttt{throw}(\text{EXC}(v)) \to \text{EXC}(v)} \quad \text{(SOS-}\lambda^{\bullet}_{\text{cbv}}\text{-Throw-Exc)}$$

Figure 2.11: SOS rules for propagating exceptions in $\lambda^{\bullet}_{\text{cbv}}$

$$\begin{array}{rll}
\mathit{Expr} \ni e ::= & \dots \mid \texttt{ref}(e) \mid \texttt{deref}(e) \mid \texttt{assign}(e,e) & \text{Expressions} \\
\mathit{Val} \ni v ::= & \dots \mid r & \text{Values} \\
r \in \mathit{Ref} \triangleq & \{\mathrm{r}_1, \mathrm{r}_2, \dots\} & \text{References} \\
\sigma \in \mathit{Store} \triangleq & \mathit{Ref} \xrightarrow{\text{fin}} \mathit{Val} & \text{Stores}
\end{array}$$

Figure 2.12: Abstract syntax for $\lambda^{\bullet}_{\text{cbv+}\texttt{ref}}$ (extends Figure 2.9)

LTTS. The LTTS is given by the tuple $\langle \Gamma, L, \to, T \rangle$ where $\Gamma \triangleq \mathit{Env} \times \mathit{Expr}$; $L \triangleq \mathbb{1}$; $\to \subseteq \Gamma \times \mathbb{1} \times \Gamma$ is as defined in Figures 2.6, 2.10, and 2.11; and $T \triangleq \mathit{Val} \cup \{\text{EXC}(v)\}$.

2.2.6 SOS for $\lambda^{\bullet}_{\text{cbv+}\texttt{ref}}$

The final extension of our example SOS semantics that we consider is the extension with ML-style references [MTHM97].

Abstract syntax. The extended abstract syntax is summarised in Figure 2.12. Stores are given by finite maps from references to values. Here, the `ref` construct allocates a reference; `deref` deallocates a reference in the current store; and `assign` is used for updating the value that a given reference refers to in the store.

Transition relation. In order to track the state of stores, our transition relation must propagate stores between transitions. Thus, the notion of configuration becomes $\Gamma \subseteq \mathit{Env} \times \mathit{Expr} \times \mathit{Store}$. Extending a language with stores in SOS gives rise to modifying all rules in a language. Since our language has abrupt termination, we also need to carefully give rules propagating exceptions for new constructs. Figure 2.13 summarises

$$\frac{\rho \vdash (e_1,\sigma) \to (e_1',\sigma')}{\rho \vdash (\texttt{plus}(e_1,e_2),\sigma) \to (\texttt{plus}(e_1',e_2),\sigma')} \quad (\text{SOS-}\lambda^\bullet_{\text{cbv+}\texttt{ref}}\text{-Plus1})$$

$$\frac{\rho \vdash (e_2,\sigma) \to (e_2',\sigma')}{\rho \vdash (\texttt{plus}(n_1,e_2),\sigma) \to (\texttt{plus}(n_1,e_2'),\sigma')} \quad (\text{SOS-}\lambda^\bullet_{\text{cbv+}\texttt{ref}}\text{-Plus2})$$

$$\frac{}{\rho \vdash (\texttt{plus}(n_1,n_2),\sigma) \to (n_1+n_2,\sigma)} \quad (\text{SOS-}\lambda^\bullet_{\text{cbv+}\texttt{ref}}\text{-Plus})$$

$$\frac{}{\rho \vdash (\lambda x.e,\sigma) \to (\langle x,e,\rho\rangle,\sigma)} \quad (\text{SOS-}\lambda^\bullet_{\text{cbv+}\texttt{ref}}\text{-Lam})$$

$$\frac{\rho \vdash (e_1,\sigma) \to (e_1',\sigma')}{\rho \vdash (e_1\ e_2,\sigma) \to (e_1'\ e_2,\sigma')} \quad (\text{SOS-}\lambda^\bullet_{\text{cbv+}\texttt{ref}}\text{-App1})$$

$$\frac{\rho \vdash (e_2,\sigma) \to (e_2',\sigma')}{\rho \vdash (v_1\ e_2,\sigma') \to (v_1\ e_2',\sigma')} \quad (\text{SOS-}\lambda^\bullet_{\text{cbv+}\texttt{ref}}\text{-App2})$$

$$\frac{\rho'[x \mapsto v_2] \vdash (e,\sigma) \to (e',\sigma')}{\rho \vdash (\langle x,e,\rho'\rangle\ v_2,\sigma) \to (\langle x,e',\rho'\rangle\ v_2,\sigma')} \quad (\text{SOS-}\lambda^\bullet_{\text{cbv+}\texttt{ref}}\text{-AppC})$$

$$\frac{}{\rho \vdash (\langle x,v,\rho'\rangle\ v_2,\sigma) \to (v,\sigma)} \quad (\text{SOS-}\lambda^\bullet_{\text{cbv+}\texttt{ref}}\text{-App})$$

$$\frac{x \in \text{dom}(\rho)}{\rho \vdash (x,\sigma) \to (\rho(x),\sigma)} \quad (\text{SOS-}\lambda^\bullet_{\text{cbv+}\texttt{ref}}\text{-Var})$$

$$\frac{}{\rho \vdash (\texttt{throw}(v),\sigma) \to (\texttt{EXC}(v),\sigma)} \quad (\text{SOS-}\lambda^\bullet_{\text{cbv+}\texttt{ref}}\text{-Throw})$$

$$\frac{\rho \vdash (e_1,\sigma) \to (e_1',\sigma')}{\rho \vdash (\texttt{catch}(e_1,e_2),\sigma) \to (\texttt{catch}(e_1',e_2),\sigma')} \quad (\text{SOS-}\lambda^\bullet_{\text{cbv+}\texttt{ref}}\text{-Catch1})$$

$$\frac{}{\rho \vdash (\texttt{catch}(v_1,e_2),\sigma) \to (v_1,\sigma)} \quad (\text{SOS-}\lambda^\bullet_{\text{cbv+}\texttt{ref}}\text{-CatchV})$$

$$\frac{}{\rho \vdash (\texttt{catch}(\texttt{EXC}(v),e_2),\sigma) \to (e_2\ v,\sigma)} \quad (\text{SOS-}\lambda^\bullet_{\text{cbv+}\texttt{ref}}\text{-CatchE})$$

Figure 2.13: Updated SOS for $\lambda^\bullet_{\text{cbv}}$ fragment of $\lambda^\bullet_{\text{cbv+}\texttt{ref}}$

how the rules for the $\lambda^\bullet_{\text{cbv}}$-fragment of our language are updated to propagate stores; Figure 2.14 summarises the rules for the fragment of our language with references; and Figure 2.15 gives rules for propagating exceptions.

If we were to consider an interleaving semantics for $\lambda^\bullet_{\text{cbv+}\texttt{ref}}$, the semantics is no longer confluent: consider, for example, the expression:

$$\texttt{seq}(\texttt{plus}(\texttt{seq}(\texttt{assign}(r,1),1),\texttt{seq}(\texttt{assign}(r,2),2)),\texttt{deref}(r))$$

Here, $\texttt{seq}(e_1,e_2) \triangleq (\lambda\_.e_2)\ e_1$. Evaluating this term in a store σ where $r \in \text{dom}(\sigma)$ gives

$$\frac{\rho \vdash (e,\sigma) \to (e',\sigma')}{\rho \vdash (\texttt{ref}(e),\sigma) \to (\texttt{ref}(e'),\sigma')} \qquad \text{(SOS-}\lambda^{\bullet}_{\text{cbv+}\texttt{ref}}\text{-Ref1)}$$

$$\frac{r \notin \text{dom}(\sigma)}{\rho \vdash (\texttt{ref}(v),\sigma) \to (r,\sigma[r \mapsto v])} \qquad \text{(SOS-}\lambda^{\bullet}_{\text{cbv+}\texttt{ref}}\text{-Ref)}$$

$$\frac{\rho \vdash (e,\sigma) \to (e',\sigma')}{\rho \vdash (\texttt{deref}(e),\sigma) \to (\texttt{deref}(e'),\sigma')} \qquad \text{(SOS-}\lambda^{\bullet}_{\text{cbv+}\texttt{ref}}\text{-Deref1)}$$

$$\frac{r \in \text{dom}(\sigma)}{\rho \vdash (\texttt{deref}(r),\sigma) \to (\sigma(r),\sigma)} \qquad \text{(SOS-}\lambda^{\bullet}_{\text{cbv+}\texttt{ref}}\text{-Deref)}$$

$$\frac{\rho \vdash (e_1,\sigma) \to (e_1',\sigma')}{\rho \vdash (\texttt{assign}(e_1,e_2),\sigma) \to (\texttt{assign}(e_1',e_2),\sigma')} \qquad \text{(SOS-}\lambda^{\bullet}_{\text{cbv+}\texttt{ref}}\text{-Assign1)}$$

$$\frac{\rho \vdash (e_2,\sigma) \to (e_2',\sigma')}{\rho \vdash (\texttt{assign}(r,e_2),\sigma) \to (\texttt{assign}(r,e_2'),\sigma')} \qquad \text{(SOS-}\lambda^{\bullet}_{\text{cbv+}\texttt{ref}}\text{-Assign2)}$$

$$\frac{r \in \text{dom}(\sigma)}{\rho \vdash (\texttt{assign}(r,v),\sigma) \to (\texttt{unit},\sigma[r \mapsto v])} \qquad \text{(SOS-}\lambda^{\bullet}_{\text{cbv+}\texttt{ref}}\text{-Assign)}$$

Figure 2.14: SOS rules for ref fragment of $\lambda^{\bullet}_{\text{cbv+}\texttt{ref}}$

$$\frac{}{\rho \vdash (\texttt{plus}(\text{EXC}(v),e),\sigma) \to (\text{EXC}(v),\sigma)} \qquad \text{(SOS-}\lambda^{\bullet}_{\text{cbv+}\texttt{ref}}\text{-Plus1-Exc)}$$

$$\frac{}{\rho \vdash (\texttt{plus}(e,\text{EXC}(v)),\sigma) \to (\text{EXC}(v),\sigma)} \qquad \text{(SOS-}\lambda^{\bullet}_{\text{cbv+}\texttt{ref}}\text{-Plus2-Exc)}$$

$$\frac{}{\rho \vdash (\text{EXC}(v)\ e,\sigma) \to (\text{EXC}(v),\sigma)} \qquad \text{(SOS-}\lambda^{\bullet}_{\text{cbv+}\texttt{ref}}\text{-App1-Exc)}$$

$$\frac{}{\rho \vdash (e\ \text{EXC}(v),\sigma) \to (\text{EXC}(v),\sigma)} \qquad \text{(SOS-}\lambda^{\bullet}_{\text{cbv+}\texttt{ref}}\text{-App2-Exc)}$$

$$\frac{}{\rho \vdash (\texttt{throw}(\text{EXC}(v)),\sigma) \to (\text{EXC}(v),\sigma)} \qquad \text{(SOS-}\lambda^{\bullet}_{\text{cbv+}\texttt{ref}}\text{-Throw-Exc)}$$

$$\frac{}{\rho \vdash (\texttt{ref}(\text{EXC}(v)),\sigma) \to (\text{EXC}(v),\sigma)} \qquad \text{(SOS-}\lambda^{\bullet}_{\text{cbv+}\texttt{ref}}\text{-Ref-Exc)}$$

$$\frac{}{\rho \vdash (\texttt{deref}(\text{EXC}(v)),\sigma) \to (\text{EXC}(v),\sigma)} \qquad \text{(SOS-}\lambda^{\bullet}_{\text{cbv+}\texttt{ref}}\text{-Deref-Exc)}$$

$$\frac{}{\rho \vdash (\texttt{assign}(\text{EXC}(v),e),\sigma) \to (\text{EXC}(v),\sigma)} \qquad \text{(SOS-}\lambda^{\bullet}_{\text{cbv+}\texttt{ref}}\text{-Assn1-Exc)}$$

$$\frac{}{\rho \vdash (\texttt{assign}(e,\text{EXC}(v)),\sigma) \to (\text{EXC}(v),\sigma)} \qquad \text{(SOS-}\lambda^{\bullet}_{\text{cbv+}\texttt{ref}}\text{-Assn2-Exc)}$$

Figure 2.15: SOS rules for propagating exceptions in $\lambda^{\bullet}_{\text{cbv+}\texttt{ref}}$

either 1 or 2, depending on the evaluation order.

We have recalled traditional approaches to giving semantics and proving properties about it using SOS. The resulting rules are simple and purely structural: they do not have any explicit representation of program context, but gives meaning to terms by means of a transition relation that is defined over the structure of program terms. Although rules are simple, each extension required us to either modify rules for existing constructs, or introduce new rules for existing constructs. We also witnessed an explosion in the number of rules required to propagate exceptions.

2.3 Modular SOS

Modular SOS [Mos04] (MSOS) was introduced in order to address the problems with poor modularity in SOS. Mosses achieves this by extending the notion of transition system we saw in Definition 2.2 to *generalised transition systems*, and by using MSOS rules for transition relations.

2.3.1 Generalised transition system semantics

Definition 2.4 recalls generalised terminal transition systems, due to Mosses [Mos04] (who calls them just "generalised transition systems" – here we include the 'terminal' modifier, in order to avoid confusion later in Chapter 3 where we consider more lax transition system variants).

Definition 2.4 (Generalised terminal transition systems) A generalised terminal transition system (GTTS) is a quadruple $\langle \Gamma, \mathbb{C}, \rightarrow, T \rangle$ where $\mathbb{C}$ is a category with morphisms L, such that $\langle \Gamma, L, \rightarrow, T \rangle$ is an LTTS.

A computation in a GTTS is a computation in the underlying LTTS such that its trace is a path in the category $\mathbb{C}$: whenever a transition labelled ℓ is followed immediately by a transition labelled ℓ', the labels ℓ, ℓ' are required to be composable in $\mathbb{C}$.

Definition 2.5 recalls what we understand by a category (where the definition here is borrowed from Pierce [Pie91], but see [ML71] or [Awo06] for more classical and detailed introductions to category theory).

Definition 2.5 A category comprises:

1. a collection of objects;
2. a collection of morphisms;
3. operations assigning to each morphism f an object $\mathrm{dom}(f)$, its domain, and an object $\mathrm{cod}(f)$, its codomain;
4. a composition operator assigning to each pair of morphisms f and g, where $\mathrm{cod}(f) = \mathrm{dom}(g)$, a composite morphism $f \mathbin{;} g : \mathrm{dom}(f) \rightarrow \mathrm{cod}(g)$, satisfying the following associative law:

for any morphisms $f : A \to B, g : B \to C$, and $h : C \to D$ (with A, B, C, and D not necessarily distinct),

$$f \fatsemi (g \fatsemi h) = (f \fatsemi g) \fatsemi h;$$

5. for each object A, an identity morphism $id_A : A \to A$ satisfying the following identity law:

   for any morphism $f : A \to B$

$$f \fatsemi id_B = f \quad \text{and} \quad id_A \fatsemi f = f.$$

Labels and configurations. Definition 2.4 states that labels in Modular SOS are morphisms in a category $\mathbb{C}$. Modular SOS enforces that configurations in GTTSs are restricted to abstract syntax and computed values, and encode auxiliary entities as objects and morphisms in the category $\mathbb{C}$.

The original foundations of Modular SOS [Mos99] used a specialised notion of label categories, and provided so-called *label transformers* for extending label categories, analogous to monad transformers in denotational semantics [Mog91, LHJ95]. Subsequently, Mosses [Mos04] adopted *indexed product categories* for labels without any loss of generality.

An indexed product category is given by some $\mathbb{C} \triangleq \prod_{i \in J} \mathbb{C}_i$, where J is a set of indices. In order to extend the product one simply adds a new index j to the index set, and a new category $\mathbb{C}_j$ to the product category $\mathbb{C}$. Modular SOS does not constrain which types of categories this product may contain, but does provide three basic *label categories*:

- *Discrete category:* there is a single (identity) morphism for each object. Such morphisms represent information that can be inspected but not changed by a transition, i.e., they describe the behaviour of auxiliary entities such as the environments we saw in Section 2.2.4.

- *Preorder category:* the set of objects are preordered by their morphisms. Such morphisms represent information that can be inspected and changed by a transition. If it is unchanged, the morphism is the identity morphism on an object. Morphisms describe the behaviour of auxiliary entities such as the stores we saw in Section 2.2.6.

- *Free monoid:* there is a single object whose morphisms represent finite sequences. Composition corresponds to sequence concatenation. Such morphisms correspond to the concatenable labels in LTTSs that we recalled in Section 2.2.3.

Definition 2.6, due to Mosses [Mos04, Definition 5], defines the indexed product category used in Modular SOS labels.

Definition 2.6 (Component category) Let RO, RW, and WO be disjoint sets of indices, and $I \triangleq RO \cup RW \cup WO$. For each $i \in I$ let a set O_i be given, such that whenever $i \in WO$, O_i is a monoid. Each O_i determines a component category $\mathbb{C}_i$, as follows:

- if $i \in RO$, then $\mathbb{C}_i$ is the discrete category with O_i as its set of objects and also as its set of (identity) morphisms;
- if $i \in RW$, then $\mathbb{C}_i$ is the preorder category with O_i as its set of objects, and $O_i \times O_i$ as its set of morphisms;
- if $i \in WO$, then $\mathbb{C}_i$ is the category with a single object, and with the monoid O_i as its set of morphisms.

MSOS adopts record-like notation, as known from, e.g., Standard ML [MTHM97], to refer to auxiliary entities by pattern-matching against morphisms. For example:

- $\{\mathbf{env}{=}\rho, \ldots\}$ specifies morphisms, such that projecting index **env** $\in I$ gives an identity morphism on ρ;
- $\{\mathbf{sto}{=}\sigma, \mathbf{sto}'{=}\sigma', \ldots\}$ specifies labels, such that projecting index **sto** $\in I$ gives a morphism between σ and σ'; and
- $\{\mathbf{out}'{=}v, \ldots\}$ specifies labels, such that projecting index $\mathbf{out}' \in I$ gives an identity morphism on v.

Here, we use '$\ldots$' as a formal meta-variable ranging over the remaining morphisms in the product. MSOS also uses a distinguished meta-variable '—' to refer to products of identity morphisms.

2.3.2 MSOS for simple arithmetic expressions

Having recalled the foundations of Modular SOS, we illustrate how it solves the problems with SOS that we saw in the previous section. We consider how to give a GTTS semantics for the simple arithmetic expressions language from Section 2.2.2.

Abstract syntax. The abstract syntax is unchanged from Figure 2.1.

Transition relation. Figure 2.16 specifies a transition relation for simple arithmetic expressions using MSOS rules. The $\{\ldots\}$ label in rules (MSOS-Plus1) and (MSOS-Plus2) is an arbitrary morphism, and the use of the meta-variable '$\ldots$' in both the premise and conclusion ensures that all auxiliary entities are propagated unchanged between the premise and conclusion. The $\{—\}$ label in rule (MSOS-Plus) ensures that no side-effects occur between the source and the target of the transition, and corresponds to an identity morphism in the underlying product category of the generalised transition system.

$$\frac{e_1 \xrightarrow{\{\ldots\}} e_1'}{\texttt{plus}(e_1, e_2) \xrightarrow{\{\ldots\}} \texttt{plus}(e_1', e_2)} \quad \text{(MSOS-Plus1)}$$

$$\frac{e_2 \xrightarrow{\{\ldots\}} e_2'}{\texttt{plus}(n_1, e_2) \xrightarrow{\{\ldots\}} \texttt{plus}(n_1, e_2')} \quad \text{(MSOS-Plus2)}$$

$$\frac{}{\texttt{plus}(n_1, n_2) \xrightarrow{\{-\}} n_1 + n_2} \quad \text{(MSOS-Plus)}$$

Figure 2.16: MSOS for simple arithmetic expressions

GTTS. The tuple $\langle \Gamma, \mathbb{C}, \rightarrow, T \rangle$ is a generalised transition system for simple arithmetic expressions, where $\Gamma \triangleq \mathit{Expr}$; $\mathbb{C}$ is a default category with a single object with a single identity arrow; $\rightarrow$ is as defined by the rules in Figure 2.16; and $T \triangleq \mathit{Val}$.

2.3.3 MSOS for λ_{cbv}

We extend the generalised transition system for simple arithmetic expressions with the call-by-value λ-calculus.

Abstract syntax. The abstract syntax is the same as in Figure 2.5.

Transition relation. Figure 2.17 summarises the MSOS rules for λ_{cbv}. Unlike the extension we saw with SOS, the MSOS rules for simple arithmetic expressions remain unchanged. Rules that access the environment do so via labels with the structure $\{\mathbf{env}{=}\rho, X\}$ where X is a variable ranging over all unmentioned label components.

GTTS. The tuple $\langle \Gamma, \mathbb{C}, \rightarrow, T \rangle$ is a generalised transition system for λ_{cbv}. Configurations Γ and terminal configurations T are the updated notions of expressions and values, respectively. Now, the indexed product category $\mathbb{C}$ comprises a single category, namely a discrete category whose objects are environments ρ, such that each object has a single identity morphism. Although the product category comprises a single object, rules still parameterise over arbitrary labels, to permit future extensions.

Closures of transition relation. It is a straightforward matter to adapt to MSOS the reflexive-transitive closure and infinite closure that we used to reason about sequences of transitions for SOS. Figure 2.18 summarises the MSOS counterparts to these relations. Here, the '$\fatsemi$' operator is the composition operation for morphisms in the underlying product category of the generalised transition system.

$$\frac{}{\lambda x.e \xrightarrow{\{\mathbf{env}=\rho,—\}} \langle x, e, \rho \rangle} \qquad \text{(MSOS-Lam)}$$

$$\frac{e_1 \xrightarrow{\{\ldots\}} e_1'}{e_1\ e_2 \xrightarrow{\{\ldots\}} e_1'\ e_2} \qquad \text{(MSOS-App1)}$$

$$\frac{e_2 \xrightarrow{\{\ldots\}} e_2'}{v_1\ e_2 \xrightarrow{\{\ldots\}} v_1\ e_2'} \qquad \text{(MSOS-App2)}$$

$$\frac{e \xrightarrow{\{\mathbf{env}=\rho'[x\mapsto v_2],\ldots\}} e'}{\langle x, e, \rho' \rangle\ v_2 \xrightarrow{\{\mathbf{env}=\rho,\ldots\}} \langle x, e', \rho' \rangle\ v_2} \qquad \text{(MSOS-AppC)}$$

$$\frac{}{\langle x, v, \rho' \rangle\ v_2 \xrightarrow{\{—\}} v} \qquad \text{(MSOS-App)}$$

$$\frac{x \in \mathrm{dom}(\rho)}{x \xrightarrow{\{\mathbf{env}=\rho,—\}} \rho(x)} \qquad \text{(MSOS-Var)}$$

Figure 2.17: MSOS for λ_{cbv} (extends Figure 2.16)

$$\frac{}{\gamma \xrightarrow{\{—\}}{}^{*}\, \gamma} \qquad \text{(MSOS-Refl)}$$

$$\frac{\gamma \xrightarrow{\ell} \gamma' \qquad \gamma' \xrightarrow{\ell'}{}^{*}\, \gamma''}{\gamma \xrightarrow{\ell \fatsemi \ell'}{}^{*}\, \gamma''} \qquad \text{(MSOS-Trans)}$$

$$\frac{\gamma \xrightarrow{\ell} \gamma' \qquad \gamma' \rightarrow^{\infty} \gamma''}{\gamma \rightarrow^{\infty}} \qquad \text{(MSOS-TransInf)}$$

Figure 2.18: Reflexive-transitive and infinite closure of an MSOS transition relation

$$\begin{aligned} \mathit{Expr} \ni e &::= \ldots \mid \mathtt{throw}(e) \mid \mathtt{catch}(e,e) \mid \mathtt{stuck} \\ \mathit{Exc} \ni \varepsilon &::= \mathrm{OK} \mid \mathrm{EXC}(v) \end{aligned}$$

Figure 2.19: Abstract syntax for $\lambda^{\bullet}_{\text{cbv}}$ (extends Figure 2.5)

2.3.4 MSOS for $\lambda^{\bullet}_{\text{cbv}}$

MSOS also supports modular abrupt termination. The idea (attributed to Klin by Mosses [Mos04, p. 216]) is to model exceptions as signals: when abrupt termination occurs, an observable signal is emitted. Constructs may listen for the signal and intercept it to handle the exception. All programs are wrapped in a top-level `program` construct that listens for signals, such that, if a signal makes its way to the top-level of the program, the `program` construct contracts the current program to a terminal configuration, thereby abruptly terminating the program.

Abstract syntax. There is no longer any need for exceptions to be in the syntactic set *Expr*. Instead, exceptions are encoded by adding a label component ranged over by a syntactic set *Exc*. We also add a new expression form `stuck` which represents a stuck computation. Figure 2.19 gives the abstract syntax for the MSOS specification of $\lambda^{\bullet}_{\text{cbv}}$. Here, OK is the unobservable signal, indicating that no exception is occurring.

Transition relation. The transition relation in Figure 2.20 specifies the semantics of abrupt termination. Once the sub-expression of `throw` has been evaluated to a value v, it makes a transition to a `stuck` expression, which has no further transitions, and emits v has an exception signal.

GTTS. Configurations and terminal configurations are the extended sets of expressions and values summarised in Figure 2.19. The indexed product category $\mathbb{C}$ is augmented with a new category that implements a free monoid: a single object with multiple identity arrows, one for each possible exception that can be thrown, and a distinguished unobservable arrow for OK.

2.3.5 MSOS for $\lambda^{\bullet}_{\text{cbv+}\mathtt{ref}}$

The extension of $\lambda^{\bullet}_{\text{cbv+}\mathtt{ref}}$ with references is also straightforward to specify in MSOS without modifying existing rules.

Abstract syntax. The abstract syntax for the new constructs is identical to that given in Figure 2.12.

$$\frac{e \xrightarrow{\{\ldots\}} e'}{\texttt{throw}(e) \xrightarrow{\{\ldots\}} \texttt{throw}(e')} \qquad \text{(MSOS-Throw1)}$$

$$\frac{}{\texttt{throw}(v) \xrightarrow{\{\mathbf{exc}'=\textsc{exc}(v),—\}} \texttt{stuck}} \qquad \text{(MSOS-Throw)}$$

$$\frac{e_1 \xrightarrow{\{\mathbf{exc}'=\textsc{ok},\ldots\}} e_1'}{\texttt{catch}(e_1,e_2) \xrightarrow{\{\mathbf{exc}'=\textsc{ok},\ldots\}} \texttt{catch}(e_1',e_2)} \qquad \text{(MSOS-Catch1)}$$

$$\frac{}{\texttt{catch}(v,e_2) \xrightarrow{\{—\}} v} \qquad \text{(MSOS-CatchV)}$$

$$\frac{e_1 \xrightarrow{\{\mathbf{exc}'=\textsc{exc}(v),\ldots\}} e_1'}{\texttt{catch}(e_1,e_2) \xrightarrow{\{\mathbf{exc}'=\textsc{ok},\ldots\}} e_2\ v} \qquad \text{(MSOS-CatchE)}$$

$$\frac{e \xrightarrow{\{\mathbf{exc}'=\textsc{ok},\ldots\}} e'}{\texttt{program}(e) \xrightarrow{\{\mathbf{exc}'=\textsc{ok},\ldots\}} \texttt{program}(e')} \qquad \text{(MSOS-Program1)}$$

$$\frac{}{\texttt{program}(v) \xrightarrow{\{—\}} v} \qquad \text{(MSOS-ProgramV)}$$

$$\frac{e \xrightarrow{\{\mathbf{exc}'=\textsc{exc}(v),\ldots\}} e'}{\texttt{program}(e) \xrightarrow{\{\mathbf{exc}'=\textsc{ok},\ldots\}} v} \qquad \text{(MSOS-ProgramE)}$$

Figure 2.20: MSOS for $\lambda^{\bullet}_{\text{cbv}}$ (extends Figure 2.17)

Transition relation. Figure 2.21 specifies the transition relation for the introduced constructs. Constructs that access the store do so via labels whose structure are given by $\{\mathbf{sto}{=}\sigma, \mathbf{sto}'{=}\sigma', X\}$ where X is a variable ranging over unmentioned label components, such that σ is the store before making the transition, and σ' is the store after making the transition.

GTTS. Configurations and terminal configurations are the extended sets of expressions and values summarised in Figure 2.19. The indexed product category $\mathbb{C}$ is augmented with a new preorder category where the objects of the category is the set of stores, and with morphisms between all objects.

2.3.6 MSOS for printing

The semantics of a simple `print` construct is also straightforward to specify in MSOS without modifying or introducing new rules for existing constructs.

$$\frac{e \xrightarrow{\{\ldots\}} e'}{\texttt{ref}(e) \xrightarrow{\{\ldots\}} \texttt{ref}(e')} \qquad \text{(MSOS-Ref1)}$$

$$\frac{r \notin \mathrm{dom}(\sigma)}{\texttt{ref}(v) \xrightarrow{\{\mathbf{sto}=\sigma,\mathbf{sto}'=\sigma[r\mapsto v],—\}} r} \qquad \text{(MSOS-Ref)}$$

$$\frac{e \xrightarrow{\{\ldots\}} e'}{\texttt{deref}(e) \xrightarrow{\{\ldots\}} \texttt{deref}(e')} \qquad \text{(MSOS-Deref1)}$$

$$\frac{r \in \mathrm{dom}(\sigma)}{\texttt{deref}(r) \xrightarrow{\{\mathbf{sto}=\sigma,\mathbf{sto}'=\sigma,—\}} \sigma(r)} \qquad \text{(MSOS-Deref)}$$

$$\frac{e_1 \xrightarrow{\{\ldots\}} e_1'}{\texttt{assign}(e_1,e_2) \xrightarrow{\{\ldots\}} \texttt{assign}(e_1',e_2)} \qquad \text{(MSOS-Assign1)}$$

$$\frac{e_2 \xrightarrow{\{\ldots\}} e_2'}{\texttt{assign}(r,e_2) \xrightarrow{\{\ldots\}} \texttt{assign}(r,e_2')} \qquad \text{(MSOS-Assign2)}$$

$$\frac{r \in \mathrm{dom}(\sigma)}{\texttt{assign}(r,v) \xrightarrow{\{\mathbf{sto}=\sigma,\mathbf{sto}'=\sigma[r\mapsto v],—\}} \texttt{unit}} \qquad \text{(MSOS-Assign)}$$

Figure 2.21: MSOS for $\lambda^{\bullet}_{\text{cbv}+\texttt{ref}}$ (extends Figure 2.20)

$$\mathit{Expr} \ni e ::= \ldots \mid \texttt{print}(e)$$

Figure 2.22: Abstract syntax for printing (extends Figure 2.12)

Abstract syntax. The abstract syntax of the extension is summarised in Figure 2.22.

Transition relation. The rules in Figure 2.23 specify the semantics of printing. These rules make use of a label component **out**′, a “write-only” label component ranged over by lists of values, corresponding to the free monoid with list concetenation as its operator, and the empty list as its unit.

We have recalled how to use MSOS for specifying and reasoning about programming language semantics. MSOS supports adding new constructs and features without modifying existing rules.

$$\frac{e \xrightarrow{\{\ldots\}} e'}{\texttt{print}(e) \xrightarrow{\{\ldots\}} \texttt{print}(e')}$$

$$\frac{}{\texttt{print}(v) \xrightarrow{\{\mathbf{out}'=[v],—\}} \texttt{unit}}$$

Figure 2.23: MSOS for printing (extends Figure 2.21)

2.4 Natural semantics

A virtue of small-step semantics is that it is flexible and permits specifying semantics with interleaving. For languages with a fixed and deterministic left-to-right order of evaluation, such as the $\lambda^{\bullet}_{\text{cbv+}\texttt{ref}}$ language, the big-step style of semantics known as *natural semantics*, due to Kahn [Kah87], is also well-suited.

2.4.1 Natural semantics vs. SOS

Natural semantics is often described as a variant of SOS. One talks about transition system semantics as being small-step, since the transitions represent a computation as a sequence of steps between intermediate states. In contrast, natural semantics is said to be big-step, since rules define a relation which relates program expressions directly to their final results in what we regard as one 'big step'.

Plotkin's original SOS exposition uses a related notion of big steps, although his notion is defined in a slightly different way: Plotkin [Plo04, page 33] uses the reflexive-transitive closure in premises of transition rules to get a big-step effect. For example, the following rule specifies a Plotkin-style big-step semantics of `plus` via the reflexive-transitive closure:

$$\frac{e_1 \to^* n_1 \qquad e_2 \to^* n_2}{\texttt{plus}(e_1, e_2) \to n_1 + n_2} \qquad \text{(SOS-Plotkin-Big-Plus)}$$

In contrast, natural semantics typically abstract from intermediate steps and from the underlying notion of transition system. Rules and proof derivations in natural semantics closely resemble proofs in natural deduction (the well-known logic framework – for an introduction see, e.g., [Man74]), which is where natural semantics derives its name from.

When natural semantics is often described as a variant of SOS, it is due to the close relationship between natural semantics and SOS: a natural semantics can be thought of as a transition system specification that abstracts from intermediate steps, and relates a set of initial configurations (the domain of a big-step relation) to a set of final configurations (the codomain of a big-step relation). We recall how to specify the example languages from Section 2.2 using natural semantics, and compare and contrast with SOS.

$$\begin{aligned} \mathit{Expr}^{\mathrm{NS}} \ni e &::= \mathtt{plus}(e,e) \mid \mathtt{num}(n) && \text{Expressions} \\ \mathit{Val}^{\mathrm{NS}} \ni v &::= n && \text{Values} \\ n \in \mathbb{N} &\triangleq \{0,1,2,\ldots\} && \text{Natural numbers} \end{aligned}$$

Figure 2.24: Abstract syntax and semantic values for simple arithmetic expressions

$$\frac{e_1 \Rightarrow n_1 \qquad e_2 \Rightarrow n_2}{\mathtt{plus}(e_1,e_2) \Rightarrow n_1 + n_2} \quad \text{(NS-Plus)}$$

$$\frac{}{\mathtt{num}(n) \Rightarrow n} \quad \text{(NS-Num)}$$

Figure 2.25: Natural semantics for simple arithmetic expressions

2.4.2 Natural semantics for simple arithmetic expressions

We consider how to specify syntax and natural semantics of our simple arithmetic language.

Abstract syntax. Natural semantics assigns meaning to program phrases expressed using abstract syntax. Natural semantics has a notion of *semantic values* that is typically distinct from the syntactic values in abstract syntax trees. Figure 2.24 defines the syntactic sets we will use to define a natural semantics for simple arithmetic expressions.

Semantics. The rules in Figure 2.25 specifies a relation $\Rightarrow \subseteq \mathit{Expr}^{\mathrm{NS}} \times \mathit{Val}^{\mathrm{NS}}$. We say a term $e \in \mathit{Expr}^{\mathrm{NS}}$ evaluates to a value $v \in \mathit{Val}^{\mathrm{NS}}$ when we can use the rules to construct an upwardly branching derivation tree whose root (conclusion) is $e \Rightarrow v$, and whose leaves are simple rules with satisfied side-conditions. For example, the following derivation tree is a proof that adding $1,2,3$, and 4 gives 10:

$$\frac{\dfrac{\dfrac{}{\mathtt{num}(1) \Rightarrow 1} \quad \dfrac{}{\mathtt{num}(2) \Rightarrow 2}}{\mathtt{plus}(\mathtt{num}(1),\mathtt{num}(2)) \Rightarrow 3} \qquad \dfrac{\dfrac{}{\mathtt{num}(3) \Rightarrow 3} \quad \dfrac{}{\mathtt{num}(4) \Rightarrow 4}}{\mathtt{plus}(\mathtt{num}(3),\mathtt{num}(4)) \Rightarrow 7}}{\mathtt{plus}(\mathtt{plus}(\mathtt{num}(1),\mathtt{num}(2)),\mathtt{plus}(\mathtt{num}(3),\mathtt{num}(4))) \Rightarrow 10}$$

Natural semantics typically leave open the order of evaluation for premises. Thus, the rule (NS-Plus) evaluates e_1 and e_2 in either left-to-right or right-to-left order. Thus, natural semantics is more well-suited for languages without interleaving.

Unlike SOS, it is less straightforward to express interleaving using natural semantics. For some natural semantics, interleaving can also be modelled by "shuffling" the effects generated by interleaving evaluations after the fact [Mit94, Mos04]. In denotational semantics, *power-domains* [Plo76] and so-called *resumptions* [Mil75] are used to this avail. Nakata and Uustalu [NU10, Uus13] show how these techniques are adaptable to coinductive natural semantics. This thesis focuses on languages with limited non-determinism.

2.4.3 Comparison of small-step and big-step semantics

In spite of interleaving being more straightforwardly expressed in terms of small-step semantics, big-step semantics has other advantages:

- *Conciseness:* natural semantics without abrupt termination or divergence are typically more concise than their small-step counterparts. For example, Table 2.2 compares the small-step semantics in Figures 2.2 and 2.16 to the big-step semantics in Figure 2.25. In the table, the '+2' for small-step is counting the rules for the transitive closure.

  Big-step semantics is more concise for specifying simple arithmetic expressions, but the picture changes dramatically in the presence of divergence and abrupt termination, as we shall see later in Sections 2.4.4 and 2.4.5.

- *Interpretive overhead:* naive small-step interpreters that implement semantics like our SOS semantics for simple arithmetic expressions (Figure 2.2) have a considerable interpretive overhead compared to big-step interpreters implementing semantics like the one in Figure 2.25. Danvy and Nielsen's work on refocusing [DN04] addresses exactly this shortcoming of small-step semantics, and shows how to deforest small-step interpreters for reduction semantics into corresponding big-step evaluators. They also quantify the speed-up resulting from refocusing, and conclude that refocused evaluation is always at least as efficient as naive small-step evaluation strategies, and often much more efficient. Bach Poulsen and Mosses [BPM14c] illustrates the similar observation that evaluating Horn-clauses in Prolog (the well-known logic programming language – for an introduction see, e.g., [BBS06]) corresponding to big-step MSOS rules have a similarly low interpretive overhead, compared to evaluating Horn-clauses in Prolog that correspond to small-step MSOS rules.

- *Pragmatic use in proofs:* being more abstract than small-step rules, certain kinds of proofs are traditionally proven using a big-step semantics, such as compiler correctness proofs [LG09, NK14, HW07, BH15], while others are more natural to model and prove using small-step semantics, such as bisimulation for languages with interleaving [Mil82].

2.4.4 Natural semantics for λ_{cbv}

We consider the extension of our simple arithmetic expressions language with the call-by-value λ-calculus, λ_{cbv}.

	Small-step	**Big-step**
Number of rules	3+2	2
Number of premises	2+2	2

Table 2.2: Comparison of conciseness of small-step and big-step semantics for simple arithmetic expressions

$$\begin{array}{rlr}
\mathit{Expr}^{\mathrm{NS}} \ni e ::= & \ldots \mid \lambda x.e \mid e\ e \mid x & \text{Expressions} \\
\mathit{Val}^{\mathrm{NS}} \ni v ::= & \ldots \mid \langle x, e, \rho \rangle & \text{Values} \\
x, y \in \mathit{Var} \triangleq & \{\mathrm{x}, \mathrm{y}, \ldots\} & \text{Variables} \\
\rho \in \mathit{Env} \triangleq & \mathit{Var} \xrightarrow{\mathrm{fin}} \mathit{Val}^{\mathrm{NS}} & \text{Environments}
\end{array}$$

Figure 2.26: Abstract syntax and semantic values for λ_{cbv} (extends Figure 2.24)

Abstract syntax. Figure 2.26 summarises the grammar for expressions and values. As with simple arithmetic expressions, the syntactic sets $\mathit{Expr}^{\mathrm{NS}}$ and $\mathit{Val}^{\mathrm{NS}}$ are distinct.

Semantics. The natural semantics for the newly introduced terms is specified in Figure 2.27. Like in SOS, introducing the new constructs changes the structure of judgments, which now incorporate an environment. The rules for the simple arithmetic fragment of λ_{cbv} have been modified in Figure 2.27 to reflect this change.

The rules in Figure 2.27 are inductively defined. Thus, the rules only give semantics for the set of programs that terminate.

Divergence. Cousot and Cousot [CC92] consider how to give and relate specifications at different levels of abstraction. Their approach to formalising diverging terms is to introduce a special $\perp$ value to denote divergence, and add rules for all constructs that propagate this value if it arises. Here, we follow Leroy and Grall [LG09] and give a separately defined relation for divergence, which corresponds to the Cousots' approach. Figure 2.28 specifies such a coinductive divergence relation, which depends on the ordinary inductive natural semantics relation for converging sub-computations.

While the inductive rules in Figure 2.27 implement arbitrary order of evaluation, the divergence rules in Figure 2.28 freezes the evaluation order for diverging computations to left-to-right order of evaluation. If we were to permit arbitrary order of evaluation, extra rules are needed for cases such as the expression $\texttt{plus}(e_1, e_2)$ where e_2 diverges before we do any evaluation of e_1. Following Leroy and Grall [LG09], we can prove that the coinductive divergence predicate describes the same set of di-

$$\frac{\rho \vdash e_1 \Rightarrow n_1 \qquad \rho \vdash e_2 \Rightarrow n_2}{\rho \vdash \texttt{plus}(e_1, e_2) \Rightarrow n_1 + n_2} \quad (\text{NS-}\lambda_{\text{cbv}}\text{-Plus})$$

$$\frac{}{\rho \vdash \texttt{num}(n) \Rightarrow n} \quad (\text{NS-}\lambda_{\text{cbv}}\text{-Num})$$

$$\frac{}{\rho \vdash \lambda x.e \Rightarrow \langle x, e, \rho \rangle} \quad (\text{NS-}\lambda_{\text{cbv}}\text{-Lam})$$

$$\frac{\rho \vdash e_1 \Rightarrow \langle x, e, \rho' \rangle \qquad \rho \vdash e_2 \Rightarrow v_2 \qquad \rho'[x \mapsto v_2] \vdash e \Rightarrow v}{\rho \vdash e_1\ e_2 \Rightarrow v} \quad (\text{NS-}\lambda_{\text{cbv}}\text{-App})$$

$$\frac{x \in \text{dom}(\rho)}{\rho \vdash x \Rightarrow \rho(x)} \quad (\text{NS-}\lambda_{\text{cbv}}\text{-Var})$$

Figure 2.27: Natural semantics for λ_{cbv}

$$\frac{\rho \vdash e_1 \overset{\infty}{\Rightarrow}}{\rho \vdash \texttt{plus}(e_1, e_2) \overset{\infty}{\Rightarrow}} \quad (\text{NS-}\lambda_{\text{cbv}}\text{-}\infty\text{-Plus1})$$

$$\frac{\rho \vdash e_1 \Rightarrow n \qquad \rho \vdash e_2 \overset{\infty}{\Rightarrow}}{\rho \vdash \texttt{plus}(e_1, e_2) \overset{\infty}{\Rightarrow}} \quad (\text{NS-}\lambda_{\text{cbv}}\text{-}\infty\text{-Plus2})$$

$$\frac{\rho \vdash e_1 \overset{\infty}{\Rightarrow}}{\rho \vdash e_1\ e_2 \overset{\infty}{\Rightarrow}} \quad (\text{NS-}\lambda_{\text{cbv}}\text{-}\infty\text{-App1})$$

$$\frac{\rho \vdash e_1 \Rightarrow \langle x, e, \rho' \rangle \qquad \rho \vdash e_2 \overset{\infty}{\Rightarrow}}{\rho \vdash e_1\ e_2 \overset{\infty}{\Rightarrow}} \quad (\text{NS-}\lambda_{\text{cbv}}\text{-}\infty\text{-App2})$$

$$\frac{\rho \vdash e_1 \Rightarrow \langle x, e, \rho' \rangle \qquad \rho \vdash e_2 \Rightarrow v_2 \qquad \rho'[x \mapsto v_2] \vdash e \overset{\infty}{\Rightarrow}}{\rho \vdash e_1\ e_2 \overset{\infty}{\Rightarrow}} \quad (\text{NS-}\lambda_{\text{cbv}}\text{-}\infty\text{-App3})$$

Figure 2.28: Natural semantics divergence predicate for λ_{cbv}

verging computations as the small-step $\overset{\infty}{\rightarrow}$ predicate for the SOS rules that we saw in Figures 2.6 and 2.8.

Proposition 2.7 ($\overset{\infty}{\Rightarrow}$ corresponds to $\overset{\infty}{\rightarrow}$ for λ_{cbv}) The natural semantics and SOS for λ_{cbv} describe the same set of diverging terms; i.e.:

$$\mathsf{source}(e) \implies \mathsf{rsource}(\rho) \implies \left(\mathsf{rnsify}(\rho) \vdash \mathsf{nsify}(e) \overset{\infty}{\Rightarrow} \iff \rho \vdash e \overset{\infty}{\rightarrow}\right)$$

where $\mathsf{source}(e), \mathsf{rsource}(\rho)$ checks that expressions $e \in \mathit{Expr}$ and $\rho \in \mathit{Env}$ have corresponding natural semantic source terms and environments; and $\mathsf{nsify} \in \mathit{Expr} \rightarrow \mathit{Expr}^{\text{NS}}$ and $\mathsf{rnsify} \in \mathit{Env} \rightarrow \mathit{Env}^{\text{NS}}$ translates SOS syntax to natural semantics syntax.

Proof. The proof follows the structure of Leroy and Grall's [LG09, Theorem 11], and relies on the fact that $\rightarrow$ is deterministic. The full proof is in Appendix A. □

We can also use the coinductively defined divergence predicate to prove that the well-known self-applicative ω expression diverges.

Proposition 2.8 (ω diverges using natural semantics) The expression ω diverges, i.e.:

$$\rho \vdash \omega \overset{\infty}{\Rightarrow}$$

Proof. The proof is by guarded coinduction. By a single application of (NS-λ_{cbv}-∞-App3) we get the goal:

$$\rho[x \mapsto \langle x, x\, x, \rho\rangle] \vdash x\, x \overset{\infty}{\Rightarrow} \tag{Goal}$$

Using this as our coinduction hypothesis, and applying (NS-λ_{cbv}-∞-App3) gives us a new goal that is identical to (Goal) above. The new goal follows by applying the coinduction hypothesis. □

The big-step proof is more straightforward than the small-step proof by guarded coinduction: big-step abstracts from intermediate small-steps, whereby the structure of closures is irrelevant, and the coinduction hypothesis can be applied directly.

The natural semantics for λ_{cbv} with divergence is no longer more concise than SOS. The rules for $\Rightarrow$ and $\overset{\infty}{\Rightarrow}$ contain duplication: for example, the premise $\rho \vdash e_1 \Rightarrow n_1$ describes the same computation in the same context in both rule (NS-λ_{cbv}-Plus) and rule (NS-λ_{cbv}-∞-Plus2). There is similar duplication occurring in rules (NS-λ_{cbv}-App), (NS-λ_{cbv}-∞-App2), and (NS-λ_{cbv}-∞-App3). Charguéraud [Cha13] calls this the *duplication problem* with natural semantics, and proposes a style of natural semantics that alleviates the duplication. We recall Charguéraud's *pretty-big-step* style in Section 2.5. Table 2.3 compares the conciseness and duplication in the natural semantics and SOS specifications of λ_{cbv}.

An alternative means of encoding divergence in natural semantics is to use trace-based semantics. Leroy and Grall [LG09, Sect. 6.3] give trace-based big-step rules,

	Small-step	**Big-step (without $\overset{\infty}{\Rightarrow}$)**	**Big-step (with $\overset{\infty}{\Rightarrow}$)**
Number of rules	9+3	5	10
Number of premises	5+4	5	14
Duplicate premises	0	0	4

Table 2.3: Comparison of conciseness of small-step and big-step semantics for λ_{cbv}

where separate relations are given for terminating and diverging traces. This relation suffers from a similar duplication problem as the problem recalled in this section. Charguéraud proposes trace-based pretty-big-step semantics, which alleviates the duplication. Nakata and Uustalu [NU09] present an alternative solution that relies on a single set of coinductively defined rules that subsumes both converging and diverging computations and their traces, drawing inspiration from the partiality monad in type theory [Cap05]. Nakata and Uustalu illustrate the use of traces on a simple imperative while-language, but the technique scales to other programming language disciplines, such as call-by-value λ-calculus as investigated and demonstrated by Danielsson [Dan12] who uses the partiality monad to give a functional operational semantics for the call-by-value λ-calculus using mixed recursive/co-recursive functions in Agda.

2.4.5 Natural semantics for $\lambda^{\bullet}_{\text{cbv}}$

Divergence is not the only language feature that gives rise to duplication in natural semantics. We recall the traditional approach to specifying abrupt termination in natural semantics, as used in, e.g., The Definition of Standard ML [MTHM97].

Abstract syntax. Figure 2.19 extends the abstract syntax for λ_{cbv} with constructs for throwing and handling exceptions. Here, the `catch` construct differs slightly from the `catch` construct we considered in connection with SOS: the expression $\texttt{catch}(e_1,x,e_2)$ explicitly indicates that e_2 may have a free variable x which will be bound by the value contained in an exception, if one occurs. The motivation for having the `catch` construct deal with binding is that, following the convention in natural semantics, values and expressions are distinct syntactic sorts, so an application such as $e_2\ v$ is not a valid expression in $\mathit{Expr}^{\text{NS}}$.

In order to distinguish abrupt and ordinary termination, we introduce a new syntactic set *Outcome*, which comprises exceptions and values.

Semantics. Figure 2.30 summarises the rules for the new constructs, which extend the set of rules from Figure 2.27 by lifting their signature to have *Outcome* as codomain, rather than *Val*. The signature of the resulting relation is $\Rightarrow\ \subseteq \mathit{Env} \times \mathit{Expr}^{\text{NS}} \times \mathit{Outcome}$.

In addition to the rules in Figure 2.30, we also need rules for propagating abrupt termination, as well as rules for the coinductive divergence predicate for the extended

$$\begin{array}{ll} \mathit{Expr}^{\mathrm{NS}} \ni e ::= \ \ldots \mid \texttt{throw}(e) \mid \texttt{catch}(e,x,e) & \text{Expressions} \\ \mathit{Outcome} \ni o ::= v \mid \mathrm{EXC}(v) & \end{array}$$

Figure 2.29: Abstract syntax and semantic values for $\lambda^{\bullet}_{\text{cbv}}$ (extends Figure 2.26)

$$\frac{\rho \vdash e \Rightarrow v}{\rho \vdash \texttt{throw}(e) \Rightarrow \mathrm{EXC}(v)} \qquad (\text{NS-}\lambda^{\bullet}_{\text{cbv}}\text{-Throw})$$

$$\frac{\rho \vdash e_1 \Rightarrow v_1}{\rho \vdash \texttt{catch}(e_1,x,e_2) \Rightarrow v_1} \qquad (\text{NS-}\lambda^{\bullet}_{\text{cbv}}\text{-CatchV})$$

$$\frac{\rho \vdash e_1 \Rightarrow \mathrm{EXC}(v) \qquad \rho[x \mapsto v] \vdash e_2 \Rightarrow o}{\rho \vdash \texttt{catch}(e_1,x,e_2) \Rightarrow o} \qquad (\text{NS-}\lambda^{\bullet}_{\text{cbv}}\text{-CatchE})$$

Figure 2.30: Natural semantics for $\lambda^{\bullet}_{\text{cbv}}$ (extends Figure 2.27)

language. Figure 2.31 specifies the exception propagation rules, and Figure 2.32 extends the previously defined rules for the divergence predicate to give the semantics of diverging terms for the newly introduced constructs.

We can observe that abrupt termination further aggravates the duplication problem with natural semantics. Standard ML side-steps the issue with exception propagation rules by the "exception convention" [MTHM97, p. 47], which implicitly generates exception propagation rules for all constructs except the `catch` construct. Although this convention is intuitively clear, the rules typically need spelling out in full detail in proof assistants such as Coq or Agda, and they clutter the rule induction hypothesis used to prove properties about relations.

Extending the natural semantics with stores and references to give a natural semantics for the $\lambda^{\bullet}_{\text{cbv+}\texttt{ref}}$ corresponding to the SOS in Section 2.2.6 is straightforward. The traditional approach, used in, e.g., The Definition of Standard ML [MTHM97], is to thread a store through all premises in existing rules from left-to-right (although The Definition leaves the threading of such premises implicit in rules – this is known as the "state convention" [MTHM97, p. 46]).

The duplication problem that natural semantics for languages with divergence and abrupt termination suffers from is alleviated by pretty-big-step semantics.

2.5 Pretty-big-step semantics

Charguéraud's [Cha13] proposed *pretty-big-step* semantics factors big-step rules into rules with at most two premises. This allows divergence and abrupt termination to be propagated with less duplication.

$$\frac{\rho \vdash e_1 \Rightarrow \texttt{EXC}(v)}{\rho \vdash \texttt{plus}(e_1, e_2) \Rightarrow \texttt{EXC}(v)} \quad \text{(NS-}\lambda^{\bullet}_{\text{cbv}}\text{-Exc-Plus1)}$$

$$\frac{\rho \vdash e_1 \Rightarrow n \qquad \rho \vdash e_2 \Rightarrow \texttt{EXC}(v)}{\rho \vdash \texttt{plus}(e_1, e_2) \Rightarrow \texttt{EXC}(v)} \quad \text{(NS-}\lambda^{\bullet}_{\text{cbv}}\text{-Exc-Plus2)}$$

$$\frac{\rho \vdash e_1 \Rightarrow \texttt{EXC}(v)}{\rho \vdash e_1\ e_2 \Rightarrow \texttt{EXC}(v)} \quad \text{(NS-}\lambda^{\bullet}_{\text{cbv}}\text{-Exc-App1)}$$

$$\frac{\rho \vdash e_1 \Rightarrow \langle x, e, \rho' \rangle \qquad \rho \vdash e_2 \Rightarrow \texttt{EXC}(v)}{\rho \vdash e_1\ e_2 \Rightarrow \texttt{EXC}(v)} \quad \text{(NS-}\lambda^{\bullet}_{\text{cbv}}\text{-Exc-App2)}$$

$$\frac{\rho \vdash e_1 \Rightarrow \langle x, e, \rho' \rangle \qquad \rho \vdash e_2 \Rightarrow v_2 \qquad \rho'[x \mapsto v_2] \vdash e \Rightarrow \texttt{EXC}(v)}{\rho \vdash e_1\ e_2 \Rightarrow \texttt{EXC}(v)} \quad \text{(NS-}\lambda^{\bullet}_{\text{cbv}}\text{-Exc-App3)}$$

$$\frac{\rho \vdash e \Rightarrow \texttt{EXC}(v)}{\rho \vdash \texttt{throw}(e) \Rightarrow \texttt{EXC}(v)} \quad \text{(NS-}\lambda^{\bullet}_{\text{cbv}}\text{-Exc-Throw)}$$

Figure 2.31: Natural semantic abrupt termination rules for $\lambda^{\bullet}_{\text{cbv}}$ (extends Figure 2.27)

$$\frac{\rho \vdash e \overset{\infty}{\Rightarrow}}{\rho \vdash \texttt{throw}(e) \overset{\infty}{\Rightarrow}} \quad \text{(NS-}\lambda^{\bullet}_{\text{cbv}}\text{-}\infty\text{-Throw)}$$

$$\frac{\rho \vdash e_1 \overset{\infty}{\Rightarrow}}{\rho \vdash \texttt{catch}(e_1, x, e_2) \overset{\infty}{\Rightarrow}} \quad \text{(NS-}\lambda^{\bullet}_{\text{cbv}}\text{-}\infty\text{-CatchV)}$$

$$\frac{\rho \vdash e_1 \Rightarrow \texttt{EXC}(v) \qquad \rho[x \mapsto v] \vdash e_2 \overset{\infty}{\Rightarrow}}{\rho \vdash \texttt{catch}(e_1, x, e_2) \overset{\infty}{\Rightarrow}} \quad \text{(NS-}\lambda^{\bullet}_{\text{cbv}}\text{-}\infty\text{-CatchE)}$$

Figure 2.32: Natural semantic divergence predicate for $\lambda^{\bullet}_{\text{cbv}}$ (extends Figure 2.28)

2.5.1 Pretty-big-step semantics for simple arithmetic expressions

We consider how to give pretty-big-step semantics for simple arithmetic expressions.

Abstract syntax. Pretty-big-step semantics introduces a new notion of intermediate expressions that are not valid input programs, but that record semantic information used in derivation rules. Intermediate expressions serve several purposes:

- they permit matching against the outcome of sub-terms such that evaluation only continues if abrupt termination or divergence does not occur; and
- they ensure derivation trees constructed using pretty-big-step rules are productive.

$$\begin{array}{rlr}
e \in \mathit{Expr}^{\mathrm{NS}} & & \text{Expressions (as in Figure 2.24)} \\
v \in \mathit{Val}^{\mathrm{NS}} & & \text{Values (as in Figure 2.24)} \\
\mathit{IntmExpr} \ni E & ::= e \mid \texttt{plus1}(o,e) \mid \texttt{plus2}(v,o) & \text{Intermediate expressions} \\
\mathit{Outcome}^{\mathrm{PBS}} \ni o & ::= v & \text{Outcomes}
\end{array}$$

Figure 2.33: Abstract syntax, values, and outcomes for simple arithmetic expressions

$$\frac{e_1 \Downarrow o_1 \qquad \texttt{plus1}(o_1,e_2) \Downarrow o}{\texttt{plus}(e_1,e_2) \Downarrow o} \quad \text{(PBS-Plus)}$$

$$\frac{e_2 \Downarrow o_2 \qquad \texttt{plus2}(o_1,o_2) \Downarrow o}{\texttt{plus1}(o_1,e_2) \Downarrow o} \quad \text{(PBS-Plus1)}$$

$$\frac{}{\texttt{plus2}(n_1,n_2) \Downarrow n_1 + n_2} \quad \text{(PBS-Plus2)}$$

$$\frac{}{\texttt{num}(v) \Downarrow n} \quad \text{(PBS-Num)}$$

Figure 2.34: Pretty-big-step semantics for simple arithmetic expressions

The second of these roles is important when we consider the coinductive interpretation of rules. We illustrate why this is the case shortly. First, we introduce pretty-big-step rules for simple arithmetic expressions.

Figure 2.33 summarises the abstract syntax of expressions, intermediate expressions, and outcomes. For simple arithmetic expressions, the only notion of outcome is values. When we consider its extension with the λ-calculus in Section 2.5.2, the set of outcomes will be augmented with divergence.

Semantics. Figure 2.34 summarises the rules for the pretty-big-step relation $\Downarrow \in \mathit{IntmExpr} \times \mathit{Outcome}^{\mathrm{PBS}}$. Each rule has at most two premises that do evaluation. In the rules (PBS-Plus) and (PBS-Plus1), the first premise fully evaluates a sub-term to its outcome, and the second premise constructs an intermediate expression, which continues evaluation if the first sub-expression did not abruptly terminate or diverge.

A feature of pretty-big-step rules is that they have a *dual* interpretation: they have both an inductive and a coinductive interpretation, and permits reasoning about both converging and diverging computations using the same set of rules. Consider the following reflexive rule for values:

$$\frac{}{v \Downarrow v} \quad \text{(PBS-ReflV)}$$

$e \in \mathit{Expr}^{\mathrm{NS}}$	Expressions (as in Figure 2.26)
$v \in \mathit{Val}^{\mathrm{NS}}$	Values (as in Figure 2.26)
$\mathit{IntmExpr} \ni E ::= ... \mid \mathtt{app1}(o,e) \mid \mathtt{app2}(v,o)$	Semantic expressions
$\mathit{Outcome}^{\mathrm{PBS}} \ni o ::= v \mid \mathrm{DIV}$	Outcomes

Figure 2.35: Abstract syntax, values, and outcomes for λ_{cbv} (extending Figure 2.33)

If we add this rule to the coinductive interpretation of the relation for simple arithmetic expressions, we would be able to construct an infinite derivation tree for the expression $\mathtt{plus}(\mathtt{num}(1),\mathtt{num}(2))$:

$$\dfrac{\mathtt{num}(1) \Downarrow 1 \qquad \dfrac{1 \Downarrow 1 \qquad \dfrac{\vdots}{\mathtt{plus}(1,\mathtt{num}(2)) \Downarrow 42}}{\mathtt{plus}(1,\mathtt{num}(2)) \Downarrow 42}}{\mathtt{plus}(\mathtt{num}(1),\mathtt{num}(2)) \Downarrow 42}$$

In other words, we could prove that adding 1 and 2 diverges. The use of intermediate expressions in the pretty-big-step rules in Figure 2.34 avoids such issues.

Comparing with the natural semantics in Section 2.4.2, pretty-big-step uses more rules and premises for semantics without divergence or abrupt termination.

2.5.2 Pretty-big-step semantics for λ_{cbv}

Extending simple arithmetic expressions with the call-by-value λ-calculus introduces the possibility of divergence.

Abstract syntax. The abstract syntax is summarised in Figure 2.35, which gives two new intermediate expression constructs for application, and augments the notion of outcome with a new term DIV for indicating divergence.

Semantics. Like natural semantics and SOS, the extension involves modifying rules to propagate the environment. Figure 2.36 summarises the rules for ordinary evaluation by means of the pretty-big-step relation $\Downarrow \in \mathit{Env} \times \mathit{IntmExpr} \times \mathit{Outcome}^{\mathrm{PBS}}$. In addition to these rules, we follow Charguéraud and introduce rules for propagating divergence. The rules are summarised in Figure 2.37.

As mentioned, pretty-big-step rules have a *dual* interpretation: they have both an inductive and a coinductive interpretation, and permits reasoning about both converging and diverging computations using the same set of rules.

$$\frac{\rho \vdash e_1 \Downarrow o_1 \qquad \rho \vdash \mathtt{plus1}(o_1, e_2) \Downarrow o}{\rho \vdash \mathtt{plus}(e_1, e_2) \Downarrow o} \qquad (\text{PBS-}\lambda_{\text{cbv}}\text{-Plus})$$

$$\frac{\rho \vdash e_2 \Downarrow o_2 \qquad \rho \vdash \mathtt{plus2}(v_1, o_2) \Downarrow o}{\rho \vdash \mathtt{plus1}(v_1, e_2) \Downarrow o} \qquad (\text{PBS-}\lambda_{\text{cbv}}\text{-Plus1})$$

$$\frac{}{\rho \vdash \mathtt{plus2}(n_1, n_2) \Downarrow n_1 + n_2} \qquad (\text{PBS-}\lambda_{\text{cbv}}\text{-Plus2})$$

$$\frac{}{\rho \vdash \lambda x.e \Downarrow \langle x, e, \rho \rangle} \qquad (\text{PBS-}\lambda_{\text{cbv}}\text{-Lam})$$

$$\frac{\rho \vdash e_1 \Downarrow o_1 \qquad \rho \vdash \mathsf{app1}(o_1, e_2) \Downarrow o}{\rho \vdash e_1\ e_2 \Downarrow o} \qquad (\text{PBS-}\lambda_{\text{cbv}}\text{-App})$$

$$\frac{\rho \vdash e_2 \Downarrow o_2 \qquad \rho \vdash \mathsf{app2}(v_1, o_2) \Downarrow o}{\rho \vdash \mathsf{app1}(v_1, e_2) \Downarrow o} \qquad (\text{PBS-}\lambda_{\text{cbv}}\text{-App1})$$

$$\frac{\rho'[x \mapsto v_2] \vdash e \Downarrow o}{\rho \vdash \mathsf{app2}(\langle x, e, \rho' \rangle, v_2) \Downarrow o} \qquad (\text{PBS-}\lambda_{\text{cbv}}\text{-App2})$$

Figure 2.36: Pretty-big-step semantics for λ_{cbv}

$$\frac{}{\rho \vdash \mathtt{plus1}(\textsc{div}, e_2) \Downarrow \textsc{div}} \qquad (\text{PBS-}\lambda_{\text{cbv}}\text{-Div-Plus1})$$

$$\frac{}{\rho \vdash \mathtt{plus2}(n_1, \textsc{div}) \Downarrow \textsc{div}} \qquad (\text{PBS-}\lambda_{\text{cbv}}\text{-Div-Plus2})$$

$$\frac{}{\rho \vdash \mathsf{app1}(\textsc{div}, e_2) \Downarrow \textsc{div}} \qquad (\text{PBS-}\lambda_{\text{cbv}}\text{-Div-App1})$$

$$\frac{}{\rho \vdash \mathsf{app2}(\langle x, e, \rho' \rangle, \textsc{div}) \Downarrow \textsc{div}} \qquad (\text{PBS-}\lambda_{\text{cbv}}\text{-Div-App2})$$

Figure 2.37: Pretty-big-step divergence rules for λ_{cbv}

Divergence. Using the rules in Figures 2.36 and 2.37, we cannot construct a finite derivation tree that produces DIV as outcome. It follows that, if we can prove (using coinduction) that there is a derivation tree whose outcome is DIV, that derivation tree must be a diverging computation. We recall some facts about the relationship between the inductive and coinductive interpretation of pretty-big-step rules, following Charguéraud [Cha13]. We use $\Downarrow^{co}$ to refer to the coinductive interpretation of the specified pretty-big-step relation, and $\Downarrow$ to refer to the inductive one.

Proposition 2.9 (λ_{cbv} does not diverge inductively) For any finite derivation tree, it holds that the outcome of evaluation is not DIV; i.e.:

$$\rho \vdash e \Downarrow o \implies o \neq \text{DIV}$$

Proof. Trivial, since $\Downarrow^{co}$ is just the coinductive interpretation of $\Downarrow$. □

Proposition 2.10 (Coinductive subsumes inductive) Any finite derivation tree is contained in the coinductive interpretation, i.e.:

$$\rho \vdash e \Downarrow o \implies \rho \vdash e \Downarrow^{co} o$$

Proof. The proof is by straightforward rule induction. □

A consequence of Propositions 2.9 and 2.10 is that, if it holds that $\rho \vdash e \Downarrow^{co}$ DIV, then we know that the expression e diverges in ρ. On the other hand, if $\rho \vdash e \Downarrow^{co} v$, it is either the case that e is a convergent computation that produces a value v, or that it diverges. This intuition is captured formally by Proposition 2.12.

Lemma 2.11

$$\rho \vdash e \Downarrow^{co} o \implies \neg(\rho \vdash e \Downarrow o) \implies \rho \vdash e \Downarrow^{co} \text{DIV}$$

Proof. Using the goal as our coinduction hypothesis, the proof proceeds by inversion on the first hypothesis. Each case that does not follow trivially (e.g., because the second hypothesis trivially holds) follows by invoking the law of excluded middle on whether or not each sub-derivation diverges or not. We consider the case for rule (PBS-λ_{cbv}-App1); the remaining cases follow a similar pattern. In Chapter 4 of this thesis, we prove that this lemma holds for any semantics whose rules match a set of rule schemas, and consider a variant of the pretty-big-step semantics for λ_{cbv} which adheres to this schema.

Case (PBS-λ_{cbv}-Plus1) From inversion and the rule we have:

$$\rho \vdash e_1 \Downarrow^{co} o_1 \tag{H1}$$

$$\rho \vdash \texttt{plus1}(o_1, e_2) \Downarrow^{co} o \tag{H2}$$

$$\neg(\rho \vdash \texttt{plus}(e_1, e_2) \Downarrow o) \tag{H3}$$

The coinduction hypothesis is:

$$\forall \rho\ e\ o.\ \rho \vdash e \Downarrow^{co} o \implies \neg(\rho \vdash e \Downarrow o) \implies \rho \vdash e \Downarrow^{co} \textsc{div} \tag{CIH}$$

The goal is:

$$\rho \vdash \texttt{plus}(e_1, e_2) \Downarrow^{co} \textsc{div} \tag{Goal}$$

We invoke the law of excluded middle on $\rho \vdash e_1 \Downarrow o_1$. We consider the negative case first:

Subcase $(\neg P)$ We apply the rule (PBS-λ_{cbv}-Plus). The first premise of the rule follows from the coinduction hypothesis, $\neg P$, and (H1).

The second premise of the rule trivially follows from the (PBS-λ_{cbv}-Div-Plus1) rule.

Subcase (P) We reason by the law of excluded middle on $\rho \vdash \texttt{plus1}(o_1, e_2) \Downarrow o$. We consider the negative case first

Subsubcase $(\neg P')$ The goal follows by applying the rule (PBS-λ_{cbv}-Plus), (H1), the coinduction hypothesis, and $\neg P'$.

Subsubcase (P') The hypothesis (H3) becomes a contradition, since P and P' imply that $\texttt{plus}(e_1, e_2)$ converges.

□

Proposition 2.12 (Coinductive implies termination or divergence) For any derivation tree in the coinductive interpretation, there is either a corresponding finite derivation tree in the inductive interpretation, or the derivation tree is infinite and we can construct a corresponding infinite tree whose outcome is DIV:

$$\rho \vdash e \Downarrow^{co} o \implies \rho \vdash e \Downarrow o \lor \rho \vdash e \Downarrow^{co} \textsc{div}$$

Proof. We reason by the law of excluded middle on $\rho \vdash e \Downarrow o$. Calling this proposition P, there are two cases to consider.

Case (P) The left disjunctive goal holds trivially.

Case $(\neg P)$ By Lemma 2.11, we get that $\rho \vdash e \Downarrow^{co} \textsc{div}$, from which the right disjunctive goal follows.

□

2.5.3 Pretty-big-step semantics for $\lambda^{\bullet}_{\text{cbv}}$

Extending pretty-big-step semantics with abrupt termination does not introduce duplication in rules.

