OPERATIONALLY-BASED MODELS OF HIGHER-ORDER IMPERATIVE PROGRAMMING LANGUAGES

A Thesis

Presented to the Faculty of the Graduate School

of the College of Computer Science

of Northeastern University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Gregory T. Sullivan [1]

College of Computer Science

Northeastern University

360 Huntington Avenue, 161CN

Boston, MA 02115, USA

gregs@ccs.neu.edu

`http://www.ccs.neu.edu/home/gregs/`

August 1997

College of Computer Science

Northeastern University

360 Huntington Avenue, 161CN

Boston, MA 02115, USA

gregs@ccs.neu.edu

`http://www.ccs.neu.edu/home/gregs/`, 1997

Abstract

We present a typed metalanguage with support for higher-order functions, a global store, and input-output. An operational, small-step semantics is given for the metalanguage, with the operations on the global store and for I/O restricted to continuation passing style. We develop, based solely on the operational semantics, a theory of term equivalence in the metalanguage, notably an Extensionality Theorem (also known as a context lemma). By considering equivalence classes of operationally equivalent metalanguage terms, we construct a term model which we then use as the target for denotational semantics of imperative programming languages. As an in-depth application of our methods, we prove correct a number of source-to-source transformations used in a compiler for Scheme.

Acknowledgements

I would like to thank Mitch Wand both for his prodigious knowledge and for his endless patience in attempting to impart some small portion of it to me.

While Mitch provided the academic enlightenment, my wonderful wife Ann gave me unflagging support, encouragement, and love. For all these things and more, I am eternally grateful. For providing joyous distraction, I thank my daughter Melissa, and for providing a powerful motivation to finish, I thank my son Jay, who was born the day after my defense.

I am grateful to my committee of, in addition to Mitch Wand: Carolyn Talcott, Will Clinger, and Karl Lieberherr. Their careful reading and thoughtful comments on earlier drafts have enormously improved both the accuracy and the readability of this thesis.

Thanks to all those that made my time at Northeastern as fruitful and enjoyable as it was, especially: Linda Seiter, Mike Cleary, Dave Gladstein, Paul Steckler, Nacho Silva-Lepe, Walter Hürsch, Ian Holland, Remy Évard, Deans Finkelstein and Chan, Richard Kelsey and Luc Longpré. Thanks also to fellow Boston semantics seminarians Bob Muller and Allyn Dimock.

And thanks to my family: Mom, Tom, Barb, and Beez, for humoring and supporting me throughout my life, and to my Dad, whose spirit still guides me.

Contents

List of Definitions

Chapter 1

Introduction

The following sections give a brief overview of the important concepts mentioned in the thesis abstract. We try to position both the problems we address, and also the methods we use, within the broader sphere of programming language research.

1.1 Functional Programming Languages

The unifying feature of programming languages described as "functional" is that functions are first-class entities: that is, functions may be passed as arguments to other functions and associated with variables. Another term for describing such programming languages is "higher-order." There seems little question that making functions first class adds power and expressiveness to a programming language. This extra functionality and abstraction also make the efficient implementation of functional programming languages more challenging. The goal of having well-performing implementations of functional programming languages has spawned an active area of research in analysis and compilation of functional programs.

An important aspect of functional programming languages is that they are amenable to formal analysis. The abstract, functional nature of these languages allows for the development of formal semantics and techniques and tools to reason about programs written in the language. Applications of these techniques include proving the correctness of source-to-source transformations (optimizations), compilers, garbage collectors, and sophisticated analyses, such as data and control flow analyses, which provide data to the other tools.

1.2 Imperative Features

The term "imperative" refers both to a set of programming language features, notably assignment and I/O, and also to a style of programming. Typical programs written in imperative programming languages such as Pascal or C have a function consisting of a sequence of statements, many of which assign values to variables, and finishing with a statement to return the value of the function. Another typically imperative feature, related to the notion of assignment, is that of a global store, or heap, and operations on it.

The "functional" languages which have enjoyed the most popularity – Scheme, other Lisp dialects, ML – have imperative features, and it is rare to find a large program in one of these languages that does not make use of some "impure" language features. From the imperative camp, object-oriented languages are evolving more high-level, abstract features reminiscent of the functional paradigm. The relatively new object-oriented programming language Java features garbage collection, which is a hallmark of functional languages.

1.3 Contribution

The work presented in this thesis is of interest for both its theoretical and its practical content.

We present a metalanguage, its formal syntax and semantics, and a model based on the operational semantics. The language and theory include the following features:

- higher-order functions,
- I/O operators,
- a global store (heap), and side-effecting operators (assignment),

- recursive types (as well as base, process, function, sum, and product types),
- possibly-infinite meaningful behavior,
- an Extensionality Theorem for the metalanguage,
- coinduction principles for reasoning both about possibly infinite types and about possibly non-terminating programs.

The noteworthy aspect of the preceding list is that we are able to combine many appealing but typically problematic features into a language that also supports powerful reasoning principles such as extensionality and coinduction. It is also notable that the entire theory is developed *operationally*, thus requiring a relatively small set of mathematical and logical concepts to understand.

After developing the syntax, semantics, and theory of the metalanguage, we then demonstrate the application of the theory to a number of examples drawn both from the literature and also from an actual compiler for the Scheme programming language. These applications include the first known proof of correctness of *assignment elimination*, an important transformation for Scheme compilers. The examples presented demonstrate that the framework we have set up is reasonable for application to actual optimizing compilers for modern higher-order functional languages with imperative features.

1.4 Related Research

There has been extensive research into issues related to imperative features of programming languages – particularly those having to do with assignment and side-effects.

One line of research has explored stack-based imperative languages such as Algol. Reynolds' "Essence of Algol" [Rey81] is similar syntactically to the metalanguages on which our own research focuses. The stack discipline of Algol has spawned much research, notably [Rey83, MS88, OT95]. The research presented in this thesis does not address stack discipline. The paper by Meyer and Sieber, [MS88], has been influential not only for its contribution to denotational semantics for Algol-like languages, but also for its presentation of a suite of "problematic" code fragments, referred to hereafter as the "Meyer-Sieber examples," which have been used as examples in numerous later papers by other authors.

Rather than starting with imperative languages, another line of research has been investigating adding store operations to functional languages. One approach to "cleanly" adding stores to functional languages has been to treat the store as a finite function and (implicitly) pass the store from function to function, exemplified by [Wad92, PJW93]. In [RV95], the purity of the functional sublanguage is preserved by saving the state before any side-effecting subprogram is executed, and restoring the state afterwards.

The Imperative Lambda Calculus (ILC) presented in [SRI91] uses the type system to restrict interaction between pure functional and imperative program phrases. Similar to the metalanguages presented in this thesis, the imperative operations operate in continuation passing style (CPS).

A few researchers have identified the concept of "names" as central to reasoning about assignment, though with widely varying results. Odersky [Ode94] writes:

> In a sense, names are the greatest common denominator of all programming languages that are not purely functional.

The $\lambda\nu$ calculus presented in [Ode94] is a purely syntactic, conservative extension of the (untyped) λ-calculus [Bar81]. The new syntactic elements, names, have dynamic scope and can be compared for equality.

The nu-calculus of Pitts and Stark [PS93] takes a similar approach, extending the (simply typed) λ-calculus with names and a test for name equality. The operational semantics of the nu-calculus, unlike that of the $\lambda\nu$ calculus, allows names to escape the local scope in which they were declared. The difference between these two "calculi of names" is demonstrated by their different behavior on the term $\nu n.n$. Odersky's $\lambda\nu$ calculus takes no steps on the term, whereas in the nu-calculus of Pitts and Stark, the term allocates a fresh name and returns it. The nu-calculus semantics, where the ν constructor has "run-time" reduction behavior, is closer to the semantics for our metalanguage, and has more of the flavor of memory allocation operators in actual programming languages. Pitts and Stark show that the nu-calculus, even without a global store or mutation, is not extensional – terms that give the same result in all applicative contexts may be differentiated by non-applicative contexts.