$e \in \mathit{Expr}^{\mathrm{NS}}$	Expressions (as in Figure 2.29)
$v \in \mathit{Val}^{\mathrm{NS}}$	Values (as in Figure 2.29)
$\mathit{IntmExpr} \ni E ::= ... \mid \texttt{throw1}(o) \mid \texttt{catch1}(o,x,e)$	Semantic expressions
$\mathit{Outcome}^{\mathrm{PBS}} \ni o ::= \ldots \mid \mathrm{EXC}(v)$	Outcomes

Figure 2.38: Abstract syntax, values for $\lambda^{\bullet}_{\mathrm{cbv}}$ (extending Figure 2.35)

$$\frac{\rho \vdash e \Downarrow o \qquad \rho \vdash \texttt{throw1}(o) \Downarrow o'}{\rho \vdash \texttt{throw}(e) \Downarrow o'} \quad (\text{PBS-}\lambda^{\bullet}_{\mathrm{cbv}}\text{-Throw})$$

$$\frac{}{\rho \vdash \texttt{throw}(v) \Downarrow \mathrm{EXC}(v)} \quad (\text{PBS-}\lambda_{\mathrm{cbv}}\text{-Throw1})$$

$$\frac{\rho \vdash e_1 \Downarrow o_1 \qquad \rho \vdash \texttt{catch1}(o_1,x,e_2) \Downarrow o}{\rho \vdash \texttt{catch}(e_1,x,e_2) \Downarrow o} \quad (\text{PBS-}\lambda^{\bullet}_{\mathrm{cbv}}\text{-Catch})$$

$$\frac{}{\rho \vdash \texttt{catch1}(v_1,x,e_2) \Downarrow v_1} \quad (\text{PBS-}\lambda^{\bullet}_{\mathrm{cbv}}\text{-CatchV})$$

$$\frac{\rho[x \mapsto v_1] \vdash e_2 \Downarrow o_2}{\rho \vdash \texttt{catch1}(\mathrm{EXC}(v_1),x,e_2) \Downarrow o_2} \quad (\text{PBS-}\lambda^{\bullet}_{\mathrm{cbv}}\text{-CatchE})$$

Figure 2.39: Pretty-big-step semantics for $\lambda^{\bullet}_{\mathrm{cbv}}$

Abstract syntax. Figure 2.38 summarises the necessary extra intermediate expression constructors for `throw`ing and `catch`ing exceptions. The notion of outcome is also augmented with exceptions, $\mathrm{EXC}(v)$.

Semantics. Following Charguéraud [Cha13], exceptions can be propagated in the same way as divergence by generalising the divergence rules from Figure 2.37 into so-called 'abort' rules, and introducing an *abort* predicate for deciding whether evaluation should continue. Figure 2.39 summarises the rules for ordinary evaluation of `throw` and `catch`, while Figure 2.40 specifies abort rules and the *abort* predicate. The rules that rely on the *abort* predicate and the rules for the predicate itself are somewhat tedious to both read and write, but Charguéraud proposes that they can be automatically generated.

It is also possible to give pretty-big-step semantics for ML-style references. Appendix C.1 recalls a pretty-big-step semantics for $\lambda^{\bullet}_{\mathrm{cbv+\texttt{ref}}}$ reminiscent of the one considered by Charguéraud [Cha13].

Pretty-big-step semantics provides a means of minimising the duplication resulting from divergence and abrupt termination in natural semantics. However, for languages

Abort predicate.

$$\frac{}{\mathit{abort}(\textsc{div})} \quad \text{(Abort-Div)}$$

$$\frac{}{\mathit{abort}(\textsc{exc}(v))} \quad \text{(Abort-Exc)}$$

Abort rules.

$$\frac{\mathit{abort}(o_1)}{\rho \vdash \texttt{plus1}(o_1, e_2) \Downarrow o_1} \quad \text{(PBS-}\lambda^{\bullet}_{\text{cbv}}\text{-Abort-Plus1)}$$

$$\frac{\mathit{abort}(o_2)}{\rho \vdash \texttt{plus2}(n_1, o_2) \Downarrow o_2} \quad \text{(PBS-}\lambda^{\bullet}_{\text{cbv}}\text{-Abort-Plus2)}$$

$$\frac{\mathit{abort}(o_1)}{\rho \vdash \texttt{app1}(o_1, e_2) \Downarrow o_1} \quad \text{(PBS-}\lambda^{\bullet}_{\text{cbv}}\text{-Abort-App1)}$$

$$\frac{\mathit{abort}(o_2)}{\rho \vdash \texttt{app2}(\langle x, e, \rho' \rangle, o_2) \Downarrow o_2} \quad \text{(PBS-}\lambda^{\bullet}_{\text{cbv}}\text{-Abort-App2)}$$

$$\frac{\mathit{abort}(o)}{\rho \vdash \texttt{throw1}(o) \Downarrow o} \quad \text{(PBS-}\lambda^{\bullet}_{\text{cbv}}\text{-Abort-Throw)}$$

Figure 2.40: Pretty-big-step abort rules for $\lambda^{\bullet}_{\text{cbv}}$

without divergence or abrupt termination, the pretty-big-step style uses more rules and premises. It also suffers from the same shortcomings as SOS and natural semantics: introducing a new auxiliary entity involves modifying rules in order to propagate it. This problem is avoided with big-step Modular SOS.

2.6 Big-step Modular SOS

Modular SOS has a big-step counterpart that avoids the problem with modifying rules to propagate auxiliary entities. A caveat is that the technique for specifying modular abrupt termination does not directly translate to big-step MSOS. We recall how to give big-step MSOS specifications, and why the modular abrupt termination technique does not translate.

2.6.1 Big-step MSOS for λ_{cbv}

The λ_{cbv} language does not have abrupt termination, and it is straightforward to give a big-step MSOS specification for the language.

$$\frac{e_1 \xrightarrow{\ell_1} n_1 \qquad e_2 \xrightarrow{\ell_2} n_2}{\mathtt{plus}(e_1, e_2) \xrightarrow{\ell_1 \fatsemi \ell_2} n_1 + n_2} \qquad \text{(BMSOS-}\lambda_{\text{cbv}}\text{-Plus)}$$

$$\frac{}{\mathtt{num}(n) \xrightarrow{\{-\}} n} \qquad \text{(BMSOS-}\lambda_{\text{cbv}}\text{-Num)}$$

$$\frac{}{\lambda x.e \xrightarrow{\{\mathbf{env}=\rho,-\}} \langle x, e, \rho \rangle} \qquad \text{(BMSOS-}\lambda_{\text{cbv}}\text{-Lam)}$$

$$\frac{e_1 \xrightarrow{\{\mathbf{env}=\rho,\ldots_1\}} \langle x, e, \rho' \rangle \qquad e_2 \xrightarrow{\ell} v_2 \qquad e \xrightarrow{\{\mathbf{env}=\rho'[x \mapsto v_2],\ldots_2\}} v}{e_1\ e_2 \xrightarrow{\{\mathbf{env}=\rho,\ldots_1\} \fatsemi \ell \fatsemi \{\mathbf{env}=\rho,\ldots_2\}} v} \qquad \text{(BMSOS-}\lambda_{\text{cbv}}\text{-App)}$$

$$\frac{x \in \mathrm{dom}(x)}{x \xrightarrow{\{\mathbf{env}=\rho,-\}} \rho(x)} \qquad \text{(BMSOS-}\lambda_{\text{cbv}}\text{-Var)}$$

Figure 2.41: Big-step MSOS for λ_{cbv}

Abstract syntax. The abstract syntax is the same as the one we considered in connection with natural semantics in Figure 2.26.

Semantics. The semantics for λ_{cbv} is straightforward to specify using MSOS labels instead of auxiliary entities. Auxiliary entities are propagated between big-step premises by insisting that premises that are evaluated consecutively have labels that are composable. In the case of labels for λ_{cbv}, where the only label component is environments, labels are composable when the environment components are equal. Figure 2.41 gives big-step MSOS rules for λ_{cbv}.

A difference between small-step and big-step MSOS is that label composition operations become ubiquitous in rules. This makes rules appear somewhat unfamiliar to the uninitiated. Another issue with big-step MSOS is that the technique for modular abrupt termination in small-step MSOS does not translate.

2.6.2 The problem with big-step modular abrupt termination

We illustrate the problem with translating the modular abrupt termination technique to big-step MSOS.

Abstract syntax. The natural semantic abstract syntax for big-step MSOS is identical to the one in Figure 2.29. Additionally, we introduce the syntactic set *Exc*, summarised in Figure 2.42, for indicating in labels whether an exception occurs or not.

Semantics. The rules in Figure 2.43 give a naive translation of the technique for modular abrupt termination from Section 2.3.4. Like we did with the small-step MSOS

$$Exc \ni \varepsilon ::= \textsc{ok} \mid \textsc{exc}(v)$$

Figure 2.42: Abstract syntax for $\lambda^{\bullet}_{\text{cbv}}$ (extending Figure 2.30)

$$\frac{e \xrightarrow{\{\mathbf{exc}'=\textsc{ok},\ldots\}} v}{\texttt{throw}(e) \xrightarrow{\{\mathbf{exc}'=\textsc{exc}(v),\ldots\}} \texttt{stuck}} \qquad \text{(BMSOS-Throw)}$$

$$\frac{e \xrightarrow{\{\mathbf{exc}'=\textsc{exc}(v),\ldots\}} v}{\texttt{throw}(e) \xrightarrow{\{\mathbf{exc}'=\textsc{exc}(v),\ldots\}} v} \qquad \text{(BMSOS-Throw-Exc)}$$

$$\frac{e_1 \xrightarrow{\{\mathbf{exc}'=\textsc{ok},\ldots\}} v}{\texttt{catch}(e_1,x,e_2) \xrightarrow{\{\mathbf{exc}'=\textsc{ok},\ldots\}} v} \qquad \text{(BMSOS-Catch)}$$

$$\frac{e_1 \xrightarrow{\{\mathbf{env}=\rho,\mathbf{exc}'=\textsc{exc}(v),\ldots_1\}} v_1 \qquad e_2 \xrightarrow{\{\mathbf{env}=\rho[x\mapsto v],\ldots_2\}} v_2}{\texttt{catch}(e_1,x,e_2) \xrightarrow{\{\mathbf{exc}'=\textsc{ok},\ldots_1\}\,\fatsemi\,\{\mathbf{env}=\rho,\ldots_2\}} v} \qquad \text{(BMSOS-Catch)}$$

$$\frac{e \xrightarrow{\{\mathbf{exc}'=\textsc{ok},\ldots\}} v}{\texttt{program}(e) \xrightarrow{\{\mathbf{exc}'=\textsc{ok},\ldots\}} v} \qquad \text{(BMSOS-ProgramV)}$$

$$\frac{e \xrightarrow{\{\mathbf{exc}'=\textsc{exc}(v),\ldots\}} v_0}{\texttt{program}(e) \xrightarrow{\{\mathbf{exc}'=\textsc{ok},\ldots\}} v} \qquad \text{(BMSOS-ProgramE)}$$

Figure 2.43: Big-step MSOS for $\lambda^{\bullet}_{\text{cbv}}$

for $\lambda^{\bullet}_{\text{cbv}}$ in Section 2.3.4, we have extended the product category that describes labels using the $\mathbf{exc}'$ write-only label component. Following Mosses [Mos04], the rules in Figure 2.20 are inductively defined, and thus describe only converging computations.

The semantics in Figure 2.43 does not propagate abrupt termination correctly. Consider the expression:

$$\texttt{program}(\texttt{plus}(\texttt{seq}(\texttt{throw}(0),\texttt{num}(1)),\omega))$$

where $\texttt{seq}(e_1,e_2) \triangleq (\lambda\_.e_2)\ e_1$. The big-step MSOS rule (BMSOS-λ_{cbv}-Plus) evaluates the first sub-expression of plus to the natural number 1, since:

$$\texttt{seq}(\texttt{throw}(0),\texttt{num}(1)) \xrightarrow{\{\mathbf{exc}'=\textsc{exc}(0),\text{—}\}} 1$$

The problem is that the second premise of (BMSOS-λ_{cbv}-Plus) does not inhibit further evaluation. Since exceptions are encoded as "write-only" entities (i.e., as a free

$$\mathbf{D}[\![\bullet]\!] \in \mathbb{N}$$
$$\mathbf{D}[\![\texttt{plus}(e_1, e_2)]\!] \triangleq \mathbf{D}[\![e_1]\!] + \mathbf{D}[\![e_2]\!]$$
$$\mathbf{D}[\![n]\!] \triangleq n$$

Figure 2.44: Denotational semantics of simple arithmetic expressions

monoid), evaluation continues to evaluate the second branch of the program, which is a divergent computation.[3] In other words, abrupt termination is not abruptly terminating using the naive translation to big-step of the technique for modular abrupt termination. We analyse the problem in Chapter 3, and propose *extensible transition system semantics* as a solution.

2.7 Modularity in Denotational Semantics

Previous sections described how Modular SOS provides a means of obtaining modularity in Structural Operational Semantics. In this section we recall how *monads* provide support for modularity in denotational semantics.

2.7.1 Lack of Modularity in Denotational Semantics

We first consider how to give a denotational semantics for simple arithmetic expressions, and how to extend it to λ_{cbv}.

Simple arithmetic expressions. Figure 2.44 defines a denotational semantics for simple arithmetic expressions. The semantics is given by a denotation function, which translates program expressions into natural numbers.

Extension to λ_{cbv}. We extend the denotational semantics for simple arithmetic expressions to λ_{cbv} using environments and closures. Figure 2.45 specifies the updated denotation function. The updated function differs from the one for simple arithmetic expressions in the following ways:

- *Updated signature:* the new denotation function is parameterized by an environment, and either returns an actual value (either a closure or a natural number, as defined in Figure 2.5) or a value in *Wrong* $\triangleq \{\Omega\}$.

- *Threading of environments:* denotations are now parameterised by environments.

[3]We cannot give a direct proof of divergence using the inductive big-step MSOS rules, so the program above fails to evaluate.

$$
\begin{aligned}
\mathbf{D}[\![\bullet]\!] &\in \mathit{Env} \to \mathit{Val} + \mathit{Wrong}\\[4pt]
\mathbf{D}[\![\mathtt{plus}(e_1,e_2)]\!]\rho &\triangleq \mathit{case}\ \mathbf{D}[\![e_1]\!](\rho)\ \mathit{of}\\
&\quad\ \mid n_1 \Rightarrow \mathit{case}\ \mathbf{D}[\![e_2]\!](\rho)\ \mathit{of}\\
&\qquad\qquad\ \mid n_2 \Rightarrow n_1 + n_2\\
&\qquad\qquad\ \mid \_ \Rightarrow \Omega\\
&\quad\ \mid \_ \Rightarrow \Omega\\[4pt]
\mathbf{D}[\![n]\!]\rho &\triangleq n\\[4pt]
\mathbf{D}[\![x]\!]\rho &\triangleq \mathit{case}\ x \in \mathrm{dom}(\rho)\ \mathit{of}\\
&\quad\ \mid \top \Rightarrow \rho(x)\\
&\quad\ \mid \_ \Rightarrow \Omega\\[4pt]
\mathbf{D}[\![\lambda x.e]\!]\rho &\triangleq \langle x, e, \rho\rangle\\[4pt]
\mathbf{D}[\![e_1\ e_2]\!]\rho &\triangleq \mathit{case}\ \mathbf{D}[\![e_1]\!](\rho)\ \mathit{of}\\
&\quad\ \mid \langle x, e, \rho'\rangle \Rightarrow \mathit{case}\ \mathbf{D}[\![e_2]\!](\rho)\ \mathit{of}\\
&\qquad\qquad\qquad\quad\ \mid v_2 \Rightarrow \mathbf{D}[\![e]\!](\rho'[x \mapsto v_2])\\
&\qquad\qquad\qquad\quad\ \mid \_ \Rightarrow \Omega\\
&\quad\ \mid \_ \Rightarrow \Omega
\end{aligned}
$$

Figure 2.45: Denotational semantics of λ_{cbv}

- *Checks for well-definedness:* addition is only well-defined for natural numbers, and application is only well-defined for functions. The updated denotation function uses case-distinctions to distinguish well-defined from undefined (or “wrong”) operations.

The denotation of variables uses case distinction on the propositional truth of $x \in \mathrm{dom}(\rho)$, where we use $\top$ to denote truth.

Lack of modularity. All denotations were modified in order to extend the semantics of simple arithmetic expressions to λ_{cbv}, both by introducing environment parameters for denotations, and case distinctions for “wrong”ness.

If we were to consider a similarly naive extension to $\lambda_{\text{cbv}+\mathtt{ref}}$ (i.e., introducing stores), this would require even more modifications of similar nature: denotations should be parameterised by stores, and the outcome of evaluation should also return stores. Instead of illustrating this non-modular, naive extension, we recall how monads provide a solution to the modularity problem with denotational semantics.

$$
\begin{aligned}
\mathbf{D}[\![\bullet]\!] &\in M\ \mathit{Val} \\
\mathbf{D}[\![\texttt{plus}(e_1,e_2)]\!] &\triangleq \mathbf{D}[\![e_1]\!] \gg= \\
&\qquad \Lambda v_1.\ \mathbf{D}[\![e_2]\!] \gg= \\
&\qquad\quad \Lambda v_2.\ \mathit{case}\ v_1, v_2\ \mathit{of} \\
&\qquad\qquad\quad \mid n_1, n_2 \Rightarrow n_1 + n_2 \\
\mathbf{D}[\![n]\!] &\triangleq \mathit{ret}\ n
\end{aligned}
$$

Figure 2.46: Denotational semantics using monads for simple arithmetic expressions

2.7.2 Denotational Semantics using Monads

Monads were proposed by Moggi [Mog89, Mog91] as a way of structuring denotational semantic descriptions. The central idea is to express variations under extension as a *monad*, i.e., a domain M parameterised by the domain being extended. We illustrate how to give a monadic semantics for simple arithmetic expressions.

Simple arithmetic expressions. In anticipation of future extensions, let $\mathit{Val} \triangleq \mathbb{N}$. We let the signature for the denotation function be:

$$\mathbf{D}[\![\bullet]\!] \in M\ \mathit{Val}$$

A monad M is equipped with two functions:

- $\mathit{ret} \in A \to M\ A$, sometimes called the “return” or “unit” of the monad; and
- $\gg= \ \in M\ A \to (A \to M\ B) \to M\ B$, sometimes called the “bind” of the monad, usually written in infix style, such that we write $m \gg= f$ for some $m \in M\ A$ and $f \in (A \to M\ B)$, .

These functions should satisfy the following *monad laws*, where we use Λ for meta-level functions:

- *Left identity:* $\mathit{ret}\ a \gg= f = f\ a$
- *Right identity:* $m \gg= \mathit{ret} = m$
- *Associativity:* $(m \gg= f) \gg= g = m \gg= (\Lambda a.\ f\ a \gg= g)$

We can use monads to structure the denotational semantics for simple arithmetic expressions as illustrated in Figure 2.46.

For example, we can define the variation when adding environments and the possibility of “going wrong” as a monad, by letting $M = \Lambda A.\ \mathit{Env} \to A + \mathit{Wrong}$, and defining

$$\begin{aligned}
\mathbf{D}[\![\bullet]\!] &\in M\ Val \\
\mathbf{D}[\![x]\!] &\triangleq lookup\ x \\
\mathbf{D}[\![e_1\ e_2]\!] &\triangleq \mathbf{D}[\![e_1]\!] \gg\!= \\
&\qquad \Lambda v_1.\ \mathbf{D}[\![e_2]\!] \gg\!= \\
&\qquad\quad \Lambda v_2.\ case\ v_1, v_2\ of \\
&\qquad\qquad\quad \mid \langle x, e, \rho' \rangle, v_2 \Rightarrow (inEnv\ \rho'[x \mapsto v_2]\ \mathbf{D}[\![e]\!]) \\
&\qquad\qquad\quad \mid \_, \_ \Rightarrow err \\
\mathbf{D}[\![n]\!] &\triangleq ret\ n
\end{aligned}$$

$$\begin{aligned}
lookup &\in Var \to M\ Val \\
inEnv &\in Env \to M\ Val \to M\ Val \\
err &\in M\ Val
\end{aligned}$$

Figure 2.47: Denotational semantics using monads for λ_{cbv} (extends Figure 2.46)

return and bind as:

$$\begin{aligned}
ret &\in A \to (Env \to A + Wrong) \\
ret\ a &\triangleq \Lambda\rho.\ a \\
\_ \gg\!= \_ &\in (Env \to A + Wrong) \to (A \to Env \to B + Wrong) \to Env \to B + Wrong \\
m \gg\!= f &\triangleq \Lambda\rho.\ case\ m\ \rho\ of \\
&\qquad\quad \mid \Omega \Rightarrow \Omega \\
&\qquad\quad \mid a \Rightarrow f\ a\ \rho
\end{aligned}$$

This monad ensures that environments and wrong values are correctly propagated when the monadic semantics for simple arithmetic expressions is extended to λ_{cbv}.

Extension to λ_{cbv}. Figure 2.47 summarises the monadic denotations λ_{cbv}. Denotations rely on two auxiliary functions, *inEnv* and *err*, whose implementation is left open in the figure. We could choose to define these functions naively, by using knowledge about M:

$$\begin{aligned}
lookup &\triangleq \Lambda\rho.\ case\ x \in \mathrm{dom}(x)\ of \top \Rightarrow \rho(x) \ \mid\ \_ \Rightarrow \Omega \\
inEnv &\triangleq \Lambda\rho.\ \Lambda m.\ m\ \rho \\
err &\triangleq \Lambda\rho.\ \Omega
\end{aligned}$$

However, if we extend the language further, we potentially have to modify these functions; e.g., if we extend with stores. *Monad transformers* provide a means of avoiding such issues.

Monad transformers [LHJ95, Mog90] provide a well-defined way of composing monads and "lifting" functions across composed monads. Following [LHJ95], a monad transformer is given by a domain T parameterised by a domain M, such that both M and $T\ M$ are monads. A monad transformer is equipped with a function $\mathit{lift} \in M\ A \to T\ M\ A$ that satisfies the following laws:

- $\mathit{lift}_T \circ \mathtt{unit}_M = \mathtt{unit}_{T\ M}$
- $\mathit{lift}_T\ (m \gg\!=_M f) = (\mathit{lift}_{T\ M}\ m) \gg\!=_{T\ M} (\mathit{lift}_T \circ f)$

Using monad transformers, we can add features without changing the meaning of the monad being transformed. For example, we can express the monad M for λ_{cbv} as a series of monad transformers, starting with the identity monad, i.e., $M \triangleq \Lambda A.\ A$, and extending it in two steps, using a monad transformer for possible-error (also known as the *maybe monad transformer*), $T_{\mathit{Error}} \triangleq \Lambda M.\ \Lambda A.\ (M\ A) + \mathit{Wrong}$, and a monad transformer for environments (also known as the *reader monad transformer*), $T_{\mathit{Env}} \triangleq \Lambda M.\ \Lambda A.\ \mathit{Env} \to M\ A$. The *lift* operation of monad transformers makes it possible to lift operations, such as *inEnv* and *err*, without modifying the definitions themselves as specifications are further extended. Following Liang et al. [LHJ95], such lifting can be specified and implemented using *type classes*, known from Haskell [Mar10].

Monad transformers and commutativity. Monads and monad transformers provide a flexible and powerful means of giving modular semantic specifications. For example, there are monads for modeling abrupt termination (using a *maybe* or *error* monad), stateful stores (using a *state* monad), partiality (using the *partiality* monad [Cap05]), continuations (using a *continuation* monad), interleaving (using a *resumption* monad [PG14, TA03]), etc. But with power comes responsibility: the order in which monads are applied has semantic significance.

For example, consider we want to extend a semantics with state and abrupt termination, using the maybe monad transformer T_{Error}, and a state monad transformer $T_{\mathit{State}} \triangleq \Lambda M.\ \Lambda A.\ S \to (M\ A \times S)$. If we transform a monad first with state, and then with errors, the resulting monad is:

$$T_{\mathit{Error}}\ (T_{\mathit{State}}\ (M\ A)) \triangleq (S \to (M\ A \times S) + \mathit{Wrong}$$

But what if we want the possibility of catching and handling errors? Error handlers would then have to synthesise the state in which evaluation should continue after an error is detected and handled, since errors do not record the state when an error occurs!

In contrast, if we commute the order of application for the two monad transformers:

$$T_{\mathit{State}}\ (T_{\mathit{Error}}\ (M\ A)) \triangleq (S \to ((M\ A + \mathit{Wrong}) \times S)$$

Now, erroneous outcomes are given by pairs consisting of an element in *Wrong* and a state in S. Thus, the semantics of a construct for handling errors can preserve the state s in which the error occurred. This corresponds to the semantics for `catch` considered in Section 2.3. Supporting this semantics is only possible if we compose monad transformers in the right order.

2.7.3 Discussion

Monads bring modularity to denotational semantic specifications. They have also become widely adopted for functional programming, in part due to influential papers by Wadler [Wad92, Wad95] advocating their usefulness, and Haskell's [Mar10] built-in support for monads. Monads are also becoming a subject of study for mechanised theorem proving; both as a means of representing certain semantic features in a functional style admitted by proof assistants [Dan12], but also as a means to modularity [DKSO13].

In spite of the relative success of monads and monad transformers, they do have some pitfalls, such as the commutative monad transformer problem. Comparing with small-step (M)SOS rules, supporting interleaving with monads is also more complicated. Section 2.2.3 shows how it is easy to give interleaving semantics; in contrast, support for interleaving in denotational and monadic semantics relies on more complicated constructions, such as powerdomains [Plo76] or resumptions [PG14].

Monads are inherently on *higher-order*. This makes them flexible, but also removes them from the goals of SOS [Plo81, Section 1.1], which is based on "simple" mathematics. MSOS stays closer to this goal: in spite of its theoretical underpinnings, extending a semantics with new entities relies on syntactically extending a set of rules and a product category. In spite of this difference, the basic categories used in MSOS [Mos04] can all be modeled by commutative monad transformers. This fact can be seen as both a strength and a weakness: it avoids the commutative monad transformer problem and makes the order in which one extends a semantics irrelevant, which can be seen as a strength. It can also be seen as a weakness, since it is less flexible. However, small-step MSOS has been shown to provide a simple basis for both abrupt termination [Mos04] and continuations [STM16].

In this thesis we focus mainly on relational semantics, and leave to future work an investigation of how to bring to MSOS the flexibility of monads; and vice versa, whether we can avoid the commutative monad problem by using ideas from MSOS. For example, Chapter 3 of this thesis presents a technique for giving semantics for both abrupt termination and continuations (as shown in Chapter 4) in a way that seems to correspond to a commutative monad transformer.

2.8 Reduction semantics

Reduction semantics with evaluation contexts were introduced in Felleisen's thesis as "a symbolic reasoning system for the core of expressive languages" [Fel87, p. 3]. Reduction semantics comprise:

$$E ::= [\,] \mid \texttt{plus}(E,e) \mid \texttt{plus}(n,E)$$

Figure 2.48: Evaluation context specification for simple arithmetic expressions

$$\frac{}{\texttt{plus}(n_1,n_2) \longrightarrow n_1 + n_2} \qquad \text{(RS-Plus)}$$

$$\frac{e \longrightarrow e'}{E[e] \longmapsto E[e']} \qquad \text{(RS-DCR)}$$

Figure 2.49: Contraction relation for simple arithmetic expressions

- an abstract syntax specification;
- an *evaluation context* specification that for a given term defines its set of possible decompositions into an evaluation context and a reducible expression (*redex*); and
- a contraction relation for contracting a redex into a new term.

We recall how reduction semantics are used to specify programming languages by summarising relevant fragments of Felleisen and Wright's [WF94] reduction semantics for an ML-like example language similar to the one considered in the previous sections of this chapter.

2.8.1 Reduction semantics for simple arithmetic expressions

We define the abstract syntax and notion of contraction for simple arithmetic expressions, and illustrate how these suffice to perform computations.

Abstract syntax. The abstract syntax of simple arithmetic expressions is the same as the one considered in connection with SOS; see Figure 2.1.

Notion of contraction. Figure 2.48 specifies the evaluation contexts for simple arithmetic expressions, where $[\,]$ represents the *empty context*. We use $E[e]$ to denote the term resulting from *plugging* (or *recomposing*) the term e into the hole of E.

The rules in Figure 2.49 define the notion of contraction for simple arithmetic expressions. The rules specify two relations: (RS-Plus) specifies $\longrightarrow$ for contracting `plus` redexes, and (RS-DCR) specifies $\longmapsto$ for: decomposing a term into a reduction context E and a reducible expression e; contracting e into e' using $\longrightarrow$; and recomposing e' back into the current evaluation context to construct a new term.

$$
\begin{aligned}
\mathit{Expr} \ni e &::= e\ e \\
\mathit{Val} \ni v &::= \lambda x.e \mid x \\
x, y \in \mathit{Var} &\triangleq \{\mathrm{x}, \mathrm{y}, \ldots\}
\end{aligned}
$$

Figure 2.50: Abstract syntax for λ-calculus

Reduction semantic specifications are more concise than SOS and MSOS, since so-called congruence rules (e.g., rules like (SOS-Plus1 and (SOS-Plus2) in Section 2.2.2) are subsumed by the (RS-DCR) rule.

Using the rules, an expression such as $\texttt{plus}(\texttt{plus}(1,1),3)$ decomposes into a context $\texttt{plus}([\ ],3)$ and redex $\texttt{plus}(1,1)$. Taking the reflexive transitive closure of $\longmapsto$, we can thus evaluate $\texttt{plus}(\texttt{plus}(1,1),3)$ using the following sequence of reduction steps:

$$
\begin{aligned}
&\texttt{plus}(\texttt{plus}(1,1),3) \\
\longmapsto\ &\texttt{plus}(2,3) \\
\longmapsto\ &5
\end{aligned}
$$

2.8.2 Reduction semantics for call-by-value λ-calculus

Another significant difference between reduction semantics and the purely structural semantics frameworks recalled in earlier sections of this chapter is that reduction semantics often use meta-level substitution instead of explicit environments.[4] This difference is apparent in the reduction semantics for the call-by-value λ-calculus.

Abstract syntax. Unlike previous approaches, reduction semantics uses meta-level substitution, so there is no need for explicit closures in the abstract syntax. The abstract syntax for the call-by-value λ-calculus is given by the grammar in Figure 2.50.

Notion of contraction. Figure 2.51 specifies the evaluation contexts and reduction rules for the call-by-value λ-calculus. In the figure, the notation $e[x \leftarrow v]$ denotes meta-level substitution, which replaces all free occurrences of x by v. This relies on the traditional distinction between free and bound variables in λ-calculus. The following definitions of free variables and substitution are adapted from Pierce's [Pie02, Definition 5.3.2 and 5.3.5].

[4]It is common in the literature to see substitution used, but seldom formally defined, in SOS and natural semantics too.

$$E ::= [\,] \mid E\ e \mid v\ E$$

$$(\lambda x.e)\ v \longrightarrow e[x \leftarrow v] \qquad \text{(RS-App)}$$

$$\frac{e \longrightarrow e'}{E[e] \longmapsto E[e']} \qquad \text{(RS-DCR)}$$

Figure 2.51: Evaluation contexts and reduction rules for λ-calculus

Definition 2.13 (Free variables) The set of free variables of an expression e, written $FV(e)$, is defined as follows:

$$\begin{aligned} FV(x) &\triangleq \{x\} \\ FV(\lambda x.e) &\triangleq FV(e) \setminus \{x\} \\ FV(e_1\ e_2) &\triangleq FV(e_1) \cup FV(e_2) \end{aligned}$$

Definition 2.14 (Capture-avoiding substitution) The following rules define substitutions:

$$\begin{aligned} x[x \leftarrow e] &\triangleq e \\ y[x \leftarrow e] &\triangleq y && \text{if } y \neq x \\ (e_1\ e_2)[x \leftarrow e] &\triangleq e_1[x \leftarrow e]\ e_2[x \leftarrow e] \\ (\lambda y.e_1)[x \leftarrow e_2] &\triangleq \lambda y.e_1[x \leftarrow e_2] && \text{if } y \neq x \text{ and } y \notin FV(e_2) \end{aligned}$$

If the condition for applying substitution under λ is not met, a function should be α-converted (Definition 2.15) in order to make the substitution applicable.

Definition 2.15 (α-conversion)

$$\lambda x.e_1 \triangleq \lambda y.e_1[x \leftarrow y] \qquad \text{where } y \notin FV(e_1)$$

While the conventions for substitution summarised above are completely standard (following, e.g., Pierce [Pie02, Chapter 5] or Barendregt [Bar84, Chapter 2]), they still play a crucial part of the formalisation, and if new binding constructs are added, such as `let`-expressions, the definitions of free variables and substitution must be extended accordingly, and proofs involving these must be extended accordingly. Such concerns are avoided when using closures with explicit substitution, although we saw earlier that there were some problems with using small-step SOS with closures too, namely in connection with using guarded coinduction to prove divergence of ω in Proposition 2.3 in Section 2.2.4.

$$Expr \ni e ::= \texttt{throw}(e) \mid \texttt{catch}(e, x, e)$$

Figure 2.52: Abstract syntax for exception handling

$$\begin{aligned} E &::= [\,] \mid \texttt{throw}(E) \mid \texttt{catch}(E, x, e) \\ C &::= [\,] \mid \texttt{throw}(C) \end{aligned}$$

$$\frac{C \neq [\,]}{C[\texttt{throw}(v)] \longrightarrow \texttt{throw}(v)} \qquad \text{(RS-Throw)}$$

$$\texttt{catch}(\texttt{throw}(v), x, e)] \longrightarrow e[x \leftarrow v] \qquad \text{(RS-Catch-Throw)}$$

$$\texttt{catch}(v, x, e) \longrightarrow v \qquad \text{(RS-Catch)}$$

$$\frac{e \longrightarrow e'}{E[e] \longmapsto E[e']} \qquad \text{(RS-DCR)}$$

Figure 2.53: Evaluation contexts and reduction rules for exception handling

2.8.3 Reduction semantics for exception handling

A merit of reduction semantics is its ability to express abrupt termination concisely.

Abstract syntax. Unlike the previous semantics for exception handling surveyed in this chapter, there is no need for explicit exception terms. The abstract syntax is given by the grammar in Figure 2.52.

Notion of contraction. Figure 2.53 specifies evaluation contexts and reduction rules for exception handling. Here, we distinguish two kinds of contexts: ordinary evaluation contexts, ranged over by E may contain arbitrary constructs, including `catch` constructs; and evaluation contexts that do not contain any `catch` constructs, ranged over by C. The motivation for the distinction is to ensure that exceptions are always propagated to the closest enclosing handler. Unlike in SOS, the semantics of propagating exceptions in reduction semantics is given by a so-called *context-sensitive* reduction rule: the rule (RS-Throw) contracts the term $\texttt{throw}(v)$ in context C to dispose of its context – i.e., to propagate the exception to the closest enclosing handler (or the top-level); in contrast, *context-insensitive* rules like (RS-Catch-Throw) or (RS-Catch) simply contract terms in *any* context.

2.8.4 Extending with other features

Reduction semantics can also deal with features such as imperative state or control constructs [Fel87, WF94, FWFD88] in a concise manner.

Extending semantics with new features may require updating the notion of substitution, as well as introducing or modifying the notions of evaluation context specifications for existing constructs. Dealing with name-binding in a generic and modular way has received some attention in the literature; see, e.g., [SNO$^+$10, NTVW15]. It seems plausible that it is possible to give a generic way of extending evaluation contexts such that adding new constructs automatically extends all relevant evaluation contexts; e.g., if we were to extend Figure 2.53 with simple arithmetic expressions we should extend both the evaluation context grammars for C and E. Tools such as PLT Redex [KCD$^+$12] and Ott [SNO$^+$10] allow modular evaluation context grammars, but this modularity is not reflected in the foundations. This thesis focuses on SOS-style specifications, but the problems with dealing with substitution and name-binding, as well as modular evaluation context extensions are problems that would be interesting to explore in future work.

2.9 Type soundness using operational semantics

Type systems are typically given by an inductively defined big-step relation that specifies a typing relation.[5] The purpose of such relations is to check *statically* (before the program is evaluated) what the type of a program is, where a type can be thought of as a class of values that a program program produces if it terminates. Following Milner [Mil78], we say that a type system is *sound* if it guarantees that a program cannot go wrong.

There are several ways of utilising type systems that give rise to different problems:

- *Type inference problem:* given a program p, what is its type (if any)?
- *Type checking problem:* given an annotated program p and type T, does p have type T?
- *Type inhabitance problem:* is there a program p of type T?

Here, we consider the type inference problem.

2.9.1 A type system for λ_{cbv}

Figure 2.55 summarises the typing rules for λ_{cbv}. Here, types are defined by the grammar in Figure 2.54 where $T_1 \rightarrow T_2$ is a function type. The judgment '$\Gamma \vdash e : T$' says that e has type T in the *type environment* (or *typing context*) Γ, where $\Gamma \subseteq \mathit{Var} \xrightarrow{\text{fin}} \mathit{Type}$.

[5]Some authors have explored giving typing relations as small-step semantics; e.g., [Ser12, KMF07].

$$\mathit{Type} \ni T ::= \mathit{nat} \mid T \rightharpoonup T$$

Figure 2.54: Types for λ_{cbv}

$$\frac{\Gamma \vdash e_1 : \mathit{nat} \qquad \Gamma \vdash e_2 : \mathit{nat}}{\Gamma \vdash \texttt{plus}(e_1, e_2) : \mathit{nat}} \quad \text{(T-Plus)}$$

$$\frac{}{\Gamma \vdash n : \mathit{nat}} \quad \text{(T-Nat)}$$

$$\frac{\Gamma[x \mapsto T_1] \vdash e : T_2}{\Gamma \vdash \lambda x.e : T_1 \rightharpoonup T_2} \quad \text{(T-Fun)}$$

$$\frac{\Gamma \vdash e_1 : T_1 \rightharpoonup T_2 \qquad \Gamma \vdash e_2 : T_1}{\Gamma \vdash e_1\ e_2 : T_2} \quad \text{(T-App)}$$

$$\frac{x \in \operatorname{dom}(\Gamma)}{\Gamma \vdash x : \Gamma(x)} \quad \text{(T-Var)}$$

Figure 2.55: Typing rules for source expressions in λ_{cbv}

An expression e has type T in a type environment Γ exactly when we can construct a derivation tree whose conclusion is $\Gamma \vdash e : T$. For example, we can infer that $\lambda x.1 + x$ has type $\mathit{nat} \rightharpoonup \mathit{nat}$ in the empty type environment '$\emptyset$':

$$\frac{\dfrac{\{x \mapsto \mathit{nat}\} \vdash 1 : \mathit{nat} \qquad \{x \mapsto \mathit{nat}\} \vdash x : \mathit{nat}}{\{x \mapsto \mathit{nat}\} \vdash 1 + x : \mathit{nat}}}{\emptyset \vdash \lambda x.1 + x : \mathit{nat} \rightharpoonup \mathit{nat}}$$

2.9.2 Small-step type soundness of λ_{cbv} using SOS

We recall differences between small-step and big-step type soundness proofs, largely following Felleisen and Wright [WF94]. In their exposition they use reduction semantics as the basis for small-step type soundness, whereas we use SOS.

Let '$\rightarrow$' be the small-step transition relation for λ_{cbv} given in Section 2.2.4. We can prove that the type system in Figure 2.55 is type sound by means of two lemmas, *progress* and *preservation* (also called *subject reduction* [CF58]), following Felleisen and Wright's Syntactic Approach to Type Soundness. To this end we define a relation $\mathit{compatible} \subseteq (\mathit{Var} \xrightarrow{\text{fin}} \mathit{Val}) \times (\mathit{Var} \xrightarrow{\text{fin}} \mathit{Type})$ for checking that a given concrete environment is compatible with a given type environment:

$$\frac{\forall x \in \operatorname{dom}(\Gamma).\ x \in \operatorname{dom}(\rho) \ \wedge\ \emptyset \vdash \rho(x) : \Gamma(x)}{\mathit{compatible}(\rho, \Gamma)} \quad \text{(TE-Compatible)}$$

As the astute reader may have noticed, the *compatible* relation has a bug: it relies on the typing relation for checking that each $\{x \mapsto v\} \in \rho$ is typed by a corresponding $\{x \mapsto T\} \in \Gamma$; but the typing relation does not specify the type of closures! In order to make it well-defined, and in order to prove progress (below), the typing relation in Figure 2.55 must be augmented by the rule:

$$\frac{compatible(\rho, \Gamma') \qquad \Gamma'[x \mapsto T_1] \vdash e : T_2}{\Gamma \vdash \langle x, e, \rho \rangle : T_1 \rightharpoonup T_2} \qquad \text{(T-Clo)}$$

Thus there is a mutual dependency between the *compatible* relation and the typing relation. One must be wary of such mutual relationships, as they might make the relation non-well-founded, and allow us to encode and prove paradoxical and inconsistent facts, such as Russel's paradox. This particular relationship is, however, easily seen to be well-founded: not only is it *strictly-positive*, and hence well-known to be well-founded [CPM90]; we could also have inlined *compatible* in the typing relation instead, which would eliminate the mutual dependency.[6] In order to prove progress by induction on the typing relation, it is convenient to break the typing rule for application into two typing rules:

$$\frac{\Gamma \vdash e_1 : T_1 \rightharpoonup T_2 \qquad \Gamma \vdash e_2 : T_1 \qquad \neg(e_1, e_2 \in Val)}{\Gamma \vdash e_1\ e_2 : T_2} \qquad \text{(T-App1)}$$

$$\frac{compatible(\rho, \Gamma') \qquad \Gamma'[x \mapsto T_1] \vdash e : T_2 \qquad \Gamma \vdash v : T_1}{\Gamma \vdash \langle x, e, \rho \rangle\ v : T_2} \qquad \text{(T-App2)}$$

These rules are easily seen to be equivalent to the rule (T-App). Replacing (T-App) by these two rules makes it straightforward to prove preservation by induction on the typing relation. We discuss the challenge with using the (T-App) rule alone in the progress proof of case (T-App1).

Lemma 2.16 (Progress)

$$e \notin Val \implies \Gamma \vdash e : T \implies compatible(\rho, \Gamma) \implies \exists e'.\ \rho \vdash e \rightarrow e'$$

Proof. The proof is by rule induction on the typing relation.

Case (T-Plus) From the induction hypothesis we have:

$$\forall \rho.\ e_1 \notin Val \implies compatible(\rho, \Gamma) \implies \exists e_1'.\ \rho \vdash e_1 \rightarrow e_1' \qquad \text{(IH1)}$$

$$\forall \rho.\ e_2 \notin Val \implies compatible(\rho, \Gamma) \implies \exists e_2'.\ \rho \vdash e_2 \rightarrow e_2' \qquad \text{(IH2)}$$

From the goal we have:

$$compatible(\rho, \Gamma) \qquad \text{(H1)}$$

$$\Gamma \vdash e_1 : nat \qquad \text{(H2)}$$

$$\Gamma \vdash e_2 : nat \qquad \text{(H3)}$$

[6]It is for didactic reasons that we gave *compatible* as separate relation.

The goal is:

$$\exists e'.\ \rho \vdash \mathtt{plus}(e_1, e_2) \rightarrow e' \qquad \text{(Goal)}$$

We reason by case analysis on whether $e_1 \in \mathit{Val}$ or not.

Subcase ($e_1 \notin \mathit{Val}$) By eliminating the existential in (IH1), the goal follows straightforwardly from the (SOS-Plus1) rule.

Subcase ($e_1 \in \mathit{Val}$) It must be the case that e_1 is some natural number n_1, or (H2) would be a contradiction. We reason by case analysis on whether $e_2 \in \mathit{Val}$ or not.

Subcase ($e_2 \notin \mathit{Val}$) The goal follows straightforwardly from (IH2).

Subcase ($e_2 \in \mathit{Val}$) It must be the case that e_2 is some natural number n_2, or (H3) would be a contradiction. The goal follows from the (SOS-λ_{cbv}-Plus) rule.

Case (T-Nat) The case leads to a contradiction, since $n \in \mathit{Val}$.

Case (T-Fun) The goal follows trivially from the (SOS-λ_{cbv}-Lam) rule.

Case (T-App1) From the induction hypothesis we have:

$$\forall \rho.\ e_1 \notin \mathit{Val} \implies \mathit{compatible}(\rho, \Gamma) \implies \exists e_1'.\ \rho \vdash e_1 \rightarrow e_1' \qquad \text{(IH1)}$$

$$\forall \rho.\ e_2 \notin \mathit{Val} \implies \mathit{compatible}(\rho, \Gamma) \implies \exists e_2'.\ \rho \vdash e_2 \rightarrow e_2' \qquad \text{(IH2)}$$

From the goal we have:

$$\mathit{compatible}(\rho, \Gamma) \qquad \text{(H1)}$$

$$\Gamma \vdash e_1 : T_1 \rightharpoonup T_2 \qquad \text{(H2)}$$

$$\Gamma \vdash e_2 : T_1 \qquad \text{(H3)}$$

$$\neg(e_1, e_2 \in \mathit{Val}) \qquad \text{(H4)}$$

The goal is:

$$\exists e'.\ \rho \vdash e_1\ e_2 \rightarrow e' \qquad \text{(Goal)}$$

We reason by case analysis on whether $e_1 \in \mathit{Val}$ or not.

Subcase ($e_1 \notin \mathit{Val}$) By eliminating the existential in (IH1), the goal follows straightforwardly from the (SOS-Plus1) rule.

Subcase ($e_1 \in \mathit{Val}$) It must be the case that e_1 is some closure $\langle x, e, \rho \rangle$, or (H2) would be a contradiction. We reason by case analysis on whether $e_2 \in \mathit{Val}$ or not.

Subcase ($e_2 \notin \mathit{Val}$) The goal follows straightforwardly from (IH2).

Subcase ($e_2 \in \mathit{Val}$) Contradiction, by (H4).

But consider if this had not been a contradiction: then $e_2 = v_2$, and we would have to prove that there was some e' such that $\rho'[x \mapsto v_2] \vdash e \rightarrow e'$ holds. But the induction hypotheses do not allow us to infer that this is the case. This is the motivation for breaking the application rule into two equivalent rules.

Case (T-App2) From the induction hypothesis we have:

$$\forall \rho.\ e \notin \mathit{Val} \implies \mathit{compatible}(\rho, \Gamma'[x \mapsto T_1]) \implies \exists e_1'.\ \rho \vdash e_1 \to e_1' \tag{IH1}$$

From the goal we have:

$$\mathit{compatible}(\rho, \Gamma') \tag{H1}$$

$$\Gamma'[x \mapsto T_1] \vdash e : T_2 \tag{H2}$$

$$\Gamma \vdash v : T_1 \tag{H3}$$

The goal is:

$$\exists e'.\ \rho \vdash \langle x, e, \rho \rangle\ v \to e' \tag{Goal}$$

We reason by case analysis on whether $e \in \mathit{Val}$ or not.

Subcase ($e \notin \mathit{Val}$) By Lemma 2.17 (see below), (H1), and (H3), we get:

$$\mathit{compatible}(\rho[x \mapsto v], \Gamma'[x \mapsto T_1]) \tag{H4}$$

The goal now follows from (H4), (IH1), and rule (SOS-λ_{cbv}-AppC).

Subcase ($e \in \mathit{Val}$) The goal follows from rule (SOS-λ_{cbv}-App).

Case (T-Var) From the goal we have:

$$\mathit{compatible}(\rho, \Gamma) \tag{H1}$$

$$x \in \mathrm{dom}(\Gamma) \tag{H2}$$

The goal is:

$$\exists e'.\ \rho \vdash x \to e' \tag{Goal}$$

From (H1) and (TE-Compatible), we get that $x \in \mathrm{dom}(\rho)$, since $x \in \mathrm{dom}(\Gamma)$. Thus $\exists v.\ \rho(x) = v$. The goal follows from this fact and rule (SOS-λ_{cbv}-Var).

□

Lemma 2.17 (Compatibility is preserved by well-typed updates)

$$\mathit{compatible}(\rho, \Gamma) \implies \forall \Gamma'.\ \Gamma' \vdash v : T \implies \forall x.\ \mathit{compatible}(\rho[x \mapsto v], \Gamma[x \mapsto T])$$

Proof. Immediate from the definition of (TE-Compatible) and from Lemma 2.18 (see below). □

Lemma 2.18 (Environment is irrelevant for typing values)

$$\Gamma \vdash v : T \implies \forall \Gamma'.\ \Gamma' \vdash v : T$$

Proof. The proof is by rule induction on the first premise. There are two cases to consider: numbers n and closures $\langle x, e, \rho \rangle$. Both cases are trivial. □

For the preservation proof, we use the typing relation given by the (T-App) rule, without splitting it into two.

Lemma 2.19 (Preservation)

$$\rho \vdash e \rightarrow e' \implies \Gamma \vdash e : T \implies \mathit{compatible}(\rho, \Gamma) \implies \Gamma \vdash e' : T$$

Proof. The proof is by rule induction on the first premise.

Case (SOS-λ_{cbv}-Plus1) The induction hypothesis gives:

$$\forall \Gamma\ T.\ \Gamma \vdash e_1 : T \implies \mathit{compatible}(\rho, \Gamma) \implies \Gamma \vdash e_1' : T \tag{IH}$$

From the goal we have:

$$\mathit{compatible}(\rho, \Gamma) \tag{H1}$$

$$\Gamma \vdash \texttt{plus}(e_1, e_2) : T \tag{H2}$$

The goal is:

$$\Gamma \vdash \texttt{plus}(e_1', e_2) : \mathit{nat} \tag{Goal}$$

By inversion on (H2), we get:

$$\Gamma \vdash e_1 : \mathit{nat} \tag{H3}$$

$$\Gamma \vdash e_2 : \mathit{nat} \tag{H4}$$

The goal follows by applying (T-App), (IH), (H1), (H3), and (H4).

Case (SOS-λ_{cbv}-Plus2) The structure of the proof for this case is similar to that for (SOS-λ_{cbv}-Plus1).

Case (SOS-λ_{cbv}-Plus) The goal follows trivially:

$$\Gamma \vdash n : \mathit{nat} \tag{Goal}$$

Case (SOS-λ_{cbv}-Lam) From the goal we have:

$$\mathit{compatible}(\rho, \Gamma) \tag{H1}$$

$$\Gamma \vdash \lambda x.e : T \tag{H2}$$

The goal is:

$$\Gamma \vdash \langle x, e, \rho \rangle : T \tag{Goal}$$

By inversion on (H2), we get:

$$\Gamma[x \mapsto T_1] \vdash e : T_2 \tag{H3}$$

The goal follows from rule (T-Clo), (H1), and (H3).

Case (SOS-λ_{cbv}-App1) The structure of the proof for this case is similar to that for (SOS-λ_{cbv}-Plus1).

Case (SOS-λ_{cbv}-App2) The structure of the proof for this case is similar to that for (SOS-λ_{cbv}-Plus1).

Case (SOS-λ_{cbv}-AppC) The structure of the proof for this case is similar to that for (SOS-λ_{cbv}-Plus1).

Case (SOS-λ_{cbv}-App) From the goal we have:

$$\Gamma \vdash \langle x, v, \rho \rangle\ v_2 : T \tag{H1}$$

The goal is:

$$\Gamma \vdash v : T \tag{Goal}$$

By inversion on (H1), we get:

$$\Gamma \vdash \langle x, v, \rho \rangle : T_1 \twoheadrightarrow T_2 \tag{H2}$$

$$\Gamma \vdash v_2 : T_1 \tag{H3}$$

The goal becomes:

$$\Gamma \vdash v : T_2 \tag{Goal$'$}$$

By inversion on (H2), we get:

$$\Gamma'[x \mapsto T_1] \vdash v : T_2 \tag{H4}$$

From Lemma 2.18, we get:

$$\Gamma \vdash v : T_2 \tag{H}$$

from which the goal follows.

Case (SOS-λ_{cbv}-Var) From the goal we have:

$$\mathit{compatible}(\rho, \Gamma) \tag{H1}$$

$$\Gamma \vdash x : T \tag{H2}$$

The goal is:

$$\Gamma \vdash \rho(x) : T \tag{Goal}$$

By inversion on (H2), we get:

$$x \in \mathrm{dom}(\Gamma) \tag{H2}$$

The goal becomes:

$$\Gamma \vdash \rho(x) : \Gamma(x) \tag{Goal}$$

The goal follows from inversion on (H1) and Lemma 2.18.

□

Proposition 2.20 (Type soundness)

$$\Gamma \vdash e : T \implies \mathit{compatible}(\rho, \Gamma) \implies e \in \mathit{Val} \vee (\exists e'.\ \rho \vdash e \to e' \wedge \Gamma \vdash e' : T)$$

Proof. If e is a value, the conclusion follows. Otherwise, progress gives us that there exists a further transition, and preservation gives us that the term resulting from the transition must be well-typed, from which the goal follows. □

For some semantics it is necessary to give typing rules for terms that are not valid source program terms in order to type intermediate expressions, i.e., expressions that never occur as source programs, and only play an administrative role for the operational semantics. This complicates the typing relation, and occasionally requires extra predicates and conditions in order to distinguish source-terms from intermediate terms [WF94, Section 6.4]. Big-step semantics do not suffer from this drawback, but have drawbacks of their own.

2.9.3 Big-step type soundness

Small-step semantics commonly uses intermediate syntax for recording values or auxiliary entities that may occur as part of abstract syntax trees during evaluation. In contrast, natural semantics usually have a clear distinction between the syntactic sorts for source programs vs. value terms. Thus, whereas the typing relation for small-step semantics must assign types to intermediate expressions, big-step semantics is free of these concerns, which in this case leads to a simpler typing relation: for example, the typing relation does not need to cover closures, since closures are values that never occur in source programs.

The typing rules for the natural semantics for λ_{cbv} given in Figure 2.27 coincide with those given in Figure 2.55, except for the rule (T-Nat) for natural numbers, which is replaced by the following rule that types numeral terms ($\texttt{num}(n)$) instead:

$$\frac{}{\Gamma \vdash \texttt{num}(n) : \mathit{nat}} \tag{T-Num}$$

$$\frac{}{\overset{V}{\vdash} n : \mathit{nat}} \qquad \text{(TV-Nat)}$$

$$\frac{\mathit{compatible}(\rho,\Gamma) \qquad \Gamma[x \mapsto T_1] \vdash e : T_2}{\overset{V}{\vdash} \langle x,e,\rho\rangle : T_1 \twoheadrightarrow T_2} \qquad \text{(TV-Clo)}$$

Figure 2.56: Value typing rules for λ_{cbv}

In order to express type soundness using a big-step relation one typically defines a value typing relation which assigns types to all values that programs may produce. We follow Milner and Tofte [MT91] and define a relation $\overset{V}{\vdash} \subseteq \mathit{Val} \times \mathit{Type}$ given by the rules in Figure 2.56, where the judgment $\overset{V}{\vdash} v : T$ asserts that the value v has type T, and *compatible* is defined as before.

The traditional approach to proving big-step type soundness [Mil78, Tof90, MT91] is to prove a statement like the following:

$$\Gamma \vdash e : T \implies \mathit{compatible}(\rho,\Gamma) \implies \rho \vdash e \Rightarrow o \implies \overset{V}{\vdash} o : T \qquad \text{(Big-Type-Preserve)}$$

Intuitively, the property says that, if evaluation converges, it produces an outcome with the same type as we inferred for the expression. But what happens if evaluation of e goes wrong? Unless the evaluation relation $\Rightarrow$ *explicitly* handles this by returning an outcome representing going wrong, we might prove the statement true for defective typing relations that permit computations that do go wrong.

The traditional approach to proving big-step type soundness thus involves introducing special "wrong" values everywhere evaluation can get stuck. Figure 2.57 summarises rules for "going wrong" that must be added to the natural semantics from Section 2.4.4. Unless one can automatically generate such wrong rules (which seems an approach that is rarely, if ever, utilised in practice), one has to add these rules manually. This makes the approach brittle: adding such rules is error-prone, and leaving out a source of "going" wrong may make the type soundness proof vacuous.

Leroy and Grall [LG09, Lemma 50] avoids giving "wrong" rules using a so-called "big-step progress" theorem, which states that programs that type-check but fail to co-evaluate must diverge – which, in turn, implies that they cannot go "wrong". Their proof relies on guarded coinduction and extensive use of the law of excluded middle for case analysis. Since the approach is based on traditional big-step divergence rules, their specification suffers from the duplication problem. Another shortcoming is that both proofs require some degree of semantic insight, in the sense of having to do inversion on typing relations and apply auxiliary lemmas in a fashion that requires some degree of semantic insight about the semantics and type system.

Charguéraud [Cha13, Section 3.3] proposes an alternative way of giving wrong transitions based on a special progress predicate, such that forgetting to add a rule for the progress predicate makes type soundness impossible to prove. This makes proving

Notion of outcome.

$$\mathit{Outcome}^{\text{WNS}} ::= v \mid \textsc{wrong}$$

Semantics of going wrong.

$$\frac{\rho \vdash e_1 \Rightarrow o_1 \qquad o_1 \notin \mathbb{N}}{\rho \vdash \mathtt{plus}(e_1, e_2) \Rightarrow \textsc{wrong}} \qquad (\text{NS-}\lambda_{\text{cbv}}\text{-Wrong-Plus1})$$

$$\frac{\rho \vdash e_1 \Rightarrow n_1 \qquad \rho \vdash e_2 \Rightarrow o_2 \qquad o_2 \notin \mathbb{N}}{\rho \vdash \mathtt{plus}(e_1, e_2) \Rightarrow \textsc{wrong}} \qquad (\text{NS-}\lambda_{\text{cbv}}\text{-Wrong-Plus2})$$

$$\frac{\rho \vdash e_1 \Rightarrow o_1 \qquad o_1 \notin \{\langle x, e, \rho \rangle\}}{\rho \vdash e_1\ e_2 \Rightarrow \textsc{wrong}} \qquad (\text{NS-}\lambda_{\text{cbv}}\text{-Wrong-App1})$$

$$\frac{\rho \vdash e_1 \Rightarrow \langle x, e, \rho' \rangle \qquad \rho \vdash e_2 \Rightarrow \textsc{wrong}}{\rho \vdash e_1\ e_2 \Rightarrow \textsc{wrong}} \qquad (\text{NS-}\lambda_{\text{cbv}}\text{-Wrong-App2})$$

$$\frac{\rho \vdash e_1 \Rightarrow \langle x, e, \rho' \rangle \qquad \rho \vdash e_2 \Rightarrow v_2 \qquad \rho'[x \mapsto v_2] \vdash e \Rightarrow \textsc{wrong}}{\rho \vdash e_1\ e_2 \Rightarrow \textsc{wrong}} \qquad (\text{NS-}\lambda_{\text{cbv}}\text{-App}')$$

$$\frac{x \notin \text{dom}(\rho)}{\rho \vdash x \Rightarrow \textsc{wrong}} \qquad (\text{NS-}\lambda_{\text{cbv}}\text{-Wrong-Var})$$

Figure 2.57: “Wrong” rules for λ_{cbv}

type soundness less error-prone, but still involves some manual effort in giving the necessary progress judgments, unless one could generate progress judgments somehow, which Charguéraud suggests might be possible.

In the next section we recall Cousot's approach to types as abstract interpretations which provides an alternative way of proving type soundness using a big-step relation. While the next section recalls Cousot's exposition that is based on semantics with explicit WRONG transitions, the approach supports proving type soundness without these, too, as we show in Chapter 6.

2.9.4 Types as abstract interpretations

Following Cousot [Cou97], we can interpret types as *abstract interpretations* [CC79]. The idea is to formalise the relationship between sets of concrete *denotations* (i.e., the set of all possible meanings of a program in any context) and sets of *typings* (i.e., the set of all possible types of a program in any context) as a *Galois connection*. The Galois connection allows us to obtain, from any set of typings the set of all possible concrete denotations that correspond to it; and vice versa, from any set of concrete denotations we can obtain the set of all possible corresponding typings. Thus, the Galois connection formalises the *most precise* typing relation. For most interesting languages, using the Galois connections type inference would be undecidable. But, following Cousot, the Galois connection provides a useful guiding principle for discovering relations that are decidable by proving that these are safe approximations of the most precise relation.

We recall how Cousot uses abstract interpretation to give meaning to types, and how this meaning can be used to prove type safety. Cousot's exposition uses a denotational semantics, but here we make the straightforward adaptation of the approach to natural semantics.

Denotations. The denotation of an expression is given by its set of possible meanings. Using the natural semantics for λ_{cbv} with the WRONG rules in Figure 2.57, we can define a function that returns this set for arbitrary expressions (where Λ is a meta-level function abstraction, and $\mathit{Env} \triangleq \mathit{Var} \xrightarrow{\text{fin}} \mathit{Val}$):

$$\mathbf{D}[\![\bullet]\!] \in \mathit{Expr} \to \mathit{Env} \to \wp(\mathit{Outcome}_\perp)$$

$$\mathbf{D}[\![e]\!] = \Lambda\rho.\{o \mid \rho \vdash e \Rightarrow o\} \cup \{\perp \mid \rho \vdash e \overset{\infty}{\Rightarrow}\}$$

Here, $\mathit{Outcome}_\perp$ denotes the disjoint union $\mathit{Outcome} + \{\perp\}$. Following standard notation from the literature on denotational semantics [SS71, Sch86], $\perp$ is a special value that indicates undefinedness and divergence.

Typings. A typing is a pair consisting of a type environment in $\mathit{TypeEnv} \triangleq \mathit{Var} \xrightarrow{\text{fin}} \mathit{Type}$ and a type. But what is the meaning of a typing? According to Cousot [Cou97], the meaning of typings is given by a Galois connection between sets of typings and sets of denotations. Recall that a Galois connection is given by a pair of functions α, γ, where we say that α is an *abstraction* function, and γ is a *concretisation* function. Following Cousot, we define a concretisation function, which relates sets of typings in $\mathbb{T} \triangleq \wp(\mathit{TypeEnv} \times \mathit{Type})$ and sets of concrete denotations in $\wp(\mathbb{D})$ where $\mathbb{D} \triangleq \mathit{Env} \to \wp(\mathit{Outcome}_\perp)$.

Figure 2.58 defines a concretisation function that assigns meaning to typings for λ_{cbv} by relating them to concrete denotations. For example, using it, the meaning of the type $\mathit{nat} \rightarrow \mathit{nat}$ is the set of all closures $\langle x, e, \rho\rangle$ which, when applied to a value of type *nat*, entails an outcome of type *nat*, meaning it either diverges or returns some natural number n. The meaning of a set of typings P is given conjunctively, i.e., all programs in $\gamma_P(P)$ must be typable by each typing $p \in P$.

Galois connection. A Galois connection is given by a pair of functions α, γ that satisfy for two partially ordered sets $\langle A, \sqsubseteq^A\rangle$ and $\langle B, \sqsubseteq^B\rangle$:

$$\alpha \in A \to B \qquad \gamma \in B \to A$$

$$\alpha(a) \sqsubseteq^B b \iff a \sqsubseteq^A \gamma(b)$$

For our $\mathbb{D}$ and $\mathbb{T}$, we want a pair of functions that satisfy:

$$\alpha \in \wp(\mathbb{D}) \to \mathbb{T} \qquad \gamma \in \mathbb{T} \to \wp(\mathbb{D})$$

$$\alpha(D) \supseteq^{\mathbb{T}} P \iff D \subseteq^{\mathbb{D}} \gamma(P)$$

$$
\begin{aligned}
\gamma_V &\in \mathit{Type} \to \wp(\mathit{Outcome}_\bot) \\
\gamma_V(\mathit{nat}) &\triangleq \mathbb{N} \cup \{\bot\} \\
\gamma_V(T_1 \twoheadrightarrow T_2) &\triangleq \left\{ \langle x, e, \rho \rangle \;\middle|\; \begin{array}{l} \forall v_1 \in \gamma_V(T_1). \\ \qquad \forall o_2.\ o_2 \in \mathbf{D}[\![e]\!](\rho[x \mapsto v_1]) \ \land\ o_2 \in \gamma_V(T_2) \end{array} \right\} \cup \{\bot\} \\
\gamma_R &\in \mathit{TypeEnv} \to \mathit{Env} \\
\gamma_R(\Gamma) &\triangleq \{\rho \mid \forall x \in \mathrm{dom}(\Gamma).\ x \in \mathrm{dom}(\rho) \ \land\ \rho(x) \in \gamma_V(\Gamma(x))\} \\
\gamma_P &\in (\mathit{TypeEnv} \times \mathit{Type}) \to \wp(\mathbb{D}) \\
\gamma_P((\Gamma, T)) &\triangleq \{d \mid \forall \rho.\ \rho \in \gamma_R(\Gamma) \implies \forall o.\ o \in d(\rho) \implies o \in \gamma_V(T)\} \\
\gamma &\in \mathbb{T} \to \wp(\mathbb{D}) \\
\gamma(P) &\triangleq \bigcap_{p \in P} \gamma_P(p) \qquad\qquad \gamma(\emptyset) \triangleq \mathbb{D}
\end{aligned}
$$

Figure 2.58: Concretisation function for λ_{cbv}

Here, the inclusion order is reversed, because types are interpreted *conjunctively*: the set of typings one can assign to a set of denotations D are valid typings for *all* denotations $d \in D$. Specifically, due to the way our concretisation function is constructed, it holds that:

$$
\begin{aligned}
& \gamma(\bigcup_{i \in \Delta} P_i) \\
=\ & \bigcap_{p \in \bigcup_{i \in \Delta} P_i} \gamma_P(p) && \text{(by def. of } \gamma) \\
=\ & \bigcap_{i \in \Delta, p \in P_i} \gamma_P(p) && \text{(by def. of } \textstyle\bigcup) \\
=\ & \bigcap_{i \in \Delta} \gamma(P_i) && \text{(by def. of } \textstyle\bigcap \text{ and } \gamma)
\end{aligned}
$$

From this fact, we can use well-known results about Galois connections between complete lattices [NNH99, Chapter 4.3] to deduce that γ uniquely defines an abstraction function α such that there is a Galois connection between $\mathbb{D}$ and $\mathbb{T}$, i.e., that:

$$\alpha(D) \supseteq T \iff D \subseteq \gamma(T)$$

Following standard notation, we write:

$$(\mathbb{T}, \supseteq) \underset{\gamma}{\overset{\alpha}{\leftrightarrows}} (\wp(\mathbb{D}), \subseteq)$$

Type soundness. An important property of the concretisation function is that it does not assign any meaning to the WRONG value. Thus, whenever it holds for some set of typings P:

$$\mathbf{D}[\![e]\!] \in \gamma(P)$$

it holds that e does not go wrong. Given this fact, we can use the Galois-connection as a guiding principle for finding a typing procedure $\mathbf{T}[\![e]\!]$ that allows us to approximate the set of actual typings for given program expressions. It must hold that:

$$\begin{aligned} \alpha(\{\mathbf{D}[\![e]\!]\}) \sqsupseteq^{\mathbb{T}} \mathbf{T}[\![e]\!] &\iff && \text{(by Galois connection)} \\ \{\mathbf{D}[\![e]\!]\} \subseteq^{\mathbb{D}} \gamma(\mathbf{T}[\![e]\!]) &\iff && \text{(by subset inclusion)} \\ \mathbf{D}[\![e]\!] \in^{\mathbb{D}} \gamma(\mathbf{T}[\![e]\!]) & && \end{aligned}$$

Here, the first line reads "**T** safely approximates the set of typings for the expression e". If we let $\mathbf{T}[\![\bullet]\!] \in \mathit{Expr} \to \mathbb{T}$ be the following typing denotation function defined in terms of the typing relation in Figure 2.55:

$$\mathbf{T}[\![e]\!] \triangleq \{(\Gamma, T) \mid \Gamma \vdash e : T\}$$

then, by unfolding definitions:

$$\begin{aligned} & \mathbf{D}[\![e]\!] \in \gamma(\mathbf{T}[\![e]\!]) \\ \iff & (\Gamma, T) \in \mathbf{T}[\![e]\!] \implies \forall \rho.\ \rho \in \gamma_R(\Gamma) \implies \\ & \qquad \forall o.\ o \in \mathbf{D}[\![e]\!](\rho) \implies o \in \gamma_V(T) && \text{(by def. of } \gamma_P \text{ and } \in) \\ \iff & \Gamma \vdash e : T \implies \forall \rho.\ \rho \in \gamma_R(\Gamma) \implies \\ & \qquad \forall o.\ (\rho \vdash e \Rightarrow o) \vee (o = \bot \wedge \rho \vdash e \overset{\infty}{\Rightarrow}) \implies o \in \gamma_V(T) && \text{(by def. of } \mathbf{T} \text{ and } \mathbf{D}) \end{aligned}$$

The result is a type soundness statement reminiscent of the type soundness statement (Big-Type-Preserve) from 2.9.3, but with some differences:

- the concretisation functions are more precise (as in, they give types to more terms) than the value-typing relation $\overset{V}{\vdash}$ and *compatible* relation; for example, the $\mathbf{D}[\![\omega]\!] \in \gamma(\mathbb{T})$, whereas ω is untypable using $\overset{V}{\vdash}$; and
- the type soundness statement based on abstract interpretation explicitly mentions the divergence relation – this difference is, however, mainly superficial, due to the way divergent computations are typed.