In [RP95], Ritter and Pitts describe a translation between Standard ML and a λ-calculus with reference types. They use Howe's method [How96] to show their translation fully abstract. Our metalanguage also has reference types.

Research by Talcott et al., especially [HMST95, MT91] overlaps quite a bit with our own. Like us, they use only operational semantics to develop a robust theory of a language with side-effects and a global store. Most differences between our research and theirs stem from different objectives: Talcott et al. study a sort of "minimal imperative language" which reflects the call-by-value, non-extensional nature of languages such as Scheme [IEE91] or ML [MTH89], whereas we have designed our metalanguage as the target of a translation from "real" programming languages and as the basis for a term model. As in Felleisen et al. [FH92], the language presented by Talcott et al. uses a syntactic representation of state, and syntactic manipulations of the store and store operations suffuse both the semantics and the proofs. Our metalanguage specifies stores as finite functions with no particular syntactic representation. An important theorem in Talcott et al.'s work is their *ciu theorem*, which states that terms are equivalent in all *reduction contexts*, à la [FF86], iff they are contextually equivalent (equivalent in all contexts). Their ciu theorem plays the same fundamental role in developing a theory for their language as does our Extensionality Theorem in developing the theory for our metalanguage. The contexts required by our Extensionality Theorem are somewhat more simple, but those differences from Talcott et al.'s ciu theorem are the result of the languages in question being designed for different purposes. The language presented in Talcott et al. is untyped and reduction is call-by-value, whereas our metalanguage has types and and supports full β-reasoning. In [HMST95], Honsell, Mason, Smith, and Talcott extend their results to a logic with which they verify the Meyer-Sieber examples, which we also prove using our setup.

The mathematical and logical tools used in this thesis draw from many sources.

Observational or contextual equivalence for programming languages was introduced by [Plo77], adapting the work of Morris [Mor68] for the λ-calculus. Also in [Plo77], Plotkin gives the syntax and typing rules of PCF, which our metalanguage extends. Operationally-based term models were introduced in [TW96], for the (untyped) λ-calculus with I/O (but no heap).

Our Extensionality Theorem corresponds to what is often called a *context lemma*. A well-known example of a context lemma is Milner's context lemma for PCF and other typed lambda calculi in [Mil77]. Milner's context lemma for PCF was proven operationally. Generalizations of Milner's results are discussed in [JM91].

Bisimulation was introduced by [HM85]. In an untyped setting, the Lazy Lambda-Calculus paper by Abramsky [Abr90] presents a context lemma as well as defining the notion of applicative bisimulation. Abramsky's work, while ground breaking, requires a strong background in domain theory to follow adequately, and his observables are very different from ours. See also [EHR91] and [Plo75].

One standard approach to proving that applicative equivalence coincides with contextual equivalence is to show first that applicative equivalence is a congruence – that is, closed under the term constructors of the language. Techniques for proving the congruence of bisimulation relations appear in [How89] and are further developed in [How96] and [How95]. We do not directly use Howe's method for proving congruence, but his work has been very influential in this area of research.

Andrew Pitts [Pit99] and his colleagues have done much to advance the use of techniques such as bisimulation and coinduction to develop theories based on operational semantics. In [Gor94b], Gordon investigates another typically imperative feature, I/O, using the same core techniques, such as bisimulation and coinduction, as we do. These techniques are generalized and presented more tutorially in [Gor95]. The languages explored by Gordon include recursive types in addition to I/O.

The general idea of a relation preserved under computation recurs often in the literature. It was used to prove program equivalence in [MT91, WO92, ORW95]; [Sta94] uses a similar notion extended to become a logical relation.

Many of the optimizations we prove correct later in the thesis are from Clinger and Hansen's "Twobit" Scheme compiler and are discussed in [CH94]. A good introduction to the implementation of functional programming languages can be found in [PJ87], although the languages discussed there do not support side-effects. Assignment elimination was introduced in the Orbit compiler [KKR+86] and used in [KH89, CH94]. It was omitted from [GSR95] because the authors were unable to prove its correctness [Ram96].