Types as abstract interpretations provides an answer to the question "what is the meaning of types" [Rey03] that is a useful basis for proving type soundness. In Chapter 6 we show that the extra precision afforded by the Galois connection in fact enables us to prove type soundness without adding "wrong" transitions to a big-step semantics. Before delving into that, we focus on the pragmatic problem with giving and relating small-step and big-step extensible specifications that none of the frameworks recalled in this chapter provide a satisfactory solution to.

2.10 Summary

We have recalled previous approaches to semantics specification from the literature. These provide varying degrees of support for language evolution and concise specification. We have postponed an overview of how to relate small-step and big-step SOS, but remark that there are standard approaches to this in the literature. Such proofs are often established manually, and are complicated by the poor support that many frameworks for SOS exhibit.

In Chapter 3 of this thesis we introduce a variant of SOS that provides better support for language evolution for both small-step and big-step semantics. This allows us to prove, once-and-for-all, in Chapters 4 and 5 the relationship between a class of small-step and big-step SOS rules that suffices to give semantics for diverse language features, including abrupt termination, divergence, and continuation-based control constructs.

We have also recalled previous approaches to proving type soundness. In particular, big-step approaches are challenging to work with, since they require explicit notions of going wrong, which is tedious and error-prone. In Chapters 6, 7, and 8 we study how types as abstract interpretations provides a proof method for proving big-step type soundness without having to add explicit "wrong" rules.

3 Extensible Transition System Semantics

Contents

MSOS supports a technique for abrupt termination as signals in transition system labels, but the technique does not directly translate to big-step semantics. We analyse the problem and propose a simple but novel variant of Mosses' generalised transition systems that we call *extensible transition systems* (XTS) as a solution. We also introduce a variant of MSOS that we call *Extensible SOS* (XSOS).

3.1 The problem with modular abrupt termination

Section 2.3.4 presented the technique for abrupt termination that is used in MSOS. In Section 2.6.2 we saw that the technique does not directly translate to big-step MSOS. Some of the drawbacks that prevent the technique from translating are also apparent in small-step MSOS with the abrupt termination technique.

Using the encoding of abrupt termination recalled in Figure 2.20, a generalised transition system abruptly terminates when an exception signal occurs in the context of a `program` expression. In other words, if an expression throws an exception that does not have an enclosing `program` term, evaluation does *not* terminate abruptly. For example, the expression $\texttt{plus}'(\texttt{throw}(0), \omega)$ does not terminate if $\texttt{plus}'$ has interleaving order of evaluation. The expression diverges instead:

$$\begin{array}{rl}
 & \texttt{plus}'(\texttt{throw}(0), \omega) \\
\xrightarrow{\{\mathbf{exc}'=\mathrm{EXC}(0),\mathbf{env}=\emptyset,—\}} & \texttt{plus}'(\texttt{stuck}, \omega) \\
\xrightarrow{\{\mathbf{exc}'=\mathrm{EXC}(0),\mathbf{env}=\emptyset,—\}}{}^{*} & \texttt{plus}'(\texttt{stuck}, \langle x, x\ x, \emptyset\rangle\ \langle x, x\ x, \emptyset\rangle) \\
\xrightarrow{\{\mathbf{exc}'=\mathrm{EXC}(0),\mathbf{env}=\emptyset,—\}}{}^{*} & \texttt{plus}'(\texttt{stuck}, \langle x, \langle x, x\ x, \emptyset\rangle\ \langle x, x\ x, \emptyset\rangle, \emptyset\rangle\ \langle x, x\ x, \emptyset\rangle) \\
\xrightarrow{\{\mathbf{exc}'=\mathrm{EXC}(0),\mathbf{env}=\emptyset,—\}} & \ldots
\end{array}$$

The root of the issue is that exceptions are not terminal configurations.

3.2 Towards a solution: abruptly terminated configurations

Figure 3.1 illustrates how encoding abrupt termination as configurations goes towards solving the problem. The figure defines a generalised transition system $\langle \Gamma, \mathbb{C}, \rightarrow, T\rangle$ where $\Gamma \triangleq \mathit{Expr} \times \mathit{ExcStat}$, $\mathbb{C} \triangleq \mathbb{1}$ (i.e., a singleton category with a single object and a single arrow), $\rightarrow$ is the transition relation in Figure 3.1, and $T \triangleq \{(v, \mathrm{OK})\} \cup \{(e, \mathrm{EXC})\}$. Here, the syntactic set *ExcStat* is an exception status flag, used to indicate whether abrupt termination has occurred or not. Programs only have further transitions insofar as they are not in an abruptly terminated state.

Using this approach, exceptions are propagated without introducing new rules for existing constructs, like in MSOS. Unlike MSOS, exceptions are now terminal configurations on their own. For example, $\texttt{plus}(\texttt{throw}(0), \texttt{plus}(1,2))$ now abruptly terminates without any top-level handler:

$$\begin{array}{rl}
 & (\texttt{plus}(\texttt{throw}(0), \texttt{plus}(1,2)), \mathrm{OK}) \\
\xrightarrow{\{—\}} & (\texttt{plus}(\texttt{stuck}, \texttt{plus}(1,2)), \mathrm{EXC}(0))
\end{array}$$

This approach does not require us to introduce new rules for existing constructs, and does not rely on top-level program expressions in order to abruptly terminate. However, it relies on an auxiliary entity as part of the configuration. This differs from generalised transition systems, in that generalised transition systems require configurations to be pure abstract syntax trees or computed values [Mos04, p. 206]. *Extensible transition systems* relax the requirement that configurations in generalised transition systems must be pure abstract syntax trees or computed values.

3.3 Extensible transition systems

We propose a simple, but novel, variant of generalised transition systems that we call *extensible transition systems*.

Definition 3.1 (Extensible transition system) An extensible transition system (XTS) is a quadruple $\langle \Gamma, \mathbb{C}, \rightarrow, F\rangle$ where $\mathbb{C}$ is a category with objects O and morphisms M such that $\langle \Gamma \times O, M, \rightarrow, F\rangle$ is a labelled transition system.

Abstract syntax.

$$\begin{array}{rll} \mathit{Expr} \ni e ::= & \texttt{plus}(e,e) \mid \texttt{throw}(e) \mid \texttt{stuck} \mid v & \text{Expressions} \\ \mathit{Val} \ni v ::= & n & \text{Values} \\ \mathit{ExcStat} \ni \varepsilon ::= & \text{OK} \mid \text{EXC}(v) & \text{Exception status} \end{array}$$

Transition relation.

$$\frac{(e, \text{OK}) \xrightarrow{\{\ldots\}} (e', \varepsilon)}{(\texttt{throw}(e), \text{OK}) \xrightarrow{\{\ldots\}} (\texttt{throw}(e'), \varepsilon)} \qquad \text{(MSOS-X-Throw1)}$$

$$\frac{}{(\texttt{throw}(v), \text{OK}) \xrightarrow{\{-\}} (\texttt{stuck}, \text{EXC}(v))} \qquad \text{(MSOS-X-Throw)}$$

$$\frac{(e_1, \text{OK}) \xrightarrow{\{\ldots\}} (e_1', \varepsilon)}{(\texttt{plus}(e_1, e_2), \text{OK}) \xrightarrow{\{\ldots\}} (\texttt{plus}(e_1', e_2), \varepsilon)} \qquad \text{(MSOS-X-Plus1)}$$

$$\frac{(e_2, \text{OK}) \xrightarrow{\{\ldots\}} (e_2', \varepsilon)}{(\texttt{plus}(n_1, e_2), \text{OK}) \xrightarrow{\{\ldots\}} (\texttt{plus}(n_1, e_2'), \varepsilon)} \qquad \text{(MSOS-X-Plus2)}$$

$$\frac{}{(\texttt{plus}(n_1, n_2), \text{OK}) \xrightarrow{\{-\}} (n_1 + n_2, \text{OK})} \qquad \text{(MSOS-X-Plus)}$$

Figure 3.1: An MSOS semantics for abrupt termination using configurations

A computation in an XTS is a computation in the underlying LTS such that its trace is a path in the category $\mathbb{C}$: a transition labelled m is followed immediately by a transition labelled m', the labels m, m' are required to be composable in $\mathbb{C}$.

This definition of XTS relies on *labelled transition systems* (LTS), rather than labelled *terminal* transition systems (LTTS). The difference between LTS and LTTS is in the distinction between *terminal* and *final* configurations. In the literature on automata theory (e.g., [HMU03]), it is common to regard *final* (or *accepting*) states as states that may have further transitions; i.e., they are computations that are allowed to terminate, but not required to. In contrast, according to Definition 2.2, terminal configurations have no possible further transitions.

Definition 3.2 (Labelled transition system) A labelled transition system (LTS) is a quadruple $\langle \Gamma, L, \rightarrow, F \rangle$ consisting of a set Γ of configurations γ, a set L of labels l, a ternary relation $\rightarrow \subseteq \Gamma \times L \times \Gamma$ of labelled transitions ($\langle \gamma, l, \gamma \rangle \in \rightarrow$ is written $\gamma \xrightarrow{l} \gamma$), and a set $F \subseteq \Gamma$ of final configurations.

A computation in an LTS (from γ_0) is a finite or infinite sequence of successive transitions $\gamma_i \xrightarrow{l_i} \gamma_{i+1}$ written $\gamma_0 \xrightarrow{l_1} \gamma_1 \xrightarrow{l_2} \ldots$, such that if the sequence is finite and has as final configuration γ_n we have $\gamma_n \in F$.

It is straightforward to translate an LTS into an LTTS, and thus it is also straightforward to give a derived notion of extensible *terminal* transition system, as we show in Section 3.6.

Apart from the difference between LTS and LTTS, the main difference between XTS and GTTS is that, in the labelled transition system underlying an XTS specification, configurations comprise of abstract syntax *and* objects of $\mathbb{C}$. This permits us to generalise the approach suggested in Section 3.2 such that we avoid the drawbacks of MSOS recalled in Section 3.1.

Being a variation on GTTS, transition relations can be defined using MSOS. But, since auxiliary entities make up an essential part of the structure of configurations, we adopt a notation that reflects the structure of configurations in rules explicitly, rather than encoding it into the names of morphisms.

3.4 Extensible SOS

We propose a variant of MSOS that we call *Extensible SOS* (XSOS). The main differences between MSOS and XSOS is that configurations in XSOS range over both abstract syntax and auxiliary entities.

3.4.1 Configurations

In XSOS, judgments have the form:

$$R \vdash \gamma_{/S} \xrightarrow{L} \gamma'_{/S'}$$

Here, each R, S, L are interpreted relative to the indexed product category $\mathbb{C} \triangleq \prod_{i \in I} \mathbb{C}_i$ in the underlying XTS such that:

- R ranges over the indexed product of all objects in the current configuration that are modelled by a discrete category in $\mathbb{C}$ (i.e., read-only entities);
- L ranges over the indexed product of all morphisms that are modelled by a free monoid in $\mathbb{C}$ (i.e., write-only entities); and
- S ranges over the indexed product of all other objects in the current configuration that are modelled by a preorder category in $\mathbb{C}$ (i.e., read-write entities).

We write $X[\mathbf{y}\ z]$ to denote the injection of z at the $\mathbf{y}$ index of an indexed product X. We also use the notation $X.\mathbf{y}$ to denote the projection of index $\mathbf{y}$ from an indexed product X.

While it is natural to model write-only entities by means of the free monoid, it is somewhat burdensome to propagate three different kinds of auxiliary entities. For the purpose of this thesis, it suffices to model free monoids as preorders without any loss of generality: recall that a free monoid is the set of all finite sequences A^* of zero or more elements for a monoid $\langle A, \cdot \rangle$, where '$\cdot$' is an associative binary operation $A \times A \to A$ for which there exists an identity element ι such that $\iota \cdot a = a \cdot \iota = a$ for any $a \in A$. Any free monoid A^* forms a preorder, using prefix-ordering as the notion of preorder. Thus, morphisms ranged over by L can be omitted in rules, and we restrict our attention to judgments of the form:

$$R \vdash \gamma_{/S} \to \gamma'_{/S'}$$

For example, consider the following rules for the `print` construct from Section 2.3.6 which rely on output by means of the label L:

$$\frac{R \vdash e_{/S} \xrightarrow{L} e'_{/S'}}{R \vdash \texttt{print}(e)_{/S} \xrightarrow{L} \texttt{print}(e')_{/S'}}$$

$$\frac{}{R \vdash \texttt{print}(v)_{/S} \xrightarrow{\textbf{out}'\,[v]} \texttt{unit}_{/S}}$$

For this thesis, we model such output using a preordered auxiliary entity instead:

$$\frac{R \vdash e_{/S} \to e'_{/S'}}{R \vdash \texttt{print}(e)_{/S} \to \texttt{print}(e')_{/S'}}$$

$$\frac{}{R \vdash \texttt{print}(v)_{/S[\textbf{out}\ vs]} \to \texttt{unit}_{/S[\textbf{out}\ (v::vs)]}}$$

Here, :: is the cons of lists.

In XSOS the small-step rules for `plus` become:

$$\frac{R \vdash e_{1/S} \to e'_{1/S'}}{R \vdash \texttt{plus}(e_1, e_2)_{/S} \to \texttt{plus}(e'_1, e_2)_{/S'}}$$

$$\frac{R \vdash e_{2/S} \to e'_{2/S'}}{R \vdash \texttt{plus}(n_1, e_2)_{/S} \to \texttt{plus}(n'_1, e_2)_{/S'}}$$

$$\frac{}{R \vdash \texttt{plus}(n_1, n_2)_{/S} \to n_1 + n_{2/S}}$$

XSOS also removes the need for big-step rules to explicitly mention label composition.[1] For example, recall the big-step MSOS rule for `plus` from Figure 2.16:

$$\frac{e_1 \xrightarrow{\ell_1} n_1 \qquad e_2 \xrightarrow{\ell_2} n_2}{\texttt{plus}(e_1, e_2) \xrightarrow{\ell_1 \fatsemi \ell_2} n_1 + n_2}$$

[1]This has both its merits and drawbacks: a merit is that XSOS rules are closer to familiar SOS rules. A drawback is that it loses some of the generality afforded by label composition: for example, the semantics of label composition could be more sophisticated than propagating entities in a left-to-right manner.

Its big-step XSOS counterpart is:

$$\frac{R \vdash e_{1/S} \Rightarrow n_{1/S'} \qquad R \vdash e_{2/S'} \Rightarrow n_{2/S''}}{R \vdash \mathtt{plus}(e_1, e_2)_{/S} \Rightarrow n_1 + n_{2/S''}}$$

Here, auxiliary entities are propagated syntactically, similar to Plotkin's SOS rules [Plo81].

3.4.2 Comparison of approaches

Table 3.1 compares SOS, MSOS, and XSOS. Each rule propagates (at least) three different auxiliary entities: environments (ρ), stores (σ), and printed values (lists of v's). The first row illustrates the semantic rules for a construct that uses environments (i.e., a 'read-only' entity in MSOS terminology); the second row, rules specifying a construct that uses imperative stores ('read-write'); and in the last row, semantic rules for a construct that produces observable output ('write-only').

Whereas SOS rules propagate *only* explicitly-mentioned auxiliary entities, MSOS and XSOS rules propagate an *open-ended* set of auxiliary entities. The means of propagation in MSOS and XSOS differs slightly: whereas MSOS propagates all unmentioned entities using a single meta-variable, namely a label which ranges over all auxiliary entities, XSOS rules are more verbose and rely on two distinct meta-variables for the two different kinds of auxiliary entities; R for discrete entities, and S for preordered auxiliary entities.

Comparing the first and last column of the figure illustrates an intentional coincidence between how judgments and rules in SOS and XSOS are written and read. The main differences between XSOS and SOS are: XSOS propagates an open-ended set of auxiliary entites; and observable output is represented by explicit concatenation onto the output stream. Thus, even though neither of the XSOS rules in the first and second row of Table 3.1 explicitly mention the **out** auxiliary entity, it is implicitly propagated, via the S, S' variables.

3.4.3 Abbreviated XSOS

In order to make rules and specifications more readable, we adopt and adapt ideas from Implicitly-Modular SOS [MN09] (I-MSOS). Mosses and New define formally the correspondence between I-MSOS rules and MSOS. For the purpose of this thesis it suffices with an informal description of how we abbreviate XSOS rules and what we mean by the abbreviations. When reasoning about XSOS rules and specifications we always use their unabbreviated forms.

We abbreviate XSOS rules by omitting meta-variables R and S in rules, such that a rule like:

$$\frac{e_1 \to e_1' \qquad e_2 \to e_2' \qquad \ldots \qquad e_n \to e'}{e \to e'}$$

	SOS	Modular SOS	Extensible SOS
Read-only	$\dfrac{\rho'[x \mapsto v] \vdash (e,\sigma) \xrightarrow{l} (e',\sigma')}{\rho \vdash (\langle x,e,\rho'\rangle\ v,\sigma) \xrightarrow{l} (\langle x,e',\rho'\rangle\ v,\sigma')}$	$\dfrac{e \xrightarrow{\{\mathbf{env}\ \rho'[x\mapsto v],\ldots\}} e'}{\langle x,e,\rho'\rangle\ v \xrightarrow{\{\mathbf{env}\ \rho,\ldots\}} \langle x,e',\rho'\rangle\ v}$	$\dfrac{R[\mathbf{env}\ \rho'[x \mapsto v]] \vdash e_{/S} \to e'_{/S'}}{R[\mathbf{env}\ \rho] \vdash \langle x,e,\rho'\rangle\ v_{/S} \to \langle x,e',\rho'\rangle\ v_{/S'}}$
	$\dfrac{}{\rho \vdash (\lambda x.e,\sigma) \to (\langle x,e,\rho\rangle,\sigma)}$	$\dfrac{}{\lambda x.e \xrightarrow{\{\mathbf{env}=\rho,-\}} \langle x,e,\rho\rangle}$	$\dfrac{}{R[\mathbf{env}\ \rho] \vdash \lambda x.e_{/S} \to \langle x,e,\rho\rangle_{/S}}$
Read-write	$\dfrac{\rho \vdash (e,\sigma) \xrightarrow{l} (e',\sigma')}{\rho \vdash (\texttt{ref}(e),\sigma) \xrightarrow{l} (\texttt{ref}(e'),\sigma')}$	$\dfrac{e \xrightarrow{\{\ldots\}} e'}{\texttt{ref}(e) \xrightarrow{\{\ldots\}} \texttt{ref}(e')}$	$\dfrac{R \vdash e_{/S} \to e'_{/S'}}{R \vdash \texttt{ref}(e)_{/S} \to \texttt{ref}(e')_{/S'}}$
	$\dfrac{r \notin \mathrm{dom}(\sigma)}{\rho \vdash (\texttt{ref}(v),\sigma) \to (r,\sigma[r \mapsto v])}$	$\dfrac{r \notin \mathrm{dom}(\sigma)}{\texttt{ref}(v) \xrightarrow{\{\mathbf{sto}=\sigma,\mathbf{sto}'=\sigma[r\mapsto v],-\}} r}$	$\dfrac{r \notin \mathrm{dom}(\sigma)}{R \vdash \texttt{ref}(v)_{/S[\mathbf{sto}\ \sigma]} \to r_{/S[\mathbf{sto}\ \sigma[r\mapsto v]]}}$
Write-only	$\dfrac{\rho \vdash (e,\sigma) \xrightarrow{l} (e',\sigma')}{\rho \vdash (\texttt{print}(e),\sigma) \xrightarrow{l} \texttt{print}(e')}$	$\dfrac{e \xrightarrow{\{\ldots\}} e'}{\texttt{print}(e) \xrightarrow{\{\ldots\}} \texttt{print}(e')}$	$\dfrac{R \vdash e_{/S} \to e'_{/S'}}{R \vdash \texttt{print}(e)_{/S} \to \texttt{print}(e')_{/S'}}$
	$\dfrac{}{\rho \vdash (\texttt{print}(v),\sigma) \xrightarrow{[v]} (\texttt{unit},\sigma)}$	$\dfrac{}{\texttt{print}(v) \xrightarrow{\{\mathbf{out}'=[v],-\}} \texttt{unit}}$	$\dfrac{}{R \vdash \texttt{print}(v)_{/S[\mathbf{out}\ vs]} \to \texttt{unit}_{/S[\mathbf{out}\ (v::vs)]}}$

Table 3.1: Comparison of small-step SOS variants

abbreviates an XSOS rule:

$$\frac{R \vdash e_{1/S} \rightarrow e'_{1/S_2} \qquad R \vdash e_{2/S_2} \rightarrow e'_{2/S_3} \qquad \dots \qquad R \vdash e_{n/S_n} \rightarrow e'_{/S'}}{R \vdash e_{/S} \rightarrow e'_{/S'}}$$

Similarly, rules that only refer to certain auxiliary entities, such as the following rule for application:

$$\frac{\mathbf{env}\ \rho \vdash e_1 \Rightarrow \langle x, e, \rho' \rangle \qquad \mathbf{env}\ \rho \vdash e_2 \Rightarrow v_2 \qquad \mathbf{env}\ \rho'[x \mapsto v_2] \vdash e \Rightarrow v}{\mathbf{env}\ \rho \vdash e_1\ e_2 \Rightarrow v}$$

abbreviates a rule where ρ is injected into the indexed product of readable auxiliary entities R at the **env** index, and all other entities are propagated as described above, i.e.:

$$\frac{\begin{array}{c} R[\mathbf{env}\ \rho] \vdash e_{1/S} \Rightarrow \langle x, e, \rho' \rangle_{/S_2} \\ R[\mathbf{env}\ \rho] \vdash e_{2/S_2} \Rightarrow v_{2/S_3} \qquad R[\mathbf{env}\ \rho'[x \mapsto v_2]] \vdash e_{/S_3} \Rightarrow v_{/S'} \end{array}}{R[\mathbf{env}\ \rho] \vdash e_1\ e_{2/S} \Rightarrow v_{/S'}}$$

We use these conventions for making rules more perspicuous, but use explicit XSOS judgments in definitions and proofs (or otherwise explicitly state what we are leaving out).

Table 3.2 gives a comparison of abbreviated XSOS rules and explicit XSOS rules, and illustrates how any auxiliary entities that are not explicitly used by in the rule are omitted. For example, for read-only entities, using abbreviated XSOS both the variables R and S are omitted and propagated as described above.

3.4.4 MSOS vs. XSOS

Adopting the syntactic left-to-right propagation strategy inherent to the big-step XSOS rule above has both pros and cons: one of the pros is that it brings XSOS rules closer to traditional SOS and natural semantics rules by removing the need to explicitly mention label composition operators in rules; a con is that this precludes more sophisticated notions of composition that label composition in MSOS supports (although the author is not aware of any literature that actually utilises this flexibility). We emphasise that XTS semantics could also be given using MSOS rules, so this is a shortcoming of XSOS, and not inherent to the underlying notion of extensible transition system. The focus of this thesis is deterministic programming language semantics, so the extra generality afforded by big-step MSOS over big-step XSOS is not relevant for the purpose of this thesis.

3.5 Extensible specifications for abrupt termination

It is straightforward to model the notion of abrupt termination from Section 3.2 in XTS in an extensible way that scales to both small-step and big-step semantics.

	Explicit XSOS	Abbreviated XSOS
Read-only	$\dfrac{R[\mathbf{env}\ \rho'[x \mapsto v]] \vdash e_{/S} \to e'_{/S'}}{R[\mathbf{env}\ \rho] \vdash \langle x, e, \rho' \rangle\ v_{/S} \to \langle x, e', \rho' \rangle\ v_{/S'}}$	$\dfrac{\mathbf{env}\ \rho'[x \mapsto v] \vdash e \to e'}{\mathbf{env}\ \rho \vdash \langle x, e, \rho' \rangle\ v \to \langle x, e', \rho' \rangle\ v}$
	$\dfrac{}{R[\mathbf{env}\ \rho] \vdash \lambda x.e_{/S} \to \langle x, e, \rho \rangle_{/S}}$	$\dfrac{}{\mathbf{env}\ \rho \vdash \lambda x.e \to \langle x, e, \rho \rangle}$
Read-write	$\dfrac{R \vdash e_{/S} \to e'_{/S'}}{R \vdash \mathtt{ref}(e)_{/S} \to \mathtt{ref}(e')_{/S'}}$	$\dfrac{e \to e'}{\mathtt{ref}(e) \to \mathtt{ref}(e')}$
	$\dfrac{r \notin \mathrm{dom}(\sigma)}{R \vdash \mathtt{ref}(v)_{/S[\mathbf{sto}\ \sigma]} \to r_{/S[\mathbf{sto}\ \sigma[r \mapsto v]]}}$	$\dfrac{r \notin \mathrm{dom}(\sigma)}{\mathtt{ref}(v)_{/\mathbf{sto}\ \sigma} \to r_{/\mathbf{sto}\ \sigma[r \mapsto v]}}$
Write-only	$\dfrac{R \vdash e_{/S} \to e'_{/S'}}{R \vdash \mathtt{print}(e)_{/S} \to \mathtt{print}(e')_{/S'}}$	$\dfrac{e \to e'}{\mathtt{print}(e) \to \mathtt{print}(e')}$
	$\dfrac{}{R \vdash \mathtt{print}(v)_{/S[\mathbf{out}\ vs]} \to \mathtt{unit}_{/S[\mathbf{out}\ v::vs]}}$	$\dfrac{}{\mathtt{print}(v)_{/\mathbf{out}\ vs} \to \mathtt{unit}_{/\mathbf{out}\ v::vs}}$

Table 3.2: Comparison of explicit and abbreviated small-step XSOS rules

3.5.1 Small-step XSOS for abrupt termination

Figure 3.2 gives a specification of `throw`ing and `catch`ing exceptions. The specification consists of:

- an abstract syntax specification, i.e., a term signature $\Sigma_{\mathtt{throw}}$;
- an indexed product category, $\mathbb{C}_{\mathtt{throw}}$, with an index **exc** comprising an abrupt termination category ($\mathbb{C}^{\mathrm{AT}}$) that we will shortly be describing, and an index **env** comprising a discrete category ($\mathbb{C}^{\mathrm{DISCRETE}}$) whose objects is the set of environments *Env*;
- a set of rules that define a predicate $\mathrm{Q} \subseteq \Gamma \times O$ for distinguishing a set of final configurations; and
- a set of rules $D_{\mathtt{throw}}$ defining a relation $\rightsquigarrow \subseteq \Gamma \times O \times \Gamma \times O$.

The specification names each constituent part so that we can refer to and combine specifications. It constitutes what we call an *extensible rule specification*, defined in Definition 3.3.

Definition 3.3 (Extensible rule specification) An *extensible rule specification* is a tuple

$$\left\langle \Sigma, \prod_{j \in J} \mathbb{C}_j, D, Q \right\rangle$$

Abstract syntax ($\Sigma_{\texttt{throw}}$).

$$\begin{aligned} \mathit{Expr} \ni e &::= \texttt{throw}(e) \mid \texttt{catch}(e,x,e) \mid \texttt{bind}(x,v,e) \mid \texttt{stuck} \\ v &\in \mathit{Val} \\ \mathit{ExcStat} \ni \varepsilon &::= \mathrm{OK} \mid \mathrm{EXC}(v) \\ \mathit{Env} &\triangleq \mathit{Var} \xrightarrow{\text{fin}} \mathit{Val} \end{aligned}$$

Auxiliary entities ($\mathbb{C}_{\texttt{throw}}$).

$$\mathbb{C}_{\texttt{throw}} \triangleq \left\{\textbf{exc} : \mathbb{C}^{\mathrm{AT}}(\mathit{Val})\ ;\ \textbf{env} : \mathbb{C}^{\mathrm{DISCRETE}}(\mathit{Env})\right\}$$

Final configurations ($Q_{\texttt{throw}}$).

$$\frac{}{\mathtt{Q}(v,S)}$$

$$\frac{}{\mathtt{Q}(e,S[\textbf{exc}\ \mathrm{EXC}(v)])}$$

Transition relation ($D_{\texttt{throw}}$).

$$\frac{e \to e'}{\texttt{throw}(e) \to \texttt{throw}(e')} \quad \text{(XSOS-Throw1)}$$

$$\frac{}{\texttt{throw}(v)_{/\textbf{exc}\ \mathrm{OK}} \to \texttt{stuck}_{/\textbf{exc}\ \mathrm{EXC}(v)}} \quad \text{(XSOS-Throw)}$$

$$\frac{e_{1/\textbf{exc}\ \mathrm{OK}} \to e'_{1/\textbf{exc}\ \mathrm{OK}}}{\texttt{catch}(e_1,x,e_2)_{/\textbf{exc}\ \mathrm{OK}} \to \texttt{catch}(e'_1,x,e_2)_{/\textbf{exc}\ \mathrm{OK}}} \quad \text{(XSOS-Catch1)}$$

$$\frac{}{\texttt{catch}(v,x,e_2) \to v} \quad \text{(XSOS-CatchV)}$$

$$\frac{e_{1/\textbf{exc}\ \mathrm{OK}} \to e'_{1/\textbf{exc}\ \mathrm{EXC}(v)}}{\texttt{catch}(e_1,x,e_2)_{/\textbf{exc}\ \mathrm{OK}} \to \texttt{bind}(x,v,e_2)_{/\textbf{exc}\ \mathrm{OK}}} \quad \text{(XSOS-CatchE)}$$

$$\frac{\textbf{env}\ \rho[x \mapsto v] \vdash e \to e'}{\textbf{env}\ \rho \vdash \texttt{bind}(x,v,e) \to \texttt{bind}(x,v,e')} \quad \text{(XSOS-Bind1)}$$

$$\frac{}{\texttt{bind}(x,v,v') \to v'} \quad \text{(XSOS-Bind)}$$

Figure 3.2: XTS for exception handling

where:

- Σ is a signature of terms ranged over by t;
- $\prod_{j \in J} \mathbb{C}_j$ is a product category with objects O, ranged over by o, and morphisms M, ranged over by m where the structure of objects and morphisms may depend on Σ;
- D is a set of rules; and
- Q is a set of rules that defines a predicate $\mathtt{Q}$ on pairs consisting of a term of signature Σ and an object in O.

Set-based operations (union, intersection, etc.) are lifted to extensible rule specifications by applying them pointwise to the constituent sets of the specification. For example, the union $rs_1 \cup rs_2$ of two rule specifications $rs_1 \triangleq \langle \Sigma_1, \prod_{j \in J_1} \mathbb{C}_j, D_1, Q_1 \rangle$ and $rs_2 \triangleq \langle \Sigma_2, \prod_{j \in J_2} \mathbb{C}_j, D_2, Q_2 \rangle$ is given by the tuple $\langle \Sigma_1 \cup \Sigma_2, \prod_{j \in J_1 \cup J_2} \mathbb{C}_j, D_1 \cup D_2, Q_1 \cup Q_2 \rangle$. This thesis makes use of extensible rule specifications to specify constructs independently and combine them as described above.

In contrast to the rules for `catch` considered in Section 2.2.5 which used λ application for binding, the `catch` construct in Figure 3.2 uses a separate `bind` construct instead. The motivation for using `bind` is to decouple the semantics of `catch` from the λ-calculus.

The product category in Figure 3.2 uses a special notion of category that is distinguished from the traditional categories used in MSOS, $\mathbb{C}^{\mathrm{AT}}$, the *abrupt termination category*. Figure 3.3 defines this category, which has a single OK object, indicating that no exception has been thrown, and as many objects as there are values, indicating that an exception recording that value has been thrown. There is a morphism from the OK object to either of the EXC(v) objects, but no morphisms in the other direction.

3.5.2 An abruptly terminating example

Using the category for abrupt termination, it is no longer necessary to wrap possibly-abruptly terminating expressions in top-level program handlers in order to abruptly terminate. For example, the expression $\mathtt{plus}'(\mathtt{throw}(0), \omega)$ from Section 3.1 now abruptly terminates. Its computation is summarised by the following transition:

$$R \vdash \mathtt{plus}'(\mathtt{throw}(0), \omega)_{/S[\mathbf{exc}\ \mathrm{OK}]} \to \mathtt{plus}'(\mathtt{stuck}, \omega)_{/S[\mathbf{exc}\ \mathrm{EXC}(0)]}$$

Here, the configuration for the judgment $R \vdash \mathtt{plus}'(\mathtt{stuck}, \omega)_{/S[\mathbf{exc}\ \mathrm{EXC}(0)]}$ is a final configuration, so we choose to terminate the computation (even though the XTS technically allows further transitions, since Definition 3.1 does not insist that final states do not have further transitions). In Section 3.6 we show that it is straightforward to derive a transition system that terminates and has no further transitions when the transition system enters a final configuration.

Collection of objects.

$$O \triangleq \{\text{OK}\} \cup \{\text{EXC}(v) \mid v \in \mathit{Val}\}$$

Collection of morphisms.

$$M \triangleq \{\text{OK} \rightarrow \text{EXC}(v) \mid v \in \mathit{Val}\} \cup \{\mathit{Id}_{\text{OK}}\} \cup \{\mathit{Id}_{\text{EXC}(v)} \mid v \in \mathit{Val}\}$$

Composition operator.
A total function '$\fatsemi$' which, for all morphisms $m_1 : O_1 \rightarrow O_2$, $m_2 : O_2 \rightarrow O_3$ produces a morphism $m : O_1 \rightarrow O_3$:

$$m_1 \fatsemi m_2 = m$$

Identity morphisms.

$$\mathit{Id}_{\text{OK}} \triangleq \text{OK} \mapsto \text{OK}$$

$$\mathit{Id}_{\text{EXC}(v)} \triangleq \text{EXC}(v) \mapsto \text{EXC}(v)$$

Figure 3.3: The category $\mathbb{C}^{\text{AT}}$ for abrupt termination

3.5.3 Pretty-big-step XSOS for simple arithmetic expressions and exception handling

Section 2.5 recalled how pretty-big-step semantics is a variant of natural semantics that allows abrupt termination and divergence to be propagated without duplicate premises. Following Charguéraud [Cha13], introducing abrupt termination or divergence in pretty-big-step semantics still involves adding so-called "abort" rules for all existing constructs in a language. Here, we show how to avoid modifying or introducing new rules for existing constructs using pretty-big-step XSOS rules.

Simple arithmetic expressions. Figure 3.4 gives an XSOS specification in pretty-big-step style for simple arithmetic expressions. The rules are in the pretty-big-step style since each fully evaluate a single sub-term before continuing with the rest of the computation. Unlike the pretty-big-step semantics recalled in Section 2.5, the pretty-big-step XSOS rules in the figure do not rely on auxiliary syntax for intermediate expressions. Recall that the motivation for relying on auxiliary syntax for intermediate expressions was to avoid infinite derivations in the coinductive interpretation for computations that are supposed to diverge. For simple arithmetic expressions there is no danger of this happening, since numeric terms and values are distinct. But the pitfall arises in connection with abrupt termination, as we shortly show.

Exception handling. Figure 3.5 gives an abbreviated XSOS specification in pretty-big-step style for throwing and catching exceptions.

Abstract syntax ($\Sigma_{\mathtt{plus}}$).

$$\begin{aligned} \mathit{Expr} \ni e &::= \mathtt{plus}(e, e) \mid \mathtt{num}(n) \mid v \\ \mathit{Val} \ni v &::= n \\ n \in \mathbb{N} &\triangleq \{0, 1, 2, \ldots\} \end{aligned}$$

Auxiliary entities ($\mathbb{C}_{\mathtt{plus}}$).

$$\mathbb{C}_{\mathtt{plus}} \triangleq \mathbb{1}$$

Final configurations ($Q_{\mathtt{plus}}$).

$$\dfrac{}{\mathtt{Q}(v, S)}$$

Evaluation relation ($D^{\mathtt{PBS}}_{\mathtt{plus}}$).

$$\dfrac{e_1 \Downarrow e'_1 \qquad \mathtt{plus}(e'_1, e_2) \Downarrow e'}{\mathtt{plus}(e_1, e_2) \Downarrow e'} \qquad \text{(XSOS-PB-Plus1)}$$

$$\dfrac{e_2 \Downarrow e'_2 \qquad \mathtt{plus}(n_1, e'_2) \Downarrow e'}{\mathtt{plus}(n_1, e_2) \Downarrow e'} \qquad \text{(XSOS-PB-Plus2)}$$

$$\dfrac{}{\mathtt{plus}(n_1, n_2) \Downarrow n_1 + n_2} \qquad \text{(XSOS-PB-Plus)}$$

$$\dfrac{}{\mathtt{num}(n) \Downarrow n} \qquad \text{(XSOS-PB-Num)}$$

Figure 3.4: Abbreviated pretty-big-step XSOS rules for simple arithmetic expressions

Figure 3.5 introduces a rule (XSOS-PB-AT-Refl) for propagating abrupt termination in all constructs. Consider the language given by taking the union of the big-step XSOS specifications in Figures 3.4 and 3.5. Using the coinductive interpretation of the rules for this language, it is possible to prove that abruptly terminated terms diverge; e.g.:

$$\dfrac{\mathtt{stuck}_{/\mathbf{exc}\ \mathtt{err}(0)} \Downarrow \mathtt{stuck}_{/\mathbf{exc}\ \mathtt{err}(0)} \qquad \dfrac{\vdots}{\mathtt{plus}(\mathtt{stuck}, \mathtt{num}(2))_{/\mathbf{exc}\ \mathtt{err}(0)} \Downarrow 42_{/\mathbf{exc}\ \mathtt{err}(0)}}}{\mathtt{plus}(\mathtt{stuck}, \mathtt{num}(2))_{/\mathbf{exc}\ \mathtt{err}(0)} \Downarrow 42_{/\mathbf{exc}\ \mathtt{err}(0)}}$$

$$\dfrac{\mathtt{throw}(0)_{/\mathbf{exc}\ \mathtt{OK}} \Downarrow \mathtt{stuck}_{/\mathbf{exc}\ \mathtt{err}(0)} \qquad \vdots}{\mathtt{plus}(\mathtt{throw}(0), \mathtt{num}(2))_{\mathbf{exc}\ \mathtt{OK}} \Downarrow 42_{/\mathbf{exc}\ \mathtt{err}(0)}}$$

In order to avoid this issue, we adopt another convention for abbreviated XSOS rules;

Abstract syntax ($\Sigma_{\texttt{throw}}$). See Figure 3.2.
Auxiliary entities ($\mathbb{C}_{\texttt{throw}}$). See Figure 3.2.
Final configurations ($Q_{\texttt{throw}}$). See Figure 3.2.
Evaluation relation ($D^{\texttt{PBS}}_{\texttt{throw}}$).

$$\frac{e \Downarrow e' \qquad \texttt{throw}(e') \Downarrow e''}{\texttt{throw}(e) \Downarrow e''} \qquad \text{(XSOS-PB-Throw1)}$$

$$\frac{}{\texttt{throw}(v)_{/\textbf{exc}\ \text{OK}} \Downarrow \texttt{stuck}_{/\textbf{exc}\ \text{EXC}(v)}} \qquad \text{(XSOS-PB-Throw)}$$

$$\frac{e_{1/\textbf{exc}\ \text{OK}} \Downarrow e'_{1/\textbf{exc}\ \text{OK}} \qquad \texttt{catch}(e'_1,x,e_2)_{/\textbf{exc}\ \text{OK}} \Rightarrow e'_{/\textbf{exc}\ \varepsilon}}{\texttt{catch}(e_1,x,e_2)_{/\textbf{exc}\ \text{OK}} \Downarrow e'_{/\textbf{exc}\ \varepsilon}} \qquad \text{(XSOS-PB-Catch1)}$$

$$\frac{}{\texttt{catch}(v,x,e_2) \Downarrow v} \qquad \text{(XSOS-PB-CatchV)}$$

$$\frac{e_{1/\textbf{exc}\ \text{OK}} \Downarrow e'_{1/\textbf{exc}\ \text{EXC}(v)} \qquad \texttt{bind}(x,v,e_2)_{/\textbf{exc}\ \text{OK}} \Downarrow e'_{/\textbf{exc}\ \varepsilon}}{\texttt{catch}(e_1,x,e_2)_{/\textbf{exc}\ \text{OK}} \Downarrow e'_{/\textbf{exc}\ \varepsilon}} \qquad \text{(XSOS-PB-CatchE)}$$

$$\frac{\textbf{env}\ \rho[x \mapsto v] \vdash e \Downarrow e' \qquad \textbf{env}\ \rho \vdash \texttt{bind}(x,v,e') \Downarrow e''}{\textbf{env}\ \rho \vdash \texttt{bind}(x,v,e) \Downarrow e''} \qquad \text{(XSOS-PB-Bind1)}$$

$$\frac{}{\texttt{bind}(x,v,v') \Downarrow v'} \qquad \text{(XSOS-PB-Bind)}$$

$$\frac{\text{Q}(e,S)}{R \vdash e_{/S} \Downarrow e_{/S}} \qquad \text{(XSOS-PB-AT-Refl)}$$

Figure 3.5: Abbreviated pretty-big-step XSOS rules for abrupt termination

Convention 3.4 An abbreviated XSOS rule with premises:

$$\frac{e_1 \to e'_1 \qquad e_2 \to e'_2 \qquad \ldots \qquad e_n \to e'}{e \to e'}$$

abbreviates a rule:

$$\frac{\begin{array}{c}\neg\text{Q}(e,S)\\ R \vdash e_{1/S} \to e'_{1/S_2} \qquad R \vdash e_{2/S_2} \to e'_{2/S_3} \qquad \ldots \qquad R \vdash e_{n/S_n} \to e'_{/S'}\end{array}}{R \vdash e_{/S} \to e'_{/S'}}$$

Convention 3.5 An abbreviated XSOS simple rule:

$$\frac{}{e \to e'}$$

abbreviates a rule:

$$\frac{\neg\text{Q}(e,S)}{R \vdash e_{/S} \to e'_{/S}}$$

These conventions ensure that abbreviated rules always have a premise ensuring that rules do not match for final configurations.

3.5.4 An abruptly terminating example

The term $\texttt{plus}(\texttt{throw}(\texttt{num}(0)),\texttt{num}(1))$ has the following derivation tree:

$$\dfrac{\dfrac{\texttt{num}(0)_{/\mathbf{exc}\ \mathrm{OK}} \Downarrow 0_{/\mathbf{exc}\ \mathrm{OK}}}{\texttt{throw}(\texttt{num}(0))_{/\mathbf{exc}\ \mathrm{OK}} \Downarrow \texttt{stuck}_{/\mathbf{exc}\ \mathrm{EXC}(0)}} \quad \begin{array}{c} \texttt{plus}(\texttt{stuck},\texttt{num}(1))_{/\mathbf{exc}\ \mathrm{EXC}(0)} \Downarrow \texttt{plus}(\texttt{stuck},\texttt{num}(1))_{/\mathbf{exc}\ \mathrm{EXC}(0)} \\ \vdots \end{array}}{\texttt{plus}(\texttt{throw}(0),\texttt{num}(1))_{/\mathbf{exc}\ \mathrm{OK}} \Downarrow \texttt{plus}(\texttt{stuck},\texttt{num}(1))_{/\mathbf{exc}\ \mathrm{EXC}(0)}}$$

Wrapping the term in a `catch` expression correctly handles the exception; i.e., the following judgment holds:

$$\mathbf{env}\ \emptyset \vdash \texttt{catch}(\texttt{plus}(\texttt{throw}(0),\texttt{num}(1)),x,x)_{/\mathbf{exc}\ \mathrm{OK}} \Downarrow 0_{/\mathbf{exc}\ \mathrm{OK}}$$

These examples show that extensible transition system semantics supports extensible specification of abrupt termination. In Chapter 4 we show how *refocusing* the small-step semantics in Figure 3.2 automatically derives the pretty-big-step XSOS specifications in Figure 3.5, and that they are equivalent. First, we consider the relationship between extensible transition systems and alternative variants of transition systems from the literature.

3.6 Correspondence with other transition system variants

Extensible transition systems are a simple but novel variant of Mosses' [Mos04] generalised transition systems. Proposition 3.8 establishes the correspondence between generalised terminal transition systems (GTTS) and extensible terminal transition systems (XTTS). Proposition 3.6 shows how to embed any XTTS in an XTS.

Proposition 3.6 For each XTS $\langle \Gamma, \mathbb{C}, \rightarrow, F \rangle$, an LTTS $\langle \Gamma^\flat, L^\flat, \rightarrow^\flat, T^\flat \rangle$ can be constructed such that for each computation of the LTTS there is a computation of the XTS with the same trace.

Proof. Let $\rightarrow^\flat \triangleq \rightarrow \setminus \rightarrow^F$ where $\rightarrow^F$ is the set of all transitions in the XTS $(\gamma, o) \rightarrow (\gamma', o')$ such that $(\gamma, o) \in F$. This embeds every computation in the LTTS into XTS. This completes the construction. $\square$

Definition 3.7 An *extensible terminal transition system* (XTTS) is an extensible transition system that corresponds to an LTTS as described in Proposition 3.6.

Proposition 3.8 For each GTTS $\langle \Gamma, \mathbb{C}, \rightarrow, T \rangle$, an XTTS $\langle \Gamma^\sharp, \mathbb{C}^\sharp, \rightarrow^\sharp, F^\sharp \rangle$ can be constructed such that for each computation of the GTTS there is a computation of the XTTS with the same trace and vice versa.

Proof. The proof is analogous to Mosses' [Mos04, Proposition 3].

Let the set of objects in $\mathbb{C}$ be given by O, and the set of morphisms given by M. Define $\Gamma^\sharp \triangleq \Gamma \times O$, and $F^\sharp \triangleq T \times O$. Then let $\gamma \xrightarrow{m} \gamma'$ hold in the GTTS iff $\gamma \times o \xrightarrow{m} \gamma' \times o$ holds in the XTTS. Now every computation $\gamma \xrightarrow{m_1} \gamma_1 \xrightarrow{m_2} \cdots$ in the GTTS has a corresponding computation $(\gamma, o) \xrightarrow{m_1} (\gamma_1, o_1) \xrightarrow{m_2} \cdots$ in the XTTS, such that if the GTTS terminates with $\gamma_n \in T$, then the XTTS has a corresponding trace with final configuration $\gamma_n \in F^\sharp$.

Conversely, suppose that $(\gamma, o) \xrightarrow{m_1} (\gamma_1, o_1) \xrightarrow{m_2} \cdots$ is a computation in the XTTS. Then $o = \text{dom}(m_1)$, and for $i \geq 1$, $o_i = \text{cod}(m_i) = \text{dom}(m_{i+1})$. Hence m_i and m_{i+1} are composable for all $i \geq 1$. Moreover, for each computation in the XTTS which terminates with $(\gamma, o) \in F^\sharp$, we always have $\gamma_n \in T$. Hence $\gamma \xrightarrow{m_1} \gamma_1 \xrightarrow{m_2} \cdots$ is a computation in the GTTS, and the traces of the two computations are the same. $\square$

3.7 Assessment and related work

We have analysed the issue with translating the technique for abrupt termination from small-step MSOS to big-step MSOS, and proposed extensible transition systems as a solution to the problem. Section 3.5 shows that XTS and XSOS support modular abrupt termination, and Section 3.6 relates XTS to other notions of transition systems in the literature. In this section we assess the proposed solution, and compare it to other frameworks and lines of research in the literature.

3.7.1 Relationship with previous approaches

The notion of abrupt termination presented in this chapter is a straightforward adaptation of Klin and Mosses' technique for abrupt termination in MSOS [Mos04]. In traditional approaches to abrupt termination, exceptions are modelled as a set of terms that are distinct from values. As such, computations produce either a value or an exception, which are naturally modelled by a sum-type, reflected, e.g., in the notion of outcome we recalled in connection with natural semantics in Section 2.4.5, where the set of outcomes correspond to the sum:

$$\mathit{Val} + \{\text{EXC}(v)\}$$

The encoding of abrupt termination presented in this chapter can be seen to emulate this sum-type, in that terminal computations always produce either a configuration which corresponds to a value, or a configuration that is abruptly terminated. Unlike the sum-type above, our encoding also records the structure of the term for which abrupt termination occurred, rather than collapsing the entire term to an exception, as is the case with the traditional approaches in the natural semantics and pretty-big-step specifications that we recalled and illustrated in Sections 2.4 and 2.5.

The fact that our encoding of abrupt termination records the structure of the abruptly terminated term makes it somewhat close in spirit to reduction semantics, where an abruptly terminated term that makes it to the top-level contains a term that represents the context in which abrupt termination occurred, such that the "hole" of the context is given by the `stuck` term. For the variant of exceptions and exception handling that this chapter considered, the extra information recorded in this term is redundant: the `catch` construct discards the abruptly terminated term. But for more interesting notions of control, having a retentive notion of abrupt termination that records the context in which abrupt termination occurred is useful. For example, Sculthorpe et al. [STM16] exploit this fact in order to give an MSOS for delimited continuations. In Section 4.6 we show how our encoding of abrupt termination enables a translation that is both straightforward and correct-by-construction of Sculthorpe et al.'s semantics into pretty-big-step semantics.

Besides being useful for giving semantics based on continuations, our retentive notion of abrupt termination could conceivably also be useful for error-reporting, since the structure of the term in which an error occurs is recorded in the abruptly terminated configuration.

This chapter proposed to use XSOS in place of MSOS, but the difference between XSOS and MSOS is mainly cosmetic: morphisms in generalised transition systems do record the necessary information required to give a similar notion of abrupt termination. Even though configurations in MSOS by convention consist of pure abstract syntax, one could record in one's transition system a set of terminal morphisms along with a set of terminal configurations. This would enable one to distinguish if evaluation should continue on a par with extensible transition systems. For this thesis we opted for a re-founding of MSOS because the notion of underlying transition system is simpler, and translates to big-step semantics in a straightforward manner.

Madlener et al. [MSvE11] have shown how to give and work with MSOS rules formally in Coq. The encoding relies on sophisticated use of Coq's type classes, but allows language features to be defined independently and combined and reasoned about. They formalise labels as morphisms between objects, where the objects themselves represent the auxiliary entities, similar to how extensible transition system semantics emphasises the use of objects instead of morphisms for providing access to auxiliary entities in rules. Later chapters of this thesis use Coq encodings of XSOS rules based on a more naive encoding that supports copy-paste reuse in Coq.

3.7.2 Other notions of modularity and extensibility

Giving modular and extensible specifications is a subject of extensive study in the literature. Modular SOS, which this thesis extends, was itself incepted as a means of bringing to SOS what monads [Mog91, CM93, Wad92] and monad transformers [LHJ95] bring to denotational semantics.

Other authors have investigated other means of giving modular specifications in SOS. Turi and Plotkin [TP97] investigated a class of operational semantics that is particularly well-behaved and which has corresponding denotational and categorical semantics. Jaskelioff et al. [JGH11] noticed that the general structure observed and described by Turi and Plotkin can be used to give *modular operational semantics*, where both syntax and semantics can be separately defined and combined. Unlike the syntactic flavour of modularity that is inherent to MSOS, modular operational semantics has a more semantic flavour that relies on sophisticated mathematical machinery.

Cartwright and Felleisen [CF94] investigate an approach to giving extensible denotational semantics. Their approach relies on having distinct domains for different kinds of behaviour, where each of these domains are comprised of extensible sum-types.

4 Refocusing in Extensible SOS

Contents

The last chapter showed how XSOS enables giving either extensible small-step or big-step specifications. This reduces the effort involved in specifying and maintaining specifications in either style as a language evolves. But for some applications it is useful to give and maintain specifications in *both* styles. For example, CompCert [Ler06] uses both big-step and small-step semantics for its correctness proofs; certain financial contract-based languages [PJES00, BBE15] use a denotational semantics (but might have used big-step to the same avail) for specification and coarse equivalences and a reduction semantics as their basis for an execution model; Klein and Nipkow [KN06] use a big-step semantics for compiler correctness, and a small-step semantics for their type soundness proof; etc. By abstracting from intermediate small-steps, big-step semantics are useful for coarse equivalences, whereas small-step is useful for finer equivalences as well as progress/preservation style type soundness proofs. However, maintaining multiple specifications of the same language is both tedious and error-prone. Such specifications are usually proven equivalent or sound in relation to one another manually.

Refocusing, due to Danvy and Nielsen [DN04], is a transformation for converting between small-step and big-step evaluation strategies for functional representations of reduction semantics. This provides a means of inter-deriving semantics at different levels of abstraction. Here, we observe that the refocusing transformation can be internalised in SOS, and in particular XSOS. This provides a means of relating small-step and big-step XSOS rules automatically in a way that is correct-by-construction.

4.1 Danvy and Nielsen's refocusing transformation

This section recalls the essential steps involved in Danvy and Nielsen's refocusing transformation. We follow Danvy [Dan09] and refocus the simple arithmetic expressions language using functional representations implemented in Standard ML [MTHM97].

Abstract syntax. The following datatype gives the abstract syntax for simple arithmetic expressions:

```
datatype Term = NUM of int | PLUS of Term * Term
```

We let values be given by integers:

```
type Value = Int
```

Notion of contraction. The evaluation context specification for simple arithmetic expressions from Figure 2.48 on page 60 is given by the following datatype:

```
datatype Ctx = C_MT
             | C_PLUS1 of Term * Ctx
             | C_PLUS2 of Value * Ctx
```

This datatype represents evaluation contexts as an 'inside-out' term in a zipper-like fashion [Hue97], where `C_MT` is the empty context, representing the top-level of the term. A term is decomposed into a potential redex and a reduction context using the following `decompose` function:

```
datatype PotRed = PR_PLUS of Term * Term

datatype ValOrDecomp = VAL of Value | DECOMP of PotRed * Ctx

fun decompose_term (PLUS (t1, t2), c)
    = decompose_term (t1, C_PLUS1 (t2, c))
  | decompose_term (NUM n, c)
    = decompose_ctx (c, n)
and decompose_ctx (C_MT, n)
    = VAL n
  | decompose_ctx (C_PLUS1 (t2, c), n)
    = decompose_term (t2, C_PLUS2 (n, c))
  | decompose_ctx (C_PLUS2 (n1, c), n2)
    = DECOMP (PR_PLUS (NUM n1, NUM n2), c)

fun decompose t = decompose_term (t, C_MT)
```

Here, `decompose_term` decomposes a term, whereas `decompose_ctx` decides based on the current context and term whether a term is in normal-form (`VAL n`), whether a redex has been found (`DECOMP _`), or whether to continue decomposing the current term.

We also define a `recompose` function for recomposing (or 'plugging') a term into a context:

```
fun recompose (C_MT, t)
    = t
  | recompose (C_PLUS1 (t2, c), t1)
    = recompose (c, PLUS (t1, t2))
  | recompose (C_PLUS2 (v, c), t2)
    = recompose (c, PLUS (NUM v, t2))
```

Now, we are equipped to define the contraction function given by the rule (RS-Plus) in Figure 2.49:

```
datatype ContractOrErr = CONTRACTUM of Term | ERROR of string

fun contract (PR_PLUS (NUM n1, NUM n2))
    = CONTRACTUM (NUM (n1+n2))
  | contract _
    = ERROR "Something went wrong"
```

Iteration. We define a `normalize` function that continuously decomposes a term into a potential redex and a context, contracts the redex, and recomposes the resulting term into the context to construct the next term in the reduction sequence:

```
datatype ResultOrWrong = RESULT of Value | WRONG of string

fun iterate (VAL v)
    = RESULT v
  | iterate (DECOMP (pr, c))
    = (case contract pr
        of (CONTRACTUM t')
           => iterate (decompose (recompose (c, t')))
        |  (ERROR s)
           => WRONG s)

fun normalize t = iterate (decompose t)
```

Here, `iterate` effectively implements the transitive closure of $\longmapsto$ from Figure 2.49 which iterates $\longmapsto$ towards a final value, if it exists.

Refocusing. The interpreter defined so far decomposes a term, contracts it, and recomposes it. Danvy and Nielsen observe that, instead of recomposing a contractum fully into the context just to decompose the term in the next step, we can plug the contractum into the current context, and continue decomposition in situ until the next redex position in found. This essentially applies the transitive closure inside sub-terms when they are visited, as opposed to visiting each sub-term multiple times in successive iterations. Figure 4.1 is from Danvy's lecture notes [Dan09, p. 68] and illustrates how refocusing avoids recomposing and decomposing the term by the dotted 'refocusing' arrow in the diagram.

A `refocus` function that achieves the effect described above is thus given exactly by the `decompose_term` function:

```
fun refocus (t, c) = decompose_term (t, c)
```

Similarly, a refocused evaluation strategy is given by replacing the call `(decompose (recompose (c, t')))` in the body of `iterate` by `(refocus (c, t'))`:

```
fun iterate' (VAL v)
    = RESULT v
  | iterate (DECOMP (pr, c))
    = (case contract pr
        of (CONTRACTUM t')
           => iterate (refocus (c, t')))
        |  (ERROR s)
           => WRONG s)

fun normalize' t = iterate (refocus (t, C_MT))
```

The result is a so-called *small-step abstract machine* [DM08], which relies on an iteration loop for driving its big-step evaluation strategy forward.

From small-step abstract machine to big-step functional evaluator. Danvy [Dan09, DM08] shows how the small-step abstract machine can be transformed into a big-step evaluator that essentially implements a big-step operational semantics. Appendix C.3 illustrates these steps.

4.1.1 Refocusing and SOS

It has been conjectured that any structural operational semantics can be expressed as a reduction semantics.[1] This brings many SOS rules within the reach of Danvy and Nielsen's refocusing transformation, by: transforming the SOS into a reduction semantics, refocusing, and converting the resulting abstract machine back into a purely structural big-step semantics. This sequence of transformational steps is provably correct: Sieczkowski et al [SBB11] has proven the correctness of refocusing formally, and the remaining transformations rely on continuation-passing transformations [Plo75], defunctionalisation [Rey72], lightweight fusion [OS07], and mechanical refactoring.

[1]Danvy [Dan08, Abstract] provides evidence to support this conjecture and attributes it to Felleisen.

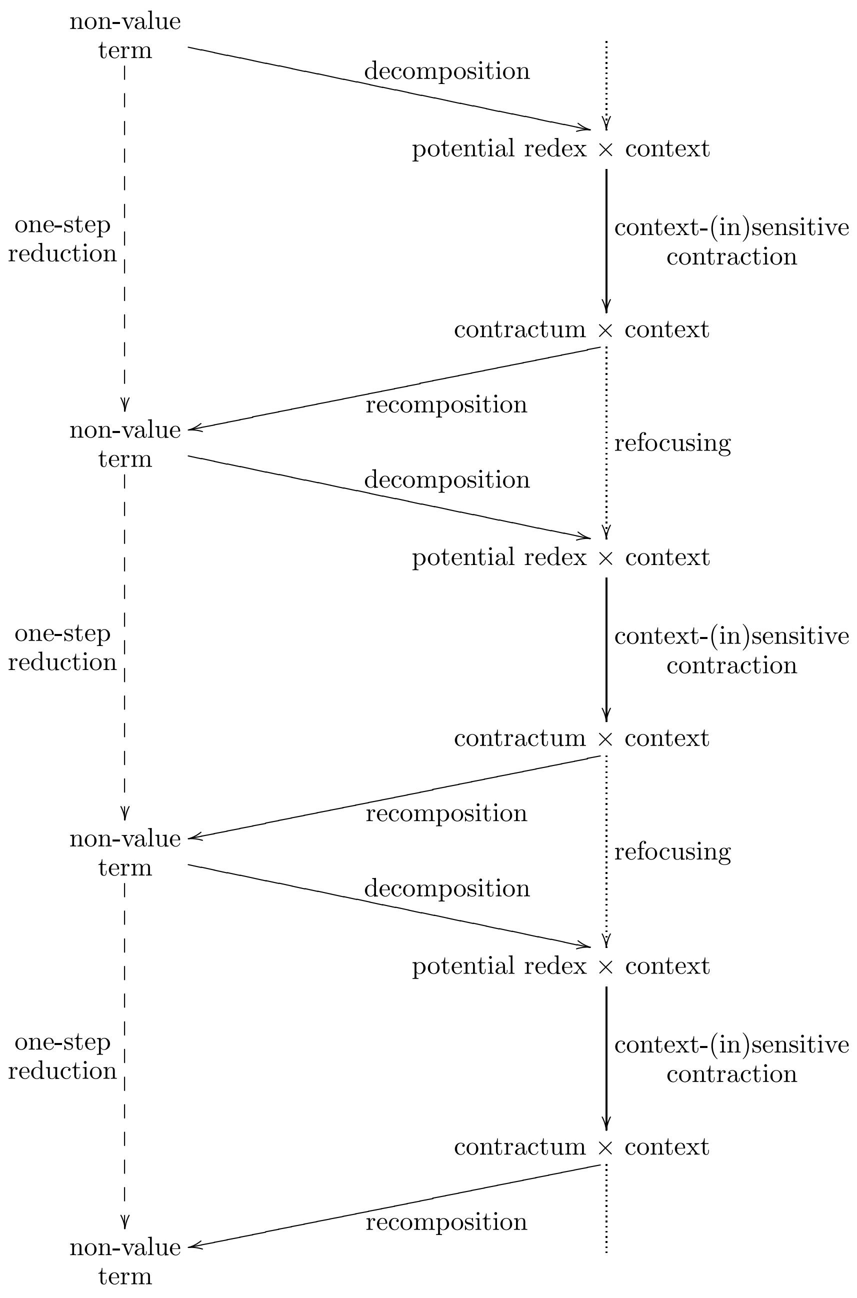

Figure 4.1: Reduction sequence with naive and refocused evaluation strategies

Here, we consider an alternative and for our purposes more direct way of going from small-step XSOS to pretty-big-step XSOS, along with a direct proof of its correctness. The rest of this chapter is based on joint work with Mosses [BPM14a], and is organised as follows:

- In Section 4.2 we show how to internalise the refocusing transformation in XSOS. This provides a simple and direct way of transforming small-step XSOS rules into corresponding pretty-big-step XSOS rules.
- Inspired by the criteria outlined by Danvy and Nielsen, Section 4.3 identifies correctness criteria for refocusing in XSOS, and gives a proof of correctness based on a rule format for XSOS without abrupt termination.
- Subsequently, we extend the format to allow rules to match against the structure of auxiliary entities in the current configurations so as to encode abrupt termination as described in Section 3.5.
- Refocusing in XSOS provides a simple means of relating extensible and purely structural specifications at different levels of abstraction in a rule format that suffices to express a wide array of language features. Section 4.6 showcases the applicability of refocusing in XSOS for deriving pretty-big-step XSOS for semantics with continuations.

4.2 Refocusing in XSOS

The essential steps involved in the refocusing transformation are:

1. eagerly applying the transitive closure inside evaluable sub-terms; and
2. fusing the iteration relation and the small-step transition relation.

While these steps are typically [DN04, Dan09, BD07, DJZ11, Zer13, Joh15, Ser12] applied to functional representations of reduction semantics, the steps are equally applicable to XSOS rules directly. Applying Danvy and Nielsen's refocusing to SOS rules gives a transformation from small-step XSOS into pretty-big-step XSOS.

The transformation essentially converts a small-step XSOS rule of the form:

$$\frac{\begin{array}{c}\neg\mathrm{Q}(e,S)\\ R' \vdash e_{/S} \to e'_{/S'}\end{array}}{R \vdash f(v_1,\ldots,v_n,e,\ldots)_{/S} \to f(v_1,\ldots,v_n,e',\ldots)_{/S'}}$$

into a pretty-big-step XSOS rule of the form:

$$\frac{\begin{array}{c}\neg\mathrm{Q}(e,S)\\ R' \vdash e_{/S} \Downarrow e'_{/S'} \qquad R \vdash f(v_1,\ldots,v_n,e',\ldots)_{/S'} \Downarrow e''_{/S''}\end{array}}{R \vdash f(v_1,\ldots,v_n,e,\ldots)_{/S} \Downarrow e''_{/S''}}$$

We illustrate how the transformation applies to the small-step XSOS for simple arithmetic expressions in Figure 4.2.

Abstract syntax ($\Sigma_{\texttt{plus}}$). (See Figure 3.4)
Auxiliary entities ($\mathbb{C}_{\texttt{plus}}$). (See Figure 3.4)
Final configurations ($Q_{\texttt{plus}}$).(See Figure 3.4)
Transition relation ($D_{\texttt{plus}}$).

$$\frac{e_1 \rightarrow e_1'}{\texttt{plus}(e_1, e_2) \rightarrow \texttt{plus}(e_1', e_2)} \quad \text{(XSOS-Plus1)}$$

$$\frac{e_2 \rightarrow e_2'}{\texttt{plus}(n_1, e_2) \rightarrow \texttt{plus}(n_1, e_2')} \quad \text{(XSOS-Plus2)}$$

$$\frac{}{\texttt{plus}(n_1, n_2) \rightarrow n_1 + n_2} \quad \text{(XSOS-Plus)}$$

$$\frac{}{\texttt{num}(n) \rightarrow n} \quad \text{(XSOS-Num)}$$

Figure 4.2: Abbreviated small-step XSOS rules for simple arithmetic expressions

$$\frac{R \vdash e_{/S} \rightarrow e'_{/S'} \qquad R \vdash e'_{/S'} \rightarrow^{\star} e''_{/S''}}{R \vdash e_{/S} \rightarrow^{\star} e''_{/S''}} \quad \text{(XSOS-Iter-Trans)}$$

$$\frac{\mathrm{Q}(e, S)}{R \vdash e_{/S} \rightarrow^{\star} e_{/S}} \quad \text{(XSOS-Iter-Refl)}$$

Figure 4.3: Iteration relation

Iteration. Figure 4.3 defines a relation $\rightarrow^{\star}$ that is only reflexive on final states. Thus, $\rightarrow^{\star}$ iterates a small-step transition relation towards a final configuration, if it exists.

Refocusing. We introduce a *refocusing rule*:

$$\frac{\begin{array}{c}\neg\mathrm{Q}(e, S) \\ R \vdash e_{/S} \rightarrow^{\star} e'_{/S'}\end{array}}{R \vdash e_{/S} \rightarrow e'_{/S'}} \quad \text{(XSOS-Refocus)}$$

This rule permits us to apply the iteration relation inside evaluable (hence the $\neg\mathrm{Q}(e, S)$ in the premise of the rule) sub-terms such that they are evaluated as they are visited. For example, whereas ordinary evaluation traverses the entire program term to construct a derivation tree like the following (where we omit inessential auxiliary entities in judgments):

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{plus}(1,2) \to 3}}{\texttt{plus}(\texttt{plus}(1,2),3) \to \texttt{plus}(3,3)}}{\texttt{plus}(\texttt{plus}(\texttt{plus}(1,2),3),5) \to \texttt{plus}(\texttt{plus}(3,3),5)} \qquad \begin{array}{c}\vdots\\ \dfrac{\dfrac{\dfrac{}{\texttt{plus}(3,3) \to 6}}{\texttt{plus}(\texttt{plus}(3,3),5) \to \texttt{plus}(6,5)} \qquad \begin{array}{c}\vdots\\ \dfrac{\dfrac{}{\texttt{plus}(6,5) \to 11} \qquad \dfrac{}{11 \to^\star 11}}{\texttt{plus}(6,5) \to^\star 11}\end{array}}{\texttt{plus}(\texttt{plus}(3,3),5) \to^\star 11}\end{array}}{\texttt{plus}(\texttt{plus}(\texttt{plus}(1,2),3),5) \to^\star 11}$$

By applying the refocusing rule, we can fully evaluate sub-terms in derivation trees; for example:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{plus}(1,2) \to 3} \qquad \dfrac{\dfrac{}{\texttt{plus}(3,3) \to 6} \qquad \dfrac{}{6 \to^\star 6}}{\texttt{plus}(3,3) \to^\star 6}}{\texttt{plus}(\texttt{plus}(1,2),3) \to^\star 6}}{\texttt{plus}(\texttt{plus}(\texttt{plus}(1,2),3),5) \to \texttt{plus}(6,5)} \qquad \dfrac{\dfrac{}{\texttt{plus}(6,5) \to 11} \qquad \dfrac{}{11 \to^\star 11}}{\texttt{plus}(6,5) \to^\star 11}}{\texttt{plus}(\texttt{plus}(\texttt{plus}(1,2),3),5) \to^\star 11}$$

The first step of the refocusing transformation is to eagerly apply the iteration relation inside sub-terms. We achieve this by specialising all rules with premises in relation to the refocusing rule; e.g., any rule:

$$\dfrac{\begin{array}{c}\neg\mathrm{Q}(e,S)\\ R' \vdash e_{/S} \to e'_{/S'}\end{array}}{R \vdash f(\ldots,e,\ldots)_{/S} \to e''_{/S''}}$$

is specialised as follows:

$$\dfrac{\begin{array}{c}\neg\mathrm{Q}(e,S)\\ \text{(XSOS-Refocus)}\,\dfrac{R' \vdash e_{/S} \to^\star e'_{/S'}}{R' \vdash e_{/S} \to e'_{/S'}}\end{array}}{R \vdash f(\ldots,e,\ldots)_{/S} \to e''_{/S''}} \equiv \dfrac{\begin{array}{c}\neg\mathrm{Q}(e,S)\\ R' \vdash e_{/S} \to^\star e'_{/S'}\end{array}}{R \vdash f(\ldots,e,\ldots)_{/S} \to e''_{/S''}}$$

The transformation produces the refocused XSOS rules in Figure 4.4.