1.4.1 On Reading the Proofs

The proofs are presented in a stylized manner which takes some getting used to. We use the `pf` style provided by Lesley Lamport and discussed in [Lam93].

Suppose that in order to prove a proposition one needs three subgoals to be proven first. The proof structure would be:

1. first subgoal
2. second subgoal
3. third subgoal

Q.E.D. use first three subgoals to finish

The proofs of each subgoal might involve sequences of steps also. Suppose that two steps are needed to prove the first subgoal. Our proof structure then looks like:

1. first subgoal
   - 1.1. first step in subproof
   - 1.2. second step in subproof

Q.E.D. tie up proof of first subgoal

2. second subgoal

3. third subgoal

Q.E.D.

In general, a proof step numbered x.y.z. labels the z^{th} subgoal in the proof of the y^{th} subgoal of the proof of the x^{th} step in the proof of the top-level proposition. We use a more compact numbering scheme, with proof steps labelled, for example, $\langle 5\rangle 2$. A step labelled $\langle x\rangle y$. presents the statement of the y^{th} step in the proof of a subgoal nested x layers deep.

If our proof has the following structure:

1. first subgoal
    1.1. first step in subproof
    1.2. second step in subproof
        1.2.1. first step in subsubproof
        1.2.2. second step in subsubproof
            1.2.2.1. first step in subsubsubproof
            Q.E.D.
        1.2.3. third step in subsubproof
        Q.E.D.
    1.3. third step in subproof
        1.3.1. first step in subsubproof
        Q.E.D.
    Q.E.D.
2. second subgoal
3. third subgoal

Q.E.D.

then the compacted labels look like the following:

$\langle 1\rangle 1$. first subgoal

⟨2⟩1. first step in subproof

⟨2⟩2. second step in subproof

⟨3⟩1. first step in subsubproof

⟨3⟩2. second step in subsubproof

⟨4⟩1. first step in subsubsubproof

Q.E.D.

⟨3⟩3. third step in subsubproof

Q.E.D.

⟨2⟩3. third step in subproof

⟨3⟩1. first step in subsubproof

Q.E.D.

Q.E.D.

⟨1⟩2. second subgoal

⟨1⟩3. third subgoal

Q.E.D.

Thus the proof step labels become shorter, but they are no longer unique. This loss of uniqueness is acceptable, because the subproof steps can logically never be used outside their "scope."

One further note on the labels in the `pf` style. While the reader may be used to the string Q.E.D. delineating the end of the proof, in the `pf` style, Q.E.D. labels the final subgoal of a proof, which may in turn involve a sub-proof which ties together any preceding steps in order to entail the current (sub)goal.

1.4.2 Some History

Originally, we proved substitution and context lemmas for an untyped version of the metalanguage. This, however, was not sufficient to prove equivalences for a source language such as Scheme. Adding the types to the metalanguage, and thence to the translations from

Scheme, restricted the possible initial continuations enough to get us the desired equivalences. Section 4.5 works out a simple example of a pair of terms which are equivalent under a typed translation but not under an untyped one.

Also, some early attempts tried making location constants related if they referenced related terms in the store (rather than requiring simple equality as we currently do). This added too much generality to the setup, and we were not able to prove, for example, a decent substitution lemma.

1.5 Outline

Chapter 2 presents the syntax for our metalanguage and its operational semantics. A term model is then developed, and various reasoning principles are derived – notably an extensionality theorem. Chapter 3 uses the theory of the metalanguage to prove a number of equivalences from the literature. Chapter 4 introduces a translation from the programming language Scheme to our metalanguage. Chapter 5 defines and then proves correct the *assignment elimination* optimization used in many Scheme compilers. Chapter 6 defines and proves correct the *lambda lifting* transformation, used in compilers for Scheme and other functional languages. We conclude in Chapter 7 with a discussion of open problems and future work.