Fusing iteration and small-step transitions. We unfold the top-level iteration rule (XSOS-Iter-Trans) from Figure 4.3 in relation to all possible rules; e.g., unfolding in relation to a refocused rule:

$$\dfrac{\neg\mathrm{Q}(e,S) \qquad R' \vdash e_{/S} \to^\star e'_{/S'}}{R \vdash f(\ldots,e,\ldots)_{/S} \to e''_{/S''}}$$

$$\frac{\neg\mathtt{Q}(e_1,S) \quad e_1 \rightarrow^\star e_1'}{\mathtt{plus}(e_1,e_2) \rightarrow \mathtt{plus}(e_1',e_2)} \tag{XSOS-R-Plus1}$$

$$\frac{\neg\mathtt{Q}(e_2,S) \quad e_2 \rightarrow^\star e_2'}{\mathtt{plus}(n_1,e_2) \rightarrow \mathtt{plus}(n_1,e_2')} \tag{XSOS-R-Plus2}$$

$$\frac{}{\mathtt{plus}(n_1,n_2) \rightarrow n_1 + n_2} \tag{XSOS-R-Plus}$$

$$\frac{}{\mathtt{num}(n) \rightarrow n} \tag{XSOS-R-Num}$$

$$\frac{R \vdash e_{/S} \rightarrow e'_{/S'} \quad R \vdash e'_{/S'} \rightarrow^\star e''_{/S''}}{R \vdash e_{/S} \rightarrow^\star e''_{/S''}} \tag{XSOS-Iter-Trans}$$

$$\frac{\mathtt{Q}(e,S)}{R \vdash e_{/S} \rightarrow^\star e_{/S}} \tag{XSOS-Iter-Refl}$$

Figure 4.4: Refocused XSOS for simple arithmetic expressions

gives:

$$\text{(XSOS-Iter-Trans)}\frac{\dfrac{\neg\mathtt{Q}(e,S) \quad R' \vdash e_{/S} \rightarrow^\star e'_{/S'}}{R \vdash f(\ldots,e,\ldots)_{/S} \rightarrow e''_{/S''}} \quad R \vdash e''_{/S''} \rightarrow^\star e'''_{/S'''}}{R \vdash f(\ldots,e,\ldots)_{/S} \rightarrow^\star e'''_{/S'''}}$$

$$\equiv \frac{\neg\mathtt{Q}(e,S) \quad R' \vdash e_{/S} \rightarrow^\star e'_{/S'} \quad R \vdash e''_{/S''} \rightarrow^\star e'''_{/S'''}}{R \vdash f(\ldots,e,\ldots)_{/S} \rightarrow^\star e'''_{/S'''}}$$

Applying this specialisation to the refocused rules from Figure 4.4 and renaming the $\rightarrow^\star$ relation to $\Downarrow$ gives the abbreviated pretty-big-step XSOS in Figure 4.5, which coincide with the rules from Figure 3.4 on page 91 (abbreviated according to Conventions 3.4 and 3.5, page 92).

This section showed how to internalise refocusing in XSOS by applying the essential steps of the transformation in XSOS rules directly, which provides a shortcut for relating purely structural small-step and pretty-big-step specifications. Next, we give a direct proof of correctness of refocusing in XSOS without abrupt termination.

4.3 Correctness of refocusing without abrupt termination

The previous section illustrated how to transform small-step rules into big-step rules. In this section, we adapt the correctness criteria for refocusing due to Danvy and Nielsen [DN04]. Subsequently we prove that small-step and pretty-big-step rules satisfying these criteria are inductively equivalent.

$$\frac{e_1 \Downarrow e_1' \quad \texttt{plus}(e_1', e_2) \Downarrow e'}{\texttt{plus}(e_1, e_2) \Downarrow e'} \quad \text{(XSOS-PB-Plus1)}$$

$$\frac{e_2 \Downarrow e_2' \quad \texttt{plus}(n_1, e_2') \Downarrow e'}{\texttt{plus}(n_1, e_2) \Downarrow e'} \quad \text{(XSOS-PB-Plus2)}$$

$$\frac{}{\texttt{plus}(n_1, n_2) \Downarrow n_1 + n_2} \quad \text{(XSOS-PB-Plus)}$$

$$\frac{}{\texttt{num}(n) \Downarrow n} \quad \text{(XSOS-PB-Num)}$$

$$\frac{\texttt{Q}(e, S)}{R \vdash e_{/S} \Downarrow e_{/S}} \quad \text{(XSOS-PB-Iter-Refl)}$$

Figure 4.5: Derived pretty-big-step XSOS rules for simple arithmetic expressions

4.3.1 Rule schemas

In order to describe the structure of rules we rely on *rule schemas* for describing sets of inference rules. An example suffices to describe what we understand by a rule schema. We write:

$$\forall e_1 \ldots e_n \; e_1' \; S \; S'. \frac{\begin{array}{c}\neg\texttt{Q}(e_1, S) \\ R' \vdash e_{1/S} \to e'_{1/S'}\end{array}}{R \vdash f(e_1, \ldots, e_n, \ldots)_{/S} \to f(e_1', \ldots, e_n, \ldots)_{/S'}}$$

for the rule schema describing the set of all inference rules whose conclusion source is some term with a term constructor f and $e_1, \ldots, e_n, \ldots$ as sub-terms that match the sort of the constructor. The $\forall$-quantification in the schema restricts which terms must be variables in the inference rules that the schema describes. Terms that are not $\forall$-quantified in the rule schema may be either variables or complex terms (i.e., terms consisting of term constructors and/or variables). Thus, the schema above permits R to be either a variable or a complex term. Similarly, the trailing dots in the list of sub-terms $e_1, \ldots, e_n, \ldots$ represent arbitrary (unrestricted) sub-terms.

An example of a rule satisfying the schema above is the rule (XSOS-Plus1) from Figure 4.2, here given in its unabbreviated form:

$$\frac{\begin{array}{c}\neg\texttt{Q}(e_1, S) \\ R \vdash e_{1/S} \to e'_{1/S'}\end{array}}{R \vdash \texttt{plus}(e_1, e_2)_{/S} \to \texttt{plus}(e_1', e_2)_{/S'}}$$

In this rule, e_1, e_2, S, and S' are variables. R is a also a variable, which is a trivial instance of a complex term, whereby it is admitted by the rule schema. The term constructor for the rule is `plus`.

4.3.2 Correctness criteria

The criteria for correctness of applying the refocusing transformation to transform a small-step extensible rule specification into a pretty-big-step extensible rule specification are given in Definition 4.1.

Definition 4.1 (Small-step left-to-right order of evaluation) Let $\mathrm{Q}(v,S)$ hold for all terms v of a distinguished syntactic sort *Val* and any S. The set of rules for a term constructor f consists of a set of simple rules and a set of rules with premises where:

- the set of rules with premises is such that there is exactly one rule that matches each of the schemas:

$$\forall e_1 \ldots e_n\ e'_1\ S\ S'. \frac{\neg\mathrm{Q}(e_1,S) \qquad R' \vdash e_{1/S} \to e'_{1/S'}}{R \vdash f(e_1,\ldots,e_n,\ldots)_{/S} \to f(e'_1,\ldots,e_n,\ldots)_{/S'}} \tag{XRS-$f1$}$$

$$\forall e_2 \ldots e_n\ e'_2\ S\ S'. \frac{\neg\mathrm{Q}(e_2,S) \qquad R' \vdash e_{2/S} \to e'_{2/S'}}{R \vdash f(v_1,e_2,\ldots,e_n,\ldots)_{/S} \to f(v_1,e'_2,\ldots,e_n,\ldots)_{/S'}} \tag{XRS-$f2$}$$

$$\vdots$$

$$\forall e_n\ e'_n\ S\ S'. \frac{\neg\mathrm{Q}(e_n,S) \qquad R' \vdash e_{n/S} \to e'_{n/S'}}{R \vdash f(v_1,\ldots,v_{n-1},e_n,\ldots)_{/S} \to f(v_1,\ldots,v_{n-1},e'_n,\ldots)_{/S'}} \tag{XRS-fn}$$

- the set of simple rules match the following schema:

$$\frac{\neg\mathrm{Q}(f(v_1,\ldots,v_n),S)}{R \vdash f(v_1,\ldots,v_n)_{/S} \to e'_{/S'}} \tag{XRS-f}$$

Refocusing small-step rules that match the schemas in Definition 4.1 gives pretty-big-step rules that match the schemas in Definition 4.2.

Definition 4.2 (Pretty-big-step left-to-right order of evaluation) Let $\mathrm{Q}(v,S)$ hold for all v of a distinguished syntactic sort *Val* and any S. The set of rules for a term constructor f consists of a set of simple rules; a set of single-premise rules; and a set of rules with premises where:

- the set of rules with premises is such that there is exactly one rule that matches each of the schemas:

$$\forall e_1 \ldots e_n\, e_1'\, S\, S'.\frac{\begin{array}{c}\neg\mathrm{Q}(e_1,S)\\ R' \vdash e_{1/S} \Downarrow e'_{1/S'} \qquad R \vdash f(e_1',\ldots,e_n,\ldots)_{/S'} \Downarrow e'_{/S''}\end{array}}{R \vdash f(e_1,\ldots,e_n,\ldots)_{/S} \Downarrow e'_{/S''}} \tag{PBXRS-$f1$}$$

$$\forall e_2 \ldots e_n\, e_2'\, S\, S'.\frac{\begin{array}{c}\neg\mathrm{Q}(e_2,S)\\ R' \vdash e_{2/S} \Downarrow e'_{2/S'} \qquad R \vdash f(v_1,e_2',\ldots,e_n,\ldots)_{/S'} \Downarrow e'_{/S''}\end{array}}{R \vdash f(v_1,e_2,\ldots,e_n,\ldots)_{/S} \Downarrow r'_{/S''}} \tag{PBXRS-$f2$}$$

$$\vdots$$

$$\forall e_n\, e_n'\, S\, S'.\frac{\begin{array}{c}\neg\mathrm{Q}(e_n,S)\\ R' \vdash e_{n/S} \Downarrow e'_{n/S'} \qquad R \vdash f(v_1,\ldots,v_{n-1},e_n',\ldots)_{/S'} \Downarrow e'_{/S''}\end{array}}{R \vdash f(v_1,\ldots,v_{n-1},e_n,\ldots)_{/S} \Downarrow e'_{/S''}} \tag{PBXRS-fn}$$

- the set of single-premise rules match the following schema:

$$\frac{\begin{array}{c}\neg\mathrm{Q}(f(v_1,\ldots,v_n),S) \qquad \neg\mathrm{Q}(e',S')\\ R \vdash e'_{/S'} \Downarrow e''_{/S''}\end{array}}{R \vdash f(v_1,\ldots,v_n)_{/S} \Downarrow e''_{/S''}} \tag{PBXRS-$f0$}$$

- the set of simple rules match the following schema:

$$\frac{\neg\mathrm{Q}(f(v_1,\ldots,v_n),S) \qquad \mathrm{Q}(e',S')}{R \vdash f(v_1,\ldots,v_n)_{/S} \Downarrow e'_{/S'}} \tag{PBXRS-f}$$

Example 4.3 The small-step semantics for simple arithmetic expressions in Figure 4.2 on page 103 matches the rule schema in Definition 4.1. Similarly, the pretty-big-step specification of simple arithmetic expressions in Figure 4.5 matches the rule schema in Definition 4.2.

4.3.3 Proof of correctness

The proof generalizes the traditional approach to relating big-step and small-step relations that is found many places in the literature [Nip06, LG09, NK14, Cio13, PCG+13]. Whereas these proofs in the literature are given on an language-by-language basis, the proof we supply here is generic up to the rule schema for left-to-right order of evaluation.

Theorem 4.4 (Refocusing is correct) For any transition relation $\rightarrow$ that implements left-to-right order of evaluation and whose refocused counterpart is a relation $\Downarrow$, it holds that:

$$\begin{array}{l} \mathrm{Q}(e', S') \implies \\ (R \vdash e_{/S} \rightarrow^* e'_{/S'} \iff R \vdash e_{/S} \Downarrow e'_{/S'}) \end{array}$$

Proof. The property is a direct consequence of Lemmas 4.5 and 4.7. □

Lemma 4.5 (Refocusing is sound) For any transition relation $\rightarrow$ that implements left-to-right order of evaluation and whose refocused counterpart is a relation $\Downarrow$, it holds that:

$$R \vdash e_{/S} \Downarrow e'_{/S'} \implies R \vdash e_{/S} \rightarrow^* e'_{/S'}$$

Proof (sketch). The proof is by rule induction on $\Downarrow$, using Lemma 4.6 below and the transitivity of $\rightarrow^*$. The full proof is in Appendix B.1. □

Lemma 4.6 (Congruence of reflexive-transitive closure) For any transition relation $\rightarrow$ that implements left-to-right order of evaluation, and where $\rightarrow$ has a rule:

$$\frac{R' \vdash e_{/S} \rightarrow e'_{/S'}}{R \vdash f(v_1, \ldots, v_n, e, \ldots)_{/S} \rightarrow f(v_1, \ldots, v_n, e', \ldots)_{/S'}}$$

it holds that:

$$\begin{array}{l} R' \vdash e_{/S} \rightarrow^* e'_{/S'} \implies \\ R \vdash f(v_1, \ldots, v_n, e, \ldots)_{/S} \rightarrow^* f(v_1, \ldots, v_n, e', \ldots)_{/S'} \end{array}$$

Proof (sketch). The proof is by rule induction on the hypothesis. The full proof is in Appendix B.1. □

Lemma 4.7 (Refocusing is complete) For any transition relation $\rightarrow$ that implements left-to-right order of evaluation and whose refocused counterpart is a relation $\Downarrow$, it holds that:

$$\begin{array}{l} R \vdash e_{/S} \rightarrow^* e'_{/S'} \implies \\ R \vdash e_{/S} \Downarrow e'_{/S'} \end{array}$$

Proof (sketch). The proof is by rule induction on the hypothesis, and uses Lemma 4.8. The full proof is in Appendix B. □

Lemma 4.8 (Pretty-big-steps can be broken up into small-steps) For any transition relation $\rightarrow$ that implements left-to-right order of evaluation and whose refocused counterpart is a relation $\Downarrow$, it holds that:

$$\begin{array}{l} \neg\mathrm{Q}(e, S) \implies R \vdash e_{/S} \rightarrow e'_{/S'} \implies R \vdash e'_{/S'} \Downarrow e''_{/S''} \implies \\ R \vdash e_{/S} \Downarrow e''_{/S''} \end{array}$$

Proof (sketch). The proof is by rule induction on the second hypothesis, reasoning on the structure of rules for $\rightarrow$, and using Lemma 4.9. The full proof is in Appendix B.1. □

Lemma 4.9 (Correspondence between terminating small-steps and pretty-big-steps) For any transition relation $\rightarrow$ that implements left-to-right order of evaluation and whose refocused counterpart is a relation $\Downarrow$, it holds that:

$$\neg\mathtt{Q}(e,S) \implies R \vdash e_{/S} \rightarrow e'_{/S'} \implies \mathtt{Q}(e',S') \implies R \vdash e_{/S} \Downarrow e'_{/S'}$$

Proof. By induction on $\rightarrow$. The only case to consider is that of simple rules; specifically, simple rules that make a transition to a terminal configuration. In this case, the goal is immediate, since by the refocusing transformation there is exactly one corresponding pretty-big-step rule of the same structure. □

In Section 4.5 we show that it is straightforward to relax the definition of left-to-right order of evaluation and extend the correctness argument for refocusing to deal with abrupt termination as well. First, we show how Theorem 4.4 suffices to automatically derive extensible pretty-big-step XSOS rules from the small-step XSOS for λ_{cbv}.

4.4 Refocusing λ_{cbv}

We have already shown how the refocused rules look for simple arithmetic expressions. In this section we consider how to give small-step and pretty-big-step XSOS rules, and how Theorem 4.4 gives the correctness of the two relative to one another.

Figure 4.6 gives a small-step extensible rule specification for the call-by-value λ-calculus. But these rules do not match the rule schema for left-to-right order of evaluation as specified by our Definition 4.1! The culprit is the application rule (XSOS-λ_{cbv}-AppC) which does evaluation inside the body of a closure:

$$\frac{\mathbf{env}\ \rho'[x \mapsto v_2] \vdash e \rightarrow e'}{\mathbf{env}\ \rho \vdash \langle x, e, \rho' \rangle\ v_2 \rightarrow \langle x, e', \rho' \rangle\ v_2}$$

There are several ways ways in which we could argue that refocusing this rule is correct. One is to use a more liberal rule schema. For example, one could conceivably permit pattern matching against values and allow evaluation inside sub-terms in a similar style to Churchill and Mosses' [CM13] bisimulation format for MSOS. This would entail a somewhat more involved rule schema than the one given in previous section. An alternative approach that we pursue here is to observe that it is straightforward to break the application construct into two constructs: one for binding the variable of the closure to the argument value (`app`), and one for forcing evaluation of the closure (`force`).[2] Figure 4.7 summarises the rules for such constructs.

[2]Appendix A of this thesis breaks the application construct up in a similar way. The challenge with

Abstract syntax ($\Sigma_{\lambda_{\text{cbv}}}$).

$$\begin{aligned}
\mathit{Expr} \ni e &::= \lambda x.e \mid e\ e \mid x \mid v \\
\mathit{Val} \ni v &::= \langle x, e, \rho \rangle \\
x, y \in \mathit{Var} &\triangleq \{\mathrm{x}, \mathrm{y}, \ldots\} \\
\rho \in \mathit{Env} &\triangleq \mathit{Var} \xrightarrow{\text{fin}} \mathit{Val}
\end{aligned}$$

Auxiliary entities ($\mathbb{C}_{\lambda_{\text{cbv}}}$).

$$\mathbb{C}_{\lambda_{\text{cbv}}} \triangleq (\mathbf{env}, \mathbb{C}^{\text{DISCRETE}}(\mathit{Env}))$$

Final configurations ($Q_{\lambda_{\text{cbv}}}$).

$$\frac{}{\mathrm{Q}(v, S)}$$

Rule specification ($D_{\lambda_{\text{cbv}}}$).

$$\frac{}{\mathbf{env}\ \rho \vdash \lambda x.e \to \langle x, e, \rho \rangle} \quad \text{(XSOS-}\lambda_{\text{cbv}}\text{-Lam)}$$

$$\frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2} \quad \text{(XSOS-}\lambda_{\text{cbv}}\text{-App1)}$$

$$\frac{e_2 \to e_2'}{\langle x, e, \rho' \rangle\ e_2 \to \langle x, e, \rho' \rangle\ e_2'} \quad \text{(XSOS-}\lambda_{\text{cbv}}\text{-App2)}$$

$$\frac{\mathbf{env}\ \rho'[x \mapsto v_2] \vdash e \to e'}{\mathbf{env}\ \rho \vdash \langle x, e, \rho' \rangle\ v_2 \to \langle x, e', \rho' \rangle\ v_2} \quad \text{(XSOS-}\lambda_{\text{cbv}}\text{-AppC)}$$

$$\frac{}{\mathbf{env}\ \rho \vdash \langle x, v, \rho' \rangle\ v_2 \to v} \quad \text{(XSOS-}\lambda_{\text{cbv}}\text{-App)}$$

$$\frac{x \in \mathrm{dom}(\rho)}{\mathbf{env}\ \rho \vdash x \to \rho(x)} \quad \text{(XSOS-}\lambda_{\text{cbv}}\text{-Var)}$$

Figure 4.6: Abbreviated XSOS rules for λ_{cbv}

$$\frac{e_1 \to e_1'}{\texttt{app}(e_1, e_2) \to \texttt{app}(e_1', e_2)} \qquad \text{(XSOS-AltApp1)}$$

$$\frac{e_2 \to e_2'}{\texttt{app}(\langle x, e, \rho \rangle, e_2) \to \texttt{app}(\langle x, e, \rho \rangle, e_2')} \qquad \text{(XSOS-AltApp2)}$$

$$\frac{}{\texttt{app}(\langle x, e, \rho \rangle, v) \to \texttt{force}(e, \rho[x \mapsto v])} \qquad \text{(XSOS-AltApp)}$$

$$\frac{\textbf{env}\ \rho' \vdash e \to e'}{\textbf{env}\ \rho \vdash \texttt{force}(e, \rho') \to \texttt{force}(e', \rho')} \qquad \text{(XSOS-Force1)}$$

$$\frac{}{\texttt{force}(v, \rho) \to v} \qquad \text{(XSOS-Force)}$$

Figure 4.7: Alternative app and force constructs for λ_{cbv}

We can now establish the correctness of refocusing the original rules for the call-by-value λ-calculus from Figure 4.6 by an argument summarised by the following diagram:

$$\begin{tikzcd}
\text{small-step } \lambda_{\text{cbv}} \arrow[r, Leftrightarrow, "(*)"] \arrow[d, Leftrightarrow] & \text{small-step } \lambda_{\texttt{app+force}} \arrow[d, Leftrightarrow, "\text{Theorem 4.4}"] \\
\text{pretty-big-step } \lambda_{\text{cbv}} \arrow[r, Leftrightarrow, "(\dagger)"] & \text{pretty-big-step } \lambda_{\texttt{app+force}}
\end{tikzcd}$$

It suffices to prove the easy horizontal properties $(*), (\dagger)$; the vertical ones follow from Theorem 4.4.

A logical next step is to consider refocusing exception handling constructs from Figure 3.2. However, the rules for `catch` do not match the rule schema for left-to-right order of evaluation. For example, the (XSOS-Catch1) rule:

$$\frac{R \vdash e_{1/S[\textbf{exc}\ \text{OK}]} \to e'_{1/S'[\textbf{exc}\ \text{OK}]}}{\texttt{catch}(e_1, x, e_2)_{/S[\textbf{exc}\ \text{OK}]} \to \texttt{catch}(e_1', x, e_2)_{/S'[\textbf{exc}\ \text{OK}]}}$$

Here, $S[\textbf{exc}\ \text{OK}]$ and $S'[\textbf{exc}\ \text{OK}]$ are not variables, so the rule does not match the rule schema in Definition 4.1.

4.5 Refocusing with abrupt termination

We relax the notion of left-to-right order of evaluation and show that this relaxation preserves the correctness of refocusing. Definition 4.10 gives the extended rule schema

having the original application rule is that the corresponding (pretty-)big-step rules are not compositional. We discuss this in connection with Lemma A.11 in Appendix A.

for small-step rules, and Definition 4.11 gives the correspondingly extended rule schema for pretty-big-step rules.

Definition 4.10 (Small-step left-to-right order of evaluation with abrupt termination) Let $\text{Q}(v,S)$ hold for all v of a distinguished syntactic sort *Val* and any S. The set of rules for a term constructor f consists of a set of simple rules and a set of rules with premises where:

- the set of rules with premises is such that there is exactly one rule for each e_i that matches each of the schemas (similar to Definition 4.10):

$$\forall e_i \ldots e_n\ S\ S'.\frac{\begin{array}{c}\neg\text{Q}(e_i,S)\\ R' \vdash e_{i/S} \rightarrow e'_{i/S'}\end{array}}{\begin{array}{l}R \vdash f(v_1,\ldots,v_{i-1},e_i,e_{i+1},\ldots,e_n,\ldots)_{/S} \rightarrow\\ \qquad f(v_1,\ldots,v_{i-1},e'_i,e_{i+1},\ldots,e_n,\ldots)_{/S'}\end{array}} \quad \text{(XRS-}fi\text{)}$$

  Alternatively, sub-term e_i may have a pair of rules instead that match the following schemas where X is a set of *final* configurations:

$$\forall e_i \ldots e_n\ S\ S'.\frac{\begin{array}{c}\neg\text{Q}(e_i,S) \qquad (e'_i,S') \in X\\ R' \vdash e_{i/S} \rightarrow e'_{i/S'}\end{array}}{R \vdash f(v_1,\ldots,v_{i-1},e_i,e_{i+1},\ldots,e_n,\ldots)_{/S} \rightarrow e''_{/S''}} \quad \text{(XRS-AT-}fi\text{)}$$

$$\forall e_i \ldots e_n\ S\ S'.\frac{\begin{array}{c}\neg\text{Q}(e_i,S) \qquad (e'_i,S') \notin X\\ R' \vdash e_{i/S} \rightarrow e'_{i/S'}\end{array}}{\begin{array}{l}R \vdash f(v_1,\ldots,v_{i-1},e_i,e_{i+1},\ldots,e_n,\ldots)_{/S} \rightarrow\\ \qquad f(v_1,\ldots,v_{i-1},e'_i,e_{i+1},\ldots,e_n,\ldots)_{/S'}\end{array}} \quad \text{(XRS-OK-}fi\text{)}$$

- the set of simple rules match the schema from Definition 4.1.

Definition 4.11 (Pretty-big-step left-to-right order of evaluation with abrupt termination) Let $\text{Q}(v,S)$ hold for all v and any S. The set of rules for a construct f consists of a set of simple rules; a set of single-premise rules; and a set of rules with premises where:

- the set of rules with premises is such that there is exactly one rule for each e_i that matches each of the schemas:

$$\forall e_i \ldots e_n\ S\ S'.\frac{\begin{array}{c}\neg\text{Q}(e_i,S) \qquad\qquad R' \vdash e_{i/S} \Downarrow e'_{i/S'}\\ R \vdash f(v_1,\ldots,v_{i-1},e'_i,e_{i+1},\ldots,e_n,\ldots)_{/S'} \Downarrow e'_{/S''}\end{array}}{R \vdash f(v_1,\ldots,v_{i-1},e_i,e_{i+1},\ldots,e_n,\ldots)_{/S} \Downarrow e'_{/S''}} \quad \text{(PBXRS-}fi\text{)}$$

Alternatively, sub-term e_i may have a pair of rules instead that match the following schemas where X is a set of *final* configurations:

$$\forall e_i \dots e_n\; S\; S'. \frac{\neg\text{Q}(e_i,S) \qquad (e_i',S') \in X \\ R' \vdash e_{i/S} \Downarrow e'_{i/S'} \qquad R \vdash e''_{/S''} \Downarrow e'''_{/S'''}}{R \vdash f(v_1,\dots,v_{i-1},e_i,e_{i+1},\dots,e_n,\dots)_{/S} \Downarrow e'''_{/S'''}} \qquad \text{(PBXRS-AT-}fi\text{)}$$

$$\forall e_i \dots e_n\; S\; S'. \frac{\neg\text{Q}(e_i,S) \qquad (e_i',S') \notin X \qquad R' \vdash e_{i/S} \Downarrow e'_{i/S'} \\ R \vdash f(v_1,\dots,v_{i-1},e_i',e_{i+1},\dots,e_n,\dots)_{/S'} \Downarrow e''_{/S''}}{R \vdash f(v_1,\dots,v_{i-1},e_i,e_{i+1},\dots,e_n,\dots)_{/S} \Downarrow e''_{/S''}} \qquad \text{(PBXRS-OK-}fi\text{)}$$

- the set of rules with a single premise and simple rules match the schemas from Definition 4.2.

The `catch` construct from Figure 3.2 on page 88 matches the alternative schema for rules with premises, since the rules for `catch` are equivalently given by letting $X \triangleq \{(e, S[\textbf{exc}\ \text{EXC}(v)])\}$ in the following rules:

$$\frac{\neg\text{Q}(e_1,S) \qquad (e_1',S') \notin X \\ R \vdash e_{1/S} \to e'_{1/S'}}{R \vdash \texttt{catch}(e_1,x,e_2)_{/S} \to \texttt{catch}(e_1',x,e_2)_{/S'}} \qquad \text{(XSOS-Catch1)}$$

$$\frac{}{\texttt{catch}(v,x,e_2) \to v} \qquad \text{(XSOS-CatchV)}$$

$$\frac{\neg\text{Q}(e_1,S) \qquad (e_1',S') \in X \\ R \vdash e_{1/S} \to e'_{1/S'}}{R \vdash \texttt{catch}(e_1,x,e_2)_{/S} \to \texttt{bind}(x,v,e_2)_{/S'[\textbf{exc}\ \text{OK}]}} \qquad \text{(XSOS-CatchE)}$$

If we can prove that refocusing is sound for rules with left-to-right order of evaluation with abrupt termination as defined in Definition 4.10, we get as a corollary that the translation between small-step and pretty-big-step rules with abrupt termination is correct.

4.5.1 Extended proof of correctness

We extend the proof of Theorem 4.4.

Theorem 4.12 (Refocusing with abrupt termination is correct) For any transition relation $\to$ that implements left-to-right order of evaluation with abrupt termination and whose refocused counterpart is $\Downarrow$, it holds that:

$$\text{Q}(e',S') \implies \\ \left(R \vdash e_{/S} \to^* e'_{/S'} \iff R \vdash e_{/S} \Downarrow e'_{/S'}\right)$$

Proof. The property is a direct consequence of Lemmas 4.13 and 4.15. □

Lemma 4.13 (Refocusing with abrupt termination is sound) For any transition relation $\rightarrow$ that implements left-to-right order of evaluation with abrupt termination and whose refocused counterpart is a relation $\Downarrow$, it holds that:

$$\begin{array}{l} R \vdash e_{/S} \Downarrow e'_{/S'} \implies \\ R \vdash e_{/S} \rightarrow^* e'_{/S'} \end{array}$$

Proof. The proof is a straightforward extension of the proof of Lemma 4.5, using Lemma 4.14 (see below). □

Lemma 4.14 (Congruence of reflexive-transitive closure with abrupt termination) For any transition relation $\rightarrow$ that implements left-to-right order of evaluation with abrupt termination, and where $\rightarrow$ has a pair of rules:

$$\frac{\begin{array}{c}\neg\mathrm{Q}(e_i,S) \qquad (e'_i,S') \in X \\ R' \vdash e_{i/S} \rightarrow e'_{i/S'}\end{array}}{R \vdash f(v_1,\ldots,v_{i-1},e_i,e_{i+1},\ldots,e_n,\ldots)_{/S} \rightarrow e''_{/S''}}$$

$$\frac{\begin{array}{c}\neg\mathrm{Q}(e_i,S) \qquad (e'_i,S') \notin X \\ R' \vdash e_{i/S} \rightarrow e'_{i/S'}\end{array}}{\begin{array}{l}R \vdash f(v_1,\ldots,v_{i-1},e_i,e_{i+1},\ldots,e_n,\ldots)_{/S} \rightarrow \\ \qquad f(v_1,\ldots,v_{i-1},e'_i,e_{i+1},\ldots,e_n,\ldots)_{/S'}\end{array}}$$

where X is a set of final configurations. For each such pair of rules, it holds that:

$$\begin{array}{l} R' \vdash e_{i/S} \rightarrow^* e'_{i/S'} \implies (e'_i,S') \in X \implies \\ R \vdash f(v_1,\ldots,v_{i-1},e_i,e_{i+1},\ldots,e_n,\ldots)_{/S} \rightarrow^* e''_{/S''} \end{array} \tag{1}$$

and:

$$\begin{array}{l} R' \vdash e_{i/S} \rightarrow^* e'_{i/S'} \implies (e'_i,S') \notin X \implies \\ R \vdash f(v_1,\ldots,v_{i-1},e_i,e_{i+1},\ldots,e_n,\ldots)_{/S} \rightarrow^* f(v_1,\ldots,v_{i-1},e'_i,e_{i+1},\ldots,e_n,\ldots)_{/S'} \end{array} \tag{2}$$

Proof. The proof of (2) follows the structure of Lemma 4.6. The proof of (1) is by rule induction on $\rightarrow^*$. The full proof is in Appendix B.1. □

Lemma 4.15 (Refocusing with abrupt termination is complete) For any transition relation $\rightarrow$ that implements left-to-right order of evaluation with abrupt termination and whose refocused counterpart is a relation $\Downarrow$, it holds that:

$$\begin{array}{l} R \vdash e_{/S} \rightarrow^* e'_{/S'} \implies (e',S') \in F \implies \\ R \vdash e_{/S} \Downarrow e'_{/S'} \end{array}$$

Proof. The proof structure is the same as Lemma 4.7, and relies on Lemma 4.16, which is an extended version of Lemma 4.8. □

Lemma 4.16 (Pretty-big-steps can be broken up into small-steps with abrupt termination) For any transition relation $\rightarrow$ that implements left-to-right order of evaluation with abrupt termination and whose refocused counterpart is a relation $\Downarrow$, it holds that:

$$R \vdash e_{/S} \rightarrow e'_{/S'} \implies R \vdash e'_{/S'} \Downarrow e''_{/S''} \implies \\ R \vdash e_{/S} \Downarrow e''_{/S''}$$

Proof. The proof is a straightforward extension of Lemma 4.8. The extended proof is in Appendix B.1. □

Lemma 4.17 (Correspondence between terminating small-steps and refocusing with abrupt termination) For any transition relation $\rightarrow$ that implements left-to-right order of evaluation with abrupt termination and whose refocused counterpart is $\Downarrow$, it holds that:

$$\neg\mathrm{Q}(e,S) \implies R \vdash e_{/S} \rightarrow e'_{/S'} \implies \mathrm{Q}(e',S') \implies \\ R \vdash e_{/S} \Downarrow e'_{/S'}$$

Proof. The proof is a straightforward extension of Lemma 4.9. The full proof is in Appendix B.1. □

A corollary of Theorem 4.12 is that the pretty-big-step rules in Figure 3.5 on page 92 correspond to the small-step rules in Figure 3.2 on page 88, and that the encoding of abrupt termination proposed in Chapter 3 translates from small-step to pretty-big-step semantics.

Another corollary of Theorem 4.12 is that it is possible to give big-step semantics for delimited continuations using big-step rules without explicit representation of program context.

4.6 Deriving pretty-big-step rules for delimited control

A long-standing problem with purely structural approaches to specification, and particularly big-step specifications, is how to deal with semantics for continuations and control: for example, in their paper on Typing First-Class Continuations in ML , Duba et al. give a natural semantics for call/cc in Standard ML and remark [DHM91, p. 170]:

> The resulting semantics differs substantially from the dynamic semantics of Standard ML. This may be taken as evidence that the addition of call/cc to Standard ML would be a substantial change, rather than an incremental modification, to the language.

Duba et al.'s natural semantics is derived by defunctionalising a continuation-passing denotational semantics, which yields a natural semantics with explicit evaluation contexts. Similarly, Roşu remarks that in small-step and big-step SOS and MSOS "it is inconvenient (and non-modular) to define complex control statements." [RŞ10, p. 399].

Recent work by Sculthorpe et al. [STM16] shows that small-step (M)SOS *does* support specifying the semantics of call/cc in a modular way. The MSOS semantics of Sculthorpe et al. uses the modular representation of exceptions due to Klin [Mos04] that we recalled in Section 2.3.4.

Here, we translate Sculthorpe et al.'s semantics into XSOS and refocus it. As a result, we get a pretty-big-step semantics for delimited control that is correct-by-construction. This provides evidence that adding call/cc to Standard ML does not require as substantial a change as alluded to by Duba et al. or Felleisen and Wright. It suffices to: use modular abrupt termination as illustrated here; and factor the natural semantics in the Definition of Standard ML [MTHM97] into pretty-big-step rules.

4.6.1 What are delimited continuations?

Felleisen [Fel88, Fel87] introduced delimited control as a generalisation of imperative control operators, such as Landin's J operator [Lan65] or Scheme's call/cc (short for *call with current continuation*) [SS75, Cli87].

Call/cc provides access to the current continuation by passing the current continuation to a function. For example, evaluating the program:

$$\texttt{call/cc}(\lambda \text{k}.\texttt{plus}((\text{k } 1), 2))$$

proceeds by binding the variable 'k' to a special kind of abstraction. When this abstraction is invoked, it returns control to the "current continuation", i.e., the continuation at the point at which `call/cc` was invoked. For the simple program above, the current continuation for `call/cc` is the top-level identity continuation, $(\lambda x.x)$. Evaluating $(\text{k } 1)$ inside the body of the abstraction thus aborts the `plus` operation, and replaces the continuation with $((\lambda x.x)\ 1)$, making the result of evaluating the program '1'.

We can also use `call/cc` in the context of other expressions:

$$\texttt{plus}(3, \texttt{call/cc}(\lambda \text{k}.\texttt{plus}((\text{k } 1), 2)))$$

In this program, the continuation at the point of evaluating `call/cc` is $(\lambda x.\texttt{plus}(3, x))$, and so the result of evaluating this program expression is 4.

Delimited continuations, as the name implies, provides a means of delimiting the current continuation. They generalise control constructs such as `call/cc` in the sense that the semantics of `call/cc` can be expressed in terms of delimited control operators. Following Felleisen [Fel88], # (pronounced *prompt*) is used to delimit continuations, and `control` invokes an abstraction using the current delimited continuation. Unlike `call/cc` which aborts the current computation when invoking a continuation, `control` merely passes the current continuation as an abstraction. Consider the program:

$$\#(\texttt{plus}(3, \texttt{control}(\lambda \text{k}.\texttt{plus}(\text{k } 1, 2))))$$

The result of evaluating this program is 6. At the point of evaluating `control`, the program binds 'k' to the current delimited continuation which, in this case, scopes the entire program, making the continuation $(\lambda x.\texttt{plus}(3,x))$. Applying the continuation in the expression $(\texttt{k}\ 1)$ does not abort the current computation, but instead continues with 4 in the left branch of the `plus` expression.

With delimited continuations we can also apply continuations multiple times:

$$\texttt{plus}(4,\#(\texttt{plus}(3,\texttt{control}(\lambda\texttt{k}.\texttt{plus}(\texttt{k}\ (\texttt{k}\ 1)),2))))$$

The result of evaluating this program is: $4+3+3+1+2=13$. For more examples, see [FWFD88, Fel88, STM16].

4.6.2 Small-step XSOS for delimited continuations

Figure 4.8 gives the straightforward translation of Sculthorpe et al.'s [STM16] MSOS semantics for delimited continuations into XSOS. The semantics relies on a distinguished set of *auxiliary* variables (that Sculthorpe et al. [STM16] call *meta-identifiers*). The rules for '#' are instances of the rule schemas in Definition 4.10. Thus, the '#' construct listens for the *control status* signal. If a signal occurs, this indicates that we are abruptly terminating, and that we are currently constructing a continuation. The signal records a fresh auxiliary variable $\bar{x}$, which is used as the binder in the delimited continuation, used when (XSOS-PromptC) constructs the continuation and applies the continuation to the abstraction v, which is allowed to use it.

The motivation for using a distinguished set of auxiliary variables and an auxiliary environment for bindings in connection with delimited control is that a continuation may contain closures with static bindings. Consider the following program:

$$\#(\langle x,\texttt{control}(v),\rho\rangle\ 1)_{/\mathbf{ctl}\ \text{NONE}}$$
$$\to v\ (\lambda y.\overline{\texttt{let}}(\bar{y},y,\langle x,\bar{y},\rho\rangle\ 1))_{/\mathbf{ctl}\ \text{NONE}}$$

If we were to use ordinary variables and environments for constructing the continuation instead, the resulting program would be:

$$\to v\ (\lambda y.\langle x,y,\rho\rangle\ 1))_{/\mathbf{ctl}\ \text{NONE}}$$

Recalling that the application rule relies on the statically scoped environment ρ of the closure, any binding of variable y in the continuation will always be shadowed by the environment ρ of the continuation. This semantic anomaly is avoided by using auxiliary environments which are not statically scoped.

4.6.3 Pretty-big-step XSOS for delimited continuations

Figure 4.9 gives the derived pretty-big-step rules for delimited continuations. By Theorem 4.12, the rules implement the same semantics as the small-step rules.

This proves that there is no inherent problem with giving semantics to delimited control-operators using purely structural small-step or pretty-big-step semantics, provided one uses the retentive notion of abrupt termination due to Klin and Mosses

Abstract syntax ($\Sigma_{\texttt{control}}$).

$$
\begin{aligned}
\mathit{Expr} \ni e &::= \#(e) \mid \texttt{control}(e) \mid \overline{\texttt{let}}(\overline{x}, e, e) \mid \overline{x} \mid \lambda x.e \mid e\ e \\
v \in \mathit{Val} &\triangleq \emptyset \\
\mathit{ControlStat} \ni c &::= \textsc{none} \mid \textsc{ctrl}(\overline{x}, v) \\
\overline{x}, \overline{y} \in \mathit{AuxVar} &\triangleq \{\overline{\mathrm{x}}, \overline{\mathrm{y}}, \ldots\} \\
\overline{\rho} \in \mathit{AuxEnv} &\triangleq \mathit{AuxVar} \xrightarrow{\text{fin}} \mathit{Val}
\end{aligned}
$$

Auxiliary entities.

$$\mathbb{C}_{\texttt{control}} \triangleq \left\{ (\mathbf{ctl}, \mathbb{C}^{\text{AT}}(\mathit{ControlStat})), (\mathbf{aenv}, \mathbb{C}^{\text{DISCRETE}}(\mathit{AuxEnv})) \right\}$$

Final configurations ($Q_{\texttt{control}}$).

$$\frac{}{\textsc{q}(v, S)} \qquad \frac{}{\textsc{q}(e, S[\mathbf{ctl}\ \textsc{ctrl}(\overline{x}, v)])}$$

Rule specification.

$$\frac{e_{/\mathbf{ctl}\ \textsc{none}} \to e'_{/\mathbf{ctl}\ \textsc{none}}}{\#(e)_{/\mathbf{ctl}\ \textsc{none}} \to \#(e')_{/\mathbf{ctl}\ \textsc{none}}} \quad \text{(XSOS-Prompt1)}$$

$$\frac{}{\#(v) \to v} \quad \text{(XSOS-PromptV)}$$

$$\frac{e_{/\mathbf{ctl}\ \textsc{none}} \to e'_{/\mathbf{ctl}\ \textsc{ctrl}(\overline{x}, v)}}{\#(e)_{/\mathbf{ctl}\ \textsc{none}} \to v\ (\lambda x. \overline{\texttt{let}}(\overline{x}, x, e'))_{/\mathbf{ctl}\ \textsc{none}}} \quad \text{(XSOS-PromptC)}$$

$$\frac{e \to e'}{\texttt{control}(e) \to \texttt{control}(e')} \quad \text{(XSOS-Control1)}$$

$$\frac{\text{fresh}(\overline{x})}{\texttt{control}(v)_{/S[\mathbf{ctl}\ \textsc{none}]} \to \overline{x}_{/S[\mathbf{ctl}\ \textsc{ctrl}(\overline{x}, v)]}} \quad \text{(XSOS-Control)}$$

$$\frac{e_1 \to e_1'}{\overline{\texttt{let}}(\overline{x}, e_1, e_2) \to \overline{\texttt{let}}(\overline{x}, e_1', e_2)} \quad \text{(XSOS-AuxLet1)}$$

$$\frac{\mathbf{aenv}\ \overline{\rho}[\overline{x} \mapsto v] \vdash e \to e'}{\mathbf{aenv}\ \overline{\rho} \vdash \overline{\texttt{let}}(\overline{x}, v, e) \to \overline{\texttt{let}}(\overline{x}, v, e')} \quad \text{(XSOS-AuxLet2)}$$

$$\frac{}{\overline{\texttt{let}}(\overline{x}, v, v') \to v'} \quad \text{(XSOS-AuxLet)}$$

$$\frac{\overline{x} \in \text{dom}(\overline{\rho})}{\mathbf{aenv}\ \overline{\rho} \vdash \overline{x} \to \overline{\rho}(\overline{x})} \quad \text{(XSOS-AuxVar)}$$

Figure 4.8: Abbreviated small-step XSOS rules for delimited control

$$\frac{e_{/\mathbf{ctl}\ \text{NONE}} \Downarrow e'_{/\mathbf{ctl}\ \text{NONE}} \qquad \#(e')_{/\mathbf{ctl}\ \text{NONE}} \Downarrow e''_{/\mathbf{ctl}\ c}}{\#(e)_{/\mathbf{ctl}\ \text{NONE}} \Downarrow e''_{/\mathbf{ctl}\ c}} \tag{XSOS-PB-Prompt1}$$

$$\frac{}{\#(v) \Downarrow v} \tag{XSOS-PB-PromptV}$$

$$\frac{e_{/\mathbf{ctl}\ \text{NONE}} \Downarrow e'_{/\mathbf{ctl}\ \text{CTRL}(\bar{x},v)} \qquad v\ (\lambda x.\overline{\texttt{let}}(\bar{x},x,e'))_{/\mathbf{ctl}\ \text{NONE}} \Downarrow e''_{/\mathbf{ctl}\ c}}{\#(e)_{\mathbf{ctl}\ \text{NONE}} \Downarrow e''_{/\mathbf{ctl}\ c}} \tag{XSOS-PB-PromptC}$$

$$\frac{e \Downarrow e' \qquad \texttt{control}(e') \Downarrow e''}{\texttt{control}(e) \Downarrow e''} \tag{XSOS-PB-Control1}$$

$$\frac{\text{fresh}(\bar{x})}{\texttt{control}(v)_{/S[\mathbf{ctl}\ \text{NONE}]} \Downarrow \bar{x}_{/S[\mathbf{ctl}\ \text{CTRL}(\bar{x},v)]}} \tag{XSOS-PB-Control}$$

$$\frac{e_1 \Downarrow e'_1 \qquad \overline{\texttt{let}}(\bar{x},e'_1,e_2) \Downarrow e''}{\overline{\texttt{let}}(\bar{x},e_1,e_2) \Downarrow e''} \tag{XSOS-PB-AuxLet1}$$

$$\frac{\mathbf{aenv}\ \bar{\rho}[\bar{x} \mapsto v] \vdash e \Downarrow e' \qquad \mathbf{aenv}\ \bar{\rho} \vdash \overline{\texttt{let}}(\bar{x},v,e') \Downarrow e''}{\mathbf{aenv}\ \bar{\rho} \vdash \overline{\texttt{let}}(\bar{x},v,e) \Downarrow e''} \tag{XSOS-PB-AuxLet2}$$

$$\frac{}{\overline{\texttt{let}}(\bar{x},v,v') \Downarrow v'} \tag{XSOS-PB-AuxLet}$$

$$\frac{\bar{x} \in \text{dom}(\bar{\rho})}{\mathbf{aenv}\ \bar{\rho} \vdash \bar{x} \Downarrow \bar{\rho}(\bar{x})} \tag{XSOS-PB-AuxVar}$$

$$\frac{\textsc{q}(e,S)}{R \vdash e_{/S} \Downarrow e_{/S}} \tag{XSOS-PB-AT-Refl}$$

Figure 4.9: Abbreviated pretty-big-step XSOS rules for delimited control

[Mos04] that records the structure of the term being abruptly terminated, and which is close in spirit to reduction semantics, as argued in Section 3.7.1. In this thesis we have adapted this notion of abrupt termination to XSOS and (pretty-)big-step semantics, which broadens the applicability of the technique.

A natural question to ask is: could we have used natural semantics instead of pretty-big-step semantics to describe delimited control? The pretty-big-step rules in Figure 4.9 crucially rely on the fact that the pretty-big-step rules preserve the structure of the current term when abruptly terminating: this permits us to construct the continuation by raising raising the control status flag. In contrast, consider if we were to have a standard natural semantics rule for adding natural numbers:

$$\frac{e_1 \Rightarrow n_1 \qquad e_2 \Rightarrow n_2}{\texttt{plus}(e_1,e_2) \Rightarrow n_1 + n_2} \tag{XNS-Plus}$$

In case a control flag is raised during evaluation of e_1, the structure of the `plus`-term is not obviously preserved as abrupt termination is propagated through the derivation tree. The necessary structure for continuations would be recovered by introducing an explicit notion of continuation (or reduction context), similarly to Duba et al. [HDM93], or by recording it implicitly in the structure of derivation tree, similarly to the approach illustrated here.

4.6.4 From delimited control to call/cc

As recalled by Sculthorpe et al. [STM16], delimited control can be used to implement the semantics of many control constructs from the literature, including *shift/reset* [DF90, BD06] and call/cc [SF90]. Our small-step XSOS in Figure 4.8 corresponds to Sculthorpe et al.'s [STM16], differing only in our choice of using XSOS and the state-based representation of modular abrupt termination introduced in Chapter 3. By Theorem 4.12, our pretty-big-step XSOS in Figure 4.9 corresponds to the small-step XSOS.

4.7 Assessment and related work

The work presented in this section provides evidence that XSOS is useful for giving and relating extensible specifications at different levels of abstraction. To reach these insights we have combined ideas from research in relating small-step and big-step evaluation strategies [DN04, Dan09, BD07, Zer13] with research on modular representation of abrupt termination in small-step MSOS [Mos04, STM16]. The correctness proof of refocusing that this Chapter presents also generalises well-known proof techniques for relating small-step and big-step purely structural semantics [Nip06, LG09, NK14, Cio13, PCG^{+}13].

Ciobâcă [Cio13] considered a similar transformation from small-step to big-step rules. Unlike us, he proposed to convert small-step SOS rules directly to natural semantics. As Section 4.6 of this thesis shows, there are definite advantages to converting to pretty-big-step instead. Ciobâcă does not consider modularity or extensibility of rules either.

This chapter showed how to internalise the refocusing transformation in XSOS, and gave a correctness proof based on rule schemas for semantics implementing left-to-right order of evaluation. These correctness criteria are closely related to the criteria given by Danvy and Nielsen in their original exposition on refocusing [DN04]. Sieczkowski et al. [SBB11] has since relaxed the criteria to include context-insensitive reductions. Both of these sets of authors consider *inner-most reduction strategies*, i.e., reduction strategies that search for the *inner-most* redex. This is in contrast to semantics relying on *outer-most reductions*. Johannsen [Joh15] studies techniques for refocusing such semantics. Our correctness criteria and argument is most closely related to the criteria of Danvy and Nielsen, although our notion of left-to-right order of evaluation in XSOS does support contractions that depend on the context (e.g., the environment, exception status flags, etc.). Our rule schemas do not support outermost reduction, however.

The rule schema that we presented here is reminiscent of results proven by several other authors. Ciobâcă provides a rule schema for big-step rules and outlines a transformation and a proof of correspondence between small-step and big-step semantics. The proof assumes that the small-step relation in confluent. In contrast, the definitions of left-to-right evaluation (Definition 4.1 and 4.10) admit non-deterministic choice between axioms but not congruence rules, and thus supports non-confluent relations.

Bodin et al. [BJS15] provide a rule format for pretty-big-step rules and utilise this to provide a framework for abstract interpretation over rules. They also provide a formalisation of this rule format in Coq. The proof we provided in this chapter is purely pen-and-paper (although it mirrors the structure of ad hoc proofs in Coq, as illustrated in the Coq formalisation accompanying this thesis[3]), and it would be interesting to see if their encoding is amenable to formalising the correctness proof of refocusing with abrupt termination in XSOS.

The rule schema introduced in this chapter also draws inspiration from work on formalising and proving meta-theory about (M)SOS [MMR10, CM13, MRG07]. These lines of work study mainly bisimulation equivalences for small-step semantics and process algebras, which typically support non-determinism. Relating the rule format proposed here to these lines of work would give a plethora of meta-theoretical results for free, including that bisimulation is a congruence. Since small-step XSOS rules express transition system semantics that closely correspond to small-step MSOS rules, we expect in particular the results of Churchill and Mosses et al. [MMR10, CM13] to carry over to languages with left-to-right order of evaluation. We leave verification of this expectation to future work.

Our motivation for internalising refocusing in XSOS was that the connection between small-step and pretty-big-step XSOS was so clear and intuitive that we found it unnatural to think of it in terms of reduction contexts. But, on further inspection, the correspondence between small-step XSOS and reduction semantics may be a lot closer than first impressions lead us to believe: as recalled in Section 2.8, some of the main differences between reduction semantics and SOS are the customary use of substitutions rather than environments in reduction semantics; and the need to distinguish different kinds of contexts in order to match the closest enclosing handler. But we do not see any inherent problems with implementing in reduction semantics the modular encodings of features recalled and presented in this thesis. If our expectation that the modular encodings could be implemented in reduction semantics is true, we might rely on Sieczkowski et al.'s correctness proof in order to relate small-step and big-step extensible specifications with their reduction semantic counterparts, instead of the proof based on rule schemas given in this chapter. We leave a further exploration of these expectations to future work.

[3]`http://cs.swansea.ac.uk/~cscbp/xtss.zip`

5 Modular Divergence in Extensible SOS

Contents

Chapter 4 showed how to systematically derive pretty-big-step versions of extensible transition system specifications (and vice versa). The derived relations were inductively defined, whereby it gives semantics to programs with *finite derivation trees*, corresponding to programs that terminate. However, many interesting languages (such as the λ_{cbv} language considered in earlier chapters) have programs that diverge. It is important to support expressing and reasoning about both converging and diverging computations.

In this chapter we present a novel and modular approach to representing divergence in pretty-big-step semantics. Using this approach, we show that pretty-big-step rules produced by the refocusing transformation in the previous section are equivalent to small-step rules for diverging computations too.

5.1 Divergence as modular abrupt termination

Section 2.5 recalled how pretty-big-step rules support more concise specification of abrupt termination and divergence than corresponding natural semantics rules. Following Charguéraud, [Cha13], divergence is represented using a distinct term DIV that is only derivable under the coinductive interpretation and only for terms that diverge. In order to propagate this term to the top-level, Charguéraud uses so-called 'abort rules' (recalled in Figure 2.40). Here, we show how to avoid such abort rules by adapting the encoding of modular abrupt termination to give semantics for modular divergence.

5.1.1 Divergence in small-step and pretty-big-step XSOS

Figure 5.2 summarises the pretty-big-step XSOS rules for λ_{cbv}. Certain computations do not coevaluate using these rules, even though we can prove that they diverge using their small-step semantics. For example applying ω to a stuck term $(0\ 0)$:

$$R \vdash \omega\ (0\ 0)_{/S} \overset{\infty}{\rightarrow}$$

but:

$$\nexists v\ S'.\ R \vdash \omega\ (0\ 0)_{/S} \Downarrow^{\mathrm{co}} v_{/S'}$$

The lack of expressiveness for pretty-big-step coevaluation, in comparison to small-step semantics, is that it entails constructing derivation trees for *all* branches of a computation. Since there are no valid derivations for $(0\ 0)$, the coinductive interpretation $\Downarrow^{\mathrm{co}}$ cannot be used to prove that the term coevaluates.

The lack of flexibility for the coinductive interpretation of big-step relation is not limited to terms with stuck sub-terms: Leroy and Grall [LG09] give Filinski's term[1] as an example of a well-typed term that diverges which is provable using small-step semantics, but does not coevaluate using big-step semantics.[2]

Traditional approaches to avoiding these issues in natural semantics include: giving a separate set of rules for proving divergence, following [CC92, LG09]; using coinductive trace-based rules, following [NU09, Dan12, LG09]; or defining values coinductively, following [Anc12]. Pretty-big-step semantics [Cha13] uses abort rules to avoid the issue, such that divergence is propagated similarly to abrupt termination. Here, we propose to use pretty-big-step semantics, but to propagate divergence using the modular encoding of exceptions instead.

5.1.2 Making divergence syntactically distinguishable

The idea is to add a new auxiliary entity for indicating divergence. Figure 5.1 summarises an extensible rule specification for divergence. Here, *DivStat* is a status flag for indicating either divergence, denoted by $\uparrow$, or convergence, denoted by '$\downarrow$'. There is a single rule for modular divergence, (XSOS-Div). The purpose of this rule is to propagate abrupt termination while disregarding the structure of terms produced by a derivation: the rule relates the configuration to an arbitrary other configuration, and only remembers the state of the divergence flag. Using the approach based on this divergence rule only allows us to infer that a computation diverges, but does not, for example, provide information about potential observable outputs of such divergence. This is analogous to the non-trace-based approaches to representing divergence due to Cousot and Cousot [CC92], Leroy and Grall [LG09], and Charguéraud [Cha13]. Thus, the structure of configurations that that are the outcomes of coevaluation resulting in a divergent state is essentially "garbage". It does, however, make it very straightforward to augment a set of big-step rules to make their coinductive interpretations contain the same set of programs as their small-step counterparts, and to express and reason about these relations in a theorem prover like Coq.

[1] So-called since the discovery of the term is attributed by Leroy and Grall to Filinski.

[2] Although Leroy and Grall uses a substitution-based semantics, we conjecture that the coinductive interpretation of semantics based on environments and closures is equivalent to the coinductive interpretation of semantics based on substitutions.

Abstract syntax.

$$\mathit{DivStat} \ni \delta ::= \uparrow \mid \downarrow$$

Auxiliary entities ($\mathbb{C}_{\text{DIV}}$).

$$\mathbb{C}_{\text{DIV}} \triangleq \{(\mathbf{div}, \mathbb{C}^{\text{AT}}(\mathit{DivStat}))\}$$

Final configurations (Q_{DIV}).

$$\frac{}{\mathrm{Q}_{\text{DIV}}(e, S[\mathbf{div} \uparrow])}$$

Rule specification.

$$\frac{}{R \vdash e_{S[\mathbf{div} \uparrow]} \Downarrow e'_{/S'[\mathbf{div} \uparrow]}} \tag{XSOS-Div}$$

Figure 5.1: Extensible rule specification for divergence

Consider the union of the extensible rule specification for divergence in Figure 5.1 and the extensible rule specification for λ_{cbv} given by the pretty-big-step rules for λ_{cbv} given in Figure 5.2.

Proposition 5.1 (ω diverges using pretty-big-step XSOS and modular divergence) The expression ω diverges in any context, i.e.:

$$\forall R\ \rho\ S\ v\ S'.\ R[\mathbf{env}\ \rho] \vdash \omega_{/S[\mathbf{div} \downarrow]} \Downarrow^{\text{co}} v_{/S'[\mathbf{div} \uparrow]}$$

Proof. The proof is by guarded coinduction. By consecutive applications of (XSOS-PB-λ_{cbv}-App1) and (XSOS-PB-λ_{cbv}-App2) we get the goal:

$$R[\mathbf{env}\ \rho] \vdash \langle x, x\ x, \rho\rangle\ \langle x, x\ x, \rho\rangle_{/S[\mathbf{div} \downarrow]} \Downarrow^{\text{co}} v_{/S'[\mathbf{div} \uparrow]} \tag{Goal}$$

Applying the rule (XSOS-PB-λ_{cbv}-AppC) from Figure 5.2 to the goal, we get the two proof obligations:

$$R[\mathbf{env}\ \rho[x \mapsto \langle x, x\ x, \rho\rangle]] \vdash x\ x_{/S[\mathbf{div} \downarrow]} \Downarrow^{\text{co}} v_{/S'[\mathbf{div} \uparrow]} \tag{Goal1}$$

$$R[\mathbf{env}\ \rho] \vdash \langle x, v, \rho\rangle\ \langle x, x\ x, \rho\rangle_{/S'[\mathbf{div} \uparrow]} \Downarrow^{\text{co}} v_{/S'[\mathbf{div} \uparrow]} \tag{Goal2}$$

The second of these proof obligations follows trivially by (XSOS-PB-λ_{cbv}-App). We can now generalize (Goal1) to get:

$$\forall S.\ R[\mathbf{env}\ \rho[x \mapsto \langle x, x\ x, \rho\rangle]] \vdash x\ x_{/S[\mathbf{div} \downarrow]} \Downarrow^{\text{co}} v_{/S'[\mathbf{div} \uparrow]} \tag{Goal1$'$}$$

We use this goal as our coinduction hypothesis. By applying (XSOS-PB-λ_{cbv}-App1), (XSOS-PB-λ_{cbv}-App2) to (Goal1), we get the goal:

$$R[\mathbf{env}\ \rho[x \mapsto \langle x, x\ x, \rho\rangle]] \vdash \langle x, x\ x, \rho\rangle\ \langle x, x\ x, \rho\rangle_{/S''[\mathbf{div} \downarrow]} \Downarrow^{\text{co}} v_{/S'[\mathbf{div} \uparrow]} \tag{Goal1$''$}$$

Abstract syntax ($\Sigma_{\lambda_{\text{cbv}}}$). (See Figure 4.6)
Auxiliary entities ($\mathbb{C}_{\lambda_{\text{cbv}}}$). (See Figure 4.6)
Final configurations ($Q_{\lambda_{\text{cbv}}}$). Rule specification ($D_{\lambda_{\text{cbv}}}$). (See Figure 4.6)

$$\frac{}{\textbf{env}\ \rho \vdash \lambda x.e \Downarrow \langle x, e, \rho \rangle} \qquad \text{(XSOS-PB-}\lambda_{\text{cbv}}\text{-Lam)}$$

$$\frac{e_1 \Downarrow e_1' \qquad e_1'\ e_2 \Downarrow e'}{e_1\ e_2 \Downarrow e'} \qquad \text{(XSOS-PB-}\lambda_{\text{cbv}}\text{-App1)}$$

$$\frac{e_2 \Downarrow e_2' \qquad \langle x, e, \rho' \rangle\ e_2' \Downarrow e'}{\langle x, e, \rho' \rangle\ e_2 \Downarrow e'} \qquad \text{(XSOS-PB-}\lambda_{\text{cbv}}\text{-App2)}$$

$$\frac{\textbf{env}\ \rho'[x \mapsto v_2] \vdash e \Downarrow e' \qquad \textbf{env}\ \rho \vdash \langle x, e', \rho' \rangle\ v_2 \Downarrow e''}{\textbf{env}\ \rho \vdash \langle x, e, \rho' \rangle\ v_2 \Downarrow e''} \qquad \text{(XSOS-PB-}\lambda_{\text{cbv}}\text{-AppC)}$$

$$\frac{}{\textbf{env}\ \rho \vdash \langle x, v, \rho' \rangle\ v_2 \Downarrow v} \qquad \text{(XSOS-PB-}\lambda_{\text{cbv}}\text{-App)}$$

$$\frac{x \in \text{dom}(\rho)}{\textbf{env}\ \rho \vdash x \Downarrow \rho(x)} \qquad \text{(XSOS-PB-}\lambda_{\text{cbv}}\text{-Var)}$$

$$\frac{\text{Q}(e, S)}{R \vdash e_{/S} \Downarrow e_{/S}} \qquad \text{(XSOS-PB-Iter-Refl)}$$

Figure 5.2: Abbreviated pretty-big-step XSOS rules for λ_{cbv}

Applying (XSOS-PB-λ_{cbv}-AppC), we get a guarded goal that matches the coinduction hypothesis. The remaining goals follow by applying (XSOS-Div). □

Proposition 5.1 provides evidence that the encoding of divergence in Figure 5.1 is useful for making divergence distinguishable in pretty-big-step semantics. In the rest of this chapter, we prove that it enables us to distinguish exactly the same set of programs that we can prove diverge using small-step semantics.

5.2 Correspondence between converging and diverging computations in refocused XSOS rules

Chapter 2 recalled different approaches to representing operational semantics, including how to give semantics for diverging computations, referring in particular to Leroy and Grall [LG09]. Here, we generalise and adapt Leroy and Grall's proofs for proving the correspondence between diverging small-step and pretty-big-step XSOS rules.

$$\frac{R \vdash e_{/S} \rightarrow e'_{/S'} \qquad R \vdash e'_{/S'} \rightarrow^* e''_{/S''}}{R \vdash e_{/S} \rightarrow^* e''_{/S''}} \quad \text{(XSOS-Trans)}$$

$$\frac{}{R \vdash e_{/S} \rightarrow^* e_{/S}} \quad \text{(XSOS-Refl)}$$

$$\frac{R \vdash e_{/S} \rightarrow e'_{/S'} \qquad R \vdash e'_{/S'} \overset{\infty}{\rightarrow}}{R \vdash e_{/S} \overset{\infty}{\rightarrow}} \quad \text{(XSOS-InfClo)}$$

Figure 5.3: The reflexive-transitive closure $\rightarrow^*$ and the infinite closure $\overset{\infty}{\rightarrow}$ of a small-step XSOS transition relation $\rightarrow$

5.2.1 Converging and diverging computations in small-step XSOS

The standard way of reasoning about divergence in small-step semantics is to define a relation $\overset{\infty}{\rightarrow}$ that contains the set of all programs with infinite sequences of reduction steps for some transition relation $\rightarrow$. Such a relation is given by the rule in Figure 5.3.

The relationship between $\overset{\infty}{\rightarrow}$ and $\rightarrow^*$ is summarised by the following Proposition.

Lemma 5.2 For any transition relation $\rightarrow$, it holds that it either gets stuck (either by going wrong, or by yielding a final configuration), or that it progresses indefinitely:

$$\left(\exists e'\ S'.\ R \vdash e_{/S} \rightarrow^* e'_{/S'} \ \wedge\ R \vdash e'_{/S'} \not\rightarrow\right) \ \vee\ R \vdash e_{/S} \overset{\infty}{\rightarrow}$$

Proof (classical). The proof is completely independent of the structure of rules for $\rightarrow$, and is analogous to the one given by Leroy and Grall [LG09, Lemma 10]: we first show $(\forall e'\ S'.\ R \vdash e_{/S} \rightarrow^* e'_{/S'} \implies \exists e''\ S''.\ R \vdash e'_{/S'} \rightarrow e''_{/S''}) \implies R \vdash e_{/S} \overset{\infty}{\rightarrow}$ by guarded coinduction. The goal follows by reasoning by the law of excluded middle on $\overset{\infty}{\rightarrow}$ and this fact. The full proof can be found in the Coq development accompanying this thesis: `http://cs.swansea.ac.uk/~cscbp/xtss.zip`. □

We want to prove that refocusing produces pretty-big-step XSOS relations that can be used to prove divergence on a par with small-step XSOS relations. By Theorem 4.12, it holds that:

$$\mathtt{Q}(e', S') \implies \left(R \vdash e_{/S} \rightarrow^* e'_{/S'} \iff R \vdash e_{/S} \Downarrow e'_{/S'}\right)$$

In this section we prove that:

$$R \vdash e_{/S[\mathbf{div}\downarrow]} \overset{\infty}{\rightarrow} \iff \left(\exists e'\ S'.\ R \vdash e_{/S[\mathbf{div}\downarrow]} \Downarrow^{\mathrm{co}} e'_{/S'[\mathbf{div}\uparrow]}\right)$$

5.2.2 Properties of coinductive interpretation

Section 2.5.2 recalled properties about the coinductive interpretation of pretty-big-step rules for λ_{cbv}. The same properties generalise to the coinductive interpretation of arbitrary extensible pretty-big-step XSOS with left-to-right order of evaluation with abrupt termination (Definition 4.10) and modular divergence (Figure 5.1). These properties are useful for relating the coinductive interpretation of the infinite closure of a small-step relation and pretty-big-step rules with modular divergence.

The first property (Lemma 5.3) says that the set of derivations we can construct using the inductive interpretation of pretty-big-step rules can also be constructed using the coinductive interpretation.

Lemma 5.3 (Coinductive interpretation subsumes inductive interpretation) For any relation $\Downarrow$ that is the refocused counterpart to a small-step relation $\rightarrow$ with left-to-right order of evaluation with abrupt termination, the following holds about its coinductive interpretation $\Downarrow^{\mathrm{co}}$:

$$\begin{aligned}&R \vdash e_{/S} \Downarrow e'_{/S'} \implies \\ &R \vdash e_{/S} \Downarrow^{\mathrm{co}} e'_{/S'}\end{aligned}$$

Proof. Trivial, since $\Downarrow^{\mathrm{co}}$ is the coinductive interpretation of $\Downarrow$. □

Lemma 5.4 proves that a term diverges if it successfully coevaluates but fails to evaluate.

Lemma 5.4 (Coevaluation without convergence implies divergence) For any relation $\Downarrow$ with left-to-right order of evaluation with abrupt termination, the following holds about its coinductive interpretation $\Downarrow^{\mathrm{co}}$:

$$\begin{aligned}&R \vdash e_{/S} \Downarrow^{\mathrm{co}} e'_{/S'} \implies \\ &\neg\left(R \vdash e_{/S} \Downarrow e'_{/S'}\right) \implies \\ &R \vdash e_{/S} \Downarrow^{\mathrm{co}} e'_{/S'[\mathbf{div}\,\uparrow]}\end{aligned}$$

Proof (classical, sketch). The proof is by guarded coinduction on $\Downarrow^{\mathrm{co}}$. The cases follow by reasoning by the law of excluded middle on $\Downarrow$. Cases follow either from a contradiction, or by guarded application of the coinduction hypothesis. The full proof is in Appendix B.2. □

A classical consequence of Lemma 5.4 is that coevaluation implies that a term either diverges or terminates with a value.

Lemma 5.5 (Coevaluation implies convergence or divergence) For any relation $\Rightarrow$ that is the refocused counterpart to a small-step relation $\rightarrow$ with left-to-right order of evaluation with abrupt termination, the following holds about its coinductive interpretation $\Downarrow^{\mathrm{co}}$:

$$\begin{aligned}&R \vdash e_{/S} \Downarrow^{\mathrm{co}} e'_{/S'} \implies \\ &R \vdash e_{/S} \Downarrow e'_{/S'} \vee R \vdash e_{/S} \Downarrow^{\mathrm{co}} e'_{/S'[\mathbf{div}\,\uparrow]}\end{aligned}$$

Proof (classical). The goal follows by reasoning by the law of excluded middle on $R \vdash e_{/S} \Downarrow e'_{/S'}$ and Lemma 5.4.

Case (P) Case analysis on the law of excluded middle gives:

$$R \vdash e_{/S} \Downarrow e'_{/S'} \tag{H1}$$

This proves the left part of the disjunctive goal.

Case $(\neg P)$ The goal and case analysis on the law of excluded middle gives:

$$R \vdash e_{/S} \Downarrow^{co} e'_{/S'} \tag{H1}$$

$$\neg(R \vdash e_{/S} \Downarrow e'_{/S'}) \tag{H2}$$

Applying Lemma 5.4 to these hypotheses proves the right part of the disjunctive goal.

□

Using these properties, we can prove that the set of divergent programs admitted by the coinductive interpretation of refocused pretty-big-step rules coincides with the set of divergent programs admitted by the infinite closure of the corresponding small-step transition relation.