Chapter 2

An Operationally-Based Term Model

We present the syntax and operational semantics of a typed metalanguage. The metalanguage extends the type system of PCF by adding recursive, reference, sum and product types and extends the operations of PCF by adding I/O and store operators. As in PCF, the rules for evaluation of terms are lazy (as opposed to call-by-value).

We turn the operational semantics of the metalanguage into a term model for a typed lambda calculus by considering equivalence classes of terms with the same observable behavior. The observable behavior we consider is interaction via the I/O operators in the metalanguage.

Finally, we prove a number of useful properties of the metalanguage. In particular, we show that the metalanguage is extensional: in order to prove two terms are contextually equivalent, one need only consider a restricted set of "extensional" contexts.

We are able to add imperative operations to PCF while maintaining a nice operational theory by restricting the evaluation of the I/O and store operators to leftmost-outermost position. The I/O and store operators are in continuation passing style (CPS), and therefore the effects of those operators can only be passed along to their continuations.

2.1 Syntax

The types in the metalanguage, ranged over by τ, are potentially infinite trees which respect the following structure:

$$o ::= \mathit{unit} \mid \mathit{bool} \mid \mathit{int} \mid \mathit{ref}(\tau) \mid \mathit{pr}$$
$$\tau ::= o \mid \tau{\to}\tau \mid \tau{+}\tau \mid \tau{\times}\tau$$

The base types o include *int* for integers, *bool* for booleans, $\mathit{ref}(\tau)$ for locations with contents of type τ, and *unit* for the singleton type. Type *pr* is for processes, which may do I/O.

We allow infinite types, so we will need to define operations on types coinductively instead of inductively. We write τ^* for the infinite type

$$\mathit{unit} + (\tau \times (\mathit{unit} + (\tau \times \ldots)))$$

of lists of elements of type τ.

The syntax of our metalanguage is given by the following grammar:

$$M ::= x \mid n \mid c \mid l^\tau \mid \lambda x{:}\tau.M \mid MM$$

with x ranging over a countably infinite set of variables, *Vars*, and l ranging over a countably infinite set of location constants, *Locs*. Locations l^τ are annotated with types and are assigned type $\mathit{ref}(\tau)$.

The types of the constants c of the metalanguage are given in Figure 1.

A *substitution* σ is a finite map from variables to metalanguage terms. If $[x \mapsto M] \in \sigma$, then applying the substitution to a term N, denoted $N\sigma$, does any α-renaming necessary to avoid capture of free variables in M by binders (λ-expressions) in N.

A *typing judgement* $\Gamma \vdash M{:}\tau$ is a triple consisting of a type environment Γ, a metalanguage term M, and a type τ. Type environments map variables to types. The judgement $\Gamma \vdash M{:}\tau$ can be read as stating that type environment Γ proves that term M has type τ if every

$$\forall \text{ types } \alpha, \beta, \gamma, \tau,$$

$\mathsf{pred}:int\to int,\ \mathsf{succ}:int\to int,$	predecessor and successor
$\mathsf{true}:bool,\ \mathsf{false}:bool,$	boolean constants
$\mathsf{if}^\tau:bool\to\tau\to\tau\to\tau,$	conditional
$\mathsf{zero?}:int\to bool,$	true if int is zero
$\mathsf{Y}^\tau:(\tau\to\tau)\to\tau,$	fixed point combinator
$\mathsf{pair}_{\alpha,\beta}:\alpha\to\beta\to(\alpha\times\beta)$	pair constructor
$\pi_{1\alpha,\beta}:(\alpha\times\beta)\to\alpha$	first projection
$\pi_{2\alpha,\beta}:(\alpha\times\beta)\to\beta$	second projection
$\mathsf{1}:unit$	single element of unit type
$\mathsf{inl}_{\alpha,\beta}:\alpha\to(\alpha+\beta)$	left injection
$\mathsf{inr}_{\alpha,\beta}:\beta\to(\alpha+\beta)$	right injection
$\mathsf{case}^\gamma_{\alpha,\beta}:(\alpha+\beta)\to(\alpha\to\gamma)\to(\beta\to\gamma)\to\gamma$	sum unboxing
$\mathsf{write}:int\to pr\to pr,$	write an integer to stdout
$\mathsf{read}:(int\to pr)\to pr,$	read integer from stdin
$\mathsf{stop}:pr,\ \mathsf{stopped}:pr,$	halt
$\mathsf{new}^\tau:\tau\to(ref(\tau)\to pr)\to pr,$	get fresh loc, initialize to 1st arg
$\mathsf{deref}^\tau:ref(\tau)\to(\tau\to pr)\to pr,$	dereference a location
$\mathsf{update}^\tau:ref(\tau)\to\tau\to pr\to pr,$	update a location
$\mathsf{eq?}_{loc}{}^\tau:ref(\tau)\to ref(\tau)\to bool$	compare locations for equality

Figure 1: Metalanguage Constants

$$\Gamma \vdash x{:}\Gamma(x) \qquad \Gamma \vdash n{:}int \qquad \Gamma \vdash l^\tau{:}ref(\tau)$$

$$\Gamma \vdash c{:}\tau \text{ where } c{:}\tau \text{ from Figure 1}$$

$$\frac{\Gamma \vdash M{:}\tau'{\rightarrow}\tau,\ \Gamma \vdash N{:}\tau'}{\Gamma \vdash MN{:}\tau}$$

$$\frac{\Gamma, x{:}\tau' \vdash M{:}\tau}{\Gamma \vdash (\lambda x{:}\tau'.M){:}\tau'{\rightarrow}\tau}$$

Figure 2: Typing Rules

free variable $x \in fv(M)$ is assigned type $\Gamma(x)$. If $\Gamma \vdash M{:}\tau$, then a *closing substitution* σ maps the free variables x_i of M to closed terms P_i respecting the type environment Γ – i.e. if $\Gamma(x_i) = \tau_i$, then $\emptyset \vdash P_i{:}\tau_i$. Thus a closing substitution σ instantiates all free variables: $fv(M\sigma) = \emptyset$. For closed well-typed terms, we often write $M{:}\tau$ rather than $\emptyset \vdash M{:}\tau$.

The rules for deriving type judgements are standard, and are given in Figure 2 for completeness.

Programs are closed terms of type pr.

Contexts are generated by the same grammar as terms, but with an additional production for a typed hole:

$$C ::= [-_\tau] \mid x \mid n \mid c \mid l^\tau \mid \lambda x{:}\tau.C \mid CC$$

A context $C[-_\tau]$ *respects* a type assignment Γ if for every variable binding $x{:}\tau'$ in effect at the hole in C, $\Gamma(x) = \tau'$.

A *state*, denoted $\langle\langle M\ ;\ \Sigma\rangle\rangle$, is a term-store pair where M is a program (a closed term of type pr) and store Σ is a finite map from locations to terms. A store must be well-formed in that all locations l^τ in the domain of Σ must map to closed terms of type τ.

2.2 Semantics of the Metalanguage

We define the operational semantics of the metalanguage in three steps. The reduction semantics of the PCF fragment of the metalanguage defines a relation $\rightarrow_{pcf}$ between closed well-typed terms. The reduction relation $\rightarrow_{sto}$ is defined on states, includes PCF reduction within term components, and gives transition rules for states with store operations in head position. Finally, we define a labeled transition relation, $\xrightarrow{\alpha}_{io}$, between states which reduce under $\rightarrow^*_{sto}$ to states headed by I/O operations.

Figure 3 presents the standard transition rules for the PCF fragment of the metalanguage, augmented with operations on sums and products. PCF reduction is only defined on well-typed terms – thus we omit the type annotations on the operators and locations. Substitution in the β rule is careful not to capture free variables in N, as usual. We define all of the redexes using the $\rightarrow_1$ relation, and then use one rule of inference to define $\rightarrow_{pcf}$ as reduction in any context. Some presentations of PCF (e.g. [Gun92]) do not allow reduction under a λ or within the operand of an application. We choose to allow reduction in all contexts, often referred to as "full β," and we will later appeal to the Church-Rosser property of PCF to prove a standardization theorem. This choice is consistent with our goal of using the metalanguage to reason about other programs – not as an example programming language itself. Note that reduction is lazy – operands do not need to reduce to a value before an operator transition is taken.