5.2.3 Soundness

First, we prove that the set of divergent programs admitted by the coinductive interpretation of refocused pretty-big-step rules with modular abrupt termination is a subset of the set of divergent programs admitted by the infinite closure of the corresponding small-step relation.

Lemma 5.6 (Coinductive refocusing is sound) For any relation $\Downarrow$ that is the refocused counterpart to a small-step relation $\rightarrow$ with left-to-right order of evaluation with abrupt termination, the following holds about its coinductive interpretation $\Downarrow^{co}$:

$$\begin{aligned} &R \vdash e_{/S[\mathbf{div}\downarrow]} \Downarrow^{co} e'_{/S'[\mathbf{div}\uparrow]} \implies \\ &R \vdash e_{/S} \overset{\infty}{\rightarrow} \end{aligned}$$

Proof. The proof is by guarded coinduction, using the goal as coinduction hypothesis. By Lemma 5.7, we get the hypotheses:

$$R \vdash e_{/S[\mathbf{div}\downarrow]} \rightarrow e''_{/S''[\mathbf{div}\downarrow]} \tag{H1}$$

$$R \vdash e''_{/S''} \Downarrow^{co} e'_{/S'[\mathbf{div}\uparrow]} \tag{H2}$$

The goal:

$$R \vdash e_{/S} \overset{\infty}{\rightarrow} \tag{Goal}$$

follows by application of (XSOS-InfClo), (H1), a guarded application of the coinduction hypothesis, and (H2). □

Lemma 5.7 (Small-step preserves coinductive refocusing) For any relation $\Downarrow$ that is the refocused counterpart to a small-step relation $\rightarrow$ with left-to-right order of evaluation with abrupt termination, the following holds about its coinductive interpretation $\Downarrow^{\mathrm{co}}$:

$$\begin{array}{l} R \vdash e_{/S[\mathbf{div}\downarrow]} \Downarrow^{\mathrm{co}} e'_{/S'[\mathbf{div}\uparrow]} \implies \\ \exists e''\ S''.\ R \vdash e_{/S[\mathbf{div}\downarrow]} \rightarrow e''_{/S''} \ \wedge\ R \vdash e''_{/S''} \Downarrow^{\mathrm{co}} e'_{/S'[\mathbf{div}\uparrow]} \end{array}$$

Proof (classical, sketch). The proof is by structural induction on e, invoking the law of the excluded middle on instantiations of Lemma 5.5, and Theorem 4.12. See Appendix B.2 for the full proof. □

5.2.4 Completeness

Second, we prove the other direction; i.e., that the set of divergent programs admitted by the infinite closure of a small-step transition relation is a subset of the set of divergent programs admitted by the coinductive interpretation of corresponding refocused pretty-big-step rules with modular abrupt termination.

Lemma 5.8 (Coinductive refocusing is complete) For any relation $\Downarrow$ that is the refocused counterpart to a small-step relation $\rightarrow$ with left-to-right order of evaluation with abrupt termination, the following holds about its coinductive interpretation $\Downarrow^{\mathrm{co}}$:

$$R \vdash e_{/S} \overset{\infty}{\rightarrow} \implies \forall e'\ S'.\ R \vdash e_{/S} \Downarrow^{\mathrm{co}} e'_{/S'[\mathbf{div}\uparrow]}$$

Proof (classical, sketch). The proof is by guarded coinduction, using the goal as coinduction hypothesis, and by inversion on the first premise, using classical reasoning for case analysis on whether a term diverges or not, Theorem 4.12, and congruence lemmas of similar structure to Lemma 5.9. See Appendix B.2 for the full proof. □

Lemma 5.9 (Congruence of infinite closure) For any small-step relation $\rightarrow$ with left-to-right order of evaluation with abrupt termination and with a rule that matches the scheme:

$$\forall e_i \ldots e_n\ S\ S'.\ \frac{\begin{array}{c}\neg \mathrm{Q}(e_i, S) \\ R' \vdash e_{i/S} \rightarrow e'_{i/S'}\end{array}}{\begin{array}{c}R \vdash f(v_1,\ldots,v_{i-1},e_i,e_{i+1},\ldots,e_n,\ldots)_{/S} \rightarrow \\ f(v_1,\ldots,v_{i-1},e'_i,e_{i+1},\ldots,e_n,\ldots)_{/S'}\end{array}} \qquad (\text{XRS-}fi)$$

For each such rule, it holds that:

$$\begin{array}{l} R \vdash f(v_1,\ldots,v_n,e_i,e_{i+1},\ldots,e_n,\ldots)_{/S} \overset{\infty}{\rightarrow} \implies \\ \neg(\exists e'\ S'.\ R' \vdash e_{i/S} \rightarrow^* e'_{/S'} \ \wedge\ \mathrm{Q}(e',S') \ \wedge \\ \qquad R \vdash f(v_1,\ldots,v_n,e',e_{i+1},\ldots,e_n,\ldots)_{/S'} \overset{\infty}{\rightarrow}) \implies \\ R' \vdash e_{i/S} \overset{\infty}{\rightarrow} \end{array}$$

Proof (sketch). The proof is by guarded coinduction and inversion on the first premise. See Appendix B.2 for the full proof. □

The proof of completeness given by Lemmas 5.9 and 5.8 differs from the proof method followed by Leroy and Grall [LG09, Lemma 10 and Theorem 11]: Leroy and Grall's proof is based on a semantics with a deterministic transition relation; in contrast, Definitions 4.1 and 4.11 do not describe deterministic relations; instead, they describe relations that satisfy what is known as *unique decomposition* in reduction semantics [XSA01, DN04]. Intuitively, non-deterministic choices between rules can only ever occur in the leaves of derivation trees for such semantics.

5.2.5 Correctness of coinductive refocusing with modular divergence

Lemma 5.6 and 5.8 show that refocusing produces pretty-big-step rules which support reasoning about divergence on a par with small-step semantics. Theorem 5.10 summarises the result.

Theorem 5.10 For any relation $\Downarrow$ that is the refocused counterpart to a small-step relation $\rightarrow$ with left-to-right order of evaluation with abrupt termination, the following holds about its coinductive interpretation $\Downarrow^{\text{co}}$:

$$R \vdash e_{/S[\mathbf{div}\downarrow]} \overset{\infty}{\rightarrow} \iff R \vdash e_{/S[\mathbf{div}\downarrow]} \Downarrow^{\text{co}} e'_{/S'[\mathbf{div}\uparrow]}$$

Proof. The theorem is a direct consequence of Lemma 5.6 and 5.8. □

This shows that it is possible to augment extensible big-step specifications to express and reason about diverging computations without modifying or introducing new rules for existing constructs.

5.3 Assessment and related work

We have shown how our approach to encoding modular abrupt termination is useful for expressing and reasoning about diverging computations. Chapter 2 recalled traditional approaches to semantic specifications and a few state-of-the-art approaches. Several alternative lines of work exist for reasoning about divergence.

5.3.1 Refocused rules vs. pretty-big-step semantics

One of the main differences between the pretty-big-step rules considered in this thesis and those of Charguéraud [Cha13] is in how we represent abrupt termination and divergence. In order to distinguish final configurations, we introduced a Q predicate in Section 3. Using this predicate, XSOS only match on non-final configurations. A consequence of this is that pretty-big-step XSOS rules ensure productivity of rules, similarly to how semantic expressions ensure productivity of Charguéraud's [Cha13] original formulation of pretty-big-step rules.

5.3.2 Trace-based semantics

The encoding of divergence considered here suffices to show that a term diverges, but provides no fine-grained information about potential observable outputs of diverging computations. Trace-based semantics do.

It is straightforward to record (possibly-infinite) traces using small-step semantics. For example, the following closure of a small-step XSOS transition relation implements the sequence of all intermediate states of a program:

$$\frac{R \vdash e_{/S} \to e_{/S'} \qquad R \vdash e_{/S'} \to^{\mathrm{T}} t}{R \vdash e_{/S} \to^{\mathrm{T}} S :: t} \qquad \text{(XSOS-Trace-Trans)}$$

$$\frac{}{R \vdash e_{/S} \to^{\mathrm{T}} \langle S \rangle} \qquad \text{(XSOS-Trace-Refl)}$$

It is equally possible to give big-step semantics that record such traces.

Leroy and Grall [LG09] proposed a version of trace-based big-step semantics, where traces are either finite lists specified by an inductive data type, or infinite streams, specified by a coinductive datatype. Using these datatypes, Leroy and Grall define two separate big-step relations: one relation that produces finite lists as traces, and one that produces infinite streams. Their rules suffer from a similar duplication problem as traditional big-step rules, and rely on classical reasoning similarly to the proofs considered in this chapter in order to reason about possibly-diverging computations.

Nakata and Uustalu [NU09] propose an alternative means of giving trace-based semantics, inspired by how traces are accumulated using the partiality monad [Cap05]. Following their approach, the coinductive interpretation of a set of rules subsumes the set of all finite and infinite traces, where a trace is a co-list, i.e., a possibly infinite list. Unlike the semantics of Leroy and Grall, Nakata and Uustalu's rules do not suffer from the duplication problem, and does not rely on classical reasoning. The language in which Nakata and Uustalu showcase their rules is of imperative nature, but Danielsson [Dan12] shows how to work with applicative semantics by means of functional operational semantics using the partiality monad, which closely corresponds to trace-based coinductive big-step semantics.

Unlike the approaches to representing and reasoning about possible-divergence considered in this chapter, Nakata and Uustalu's trace-based coinductive big-step support constructive reasoning in order to reason about and relate possibly-diverging programs and semantics. However, most proof assistants like Coq or Agda have somewhat limited support for coinductive reasoning, which is more restrictive than inductive reasoning. Furthermore, while the extra structure that trace-based semantics provides is useful for many applications, there are also applications where all we are concerned with is whether a program diverges or terminates with a final configuration, such as type systems (see, e.g., Chapters 6, 7, and 8 of this thesis).

Charguéraud [Cha13] shows how pretty-big-step rules scale to express trace-based semantics too. His semantics is closer in spirit to Leroy and Grall's than Nakata and Uustalu, in that finite and infinite streams are distinguished inductive and coinductive datatypes. However, Charguéraud's trace-based pretty-big-step rules avoid the duplication problem. We expect that it is straightforward to adapt trace-based semantics to XSOS. This could conceivably be done by a monad-like lifting of pretty-big-step rules. Alternatively, trace-based rules could be derived by refocusing the trace-based closure of small-step rules (XSOS-Trace-Trans) and (XSOS-Trace-Refl) in relation to a set of small-step rules to obtain big-step trace-based semantics. We leave further exploration of this expectation to future work.

This chapter has extends the state-of-the-art in representing divergence in three ways:

- it provides a modular way of extending pretty-big-step semantics with simple divergence without abort rules;
- it provides a generic proof of correspondence for pretty-big-step and small-step semantics; and
- it provides a generalised proof method for relating a small-step semantics to a big-step semantics using guarded coinduction in a way that does not rely on the determinism of the small-step relation.

6 Big-Step Type Soundness using Types as Abstract Interpretations

Contents

Section 2.9.4 recalled how Cousot's [Cou97] types as abstract interpretations provide a means of proving type soundness using big-step semantics with an explicit notion of "going wrong". This chapter shows that the approach also works for proving type soundness using natural semantics without artificial wrong transitions, and how to implement the approach in a proof assistant like Coq.

The contents of this chapter is based on joint work [BPMT15] with Paolo Torrini and Peter D. Mosses.

6.1 Monotype abstraction of big-step λ_{cbv}

We consider how to abstract λ_{cbv} using the natural semantics from Figure 2.27 without artificial wrong transitions.

Types and denotations. As in Section 2.9, we consider the simple types given by the grammar:

$$\mathit{Type} \ni T ::= \mathit{nat} \mid T \rightarrow T$$

Type environments and typings are also defined as before:

$$\Gamma \in \mathit{TypeEnv} \triangleq \mathit{Var} \xrightarrow{\text{fin}} \mathit{Type}$$

$$\mathit{Typing} \triangleq \mathit{TypeEnv} \times \mathit{Type}$$

$$\mathbb{T} \triangleq \wp(\mathit{Typing})$$

We define the denotational meaning of expressions in terms of the relation given by the inductive rules in Figure 2.27 without explicit rules for going wrong:

$$\mathbf{D}[\![\bullet]\!] \in \mathit{Expr} \to \mathbb{D}$$
$$\mathbf{D}[\![e]\!] \triangleq \Lambda\rho.\{v \mid \rho \vdash e \Rightarrow v\}$$

Here, $\mathbb{D} \triangleq \mathit{Env} \to \wp(\mathit{Val})$. The denotation function above differs from the denotation function recalled in Section 2.9.4: the denotation function defined above returns the empty set if a program gets stuck or diverges. For example, the denotation of $\mathbf{D}[\![x]\!](\emptyset)$ (i.e., a free variable) gives the empty set. In contrast, since we were using a semantics with an explicit notion of going wrong, the sets returned by the denotations function that we adapted from Cousot in Section 2.9.4 is guaranteed to be non-empty since the union of $\Rightarrow$ and $\overset{\infty}{\Rightarrow}$ from Section 2.9.4 is total for expressions and environments.

Abstraction function. Section 2.9.4 followed Cousot and used a concretisation function that gives meanings to types by relating sets of typings to sets of denotations. Since the concretisation and abstraction functions uniquely define one another, we might equally well use an abstraction function. Here, we opt for formalising the abstraction function, which relates denotations to their type.[1] The abstraction function we want is one that returns the set of all typings that can be assigned to the denotation of a given program. Figure 6.1 defines such an abstraction function.

Like the concretisation function we adapted from Cousot [Cou97, p. 317] in Figure 2.58, the abstraction function assigns typings to denotations in a conjunctive manner; i.e., the set of typings one can assign to a set of denotations D are valid typings for all denotations $d \in D$. Recalling that denotation functions return the empty set for faulty programs, it follows that, if a set of denotations D contains a faulty program with denotation d, it is not assigned a type, since $\alpha_D(d) = \emptyset$.

Using the abstraction function, we can establish the following Galois connection, where γ is uniquely defined by α:

$$(\mathbb{T}, \supseteq) \underset{\gamma}{\overset{\alpha}{\leftrightarrows}} (\wp(\mathbb{D}), \subseteq)$$

The Galois connection follows from the same line of reasoning as in Section 2.9.4.

Type soundness. Figure 6.2 recalls the typing relation for λ_{cbv}. Using that relation, we define the following typing function which returns a set of typings for a given expression:

$$\mathbf{T}[\![e]\!] \triangleq \{(\Gamma, T) \mid \Gamma \vdash e : T\}$$

[1]Our motivation for opting for abstraction over concretisation is didactic: the definition of the abstraction function reads similarly to a traditional typing relation. A concretisation function captures the same meaning, but reads in the opposite direction.

$$
\begin{aligned}
\alpha_R &\in \mathit{Env} \to \wp(\mathit{TypeEnv}) \\
\alpha_R(\rho) &\triangleq \{\Gamma \mid \forall x \in \mathrm{dom}(\rho).\ x \in \mathrm{dom}(\Gamma) \ \wedge\ \Gamma(x) \in \alpha_V(\rho(x))\} \\
\alpha_V &\in \mathit{Val} \to \wp(\mathit{Type}) \\
\alpha_V(n) &\triangleq \{\mathit{nat}\} \\
\alpha_V(\langle x, e, \rho\rangle) &\triangleq \left\{ T_1 \rightarrow T_2 \,\middle|\, \begin{array}{l} \forall v_1.\ T_1 \in \alpha_V(v_1) \implies \\ \qquad \exists v_2.\ v_2 \in \mathbf{D}[\![e]\!](\rho[x \mapsto v_1]) \ \wedge\ T_2 \in \alpha_V(v_2) \end{array} \right\} \\
\alpha_D &\in \mathbb{D} \to \mathbb{T} \\
\alpha_D(d) &\triangleq \{(\Gamma, T) \mid \forall \rho.\ \Gamma \in \alpha_R(\rho) \implies \exists v.\ v \in d(\rho) \ \wedge\ T \in \alpha_V(v)\} \\
\alpha &\in \wp(\mathbb{D}) \to \mathbb{T} \\
\alpha(D) &\triangleq \bigcap_{d \in D} \alpha_D(d) \qquad\qquad \alpha(\emptyset) \triangleq \mathbb{T}
\end{aligned}
$$

Figure 6.1: Abstraction function for λ_{cbv}

$$\frac{\Gamma \vdash e_1 : \mathit{nat} \qquad \Gamma \vdash e_2 : \mathit{nat}}{\Gamma \vdash \texttt{plus}(e_1, e_2) : \mathit{nat}} \tag{T-Plus}$$

$$\frac{}{\Gamma \vdash \texttt{num}(n) : \mathit{nat}} \tag{T-Nat}$$

$$\frac{\Gamma[x \mapsto T_1] \vdash e : T_2}{\Gamma \vdash \lambda x.e : T_1 \rightarrow T_2} \tag{T-Fun}$$

$$\frac{\Gamma \vdash e_1 : T_1 \rightarrow T_2 \qquad \Gamma \vdash e_2 : T_1}{\Gamma \vdash e_1\ e_2 : T_2} \tag{T-App}$$

$$\frac{x \in \mathrm{dom}(\Gamma)}{\Gamma \vdash x : \Gamma(x)} \tag{T-Var}$$

Figure 6.2: Typing rules for λ_{cbv}

Proving that this typing relation safely approximates the set of all typings for well-typed programs amounts to proving type soundness; i.e.:

$$\begin{aligned}
& \mathbf{T}[\![e]\!] \subseteq \alpha(\{\mathbf{D}[\![e]\!]\}) && \\
\iff\ & \mathbf{T}[\![e]\!] \subseteq \alpha_D(\mathbf{D}[\![e]\!]) && \text{(by def. of } \alpha\text{)} \\
\iff\ & (\Gamma, T) \in \mathbf{T}[\![e]\!] \implies \forall \rho.\, \Gamma \in \alpha_R(\rho) \implies && \\
& \quad \exists v.\ v \in \mathbf{D}[\![e]\!](\rho) \ \wedge\ T \in \alpha_V(v) && \text{(by def. of } \alpha_D \text{ and } \subseteq\text{)} \\
\iff\ & \Gamma \vdash e : T \implies \forall \rho.\, \Gamma \in \alpha_R(\rho) \implies && \\
& \quad \exists v.\ \rho \vdash e \Rightarrow v \ \wedge\ T \in \alpha_V(v) && \text{(by def. of } \mathbf{T} \text{ and } \mathbf{D}\text{)}
\end{aligned}$$

Proposition 6.1 (Type soundness)

$$\begin{aligned}
& \Gamma \vdash e : T \implies \forall \rho.\, \Gamma \in \alpha_R(\rho) \implies \\
& \quad \exists v.\ \rho \vdash e \Rightarrow v \ \wedge\ T \in \alpha_V(v)
\end{aligned}$$

Proof. By structural induction on the typing relation. The interesting cases are those for λ-abstractions and application.

Case (T-Fun) Unfolding the abstraction function in the conclusion, we are required to show that the body of the λ-abstraction when applied to a value of type T_1 returns a value of T_2. This fact follows by invoking Lemma 6.2 (see below) to show that $\Gamma[x \mapsto T_1] \in \alpha_R(\rho[x \mapsto v_1])$ for $T_1 \in \alpha_V(v_1)$, and by the induction hypothesis.

Case (T-App) The goal follows straightforwardly from the induction hypothesis. The crucial step is proving that the body of the abstraction evaluates to a value of the right type. This follows from the induction hypothesis for the first premise, which gives us $T_1 \rightarrowtriangle T_2 \in \alpha_V(\langle x, e, \rho \rangle)$, which, by definition of α_V, gives us:

$$\forall v_1.\ T_1 \in \alpha_V(v_1) \implies \exists v_2.\ \rho[x \mapsto v_1] \vdash e \Rightarrow v_2 \ \wedge\ T_2 \in \alpha_V(v_2)$$

which is exactly what we need in order to complete the proof.

□

Lemma 6.2 (Abstraction preserves environment updates)

$$\Gamma \in \alpha_R(\rho) \implies T \in \alpha_V(v) \implies \Gamma[x \mapsto T] \in \alpha_R(\rho[x \mapsto v])$$

Proof. Immediate from the definition of α_R. □

This shows that abstract interpretation allows us to prove strong type soundness using big-step relations without artificial “wrong” transitions. The key is the precision afforded by the abstraction function for closures: the abstraction function gives us the information that is lacking from the induction hypothesis, namely that closures do not get stuck.

A corollary of the proof of strong type soundness based on inductive natural semantics is that the λ-calculus is *strongly normalising*, i.e., all computations terminate. While this is well-known [Bar92, Tai67, Pie02]. Here, the fact follows naturally from the type soundness proof itself.

Corollary 6.3 The simply-typed λ-calculus is strongly normalising.

Proof. The corollary is a straightforward consequence of Proposition 6.1, which establishes that, for any λ_{cbv} program that type-checks, there exists a *finite* derivation which produces a value of the right type. Since λ_{cbv} is deterministic, it follows that all derivations that type-check are finite. $\square$

6.2 Well-foundedness and proof mechanisation in Coq

The proof of Proposition 6.1 follows by induction and from the definition of our abstraction functions. The abstraction functions are defined as a set computed by recursive equations. When dealing with self-referential set-theoretic functions, we must take care that they do not give rise to paradoxes that would allow us to prove anything. For this reason, most proof assistants have rigorous checks in place in order to ensure that user-defined relations and functions preserves the consistency of the underlying logic. In this section we consider how to work with abstraction functions in Coq.

First attempt: relational encoding. Looking at the abstraction function for values, α_V, it is not immediately obvious that the function is well-founded. Indeed, a relational encoding of α_V is rejected by Coq[2]. We may attempt to define it for the following encoding of the λ_{cbv} language:

```
Inductive Expr := V (v:Val)
                | PLUS (e1 e2:Expr)
                | VAR (x:nat)
                | LAM (x:nat) (e:Expr)
                | APP (e1 e2:Expr)
     with Val  := NAT (n:nat)
                | CLO (x:nat) (e:Expr) (r:Map Val).

Inductive type := N
                | FUN (T1 T2:type).
```

Now, we can attempt to define the following relation, ‘`In_AlphaV`’, with two rules, ‘`IA_NAT`’ and ‘`IA_CLO`’:

[2]Version 8.4pl6

```
Inductive In_AlphaV : type -> Val -> Prop :=
| IA_NAT :
    forall n,
      In_AlphaV N (NAT n)
| IA_CLO :
    forall T1 T2 x e r v2,
      (In_AlphaV T2 v2 ->
       (exists v1, Eval (map_update r x v2) e v1 /\
                   In_AlphaV T1 v1)) ->
      In_AlphaV (FUN T1 T2) (CLO x e r).
```

Here, Eval is the Coq counterpart to the relation given by the natural semantics for λ_{cbv} (Figure 2.27). But when we try to load this definition in Coq, we get the error message:

```
Error: Non strictly positive occurrence of "In_AlphaV" in
 "forall (T1 T2 : type) (x : nat) (e : Expr) (r : Map) (v2 : Val),
  (In_AlphaV T2 v2 ->
   exists v1:Val, Eval (map_update r x v2) e v1 /\ In_AlphaV T1 v1) ->
  In_AlphaV (FUN T1 T2) (CLO x e r)".
```

The strict-positivity requirement [CPM90] is in place to rule out a class of non-well-founded relations that would make the logic of the underlying proof assistants inconsistent. Unfortunately, the requirement also rules out certain relations that do not give rise to inconsistencies. Here, the non-strictly-positive occurrence of In_AlphaV is the occurrence to the left of the arrow in the premise for the constructor IA_CLO. In order to work with the abstraction function in Coq we resort to a functional encoding of the abstraction function.

Functional encoding. Recursive function in Coq are given as Fixpoints that must be guaranteed to always terminate. Noticing that each recursive call of α_V is on a type term that is strictly smaller than the input type term, we can express the function such that it is accepted by Coq's termination checker as structurally decreasing on the input type term, T:

```
Fixpoint in_alphaV (T:type) (v:Val) : Prop :=
  match v with
    | NAT n => T = N
    | CLO x e r =>
      match T with
        | FUN T2 T1 =>
          forall v2,
            in_alphaV T2 v2 ->
            exists v1, Eval (map_update r x v2) e v1 /\
                      in_alphaV T1 v1
        | _ => False
      end
  end.
```

This function generates a proposition corresponding to the set of all pairs (T, v) for which it holds that $T \in \alpha_V(v)$.

It is straightforward to implement the abstraction function for environments α_R from Figure 6.1 in Coq using a functional encoding. The Coq formalisation accompanying this thesis[3] contains the full formalisation, as well as Coq counterpart to the type soundness proof for λ_{cbv} that Proposition 6.1 outlines.

6.3 Polytype abstraction

Cousot [Cou97] shows how to give concretisation functions for Hindley-Milner-Damas polymorphism [Hin69, Mil78, Dam84], the typing discipline used in Standard ML. Whereas Cousot uses a concretisation function and a denotational semantics with an explicit notion of going wrong, this section defines an abstraction function and uses this to define type soundness of Hindley-Milner-Damas polymorphism using a natural semantics without an explicit notion of going wrong.

Extension with let. Hindley-Milner-Damas polymorphism is also known as let-polymorphism, since polymorphic quantification is restricted to `let` expressions. We augment the natural semantics for λ_{cbv} (Section 2.4.4) by a `let` construct:

$$\mathit{Expr}^{\mathrm{NS}} \ni e ::= \ldots \mid \mathtt{let}\ x = e\ \mathtt{in}\ e$$

$$\frac{\rho \vdash e_2 \Rightarrow v_2 \qquad \rho[x \mapsto v_2] \vdash e_1 \Rightarrow v_1}{\rho \vdash \mathtt{let}\ x = e_2\ \mathtt{in}\ e_1 \Rightarrow v_1}$$

Notion of typing. Traditionally, Hindley-Milner types [Hin69, Mil78, WF94] use syntactic substitution and unification. Here, we follow Cousot [Cou97] and define polytypes as sets of monotypes:

$$\Xi \in \mathit{PolyTypeEnv} \triangleq \mathit{Var} \xrightarrow{\text{fin}} \wp(\mathit{Type})$$

Our notion of typing becomes:

$$(\Xi, T) \in \mathbb{P} \triangleq \mathit{PolyTypeEnv} \times \mathit{Type}$$

Abstraction function. Figure 6.3 defines abstraction functions for polytype environments. The only differences (apart from naming) are in the definitions of α_R^{HM} and α_D^{HM}, which now refer to polytype environments.

[3] `http://cs.swansea.ac.uk/~cscbp/xtss.zip`

$$\begin{aligned}
\alpha_R^{\mathrm{HM}} &\in \mathit{Env} \to \wp(\mathit{PolyTypeEnv}) \\
\alpha_R^{\mathrm{HM}}(\rho) &\triangleq \left\{ \Xi \;\middle|\; \begin{array}{l} \forall x \in \mathrm{dom}(\rho). \\ \qquad x \in \mathrm{dom}(\Xi) \wedge \Xi(x) \neq \emptyset \wedge \Xi(x) \subseteq \alpha_V^{\mathrm{HM}}(\rho(x)) \end{array} \right\} \\
\alpha_V^{\mathrm{HM}} &\in \mathit{Val} \to \wp(\mathit{Type}) \\
\alpha_V^{\mathrm{HM}}(n) &\triangleq \{\mathit{nat}\} \\
\alpha_V^{\mathrm{HM}}(\langle x, e, \rho \rangle) &\triangleq \left\{ T_1 \rightarrow T_2 \;\middle|\; \begin{array}{l} \forall v_1.\ T_1 \in \alpha_V^{\mathrm{HM}}(v_1) \implies \\ \qquad \exists v_2.\ v_2 \in \mathbf{D}[\![e]\!](\rho[x \mapsto v_1]) \wedge T_2 \in \alpha_V^{\mathrm{HM}}(v_2) \end{array} \right\} \\
\alpha_D^{\mathrm{HM}} &\in \mathbb{D} \to \mathbb{P} \\
\alpha_D^{\mathrm{HM}}(d) &\triangleq \{(\Xi, T) \mid \forall \rho.\ \Xi \in \alpha_R^{\mathrm{HM}}(\rho) \implies \exists v.\ v \in d(\rho) \wedge T \in \alpha_V^{\mathrm{HM}}(v)\} \\
\alpha^{\mathrm{HM}} &\in \wp(\mathbb{D}) \to \mathbb{P} \\
\alpha^{\mathrm{HM}}(D) &\triangleq \bigcap_{d \in D} \alpha_D^{\mathrm{HM}}(d) \qquad\qquad \alpha^{\mathrm{HM}}(\emptyset) \triangleq \mathbb{P}
\end{aligned}$$

Figure 6.3: Abstraction function for λ_{cbv} with Hindley-Milner-Damas polymorphism

Type soundness. Figure 6.4 defines a typing relation for Hindley-Milner-Damas polymorphism. The main difference from the monotype system in Figure 2.55 is the rule for let (P-Let), which infers a non-empty set of monotypes P (i.e., a polytype) that contains valid types for e_2. The rule binds x to the polytype P when inferring the type of e_1. Here, it is crucial that the polytype P is non-empty: if we were to permit it to be empty, we would be able to infer the types of some `let` expressions that contain stuck-terms, such as:

$$\mathtt{let}(x, \mathtt{plus}(\mathtt{num}(1), (\lambda y.y)), \mathtt{num}(2))$$

Proposition 6.4 (Polytyping is sound for λ_{cbv} with `let` expressions)

$$\Gamma \vdash e : T \implies \forall \rho.\ \Gamma \in \alpha_R(\rho) \implies \exists v.\ \rho \vdash e \Rightarrow v \wedge T \in \alpha_V(v)$$

Proof. The proof follows the same line of reasoning as Proposition 6.1, except that Lemma 6.5 is used instead of Lemma 6.2, and with the addition of the new case for `let`.

Case (P-Let) The induction hypotheses are:

$$\forall T \in P.\ \forall \rho.\ \Xi \in \alpha_R^{\mathrm{HM}}(\rho) \implies \exists v_1.\ \rho \vdash e_1 \Rightarrow v_1 \wedge T_1 \in \alpha_V^{\mathrm{HM}}(v_1) \tag{IH1}$$

$$\forall \rho.\ \Xi[x \mapsto P] \in \alpha_R(\rho) \implies \exists v_2.\ \rho \vdash e_2 \Rightarrow v_2 \wedge T_2 \in \alpha_V(v_2) \tag{IH2}$$

The goal is to find a value v_1 that e_1 evaluates to, and which is typed by all $T \in P$. If we can show that such a value exists, then the goal follows straightforwardly from

$$\frac{\Xi \vdash e_1 : nat \qquad \Xi \vdash e_2 : nat}{\Xi \vdash \mathtt{plus}(e_1, e_2) : nat} \tag{P-Plus}$$

$$\frac{}{\Xi \vdash n : nat} \tag{P-Nat}$$

$$\frac{T \in \Xi(x)}{\Xi \vdash x : T} \tag{P-Var}$$

$$\frac{\Xi[x \mapsto \{T_2\}] \vdash e : T_1}{\Xi \vdash \lambda x.e : T_2 \twoheadrightarrow T_1} \tag{P-Fun}$$

$$\frac{\Xi \vdash e_1 : T_2 \twoheadrightarrow T_1 \qquad \Xi \vdash e_2 : T_2}{\Xi \vdash e_1\ e_2 : T_1} \tag{P-App}$$

$$\frac{P \neq \emptyset \qquad (\forall T \in P.\ \Xi \vdash e_2 : T) \qquad \Xi[x \mapsto P] \vdash e_1 : T_1}{\Xi \vdash \mathtt{let}\ x = e_2\ \mathtt{in}\ e_1 : T_1} \tag{P-Let}$$

Figure 6.4: Typing rules for λ_{cbv} with Hindley-Milner-Damas polymorphism

the induction hypotheses and Lemma 6.5. The existence of such a value is proven as follows:

- Choose any $T_1 \in P$.
- From (IH1), we get $\rho \vdash e_1 \Rightarrow v_1$ such that $T_1 \in \alpha_V(v_1)$.
- Observe that v_1 must be typed by any $T \in P$: for any $T \in P$, we can use (IH1) to obtain a proof of $\rho \vdash e_1 \Rightarrow v'_1$ such that $T \in \alpha_V(v'_1)$; but since $\Rightarrow$ is deterministic, we get $T \in \alpha_V(v_1)$.

The rest of the case follows from the induction hypotheses, Lemma 6.5, and this fact.

□

Lemma 6.5 (Abstraction preserves environment updates)

$$\Xi \in \alpha_R^{\text{HM}}(\rho) \implies P \neq \emptyset \implies (\forall T \in P.\ T \in \alpha_V^{\text{HM}}(v)) \implies \Xi[x \mapsto P] \in \alpha_R^{\text{HM}}(\rho[x \mapsto v])$$

Proof. Immediate from the definition of α_R^{HM}. □

As with the proof of correctness for the simple type system considered in Section 6.1, strong normalisation for λ-calculus with Hindley-Milner-Damas polymorphism follows as a straightforward corollary of the type soundness in Proposition 6.4.

Corollary 6.6 λ-calculus with Hindley-Milner-Damas polymorphism is strongly normalising.

Proof. The corollary is a straightforward consequence of Proposition 6.4, which establishes that, for any λ_{cbv} program that type-checks, there exists a *finite* derivation which produces a value of the right type. Since λ_{cbv} is deterministic, it follows that all derivations that type-check are finite. □

6.4 Assessment and related work

Types as abstract interpretations has been considered by several authors. Most of these use denotational semantics with explicit notions of going wrong [Cou97, MJ86, CF93, Sim14, Gal14]. The close relationship between denotational semantics and natural semantics makes it straightforward to apply in the context of this framework; indeed, Monsuez [Mon95] studies types as abstract interpretations for System F [Gir72, Rey74] using natural semantics with an explicit notion of going wrong.[4]

The main contribution of this chapter consists in highlighting the fact that abstract interpretation does not require us to use a semantics with an explicit notion of going wrong: it suffices to have a sufficiently strong relation between types and their meaning. This follows straightforwardly by viewing types as abstract interpretations following Cousot [Cou97], and as investigated by others. Part of the reason why this fact has received so little attention may be that that adding the notion of "wrong" to denotational semantics is straightforward to express by means of pattern-matching [CF93, p. 120]. This is more challenging in natural semantics, where divergence and going wrong is typically indistinguishable, in the sense that neither diverging nor wrong programs are contained in the defined relation. This chapter shows that this distinction is not inherently necessary for type disciplines that ensure strong normalisation. In the next chapter, we consider how to extend the approach to an ML-style type system where strong normalisation is not guaranteed.

Ancona [Anc12] also provides a solution to proving type soundness using a big-step semantics without an explicit notion of wrong. Unlike us, Ancona studies a Java-like language. In order to prove its type soundness he gives a coinductive natural semantics without an explicit notion of going wrong. His approach appears to be related to the types as abstract interpretations approach, and makes the same observation that this chapter emphasises, namely that a sufficiently precise formalisation of the relationship between typings and concrete derivations enables proofs of type soundness without an explicit notion of going wrong. Although Ancona does not explicitly relate his approach to abstract interpretation, the crucial part of his proof appears to rely on a concretisation relation in the style of abstract interpretation.

It seems there is a close relationship between abstraction functions and logical relations [Tai67, Plo73, Pie02]. Ahmed [Ahm04] uses logical relations to give meaning to reference types and to derive typing rules for logical relations, in a very similar spirit

[4]Although Monsuez' natural semantics for call-by-value λ-calculus in [Mon95, Figure 1] does not have an explicit notion of going wrong, he adds explicit errors when defining his abstraction function [Mon95, Section 5.1].

to the types as abstract interpretations approach due to Cousot. In contrast to Cousot's approach and the approach taken here, Ahmed's proofs rely on dynamic semantics defined using a small-step transition relation.

Chapman [Cha09] also uses logical relations to prove strong normalisation of (dependently-typed) λ-calculus. His semantics is defined using typed syntax, whereby the semantics only captures well-typed terms. In contrast, the λ_{cbv} in this thesis is untyped, but realises several type models, including the monotyped and Hindley-Milner-Damas polytyped type systems in Sections 6.1 and 6.3 of this thesis.[5]

[5] One could say that Chapman's semantics uses Church-typing whereas our semantics uses Curry-typing (see, e.g., [Hin97] for a good introduction and discussion of this distinction).

7 Type Inference for References using Types as Abstract Interpretations

Contents

Types as abstract interpretations is useful for proving type soundness using big-step semantics by avoiding the tedium of adding explicit error rules. This addresses a concern with big-step type soundness proofs. Another concern with big-step type soundness is summarised by Felleisen and Wright [WF94] who, based on a survey of big-step type soundness proofs in the literature, conclude:

> A seemingly minor extension to a language may require a complete restructuring of its denotational or structural operational semantics, and may therefore require a completely new approach to re-establish soundness.

Their survey comprises several type soundness proofs from the literature concerning type inference for Hindley-Milner-Damas polymorphism with references. In this chapter we consider how types as abstract interpretations can be used to prove type soundness for such type systems.

This chapter provides two contributions over the previous chapter: firstly, we adapt the technique for type soundness from previous chapter to XSOS. Secondly, the language considered in this chapter does not satisfy strong normalisation. This poses some challenges for the technique: the abstraction function now needs to describe the types of both diverging and converging computations; and it is no longer obvious that the abstraction function is well-founded. We show how to adapt the technique to this setting and discuss these challenges. Our adaptation of the technique does not change the central arguments of the type soundness proof considerably.

Abstract syntax ($\Sigma^{\mathrm{HM}}_{\lambda_{\mathrm{cbv}}}$).

$$\begin{aligned}
\mathit{Type} \ni T &::= \mathit{nat} \mid T \rightharpoonup T \\
\mathit{Expr}^{\mathrm{NS}} \ni e &::= \texttt{plus}(e,e) \mid \texttt{num}(n) \mid \lambda x.e \mid e\ e \mid x \\
\mathit{Val}^{\mathrm{NS}} \ni v &::= n \mid \langle x, e, \rho \rangle \\
x \in \mathit{Var} &\triangleq \{\mathrm{x}, \mathrm{y}, \ldots\} \\
\Xi \in \mathit{PolyTypeEnv} &\triangleq \mathit{Var} \xrightarrow{\text{fin}} \wp(\mathit{Type})
\end{aligned}$$

Auxiliary entities ($\mathbb{C}^{\mathrm{HM}}_{\lambda_{\mathrm{cbv}}}$).

$$\mathbb{C}^{\mathrm{HM}}_{\lambda_{\mathrm{cbv}}} \triangleq \left\{ \mathbf{env} : \mathbb{C}^{\mathrm{DISCRETE}}(\mathit{PolyTypeEnv}) \right\}$$

Final configurations. (None required)
Rule specification.

$$\frac{e_1 : \mathit{nat} \qquad e_2 : \mathit{nat}}{\texttt{plus}(e_1, e_2) : \mathit{nat}} \quad \text{(XP-Plus)}$$

$$\frac{}{n : \mathit{nat}} \quad \text{(XP-Nat)}$$

$$\frac{T \in \Xi(x)}{\mathbf{env}\ \Xi \vdash x : T} \quad \text{(XP-Var)}$$

$$\frac{\mathbf{env}\ \Xi[x \mapsto \{T_2\}] \vdash e : T_1}{\mathbf{env}\ \Xi \vdash \lambda x.e : T_2 \rightharpoonup T_1} \quad \text{(XP-Fun)}$$

$$\frac{e_1 : T_2 \rightharpoonup T_1 \qquad e_2 : T_2}{e_1\ e_2 : T_1} \quad \text{(XP-App)}$$

Figure 7.1: Abbreviated XSOS typing rules for λ_{cbv}

7.1 Abstracting extensible specifications

We consider how to apply the types as abstract interpretations approach to a dynamic semantics of λ_{cbv}, and prove the type soundness of an extensible type system specification, specified using XSOS. For extensible type system specifications we restrict our attention to judgments of the form $G \vdash e : T$, such that G is an indexed product of read-only entities, and T is a type.

Figure 7.1 summarises an extensible specification of the syntax and typing rules for λ_{cbv} from Figure 6.4, with the exception of the specification of `let`, which is specified in Figure 7.2.[1]

[1]We specify `let` separately in anticipation of the future extension of our language with ML-style references which requires us to use different typing rules for `let`.

Abstract syntax ($\Sigma^{\mathtt{HM}}_{\mathtt{let}}$).

$$
\begin{aligned}
T &\in \mathit{Type} \\
\mathit{Expr}^{\mathrm{NS}} \ni e &::= \mathtt{let}\ x = e\ \mathtt{in}\ e \\
v &\in \mathit{Val}^{\mathrm{NS}} \\
x &\in \mathit{Var} \\
\Xi \in \mathit{PolyTypeEnv} &\triangleq \mathit{Var} \xrightarrow{\text{fin}} \wp(\mathit{Type})
\end{aligned}
$$

Auxiliary entities ($\mathbb{C}^{\mathtt{HM}}_{\mathtt{let}}$).

$$\mathbb{C}^{\mathrm{HM}}_{\mathtt{let}} \triangleq \left\{\mathbf{env} : \mathbb{C}^{\mathrm{DISCRETE}}(\mathit{PolyTypeEnv})\right\}$$

Final configurations. (None required)
Rule specification.

$$\frac{P \neq \emptyset \qquad (\forall T \in P.\ \mathbf{env}\ \Xi \vdash e_2 : T) \qquad \mathbf{env}\ \Xi[x \mapsto P] \vdash e_1 : T_1}{\mathbf{env}\ \Xi \vdash \mathtt{let}\ x = e_2\ \mathtt{in}\ e_1 : T_1} \quad \text{(XP-Let)}$$

Figure 7.2: Extensible typing rule for `let` with Hindley-Milner-Damas polymorphism

Given a set of extensible typing rules and a set of extensible pretty-big-step XSOS rules for λ_{cbv} (Figure 5.2 on page 126 with let-expressions given in Figure 7.3 on page 150), we investigate whether we can exploit their extensibility in big-step type soundness proofs using types as abstract interpretations. We start our investigation by considering how to specify the abstraction function for relating typings and denotations.

Denotations using XSOS. The last chapter gave abstraction functions for λ_{cbv} based on a notion of denotation that only contained the inductive interpretation of the evaluation relation. If we want an abstraction function that is extensible to type disciplines that may not ensure strong normalisation, we should include diverging computations in denotations.[2] Here, we consider the well-known extension of the Hindley-Milner-Damas typing system with references, which is known to be a typing discipline that does not satisfy the strong normalisation property.[3]

Figure 7.4 gives a denotation function for an XSOS relation using its inductive ($\Downarrow$) and coinductive interpretation ($\Downarrow^{\mathrm{co}}$). Its signature distinguishes two sets of objects:

[2]This is assuming that we are interested in extensions that produce typing disciplines that do not ensure strong normalisation. If we were to restrict our attention to typing disciplines that do ensure strong termination, the kinds of denotations and abstraction considered in the previous chapter would provide a useful guiding principle for investigating type systems that satisfy this restriction.

[3]For example, we can define recursive and diverging functions using “Landin’s knot” – a folklore term used to describe recursion using the store.

Abstract syntax ($\Sigma_{\texttt{let}}$).

$$\begin{aligned} \mathit{Expr} \ni e &::= \texttt{let}\ x = e\ \texttt{in}\ e \\ v &\in \mathit{Val}^{\mathrm{NS}} \\ x &\in \mathit{Var} \\ \sigma \in \mathit{Env} &\triangleq \mathit{Var} \xrightarrow{\text{fin}} \mathit{Val}^{\mathrm{NS}} \end{aligned}$$

Auxiliary entities ($\mathbb{C}_{\texttt{let}}$).

$$\mathbb{C}_{\texttt{let}} \triangleq \{(\mathbf{env}, \mathbb{C}^{\mathrm{DISCRETE}}(\mathit{Env}))\}$$

Terminal configurations. (None required)
Rule specification.

$$\frac{e_1 \Downarrow e_1' \qquad \texttt{let}\ x = e_1' \texttt{in}\ e_2 \Downarrow e'}{\texttt{let}\ x = e_1\ \texttt{in}\ e_2 \Downarrow e'} \quad \text{(XSOS-PB-Let1)}$$

$$\frac{\mathbf{env}\ \rho[x \mapsto v_1] \vdash e_2 \Downarrow e_2' \qquad \mathbf{env}\ \rho \vdash \texttt{let}\ x = v_1 \texttt{in}\ e_2' \Downarrow e'}{\mathbf{env}\ \rho \vdash \texttt{let}\ x = v_1\ \texttt{in}\ e_2 \Downarrow e'} \quad \text{(XSOS-PB-Let2)}$$

$$\frac{}{\texttt{let}\ x = v_1\ \texttt{in}\ v_2 \Downarrow v_2} \quad \text{(XSOS-PB-Let)}$$

Figure 7.3: Abbreviated pretty-big-step XSOS rules for `let`

$$\begin{aligned} &\mathbf{D}^{\mathrm{X}}[\![\bullet]\!] \in \mathit{Expr} \to O_R \to O_S \to \wp(\mathit{Val}_\perp \times O_S) \\ &\mathbf{D}^{\mathrm{X}}[\![e]\!] = \Lambda R.\ \Lambda S.\ \big\{(o, S') \bigm| R \vdash e_{/S} \Downarrow o_{/S'}\big\} \cup \\ &\qquad\qquad \big\{(\perp, S'[\mathbf{div} \uparrow]) \bigm| (\forall v\ S'.\ R \vdash e_{/S} \Downarrow^{\mathrm{co}} v_{/S'[\mathbf{div}\uparrow]})\big\} \end{aligned}$$

Figure 7.4: Extensible denotation function $\mathbf{D}^{\mathrm{X}}[\![\bullet]\!]$

O_R is the subset of objects in the underlying product category ranged over by R, and similarly O_S are the objects ranged over by S.

Abstraction function. In contrast to the semantics for λ_{cbv} that we considered in the previous chapter, the XSOS rules for λ_{cbv} thread stateful auxiliary entities ranged over by S through judgments in rules. They also propagate an indexed product of entities ranged over by R between the source and premises in rules.

The value produced by a function may depend on the auxiliary entities R and S, and hence the abstraction function for values must depend on these. In order to reflect this dependency, we could parameterise the abstraction function by these entities. Parameterising the abstraction function by the auxiliary entities would make the value abstraction function have a signature like:

$$\alpha_V^{\text{XHM}} \in O \to Val^{\text{NS}} \to Type$$

where O ranges over all objects in the underlying product category. But when entities in O vary, the outcomes resulting from executing functions may vary depending on the contents of O. This is problematic, since function types are typically inferred *once and for all*. This means that the type of the outcome of evaluating a function should remain invariant, or at least consistent, in any context where the function can be applied. Thus, if we want to infer the type of a function once and for all, we can do one of two things:

- either we can ensure that changes to O cannot affect the type of values; or
- we must somehow record in function types the assumptions about the context given by O, such that we can check statically that functions are always applied in configurations (and affects configurations in ways) that satisfy the same assumptions.[4]

The first of these possibilities is the simplest, whereas the latter is the more flexible, and reflects what is sometimes called *effect type systems* [TJ94, HMN04] and using typings as types [Jim96, Wel02, MSBP15]. We investigate the latter approach in the next chapter, and opt for the first approach here. This is in line with the traditional approach to typing references in ML-like languages [MTHM97, Tof90, Wri95].

Since R is propagated between source and premise in all rules, it must not affect the type of values: if it did, the same value would be allowed to have different types in different contexts. While for λ_{cbv} it holds that R only contains environments which do not affect the type of values (since environments in closures "shadow" the bindings in the dynamic environment) we also need to assume that any future extensions of R satisfies the same property. Assumption 7.1 records the assumption that only the environment is allowed to vary.

Assumption 7.1 The environment is the only entity in R that can affect the outcome of computations:

$$R[\textbf{env}\ \rho] \vdash e_{/S} \Downarrow^{\infty} e'_{/S'} \iff R'[\textbf{env}\ \rho] \vdash e_{/S} \Downarrow^{\infty} e'_{/S'}$$

Recall that, since the coinductively defined $\Downarrow^{\infty}$ subsumes the inductively defined $\Downarrow$ (by Lemma 5.3), Assumption 7.1 applies to both the inductive and coinductive interpretation of an evaluation relation.

[4]Indeed, the function type already records to the left of an arrow what type of values the function can be applied to, and typing rules check this.

Consider the following candidate for an abstraction function for closures:

$$\alpha_V^{\text{XHM0}} \in O_S \to Val_\perp^{\text{NS}} \to \wp(Type)$$

$$\alpha_V^{\text{XHM0}}(S, \langle x, e, \rho \rangle) \triangleq \left\{ T_1 \multimap T_2 \;\middle|\; \begin{array}{l} \forall R\ v_1.\ T_1 \in \alpha_V^{\text{HM}}(S, v_1) \implies \\ \quad \exists v_2\ S'.\ (v_2, S') \in \mathbf{D}[\![e]\!](R[\mathbf{env}\ \rho[x \mapsto v_1]])(S) \wedge \\ \qquad T_2 \in \alpha_V^{\text{XHM0}}(S', v_2) \end{array} \right\}$$

While this function assigns types to closures in a sound manner, it does not provide any means of ensuring that the closure has the same type if S varies. To capture this fact, we introduce some assumptions about states. In particular, we assume that they are preordered, and that types are preserved between preordered states. Assumptions 7.2 and 7.3 summarise these assumptions. When we add stateful entities later, we prove that these assumptions are satisfied.

Assumption 7.2 States are preordered by $\ll$:

$$R \vdash e_{/S} \Downarrow v_{/S'} \implies S \ll S'$$

Assumption 7.3 The preordering $\ll$ preserves value abstraction.

$$S \ll S' \implies \alpha_V(S, v) \subseteq \alpha_V(S', v)$$

Figure 7.5 summarises two abstraction functions:

- α_V^{XHM} is the value abstraction function, which uses the preordering of states to give meaning to closures in the current and *all subsequent* states, and types divergence with any type; and
- α_R^{XHM} is the read-only abstraction function for the set of entities modelled by discrete categories in the underlying indexed product category, i.e., those ranged over by R in rules.

The abstraction functions for denotations and sets of denotations are defined in a similar way as in Chapter 6, and Figure 7.9 omits the definition of these functions.

Type soundness. Following the same unfolding steps as illustrated in previous chapter gives the proof statement in Proposition 7.4.

Proposition 7.4 (Type soundness for λ_{cbv} using XSOS)

$$\begin{array}{l} G \vdash e : T \implies G \in \alpha_R^{\text{XHM}}(S, R) \implies \\ \exists v_\perp\ S'.\ (v_\perp, S') \in \mathbf{D}^{\text{X}}[\![e]\!](R)(S) \ \wedge\ T \in \alpha_V^{\text{XHM}}(S', v_\perp) \end{array}$$

$$
\begin{aligned}
\alpha_R^{\mathrm{XHM}} &\in O_S \to O_R \to \wp(O_G) \\
\alpha_R^{\mathrm{XHM}}(S,R) &\triangleq \left\{ G[\mathbf{env}\ \Xi] \;\middle|\; \begin{array}{l} \forall x \in \mathrm{dom}(R.\mathbf{env}). \\ \quad x \in \mathrm{dom}(\Xi) \wedge \Xi(x) \neq \emptyset \wedge \\ \qquad \forall T \in \Xi(x).\ T \in \alpha_V^{\mathrm{XHM}}(S, R.\mathbf{env}(x)) \end{array} \right\} \\
\alpha_V^{\mathrm{XHM}} &\in O_S \to Val_\perp^{\mathrm{NS}} \to \wp(Type) \\
\alpha_V^{\mathrm{XHM}}(S,\perp) &\triangleq Type \\
\alpha_V^{\mathrm{XHM}}(S,n) &\triangleq \{nat\} \\
\alpha_V^{\mathrm{XHM}}(S,\langle x,e,\rho\rangle) &\triangleq \left\{ T_1 \rightarrow T_2 \;\middle|\; \begin{array}{l} \forall R\ v_1.\ T_1 \in \alpha_V^{\mathrm{XHM}}(S,v_1) \implies S \ll S' \implies \\ \quad \exists v_2\ S''.\ (v_2,S'') \in \mathbf{D}[\![e]\!](R[\mathbf{env}\ \rho[x \mapsto v_1]])(S') \wedge \\ \qquad T_2 \in \alpha_V^{\mathrm{XHM}}(S'',v_2) \end{array} \right\}
\end{aligned}
$$

Figure 7.5: Abstraction function for λ_{cbv} with Hindley-Milner-Damas polymorphism

Proof (sketch). Each case requires subcase analysis everywhere divergence may arise. These subcases are completely mechanical and straightforward to prove. The remaining inductive subcases follow by the same line of reasoning as in Proposition 7.4, except that the cases for function abstraction and application now include extra reasoning steps utilising the preorder to ensure that abstractions are well-typed. See the Coq formalisation accompanying this thesis for the full details: `http://cs.swansea.ac.uk/~cscbp/xtss.zip`. □

7.2 Extensible type soundness for $\lambda_{\texttt{cbv+ref}}$

It is well-known that the naive extension of the Hindley-Milner-Damas typing discipline with ML-style references is unsound [Tof90, Dam84, WF94]. We consider how the extension of λ_{cbv} with ML-style references affects the proof of Proposition 7.4.

Semantics and denotations. The pretty-big-step XSOS rules for ML-style references are summarised in Figure 7.7. It is completely analogous to the SOS semantics that we recalled in Section 2.2.6.

Denotations are defined as follows: let $\lambda_{\texttt{cbv+ref}}$ be the semantics resulting from extending λ_{cbv} with the semantics for references given in Figure 7.7. The denotation function $\mathbf{D}^{\text{X-REF}}[\![e]\!]$ for the resulting semantics is defined as in Figure 7.4, but using the extended evaluation relations instead.

Notion of typing and abstraction. The syntax and typing rules for ML-style references, following [MTHM97, Tof90], is given in Figure 7.7.

Abstract syntax ($\Sigma_{\texttt{ref}}$).

$$\begin{aligned} \mathit{Expr} \ni e &::= \texttt{ref}(e) \mid \texttt{deref}(e) \mid \texttt{assign}(e,e) \mid v \\ \mathit{Val}^{\mathrm{NS}} \ni v &::= r \mid \texttt{unit} \\ r \in \mathit{Ref} &\triangleq \{\mathrm{r}_1, \mathrm{r}_2, \ldots\} \\ \sigma \in \mathit{Store} &\triangleq \mathit{Ref} \xrightarrow{\mathrm{fin}} \mathit{Val}^{\mathrm{NS}} \end{aligned}$$

Auxiliary entities ($\mathbb{C}_{\texttt{ref}}$).

$$\mathbb{C}_{\texttt{ref}} \triangleq \{(\mathbf{sto}, \mathbb{C}^{\mathrm{PREORDER}}(\mathit{Store}))\}$$

Terminal configurations. (None required)
Rule specification.

$$\frac{e \Downarrow e' \quad \texttt{ref}(e') \Downarrow e''}{\texttt{ref}(e) \Downarrow e''} \qquad \text{(XSOS-PB-Ref1)}$$

$$\frac{r \notin \mathrm{dom}(\sigma)}{\texttt{ref}(v)_{/\mathbf{sto}\ \sigma} \Downarrow r_{/\mathbf{sto}\ \sigma[r \mapsto v]}} \qquad \text{(XSOS-PB-Ref)}$$

$$\frac{e \Downarrow e' \quad \texttt{deref}(e') \Downarrow e''}{\texttt{deref}(e) \Downarrow e''} \qquad \text{(XSOS-PB-Deref1)}$$

$$\frac{r \in \mathrm{dom}(\sigma)}{\texttt{deref}(r)_{/\mathbf{sto}\ \sigma} \Downarrow \sigma(r)_{/\mathbf{sto}\ \sigma}} \qquad \text{(XSOS-PB-Deref)}$$

$$\frac{e_1 \Downarrow e_1' \quad \texttt{assign}(e_1', e_2) \Downarrow e'}{\texttt{assign}(e_1, e_2) \Downarrow e'} \qquad \text{(XSOS-PB-Assign1)}$$

$$\frac{e_2 \Downarrow e_2' \quad \texttt{assign}(r, e_2') \Downarrow e'}{\texttt{assign}(r, e_2) \Downarrow e'} \qquad \text{(XSOS-PB-Assign2)}$$

$$\frac{r \in \mathrm{dom}(\sigma)}{\texttt{assign}(r, v)_{/S[\mathbf{sto}\ \sigma]} \Downarrow \texttt{unit}_{/S[\mathbf{sto}\ \sigma[r \mapsto v]]}} \qquad \text{(XSOS-PB-Assign)}$$

Figure 7.6: Abbreviated pretty-big-step XSOS rules for references

Abstract syntax ($\Sigma^{\texttt{HM}}_{\texttt{ref}}$).

$$\mathit{Type} \ni T ::= T\ \mathit{ref} \mid \mathit{unit}$$

Auxiliary entities. (None required)
Terminal configurations. (None required)
Rule specification. (Syntax of expressions given by $\Sigma_{\lambda_{\text{cbv}}}$ in Figure 7.1)

$$\frac{e : T}{\texttt{ref}(e) : T\ \mathit{ref}} \quad \text{(XT-Ref)}$$

$$\frac{e : T\ \mathit{ref}}{\texttt{deref}(e) : T} \quad \text{(XT-Deref)}$$

$$\frac{e_1 : T\ \mathit{ref} \qquad e_2 : T}{\texttt{assign}(e_1, e_2) : \mathit{unit}} \quad \text{(XT-Deref)}$$

Figure 7.7: Abbreviated extensible typing rules for references

There is only a single kind new value introduced by the extension of λ_{cbv} to $\lambda_{\text{cbv+}\texttt{ref}}$, namely references r. We might consider the following candidate value abstraction function for giving meaning to reference types:

$$\alpha_V^{\text{XHM-R0}}(S, r) \triangleq \left\{ T\ \mathit{ref} \;\middle|\; r \in \text{dom}(S.\textbf{sto}) \ \wedge\ \exists v.\ S.\textbf{sto}(r) = v \ \wedge\ T \in \alpha_V^{\text{XHM-R0}}(S, v) \right\}$$

I.e., the type of a reference r is given in terms of the type of the value v contained in reference r in the current store $S.\textbf{sto}$. But this function fails to record a crucial fact about ML-style references: the type of references never change. If this is not enforced, then types are not preserved as stores vary during execution; for example, for two stores $\sigma_1 \triangleq \{\mathrm{r}_1 \mapsto 1\}$ and $\sigma_2 \triangleq \{\mathrm{r}_1 \mapsto \texttt{unit}\}$:

$$(\mathit{nat\ ref}) \in \alpha_V^{\text{XHM-R0}}(S[\textbf{sto}\ \sigma_1], \mathrm{r}_1) \ \text{ but } \ (\mathit{nat\ ref}) \notin \alpha_V^{\text{XHM-R0}}(S[\textbf{sto}\ \sigma_2], \mathrm{r}_1)$$

This is a potential hindrance for inferring the type of closures once-and-for-all.

The preordering $\ll$ was introduced exactly to the avail of typing closures in the current and all subsequent stores. Thus, we would like $\ll$ to record and enforce that the type of references in the store never change. But we cannot infer that the semantics in Figure 7.7 alone admits this notion of preorder: for example, the following program starts with a store σ_1 and terminates with σ_2:

$$\texttt{assign}(\mathrm{r}_1, \texttt{unit})_{/S[\textbf{sto}\ \sigma_1]}$$

I.e., using the denotation function as is, the type of references in the store may change. In order to give more precise abstraction functions for ML references, we refine the semantics of references such that it safely approximates the actual semantics in Figure 7.7, and reflects the desired property of the preorder: that evaluation preserves the type of references in the store.

Abstract syntax ($\Sigma^{\texttt{XHM-R}}_{\texttt{ref}}$). (See Figure 7.7)

$$\varsigma \in \mathit{TypeStore} \triangleq \mathit{Ref} \xrightarrow{\text{fin}} \mathit{Type}$$

Auxiliary entities ($\mathbb{C}^{\texttt{XHM-R}}_{\texttt{ref}}$).

$$\mathbb{C}^{\text{XHM-R}}_{\texttt{ref}} \triangleq \{(\textbf{sto}, \mathbb{C}^{\text{PREORDER}}(\mathit{Store})); (\textbf{tsto}, \mathbb{C}^{\text{PREORDER}}(\mathit{TypeStore}))\}$$

Terminal configurations. (None required)
Rule specification.

$$\frac{e \Downarrow e' \quad \texttt{ref}(e') \Downarrow e''}{\texttt{ref}(e) \Downarrow e''} \qquad \text{(XSOS-PB-Ref1)}$$

$$\frac{r \notin \text{dom}(\sigma) \quad T \in \alpha^{\text{XHM-R}}_{V}(S[\textbf{sto}\ \sigma, \textbf{tsto}\ \varsigma], v)}{\texttt{ref}(v)_{/S[\textbf{sto}\ \sigma, \textbf{tsto}\ \varsigma]} \Downarrow r_{/S[\textbf{sto}\ \sigma[r \mapsto v], \textbf{tsto}\ \varsigma[r \mapsto T]]}} \qquad \text{(XSOS-PB-Ref)}$$

$$\frac{e \Downarrow e' \quad \texttt{deref}(e') \Downarrow e''}{\texttt{deref}(e) \Downarrow e''} \qquad \text{(XSOS-PB-Deref1)}$$

$$\frac{r \in \text{dom}(\sigma)}{\texttt{deref}(r)_{/S[\textbf{sto}\ \sigma]} \Downarrow \sigma(r)_{/S[\textbf{sto}\ \sigma]}} \qquad \text{(XSOS-PB-Deref)}$$

$$\frac{e_1 \Downarrow e_1' \quad \texttt{assign}(e_1', e_2) \Downarrow e'}{\texttt{assign}(e_1, e_2) \Downarrow e'} \qquad \text{(XSOS-PB-Assign1)}$$

$$\frac{e_2 \Downarrow e_2' \quad \texttt{assign}(r, e_2') \Downarrow e'}{\texttt{assign}(r, e_2) \Downarrow e'} \qquad \text{(XSOS-PB-Assign2)}$$

$$\frac{r \in \text{dom}(\sigma) \quad r \in \text{dom}(\varsigma) \quad \varsigma(r) \in \alpha^{\text{XHM-R}}_{V}(S[\textbf{sto}\ \sigma, \textbf{tsto}\ \varsigma], v)}{\texttt{assign}(r, v)_{/S[\textbf{sto}\ \sigma, \textbf{tsto}\ \varsigma]} \Downarrow \texttt{unit}_{/S[\textbf{sto}\ \sigma[r \mapsto v], \textbf{tsto}\ \varsigma]}} \qquad \text{(XSOS-PB-Assign)}$$

Figure 7.8: Abbreviated pretty-big-step XSOS rules for references with refined semantics

Refined semantics for references. We refine the semantics for references by introducing a type store as a new auxiliary entity that records which types references are assigned to, and by introducing refined rules for references that utilise it. Figure 7.8 summarises the refined semantics. It relies on the abstraction function $\alpha^{\text{XHM-R}}_{V}$ in order to choose precise types for references, where $\alpha^{\text{XHM-R}}_{V}$ is defined in Figure 7.9. Here, the only change between the $\alpha^{\text{XHM}}_{\{R,V\}}$ and $\alpha^{\text{XHM-R}}_{\{R,V\}}$ abstraction functions is the naming and the addition of the case for references in $\alpha^{\text{XHM-R}}_{V}$.

$$
\begin{aligned}
\alpha_R^{\text{XHM-R}} &\in O_S \to O_R \to \wp(\mathit{TypeEnv}) \\
\alpha_R^{\text{XHM-R}}(S,R) &\triangleq \left\{ G[\mathbf{env}\ \Xi] \;\middle|\; \begin{array}{l} \forall x \in \mathrm{dom}(R.\mathbf{env}). \\ \quad x \in \mathrm{dom}(\Xi) \wedge \Xi(x) \neq \emptyset \wedge \\ \qquad \forall T \in \Xi(x).\ T \in \alpha_V^{\text{XHM-R}}(S, R.\mathbf{env}(x)) \end{array} \right\} \\
\alpha_V^{\text{XHM-R}} &\in O_S \to \mathit{Val}_\bot^{\text{NS}} \to \wp(\mathit{Type}) \\
\alpha_V^{\text{XHM-R}}(S,\bot) &\triangleq \mathit{Type} \\
\alpha_V^{\text{XHM-R}}(S,n) &\triangleq \{\mathit{nat}\} \\
\alpha_V^{\text{XHM-R}}(S,\langle x,e,\rho\rangle) &\triangleq \left\{ T_1 \rightarrowtriangle T_2 \;\middle|\; \begin{array}{l} \forall R\ v_1\ S'.\ S \ll S' \implies T_1 \in \alpha_V^{\text{XHM-R}}(S',v_1) \implies \\ \quad \exists v_2\ S''.\ (v_2,S'') \in \mathbf{D}[\![e]\!](R[\mathbf{env}\ \rho[x \mapsto v_1]])(S') \wedge \\ \qquad T_2 \in \alpha_V^{\text{XHM-R}}(S'',v_2) \end{array} \right\} \\
\alpha_V^{\text{XHM-R}}(S,r) &\triangleq \left\{ T\ \mathit{ref} \;\middle|\; \begin{array}{l} r \in \mathrm{dom}(S.\mathbf{sto}) \wedge r \in \mathrm{dom}(S.\mathbf{tsto}) \wedge \\ \quad S.\mathbf{tsto}(r) \in \alpha_V^{\text{XHM-R}}(S, S.\mathbf{sto}(r)) \end{array} \right\}
\end{aligned}
$$

Figure 7.9: Abstraction function for $\lambda_{\mathtt{cbv+ref'}}$

Mutual dependency. As the astute reader may have noticed, there is a mutual dependency between the abstraction function and the dynamic semantics. While a potential pitfall with relying on such a mutual dependency is paradoxical reasoning, we do not expect it to give rise to inconsistencies, since we expect it to be an instance of *induction-recursion* [Dyb00, DS06, GMNF13]. We discuss the mutual dependency further in Section 7.3.2.

Coq does not support induction-recursion, i.e., expressing functions and types such that they mutually depend on each other. And since the abstraction function does not correspond to a strictly-positive type, it is not straightforward to express as two mutually dependent relations either. In the Coq code accompanying this thesis, we work around the issue as follows:

1. We axiomatise the existence if a "type oracle" for checking that a given type is contained in the set of types that a given value can have. Thus, instead of $\alpha_V^{\text{XHM-R}}$, we use a function $\mathit{typeOracle} \in \mathit{Val} \to \wp(\mathit{Type})$, whose existence we axiomatise in Coq.

2. After specifying both the refined semantics (using *typeOracle* instead of $\alpha_V^{\text{XHM-R}}$) and the abstraction function (expressed as in Figure 7.9), we axiomatise that the two functions *typeOracle* and $\alpha_V^{\text{XHM-R}}$ are equivalent:

$$T \in \mathit{typeOracle}(v) \iff T \in \alpha_V^{\text{XHM-R}}(v)$$

Besides the potential pitfall with relying on the mutual dependency described here, there is also a potential philosophical issue with the refined semantics: big-step SOS typically embody an *operational* approach to dynamic semantics. But the refined semantics now depends on the abstraction function, which is undecidable. We stress that the purpose of the refined semantics is to make explicit the fact that the ML reference discipline preserves the type of references, not to act as an operational semantics for the language.

The refined semantics for $\lambda_{\text{cbv+}\texttt{ref}}$, which we call $\lambda_{\text{cbv+}\texttt{ref}'}$, is given by the union of the rule specification of the refined `ref` semantics in Figure 7.8 and the rule specification of λ_{cbv} from Figure 5.2 with `let` (Figure 7.3).

Preordering for states. The notion of preorder should ensure that the type store remains invariant and well-typed between transitions, where well-typedness is defined in Definition 7.5.

Definition 7.5 (Well-typedness of states) A state S is *well-typed* when all references in the store $S.\mathbf{sto}$ are typed by a corresponding reference in the type store $S.\mathbf{tsto}$.

$$wt(S) \iff \big(\mathrm{dom}(S.\mathbf{tsto}) = \mathrm{dom}(S.\mathbf{sto}) \wedge \forall r \in \mathrm{dom}(S.\mathbf{tsto}).\ S.\mathbf{tsto}(r) \in \alpha_V^{\text{XHM-R}}(S, S.\mathbf{sto}(r))\big)$$

Definition 7.6 defines a relation $\ll$ that records exactly what we need. Lemma 7.7 shows that $\ll$ is a preorder, i.e., that it is a reflexive and transitive relation.

Definition 7.6 (Preordering for states)

$$S_1 \ll S_2 \iff (wt(S_1) \implies wt(S_2) \wedge \forall r \in \mathrm{dom}(S_1).\ r \in \mathrm{dom}(S_2))$$

Lemma 7.7 ($\ll$ is reflexive and transitive)

$$(\forall S.\ S \ll S) \wedge \big(\forall S\ S'\ S''.\ S \ll S' \implies S' \ll S'' \implies S \ll S''\big)$$

Proof. Both reflexivity and transitive follow straightforwardly from the definition of $\ll$. □

Importantly, the notion of preorder in Definition 7.6 satisfies Assumptions 7.2 and 7.3, as shown by Lemmas 7.8 and 7.9.