Recursion is supported by the Y operator. As a simple example of its use, following is the definition of a *plus* function:

$$\begin{aligned} plus \stackrel{def}{=} (&\mathsf{Y}^{int\rightarrow int\rightarrow int}(\lambda P{:}int\rightarrow int\rightarrow int.\lambda x{:}int.\lambda y{:}int. \\ &\quad \mathsf{if}(\mathsf{zero?}\,y)x(P\ (\mathsf{succ}\,x)\ (\mathsf{pred}\,y))) \end{aligned}$$

The term *plus* has type $int\rightarrow int\rightarrow int$, as it should.

Figure 4 defines the transition relation $\rightarrow_{sto}$ between states. The first rule incorporates PCF reduction into $\rightarrow_{sto}$; if the term component of a state may take a single PCF transition, the state pair may take the corresponding transition. The new operator allocates a new

$$
\begin{array}{ll}
(\lambda x.M)N \rightarrow_1 M[N/x] & (\beta) \\
\mathsf{Y}M \rightarrow_1 M(\mathsf{Y}M) & \\
\mathsf{pred}\, 0 \rightarrow_1 0 & \\
\mathsf{pred}\, n \rightarrow_1 n',\ \text{if } n > 0 \text{ and } n = n' + 1 & \\
\mathsf{zero?}\, 0 \rightarrow_1 \mathsf{true} & \\
\mathsf{zero?}\, n \rightarrow_1 \mathsf{false},\ \text{if } n \neq 0 & \\
\mathsf{if}\, \mathsf{true}\, MN \rightarrow_1 M & \\
\mathsf{if}\, \mathsf{false}\, MN \rightarrow_1 N & \\
\mathsf{eq?}_{loc}\, l_1 l_2 \rightarrow_1 \mathsf{true},\ \text{if } l_1 = l_2 & \\
\mathsf{eq?}_{loc}\, l_1 l_2 \rightarrow_1 \mathsf{false},\ \text{if } l_1 \neq l_2 &
\end{array}
$$

$$
\begin{array}{l}
\pi_1(\mathsf{pair}\, MN) \rightarrow_1 M \\
\pi_2(\mathsf{pair}\, MN) \rightarrow_1 N \\
\mathsf{case}(\mathsf{inl}\, M)NP \rightarrow_1 (NM) \\
\mathsf{case}(\mathsf{inr}\, M)NP \rightarrow_1 (PM)
\end{array}
$$

$$
\frac{M \rightarrow_1 N}{C[M] \rightarrow_{pcf} C[N]}
$$

Figure 3: PCF Reduction Rules

location in the store, initializes the fresh location to its first argument, and invokes its second argument with the new location. The deref operator invokes its second argument with the value in the store at the location given by its first argument. The update operator destructively updates the location given by its first argument to be the term given by its second argument and then invokes its third argument. Like PCF reduction, store operations are "lazy" in that store update is done with terms that are not necessarily in normal form. The reflexive, transitive closure of $\rightarrow_{sto}$ is denoted by $\rightarrow^*_{sto}$. If a state $\langle\!\langle M \,;\, \Sigma \rangle\!\rangle$ takes n steps to state $\langle\!\langle M' \,;\, \Sigma' \rangle\!\rangle$, we write $\langle\!\langle M \,;\, \Sigma \rangle\!\rangle \overset{n}{\rightarrow}_{sto} \langle\!\langle M' \,;\, \Sigma' \rangle\!\rangle$. The domain (a set of locations) and range (a set of terms) of a store Σ are denoted by $dom(\Sigma)$ and $rng(\Sigma)$, respectively.