Lemma 7.8 ($\lambda_{\text{cbv+}\texttt{ref}'}$ satisfies Assumption 7.2)

$$R \vdash e_{/S} \Downarrow e'_{/S'} \implies S \ll S'$$

Proof. The proof is by straightforward rule induction on the evaluation relation, using the definition of preorder in Definition 7.6. □

Lemma 7.9 ($\lambda_{\mathrm{cbv}+\mathtt{ref}'}$ satisfies Assumption 7.3)

$$S \ll S' \implies \alpha_V^{\text{XHM-R}}(S,v) \subseteq \alpha_V^{\text{XHM-R}}(S',v)$$

Proof. The property is equivalently expressed as:

$$S \ll S' \implies T \in \alpha_V^{\text{XHM-R}}(S,v) \implies T \in \alpha_V^{\text{XHM-R}}(S',v)$$

The proof is by structural induction on the structure of T, using the transitivity of '$\ll$' for the closure case. □

First attempt: type soundness. We can now attempt to prove type soundness using the definitions given in this section. Indeed, most of the cases will succeed, the only exception being `let` expressions. Recall from Proposition 6.4 that the `let` case of the proof relies on the determinism of the evaluation relation. The same argument would suffice to prove the property required for the `let` case with references. However, the refined semantics for references is *not* deterministic: when allocating a reference, the rule (XSOS-PB-Ref) makes a non-deterministic choice from the set of possible types for the value being assigned to the allocated reference.[5] Thus, the resulting evaluation relation is only deterministic up-to the structure of type stores.

This is the well-known problem with extending a Hindley-Milner-Damas type system with references, summarised by Tofte [Tof90, p. 11] as follows:

> The naive extension of the polymorphic type discipline fails because it admits generalisation on type variables that occur free in the store typing.

Our notion of polymorphism uses sets as polytypes rather than variables. Thus in our setting, the problem is summarised as: the naive extension of the polymorphic type discipline fails because it admits generalisation over stores that are inconsistently typed. The following program is a classic counter-example:

$$\begin{array}{l}\mathtt{let}\ r = \mathtt{ref}(\lambda x.x)\ \mathtt{in}\\ \mathtt{seq}(\mathtt{assign}(r, \mathtt{ref}(\lambda x.\mathtt{plus}(x, \mathtt{num}(1)))), (\mathtt{deref}(r)\ \mathtt{unit}))\end{array}$$

This program contains a type error (trying to add a natural number and `unit`), but it type-checks using the typing rules in Figures 7.1, 7.2, and 7.7. The problem is that the rule for `let` generalisation permits us to quantify over types that correspond to inconsistent type stores: using the rule (XP-Let), we can infer that r has the polytype $\{(nat \multimap nat)\ ref, (unit \multimap unit)\ ref\}$. These types admit both a type store where the reference bound to r has type $nat \multimap nat$ and a type store where it has type $unit \multimap unit$, even though the store after the assignment in the second line of the program is only consistent with the store where the allocated reference refers to a value of type $nat \multimap nat$.

[5] Technically, the rule also makes a non-deterministic choice of name for the new reference. We assume that there is a systematic way in which such names could be generated deterministically, however.

Abstract syntax ($\Sigma^{\mathtt{HM}}_{\mathtt{let}-VR}$).

$$
\begin{aligned}
sv \in SVal &::= \{\lambda x.e\} \cup \{\mathtt{num}(v)\} \\
v &\in Val^{\mathrm{NS}} \\
x &\in Var \\
\Xi \in PolyTypeEnv &\triangleq Var \xrightarrow{\text{fin}} \wp(Type)
\end{aligned}
$$

Auxiliary entities ($\mathbb{C}^{\mathtt{HM}}_{\mathtt{let}}$).

$$\mathbb{C}^{\mathtt{HM}}_{\mathtt{let}} \triangleq \{\mathbf{env} : \mathbb{C}^{\mathrm{DISCRETE}}(PolyTypeEnv)\}$$

Final configurations. (None required)
Rule specification.

$$\frac{P \neq \emptyset \qquad (\forall T \in P.\ \mathbf{env}\ \Xi \vdash sv_2 : T) \qquad \mathbf{env}\ \Xi[x \mapsto P] \vdash e_1 : T_1}{\mathbf{env}\ \Xi \vdash \mathtt{let}\ x = sv_2\ \mathtt{in}\ e_1 : T_1} \quad \text{(XP-Let-V)}$$

$$\frac{e_2 \notin SVal \qquad \mathbf{env}\ \Xi \vdash e_2 : T_2 \qquad \mathbf{env}\ \Xi[x \mapsto \{T_2\}] \vdash e_1 : T_1}{\mathbf{env}\ \Xi \vdash \mathtt{let}\ x = e_2\ \mathtt{in}\ e_1 : T_1} \quad \text{(XP-Let-E)}$$

Figure 7.10: Extensible typing rules for `let` with value restricted Hindley-Milner-Damas polymorphism

Type soundness using the value restriction. Several solutions have been devised in order to deal with polymorphic generalisation with references [Dam84, Tof90, LW91, TJ94, WF94, Wri95, Gar04]. Of these solutions, Wright's [Wri95] *value restriction* is the simplest. The idea is to only allow polymorphic generalisation for values, such that we guarantee that the stores over which types implicitly quantify are consistent with one another. Figure 7.10 summarises the rules for `let` with the value restriction, where we introduce the sort *SVal* for distinguishing the set of source terms that represent values; i.e., functions and numerals. With this restriction in place, it becomes possible to prove that the type system given by the union of the specifications in Figure 7.1 and 7.10 is sound.[6] The proof relies on Assumptions 7.2 and 7.3.

We need to make one small adjustment to the type soundness proof statement: we must assume that the store S in which evaluation begins is well-typed, i.e., $wt(S)$. Proposition 7.10 proves type soundness of $\lambda_{\text{cbv}+\mathtt{ref}'}$ with the value restriction, where $\mathbf{D}^{\text{X-REF}'}$ is the denotation function for $\lambda_{\text{cbv}+\mathtt{ref}'}$.

[6]This is the reason why we wanted separate specification of `let` and the remaining constructs for λ_{cbv}: so that we could replace the rule for `let` which interacts with the reference extension.

Proposition 7.10 (Type soundness of $\lambda_{\text{cbv+}\texttt{ref}'}$)

$$\begin{aligned}&G \vdash e : T \implies wt(S) \implies G \in \alpha_R^{\text{XHM-R}}(S,R) \implies \\ &\exists v_\perp\ S'.\ (v_\perp, S') \in \mathbf{D}^{\text{X-REF}'}[\![e]\!](R)(S) \ \wedge\ \alpha_V^{\text{XHM-R}}(S', v_\perp)\end{aligned}$$

Proof. The proof is by induction on the structure of the typing relation. The newly added cases for references follow straightforwardly, as does the non-polymorphic `let`-case. The polymorphic `let`-case for values follow by a similar line of reasoning as in Proposition 6.4, except there is no need for alluding to determinism, since all terms of sort *SVal* have a single unobservable transition that does not alter the store. We refer the reader to the Coq formalisation accompanying this thesis for the full proof: `http://cs.swansea.ac.uk/~cscbp/xtss.zip`. □

From refined to standard semantics of references. Our proof of soundness in Proposition 7.10 relies on a refined semantics for references; but our original goal was to prove that the original rules we gave for references in Figure 7.7 were type sound. It is straightforward to show that the semantics of $\lambda_{\text{cbv+}\texttt{ref}'}$ is a sound approximation of $\lambda_{\text{cbv+}\texttt{ref}}$ (Proposition 7.11). As a corollary of this relationship, it follows that the type system is also sound for $\lambda_{\text{cbv+}\texttt{ref}}$.

Proposition 7.11 ($\lambda_{\text{cbv+}\texttt{ref}'}$ is sound w.r.t. $\lambda_{\text{cbv+}\texttt{ref}}$) Let $\Downarrow^{\text{co}}$ denote the coinductive interpretation of the evaluation relation for $\lambda_{\text{cbv+}\texttt{ref}}$, and $\Downarrow^{\text{co}\prime}$ denote the coinductive interpretation of the evaluation relation for $\lambda_{\text{cbv+}\texttt{ref}'}$; then:

$$R \vdash e_{/S[\mathbf{tsto}\ \varsigma]} \Downarrow^{\text{co}\prime} e'_{/S'[\mathbf{tsto}\ \varsigma']} \implies \forall \varsigma''.\ R \vdash e_{/S[\mathbf{tsto}\ \varsigma'']} \Downarrow^{\text{co}} e'_{/S'[\mathbf{tsto}\ \varsigma'']}$$

Proof. The proof is by straightforward coinduction on $\Downarrow^{\text{co}}$. □

Corollary 7.12 (The denotations of $\lambda_{\text{cbv+}\texttt{ref}'}$ is sound w.r.t. $\lambda_{\text{cbv+}\texttt{ref}}$) Let $\mathbf{D}^{\text{X-REF}}[\![\bullet]\!]$ be the denotation function for $\lambda_{\text{cbv+}\texttt{ref}}$, and $\mathbf{D}^{\text{X-REF}'}[\![\bullet]\!]$ be the denotation function for $\lambda_{\text{cbv+}\texttt{ref}'}$; then:

$$(v_\perp, S'[\mathbf{tsto}\ \varsigma]) \in \mathbf{D}^{\text{X-REF}'}[\![e]\!](R)(S) \implies (v_\perp, S'[\mathbf{tsto}\ \varsigma]) \in \mathbf{D}^{\text{X-REF}}[\![e]\!](R)(S[\mathbf{tsto}\ \varsigma])$$

Proof. The property follows from unfolding the definitions of $\mathbf{D}^{\text{X-REF}}[\![\bullet]\!]$ and $\mathbf{D}^{\text{X-REF}'}[\![\bullet]\!]$, Proposition 7.11, and the fact that coevaluation implies either inductive convergence or divergence (Lemma 5.5). □

Corollary 7.13 ($\lambda_{\text{cbv+}\texttt{ref}}$ is type sound) Let $\mathbf{D}^{\text{X-REF}}[\![\bullet]\!]$ be the denotation function for $\lambda_{\text{cbv+}\texttt{ref}}$, and $\mathbf{D}^{\text{X-REF}'}[\![\bullet]\!]$ be the denotation function for $\lambda_{\text{cbv+}\texttt{ref}'}$; then:

$$\begin{aligned}&G \vdash e : T \implies wt(S) \implies G \in \alpha_R^{\text{XHM-R}}(S,R) \implies \\ &\exists v_\perp\ S'[\mathbf{tsto}\ \varsigma].\ (v_\perp, S'[\mathbf{tsto}\ \varsigma]) \in \mathbf{D}^{\text{X-REF}}[\![e]\!](R)(S[\mathbf{tsto}\ \varsigma]) \ \wedge\ \alpha_V^{\text{XHM-R}}(S'[\mathbf{tsto}\ \varsigma], v_\perp)\end{aligned}$$

Proof. By invoking Proposition 7.10 on the premises, we get:

$$\exists v_\perp\ S'[\mathbf{tsto}\ \varsigma].\ (v_\perp, S'[\mathbf{tsto}\ \varsigma]) \in \mathbf{D}^{\text{X-REF}'}[\![e]\!](R)(S) \ \wedge\ \alpha_V^{\text{XHM-R}}(S'[\mathbf{tsto}\ \varsigma], v_\perp)$$

The goal follows by existential elimination and Corollary 7.12. □

7.3 Assessment and related work

This chapter investigated two research questions: whether types as abstract interpretations scales as a proof method; and whether extensible transition system semantics is useful for making proofs less prone to restructuring as languages are extended. We investigated these questions by making the following contributions:

- we have given a types as abstract interpretations account of ML-style references; and
- we have adapted the types as abstract interpretations approach to XSOS.

We assess these contributions against the stated research questions.

7.3.1 Proof method

The extension with references required us to update our notion of typing to include stores, and carefully assess which information to record in abstraction functions. The careful assessment was carried out by attempting to carry out the proof and observing where stronger requirements are required. In other words, proof acts as a guiding principle for inferring where we need to strengthen requirements in either the abstraction functions or in the typing rules that approximates the most precise abstraction. Indeed, Cousot stresses that abstract interpretation is useful as a design methodology for constructing type systems that are safe by construction [Cou97, p. 321]:

> It is interesting to note that instead of "formalizing the type system by a set of type rules, and verifying that program execution of well-typed programs cannot produce type errors" [Car96], the abstract interpretation design methodology ensures that type systems will be sound by construction, this soundness requirement being used as a guideline for designing the type system.

Using inductive proofs as a means of calculation is an idea that has recently been explored by Bahr and Hutton to calculate correct compilers [BH15]. By so-called *constructive induction* [Bac03, Chapter 12.4] they use a denotational semantics and the correctness criterion to use only the proof of correctness and the denotational semantics as a guiding principle for inferring both the instruction set and execution function for a virtual machine, as well as the compilation function for translating a program into virtual machine functions.

There is an analogy between our use of types as abstract interpretations and Bahr and Hutton's work: using types as abstract interpretations, we are given a correctness criterion (the type soundness statement which constitutes the Galois connection) and a dynamic semantics, and we want to infer the structure of what constitutes types, their meaning (abstraction/concretisation functions), and a safe abstraction of this meaning. The many unknowns involved in this process makes it all the more practical to use the proof as a guiding principle.

In this chapter we started from a well-known type system and used the proof as a guiding principle for inferring the structure of a meaning function for types. The steps involved in this calculation were equal parts involved and illuminating: involved, since they force us to formally express the intuitive meaning of types, which is not always straightforward; the mutually dependent abstraction function and evaluation relation is a witness to this (although, as we remark below, there are ways in which we might have avoided this). The calculation is illuminating for the same reason that it is involved: it forces us to be conscious about the meaning of types. We believe that types as abstract interpretations could provide a useful basis for inferring new and novel type systems in a principled manner. Indeed, using abstract interpretation as the basis of calculating and discovering new analysis frameworks for programming languages seems an approach that is gaining traction [VHM10, Mig10, SDM$^+$13].

A natural question to ask ourselves: is giving and working with abstraction/concretisation functions really necessary in order to take a calculational approach to type systems? Could we not have used the traditional approach to proving big-step type soundness, or even a small-step argument? It seems plausible that a small-step progress/preservation style proof might provide a good basis for inferring the structure of typing rules, too. The traditional approach to big-step type soundness based on "wrong" rules appears somewhat more problematic, in that failure to add a "wrong" rule might compromise the calculation, allowing us to infer that a set of typing rules are sound, even if they admit programs that go wrong.

7.3.2 Pitfall: well-foundedness

A pitfall with using types as abstract interpretations is that it can sometimes be challenging to express the precise notion of type in a manner that is obviously well-founded.

In this chapter, the pitfall is even more evident, in particular in the refined semantics for references, where we gave an abstraction function and evaluation relation that are mutually dependent on each other. We do not believe that this mutual dependency is problematic from the perspective of type soundness and Proposition 7.10, for two reasons (where we ignore the fact that it is well-known that the type system we considered is sound for ML-style references [Wri95, WF94] and focus on the proof method):

- I have not been able to find examples of programs that make the refined semantics assign inconsistent types to references or outcomes. For example, one might expect cyclic references to be problematic. This does not appear to be a problem with the refined abstraction function. For example, the following cyclic store from Pierce [Pie02, p. 163] is typable using our abstraction functions and notion of well-typedness:

$$\begin{aligned}\sigma \triangleq \{\ & r_1 \mapsto \langle x, (\mathtt{deref}(s)\ x), \{s \mapsto r_2\}\rangle, \\ & r_2 \mapsto \langle x, (\mathtt{deref}(s)\ x), \{x \mapsto r_1\}\rangle\ \}\end{aligned}$$

  Here, the closure contained in r_1 refers to the reference r_2, and vice versa. Given the abstraction function for the refined semantics, this store is typed by the following type store:

$$\begin{aligned}\varsigma \triangleq \{\ & r_1 \mapsto (nat \rightharpoonup nat)\ ref, \\ & r_2 \mapsto (nat \rightharpoonup nat)\ ref\ \}\end{aligned}$$

Each of the closures in σ diverge when applied to a natural number. Since $nat \in \alpha_V^{\text{XHM-R}}(S,\bot)$ it holds that the configuration with σ and ς is well-typed, i.e., $wt(S[\textbf{sto}\ \sigma, \textbf{tsto}\ \varsigma])$:

$$\begin{aligned}&\mathrm{dom}(\sigma) = \mathrm{dom}(\varsigma)\ \land \\ &\quad nat \rightharpoonup nat \in \alpha_V^{\text{XHM-R}}(S[\textbf{sto}\ \sigma, \textbf{tsto}\ \varsigma], \sigma(r_1))\ \land \\ &\quad nat \rightharpoonup nat \in \alpha_V^{\text{XHM-R}}(S[\textbf{sto}\ \sigma, \textbf{tsto}\ \varsigma], \sigma(r_2))\end{aligned}$$

- Recent work on the datatypes à la carte approach to extensible datatypes [Swi08] in Coq shows how to deal with mutually-recursive definitions by using Mendler style recursion [DdSOS13, DKSO13]. Preliminary experiments conducted together with Paolo Torrini of encoding abstraction functions in a similar style suggests that this means of encoding could be useful for encoding the kind of mutual dependence found in the refined semantics.

The mutual dependency in the refined semantics used in our proof could also be avoided by opting for a proof closer to the style of Tofte's proof [Tof90], by embodying the preordering of stores in the proof statement itself. Recall the the abstraction function for denotations used for the refined semantics is defined as follows:

$$\alpha_D^{\text{XHM-R}} \in \mathbb{D} \to \mathbb{T}$$

$$\alpha_D^{\text{XHM-R}}(d) \triangleq \left\{ (G,T)\ \middle|\ \begin{array}{l}\forall S\ R.\ G \in \alpha_R^{\text{XHM-R}}(S,R) \implies \\ \exists v\ S'.\ (v,S') \in d(R,S)\ \land\ T \in \alpha_V^{\text{XHM-R}}(S,v)\end{array} \right\}$$

Following Tofte, we could alternatively define it as:

$$\alpha_D^{\text{XHM-R}} \in \mathbb{D} \to \mathbb{T}$$

$$\alpha_D^{\text{XHM-R}}(d) \triangleq \left\{ (G,T)\ \middle|\ \begin{array}{l}\forall SR.\ G \in \alpha_R^{\text{XHM-R}}(S,R) \implies \\ \qquad \exists v\ S'.\ (v,S') \in d(R,S)\ \land\ T \in \alpha_V^{\text{XHM-R}}(S,v)\ \land\ S \ll S'\end{array} \right\}$$

This avoids the need for instrumenting the semantics, but changes the structure of the induction proof that we gave in Proposition 7.4 for λ_{cbv} based on pretty-big-step XSOS. If this is truly necessary, it would provide evidence that the answer is no to the research question: "Do extensible transition system semantics facilitate alleviating the drawback of big-step type soundness that extensions may require a complete restructuring of semantics and proof?" Here, we have outlined a proof and an approach that does not require such restructuring. We leave it to future work to investigate the well-foundedness of the refined semantics on which the proof is based.

Another argument for the morality of our approach is that our relation in fact appears to be an instance of an *inductive-recursive* definition [Dyb00, DS06, GMNF13], i.e., an inductive datatype (in our case, an evaluation relation) and a recursive function (in our case, an abstraction function) that is mutually defined. Our relation might also correspond to an instance of *induction-induction* [NF13]. We leave it to future work to verify that the criteria for inductive-recursive definitions [Dyb00, DS06, GMNFS13, GMNF15] or inductive-inductive definitions [NF13] are satisfied. Similarly for coinductive-recursive definitions, since the evaluation relation is *dually* defined. Capretta [Cap13] has investigated such definitions.

7.3.3 Extensible proofs

The extension of our semantics with references did not require us to change the cases for existing constructs. The extensibility stems from the fact that, using XSOS, even side-effect free languages propagate stateful configurations. While this makes side-effect free semantics somewhat less straightforward to reason about, it allows us to reason about configurations with an open-ended set of objects in it, and makes it feasible to extend the semantics without reformulating the semantics, and, as this chapter suggests, facilitates reuse of proofs.

Our approach introduced a notion of preorder that records an invariant that must hold between transitions. We might have used the evaluation relation everywhere we use the preorder instead. The preorder records just the necessary invariant, whereas we would have to explicitly derive the invariant from the evaluation relation in proofs, had we used that instead. It seems like this might also aggravate the potential problem with the mutual dependency between the abstraction function and evaluation relation even further.

7.3.4 Representation of divergence

In order to represent and distinguish divergence, we gave denotation functions which contain the union of the set of all finite derivations and the set of infinite derivations. As remarked, the distinction between these two sets gives rise to extra case analysis in proofs, though most of this case analysis is completely mechanical. The literature offers several suggestions for how to best represent divergence in big-step semantics, many of which we recalled in Chapter 5. It would be interesting to see how to adapt such representations to the types as abstract interpretations proof approach. Particularly interesting approaches to consider adapting would be trace-based semantics [NU09] and Danielsson's functional operational semantics using the partiality monad [Dan12], as well as type soundness using clock-based semantics [Sie13].

7.3.5 Related work

In this chapter we investigated how to use types as abstract interpretations for proving type soundness of a language with functions and ML-style references. This problem has been considered by several other authors. Tofte [Tof90], whose work this chapter draws some inspiration from, gave a type soundness proofs using a big-step semantics with an explicit notion of going wrong. Harper and Stone [HS00b] gives a "type-theoretic interpretation" of Standard ML by translating it into a typed intermediate language based on a small-step semantics. They conclude [HS00b, Section 4]:

> The internal language admits a clean formulation of the soundness theorem that does not rely on instrumentation of the rules with explicit "wrong" transitions. To state soundness in the framework of The Definition requires that the dynamic semantics be instrumented with such error transitions, which would significantly increase the number of rules required.

Harper and Stone state that they draw inspiration from Felleisen and Wright's [WF94] syntactic approach to type soundness, which also takes a small-step approach to proving type soundness of a language very similar to the one considered here. Indeed, their approach studies specifically how to give type soundness proofs for Hindley-Milner-Damas style polymorphic type systems. Here, we have used types as abstract interpretations to give a type soundness proof of a polymorphic type system using a big-step style reminiscent of the one in Standard ML but without explicit "wrong" transitions.

We also considered the extent to which our use of XSOS facilitated reuse of cases when proving type soundness. The reuse that we obtained was of a syntactic nature (we could call it "copy-paste reuse"), in that we were able to reuse the text of the proof for λ_{cbv} without `let`, rather than the proof itself. One of the motivations behind Felleisen and Wright's syntactic approach is also being able to easily extend type soundness proofs as languages evolve, and the extent to which they reuse their proofs is also of a syntactic nature. There are several authors who have investigated a more semantic approach to reuse, in the sense of being able to combine proofs in proof assistants. Examples include Delaware et al.'s work on Meta-Theory à la Carte [DdSOS13, DKSO13], Schwaab and Siek's modular type-safety proofs in Agda [SS13], and Madlener's formal component-based semantics which shows how to obtain semantic reuse of proofs based on MSOS [MSvE11], and Torrini and Schrijver's modular datatypes with Mendler induction [TS15].

Part of the flexibility of our approach was also afforded by our use of pretty-big-step semantics, which makes it possible to deal with abrupt termination, divergence, and even control constructs without any reformulation. Bodin et al. [BJS15] illustrates that the extensible nature of the pretty-big-step style also constitutes a useful basis for instrumenting semantics and deriving program analyses using abstract interpretation. The type soundness proof for references also relied on instrumentation, but of a somewhat different nature to the kind of instrumentation of Bodin et al.

8 Type and Effect Inference using Types as Abstract Interpretation

Contents

The naive extension with references of Hindley-Milner-Damas polymorphism fails because it admits generalisation over stores that are inconsistently typed. The value restriction solves the problem by inhibiting let-polymorphic generalisation, but also unfairly rejects various programs that cannot go wrong.

In this chapter we propose an alternative store-based type system that tracks the types of references more closely by recording the type of references using type stores. This enables unrestricted let-polymorphic generalisation, and permits type checking programs that would otherwise be rejected using the value restriction. The flexibility comes at the cost of more rigid typing of references which unfairly rejects some programs that type check using ML-style reference types. Store-based typing is closely related to *alias typing* [SWM00] and so-called *type and effect* systems [HMN04, Wri92, TJ94]. In addition to checking what kinds of values a program produces, type and effect systems also track other effects, such as allocating, reading, and updating references. We compare and contrast our approach with ML-style references types (in Sections 8.1 and 8.2) as well as alternative approaches from the literature (Section 8.4), and outline a type soundness argument for the type system (Section 8.3) by adapting the proof method from the previous chapter.

The type system this chapter proposes is a variant of the type system in [BPMT15] which was joint work with Mosses and Torrini. Whereas the type system in [BPMT15] supports so-called *strong updates* (i.e., it types programs where the types of references may change during evaluation), the type system here is based on invariant updates (i.e., the types of references do not change during evaluation).

8.1 The problem with let-polymorphic generalisation with references

Before delving into store-based types, we recall the problem with let-polymorphic generalisation with references. Section 7.2 presented the following example of a faulty program which type checks using unrestricted let-polymorphic generalisation with references:

$$\begin{array}{l}\texttt{let}\ \mathrm{r} = \texttt{ref}(\lambda \mathrm{x}.\mathrm{x})\ \texttt{in} \\ \texttt{seq}(\texttt{assign}(\mathrm{r}, \texttt{ref}(\lambda \mathrm{x}.\texttt{plus}(\mathrm{x}, \texttt{num}(1)))), \texttt{deref}(\mathrm{r})\ \texttt{unit})\end{array}$$

With unrestricted let-polymorphism, we can infer that the variable ‘r’ in the program above has the the following polytype:

$$\{(T \rightarrowtriangle T)\ \texttt{ref} \mid T \in \mathit{Type}\}$$

This type describes the set of all references that are typed by the identity function type $T \rightarrowtriangle T$. But references may vary during execution, whereby this type describes stores that are inconsistently typed. For example, in the program above, the store after making the assignment in the second line of the program is only consistent with a store where the reference bound by ‘r’ has type $\mathit{nat} \rightarrowtriangle \mathit{nat}$.

8.1.1 The value restriction

The value restriction solves the problem by inhibiting any generalisation over stores: only values (functions and numbers) are generalisable, whereby we only generalise expressions that do not alter the store. Thus, the polytypes produced by polymorphic generalisation trivially correspond to stores that are consistent. For example, the value restriction rejects the faulty program above.

But the value restriction also restricts polymorphic generalisation of many programs that are *not* inconsistently typed. Consider the following well-typed program which is accepted with the value restriction:

$$\begin{array}{l}\texttt{let}\ \mathrm{id} = (\lambda \mathrm{x}.\mathrm{x})\ \texttt{in} \\ \texttt{seq}(\mathrm{id}\ \texttt{num}(1), \mathrm{id}\ \texttt{unit})\end{array}$$

The following equivalent program, however, is not accepted by the value restriction, since ‘y’ is not a value:

$$\begin{array}{l}\texttt{let}\ \mathrm{id} = (\lambda \mathrm{x}.\mathrm{x})\ (\lambda \mathrm{y}.\mathrm{y})\ \texttt{in} \\ \texttt{seq}(\mathrm{id}\ \texttt{num}(1), \mathrm{id}\ \texttt{unit})\end{array}$$

But there are no stores involved, and polymorphic generalisation is clearly safe.

8.1.2 η-expansion

The example above is somewhat contrived. Wright’s [Wri95] study of realistic Standard ML programs concludes:

> We found that most ML programs either satisfy the restriction of polymorphism to values already, or they can be modified to do so with a few η-expansions.

The η-expansion technique that Wright refers to consists in η-expanding a term to make it into a value which can be generalised. The following program illustrates how such η-expansion makes it possible to type check the contrived example program above that was rejected using the value restriction:

$$\begin{array}{l}\texttt{let}\ \mathrm{id} = (\lambda \mathrm{z}.(\lambda \mathrm{x}.\mathrm{x})\ (\lambda \mathrm{y}.\mathrm{y})\ \mathrm{z})\ \texttt{in} \\ \texttt{seq}(\mathrm{id}\ \texttt{num}(1), \mathrm{id}\ \texttt{unit})\end{array}$$

For this program, such η-expansion is unproblematic. For more interesting programs, η-expansion can be problematic, however.

Consider the following example where we would like to generalise over the result of applying some computationally expensive but pure function (i.e., a function that does not alter the store), `expensive_function` to `unit`:

$$\begin{array}{l}\texttt{let}\ \mathrm{f} = (\texttt{expensive\_function unit})\ \texttt{in} \\ \texttt{seq}(\mathrm{f}\ \texttt{num}(1), \mathrm{f}\ \texttt{num}(2))\end{array}$$

We can modify and η-expand this program as follows:

$$\begin{array}{l}\texttt{let}\ \mathrm{f} = (\lambda \mathrm{u}.(\lambda x.\texttt{expensive\_function unit})u)\ \texttt{in} \\ \texttt{seq}(\mathrm{f}\ \texttt{num}(1), \mathrm{f}\ \texttt{num}(2))\end{array}$$

If the application `expensive_function unit` is computationally expensive, the expanded program is more expensive than the original program: the call-by-value semantics of application means that the application `expensive_function unit` is computed twice (once for each call of f); in contrast, the original program evaluated `expensive_function unit` once in the binding expression for f.

8.2 A store-based type system

We propose a novel type system that permits let-polymorphic generalisation without the value restriction. The idea is that the type system records store typings by means of type stores in the type system. Threading such stores through the type system provides valuable information for polymorphic generalisation.

Figure 8.2 defines the abstract syntax for types and typings, as well as typing rules for store-based types. We consider and explain the novel additions as compared with ML-style references:

Aliases. We use *aliases* to represent a set of runtime references. Our use of the word *alias* differs from [SWM00]: they use it mainly to describe pointers from references to references. In contrast, we use the word alias in a broader sense to describe the type-level counter-part to a run-time reference. *Alias type stores* map aliases to types, where

Abstract syntax.

$$a \in \mathit{Alias} \triangleq \{\mathrm{a}_1, \mathrm{a}_2, \ldots\}$$

$$\mathit{Type} \ni T ::= T \rightarrowtriangle^{\Sigma} T \mid \mathit{nat} \mid \mathit{aref}(T, a)$$

$$\Sigma \in \mathit{AliasTypeStore} \triangleq \mathit{Alias} \xrightarrow{\text{fin}} \mathit{Type}$$

$$\Xi \in \mathit{PolyTypeEnv} \triangleq \mathit{Var} \xrightarrow{\text{fin}} \mathit{Type}$$

Auxiliary entities.

$$\mathbb{C}^{\text{TYPE}}_{\texttt{ref}} \triangleq \{(\mathbf{sto}, \mathbb{C}^{\text{PREORDER}}(\mathit{TypeStore}))\}$$

Final configurations. (None required)
Rule specification.

$$\frac{e_1 : \mathit{nat} \qquad e_2 : \mathit{nat}}{\texttt{plus}(e_1, e_2) : \mathit{nat}} \tag{XP-Plus}$$

$$\frac{}{n : \mathit{nat}} \tag{XP-Nat}$$

$$\frac{T \in \Xi(x)}{\mathbf{env}\ \Xi \vdash x : T} \tag{XP-Var}$$

$$\frac{\mathbf{env}\ \Xi[x \mapsto \{T_2\}] \vdash e_{/\mathbf{sto}\,\Sigma} : T_{1/\mathbf{sto}\,\Sigma'}}{\mathbf{env}\ \Xi \vdash (\lambda x.e)_{/\mathbf{sto}\,\Sigma} : (T_2 \rightarrowtriangle^{\Sigma - \Sigma'} T_1)_{/\mathbf{sto}\,\Sigma}} \tag{XPZ-Fun}$$

$$\frac{e_{1/\mathbf{sto}\,\Sigma} : (T_2 \rightarrowtriangle^{\Sigma_0} T_1)_{/\mathbf{sto}\,\Sigma'} \qquad e_{2/\mathbf{sto}\,\Sigma'} : T_{2/\mathbf{sto}\,\Sigma''}}{e_1\ e_{2/\mathbf{sto}\,\Sigma} : T_{1/\Sigma''[\Sigma_0]}} \tag{XPZ-App}$$

$$\frac{P \neq \emptyset \qquad (\forall T_1 \in P.\ \mathbf{env}\ \Xi \vdash e_1 : T_1) \qquad \mathbf{env}\ \Xi[x \mapsto P] \vdash e_2 : T_2}{\mathbf{env}\ \Xi \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : T_2} \tag{XPZ-Let}$$

$$\frac{\begin{array}{c} a \notin \mathrm{dom}(\Sigma') \\ e_{/\mathbf{sto}\,\Sigma} : T_{/\mathbf{sto}\,\Sigma'} \end{array}}{\texttt{ref}(e)_{/\mathbf{sto}\,\Sigma} : \mathit{aref}(T, a)_{/\mathbf{sto}\,\Sigma'[a \mapsto T]}} \tag{XPZ-Ref}$$

$$\frac{e : \mathit{aref}(T, a)}{\texttt{deref}(e) : T} \tag{XPZ-Deref}$$

$$\frac{e_1 : \mathit{aref}(T, a) \qquad e_2 : T}{\texttt{assign}(e_1, e_2) : \mathit{unit}} \tag{XPZ-Asgn}$$

$$\Sigma - \Sigma' \triangleq \{(a \mapsto T) \in \Sigma \mid a \notin \mathrm{dom}(\Sigma')\} \tag{TA-Diff}$$

$$\begin{array}{l} \Sigma[\Sigma'] \triangleq \Sigma \cup \theta(\Sigma') \text{ such that } \mathrm{dom}(\theta) \triangleq \mathrm{dom}(\Sigma') \\ \qquad\qquad\qquad\qquad \text{and } \mathit{dom}(\Sigma) \cap \mathrm{dom}(\theta(\Sigma')) = \emptyset \end{array} \tag{TA-Extend}$$

the notion of reference type differs from ML-style reference types by recording an alias as well as a type. An *alias reference type* $\mathit{aref}(T,a)$ represents a reference containing a value of type T, aliased by a in the type store. It is not strictly necessary to record the T in $\mathit{aref}(T,a)$, since alias type stores map aliases to types; we include it to make it easier to read types independently of type stores.

Function types. Applying a function may result in allocation of new references. This effect is reflected in function types by annotating the arrow of the function type with an alias type store recording the set of references that will be allocated when the function is applied. The rule (XPZ-Fun) reflects this by typing the body of the function using the current alias type store Σ to infer its type and updated alias type store Σ'. The rule does not propagate the updated map Σ' to the conclusion. Instead, the difference between Σ and Σ' is recorded in an annotation on the function arrow, where the difference is defined by equation (TA-Diff) in Figure 8.2. Modelling maps as sets of pairs of an alias a and a type T, $(a \mapsto T)$, the equation specifies that the difference between alias type stores Σ and Σ' is the set of all pairs $(a \mapsto T) \in \Sigma$ such that a is not in the domain of the map Σ'.

The application rule (XPZ-App) ensures that applying a function allocates aliases recorded in the function type, by adding the aliases Σ_0 to the alias type store Σ''. The rule ensures that aliases being added to the store are fresh: the operation $\Sigma[\Sigma']$ produces a store Σ'' containing all the mappings in Σ, and applies a substitution θ to Σ' (written $\theta(\Sigma')$) which renames all aliases such that the domain of Σ and $\theta(\Sigma')$ become disjoint. To prevent renaming of aliases other than those in the domain of Σ', we require that the domain of the substitution (i.e., the set of aliases that are renamed) is exactly the domain of Σ'.

Typing of references. The rules for references, (XPZ-Ref), (XPZ-Deref), and (XPZ-Asgn), resemble those for ML-reference typing. The main difference is in the (XPZ-Ref) rule which adds a fresh alias to the alias type store to reflect that a reference of the given type is being allocated. Since alias reference types $\mathit{aref}(T,a)$ record the type of the reference it represents there is no need for the rules (XPZ-Deref) and (XPZ-Asgn) to consult the alias type store, except to update it in connection with assignment. Like ML-reference types, store-based types are invariant.

Polymorphism. The rule (XPZ-Let) admits unrestricted generalisation over the types, but not over alias type stores. This ensures that stores are consistently typed. For example, the faulty program from earlier does not type-check:

```
let r = ref(λx.x) in
seq(assign(r,ref(λx.plus(x,num(1)))),deref(r) unit)
```

The polytype of the expression bound by 'r' is the singleton set $\{\mathit{aref}(\mathit{nat} \rightarrow^{\emptyset} \mathit{nat}, a)\}$ for some fresh alias a, bound in the alias type store to $(a \mapsto \mathit{nat} \rightarrow^{\emptyset} \mathit{nat})$. Polymorphic generalisation is restricted, since the alias a is in the type store, and the type store is not generalised by the rule.

Many programs that are rejected using the value restriction are accepted using store-based types. The following program now type-checks, for example:

$$\begin{array}{l}\texttt{let}\ \mathrm{y} = \lambda\mathrm{x.x}\ \texttt{in}\\ \texttt{let}\ \mathrm{id} = \mathrm{y}\ \texttt{in}\\ \texttt{seq}(\mathrm{id}\ \texttt{num}(1), \mathrm{id}\ \texttt{unit})\end{array}$$

The program is straightforwardly generalised, since types without free aliases are treated polymorphically. Thus, the polytyping for ‘y’ is inherited for ‘id’:

$$\{T \rightarrow\!\!\triangleright^{\emptyset} T \mid T \in \mathit{Type}\}$$

Similarly, the type of the program using the `expensive_function` is generalisable without eta-expansion:

$$\begin{array}{l}\texttt{let}\ \mathrm{f} = (\texttt{expensive\_function unit})\ \texttt{in}\\ \texttt{seq}(\mathrm{f}\ \texttt{num}(1), \mathrm{f}\ \texttt{unit})\end{array}$$

Functions that allocate aliases can also be polymorphically generalised. Consider, for example, the following program which binds a function for constructing new references:

$$\begin{array}{l}\texttt{let}\ \mathrm{mkref} = \lambda\mathrm{x}.\texttt{ref}(\mathrm{x})\ \texttt{in}\\ \texttt{seq}(\mathrm{mkref}\ \texttt{num}(1), \mathrm{mkref}\ \texttt{unit})\end{array}$$

The function bound by ‘mkref’ allocates a fresh reference, and we can infer the following polytype for it in the binder of the program above:

$$\{T \rightarrow\!\!\triangleright^{\{a \mapsto T\}} \mathit{aref}(T, a)\}$$

Although this polytype generalises over aliases, it has little impact on inference of the remainder of the program, since aliases are substituted in the application rule (XPZ-App) to ensure that they are fresh when functions are applied.

These examples illustrate how the extra information afforded by recording aliases in alias type stores is useful for deciding when polymorphic generalisation is safe. But the extra constraints of alias types also restricts certain programs from type checking.

Rigid reference typing. Alias reference types record aliases. This makes types involving aliases which are mapped to the same type syntactically distinct. For example, $\mathit{aref}(T, a_1)$ and $\mathit{aref}(T, a_2)$ where $a_1 \neq a_2$ cannot be used in the same context for the typing rules in Figure 8.2. This restricts typing certain programs that cannot go wrong, and that would be accepted by a type system with ML-style references. For example, a program such as the following is rejected:

$$(\lambda\mathrm{id}.\texttt{seq}(\mathrm{id}\ \texttt{ref}(1), \mathrm{id}\ \texttt{ref}(2)))\ (\lambda\mathrm{x.x})$$

The application rule (XPZ-App) does not admit polymorphic generalisation, so the function type scoped by id can only have either of the two types $aref(nat, a_1) \multimap^{\emptyset} aref(nat, a_1)$ or $aref(nat, a_2) \multimap^{\emptyset} aref(nat, a_2)$ where $a_1 \neq a_2$.

We could conceivably relax the type system to make it more flexible for programs such as the one above by introducing a suitable notion of type equality. A candidate rule would be:

$$\frac{a, a' \in \mathrm{dom}(\Sigma) \qquad \Sigma(a) = \Sigma(a') \qquad e_{/\mathbf{sto}\,\Sigma} : aref(T, a')_{/\mathbf{sto}\,\Sigma}}{e_{/\mathbf{sto}\,\Sigma} : aref(T, a)_{/\mathbf{sto}\,\Sigma}} \qquad \text{(XPZ-RefEq)}$$

A curious feature of store-based types is that type annotations may also inhibit type inference for programs that cannot go wrong. For example, the following program is rejected using store-based types:

$$(\lambda \mathrm{id}.\mathtt{seq}(\mathrm{id}\ \mathrm{id}, \mathrm{id}\ (\lambda \mathrm{x}.\mathtt{seq}(\mathtt{ref}(\mathtt{num}(1)), \mathrm{x})))) \ (\lambda \mathrm{x}.\mathrm{x})$$

This program applies the identity function to, respectively, itself and the function $(\lambda \mathrm{x}.\mathtt{seq}(\mathtt{ref}(\mathtt{num}(1)), \mathrm{x}))$ which allocates a reference and returns the value it is applied to. The program is rejected since the two types of these functions are distinct using store-based types: $T \multimap^{\emptyset} T$ and $T \multimap^{\{(a \mapsto T)\}} T$ for some a and T.

We could conceivably relax the type system with a suitable notion of sub-typing to make it more flexible for such programs. A candidate rule would be:

$$\frac{\Sigma \subseteq \Sigma' \qquad e : T_1 \multimap^{\Sigma'} T_2}{e : T_1 \multimap^{\Sigma} T_2} \qquad \text{(XPZ-StoreSub)}$$

where $\Sigma \subseteq \Sigma'$ checks that all mappings in Σ are also in Σ'.

The rest of this chapter focuses on the type system in Figure 8.2 without the candidate rules above.

8.3 Types as abstract interpretations for store-based types

We consider how to give meaning to store-based types using types as abstract interpretations. As with classic ML-style references, the store-based type system considered here uses invariant typing of store references. In order to give meaning to these, we use a refined semantics, similar to the refined semantics for ML reference types from the previous chapter.

8.3.1 Refined semantics for references

As in the previous section, we instrument the semantics to simplify the abstraction functions and type soundness proof. The refined semantics is given in Figure 8.1. The refinements are:

- references r are annotated with aliases a (written r^a) to relate them to entries in the runtime alias type store;
- recording the type of references in the runtime type store in rule (XSOS-PB-Ref); and
- checking that assignments preserve types in rule (XSOS-PB-Assign).

In the rest of this chapter we use $\lambda_{\text{cbv+}\mathtt{ref}}$ to refer to the traditional semantics for references given by the union of the extensible rule specifications for λ_{cbv} in Figure 5.2 on page 126, the rules for `let` in Figure 7.3 on page 150, and the rules for references in Figure 7.7 on page 155. We use $\lambda^{\text{SB}}_{\text{cbv+}\mathtt{ref}}$ to refer to the semantics given by these rules, but using the rules in Figure 8.1 for references instead.

We shortly consider how to define abstraction functions for the refined semantics. First, we state and prove two lemmas: one that relates refined evaluation for $\lambda^{\text{SB}}_{\text{cbv+}\mathtt{ref}}$ to ordinary evaluation for $\lambda_{\text{cbv+}\mathtt{ref}}$, and one that proves that refined evaluation is deterministic up-to the runtime alias type stores (Lemma 8.2).

Lemma 8.1 (Refined evaluation is sound wrt. ordinary evaluation) Using $\Downarrow^{\text{SB}}$ to denote the refined evaluation relation for $\lambda^{\text{SB}}_{\text{cbv+}\mathtt{ref}}$, and $\Downarrow^{\text{REF}}$ to denote the semantics for $\lambda_{\text{cbv+}\mathtt{ref}}$, it holds that:

$$R \vdash e_{/S} \Downarrow^{\text{SB}} e'_{/S'} \implies \mathit{erase}(R \vdash e_{/S} \Downarrow^{\text{REF}} e'_{/S'})$$

where *erase* erases all alias annotations on references in all-sub-terms.

Proof. The proof is by straightforward rule induction. □

Lemma 8.2 (Refined evaluation is deterministic up-to runtime alias type stores) The following holds about $\lambda^{\text{SB}}_{\text{cbv+}\mathtt{ref}}$:

$$R \vdash e_{/S} \Downarrow e'_{/S'} \implies R \vdash e_{/S} \Downarrow e''_{/S''} \implies e' = e'' \wedge \forall \Sigma.\ S'[\textbf{sto}\ \Sigma] = S''[\textbf{sto}\ \Sigma]$$

Proof. The proof is by straightforward rule induction on the first premise and inversion on the second. □

8.3.2 Typing and abstraction

We define the notion of typing and abstraction for store-based types. The set of typings is given by:

$$\mathit{Typing} \triangleq \mathit{TypeEnv} \times \mathit{AliasTypeStore} \times \mathit{Type} \times \mathit{AliasTypeStore}$$

The objective is to relate this set of typings to sets of denotations by means of an abstraction function. As before, we define an abstraction functions for each separate kind of entity in typings: a value abstraction function α_V and an environment abstraction function, α_R. Additionally, we introduce a store abstraction function, α_S for relating type-level and runtime type stores. We discuss each in turn, starting with store abstraction, since that is a novelty compared with earlier chapters.

Abstract syntax ($\Sigma^{\mathtt{SB}}_{\mathtt{ref}}$).

$$
\begin{aligned}
\mathit{Expr} \ni e &::= \mathtt{ref}(e) \mid \mathtt{deref}(e) \mid \mathtt{assign}(e,e) \mid v \\
\mathit{Val}^{\mathrm{NS}} \ni v &::= r^a \\
r \in \mathit{Ref} &\triangleq \{\mathrm{r}_1, \mathrm{r}_2, \ldots\} \\
a \in \mathit{Alias} &\triangleq \{\mathrm{a}_1, \mathrm{a}_2, \ldots\} \\
\sigma \in \mathit{Store} &\triangleq \mathit{Ref} \times \mathit{Alias} \xrightarrow{\mathrm{fin}} \mathit{Val}^{\mathrm{NS}} \\
\Sigma \in \mathit{AliasTypeStore} &\triangleq \mathit{Alias} \xrightarrow{\mathrm{fin}} \mathit{Type}
\end{aligned}
$$

Auxiliary entities ($\mathbb{C}^{\mathtt{SB}}_{\mathtt{ref}}$).

$$\mathbb{C}^{\mathtt{SB}}_{\mathtt{ref}} \triangleq \{(\mathbf{sto}, \mathbb{C}^{\mathrm{PREORDER}}(\mathit{Store})); (\mathbf{tsto}, \mathbb{C}^{\mathrm{PREORDER}}(\mathit{AliasTypeStore}))\}$$

Terminal configurations. (None required)
Rule specification.

$$\frac{e \Downarrow e' \qquad \mathtt{ref}(e') \Downarrow e''}{\mathtt{ref}(e) \Downarrow e''} \quad \text{(XSOS-PB-Ref1)}$$

$$\frac{r \notin \mathrm{dom}(\sigma) \qquad T \in \alpha^{\mathrm{SB}}_V(S[\mathbf{sto}\ \sigma, \mathbf{tsto}\ \Sigma], v)}{\mathtt{ref}(v)_{/S[\mathbf{sto}\ \sigma, \mathbf{tsto}\ \Sigma]} \Downarrow r^a_{/S[\mathbf{sto}\ \sigma[r^a \mapsto v], \mathbf{tsto}\ \Sigma[a \mapsto T]]}} \quad \text{(XSOS-PB-Ref)}$$

$$\frac{e \Downarrow e' \qquad \mathtt{deref}(e') \Downarrow e''}{\mathtt{deref}(e) \Downarrow e''} \quad \text{(XSOS-PB-Deref1)}$$

$$\frac{r \in \mathrm{dom}(\sigma)}{\mathtt{deref}(r^a)_{/S[\mathbf{sto}\ \sigma]} \Downarrow \sigma(r^a)_{/S[\mathbf{sto}\ \sigma]}} \quad \text{(XSOS-PB-Deref)}$$

$$\frac{e_1 \Downarrow e_1' \qquad \mathtt{assign}(e_1', e_2) \Downarrow e'}{\mathtt{assign}(e_1, e_2) \Downarrow e'} \quad \text{(XSOS-PB-Assign1)}$$

$$\frac{e_2 \Downarrow e_2' \qquad \mathtt{assign}(r^a, e_2') \Downarrow e'}{\mathtt{assign}(r^a, e_2) \Downarrow e'} \quad \text{(XSOS-PB-Assign2)}$$

$$\frac{r \in \mathrm{dom}(\sigma) \qquad a \in \mathrm{dom}(\Sigma) \qquad \Sigma(a) \in \alpha^{\mathrm{SB}}_V(S[\mathbf{sto}\ \sigma, \mathbf{tsto}\ \Sigma], v)}{\mathtt{assign}(r^a, v)_{/S[\mathbf{sto}\ \sigma, \mathbf{tsto}\ \Sigma]} \Downarrow \mathtt{unit}_{/S[\mathbf{sto}\ \sigma[r^a \mapsto v], \mathbf{tsto}\ \Sigma]}} \quad \text{(XSOS-PB-Assign)}$$

Figure 8.1: Abbreviated pretty-big-step XSOS rules for references with refined semantics for store-based typing

Store abstraction. Using the refined evaluation relation from Figure 8.1, the store abstraction function simply relates runtime alias type stores to themselves:

$$\alpha_S^{\text{SB}} \in O_S \to O_Z$$
$$\alpha_S^{\text{SB}}(S[\textbf{tsto}\,\Sigma]) \triangleq Z[\textbf{sto}\,\Sigma]$$

Value abstraction. The interesting abstraction functions for values are those for closures and references. The main difference from the abstraction function for closures in the previous chapter is that closures now record information about how evaluating the body of the closure affects the store. The case for the closure is an adaptation of the abstraction function from previous chapter, the only change being the additional requirement that the annotation Σ on function arrows safely abstract the set of references that are allocated when the function is called:

$$\alpha_V^{\text{SB}}(S, \langle x, e, \rho \rangle) \triangleq \left\{ T_1 \rightharpoonup^{\Sigma} T_2 \;\middle|\; \begin{array}{l} \forall R\, Z\, v_1\, S'.\, S \ll S' \implies T_1 \in \alpha_V^{\text{SB}}(S', v_1) \implies \\ \qquad \exists v_2\, S''.\, (v_2, S'') \in \mathbf{D}[\![e]\!](R[\textbf{env}\,\rho[x \mapsto v_1]])(S') \wedge \\ \qquad\quad T_2 \in \alpha_V^{\text{SB}}(S'', v_2) \wedge \\ \qquad\quad Z[\textbf{sto}\,\Sigma] = \alpha_S^{\text{SB}}(S''.\textbf{tsto} - S'.\textbf{tsto}) \end{array} \right\}$$

The abstraction function for stores captures the meaning of aliases. Thanks to our use of a refined semantics, the abstraction function for references is relatively simple:

$$\alpha_V^{\text{SB}}(S[\textbf{tsto}\,\Sigma], r^a) \triangleq \{aref(T, a) \mid \Sigma(a) = T\}$$

Environment abstraction. It only remains to define the environment abstraction function, which is similar to the one from previous chapter:

$$\alpha_R^{\text{SB}}(S, R) \triangleq \left\{ G[\textbf{env}\,\Xi] \;\middle|\; \begin{array}{l} \forall x \in \text{dom}(R.\textbf{env}). \\ \qquad x \in \text{dom}(\Xi) \wedge \Xi(x) \neq \emptyset \wedge \Xi(x) \subseteq \alpha_V^{\text{SB}}(S, R.\textbf{env}(x)) \end{array} \right\}$$

Well-typedness of runtime stores. The abstraction function α_S^{SB} relates runtime type stores to alias type stores in the type system. It does not say anything about the well-typedness of those runtime type stores, however. The following *well-typedness* property checks two things: for every alias in the runtime type store there is a corresponding annotated reference in the store, and vice versa; and all values in the store are typed by the corresponding type in the type store:

$$\begin{array}{l} wt(S) \iff (\forall a.\, (a \in \text{dom}(S.\textbf{tsto}) \iff \exists r.\, r^a \in \text{dom}(S.\textbf{sto}))) \wedge \\ \qquad\qquad \forall r^a \in \text{dom}(S.\textbf{sto}).\, S.\textbf{tsto}(a) \in \alpha_V^{\text{SB}}(S, S.\textbf{sto}(r^a))) \end{array}$$

Preordering. The value abstraction function α_V^{SB} above relies on a preordering of states. As in previous chapter, the role of the preorder is to capture the fact that extended stores preserve well-typedness, and preserve the types of references:

$$S_1 \ll S_2 \iff (wt(S_1) \implies wt(S_2) \wedge (S_1.\textbf{tsto} \subseteq S_2.\textbf{tsto}))$$

Lemma 8.3 shows that $\ll$ is a preorder.

Lemma 8.3 ($\ll$ is reflexive and transitive)

$$(\forall S.\, S \ll S) \;\wedge\; \big(\forall S\, S'\, S''.\, S \ll S' \implies S' \ll S'' \implies S \ll S''\big)$$

Proof. Both reflexivity and transitive follow straightforwardly from the definition of $\ll$. □

The preorder $\ll$ satisfies Assumptions 7.2 and 7.3 from previous chapter, proven in Lemmas 8.4 and 8.5.

Lemma 8.4 ($\lambda^{\mathrm{SB}}_{\mathrm{cbv+}\mathtt{ref}}$ satisfies Assumption 7.2) The following holds for the refined semantics, $\lambda^{\mathrm{SB}}_{\mathrm{cbv+}\mathtt{ref}}$:

$$R \vdash e_{/S} \Downarrow e'_{/S'} \implies S \ll S'$$

Proof. The proof is by straightforward rule induction on the evaluation relation. □

Lemma 8.5 ($\lambda^{\mathrm{SB}}_{\mathrm{cbv+}\mathtt{ref}}$ satisfies Assumption 7.3) The following holds for the refined semantics, $\lambda^{\mathrm{SB}}_{\mathrm{cbv+}\mathtt{ref}}$:

$$T \in \alpha^{\mathrm{SB}}_V(S, v) \implies S \ll S' \implies T \in \alpha^{\mathrm{SB}}_V(S', v)$$

Proof. The proof is by structural induction on the structure of T, using the transitivity of $\ll$ for the closure case. □

Thus equipped, we are ready to prove type soundness.

8.3.3 Type soundness

The soundness of store-based types is summarised by Theorem 8.6.

Theorem 8.6 The store-based type system in Figure 8.2 is type sound for $\lambda_{\mathrm{cbv+}\mathtt{ref}}$.

Proof. The proof is a direct consequence of Lemma 8.7 below and 8.1 (page 174). □

Lemma 8.7 The store-based type system in Figure 8.2 is type sound for the refined semantics in Figure 8.1 for $\lambda_{\mathrm{cbv+}\mathtt{ref}}$.

$$\begin{aligned}&G \vdash e_{/Z} : T_{/Z'} \implies G \in \alpha^{\mathrm{SB}}_R(S, R) \implies Z \in \alpha^{\mathrm{SB}}_S(S) \implies\\ &\qquad \exists v\, S'.\, R \vdash e_{/S} \Downarrow v_{/S'} \;\wedge\; T \in \alpha^{\mathrm{SB}}_V(S', v) \;\wedge Z' \in \alpha^{\mathrm{SB}}_S(S')\end{aligned}$$

Proof (sketch). The Coq formalisation accompanying this thesis provides an outline which relies on an axiomatisation of the relationship between aliases and references, as opposed to the annotation-based approach described above. The axiomatisation consists in asserting that there exists a substitution which relates any reference to an alias in the type store. This is equivalent to the abstraction function based on annotations.

Most cases of the proof follow by straightforward induction and the definition of abstraction, well-typedness, preordering, and the fact that evaluation preserves types (Lemmas 8.4 and 8.5). We highlight the central argument about polymorphism and refer the reader to the Coq axiomatisation for the remaining details: `http://cs.swansea.ac.uk/~cscbp/xtss.zip`.

Case (XSOS-PB-Let) The rules in Figure 8.2 permits polymorphic quantification in `let`-expressions. The central argument of this case is similar to that considered in previous chapters.

Recall the (unabbreviated) rule for `let`:

$$\frac{P \neq \emptyset \qquad (\forall T_1 \in P.\ G[\mathbf{env}\ \Xi] \vdash e_{1/Z} : T_{1/Z'}) \qquad G[\mathbf{env}\ \Xi[x \mapsto P]] \vdash e_{2/Z'} : T_{2/Z''}}{G[\mathbf{env}\ \Xi] \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_{2/Z} : T_{2/Z''}}$$

The induction hypotheses are:

$$\begin{aligned}&\forall T_1 \in P.\ \forall R\ S.\ wt(S) \implies \\ &\quad G[\mathbf{env}\ \Xi] \in \alpha_R^{\text{SB}}(S,R) \implies \\ &\quad Z = \alpha_S^{\text{SB}}(S) \implies \\ &\quad \exists v_1 S'.\ R \vdash e_{1/S} \Downarrow v_{1/S'} \ \wedge\ T_1 \in \alpha_V(S',v_1) \ \wedge\ Z' = \alpha_S^{\text{SB}}(S')\end{aligned} \tag{IH1}$$

$$\begin{aligned}&\forall R\ S.\ wt(S) \implies \\ &\quad G[\mathbf{env}\ \Xi[x \mapsto P]] \in \quad \alpha_R^{\text{SB}}(S,R) \implies \\ &\quad Z' = \alpha_S^{\text{SB}}(S) \implies \\ &\quad \exists v\ S'.\ R \vdash e_{2/S} \Downarrow v_{/S'} \ \wedge\ T_2 \in \alpha_V^{\text{SB}}(S',v) \ \wedge\ Z'' = \alpha_S(S')\end{aligned} \tag{IH2}$$

The goal is to find a configuration v_{1/S_1} for which $R \vdash e_{1/S} \Rightarrow v_{1/S_1}$, and which is typed by all $T \in P$ under Z_1, where Z_1 is the type store after inferring the type of the first premise, i.e. $G \vdash e_{1/Z} : T_{1/Z_1}$. If we can show that a configuration exists which is typed by all $T \in P$ under Z_2, the goal follows straightforwardly from the induction hypotheses and Lemma 8.8 (below). The existence of such a configuration is proven as follows:

- Choose any $T_0 \in P$.
- From the induction hypothesis, we get $R \vdash e_{1/S} \Rightarrow v_{1/S_1}$ such that $T_0 \in \alpha_V(S_1, v_1)$ and $Z_1 \in \alpha_S(S_1)$. (H). We prove that this v_{1/S_1} is typed by all $T \in P$
- For any $T_0' \in P$, we can use (IH1) to obtain a proof of $R \vdash e_{1/S} \Rightarrow v'_{1/S'_1}$ such that $T_0' \in \alpha_V(S_1', v_1')$ and $Z_1 = \alpha_S(S_1')$.
- We know from Lemma 8.2 above that $\Downarrow$ is deterministic up-to-the runtime type stores:
$$v_1 = v_1' \ \wedge\ \forall \Sigma.\ S_1[\mathbf{tsto}\ \Sigma] = S_1'[\mathbf{tsto}\ \Sigma]$$
- Observe that the type stores of S_1 and S_1' must be identical: since it holds that $Z_1 = \alpha_S(S_1)$ and also $Z_1 = \alpha_S(S_1')$, it must hold that:
$$Z_1.\mathbf{sto} = S_1.\mathbf{tsto} = S_1'.\mathbf{tsto}$$

which in turn implies:

$$S_1 = S'_1$$

Thus, v_{1/S_1} is typed by any $T \in P$.

The rest of the case follows from the induction hypotheses, Lemma 8.8, and this fact.

□

Lemma 8.8 (Abstraction preserves environment updates)

$$\begin{array}{l} G[\mathbf{env}\ \Xi] \in \alpha_R^{\text{SB}}(S, R[\mathbf{env}\ \rho]) \implies P \neq \emptyset \implies \\ \forall T \in P.\ T \in \alpha_V^{\text{SB}}(v)) \implies G[\mathbf{env}\ \Xi[x \mapsto P]] \in \alpha_R^{\text{SB}}(S, R[\mathbf{env}\ \rho[x \mapsto v]]) \end{array}$$

Proof. Immediate from the definition of α_R^{SB}. □

8.4 Assessment and related work

This chapter considered a type system that tracks how evaluation affects the store in a more fine-grained manner than traditional ML-style reference typing. We also considered types as abstract interpretations as a means of proving the type system sound. We assess the type system and its proof, and compare and contrast it with previous lines of work.

8.4.1 Related type systems

We compare and contrast store-based types with related type systems in the literature.

Type and effect systems According to Henglein et al.:

> Classical type systems express properties of values, not the computations leading to those values. Effect types describe all important effects of computation, not just their results.

The store-based type system considered in this chapter matches this description, but there are several differences between store-based type systems considered here and traditional type and effect systems in the literature [HMN04, NN99, TJ94, TT97, VJ95], the main difference being how effects are recorded: usually, effects are more abstract and record information such as which regions a program may affect, and such type systems generally do not rely on a store at the type level for inhibiting generalisation.

Alias types. The system this chapter proposes tracks the type of references in the store in a similarly fine-grained manner to Morrisett et al.'s works on Alias Types [SWM00], the Calculus of Capabilities [CWM99], and typed assembly language [Mor04]. The main focus of these lines of work is type systems for low-level languages written in continuation-passing style with explicit allocation of memory regions, where a region is a memory fragment which binds certain locations of certain types. They provide polymorphism at three different levels: polymorphism over types, locations, and regions. In their work, as in this chapter, polymorphism over types does not rely on restrictions such as the value restriction, since the type system accurately records the state of the current store. Our proposed reference type equality rule (XPZ-RefEq) appears to be related to their type location polymorphism: our proposed rule allows us to exchange any location with any other equivalent location, whereby all aliases implicitly become polymorphic.

Other approaches to relaxing the value restriction. Wright [Wri95] provides an overview of previous approaches to Hindley-Milner-Damas polymorphism in ML-like languages [Dam84, Tof90, MTH89, LW91, Wri92]. Not covered by Wright is Garrigues more recent work [Gar04], which uses a sub-typing based approach to relax the value restriction by generalising type variables that occur only at covariant positions.

Other type systems with type-level stores. In recent work on Dependent Types for JavaScript [CHJ12], Chugh et al. uses stores at the type level in a similar manner to the type system presented here. Unlike us, Chugh et al.'s type system supports strong updates, and uses dependent types for typing functions.

8.4.2 Types as abstract interpretations

The type soundness proof that this section recalled was an extension of the types as abstract interpretations approach from previous chapters to deal with type and effect systems. Many other authors have considered similar approaches to giving and proving the correctness of type and effect systems.

Recently, Galletta [Gal14] presented an abstract interpretation framework for type and effect systems. His framework is reminiscent of classical type and effect systems [HMN04, NN99, TJ94, TT97, VJ95]. It is not clear if the framework translates to a type system such as the one suggested here.

Viewing types as logical relations bears a lot of resemblance with viewing types as abstract interpretations. Ahmed's thesis [Ahm04] considers how to give meaning to ML reference types using logical relations, and uses the logical relation for proving type soundness. A significant difference between her approach and the one considered here (i.e., due to Cousot [Cou97]) is that her work is based on a small-step semantics.

8.4.3 Extensibility

The change from a classic type system approach to a type system involving effects required us to change the structure of types and typing rules for functions and applications. Our abstraction functions and type soundness proof also changed somewhat from ML reference types, the introduction of stores at the type level being the major change. While it is reasonable to expect that both types, rules, and proofs change when the fundamental meaning of types changes, perhaps one could have obtained a more flexible approach to giving and reasoning about rules if such reasoning was based on *typings*. If we could make types, typing relations and abstraction functions extensible up to the notion of typing, we might obtain proofs that are more amenable to type system extensions that go beyond classic type systems without considerable reformulation. Wells [Wel02] and Jim [Jim96] both argue the importance and usefulness of using typings as the basis for type systems and typing rules. Together with Mosses and Sculthorpe, the author has considered a preliminary investigation of how this works for a type system for dynamic scope [MSBP15]. It would be interesting to see how to the use of typings, rather than types, affects the extensibility of proofs.

9 Discussion and Future Directions

Contents

In this thesis, we have studied how to specify and relate operational semantics at various level of abstraction. Our thesis was:

> *Extensible transition system semantics provides a basis for giving and relating extensible and purely structural semantics at different levels of abstraction.*

We have confirmed this thesis by introducing extensible transitions system semantics as a basis for giving both small-step and big-step specifications of programming language semantics. Using the refocusing transformation, we proved that such small-step and pretty-big-step XSOS specifications are equivalent. Using types as abstract interpretations, we showed how to prove big-step type soundness based on extensible transition system semantics without artificial "wrong" transitions. Finally, we presented a novel type system for Hindley/Milner style polymorphism by recording effects in a type-level store. These contributions show that extensible transition system semantics provides a basis for giving and relating purely structural semantics at different levels of abstraction (big-step, small-step, and type systems).

This chapter recalls and assesses the main results and challenges in connection with each of these contributions, and outlines interesting directions to pursue in future research.

9.1 Extensible transition system semantics

Our main motivation for introducing and using extensible transition systems was to provide a more straightforward way of dealing with abrupt termination in both small-step and big-step purely structural semantics. As illustrated and argued throughout this thesis, XSOS achieves this goal by providing a unified way of expressing abrupt termination in both small-step and big-step semantics, such that rules for abrupt termination can be given once-and-for-all in either style, and such that these provably correspond to each other.

In order to represent abrupt termination and divergence in big-step semantics, we use Charguéraud's [Cha13] pretty-big-step style, which was introduced precisely to minimise the duplication problem for semantics with abrupt termination and divergence. Our pretty-big-step differs from Charguéraud's in a number of ways, however: notably, XSOS does not rely on abort rules for propagating abrupt termination or divergence if it arises. Our XSOS rules instead relies on a predicate for indicating whether configurations are final, such that we inhibit further computation for final configurations, such as values, abrupt termination, or computations that were supposed to have been executed after divergence has occurred. This means that we avoid the need to introduce abort rules if we extend a language with abrupt termination or divergence.

Our use of predicates for distinguishing final configurations also motivated our use of abbreviations in XSOS rules, in order to avoid having to write these out in rules always. Our abbreviations are inspired by Mosses and New's I-MSOS framework [MN09]. We believe that all of the abbreviations we introduced in Chapter 3 could have been formalised in a similar manner as Mosses and New's I-MSOS framework. Such a formalisation would enable alternative interpretations of propagating auxiliary entities between premises in rules, similar to [MN09, p. 61]. One might, for example, express interleaving by adopting a different interpretation of how effects arising in different premises of rules compose in terms of the underlying category. We expect that adapting I-MSOS to XSOS (or vice versa) should be relatively straightforward, due to the close relationship between XTTS and Mosses' GTTS [Mos04].

Mosses introduced MSOS as a solution to the modularity problem for SOS that is as effective as monad transformers are for denotational semantics [Mos04]. Indeed, the label propagation strategies in MSOS and abbreviations of I-MSOS exhibit a monadic structure: MSOS read-only entities are analogous to the *reader monad*; read-write entities are analogous to the *state monad*; and write-only entities are analogous to the *writer monad*. This thesis introduced an additional entity, namely entities modelled by the abrupt termination category (introduced in Section 3.5.1 of this thesis). As discussed in Chapter 3, using this category for abrupt termination is reminiscent of having a sum-type, similar to the *maybe monad*, but with a minor twist that the 'None' of the monad records the structure of the term in which abrupt termination occurred. This twist is crucial for giving semantics for control constructs, following Sculthorpe et al. [STM16] and Section 4.6 of this thesis.

Functional programming and denotational semantics recognises monads that permit much more sophisticated yet modular ways of altering how control and data flows in programs and semantics. Thus, it would be interesting to investigate the relationship between monads and MSOS/XSOS further.

Such an investigation might provide insights that might permit big-step XSOS rules to deal with interleaving and concurrency. Mosses [Mos04] suggests that composition in the underlying label category could be interpreted as a “shuffle” of the sequence (trace) of label components that a computation visits. Mitchell [Mit94] showed that this, indeed, provides a means of modelling concurrency using big-step semantics. The approach models concurrency and non-determinism by listening for input via ports. A big-step relation then generates the trace of inputs that leads to the value produced. Uustalu [Uus13] generalises this take on modelling big-step semantics even further by providing a relational representation of *resumptions*. Resumptions are a technique for dealing with concurrency in denotational semantics [Plo76], whose monadic counterpart is known as the *resumption monad* [PG14, HS00a, TA03]. Although the approach to dealing with concurrency is very general, Uustalu’s [Uus13] big-step rules read quite differently from big-step rules.

Although we provided a big-step semantics for delimited continuations in Section 4.6, it is not obvious how suitable this semantics is for reasoning about delimited continuations: most accounts of delimited continuations and call/cc in the literature are based on small-step semantics. It would be interesting to compare/contrast the pragmatic properties of small-step/big-step semantics for delimited continuations for different kinds of proofs. Big-step semantics is, for example, easier to work with in compiler correctness proofs. Is this true for semantics with continuations, too?

One of the main motivations behind this thesis was to provide techniques that apply to give and relate extensible specifications of the semantics of funcons, developed as part of the PLanCompS project. There are other lines of research that use (I-)MSOS as the basis for semantic specification: DynSem [VNV15] and the Spoofax workbench [VWT$^+$14] uses a variant of (I-)MSOS as the basis for execution [VNV15] and verification [VWT$^+$14]. We expect that our formulation of refocusing, in particular, would be relatively straightforward to adapt to DynSem rules.

9.2 Refocusing in XSOS

Chapter 4 presented an internalisation of Danvy and Nielsen’s transformation [DN04] in XSOS, and the subsequent Chapter 5 shows how the rules resulting from refocusing are amenable to reasoning about diverging computations as well as converging computations. The internalisation adopted and adapted the correctness criteria from [DN04], and provides a novel alternative to the correctness proof of Sieczkowski et al. [SBB11]. Sieczkowski et al.’s proof is based on an axiomatisation of reduction semantics. In contrast, our proof is based on a rule scheme that generalises well-known proof methods for relating small-step and big-step semantics from the literature on SOS [Nip06, LG09, NK14, Cio13, PCG$^+$13].

It is well-known that reduction semantics and structural operational semantics are closely related, in particular for context-insensitive rules [Dan04, Dan08]. Our motivation for internalising the framework of XSOS in this thesis was our interest in applying the transformation to small-step XSOS rules for funcons to derive (pretty-)big-step XSOS rules, where many of the small-step XSOS rules are context-sensitive, in that they crucially rely on the propagation of auxiliary entities in rules (such as environments).

One might have side-stepped such issues by using Curien's framework for explicit substitutions [Cur91, ACCL91]. Biernacka and Danvy [BD07] investigate this approach in their syntactic correspondence, where they show how to refocus a reduction semantics based on explicit substitutions, and end up with an environment-based abstract machine. One might also have expressed the semantics by means of context-sensitive rules that satisfy the correctness criteria given by Sieczkowsi et al. An additional artifact resulting from such an adaptation would be a reduction semantic representation for funcons, which could be useful in its own right. We leave an exploration of this direction to future work.

Danvy et al.'s work focuses mainly on functional representations of reduction semantics and abstract machines. Working with such representations is somewhat challenging in theorem provers, such as Coq or Agda, where functions must either be guaranteed to *terminate* (structurally decreasing on input), or guaranteed to be *productive* (structurally increasing on output). One way of avoiding this is to introduce a "clock" in functions, which trivially decreases with each recursive call. Recently, Dubois et al. [TDD12] investigates how to automatically derive such functional representations from inductive specifications in Coq. Ramana et al. use a similar technique for expressing and reasoning about CakeML, a verified implementation of ML [KMNO14] in the proof assistant HOL4.[1]

9.3 Types as abstract interpretations

Chapter 6 considered a straightforward adaptation to SOS of Cousot's types as abstract interpretations [Cou97]. This allowed us to prove type soundness and strong normalisation in one fell swoop. The technique only works because we were working with a deterministic language, however. For non-deterministic languages, the type soundness statement used in that chapter is too weak. The type soundness property can be summarised as: "if we can infer that the program is well-typed, then there exists a value to which the program evaluates, and which has the expected type". But if the program is non-deterministic, the existence of a trace that leads to a value of the right type does not imply that there are no traces which go wrong. For such semantics, one has to resort to different means of proving type soundness, such as a more traditional preservation-style proof using a semantics with explicit "wrong" transitions.

For deterministic and strongly normalising languages, however, types as abstract interpretations provides a simple and straightforward way of proving both type soundness and strong normalisation.

Our Coq encoding of abstraction functions are expressed as functions that are structurally decreasing on types. This naturally only works for finite types. It is not clear if it is possible to represent and work with infinite types in Coq using types as abstract interpretations in an equally straightforward way. This would be interesting to investigate further. A related research question is whether the types as abstract interpretations

[1]`http://hol.sourceforge.net`

approach considered in Chapter 6 of this thesis is a useful means of calculating (in the sense of Bahr and Hutton [BH15]) calculi that satisfy strong normalisation?

9.4 Modular big-step type soundness

One of the questions we asked ourselves in Chapter 6 was whether types as abstract interpretations provides a feasible means of giving extensible proofs of type soundness. Our study does not provide a clear-cut answer to this question. While the type soundness statement does not change as we extend our language, the proof that we gave relied on an instrumented semantics, and one that relies on a mutual dependency between the abstraction functions and the semantics, which we axiomatised in Coq.

As for the mutual dependency, we expect that it is not so severe as it may appear: the relation and function appears to be an instance of *induction-recursion* [Dyb00, DS06, GMNF13], which is known to be consistent.

9.5 Type and effect soundness

Chapter 8 provides a novel type system for Hindley-Milner-Damas polymorphism in the presence of references. We also provided a proof based on an axiomatisation in Coq, where both the mutual dependence between semantics/abstraction function is axiomatised, but also the relationship between type-level aliases and actual runtime references. The axiomatisation consists of an axiom in Coq saying that there is a substitution which maps any references to its corresponding alias such that every reference that occurs in the store is mapped to an alias in the type-level store; and, conversely, every alias in the type-level store is mapped to one or more references.

We do expect it to be straightforward to replace the axiomatisation with the refined semantics based on annotated references in Section 8.3.1, instead of the axiomatisation that is in our current Coq proof. We do not expect this to change the structure of the proof substantially. We leave it to future work to verify this.

Besides analysing the relationship between store-based types and previous approaches for dealing with types and effects for Hindley-Milner-Damas type systems outlined in Section 8.4, an interesting question in connection with the store-based type system in Chapter 8 is whether it is sound to translate store-based types to ML-style reference types [MTHM97].

9.6 Conclusion

This thesis studied how to specify, relate, and work with operational semantics that support language evolution at different levels of abstraction, focusing particularly on purely structural operational semantics, i.e., semantics without any explicit syntactic representation of program context. We proposed a simple but novel variant of Mosses' generalised transition systems [Mos04] that we called *extensible transition systems*. Our thesis was:

Extensible transition system semantics provides a basis for giving and relating extensible and purely structural semantics at different levels of abstraction.

Chapters 3 through to 8 shows that extensible transition system semantics provides a basis for giving and relating extensible and purely structural small-step and big-step semantics, as well as type (and effect) systems. We also considered types as abstract interpretations as an interesting proof method for big-step type soundness without explicit “wrong” transitions.

Extensible transition system semantics provides a means of specifying, once-and-for-all, a variety of different language features, including applicative and imperative features, abrupt termination, divergence, and even continuations. Such semantics have provably corresponding small-step and big-step interpretations. In the process of exploring extensible transition system semantics, we have seen a number of techniques that might be useful for other settings, including a generalised proof method for relating diverging computations in small-step and big-step SOS (Chapter 5), and using types as abstract interpretations for a direct simultaneous proof of strong type soundness and strong normalisation. Future work includes investigating the correspondence between XSOS and reduction semantics, investigating the mutual dependency used in our instrumented semantics for ML references, and investigating store-based types further.

A Proofs for Chapter 2

This appendix contains proofs of various propositions recalled in the background chapter of this thesis, and specifically Section 2.4. The proofs are all about the λ_{cbv} language, using the small-step relations $(\rightarrow, \overset{\infty}{\rightarrow})$ defined in Sections 2.2.2 and 2.2.4, and the big-step relations $(\Rightarrow, \overset{\infty}{\Rightarrow})$ defined in Sections 2.4.2 and 2.4.4.

All of the proofs follow a similar structure to those of Leroy and Grall [LG09]. Unlike Leroy and Grall who study a semantics based on meta-level substitution, the semantics we study here is based on semantics with environments. Furthermore, the semantics we study use two different term and value universes for the small-step and big-step relations. This entails some extra plumbing in the proofs. Chapters 4 and 5 consider refocused extensible XSOS rules, whose correctness is proven using generalised versions of Leroy and Grall's proofs. The proofs presented in these chapters essentially provide an automatic way of relating small-step and big-step semantics without all the tedious proof plumbing used in this appendix.

A.1 Expression sorts

The relations $\rightarrow, \Rightarrow$ rely on two different expression sorts:

Natural semantic syntax	**SOS syntax**
$\begin{aligned} Expr^{\text{NS}} \ni e ::= {} & \texttt{plus}(e,e) \mid \texttt{num}(n) \\ \mid {} & \lambda x.e \mid e\ e \mid x \end{aligned}$	$\begin{aligned} Expr \ni e ::= {} & \texttt{plus}(e,e) \mid v \\ \mid {} & \lambda x.e \mid e\ e \mid x \end{aligned}$
$Val^{\text{NS}} \ni v ::= n \mid \langle x, e, \rho \rangle$	$Val \ni v ::= n \mid \langle x, e, \rho \rangle$
$\rho \in Env^{\text{NS}} \triangleq Var \rightarrow Val^{\text{NS}}$	$\rho \in Env \triangleq Var \rightarrow Val$

We relate the two semantics by defining a big-step relation operating on the SOS syntax instead of the natural semantic syntax. We use '$\Rightarrow$' and '$\overset{\infty}{\Rightarrow}$' to refer to the natural semantic big-step relations from Section 2.4.2 and 2.4.4; and $\overset{\text{S}}{\Rightarrow}$ and $\overset{\text{S}\infty}{\Longrightarrow}$ to refer to the big-step relation operating on SOS syntax instead. Figures A.1 and A.2 define these relations.

The $\overset{\text{S}}{\Rightarrow}$ and $\overset{\text{S}\infty}{\Longrightarrow}$ relations correspond to $\Rightarrow$ and $\overset{\infty}{\Rightarrow}$. In order to prove this, we first define some simple auxiliary predicates: $\mathsf{source}(\_), \mathsf{vsource}(\_), \mathsf{rsource}(\_)$ identify

$$\frac{}{\rho \vdash v \overset{\mathrm{S}}{\Rightarrow} v} \quad \text{(SNS-}\lambda_{\text{cbv}}\text{-Refl)}$$

$$\frac{\rho \vdash e_1 \overset{\mathrm{S}}{\Rightarrow} n_1 \qquad \rho \vdash e_2 \overset{\mathrm{S}}{\Rightarrow} n_2}{\rho \vdash \mathtt{plus}(e_1, e_2) \overset{\mathrm{S}}{\Rightarrow} n_1 + n_2} \quad \text{(SNS-}\lambda_{\text{cbv}}\text{-Plus)}$$

$$\frac{}{\rho \vdash \lambda x.e \overset{\mathrm{S}}{\Rightarrow} \langle x, e, \rho \rangle} \quad \text{(SNS-}\lambda_{\text{cbv}}\text{-Lam)}$$

$$\frac{\rho \vdash e_1 \overset{\mathrm{S}}{\Rightarrow} \langle x, e, \rho' \rangle \qquad \rho \vdash e_2 \overset{\mathrm{S}}{\Rightarrow} v_2 \qquad \rho'[x \mapsto v_2] \vdash e \overset{\mathrm{S}}{\Rightarrow} v}{\rho \vdash e_1\ e_2 \overset{\mathrm{S}}{\Rightarrow} v} \quad \text{(SNS-}\lambda_{\text{cbv}}\text{-App)}$$

$$\frac{x \in \mathrm{dom}(\rho)}{\rho \vdash x \overset{\mathrm{S}}{\Rightarrow} \rho(x)} \quad \text{(SNS-}\lambda_{\text{cbv}}\text{-Var)}$$

Figure A.1: Natural semantics for converging computations of λ_{cbv} with SOS syntax

$$\frac{\rho \vdash e_1 \overset{\mathrm{S}\infty}{\Longrightarrow}}{\rho \vdash \mathtt{plus}(e_1, e_2) \overset{\mathrm{S}\infty}{\Longrightarrow}} \quad \text{(SNS-}\lambda_{\text{cbv}}\text{-}\infty\text{-Plus1)}$$

$$\frac{\rho \vdash e_1 \overset{\mathrm{S}}{\Rightarrow} n \qquad \rho \vdash e_2 \overset{\mathrm{S}\infty}{\Longrightarrow}}{\rho \vdash \mathtt{plus}(e_1, e_2) \overset{\mathrm{S}\infty}{\Longrightarrow}} \quad \text{(SNS-}\lambda_{\text{cbv}}\text{-}\infty\text{-Plus2)}$$

$$\frac{\rho \vdash e_1 \overset{\mathrm{S}\infty}{\Longrightarrow}}{\rho \vdash e_1\ e_2 \overset{\mathrm{S}\infty}{\Longrightarrow}} \quad \text{(SNS-}\lambda_{\text{cbv}}\text{-}\infty\text{-App1)}$$

$$\frac{\rho \vdash e_1 \overset{\mathrm{S}}{\Rightarrow} \langle x, e, \rho' \rangle \qquad \rho \vdash e_2 \overset{\mathrm{S}\infty}{\Longrightarrow}}{\rho \vdash e_1\ e_2 \overset{\mathrm{S}\infty}{\Longrightarrow}} \quad \text{(SNS-}\lambda_{\text{cbv}}\text{-}\infty\text{-App2)}$$

$$\frac{\rho \vdash e_1 \overset{\mathrm{S}}{\Rightarrow} \langle x, e, \rho' \rangle \qquad \rho \vdash e_2 \overset{\mathrm{S}}{\Rightarrow} v_2 \qquad \rho'[x \mapsto v_2] \vdash e \overset{\mathrm{S}\infty}{\Longrightarrow}}{\rho \vdash e_1\ e_2 \overset{\mathrm{S}\infty}{\Longrightarrow}} \quad \text{(SNS-}\lambda_{\text{cbv}}\text{-}\infty\text{-App3)}$$

Figure A.2: Natural semantics for diverging computations of λ_{cbv} with SOS syntax

subsets of SOS syntax expressions, values, and environments which correspond to natural semantic expressions, values, and environments. We also define relations $\mathsf{nsify}(\_), \mathsf{vnsify}(\_), \mathsf{rnsify}(\_)$ that map SOS syntax expressions, values, and environments to natural semantic expressions, values, and environments.

Definition A.1 Let $\mathsf{source} \subseteq \mathit{Expr}$ be a predicate that identifies *source programs*, defined as follows, where we write $\mathsf{source}(e)$ for $e \in \mathsf{source}$:

$$\begin{aligned}
&\mathsf{source}(\mathtt{plus}(e_1, e_2)) \text{ if } \mathsf{source}(e_1) \text{ and } \mathsf{source}(e_2)\\
&\mathsf{source}(n)\\
&\mathsf{source}(\lambda x.e) \text{ if } \mathsf{source}(e)\\
&\mathsf{source}(e_1\, e_2) \text{ if } \mathsf{source}(e_1) \text{ and } \mathsf{source}(e_2)\\
&\mathsf{source}(x)
\end{aligned}$$

Let $\mathsf{vsource} \subseteq \mathit{Val}$ be a predicate that identifies *values resulting from evaluating source programs*, defined as follows, where we write $\mathsf{vsource}(v)$ for $v \in \mathsf{vsource}$:

$$\begin{aligned}
&\mathsf{vsource}(n)\\
&\mathsf{vsource}(\langle x, e, \rho \rangle) \text{ if } \mathsf{source}(e) \text{ and } \mathsf{rsource}(\rho)
\end{aligned}$$

Let $\mathsf{rsource} \subseteq \mathit{Env}$ be a predicate that identifies environments whose values are all in $\mathsf{vsource}$:

$$\mathsf{rsource}(\rho) \text{ if } \forall x \in \mathrm{dom}(\rho).\ \mathsf{vsource}(\rho(x))$$

Definition A.2 Let $\mathsf{nsify} \in \mathit{Expr} \to \mathit{Expr}^{\mathrm{NS}}$ translate SOS syntax terms to natural semantic expressions:

$$\begin{aligned}
\mathsf{nsify}(\mathtt{plus}(e_1, e_2)) &= \mathtt{plus}(\mathsf{nsify}(e_1), \mathsf{nsify}(e_2))\\
\mathsf{nsify}(n) &= \mathtt{num}(n)\\
\mathsf{nsify}(\langle x, e, \rho \rangle) &= \mathtt{num}(0)\\
\mathsf{nsify}(\lambda x.e) &= \lambda x.e\\
\mathsf{nsify}(e_1\, e_2) &= \mathsf{nsify}(e_1)\, \mathsf{nsify}(e_2)\\
\mathsf{nsify}(x) &= x
\end{aligned}$$

Let $\mathsf{vnsify} \in \mathit{Val} \to \mathit{Val}^{\mathrm{NS}}$ translate SOS syntax values to natural semantic values:

$$\begin{aligned}
\mathsf{vnsify}(n) &= n\\
\mathsf{vnsify}(\langle x, e, \rho \rangle) &= \langle x, \mathsf{nsify}(e), \mathsf{rnsify}(\rho) \rangle
\end{aligned}$$

Let $\mathsf{rnsify} \in \mathit{Env} \to \mathit{Env}^{\mathrm{NS}}$ translate SOS syntax environments to natural semantic environments:

$$\mathsf{rnsify}(\rho) = \{x \mapsto \mathsf{vnsify}(\rho(x)) \mid x \in \mathrm{dom}(\rho)\}$$

Here, nsify translates closures to the natural number '0'; since closures are not source terms, they are irrelevant for the properties we now prove.

A.2 Correspondence between converging computations

Lemma A.3 ($\overset{\mathsf{S}}{\Rightarrow}$ corresponds to $\Rightarrow$) It holds that:

$$\mathsf{rsource}(\rho) \implies \mathsf{source}(e) \implies \\ \Big(\mathsf{rnsify}(\rho) \vdash \mathsf{nsify}(e) \overset{\mathsf{S}}{\Rightarrow} \mathsf{vnsify}(v) \iff \rho \vdash e \Rightarrow v\Big)$$

Proof. Each direction is proven by rule induction. For the $\Rightarrow$-to-$\overset{\mathsf{S}}{\Rightarrow}$ direction, the only interesting case is that for the application, (NS-λ_{cbv}-App). The rest follow straightforwardly from the induction hypothesis, the compositional nature of nsify, and the definitions of vnsify, rnsify.

Case (NS-λ_{cbv}-App) From the goal, (NS-λ_{cbv}-App) and the definition of source, we have:

$$\mathsf{rsource}(\rho) \tag{H1}$$

$$\mathsf{source}(e_1) \tag{H2}$$

$$\mathsf{source}(e_2) \tag{H3}$$

From the induction hypothesis, we have:

$$\mathsf{rsource}(\rho) \implies \mathsf{source}(e_1) \implies \\ \mathsf{rnsify}(\rho) \vdash \mathsf{nsify}(e_1) \overset{\mathsf{S}}{\Rightarrow} \mathsf{vnsify}(\langle x, e, \rho' \rangle) \tag{IH1}$$

$$\mathsf{rsource}(\rho) \implies \mathsf{source}(e_2) \implies \\ \mathsf{rnsify}(\rho) \vdash \mathsf{nsify}(e_2) \overset{\mathsf{S}}{\Rightarrow} \mathsf{vnsify}(v_2) \tag{IH2}$$

$$\mathsf{rsource}(\rho'[x \mapsto v_2]) \implies \mathsf{source}(e) \implies \\ \mathsf{rnsify}(\rho'[x \mapsto v_2]) \vdash \mathsf{nsify}(e) \overset{\mathsf{S}}{\Rightarrow} \mathsf{vnsify}(v) \tag{IH3}$$

The goal is:

$$\mathsf{rnsify}(\rho) \vdash \mathsf{nsify}(e_1\, e_2) \Rightarrow \mathsf{vnsify}(v) \tag{Goal}$$

The hypotheses for (IH1) and (IH2) are trivially satisfied. In order to satisfy the hypotheses for (IH3), we apply Lemma A.4 (see below) to (IH1) and (IH2). The goal now follows straightforwardly.

The $\overset{S}{\Rightarrow}$-to-$\Rightarrow$ direction of the proof is equally straightforward. □

Lemma A.4 (Source is preserved by evaluation) It holds that:

$$\mathsf{rsource}(\rho) \implies \mathsf{source}(e) \implies \rho \vdash e \overset{S}{\Rightarrow} v \implies \mathsf{vsource}(v)$$

Proof. The proof is by rule induction on $\overset{S}{\Rightarrow}$. Each case follows straightforwardly from the induction hypotheses, and from the definitions of $\mathsf{rsource}, \mathsf{source}, \mathsf{vsource}$. □

Lemma A.5 ($\overset{S\infty}{\Longrightarrow}$ corresponds to $\overset{\infty}{\Rightarrow}$) It holds that:

$$\mathsf{source}(e) \implies \left(\mathsf{nsify}(e) \overset{S\infty}{\Longrightarrow} \iff e \overset{\infty}{\Rightarrow}\right)$$

Proof. The proof of each direction is by coinduction, and follows along the same line of reasoning as Lemma A.3. The proof also uses this lemma in order to relate the converging branches for $\overset{S\infty}{\Longrightarrow}$ and $\overset{\infty}{\Rightarrow}$. □

Lemma A.6 ($\overset{S}{\Rightarrow}$ implies $\rightarrow^*$) It holds that:

$$\rho \vdash e \overset{S}{\Rightarrow} v \implies \rho \vdash e \rightarrow^* v$$

Proof. The proof proceeds by rule induction on $\overset{S}{\Rightarrow}$. The simple rules follow trivially. The rest follow straightforwardly from the congruence of $\rightarrow^*$ proven in Lemma A.7 below. We show the case for `plus`. The rest are analogous.

Case (SNS-λ_{cbv}-Plus) We have as induction hypotheses:

$$\rho \vdash e_1 \rightarrow^* n_1 \tag{IH1}$$

$$\rho \vdash e_2 \rightarrow^* n_2 \tag{IH2}$$

The goal is:

$$\rho \vdash \mathtt{plus}(e_1, e_2) \rightarrow^* n_1 + n_2 \tag{Goal}$$

By transitivity of $\rightarrow^*$, the (Goal) holds exactly when:

$$\rho \vdash \mathtt{plus}(e_1, e_2) \rightarrow^* \mathtt{plus}(n_1, e_2) \tag{Goal1}$$

$$\rho \vdash \mathtt{plus}(n_1, e_2) \rightarrow^* \mathtt{plus}(n_1, n_2) \tag{Goal2}$$

$$\rho \vdash \mathtt{plus}(n_1, n_2) \rightarrow^* n_1 + n_2 \tag{Goal3}$$

The first two goals follow from Lemma A.7 and the induction hypotheses. The last goal follows from (Trans), (SOS-λ_{cbv}-Plus), and (Refl)

□

Lemma A.7 (Congruence of $\rightarrow^*$) It holds that:

(a) $\rho \vdash e_1 \rightarrow^* e_1' \implies \rho \vdash \mathtt{plus}(e_1, e_2) \rightarrow^* \mathtt{plus}(e_1', e_2)$

(b) $\rho \vdash e_2 \rightarrow^* e_2' \implies \rho \vdash \mathtt{plus}(e_1, e_2) \rightarrow^* \mathtt{plus}(e_1', e_2)$

(c) $\rho \vdash e_1 \rightarrow^* e_1' \implies \rho \vdash e_1\, e_2 \rightarrow^* e_1'\, e_2$

(d) $\rho \vdash e_2 \rightarrow^* e_2' \implies \rho \vdash \langle x, e, \rho' \rangle\, e_2 \rightarrow^* \langle x, e, \rho' \rangle\, e_2'$

(e) $\rho'[x \mapsto v_2] \vdash e \rightarrow^* e' \implies \rho \vdash \langle x, e, \rho' \rangle\, v_2 \rightarrow^* \langle x, e', \rho' \rangle\, v_2$

Proof. Each of the properties above follow by the same line of reasoning. We consider the case for property (a); the rest are analogous. The proof proceeds by rule induction on $\rightarrow^*$.

Case (Refl) The goal follows trivially.

Case (Trans) We have as a hypotheses:

$$\rho \vdash e_1 \rightarrow e_1' \tag{H1}$$

$$\rho \vdash e_1' \rightarrow^* e_1'' \tag{H2}$$

The induction hypothesis gives:

$$\rho \vdash \mathtt{plus}(e_1', e_2) \rightarrow^* \mathtt{plus}(e_1'', e_2) \tag{IH}$$

The goal is:

$$\rho \vdash \mathtt{plus}(e_1, e_2) \rightarrow^* \mathtt{plus}(e_1'', e_2) \tag{Goal}$$

By transitivity of $\rightarrow^*$, we get the two goals:

$$\rho \vdash \mathtt{plus}(e_1, e_2) \rightarrow^* \mathtt{plus}(e_1', e_2) \tag{Goal1}$$

$$\rho \vdash \mathtt{plus}(e_1', e_2) \rightarrow^* \mathtt{plus}(e_1'', e_2) \tag{Goal2}$$

(Goal1) follows by applying (Trans), (SOS-λ_{cbv}-Plus1), H1, and (Refl). (Goal2) follows from the induction hypothesis.

□

Lemma A.8 ($\rightarrow^*$ implies $\overset{\mathrm{S}}{\Rightarrow}$) It holds that:

$$\rho \vdash e \rightarrow^* v \implies \rho \vdash e \overset{\mathrm{S}}{\Rightarrow} v$$

Proof. The proof follows by induction on $\rightarrow^*$ and by Lemma A.9 below. □

Lemma A.9 (Small step and $\overset{\mathrm{S}}{\Rightarrow}$ can be fused to big step) It holds that:

$$\rho \vdash e \rightarrow e' \implies \rho \vdash e' \overset{\mathrm{S}}{\Rightarrow} v \implies \rho \vdash e \overset{\mathrm{S}}{\Rightarrow} v$$

Proof. The proof is by rule induction on $\to$. The cases all follow a similar structure; we consider the case for (SNS-λ_{cbv}-Plus).

Case (SNS-λ_{cbv}-Plus) From the goal we have the following hypotheses:

$$\rho \vdash e_1 \to e_1' \tag{H1}$$

$$\rho \vdash \texttt{plus}(e_1', e_2) \overset{\text{S}}{\Rightarrow} v \tag{H2}$$

From the induction hypothesis we have:

$$\forall v_1.\ \rho \vdash e_1' \overset{\text{S}}{\Rightarrow} v_1 \implies \rho \vdash e_1 \overset{\text{S}}{\Rightarrow} v_2 \tag{IH}$$

The goal is:

$$\rho \vdash \texttt{plus}(e_1, e_2) \overset{\text{S}}{\Rightarrow} v \tag{Goal}$$

By inversion on (H2), we get:

$$\rho \vdash e_1' \overset{\text{S}}{\Rightarrow} n_1 \tag{H3}$$

$$\rho \vdash e_2 \overset{\text{S}}{\Rightarrow} n_2 \tag{H4}$$

$$v = n_1 + n_2 \tag{H5}$$

The goal follows from rule (SNS-λ_{cbv}-Plus), (IH), (H3), (H4), and (H5).

□

A.3 Correspondence between diverging computations

In order to prove Lemma A.11 below, we introduce a slight variant of the small-step semantics for λ_{cbv} from Figure 2.6 on page 21. The motivation for this variation is motivated by Lemma A.11 below: the non-compositional nature of the application rule means that induction alone is not sufficient for the proof. In comparison, Leroy and Grall's [LG09] small-step rules do not have the same defect, since their small-step rules are compositional, due to using substitution instead of environments.

The variation between the semantics for λ_{cbv} from Figure 2.6 on page 21 and the one we are about to introduce consists in replacing application with two constructs, `app` and `force`:

$\mathit{Expr}^{\text{F}} \ni e ::= \ldots \mid \lambda x.e \mid \texttt{app}(e,e) \mid \texttt{force}(x,v,\rho,e) \mid x$	Expressions
$\mathit{Val}^{\text{F}} \ni v ::= \ldots \mid \langle x,e,\rho \rangle$	Values
$x,y \in \mathit{Var} \triangleq \{\mathrm{x},\mathrm{y},\ldots\}$	Variables
$\rho \in \mathit{Env}^{\text{F}} \triangleq \mathit{Var} \xrightarrow{\text{fin}} \mathit{Val}^{\text{F}}$	Environments

We let the semantics of this updated syntax be given by all rules for λ_{cbv} from Figure 2.6 on page 21, except we replace the rules for application by the following:

$$\frac{\rho \vdash e_1 \to^{\text{F}} e_1'}{\rho \vdash \texttt{app}(e_1, e_2) \to^{\text{F}} \texttt{app}(e_1', e_2)} \quad (\text{SOS-}\lambda_{\text{cbv}}\text{-App1}')$$

$$\frac{\rho \vdash e_2 \to^{\text{F}} e_2'}{\rho \vdash \texttt{app}(\langle x, e, \rho' \rangle, e_2) \to^{\text{F}} \texttt{app}(\langle x, e, \rho' \rangle, e_2')} \quad (\text{SOS-}\lambda_{\text{cbv}}\text{-App2}')$$

$$\frac{}{\rho \vdash \texttt{app}(\langle x, e, \rho' \rangle, v_2) \to^{\text{F}} \texttt{force}(x, v, \rho', e)} \quad (\text{SOS-}\lambda_{\text{cbv}}\text{-AppF})$$

$$\frac{\rho'[x \mapsto v] \vdash e \to^{\text{F}} e'}{\rho \vdash \texttt{force}(x, v, \rho', e) \to^{\text{F}} \texttt{force}(x, v, \rho', e')} \quad (\text{SOS-}\lambda_{\text{cbv}}\text{-Force1})$$

$$\frac{}{\rho \vdash \texttt{force}(x, v_0, \rho', v) \to^{\text{F}} v} \quad (\text{SOS-}\lambda_{\text{cbv}}\text{-Force})$$

It is easily seen that the transition relation $\to^{\text{F}}$ for the λ_{cbv} semantics with app and force is equivalent to the original transition relation $\to$ for λ_{cbv}: for any step we can make using $\to$, we can make either one or two steps using $\to^{\text{F}}$ to end up in an equivalent configuration (two steps in case of application, such that each semantics does a single step inside the body of the function being applied); and vice versa, if $\to^{\text{F}}$ does two steps, or one step that results in a value, we do either two steps or one step using $\to$ to end up in an equivalent configuration (one step whenever we either make a transition to a value, or one step in the case of application, such that each semantics only does a single step inside the body of the function being applied).

Let f2e be a function that translates any $e \in \mathit{Expr}^{\text{F}}$ to an $e \in \mathit{Expr}$:

$$\begin{aligned}
\mathsf{f2e} &\in \mathit{Expr}^{\text{F}} \to \mathit{Expr} \\
\mathsf{f2e}(\texttt{app}(e_1, e_2)) &\triangleq \mathsf{f2e}(e_1)\ \mathsf{f2e}(e_2) \\
\mathsf{f2e}(\texttt{force}(x, v, \rho, e)) &\triangleq \mathsf{f2e}(\langle x, e, \rho \rangle)\ \mathsf{f2v}(v) \\
\mathsf{f2e}(\texttt{plus}(e_1, e_2)) &\triangleq \texttt{plus}(\mathsf{f2e}(e_1), \mathsf{f2e}(e_2)) \\
\mathsf{f2e}(\lambda x.e) &\triangleq \lambda x.\mathsf{f2e}(e) \\
\mathsf{f2e}(v) &\triangleq \mathsf{f2v}(v)
\end{aligned}$$

$$\begin{aligned}
\mathsf{f2v} &\in \mathit{Val}^{\text{F}} \to \mathit{Val} \\
\mathsf{f2v}(\langle x, e, \rho \rangle) &\triangleq \langle x, \mathsf{f2e}(e), \mathsf{f2r}(\rho) \rangle \\
\mathsf{f2v}(n) &\triangleq n \\
\\
\mathsf{f2r} &\in \mathit{Env}^{\text{F}} \to \mathit{Env} \\
\mathsf{f2r}(\rho) &\triangleq \{x \mapsto \mathsf{f2v}(\rho(x)) \mid x \in \text{dom}(\rho)\}
\end{aligned}$$

As argued above, if we can prove $\mathsf{f2r}(\rho) \vdash \mathsf{f2e}(e) \overset{\text{S}\infty}{\Longrightarrow} \implies \rho \vdash e \overset{\infty}{\to}^{\text{F}}$, where $\overset{\infty}{\to}^{\text{F}}$ is the infinite closure of $\to^{\text{F}}$, it also holds that $\rho \vdash e \overset{\text{S}\infty}{\Longrightarrow} \implies \rho \vdash e \overset{\infty}{\to}$.

Lemma A.10 ($\overset{\text{S}\infty}{\Longrightarrow}$ implies $\overset{\infty}{\to}$) It holds that:

$$\rho \vdash e \overset{\text{S}\infty}{\Longrightarrow} \implies \rho \vdash \mathsf{e2f}(e) \overset{\infty}{\to}^{\text{F}}$$

Proof. The proof is by coinduction, using the goal as coinduction hypothesis. The proof follows by (TransInf) and Lemma A.11 below. □

Lemma A.11 ($\overset{\infty}{\Rightarrow}$ can be broken up into smaller steps) It holds that:

$$\mathsf{f2r}(\rho) \vdash \mathsf{f2e}(e) \overset{\mathrm{S}\infty}{\Longrightarrow} \implies \exists e'.\ \rho \vdash e \rightarrow^{\mathrm{F}} e' \ \land\ \mathsf{f2r}(\rho) \vdash \mathsf{f2e}(e') \overset{\mathrm{S}\infty}{\Longrightarrow}$$

Proof. The proof is by structural induction on e and inversion on $\overset{\mathrm{S}\infty}{\Longrightarrow}$. Even though the proof is by induction on e, we consider cases for $\overset{\mathrm{S}\infty}{\Longrightarrow}$, since we do inversion on this relation. The cases follow a similar structure. We consider the cases for (SNS-λ_{cbv}-∞-Plus1), (SNS-λ_{cbv}-∞-Plus2), and (SNS-λ_{cbv}-∞-App3). The remaining cases follow by similar reasoning.

Case (SNS-λ_{cbv}-∞-Plus1) The induction hypothesis gives us:

$$\exists e_1'.\ \rho \vdash e_1 \rightarrow e_1' \ \land\ \mathsf{f2r}(\rho) \vdash \mathsf{f2e}(e_1') \overset{\mathrm{S}\infty}{\Longrightarrow} \tag{IH1}$$

The goal is:

$$\exists e'.\ \rho \vdash \texttt{plus}(e_1, e_2) \rightarrow e' \ \land\ \mathsf{f2r}(\rho) \vdash \mathsf{f2e}(e') \overset{\mathrm{S}\infty}{\Longrightarrow} \tag{Goal}$$

The goal follows by instantiating e' as $\texttt{plus}(e_1', e_2)$, and by (IH1), rule (SOS-λ_{cbv}-Plus1), the definition of f2e, and rule (SNS-λ_{cbv}-∞-Plus1).

Case (SNS-λ_{cbv}-∞-Plus2) From the rule (SNS-λ_{cbv}-∞-Plus2) and the definition of f2e we have:

$$\mathsf{f2r}(\rho) \vdash \mathsf{f2e}(e_1) \overset{\mathrm{S}}{\Rightarrow} n_1 \tag{H1}$$

$$\mathsf{f2r}(\rho) \vdash \mathsf{f2e}(e_2) \overset{\mathrm{S}\infty}{\Longrightarrow} \tag{H2}$$

The induction hypothesis gives us:

$$\exists e_2'.\ \rho \vdash e_2 \rightarrow e_2' \ \land\ \mathsf{f2r}(\rho) \vdash \mathsf{f2e}(e_2') \overset{\mathrm{S}\infty}{\Longrightarrow} \tag{IH2}$$

The goal is:

$$\exists e'.\ \rho \vdash \texttt{plus}(e_1, e_2) \rightarrow e' \ \land\ \mathsf{f2r}(\rho) \vdash \mathsf{f2e}(e') \overset{\mathrm{S}\infty}{\Longrightarrow} \tag{Goal}$$

From Lemma A.6, (H1), and the correspondence between $\rightarrow$ and $\rightarrow^{\mathrm{F}}$, it follows that:

$$\rho \vdash e_1 \rightarrow^{\mathrm{F}^*} n_1 \tag{H3}$$

By inversion on (H3), it either holds that:

- e_1 has a transition step, in which case the goal follows from this fact, Lemma A.8, and (H2);

- $e_1 = n_1$, in which case the goal follows from (IH2).

Case (SNS-λ_{cbv}-∞-App3) Since we are doing structural induction on $e \in Expr^{\text{F}}$, there are really two different cases to consider for (SNS-λ_{cbv}-∞-App3): either $e = \mathsf{app}(e_1, e_2)$ for some e_1, e_2, or $e = \mathtt{force}(x, v, \rho, e')$ for some x, v, ρ, e'.

Subcase ($e = \mathsf{app}(e_1, e_2)$) From the rule (SNS-$\lambda_{\text{cbv}}$-$\infty$-App3) we have:

$$\mathsf{f2r}(\rho) \vdash \mathsf{f2e}(e_1) \overset{\text{S}}{\Rightarrow} \langle x, e, \rho' \rangle \tag{H1}$$

$$\mathsf{f2r}(\rho) \vdash \mathsf{f2e}(e_2) \overset{\text{S}}{\Rightarrow} v_2 \tag{H2}$$

$$\rho'[x \mapsto v_2] \vdash e \overset{\text{S}\infty}{\Longrightarrow} \tag{H3}$$

The induction hypothesis gives us:

$$\exists e_1'.\ \rho \vdash e_1 \to e_1' \ \wedge\ \mathsf{f2r}(\rho) \vdash \mathsf{f2e}(e_1') \overset{\text{S}\infty}{\Longrightarrow} \tag{IH1}$$

$$\exists e_2'.\ \rho \vdash e_2 \to e_2' \ \wedge\ \mathsf{f2r}(\rho) \vdash \mathsf{f2e}(e_2') \overset{\text{S}\infty}{\Longrightarrow} \tag{IH2}$$

The goal is:

$$\exists e'.\ \rho \vdash \mathsf{app}(e_1, e_2) \to e' \ \wedge\ \mathsf{f2r}(\rho) \vdash \mathsf{f2e}(e') \overset{\text{S}\infty}{\Longrightarrow} \tag{Goal}$$

From Lemma A.6, (H1), and the correspondence between $\to$ and $\to^{\text{F}}$, it follows that:

$$\rho \vdash e_1 \to^{\text{F}^*} \langle x, e, \rho' \rangle \tag{H4}$$

By inversion on (H4), it either holds that e_1 has a transition step, in which case the goal follows from this fact, Lemma A.8, (H2), and (H3); or $\mathsf{f2e}(e_1) = \langle x, e, \rho' \rangle$. In the latter case, we proceed by similar reasoning on (H2): from Lemma A.6, (H2), and the correspondence between $\to$ and $\to^{\text{F}}$, it follows that:

$$\rho \vdash e_2 \to^{\text{F}^*} v_2 \tag{H5}$$

By inversion on (H5), it either holds that e_2 has a transition step, in which case the goal follows from (SNS-λ_{cbv}-Refl), this fact, Lemma A.8, and (H3); or $\mathsf{f2e}(e_2) = v_2$. In the latter case, the goal follows from (SOS-λ_{cbv}-AppF), the definition of f2e, (SNS-λ_{cbv}-∞-App3), and (H3).

Subcase ($e = \mathtt{force}(x, \rho', v, e')$) From the rule (SNS-$\lambda_{\text{cbv}}$-$\infty$-App3) we have:

$$\mathsf{f2r}(\rho) \vdash \mathsf{f2e}(e_1) \overset{\text{S}}{\Rightarrow} \langle x, e, \rho' \rangle \tag{H1}$$

$$\mathsf{f2r}(\rho) \vdash \mathsf{f2e}(e_2) \overset{\text{S}}{\Rightarrow} v_2 \tag{H2}$$

$$\rho'[x \mapsto v_2] \vdash e \overset{\text{S}\infty}{\Longrightarrow} \tag{H3}$$

Since f2e is surjective (Lemma A.12), the following hypotheses follow from (H1-H3):

$$\mathsf{f2r}(\rho) \vdash \mathsf{f2e}(e_1) \overset{\mathrm{S}}{\Rightarrow} \mathsf{f2v}(\langle x, e, \rho' \rangle) \tag{H1$'$}$$

$$\mathsf{f2r}(\rho) \vdash \mathsf{f2e}(e_2) \overset{\mathrm{S}}{\Rightarrow} \mathsf{f2v}(v_2) \tag{H2$'$}$$

$$\mathsf{f2r}(\rho'[x \mapsto v_2]) \vdash \mathsf{f2e}(e) \overset{\mathrm{S}\infty}{\Longrightarrow} \tag{H3$'$}$$

From the induction hypothesis we get:

$$\forall \rho_0.\ \exists e''.\ \rho_0 \vdash e' \to e'' \ \wedge\ \mathsf{f2r}(\rho_0) \vdash \mathsf{f2e}(e'') \overset{\mathrm{S}\infty}{\Longrightarrow} \tag{IH}$$

From this it follows that:

$$\rho'[x \mapsto v] \vdash e' \to e'' \ \wedge\ \mathsf{f2r}(\rho'[x \mapsto v]) \vdash \mathsf{f2e}(e'') \overset{\mathrm{S}\infty}{\Longrightarrow} \tag{IH$'$}$$

The goal is:

$$\exists e''.\rho \vdash \mathtt{force}(x, \rho', v, e') \to e'' \wedge \mathsf{f2r}(rho) \vdash \mathsf{f2e}(e'') \overset{\mathrm{S}\infty}{\Longrightarrow} \tag{A.1}$$

The goal follows by instantiating e'' to $\mathtt{force}(x, \rho', v, e')$, (IH$'$), the definition of f2e, (H1$'$), and (H2$'$).

□

Lemma A.12 f2e is a surjective function from $\mathit{Expr}^{\mathrm{F}}$ to Expr.

Proof. Trivial, by definition of f2e. □

Lemma A.13 ($\overset{\infty}{\to}$ implies $\overset{\mathrm{S}\infty}{\Longrightarrow}$) It holds that:

$$\rho \vdash e \overset{\infty}{\to} \ \Longrightarrow\ \rho \vdash e \overset{\mathrm{S}\infty}{\Longrightarrow}$$

Proof (classical). The proof is by guarded coinduction, using the goal as coinduction hypothesis. We reason by case analysis on e. Each case follows by similar reasoning steps. We show these steps for `plus`. The rest of the cases either follow trivially (by contradiction, e.g., for simple rules that obviously converge), or follow a similar pattern.

Case ($\mathtt{plus}(e_1, e_2)$) We have the following hypothesis and goal:

$$\rho \vdash \mathtt{plus}(e_1, e_2) \overset{\infty}{\to} \tag{H1}$$

$$\rho \vdash \mathtt{plus}(e_1, e_2) \overset{\mathrm{S}\infty}{\Longrightarrow} \tag{Goal}$$

We reason by case analysis on whether e_1 diverges or not using Lemma A.14 below.

Subcase (Convergence) From the case analysis on Lemma A.14, we get:

$$\rho \vdash e_1 \rightarrow^* e_1' \tag{H2}$$

$$\rho \vdash e_1' \not\rightarrow \tag{H3}$$

By applying Lemma A.15 on (H1) and (H2), we get:

$$\rho \vdash \texttt{plus}(e_1', e_2) \overset{\infty}{\rightarrow} \tag{H4}$$

It must be the case that e_1' is some natural number n_1, or (H4) would be a contradiction. We have:

$$\rho \vdash \texttt{plus}(n_1, e_2) \overset{\infty}{\rightarrow} \tag{H4$'$}$$

The goal follows by applying (SNS-λ_{cbv}-∞-Plus1), Lemma A.8, (H2), the coinduction hypothesis, and (H4).

□

Lemma A.14 (Small-step either converges or diverges) It holds that:

$$\left(\exists e'.\, \rho \vdash e \rightarrow^* e' \;\wedge\; R \vdash e'_{/S'} \not\rightarrow\right) \;\vee\; R \vdash e_{/S} \overset{\infty}{\rightarrow}$$

Proof (classical). The proof is analogous to the one given by Leroy and Grall [LG09, Lemma 10]: we first show $(\forall e'.\, \rho \vdash e \rightarrow^* e' \implies \exists e''.\, \rho \vdash e' \rightarrow e'') \implies \rho \vdash e \overset{\infty}{\rightarrow}$ by guarded coinduction. The goal follows by reasoning by the law of excluded middle on $\overset{\infty}{\rightarrow}$ and this fact. An analogous proof is provided in Chapter 5 of this thesis (Lemma 5.2). □

Lemma A.15 (Congruence of $\overset{\infty}{\rightarrow}$) It holds that:

(a) $\rho \vdash e_1 \rightarrow^* e_1' \implies \rho \vdash \texttt{plus}(e_1, e_2) \overset{\infty}{\rightarrow} \implies \texttt{plus}(e_1', e_2) \overset{\infty}{\rightarrow}$

(b) $\rho \vdash e_2 \rightarrow^* e_2' \implies \rho \vdash \texttt{plus}(n_1, e_2) \overset{\infty}{\rightarrow} \implies \texttt{plus}(n_1, e_2') \overset{\infty}{\rightarrow}$

(c) $\rho \vdash e_1 \rightarrow^* e_1' \implies \rho \vdash e_1\, e_2 \overset{\infty}{\rightarrow} \implies \rho \vdash e_1' \overset{\infty}{\rightarrow}$

(d) $\rho \vdash e_2 \rightarrow^* e_2' \implies \rho \vdash \langle x, e, \rho \rangle\, e_2 \implies \rho \vdash \langle x, e, \rho \rangle\, e_2' \overset{\infty}{\rightarrow}$

(e) $\rho'[x \mapsto v_2] \vdash e \rightarrow^* e' \implies \rho \vdash \langle x, e, \rho' \rangle\, v_2 \implies \rho \vdash \langle x, e', \rho' \rangle\, v_2$

Proof. The proofs have a similar structure. We show the proof for property (a) and (e); the rest are analogous.

Case (Proof of (a))

Subcase (Refl) Trivial.

Subcase (Trans) From (Trans) and the goal, we have:

$$\rho \vdash e_1 \to e_1' \tag{H1}$$

$$\rho \vdash \texttt{plus}(e_1, e_2) \overset{\infty}{\to} \tag{H2}$$

From the induction hypothesis we have:

$$\rho \vdash \texttt{plus}(e_1', e_2) \overset{\infty}{\to} \implies \rho \vdash \texttt{plus}(e_1'' \overset{\infty}{\to} \tag{IH}$$

The goal is:

$$\rho \vdash \texttt{plus}(e_1'', e_2) \overset{\infty}{\to} \tag{Goal}$$

By inversion on $\overset{\infty}{\to}$ and subsequently $\to$ in (H2) we get:

$$\rho \vdash e_1 \to e_1''' \tag{H2$'$}$$

$$\rho \vdash \texttt{plus}(e_1''', e_2) \overset{\infty}{\to} \tag{H2$''$}$$

Since $\to$ is deterministic, we get:

$$\rho \vdash e_1 \to e_1' \tag{H2$'''$}$$

$$\rho \vdash \texttt{plus}(e_1', e_2) \overset{\infty}{\to} \tag{H2$''''$}$$

The goal follows from the induction hypothesis and (H2$''''$).

Case (Proof of (e)) The proof is by rule induction on $\to^*$.

Subcase (Refl) Trivial.

Subcase (Trans) From (Trans) and the goal, we have:

$$\rho'[x \mapsto v_2] \vdash e \to e' \tag{H1}$$

$$\rho \vdash \langle x, e, \rho' \rangle\, v_2 \overset{\infty}{\to} \tag{H2}$$

From the induction hypothesis we have:

$$\rho \vdash \langle x, e', \rho' \rangle\, v_2 \overset{\infty}{\to} \implies \rho \vdash \langle x, e'', \rho' \rangle\, v_2 \overset{\infty}{\to} \tag{IH}$$

The goal is:

$$\rho \vdash \langle x, e'', \rho' \rangle\, v_2 \overset{\infty}{\to} \tag{Goal}$$

By inversion on $\overset{\infty}{\to}$ and subsequently $\to$ in (H2) we get:

$$\rho'[x \mapsto v_2] \vdash e \to e''' \tag{H2$'$}$$

$$\rho'[x \mapsto v_2] \vdash e''' \overset{\infty}{\to} \tag{H2$''$}$$

Since $\to$ is deterministic, we get:

$$\rho'[x \mapsto v_2] \vdash e \to e' \tag{H2$'''$}$$

$$\rho'[x \mapsto v_2] \vdash e' \overset{\infty}{\to} \tag{H2$''''$}$$

The goal follows from the induction hypothesis and (H2$''''$).

□

Lemma A.16 (Determinism of $\rightarrow$) It holds that:

$$\rho \vdash e \rightarrow e' \implies \rho \vdash e \rightarrow e'' \implies e' = e''$$

Proof. Straightforward proof by rule induction on $\rightarrow$ for the first hypothesis. □

Now, the proof of the main property of interest follows:

Proposition 2.7 ($\overset{\infty}{\Rightarrow}$ corresponds to $\overset{\infty}{\rightarrow}$ for λ_{cbv}) The natural semantics and SOS for λ_{cbv} describe the same set of diverging terms; i.e.:

$$\mathsf{source}(e) \implies \mathsf{rsource}(\rho) \implies \left(\mathsf{rnsify}(\rho) \vdash \mathsf{nsify}(e) \overset{\infty}{\Rightarrow} \iff \rho \vdash e \overset{\infty}{\rightarrow}\right)$$

where $\mathsf{source}(e), \mathsf{rsource}(\rho)$ checks that expressions $e \in \mathit{Expr}$ and $\rho \in \mathit{Env}$ have corresponding natural semantic source terms and environments; and $\mathsf{nsify} \in \mathit{Expr} \rightarrow \mathit{Expr}^{\text{NS}}$ and $\mathsf{rnsify} \in \mathit{Env} \rightarrow \mathit{Env}^{\text{NS}}$ translates SOS syntax to natural semantics syntax.

Proof. Each direction of the proof follows from Lemma A.5, and Lemma A.10 and A.13 above. □

B Proofs for Refocusing in XSOS

This appendix contains proofs that are not in the Coq development accompanying this thesis. Section B.1 of this appendix contains proofs for relating finite computations for small-step and big-step XSOS (Chapter 4), while Section B.2 contains proofs for relating infinite computations for small-step and big-step XSOS (Chapter 5).

B.1 Proofs for Chapter 4

In this section we provide proofs for Chapter 4:

- Section B.1.1 contains proofs of soundness and completeness of refocusing for rules with left-to-right order of evaluation for XSOS without modular abrupt termination (Definitions 4.1 and 4.2).

- Section B.1.2 contains proofs of soundness and completeness of refocused rules with modular abrupt termination (Definitions 4.10 on page 113 and 4.11 on page 114).

B.1.1 Correctness of refocusing

The lemmas in this section are the basis for the first main result in Chapter 4, namely Theorem 4.4 on page 109.

Lemma 4.5 (Refocusing is sound) For any transition relation $\rightarrow$ that implements left-to-right order of evaluation and whose refocused counterpart is a relation $\Downarrow$, it holds that:

$$R \vdash e_{/S} \Downarrow e'_{/S'} \implies R \vdash e_{/S} \rightarrow^* e'_{/S'}$$

Proof. By rule induction on $\Downarrow$. There are three kinds of rules to consider: rules with premises (PBXRS-fi); rules with a single premise (PBXRS-$f0$); simple rules (PBXRS-f); and the reflexive rule for final configurations (XSOS-PB-Iter-Refl).

Case (XRS-fi) The induction hypothesis gives us:

$$R' \vdash e_{/S} \to^* e'_{/S'} \tag{IH1}$$

$$R \vdash f(v_1, \ldots, v_n, e', \ldots)_{/S'} \to^* e''_{/S''} \tag{IH2}$$

The goal is:

$$R \vdash f(v_1, \ldots, v_n, e, \ldots)_{/S'} \to^* e''_{/S''} \tag{Goal}$$

Applying Lemma 4.6 (see below) to (IH1) gives:

$$R \vdash f(v_1, \ldots, v_n, e, \ldots)_{/S} \to^* f(v_1, \ldots, v_n, e', \ldots)_{/S'} \tag{IH1$'$}$$

The goal follows by transitivity of $\to^*$.

Case (PBXRS-$f0$) From (PBXRS-$f0$), we have the hypothesis:

$$\neg\mathrm{Q}(f(v_1, \ldots, v_n, e, \ldots), S) \tag{H1}$$

The induction hypothesis gives us:

$$R \vdash e_{0/S_0} \to^* e'_{/S'} \tag{IH}$$

The goal is:

$$R \vdash f(v_1, \ldots, v_n, \ldots)_{/S} \to^* e'_{/S'} \tag{Goal}$$

Since $\Downarrow$ is the refocused counterpart of $\to$ we know that there is exactly one simple small-step rule corresponding to the single-premise rule (PBXRS-$f0$). Such a small-step rule must have the form:

$$\frac{\neg\mathrm{Q}(f(v_1, \ldots, v_n, \ldots), S)}{R \vdash f(v_1, \ldots, v_n, \ldots)_{/S} \to e_{0/S_0}}$$

The goal follows by application of (Trans), the small-step rule, (H1), and (IH).

Case (PBXRS-f) The case is analogous to the case for (PBXRS-$f0$).

Case (XSOS-PB-Iter-Refl) The goal is immediate.

□

Lemma 4.6 (Congruence of reflexive-transitive closure) For any transition relation $\to$ that implements left-to-right order of evaluation, and where $\to$ has a rule:

$$\frac{R' \vdash e_{/S} \to e'_{/S'}}{R \vdash f(v_1, \ldots, v_n, e, \ldots)_{/S} \to f(v_1, \ldots, v_n, e', \ldots)_{/S'}}$$

it holds that:

$$R' \vdash e_{/S} \to^* e'_{/S'} \implies$$
$$R \vdash f(v_1, \ldots, v_n, e, \ldots)_{/S} \to^* f(v_1, \ldots, v_n, e', \ldots)_{/S'}$$

Proof. By rule induction on $\rightarrow^*$.

Case (Trans) From (Trans) we have the hypothesis:

$$R' \vdash e_{/S} \rightarrow e_{0/S_0} \tag{H2}$$

From the induction hypothesis we have:

$$R \vdash f(v_1,\ldots,v_n,e_0,\ldots)_{/S_0} \rightarrow^* f(v_1,\ldots,v_n,e',\ldots)_{/S'} \tag{IH}$$

The goal is:

$$R \vdash f(v_1,\ldots,v_n,e,\ldots)_{/S} \rightarrow^* f(v_1,\ldots,v_n,e',\ldots)_{/S'} \tag{Goal}$$

We know that there is a small-step rule of the form:

$$\frac{\begin{array}{c}\neg\mathrm{Q}(e,S)\\ R' \vdash e_{/S} \rightarrow e_{0/S_0}\end{array}}{R \vdash f(v_1,\ldots,v_n,e,\ldots)_{/S} \rightarrow f(v_1,\ldots,v_n,e_0,\ldots)_{/S_0}}$$

If $\mathrm{Q}(e,S)$ then, by inversion on (IH), we get $(e,S) = (e_0,S_0)$ and the goal follows. If $\neg\mathrm{Q}(e,S)$ then the goal follows by (Trans) and the induction hypothesis.

Case (Refl) The goal is immediate.

□

Lemma 4.7 (Refocusing is complete) For any transition relation $\rightarrow$ that implements left-to-right order of evaluation and whose refocused counterpart is a relation $\Downarrow$, it holds that:

$$\begin{array}{l} R \vdash e_{/S} \rightarrow^* e'_{/S'} \implies \\ R \vdash e_{/S} \Downarrow e'_{/S'} \end{array}$$

Proof. By rule induction on $\rightarrow^*$. The central argument of the proof is Lemma 4.8 (see below).

Case (Trans) From (Trans) we have the hypothesis:

$$R \vdash e_{/S} \rightarrow e_{0/S_0} \tag{H1}$$

From the induction hypothesis we have:

$$R \vdash e_{0/S_0} \Downarrow e'_{/S'} \tag{IH}$$

The goal follows by application of Lemma 4.8.

Case (Refl) The goal is immediate.

□

Lemma 4.8 (Pretty-big-steps can be broken up into small-steps) For any transition relation $\rightarrow$ that implements left-to-right order of evaluation and whose refocused counterpart is a relation $\Downarrow$, it holds that:

$$\neg\mathrm{Q}(e,S) \implies R \vdash e_{/S} \rightarrow e'_{/S'} \implies R \vdash e'_{/S'} \Downarrow e''_{/S''} \implies \\ R \vdash e_{/S} \Downarrow e''_{/S''}$$

Proof. By rule induction on $\rightarrow$.

Case (XRS-fi) From (XRS-fi) and the goal we have the hypotheses:

$$\neg\mathrm{Q}(e_0,S) \tag{H1}$$

$$R' \vdash e_{0/S} \rightarrow e'_{0/S_0} \tag{H2}$$

$$R \vdash f(v_1,\ldots,v_n,e'_0,\ldots)_{/S_0} \Downarrow e''_{/S''} \tag{H3}$$

From the induction hypothesis we have:

$$\forall e_1\ S_1.\ R' \vdash e'_{0/S_0} \Downarrow e_{1/S_1} \implies R' \vdash e_{0/S} \Downarrow e_{1/S_1} \tag{IH}$$

The goal is:

$$R \vdash f(v_1,\ldots,v_n,e_0,\ldots)_{/S} \Downarrow e''_{/S''}$$

We reason by inversion on (H3). For the premise it either holds that: a rule with premises matches; a rule with a single premise matches; a simple rule matches; or the configuration is abruptly terminated, i.e., $\mathrm{Q}(f(v_1,\ldots,v_n,e_0,\ldots),S)$.

Subcase (PBXRS-fi) It is either the case that we are evaluating the sub-term e'_0, or we are evaluating the next sub-term in the sequence.

In the latter case, e'_0 must be some value v_0, and (H2) must therefore be of the form:

$$R' \vdash e_{0/S} \rightarrow v_{0/S_0} \tag{H2}$$

The goal follows from applying Lemma 4.9 (see below) to (H2), and by application of the pretty-big-step counterpart to (XRS-fi), i.e., (PBXRS-fi).

In the former case, we get as hypotheses from inversion:

$$\neg\mathrm{Q}(e'_0,S_0) \tag{H4}$$

$$R' \vdash e'_{0/S_0} \Downarrow e'_{1/S_1} \tag{H5}$$

$$R \vdash f(v_1,\ldots,v_n,e'_1,\ldots)_{/S_1} \Downarrow e''_{/S''} \tag{H6}$$

By applying (IH) to (H5), the goal now follows by application of the pretty-big-step counterpart to (XRS-fi), i.e., (PBXRS-fi).

Subcase (PBXRS-$f0$) The sub-term e'_0 must be a value, or the rule (PBXRS-f) would not match. Thus (H2) must be of the form:

$$R' \vdash e_{0/S} \rightarrow v_{0/S_0} \tag{H2}$$

The goal follows from applying Lemma 4.9 (see below) to (H2), and by application of the pretty-big-step counterpart to (XRS-fi), i.e., (PBXRS-fi).

Subcase (PBXRS-f) Analogous to (PBXRS-$f0$).

Subcase (XSOS-PB-Iter-Refl) The configuration e'_{0/S_0} must be abruptly terminated. The goal follows by applying Lemma 4.9 to (H2), and by application of the pretty-big-step counterpart to (XRS-fi), i.e., (PBXRS-fi).

Case (XRS-f) From the goal we have the hypothesis:

$$R \vdash e'_{/S'} \Downarrow e''_{/S''} \tag{H1}$$

From the induction hypothesis we have:

$$\forall e_1\ S_1.\ R' \vdash e'_{/S'} \Downarrow e_{1/S_1} \implies R' \vdash e_{/S} \Downarrow e_{1/S_1} \tag{IH}$$

The goal is:

$$R \vdash f(v_1, \ldots, v_n, e, \ldots)_{/S} \Downarrow e''_{/S''} \tag{Goal}$$

The goal follows by application of the pretty-big-step counterpart to (XRS-f), i.e., either (PBXRS-$f0$) or (PBXRS-f), (IH), and (H1).

□

B.1.2 Correctness of refocusing with abrupt termination

The lemmas in this section are the basis for the second main result in Chapter 4, namely Theorem 4.12 on page 115.

Lemma 4.14 (Congruence of reflexive-transitive closure with abrupt termination) For any transition relation $\to$ that implements left-to-right order of evaluation with abrupt termination, and where $\to$ has a pair of rules:

$$\frac{\begin{array}{c}\neg\mathrm{Q}(e_i, S) \qquad (e'_i, S') \in X \\ R' \vdash e_{i/S} \to e'_{i/S'}\end{array}}{R \vdash f(v_1, \ldots, v_{i-1}, e_i, e_{i+1}, \ldots, e_n, \ldots)_{/S} \to e''_{/S''}}$$

$$\frac{\begin{array}{c}\neg\mathrm{Q}(e_i, S) \qquad (e'_i, S') \notin X \\ R' \vdash e_{i/S} \to e'_{i/S'}\end{array}}{\begin{array}{l}R \vdash f(v_1, \ldots, v_{i-1}, e_i, e_{i+1}, \ldots, e_n, \ldots)_{/S} \to \\ \qquad f(v_1, \ldots, v_{i-1}, e'_i, e_{i+1}, \ldots, e_n, \ldots)_{/S'}\end{array}}$$

where X is a set of final configurations. For each such pair of rules, it holds that:

$$\begin{array}{l}R' \vdash e_{i/S} \to^* e'_{i/S'} \implies (e'_i, S') \in X \implies \\ R \vdash f(v_1, \ldots, v_{i-1}, e_i, e_{i+1}, \ldots, e_n, \ldots)_{/S} \to^* e''_{/S''}\end{array} \tag{1}$$

and:

$$R' \vdash e_{i/S} \rightarrow^* e'_{i/S'} \implies (e'_i, S') \notin X \implies \\ R \vdash f(v_1, \ldots, v_{i-1}, e_i, e_{i+1}, \ldots, e_n, \ldots)_{/S} \rightarrow^* f(v_1, \ldots, v_{i-1}, e'_i, e_{i+1}, \ldots, e_n, \ldots)_{/S'} \tag{2}$$

Proof. By rule induction on $\rightarrow^*$.

Case (Trans) From (Trans) and the goal we have the hypotheses:

$$\neg \mathrm{Q}(e, S) \tag{H1}$$

$$R' \vdash e_{i/S} \rightarrow e_{0/S_0} \tag{H2}$$

From the induction hypothesis we have:

$$R \vdash f(v_1, \ldots, v_{i-1}, e_0, e_{i+1}, \ldots, e_n, \ldots)_{/S_0} \rightarrow^* e''_{S''} \tag{IH}$$

The goal is:

$$R \vdash f(v_1, \ldots, v_{i-1}, e_i, e_{i+1}, \ldots, e_n, \ldots)_{/S} \rightarrow^* e''_{/S''} \tag{Goal}$$

It is either the case that $(e_0, S_0) \in X$, or that it is not.

Subcase ($(e_0, S_0) \in X$) By assumption, there is a rule:

$$\frac{\neg \mathrm{Q}(e, S) \qquad (e_0, S_0) \in X \\ R' \vdash e_{i/S} \rightarrow e_{0/S_0}}{R \vdash f(v_1, \ldots, v_{i-1}, e_i, e_{i+1}, \ldots, e_n, \ldots)_{/S} \rightarrow e''_{/S''}}$$

The goal follows by application of (Trans), this rule, (H1), (H2), and (Refl).

Subcase ($(e_0, S_0) \notin X$) By assumption, there is a rule:

$$\frac{\neg \mathrm{Q}(e, S) \qquad (e_0, S_0) \notin X \\ R' \vdash e_{i/S} \rightarrow e_{0/S_0}}{R \vdash f(v_1, \ldots, v_{i-1}, e_i, e_{i+1}, \ldots, e_n, \ldots)_{/S} \rightarrow \\ f(v_1, \ldots, v_{i-1}, e_0, e_{i+1}, \ldots, e_n, \ldots)_{/S_0}}$$

The goal follows by application of (Trans), this rule, (H1), (H2), and (IH).

Case (Refl) The goal is immediate.

□

Lemma 4.16 (Pretty-big-steps can be broken up into small-steps with abrupt termination) For any transition relation $\rightarrow$ that implements left-to-right order of evaluation with abrupt termination and whose refocused counterpart is a relation $\Downarrow$, it holds that:

$$R \vdash e_{/S} \rightarrow e'_{/S'} \implies R \vdash e'_{/S'} \Downarrow e''_{/S''} \implies \\ R \vdash e_{/S} \Downarrow e''_{/S''}$$

Proof. The only existing case of the induction proof in Lemma 4.8 that needs extending is (XRS-fi): the reasoning by inversion on (H3) must be extended with two new subcases for rules (PBXRS-AT-fi) and (PBXRS-OK-fi). Both of these subcases follow by the same line of reasoning as subcase (PBXRS-fi) in Lemma 4.8. Here, we give the two new cases for the small-step rule induction.

Case (XRS-AT-fi) From (XRS-AT-fi) and the goal we have the hypotheses:

$$\neg\mathrm{Q}(e_0, S) \tag{H1}$$

$$R' \vdash {e_0}_{/S} \rightarrow {e'_0}_{/S_0} \tag{H2}$$

$$(e'_0, S_0) \in X \tag{H3}$$

$$R \vdash e'_{/S'} \Downarrow e''_{/S''} \tag{H4}$$

From the induction hypothesis we have:

$$\forall e_1\ S_1.\ R' \vdash {e'_0}_{/S_0} \Downarrow {e_1}_{/S_1} \implies R' \vdash {e_0}_{/S} \Downarrow {e_1}_{/S_1} \tag{IH}$$

The goal is:

$$R \vdash f(v_1, \ldots, v_n, e_0, \ldots)_{/S} \Downarrow e''_{/S''}$$

Since (e'_0, S_0) is a final configuration, the goal follows by applying Lemma 4.17 in (H2), application of the pretty-big-step counterpart to (XRS-AT-fi), i.e., (PBXRS-AT-fi).

Case (XRS-OK-fi) This case is analogous to the extended version of (XRS-fi) described above.

□

Lemma 4.17 (Correspondence between terminating small-steps and refocusing with abrupt termination) For any transition relation $\rightarrow$ that implements left-to-right order of evaluation with abrupt termination and whose refocused counterpart is $\Downarrow$, it holds that:

$$\begin{aligned}&\neg\mathrm{Q}(e, S) \implies R \vdash e_{/S} \rightarrow e'_{/S'} \implies \mathrm{Q}(e', S') \implies \\ &R \vdash e_{/S} \Downarrow e'_{/S'}\end{aligned}$$

Proof. By induction on $\rightarrow$. The case for simple rules (XRS-f) remains unchanged. It only remains to consider the case for instances of (XRS-AT-fi). We get as hypotheses:

$$\neg\mathrm{Q}(e, S) \tag{H1}$$

$$R' \vdash e_{/S} \rightarrow {e_0}_{/S_0} \tag{H2}$$

$$(e_0, S_0) \in X \tag{H3}$$

$$\mathrm{Q}(e', S') \tag{H4}$$

From the induction hypothesis we have:

$$\forall e_1\ S_1.\ \mathrm{Q}(e_1, S_1) \implies R' \vdash e_{/S} \Downarrow {e_1}_{/S_1} \tag{IH}$$

The goal is:

$$R \vdash f(v_1, \ldots, v_n, e, \ldots)_{/S} \Downarrow e'_{/S'} \tag{Goal}$$

Since X are final configurations, we know that:

$$\mathrm{Q}(e_0, S_0) \tag{H3$'$}$$

Applying (IH) to (H3$'$), it follows that:

$$R' \vdash e_{/S} \Downarrow e_{0/S_0} \tag{H3$''$}$$

The goal follows by application of a corresponding pretty-big-step rule of the form (PBXRS-AT-fi), (H1), (H3$''$), (H3), (XSOS-PB-Iter-Refl), and (H4). □

B.2 Proofs for Chapter 5

In this section we provide proofs for Chapter 5, proving the soundness and completeness of coinductive refocused pretty-big-step XSOS rules with modular abrupt termination and divergence (Definition 4.10 and 4.11, and Figure 5.1) relative to their small-step XSOS counterparts. The lemmas in this section are the basis for the main result in Chapter 5, namely Theorem 5.10 page 131.

Lemma 5.4 (Coevaluation without convergence implies divergence) For any relation $\Downarrow$ with left-to-right order of evaluation with abrupt termination, the following holds about its coinductive interpretation $\Downarrow^{\mathrm{co}}$:

$$\begin{aligned} &R \vdash e_{/S} \Downarrow^{\mathrm{co}} e'_{/S'} \implies \\ &\neg\left(R \vdash e_{/S} \Downarrow e'_{/S'}\right) \implies \\ &R \vdash e_{/S} \Downarrow^{\mathrm{co}} e'_{/S'[\mathbf{div}\,\uparrow]} \end{aligned}$$

Proof. The proof is by guarded rule coinduction on $\Downarrow^{\mathrm{co}}$ and inversion on the first hypothesis. We use the goal as coinduction hypothesis, i.e.:

$$\begin{aligned} &\forall R\ e\ S\ e'\ S'. \\ &\quad R \vdash e_{/S} \Downarrow^{\mathrm{co}} e'_{/S'} \implies \\ &\quad \neg\left(R \vdash e_{/S} \Downarrow e'_{/S'}\right) \implies \\ &\quad R \vdash e_{/S} \Downarrow^{\mathrm{co}} e'_{/S'[\mathbf{div}\,\uparrow]} \end{aligned} \tag{CIH}$$

We consider each of the cases resulting from inversion on the first hypothesis.

Case (PBXRS-fi) From inversion and the goal we have:

$$\neg \mathrm{Q}(e_i, S) \tag{H1}$$

$$R \vdash e_{1/S} \Downarrow^{\mathrm{co}} e'_{1/S'} \tag{H2}$$

$$R \vdash f(v_1, \ldots, v_{i-1}, e'_i, e_{i+1}, \ldots, e_n, \ldots)_{/S'} \Downarrow^{\mathrm{co}} e''_{/S''} \tag{H3}$$

$$\neg\left(R \vdash f(v_1, \ldots, v_{i-1}, e_i, e_{i+1}, \ldots, e_n, \ldots)_{/S} \Downarrow e''_{/S''}\right) \tag{H4}$$

The goal is:

$$R \vdash f(v_1,\ldots,v_{i-1},e_i,e_{i+1},\ldots,e_n,\ldots)_{/S} \Downarrow^{\text{co}} e''_{/S''[\mathbf{div}\uparrow]} \tag{Goal}$$

We reason by the law of excluded middle on:

$$R \vdash e_{i/S} \Downarrow e'_{i/S'} \tag{P}$$

Subcase (P) From the law of excluded middle we have:

$$R \vdash e_{i/S} \Downarrow e'_{i/S'} \tag{H5}$$

Reasoning by the law of excluded middle on the second premise, it either holds that:

$$R \vdash f(v_1,\ldots,v_{i-1},e'_i,e_{i+1},\ldots,e_n,\ldots)_{/S'} \Downarrow e''_{/S''}$$

or it does not. If it does, then (H4) is a contradiction. If it does not, then the goal follows from the available hypotheses and Lemma 5.3.

Subcase ($\neg P$) From the law of excluded middle we have:

$$\neg\left(R \vdash e_{i/S} \Downarrow e'_{i/S'}\right) \tag{H5}$$

The goal follows by applying the appropriate constructor for (PBXRS-fi) to the goal, (H1), (CIH), (H2), (H5), and (XSOS-PB-Iter-Refl).

Case (PBXRS-$f0$) From inversion and the goal we have:

$$\neg\mathrm{Q}(e',S') \tag{H1}$$

$$\neg\mathrm{Q}(f(v_1,\ldots,v_n),S) \tag{H2}$$

$$R \vdash e'_{/S'} \Downarrow^{\text{co}} e''_{/S''} \tag{H3}$$

$$\neg\left(R \vdash f(v_1,\ldots,v_n)_{/S} \Downarrow e''_{/S''}\right) \tag{H4}$$

The goal is:

$$R \vdash f(v_1,\ldots,v_n)_{/S} \Downarrow^{\text{co}} e''_{/S''} \tag{Goal}$$

We invoke the law of excluded middle on:

$$R \vdash e'_{/S'} \Downarrow e''_{/S''}$$

When the hypothesis is true, (H4) is a contradiction. When it is false, the goal follows from the available hypotheses and the coinduction hypothesis.

Case (PBXRS-f) From inversion and the goal we have:

$$\neg\mathrm{Q}(f(v_1,\ldots,v_n),S) \tag{H1}$$

$$\mathrm{Q}(e',S') \tag{H2}$$

From this, the goal is immediate:

$$R \vdash f(v_1,\ldots,v_n)_{/S} \Downarrow^{\mathrm{co}} e'_{/S'} \tag{Goal}$$

Case (PBXRS-AT-fi) The case is analogous to (PBXRS-fi) where, in addition to doing classical reasoning on $\Downarrow$, we do case analysis on whether the configuration resulting from evaluation is in X or not.

Case (PBXRS-OK-fi) The case is analogous to (PBXRS-fi) where, in addition to doing classical reasoning on $\Downarrow$, we do case analysis on whether the configuration resulting from evaluation is in X or not.

□

Lemma 5.7 (Small-step preserves coinductive refocusing) For any relation $\Downarrow$ that is the refocused counterpart to a small-step relation $\rightarrow$ with left-to-right order of evaluation with abrupt termination, the following holds about its coinductive interpretation $\Downarrow^{\mathrm{co}}$:

$$\begin{aligned}&R \vdash e_{/S[\mathbf{div}\downarrow]} \Downarrow^{\mathrm{co}} e'_{/S'[\mathbf{div}\uparrow]} \implies\\ &\exists e''\ S''.\ R \vdash e_{/S[\mathbf{div}\downarrow]} \rightarrow e''_{/S''} \ \wedge\ R \vdash e''_{/S''} \Downarrow^{\mathrm{co}} e'_{/S'[\mathbf{div}\uparrow]}\end{aligned}$$

Proof. The proof is by structural induction on the term e in the first All. hypothesis terms that do not match the structure of a rule can be disposed of by inversion on the first premise. Thus, in our proof we consider cases for *rules*, even through our proof is by induction on *terms*.

Case (PBXRS-fi) From inversion and the goal we have:

$$\neg\mathrm{Q}(e,S) \tag{H1}$$

$$R \vdash e_{i/S[\mathbf{div}\downarrow]} \Downarrow^{\mathrm{co}} e'_{i/S'} \tag{H2}$$

$$R \vdash f(v_1,\ldots,v_{i-1},e'_i,e_{i+1},\ldots,e_n,\ldots)_{/S'} \Downarrow^{\mathrm{co}} e''_{/S''[\mathbf{div}\uparrow]} \tag{H3}$$

From the induction hypothesis we get:

$$\begin{aligned}&\forall e'_i\ S'.\ R \vdash e_{i/S[\mathbf{div}\downarrow]} \Downarrow^{\mathrm{co}} e'_{i/S'[\mathbf{div}\uparrow]} \implies\\ &\quad \exists e_0\ S_0.\ R \vdash e_{i/S[\mathbf{div}\downarrow]} \rightarrow e_{0/S_0} \ \wedge\ R \vdash e_{0/S_0} \Downarrow^{\mathrm{co}} e'_{i/S'[\mathbf{div}\uparrow]}\end{aligned} \tag{IH}$$

The goal is:

$$\exists e_1\ S_1.\ R \vdash f(v_1,\ldots,v_{i-1},e_i,e_{i+1},\ldots,e_n,\ldots)_{/S[\mathbf{div}\downarrow]} \Downarrow^{\mathrm{co}} e_{1/S_1} \ \wedge\ R \vdash e_{1/S_1} \Downarrow^{\mathrm{co}} e''_{/S''[\mathbf{div}\uparrow]} \tag{Goal}$$

By applying Lemma 5.5 to (H2), we get:

$$R \vdash e_{i/S[\mathbf{div}\downarrow]} \Downarrow e'_{i/S'} \ \vee\ R \vdash e_{i/S[\mathbf{div}\downarrow]} \Downarrow^{co} e'_{i/S'[\mathbf{div}\uparrow]} \tag{H4}$$

We reason by case analysis on (H4).

Subcase ($R \vdash e_{i/S[\mathbf{div}\downarrow]} \Downarrow e'_{i/S'}$) By Theorem 4.12, we get:

$$R \vdash e_{i/S[\mathbf{div}\downarrow]} \to^* e'_{i/S'} \tag{H4$'$}$$

By inversion on this hypothesis, we get:

$$R \vdash e_{i/S[\mathbf{div}\downarrow]} \to e'_{0/S_0} \tag{H5}$$

$$R \vdash e'_{0/S_0} \to^* e'_{i/S'} \tag{H6}$$

By application of Theorem 4.12 and Lemma 5.3 in (H5), we get:

$$R \vdash e'_{0/S_0} \Downarrow^{co} e'_{i/S'} \tag{H6$'$}$$

The goal follows from (H1), (XRS-fi), (PBXRS-fi), (H6′), and (H3).

Subcase ($R \vdash e_{i/S[\mathbf{div}\downarrow]} \Downarrow^{co} e'_{i/S'}$) By the induction hypothesis, we get:

$$R \vdash e_{i/S[\mathbf{div}\downarrow]} \to e_{0/S_0} \tag{H4}$$

$$R \vdash e_{0/S_0} \Downarrow^{co} e'_{i/S'[\mathbf{div}\uparrow]} \tag{H5}$$

The goal follows from (H1), (H4), (XRS-fi), (PBXRS-fi), (H5), and (XSOS-PB-Iter-Refl).

Case (PBXRS-$f0$) The case is analogous to (PBXRS-fi).

Case (PBXRS-f) The case is a contradiction: no rules explicitly change the **div** flag. Thus, there is no rule instance matching (PXRS-f) where:

$$R \vdash e_{/S[\mathbf{div}\downarrow]} \Downarrow^{co} e'_{/S'[\mathbf{div}\uparrow]}$$

Case (PBXRS-AT-fi) The case is analogous to (PBXRS-fi).

Case (PBXRS-OK-fi) The case is analogous to (PBXRS-fi).

□

Lemma 5.8 (Coinductive refocusing is complete) For any relation $\Downarrow$ that is the refocused counterpart to a small-step relation $\to$ with left-to-right order of evaluation with abrupt termination, the following holds about its coinductive interpretation $\Downarrow^{co}$:

$$R \vdash e_{/S} \xrightarrow{\infty} \implies \forall e'\ S'.\ R \vdash e_{/S} \Downarrow^{co} e'_{/S'[\mathbf{div}\uparrow]}$$

Proof. The proof is by guarded coinduction, using the goal as coinduction hypothesis, and inversion on the first premise. From the goal and this inversion, we have:

$$R \vdash e_{/S} \to e'_{/S'} \tag{H1}$$

$$R \vdash e'_{/S'} \overset{\infty}{\to} \tag{H2}$$

We reason by inversion on (H1), considering each case in turn:

Case (XRS-fi) From the inversion we get the hypotheses:

$$\neg \mathrm{Q}(e_i, S) \tag{H3}$$

$$R' \vdash e_{i/S} \to e'_{i/S'} \tag{H4}$$

$$R \vdash f(v_1, \dots, v_n, e_i, e_{i+1}, \dots, e_n, \dots)_{/S} \overset{\infty}{\to} \tag{H5}$$

We invoke the law of excluded middle on:

$$\exists e''\ S''.\ R' \vdash e_{i/S} \to^* e''_{/S''} \ \wedge\ \mathrm{Q}(e'', S'')\ \wedge\ R \vdash f(v_1, \dots, v_n, e'', e_{i+1}, \dots, e_n, \dots)_{/S''} \overset{\infty}{\to} \tag{P}$$

Subcase (P) When P holds, we eliminate the existential and conjunctions to get the hypotheses:

$$R' \vdash e_{i/S} \to^* e''_{/S''} \tag{H6}$$

$$\mathrm{Q}(e'', S'') \tag{H7}$$

$$R \vdash f(v_1, \dots, v_n, e'', e_{i+1}, \dots, e_n, \dots)_{/S''} \overset{\infty}{\to} \tag{H8}$$

Applying Theorem 4.12 to (H6), using (H7), we get:

$$R' \vdash e_{i/S} \Downarrow e''_{/S''} \tag{H6$'$}$$

The goal follows by applying the corresponding rule (PBXRS-fi), (H3), Lemma 5.3, (H6$'$), the coinduction hypothesis, and (H8).

Subcase ($\neg P$) If P does not hold, we apply Lemma 5.9 to get the hypothesis:

$$R' \vdash e_{i/S} \overset{\infty}{\to} \tag{H6}$$

The goal follows by applying the corresponding rule (PBXRS-fi), (H3), the coinduction hypothesis, (H6), and (XSOS-PB-Iter-Refl).

Case (XRS-AT-fi) The case is analogous to (XRS-fi), using Lemma B.1 (below).

Case (XRS-OK-fi) The case is analogous to (XRS-fi), using Lemma B.1 (below).

Case (XRS-f) From the inversion we get the hypotheses:

$$\neg\mathrm{Q}(f(v_1,\ldots,v_n),S) \tag{H3}$$

$$R \vdash f(v_1,\ldots,v_n)_{/S} \rightarrow e'_{/S'} \tag{H4}$$

$$R \vdash e'_{/S'} \overset{\infty}{\rightarrow} \tag{H5}$$

There are two possibilities: either $e'_{/S'}$ is a final configuration, in which case (H5) is a contradiction; otherwise it holds that:

$$\neg\mathrm{Q}(e',S') \tag{H6}$$

and there is a corresponding single-premise rule whose structure matches (PBXRS-$f0$). The goal follows by applying this rule, (H3), (H6), the coinduction hypothesis, and (H5).

□

Lemma 5.9 (Congruence of infinite closure) For any small-step relation $\rightarrow$ with left-to-right order of evaluation with abrupt termination and with a rule that matches the scheme:

$$\forall e_i \ldots e_n\, S\, S'. \frac{\begin{array}{c}\neg\mathrm{Q}(e_i,S)\\ R' \vdash e_{i/S} \rightarrow e'_{i/S'}\end{array}}{\begin{array}{c}R \vdash f(v_1,\ldots,v_{i-1},e_i,e_{i+1},\ldots,e_n,\ldots)_{/S} \rightarrow\\ f(v_1,\ldots,v_{i-1},e'_i,e_{i+1},\ldots,e_n,\ldots)_{/S'}\end{array}} \tag{XRS-fi}$$

For each such rule, it holds that:

$$\begin{array}{l}R \vdash f(v_1,\ldots,v_n,e_i,e_{i+1},\ldots,e_n,\ldots)_{/S} \overset{\infty}{\rightarrow} \implies\\ \neg(\exists e'\, S'.\, R' \vdash e_{i/S} \rightarrow^* e'_{/S'} \wedge \mathrm{Q}(e',S') \wedge\\ \qquad R \vdash f(v_1,\ldots,v_n,e',e_{i+1},\ldots,e_n,\ldots)_{/S'} \overset{\infty}{\rightarrow}) \implies\\ R' \vdash e_{i/S} \overset{\infty}{\rightarrow}\end{array}$$

Proof. The proof is by guarded coinduction, using the goal as coinduction hypothesis, and inversion on the first premise. By inversion and from the goal, we get the hypotheses:

$$R \vdash f(v_1,\ldots,v_n,e_i,e_{i+1},\ldots,e_n,\ldots)_{/S} \rightarrow e_{0/S_0} \tag{H1}$$

$$R \vdash e_{0/S_0} \overset{\infty}{\rightarrow} \tag{H2}$$

$$\neg\left(\exists e'\, S'.\, R' \vdash e_{i/S} \rightarrow^* e'_{/S'} \wedge \mathrm{Q}(e',S') \wedge R \vdash f(v_1,\ldots,v_n,e',e_{i+1},\ldots,e_n,\ldots)_{/S'} \overset{\infty}{\rightarrow}\right) \tag{H3}$$

We reason by inversion on (H1). Due to the assumption that $\rightarrow$ implements left-to-right order of evaluation, we only need to consider the case in which the first premise is a rule of the form (XRS-*fi*), i.e., we get as result from inversion the hypotheses:

$$R' \vdash e_{i/S} \rightarrow e'_{i/S'_i} \tag{H4}$$

$$R \vdash f(v_1, \ldots, v_n, e'_i, e_{i+1}, \ldots, e_n, \ldots)_{/S'_i} \overset{\infty}{\rightarrow} \tag{H2$'$}$$

For all other cases, it holds that e_i is a value, whereby the second hypothesis in the goal becomes a contradiction.

A straightforward consequence of (H3) and (H4) is:

$$\neg \left(\exists e'\ S'.\ R' \vdash e'_{i/S'_i} \rightarrow^* e'_{/S'} \ \wedge\ \mathsf{Q}(e', S') \ \wedge\ R \vdash f(v_1, \ldots, v_n, e', e_{i+1}, \ldots, e_n, \ldots)_{/S'} \overset{\infty}{\rightarrow} \right) \tag{H5}$$

The goal follows by an application of the (XSOS-InfClo) rule, (H4), the coinduction hypothesis, (H2$'$), and (H45). □

Lemma B.1 (Congruence of infinite closure with abrupt termination) For any small-step relation $\rightarrow$ with left-to-right order of evaluation with abrupt termination and with a pair of rules that matches the scheme:

$$\forall e_i \ldots e_n\ S\ S'. \frac{\neg\mathsf{Q}(e_i, S) \qquad (e'_i, S') \in X \qquad R' \vdash e_{i/S} \rightarrow e'_{i/S'}}{R \vdash f(v_1, \ldots, v_{i-1}, e_i, e_{i+1}, \ldots, e_n, \ldots)_{/S} \rightarrow e''_{/S''}} \tag{XRS-AT-fi}$$

$$\forall e_i \ldots e_n\ S\ S'. \frac{\neg\mathsf{Q}(e_i, S) \qquad (e'_i, S') \notin X \qquad R' \vdash e_{i/S} \rightarrow e'_{i/S'}}{R \vdash f(v_1, \ldots, v_{i-1}, e_i, e_{i+1}, \ldots, e_n, \ldots)_{/S} \rightarrow f(v_1, \ldots, v_{i-1}, e'_i, e_{i+1}, \ldots, e_n, \ldots)_{/S'}} \tag{XRS-OK-fi}$$

For each such pair of rules, it holds that:

$$\begin{aligned}
&R \vdash f(v_1, \ldots, v_n, e_i, e_{i+1}, \ldots, e_n, \ldots)_{/S} \overset{\infty}{\rightarrow} \implies \\
&\neg \left(\exists e'\ S'.\ R' \vdash e_{i/S} \rightarrow^* e'_{/S'} \ \wedge\ \mathsf{Q}(e', S') \ \wedge\ R \vdash f(v_1, \ldots, v_n, e', e_{i+1}, \ldots, e_n, \ldots)_{/S'} \overset{\infty}{\rightarrow} \right) \implies \\
&R \vdash e_{i/S} \overset{\infty}{\rightarrow}
\end{aligned}$$

Proof. The proof is by guarded coinduction and inversion on the first premise, following the same structure of Lemma 5.9. □

C Specifications

This appendix provides specifications for:

- pretty-big-step SOS for $\lambda^{\bullet}_{\text{cbv}+\texttt{ref}}$ (i.e., call-by-value λ-calculus with ML-style references and abrupt termination) in Section C.1;
- Plotkin-style propagation of exceptions using SOS in Section C.2; and
- a functional derivation (following Danvy [Dan09]) for going from reduction-based to reduction-free normalization.

C.1 Pretty-big-step semantics for $\lambda^{\bullet}_{\text{cbv}+\texttt{ref}}$

This section recalls how to give big-step semantics for ML-style references using pretty-big-step semantics, extending the pretty-big-step semantics in Section 2.5.3 (page 49). The extension does not introduce any significant changes to existing rules; the main change is that outcomes now record the store σ resulting from abrupt termination.

Abstract syntax.

$$
\begin{array}{rll}
\mathit{Expr}^{\text{NS}} \ni e ::= & \ldots \mid \texttt{ref}(e) \mid \texttt{deref}(e) \mid \texttt{assign}(e,e) & \text{Expressions} \\
\mathit{Val}^{\text{NS}} \ni v ::= & \ldots \mid r & \text{Values} \\
\mathit{IntmExpr} \ni E ::= & \cdots \mid \texttt{ref1}(o) \mid \texttt{deref1}(o) & \text{Semantic expressions} \\
 & \mid \texttt{assign1}(o,e) \mid \texttt{assign2}(o,o) & \\
\mathit{Outcome}^{\text{PBS}} \ni o ::= & \text{TER}(v,\sigma) \mid \text{DIV} \mid \text{EXC}(v) & \text{Outcomes}
\end{array}
$$

Semantics.

$$\frac{\rho \vdash (e_1,\sigma) \Downarrow o_1 \qquad \rho \vdash (\texttt{plus1}(o_1,e_2),\sigma) \Downarrow o}{\rho \vdash (\texttt{plus}(e_1,e_2),\sigma) \Downarrow o} \quad (\text{PBS-}\lambda^{\bullet}_{\text{cbv+}\texttt{ref}}\text{-Plus})$$

$$\frac{\rho \vdash (e_2,\sigma) \Downarrow o_2 \qquad \rho \vdash (\texttt{plus2}(v_1,o_2),\sigma) \Downarrow o}{\rho \vdash (\texttt{plus1}(\text{TER}(v_1,\sigma),e_2),\sigma_0) \Downarrow o} \quad (\text{PBS-}\lambda^{\bullet}_{\text{cbv+}\texttt{ref}}\text{-Plus1})$$

$$\frac{}{\rho \vdash (\texttt{plus2}(n_1,\text{TER}(n_2,\sigma)),\sigma_0) \Downarrow \text{TER}(n_1+n_2,\sigma)} \quad (\text{PBS-}\lambda^{\bullet}_{\text{cbv+}\texttt{ref}}\text{-Plus2})$$

$$\frac{}{\rho \vdash (\lambda x.e,\sigma) \Downarrow \text{TER}(\langle x,e,\rho\rangle,\sigma)} \quad (\text{PBS-}\lambda^{\bullet}_{\text{cbv+}\texttt{ref}}\text{-Lam})$$

$$\frac{\rho \vdash (e_1,\sigma) \Downarrow o_1 \qquad \rho \vdash (\texttt{app1}(o_1,e_2),\sigma) \Downarrow o}{\rho \vdash (e_1\ e_2,\sigma) \Downarrow o} \quad (\text{PBS-}\lambda^{\bullet}_{\text{cbv+}\texttt{ref}}\text{-App})$$

$$\frac{\rho \vdash (e_2,\sigma) \Downarrow o_2 \qquad \rho \vdash (\texttt{app2}(v_1,e_2),\sigma) \Downarrow o}{\rho \vdash (\texttt{app1}(\text{TER}(v_1,\sigma),e_2),\sigma_0) \Downarrow o} \quad (\text{PBS-}\lambda^{\bullet}_{\text{cbv+}\texttt{ref}}\text{-App1})$$

$$\frac{\rho'[x \mapsto v_2] \vdash (e,\sigma) \Downarrow o}{\rho \vdash (\texttt{app2}(\langle x,e,\rho'\rangle,\text{TER}(v_2,\sigma)),\sigma_0) \Downarrow o} \quad (\text{PBS-}\lambda^{\bullet}_{\text{cbv+}\texttt{ref}}\text{-App2})$$

$$\frac{\rho \vdash (e,\sigma) \Downarrow o \qquad \rho \vdash (\texttt{ref1}(o),\sigma) \Downarrow o'}{\rho \vdash (\texttt{ref}(e),\sigma) \Downarrow o'} \quad (\text{PBS-}\lambda^{\bullet}_{\text{cbv+}\texttt{ref}}\text{-Ref})$$

$$\frac{r \notin \sigma}{\rho \vdash (\texttt{ref1}(\text{TER}(v,\sigma)),\sigma_0) \Downarrow \text{TER}(r,\sigma[r \mapsto v])} \quad (\text{PBS-}\lambda^{\bullet}_{\text{cbv+}\texttt{ref}}\text{-Ref1})$$

$$\frac{\rho \vdash (e,\sigma) \Downarrow o \qquad \rho \vdash (\texttt{deref1}(o),\sigma) \Downarrow o'}{\rho \vdash (\texttt{deref}(e),\sigma) \Downarrow o'} \quad (\text{PBS-}\lambda^{\bullet}_{\text{cbv+}\texttt{ref}}\text{-Deref})$$

$$\frac{r \in \sigma}{\rho \vdash (\texttt{deref1}(\text{TER}(r,\sigma)),\sigma_0) \Downarrow \text{TER}(\sigma(r),\sigma)} \quad (\text{PBS-}\lambda^{\bullet}_{\text{cbv+}\texttt{ref}}\text{-Deref1})$$

$$\frac{\rho \vdash (e_1,\sigma) \Downarrow o_1 \qquad \rho \vdash (\texttt{assign1}(o_1,e_2),\sigma) \Downarrow o}{\rho \vdash (\texttt{assign}(e_1,e_2),\sigma) \Downarrow o} \quad (\text{PBS-}\lambda^{\bullet}_{\text{cbv+}\texttt{ref}}\text{-Assign})$$

$$\frac{\rho \vdash (e_2,\sigma) \Downarrow o_2 \qquad \rho \vdash (\texttt{assign2}(v_1,o_2),\sigma) \Downarrow o}{\rho \vdash (\texttt{assign1}(\text{TER}(v_1,\sigma),e_2),\sigma_0) \Downarrow o} \quad (\text{PBS-}\lambda^{\bullet}_{\text{cbv+}\texttt{ref}}\text{-Assign1})$$

$$\frac{r \in \sigma}{\rho \vdash (\texttt{assign2}(r,\text{TER}(v,\sigma)),\sigma_0) \Downarrow \text{TER}(\texttt{unit},\sigma[r \mapsto v])} \quad (\text{PBS-}\lambda^{\bullet}_{\text{cbv+}\texttt{ref}}\text{-Assign2})$$

$$\frac{\rho \vdash (e_1,\sigma) \Downarrow o_1 \qquad \rho \vdash (\texttt{catch1}(o_1,x,e_2),\sigma) \Downarrow o}{\rho \vdash (\texttt{catch}(e_1,x,e_2),\sigma) \Downarrow o} \quad (\text{PBS-}\lambda^{\bullet}_{\text{cbv+}\texttt{ref}}\text{-Catch})$$

$$\frac{}{\rho \vdash (\texttt{catch1}(\text{TER}(v_1,\sigma),x,e_2),\sigma_0) \Downarrow \text{TER}(v_1,\sigma)} \quad (\text{PBS-}\lambda^{\bullet}_{\text{cbv+}\texttt{ref}}\text{-CatchV})$$

$$\frac{\rho[x \mapsto v] \vdash (e_2,\sigma) \Downarrow o}{\rho \vdash (\texttt{catch1}(\text{EXC}(v),e_2),\sigma) \Downarrow o} \quad (\text{PBS-}\lambda^{\bullet}_{\text{cbv+}\texttt{ref}}\text{-Catch})$$

$$\frac{abort(o_1)}{\rho \vdash (\texttt{plus1}(o_1, e_2), \sigma) \Downarrow o_1} \quad (\text{PBS-}\lambda_{\text{cbv}+\texttt{ref}}\text{-Abort-Plus1})$$

$$\frac{abort(o_2)}{\rho \vdash (\texttt{plus2}(n_1, o_2), \sigma) \Downarrow o_2} \quad (\text{PBS-}\lambda_{\text{cbv}+\texttt{ref}}\text{-Abort-Plus2})$$

$$\frac{abort(o_1)}{\rho \vdash (\texttt{app1}(o_1, e_2), \sigma) \Downarrow o_1} \quad (\text{PBS-}\lambda_{\text{cbv}+\texttt{ref}}\text{-Abort-App1})$$

$$\frac{abort(o_2)}{\rho \vdash (\texttt{app2}(\langle x, e, \rho' \rangle, o_2), \sigma) \Downarrow \text{DIV}} \quad (\text{PBS-}\lambda_{\text{cbv}+\texttt{ref}}\text{-Abort-App2})$$

$$\frac{abort(o)}{\rho \vdash (\texttt{ref1}(o), \sigma) \Downarrow o} \quad (\text{PBS-}\lambda_{\text{cbv}+\texttt{ref}}\text{-Abort-Ref1})$$

$$\frac{abort(o)}{\rho \vdash (\texttt{deref1}(o), \sigma) \Downarrow o} \quad (\text{PBS-}\lambda_{\text{cbv}+\texttt{ref}}\text{-Abort-Deref1})$$

$$\frac{abort(o_1)}{\rho \vdash (\texttt{assign1}(o_1, e_2), \sigma) \Downarrow o_1} \quad (\text{PBS-}\lambda_{\text{cbv}+\texttt{ref}}\text{-Abort-Assign1})$$

$$\frac{abort(o_2)}{\rho \vdash (\texttt{assign2}(r, o_2), \sigma) \Downarrow o_2} \quad (\text{PBS-}\lambda_{\text{cbv}+\texttt{ref}}\text{-Abort-Assign2})$$

$$\frac{abort(o)}{\rho \vdash (\texttt{throw1}(o), \sigma) \Downarrow o} \quad (\text{PBS-}\lambda^{\bullet}_{\text{cbv}+\texttt{ref}}\text{-Abort-Throw})$$

C.2 Plotkin-style propagation of abrupt termination in SOS

We illustrate Plotkin's [Plo81] approach to propagating abrupt termination in SOS rules for $\lambda^{\bullet}_{\text{cbv}}$. The difference is summarised by comparing the rules from Figure 2.11 to the Plotkin-style propagation rules in Figure C.1.

Plotkin's approach to propagating exceptions is, for each congruence rule in the language, to have a rule with a premise that results in abrupt termination. Unlike the approach we saw in Section 2.2.5, propagation à la Plotkin propagates the exception instantaneously. Using $\rightarrow^{2.11}$ for the transition relation for $\lambda^{\bullet}_{\text{cbv}}$ with the propagation rules from Figure 2.11, and $\rightarrow^{C.1}$ the transition relation with the propagation rules in Figure C.1. Consider the expression $\texttt{plus}(1, \texttt{plus}(2, \texttt{plus}(\texttt{throw}(0), 3)))$. Using Plotkin-propagation, this expression causes an exception in a single transition (where we omit the irrelavant parts of the configuration, such as the store and environment):

$$\begin{array}{rl} & (\texttt{plus}(1, \texttt{plus}(2, \texttt{plus}(\texttt{throw}(0), 3))), \sigma) \\ \rightarrow^{C.1} & (\text{EXC}(0), \sigma) \end{array}$$

$$\frac{\rho \vdash (e_1, \sigma) \to (\text{EXC}(v), \sigma')}{\rho \vdash (\texttt{plus}(e_1, e_2), \sigma) \to (\text{EXC}(v), \sigma')} \quad \text{(SOS-Plus1-PExc)}$$

$$\frac{\rho \vdash (e_2, \sigma) \to (\text{EXC}(v), \sigma')}{\rho \vdash (\texttt{plus}(n_1, e_2), \sigma) \to (\text{EXC}(v), \sigma')} \quad \text{(SOS-Plus2-PExc)}$$

$$\frac{\rho \vdash (e_1, \sigma) \to (\text{EXC}(v), \sigma')}{\rho \vdash (e_1\ e_2, \sigma) \to (\text{EXC}(v), \sigma')} \quad \text{(SOS-App1-PExc)}$$

$$\frac{\rho \vdash (e_2, \sigma) \to (\text{EXC}(v), \sigma')}{\rho \vdash (v_1\ e_2, \sigma) \to (\text{EXC}(v), \sigma')} \quad \text{(SOS-App2-PExc)}$$

$$\frac{\rho \vdash (e, \sigma) \to (\text{EXC}(v), \sigma')}{\rho \vdash (\texttt{ref}(e), \sigma) \to (\text{EXC}(v), \sigma')} \quad \text{(SOS-Ref-PExc)}$$

$$\frac{\rho \vdash (e, \sigma) \to (\text{EXC}(v), \sigma')}{\rho \vdash (\texttt{deref}(e), \sigma) \to (\text{EXC}(v), \sigma')} \quad \text{(SOS-Deref-PExc)}$$

$$\frac{\rho \vdash (e_1, \sigma) \to (\text{EXC}(v), \sigma')}{\rho \vdash (\texttt{assign}(e_1, e_2), \sigma) \to (\text{EXC}(v), \sigma')} \quad \text{(SOS-Assn1-PExc)}$$

$$\frac{\rho \vdash (e_2, \sigma) \to (\text{EXC}(v), \sigma')}{\rho \vdash (\texttt{assign}(r, e_2), \sigma) \to (\text{EXC}(v), \sigma')} \quad \text{(SOS-Assn2-PExc)}$$

$$\frac{\rho \vdash (e, \sigma) \to (\text{EXC}(v), \sigma')}{\rho \vdash (\texttt{throw}(e), \sigma) \to (\text{EXC}(v), \sigma')} \quad \text{(SOS-Throw-PExc)}$$

Figure C.1: SOS rules à la Plotkin for propagating exceptions

Using the propagation rules from Section 2.2.5, however, requires multiple steps to propagate the exception:

$$\begin{array}{ll}
 & \texttt{plus}(1, \texttt{plus}(2, \texttt{plus}(\texttt{throw}(0), 3))) \\
\to^{2.11} & \texttt{plus}(1, \texttt{plus}(2, \texttt{plus}(\text{EXC}(0), 3))) \\
\to^{2.11} & \texttt{plus}(1, \texttt{plus}(2, \text{EXC}(0))) \\
\to^{2.11} & \texttt{plus}(1, \text{EXC}(0)) \\
\to^{2.11} & \text{EXC}(0)
\end{array}$$

Exception propagation à la Plotkin, however, still requires us to add rules for existing constructs, so it is a non-modular extension.

C.3 From reduction-based to reduction-free normalisation for simple arithmetic expressions

This appendix shows the steps involved in deriving a big-step functional evaluator for going from a reduction-based (big-step evaluation strategy iterated by an `iterate` function) to reduction-free normalisation function (big-step tail-recursive evaluation function), following Danvy's From Reduction-Based to Reduction-Free Normalization [Dan09].

```
datatype Term = NUM of int
              | PLUS of Term * Term

type Value = int

(* Prelude to a reduction semantics *)
datatype PotRed = PR_PLUS of Term * Term

datatype Found = FVAL of Value | POTRED of PotRed

fun search (PLUS (NUM n1, NUM n2))
    = POTRED (PR_PLUS (NUM n1, NUM n2))
  | search (PLUS (t1, t2))
    = (case (search t1)
        of (FVAL v) => (search t2)
         | (POTRED p) => POTRED p)
  | search (NUM n) = FVAL n

(* CPS transform *)
fun search_k (PLUS (t1, t2), k)
    = search_k (t1,
                fn (FVAL n1) =>
                  search_k (t2,
                            fn (FVAL n2) =>
                              k (POTRED (PR_PLUS (NUM n1, NUM n2)))
                             | (POTRED p) =>
                              k (POTRED p))
                 | (POTRED p) =>
                  k (POTRED p))
  | search_k (NUM n, k) = k (FVAL n)

(* Simplification *)
fun search_k' (PLUS (t1, t2), k)
    = search_k' (t1,
                 fn n1 =>
                 search_k' (t2, fn n2 =>
                                POTRED (PR_PLUS (NUM n1, NUM n2))))
  | search_k' (NUM n, k) = k n

(* Defunctionalization *)
datatype Ctx = C_MT
             | C_PLUS1 of Term * Ctx
             | C_PLUS2 of Value * Ctx

fun search_c (PLUS (t1, t2), c)
    = search_c (t1, C_PLUS1 (t2, c))
  | search_c (NUM n, c)
    = apply_c (c, n)
and apply_c (C_MT, n)
    = FVAL n
  | apply_c (C_PLUS1 (t2, c), n)
    = search_c (t2, (C_PLUS2 (n, c)))
  | apply_c (C_PLUS2 (n1, c), n2)
    = POTRED (PR_PLUS (NUM n1, NUM n2))

(* Decomposition *)
datatype ValOrDecomp = VAL of Value
                     | DECOMP of PotRed * Ctx

fun decompose_t (PLUS (t1, t2), c)
    = decompose_t (t1, C_PLUS1 (t2, c))
  | decompose_t (NUM n, c)
    = decompose_c (c, n)
and decompose_c (C_MT, n)
    = VAL n
  | decompose_c (C_PLUS1 (t2, c), n)
    = decompose_t (t2, C_PLUS2 (n, c))
  | decompose_c (C_PLUS2 (n1, c), n2)
    = DECOMP (PR_PLUS (NUM n1, NUM n2), c)

fun decompose t = decompose_t (t, C_MT)

(* Recomposition *)
fun recompose (C_MT, t)
    = t
  | recompose (C_PLUS1 (t2, c), t1)
    = recompose (c, PLUS (t1, t2))
  | recompose (C_PLUS2 (v, c), t2)
    = recompose (c, PLUS (NUM v, t2))
```

```
(* Notion of contraction *)
datatype ContractOrErr = CONTRACTUM of Term | ERROR of string

fun contract (PR_PLUS (NUM n1, NUM n2))
    = CONTRACTUM (NUM (n1+n2))
  | contract _
    = ERROR "Invalid operands for addition"

(* One-step reduction *)
datatype ReductOrStuck = REDUCT of Term
                       | STUCK of string

fun reduce t
    = (case decompose t
        of (VAL v)
           => STUCK "Irreducible term"
         | (DECOMP (pr, k))
           => (case contract pr
                of (CONTRACTUM t')
                   => REDUCT (recompose (k, t'))
                 | (ERROR s)
                   => STUCK s))

(* Reduction-based normalization *)
datatype result_or_wrong = RESULT of Value
                         | WRONG of string

fun iterate (VAL v)
    = RESULT v
  | iterate (DECOMP (pr, k))
    = (case contract pr
        of (CONTRACTUM t')
           => iterate (decompose (recompose (k, t')))
        |  (ERROR s)
           => WRONG s)

fun normalize t = iterate (decompose t)

(* Refocusing *)
fun refocus0 (t, k) = decompose (recompose (k, t))

fun iterate0 (VAL n)
    = RESULT n
  | iterate0 (DECOMP (pr, k))
    = (case contract pr
        of (CONTRACTUM t')
           => iterate0 (refocus0 (t', k))
         | (ERROR s)
           => WRONG s)

fun normalize0 t = iterate (refocus0 (t, C_MT))

fun refocus1 (t, k) = decompose_t (t, k)

fun iterate1 (VAL n)
    = RESULT n
  | iterate1 (DECOMP (pr, k))
    = (case contract pr
        of (CONTRACTUM t')
           => iterate1 (refocus1 (t', k))
         | (ERROR s)
           => WRONG s)

fun normalize1 t = iterate1 (refocus1 (t, C_MT))

(* Inlining the contraction function *)

fun refocus2 (t, k) = decompose_t (t, k)

fun iterate2 (VAL n)
    = RESULT n
  | iterate2 (DECOMP (PR_PLUS (NUM n1, NUM n2), k))
    = iterate2 (refocus1 (NUM (n1+n2), k))
  | iterate2 (DECOMP (_, _))
    = WRONG "Invalid operands for addition"

fun normalize2 t = iterate2 (refocus2 (t, C_MT))

(* Lightweight fusion *)
fun iterate3 (VAL n)
    = RESULT n
  | iterate3 (DECOMP (PR_PLUS (NUM n1, NUM n2), c))
    = iterate3_t (NUM (n1+n2), c)
  | iterate3 (DECOMP (_, _))
    = WRONG "Invalid operands for addition"
and iterate3_t (PLUS (t1, t2), c)
    = iterate3_t (t1, C_PLUS1 (t2, c))
  | iterate3_t (NUM n, c)
    = iterate3_c (c, n)
and iterate3_c (C_MT, n)
    = iterate3 (VAL n)
  | iterate3_c (C_PLUS1 (t2, c), n)
    = iterate3_t (t2, C_PLUS2 (n, c))
  | iterate3_c (C_PLUS2 (n1, c), n2)
    = iterate3 (DECOMP (PR_PLUS (NUM n1, NUM n2), c))

fun normalize3 t = iterate3_t (t, C_MT)

(* Transition compression *)
fun iterate4_t (PLUS (t1, t2), c)
    = iterate4_t (t1, C_PLUS1 (t2, c))
  | iterate4_t (NUM n, c)
    = iterate4_c (c, n)
and iterate4_c (C_MT, n)
    = RESULT n
  | iterate4_c (C_PLUS1 (t2, c), n)
    = iterate4_t (t2, C_PLUS2 (n, c))
  | iterate4_c (C_PLUS2 (n1, c), n2)
    = iterate4_c (c, n1+n2)

fun normalize4 t = iterate4_t (t, C_MT)

(* Refunctionalization *)
fun iterate5 (PLUS (t1, t2), k)
    = iterate5 (t1, fn n1 =>
                  iterate5 (t2, fn n2 =>
                                k (n1+n2)))
  | iterate5 (NUM n, k)
    = k n

fun normalize5 t = iterate5 (t, fn n => n)

(* Back to direct-style *)
fun iterate6 (PLUS (t1, t2))
    = (case iterate6 t1
        of n1 => (case iterate6 t2
                   of n2 =>
                      (n1 + n2)))
  | iterate6 (NUM n)
    = n
```

```
fun normalize6 (PLUS (t1, t2))
    = normalize6 t1 + normalize6 t2
  | normalize6 (NUM n)
    = n
```

Bibliography

[ACCL91] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. *J. Funct. Program.*, 1(4):375–416, 1991.

[Ahm04] Amal Jamil Ahmed. *Semantics of Types for Mutable State*. PhD thesis, Princeton University, 2004.

[AJM00] Samson Abramsky, Radha Jagadeesan, and Pasquale Malacaria. Full abstraction for PCF. *Information and Computation*, 163(2):409 – 470, 2000.

[Anc12] Davide Ancona. Soundness of object-oriented languages with coinductive big-step semantics. In James Noble, editor, *ECOOP'12*, volume 7313 of *LNCS*, pages 459–483. Springer, 2012.

[Awo06] Steve Awodey. *Category theory*, volume 49 of *Oxford Logic Guides*. Oxford University Press, 2006.

[Bac03] Roland Backhouse. *Program Construction: Calculating Implementations from Specifications*. John Wiley and Sons, Inc., 2003.

[Bar84] H. P. Barendregt. *The Lambda Calculus Its Syntax and Semantics*, volume 103. North Holland, revised edition, 1984.

[Bar92] H. P. Barendregt. Lambda calculi with types. In S. Abramsky, Dov M. Gabbay, and S. E. Maibaum, editors, *Handbook of Logic in Computer Science (Vol. 2)*, pages 117–309. Oxford University Press, Inc., New York, NY, USA, 1992.

[BBE15] Patrick Bahr, Jost Berthold, and Martin Elsman. Certified symbolic management of financial multi-party contracts. Submitted to ICFP'15, 2015.

[BBS06] Patrick Blackburn, Johan Bos, and Kristina Striegnitz. *Learn Prolog Now!*, volume 7 of *Texts in Computing*. College Publications, 2006.

[BC04] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.

[BD06] Dariusz Biernacki and Olivier Danvy. A simple proof of a folklore theorem about delimited control. *J. Funct. Program.*, 16(3):269–280, May 2006.

[BD07] Magorzata Biernacka and Olivier Danvy. A syntactic correspondence between context-sensitive calculi and abstract machines. *Theor. Comput. Sci.*, 375(1-3):76–108, April 2007.

[BD09] Magorzata Biernacka and Olivier Danvy. Towards compatible and interderivable semantic specifications for the scheme programming language, part ii: Reduction semantics and abstract machines. In Jens Palsberg, editor, *Semantics and Algebraic Specification: Essays Dedicated to Peter D. Mosses*, volume 5700 of *LNCS*, pages 186–206. Springer, 2009.

[BDN09] Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of Agda – a functional language with dependent types. In *TPHOLs'09*, pages 73–78. Springer, 2009.

[BH15] Patrick Bahr and Graham Hutton. Calculating correct compilers. *J. of Funct. Program.*, 25, September 2015.

[BJS15] Martin Bodin, Thomas Jensen, and Alan Schmitt. Certified abstract interpretation with pretty-big-step semantics. In *CPP'15*, pages 29–40, 2015.

[BPM14a] Casper Bach Poulsen and Peter D. Mosses. Deriving pretty-big-step semantics from small-step semantics. In *ESOP'14*, volume 8410 of *LNCS*, pages 270–289. Springer, 2014.

[BPM14b] Casper Bach Poulsen and Peter D. Mosses. Divergence as state in coinductive big-step semantics. Presented at NWPT'14, 2014.

[BPM14c] Casper Bach Poulsen and Peter D. Mosses. Generating specialized interpreters for Modular Structural Operational Semantics. In *LOPSTR'13*, volume 8901 of *LNCS*, pages 220–236. Springer, 2014.

[BPM16] Casper Bach Poulsen and Peter D. Mosses. Flag-based big-step semantics. To appear in *The Journal of Logical and Algebraic Methods in Programming*. `http://plancomps.org/flag-based-big-step`, 2016.

[BPMT15] Casper Bach Poulsen, Peter D. Mosses, and Paolo Torrini. Imperative polymorphism by store-based types as abstract interpretations. In *PEPM'15*, pages 3–8. ACM, 2015.

[Cap05] Venanzio Capretta. General recursion via coinductive types. *Logical Methods in Computer Science*, 1(2):1–18, 2005.

[Cap13] Venanzio Capretta. Wander types: A formalization of coinduction-recursion (special issue: Advanced programming techniques for construction of robust, generic and evolutionary programs). *Progress in Informatics*, 10:47–63, 2013.

[Car96] Luca Cardelli. Type systems. *ACM Comput. Surv.*, 28(1):263–264, March 1996.

[CC79] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *POPL'79*, pages 269–282. ACM, 1979.

[CC92] Patrick Cousot and Radhia Cousot. Inductive definitions, semantics and abstract interpretations. In *POPL'92*, pages 83–94. ACM, 1992.

[CF58] H. B. Curry and R. Feys. *Combinatory Logic, Volume I*. North-Holland, 1958. Second printing 1968.

[CF93] Mario Coppo and Alberto Ferrari. Type inference, abstract interpretation and strictness analysis. *Theor. Comp. Sci.*, 121(12):113 – 143, 1993.

[CF94] Robert Cartwright and Matthias Felleisen. Extensible denotational language specifications. In *TACS'94*, pages 244–272. Springer, 1994.

[Cha09] James Chapman. *Type checking and normalisation*. PhD thesis, University of Nottingham, 2009.

[Cha13] Arthur Charguéraud. Pretty-big-step semantics. In Matthias Felleisen and Philippa Gardner, editors, *ESOP'13*, volume 7792 of *LNCS*, pages 41–60. Springer, 2013.

[CHJ12] Ravi Chugh, David Herman, and Ranjit Jhala. Dependent types for JavaScript. *SIGPLAN Not.*, 47(10):587–606, October 2012.

[Chl11] Adam Chlipala. *Certified Programming with Dependent Types*. MIT Press, 2011.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *J. Symbolic Logic*, 5(2):56–68, 06 1940.

[Cio13] Ştefan Ciobâcă. From small-step semantics to big-step semantics, automatically. In Einar Broch Johnsen and Luigia Petre, editors, *IFM'13*, volume 7940 of *LNCS*, pages 347–361. Springer, 2013.

[Cli87] William Clinger. The scheme environment: Continuations. *SIGPLAN Lisp Pointers*, 1(2):22–28, June 1987.

[CM93] Pietro Cenciarelli and Eugenio Moggi. A syntactic approach to modularity in denotational semantics. In *Category Theory and Computer Science*, 1993.

[CM13] Martin Churchill and Peter D. Mosses. Modular bisimulation theory for computations and values. In Frank Pfenning, editor, *FoSSaCS'13*, volume 7794 of *LNCS*, pages 97–112. Springer, 2013.

[CMM13] Martin Churchill, Peter D. Mosses, and Mohammad Reza Mousavi. Modular semantics for transition system specifications with negative premises. In Pedro R. D'Argenio and Hernán Melgratti, editors, *CONCUR'13*, pages 46–60. Springer, 2013.

[CMST15] Martin Churchill, Peter D. Mosses, Neil Sculthorpe, and Paolo Torrini. Reusable components of semantic specifications. In *Transactions on Aspect-Oriented Software Development XII*, volume 8989 of *LNCS*, pages 132–179. Springer, 2015.

[Cou97] Patrick Cousot. Types as abstract interpretations. In *POPL'97*, pages 316–331. ACM, 1997.

[CPM90] Thierry Coquand and Christine Paulin-Mohring. Inductively defined types. In Per Martin-Löf and Grigori Mints, editors, *COLOG-88*, volume 417 of *LNCS*, pages 50–66. Springer, 1990.

[CR36] A. Church and J. B. Rosser. Some properties of conversion. *Transactions of the American Mathematical Society*, 39:472–482, 1936.

[Cur91] Pierre-Louis Curien. An abstract framework for environment machines. *Theor. Comp. Sci.*, 82(2):389–402, 1991.

[CWM99] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *POPL'99*, pages 262–275. ACM, 1999.

[Dam84] L. Damas. *Type Assignment in Programming Languages*. PhD thesis, University of Edinburgh, 1984.

[Dan04] Olivier Danvy. On evaluation contexts, continuations, and the rest of the computation. In *Proceedings of the 4th ACM SIGPLAN Workshop on Continuations (CW)*, 2004.

[Dan08] Olivier Danvy. Defunctionalized interpreters for programming languages. In James Hook and Peter Thiemann, editors, *ICFP'08*, pages 131–142. ACM, 2008.

[Dan09] Olivier Danvy. From reduction-based to reduction-free normalization. In *AFP'08*, pages 66–164. Springer, 2009.

[Dan12] Nils Anders Danielsson. Operational semantics using the partiality monad. In *ICFP'12*, pages 127–138. ACM, 2012.

[DdSOS13] Benjamin Delaware, Bruno C. d. S. Oliveira, and Tom Schrijvers. Meta-theory à la carte. In *POPL'13*, pages 207–218. ACM, 2013.

[DF90] Olivier Danvy and Andrzej Filinski. Abstracting control. In *LFP '90*, pages 151–160. ACM, 1990.

[DHM91] Bruce Duba, Robert Harper, and David MacQueen. Typing first-class continuations in ML. In *POPL'91*, pages 163–173. ACM, 1991.

[DJZ11] Olivier Danvy, Jacob Johannsen, and Ian Zerny. A walk in the semantic park. In *PEPM'11*, pages 1–12. ACM, 2011.

[DKSO13] Benjamin Delaware, Steven Keuchel, Tom Schrijvers, and Bruno C.d.S. Oliveira. Modular monadic meta-theory. In *ICFP'13*, pages 319–330. ACM, 2013.

[DM08] Olivier Danvy and Kevin Millikin. On the equivalence between small-step and big-step abstract machines: A simple application of lightweight fusion. *Inf. Process. Lett.*, 106(3):100–109, 2008.

[DN04] Olivier Danvy and Lasse R. Nielsen. Refocusing in reduction semantics. BRICS Research Series RS-04-26, Department of Computer Science, Aarhus University, 2004.

[DS06] Peter Dybjer and Anton Setzer. Indexed inductionrecursion. *The Journal of Logic and Algebraic Programming*, 66(1):1 – 49, 2006.

[Dyb00] Peter Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *Journal of Symbolic Logic*, 65:525–549, 6 2000.

[ECM06] ECMA. *C# Language Specification*. ECMA (European Association for Standardizing Information and Communication Systems), June 2006.

[Fel87] Matthias Felleisen. *The Calculi of Lambda-v-CS Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages*. PhD thesis, Department of Computer Science, Indiana University, 1987.

[Fel88] Matthias Felleisen. The theory and practice of first-class prompts. In *POPL'88*, pages 180–190. ACM, 1988.

[FWFD88] Matthias Felleisen, Mitch Wand, Daniel Friedman, and Bruce Duba. Abstract continuations: A mathematical semantics for handling full jumps. In *Conference on LISP and Functional Programming*, pages 52–62. ACM, 1988.

[Gal14] Letterio Galletta. An abstract interpretation framework for type and effect systems. *Fundamenta Informaticae*, 134, January 2014.

[Gar04] Jacques Garrigue. Relaxing the value restriction. In Yukiyoshi Kameyama and Peter J. Stuckey, editors, *FLOPS'04*, volume 2998 of *LNCS*, pages 196–213. Springer, 2004.

[Gir72] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.

[GJS$^+$13] James Gosling, Bill Joy, Guy L. Steele, Jr., Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 7 Edition*. Addison-Wesley Professional, 1st edition, 2013.

[GMNF13] Neil Ghani, Lorenzo Malatesta, and Fredrik Nordvall Forsberg. Positive inductive-recursive definitions. In Reiko Heckel and Stefan Milius, editors, *CALCO'13*, volume 8089 of *LNCS*, pages 19–33. Springer, 2013.

[GMNF15] Neil Ghani, Lorenzo Malatesta, and Fredrik Nordvall Forsberg. Positive inductive-recursive definitions. *Logical Methods in Computer Science*, 11(1), 2015.

[GMNFS13] Neil Ghani, Lorenzo Malatesta, Fredrik Nordvall Forsberg, and Anton Setzer. Fibred data types. In *LICS 2013*, pages 243 – 252, 2013.

[Gro93] Jan Friso Groote. Transition system specifications with negative premises. *Theoretical Computer Science*, 118(2):263–299, 1993.

[HDM93] Robert Harper, Bruce F. Duba, and David Macqueen. Typing first-class continuations in ML. *JFP*, 3:465–484, 10 1993.

[HHPJW07] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of Haskell: Being lazy with class. In *HOPL III*, pages 12–1–12–55. ACM, 2007.

[Hin69] R. Hindley. The principal type-scheme of an object in combinatory logic. *Trans. Amer. Math. Soc*, 146:29–60, December 1969.

[Hin97] J. Roger Hindley. *Basic Simple Type Theory*. Cambridge University Press, 1997.

[HMN04] Fritz Henglein, Henning Makholm, and Henning Niss. Effect types and region-based memory management. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 3. The MIT Press, 2004.

[HMU03] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation - international edition (2. ed)*. Addison-Wesley, 2003.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978.

[How80] William A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 479–490. Academic Press, 1980. Reprint of 1969 article.

[HS00a] Peter Hancock and Anton Setzer. Interactive programs in dependent type theory. In P. Clote and H. Schwichtenberg, editors, *CSL'00*, volume 1862 of *LNCS*, pages 317–331. Springer, 2000.

[HS00b] Robert Harper and Christopher Stone. A type-theoretic interpretation of Standard ML. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction: Essays in Honour of Robin Milner*, pages 341–387. MIT Press, 2000.

[Hue97] Gérard Huet. The zipper. *J. Funct. Program.*, 7(5):549–554, September 1997.

[HW07] Graham Hutton and Joel Wright. What is the meaning of these constant interruptions? *J. Funct. Program.*, 17:777–792, 2007.

[JGH11] Mauro Jaskelioff, Neil Ghani, and Graham Hutton. Modularity and implementation of mathematical operational semantics. *ENTCS*, 229(5):75 – 95, 2011. MSFP'08.

[Jim96] Trevor Jim. What are principal typings and what are they good for? In *POPL'96*, pages 42–53. ACM, 1996.

[Joh15] Jacob Johannsen. *On Computational Small Steps and Big Steps: Refocusing for Outermost Reduction*. PhD thesis, Department of Computer Science, Unversity of Aarhus, 2015.

[Kah87] G. Kahn. Natural semantics. In *STACS'87*, volume 247 of *LNCS*, pages 22–39. Springer, 1987.

[KCD$^+$12] Casey Klein, John Clements, Christos Dimoulas, Carl Eastlund, Matthias Felleisen, Matthew Flatt, Jay A. McCarthy, Jon Rafkind, Sam Tobin-Hochstadt, and Robert Bruce Findler. Run your research: On the effectiveness of lightweight mechanization. In *POPL'12*, pages 285–296. ACM, 2012.

[KMF07] George Kuan, David MacQueen, and RobertBruce Findler. A rewriting semantics for type inference. In Rocco De Nicola, editor, *Programming Languages and Systems*, volume 4421 of *LNCS*, pages 426–440. Springer, 2007.

[KMNO14] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: A verified implementation of ML. In *POPL '14*, pages 179–191. ACM, 2014.

[KN06] Gerwin Klein and Tobias Nipkow. A machine-checked model for a Java-like language, virtual machine, and compiler. *ACM Trans. Program. Lang. Syst.*, 28(4):619–695, July 2006.

[Lan64] Peter J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320, January 1964.

[Lan65] P. J. Landin. Correspondence between algol 60 and church's lambda-notation: Part i. *Commun. ACM*, 8(2):89–101, February 1965.

[Ler06] Xavier Leroy. Formal certification of a compiler back-end or: Programming a compiler with a proof assistant. In *POPL'06*, pages 42–54. ACM, 2006.

[LG09] Xavier Leroy and Hervé Grall. Coinductive big-step operational semantics. *Information and Computation*, 207(2):284–304, 2009.

[LHJ95] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *POPL'95*, pages 333–343. ACM, 1995.

[LW91] Xavier Leroy and Pierre Weis. Polymorphic type inference and assignment. In *POPL'91*, pages 291–302. ACM, 1991.

[Man74] Zohar Manna. *Mathematical Theory of Computation*. McGraw-Hill, Inc., 1974.

[Mar10] Simon Marlow. *Haskell 2010 Language Report*, 2010.

[Mig10] Matthew Might. Abstract interpreters for free. In Radhia Cousot and Matthieu Martel, editors, *Static Analysis*, volume 6337 of *LNCS*, pages 407–421. Springer, 2010.

[Mil75] Robin Milner. Processes: A mathematical model of computing agents. In *Logic Colloquium'73*, Studies in logic and the foundations of mathematics, pages 157–153. North-Holland Pub. Co., 1975.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348 – 375, 1978.

[Mil82] Robin Milner. *A Calculus of Communicating Systems*. Springer, 1982.

[Mit94] Kevin Mitchell. Concurrency in a natural semantics. Technical Report ECS-LFCS-94-311, The University of Edinburgh, 1994.

[MJ86] Alan Mycroft and Neil D. Jones. A relational framework for abstract interpretation. In Harald Ganzinger and Neil D. Jones, editors, *Programs as Data Objects*, volume 217 of *LNCS*, pages 156–171. Springer, 1986.

[ML71] Saunders Mac Lane. *Categories for the Working Mathematician*. Number 5 in Graduate Texts in Mathematics. Springer, 1971.

[MMR10] Peter D. Mosses, Mohammad Reza Mousavi, and Michel A. Reniers. Robustness of equations under operational extensions. In Sibylle B. Fröschle and Frank D. Valencia, editors, *EXPRESS'10*, volume 41 of *EPTCS*, pages 106–120, 2010.

[MN09] Peter D. Mosses and Mark J. New. Implicit propagation in Structural Operational Semantics. *ENTCS*, 229(4):49–66, 2009.

[Mog89] E. Moggi. Computational lambda-calculus and monads. In *LICS*, pages 14–23. IEEE Press, 1989.

[Mog90] Eugenio Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, The University of Edinburgh, 1990.

[Mog91] Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, 1991.

[Mon95] Bruno Monsuez. System F and abstract interpretation. In Alan Mycroft, editor, *SAS'95*, volume 983 of *LNCS*, pages 279–295. Springer, 1995.

[Mor04] Greg Morrisett. Typed assembly language. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 4. The MIT Press, 2004.

[Mos90] Peter D. Mosses. Denotational semantics. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science (Vol. B)*, pages 575–631. MIT Press, 1990.

[Mos99] Peter D. Mosses. Foundations of Modular SOS. In Mirosław Kutyłowski, Leszek Pacholski, and Tomasz Wierzbicki, editors, *MFCS'99*, volume 1672 of *LNCS*, pages 70–80. Springer, 1999.

[Mos04] Peter D. Mosses. Modular structural operational semantics. *J. Log. Algebr. Program.*, 60-61:195–228, 2004.

[MRG07] Mohammad Reza Mousavi, Michel A. Reniers, and Jan Friso Groote. SOS formats and meta-theory: 20 years after. *Theor. Comp. Sci.*, 373(3):238 – 272, 2007.

[MSBP15] Peter Mosses, Neil Sculthorpe, and Casper Bach Poulsen. Using typings as types, 2015. Presented at NWPT'15.

[MSvE11] Ken Madlener, Sjaak Smetsers, and Marko C. J. D. van Eekelen. Formal component-based semantics. In Michel A. Reniers and Pawel Sobocinski, editors, *SOS'11*, volume 62 of *EPTCS*, pages 17–29, 2011.

[MT91] Robin Milner and Mads Tofte. Co-induction in relational semantics. *Theor. Comp. Sci.*, 87(1):209 – 220, 1991.

[MTH89] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1989.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML, Revised*. MIT Press, 1997.

[NF13] Fredrik Nordvall Forsberg. *Inductive-inductive definitions*. PhD thesis, Swansea University, 2013.

[NH09] Keiko Nakata and Masahito Hasegawa. Small-step and big-step semantics for call-by-need. *J. Funct. Program.*, 19(6):699–722, November 2009.

[Nip06] Tobias Nipkow. Jinja: Towards a comprehensive formal semantics for a Java-like language. In H. Schwichtenberg and K. Spies, editors, *Proof Technology and Computation*, pages 247–277. IOS Press, 2006.

[NK14] Tobias Nipkow and Gerwin Klein. *Concrete Semantics: With Isabelle/HOL*. Springer, 2014.

[NN99] Flemming Nielson and Hanne Riis Nielson. Type and effect systems. In Ernst-Rüdiger Olderog and Bernhard Steffen, editors, *Correct System Design*, volume 1710 of *LNCS*, pages 114–136. Springer, 1999.

[NNH99] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999.

[NTVW15] Pierre Neron, Andrew Tolmach, Eelco Visser, and Guido Wachsmuth. A theory of name resolution. In Jan Vitek, editor, *ESOP'15*, volume 9032 of *LNCS*, pages 205–231. Springer, 2015.

[NU09] Keiko Nakata and Tarmo Uustalu. Trace-based coinductive operational semantics for while. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *TPHOLs'09*, volume 5674 of *LNCS*, pages 375–390. Springer, 2009.

[NU10] Keiko Nakata and Tarmo Uustalu. Resumptions, weak bisimilarity and big-step semantics for while with interactive I/O: an exercise in mixed induction-coinduction. In Luca Aceto and Pawel Sobocinski, editors, *SOS'10*, volume 32 of *EPTCS*, pages 57–75, 2010.

[OS07] Atsushi Ohori and Isao Sasano. Lightweight fusion by fixed point promotion. In *POPL'07*, pages 143–154. ACM, 2007.

[PCG$^+$13] Benjamin C. Pierce, Chris Casinghino, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. *Software Foundations*. 2013.

[PG14] Maciej Piróg and Jeremy Gibbons. The coinductive resumption monad. *ENTCS*, 308:273–288, 2014. MFPS'14.

[Pie91] Benjamin C. Pierce. *Basic Category Theory for Computer Scientists*. The MIT Press, 1991.

[Pie02] Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.

[PJES00] Simon Peyton Jones, Jean-Marc Eber, and Julian Seward. Composing contracts: An adventure in financial engineering (functional pearl). In *ICFP'00*, pages 280–292. ACM, 2000.

[Plo73] Gordon D. Plotkin. Lambda-definability and logical relations, 1973. Memorandum SAI-RM 4, School of Artificial Intelligence, University of Edinburgh.

[Plo75] G.D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theor. Comp. Sci.*, 1(2):125 – 159, 1975.

[Plo76] Gordon D. Plotkin. A powerdomain construction. *SIAM J. of Comput.*, 5(3):452–487, 1976.

[Plo81] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report FN-19, Aarhus University, 1981. Later published in [Plo04].

[Plo04] Gordon D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004.

[Pol95] Robert Pollack. Polishing up the Tait-Martin-Löf proof of the Church-Rosser theorem. Unpublished, January 1995.

[Rey72] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM Annual Conference*, pages 717–740. ACM, 1972.

[Rey74] John C. Reynolds. Towards a theory of type structure. In B. Robinet, editor, *Programming Symposium*, volume 19 of *LNCS*, pages 408–425. Springer, 1974.

[Rey03] John C. Reynolds. What do types mean? from intrinsic to extrinsic semantics. In Annabelle McIver and Carroll Morgan, editors, *Programming Methodology*, Monographs in Computer Science, pages 309–327. Springer, 2003.

[RȘ10] Grigore Roșu and Traian Florin Șerbănută. An overview of the K semantic framework. *The Journal of Logic and Algebraic Programming*, 79(6):397 – 434, 2010.

[San11] Davide Sangiorgi. *Introduction to Bisimulation and Coinduction*. Cambridge University Press, 2011.

[SBB11] Filip Sieczkowski, Magorzata Biernacka, and Dariusz Biernacki. Automating derivations of abstract machines from reduction semantics:. In Jurriaan Hage and Marco T. Morazán, editors, *IFL'10*, volume 6647 of *LNCS*, pages 72–88. Springer, 2011.

[Sch86] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. William C. Brown Publishers, Dubuque, IA, USA, 1986.

[SDM$^+$13] Ilya Sergey, Dominique Devriese, Matthew Might, Jan Midtgaard, David Darais, Dave Clarke, and Frank Piessens. Monadic abstract interpreters. *SIGPLAN Not.*, 48(6):399–410, June 2013.

[Ser12] Ilya Sergey. *Operational Aspects of Type Systems*. PhD thesis, KU Leuven, 2012.

[SF90] Dorai Sitaram and Matthias Felleisen. Control delimiters and their hierarchies. *Lisp Symb. Comput.*, 3(1):67–99, May 1990.

[Sie13] Jeremy Siek. Type safety in three easy lemmas, May 2013. `http://siek.blogspot.co.uk/2013/05/type-safety-in-three-easy-lemmas.html`.

[Sim14] Axel Simon. Deriving a complete type inference for Hindley-Milner and vector sizes using expansion. *Sci. Comput. Program.*, 95:254–271, 2014.

[SNO$^+$10] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strnisa. Ott: Effective tool support for the working semanticist. *J. Funct. Program.*, 20(1):71–122, 2010.

[SS71] Dana Scott and Christopher Strachey. Toward a mathematical semantics for computer languages. Programming Research Group Technical Monograph PRG-6, Oxford Univ. Computing Lab., 1971.

[SS75] Gerald J. Sussman and Guy L. Steele, Jr. An interpreter for extended lambda calculus. Technical report, MIT, 1975.

[SS13] Christopher Schwaab and Jeremy G. Siek. Modular type-safety proofs in Agda. In *PLPV'13*, pages 3–12, New York, NY, USA, 2013. ACM.

[STM16] Neil Sculthorpe, Paolo Torrini, and Peter D. Mosses. A modular structural operational semantics for delimited continuations. In *2015 Workshop on Continuations*, Electronic Proceedings in Theoretical Computer Science. 2016. To appear.

[Swi08] Wouter Swierstra. Data types á la carte. *J. Funct. Program.*, 18(4):423–436, July 2008.

[SWM00] Frederick Smith, David Walker, and Gregory Morrisett. Alias types. In *ESOP'00*, volume 1782 of *LNCS*, pages 366–381. Springer, 2000.

[Sym12] Don Syme. *The F# 3.0 Language Specification*. Microsoft Corporation, 2012.

[TA03] Andrew Tolmach and Sergio Antoy. A monadic semantics for core Curry. *ENTCS*, 86(3):16 – 34, 2003. WFLP'03.

[Tai67] W. W. Tait. Intensional interpretations of functionals of finite type I. *J. Symbolic Logic*, 32(2):198–212, 06 1967.

[TDD12] Pierre-Nicolas Tollitte, David Delahaye, and Catherine Dubois. Producing certified functional code from inductive specifications. In Chris Hawblitzel and Dale Miller, editors, *CPP'12*, volume 7679 of *LNCS*, pages 76–91. Springer, 2012.

[TJ94] Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. *Inf. Comput.*, 111(2):245–296, 1994.

[Tof90] Mads Tofte. Type inference for polymorphic references. *Inf. Comput.*, 89(1):1–34, September 1990.

[TP97] Daniele Turi and Gordon Plotkin. Towards a mathematical operational semantics. In *LICS'97*, pages 280–291. IEEE, 1997.

[TS15] Paolo Torrini and Tom Schrijvers. Reasoning about modular datatypes with mendler induction. In Ralph Matthes and Matteo Mio, editors, *FICS'15*, volume 191 of *EPTCS*, pages 143–157. Open Publishing Association, 2015.

[TT97] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[Uus13] Tarmo Uustalu. Coinductive big-step semantics for concurrency. In Nobuko Yoshida and Wim Vanderbauwhede, editors, *PLACES'13*, volume 137 of *EPTCS*, pages 63–78, 2013.

[vG04] Rob Jan van Glabbeek. The meaning of negative premises in transition system specifications {II}. *The Journal of Logic and Algebraic Programming*, 60–61:229–258, 2004.

[VHM10] David Van Horn and Matthew Might. Abstracting abstract machines. In *ICFP'10*, pages 51–62. ACM, 2010.

[Vis97] Eelco Visser. *Syntax definition for language prototyping*. PhD thesis, University of Amsterdam, 1997.

[VJ95] Jerome Vouillon and Pierre Jouvelot. Type and effect systems via abstract interpretation. Technical Report A/273/CRI, Ecole des Mines, Paris, 1995.

[VNV15] Vlad A. Vergu, Pierre Neron, and Eelco Visser. Dynsem: A DSL for dynamic semantics specification. In *RTA'15*, pages 365–378, 2015.

[VWT$^+$14] Eelco Visser, Guido Wachsmuth, Andrew P. Tolmach, Pierre Neron, Vlad A. Vergu, Augusto Passalaqua, and Gabriël Konat. A language designer's workbench: A one-stop-shop for implementation and verification of language designs. In Andrew P. Black, Shriram Krishnamurthi, Bernd Bruegge, and Joseph N. Ruskiewicz, editors, *Onward'14*, pages 95–111. ACM, 2014.

[Wad92] Philip Wadler. The essence of functional programming. In *POPL'92*, pages 1–14. ACM, 1992.

[Wad95] Philip Wadler. Monads for functional programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 24–52, London, UK, UK, 1995. Springer-Verlag.

[Wel02] J.B. Wells. The essence of principal typings. In Peter Widmayer, Stephan Eidenbenz, Francisco Triguero, Rafael Morales, Ricardo Conejo, and Matthew Hennessy, editors, *ICALP'02*, volume 2380 of *LNCS*, pages 913–925. Springer, 2002.

[WF94] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, 1994.

[Win93] Glynn Winskel. *The formal semantics of programming languages: an introduction*. MIT Press, 1993.

[Wri92] Andrew K. Wright. Typing references by effect inference. In *ESOP'92*, volume 582 of *LNCS*, pages 473–491. Springer, 1992.

[Wri95] Andrew K. Wright. Simple imperative polymorphism. *Lisp Symb. Comput.*, 8(4):343–355, December 1995.

[XSA01] Yong Xiao, Amr Sabry, and Zena M. Ariola. From syntactic theories to interpreters: Automating the proof of unique decomposition. *Higher Order Symbol. Comput.*, 14(4):387–409, December 2001.

[Zer13] Ian Zerny. *The Interpretation and Inter-derivation of Small-step and Big-step Specifications*. PhD thesis, Department of Computer Science, Aarhus University, 2013.