The meta-operation $locs(M)$ simply returns any locations referenced in M and is easily defined via structural induction:

$$
\begin{array}{l}
M \rightarrow_{pcf} M' \Rightarrow \langle\!\langle M \;;\; \Sigma \rangle\!\rangle \rightarrow_{sto} \langle\!\langle M' \;;\; \Sigma \rangle\!\rangle \\
\langle\!\langle \mathsf{new}^{\tau}\; M\; N \;;\; \Sigma \rangle\!\rangle \rightarrow_{sto} \langle\!\langle N\; l^{\tau} \;;\; \Sigma[l^{\tau} \mapsto M] \rangle\!\rangle, \quad \text{if} \\
\qquad\qquad\qquad\qquad l^{\tau} \notin dom(\Sigma) \cup locs(rng(\Sigma), M, N) \\
\langle\!\langle \mathsf{deref}^{\tau}\; l^{\tau}\; M \;;\; \Sigma \rangle\!\rangle \rightarrow_{sto} \langle\!\langle M\; N \;;\; \Sigma \rangle\!\rangle \qquad \text{if } \Sigma(l^{\tau}) = N \\
\langle\!\langle \mathsf{update}^{\tau}\; l^{\tau} M\; N \;;\; \Sigma \rangle\!\rangle \rightarrow_{sto} \langle\!\langle N \;;\; \Sigma[l^{\tau} \mapsto M] \rangle\!\rangle \text{ if } l^{\tau} \in dom(\Sigma)
\end{array}
$$

Figure 4: Operational Semantics for Store Operators

Definition 2.2.1

1. *if* $M = l$, $locs(M) = \{l\}$,
2. *if* $M = (M_1\; M_2)$, $locs(M) = locs(M_1) \cup locs(M_2)$, *and*
3. *if* $M = (\lambda x{:}\tau.M_1)$, $locs(M) = locs(M_1)$,
4. *otherwise,* $locs(M) = \emptyset$.

When applied to a set S of terms,

$$locs(S) = \bigcup_{M \in S} locs(M)$$

Also,

$$locs(M_1, \ldots, M_n) = locs(M_1) \cup \ldots \cup locs(M_n)$$

A labeled transition system $\xrightarrow{\alpha}_{io}$ for the I/O operators is defined in Figure 5. The write operator writes its integer argument and invokes its continuation. The read operator reads an integer and invokes its continuation with the number read. The stop operator takes a $\surd$ transition to the stopped state $\langle\!\langle \mathsf{stopped} \;;\; \Sigma' \rangle\!\rangle$, from which there are no transitions.

Note that the write I/O operator, and the deref and update store operators, are strict in their first arguments.

$$\frac{\langle\!\langle M\ ;\Sigma\rangle\!\rangle \rightarrow^{*}_{sto} \langle\!\langle \mathsf{write}\, n\ M'\ ;\Sigma'\rangle\!\rangle}{\langle\!\langle M\ ;\Sigma\rangle\!\rangle \xrightarrow{!n}_{io} \langle\!\langle M'\ ;\Sigma'\rangle\!\rangle} \qquad \frac{\langle\!\langle M\ ;\Sigma\rangle\!\rangle \rightarrow^{*}_{sto} \langle\!\langle \mathsf{read}\ M'\ ;\Sigma'\rangle\!\rangle}{\langle\!\langle M\ ;\Sigma\rangle\!\rangle \xrightarrow{?n}_{io} \langle\!\langle M'\ n\ ;\Sigma'\rangle\!\rangle}$$

$$\frac{\langle\!\langle M\ ;\Sigma\rangle\!\rangle \rightarrow^{*}_{sto} \langle\!\langle \mathsf{stop}\ ;\Sigma'\rangle\!\rangle}{\langle\!\langle M\ ;\Sigma\rangle\!\rangle \xrightarrow{\surd}_{io} \langle\!\langle \mathsf{stopped}\ ;\Sigma'\rangle\!\rangle}$$

Figure 5: Semantics for I/O Operators

2.3 State Equivalence – Bisimulation

A standard notion of equivalence for labeled transition systems is that of *strong bisimulation*. For a good introduction to bisimulation, see [Mil90]. It is interesting to note that labeled transition systems and various notions of bisimulation have historically been the basic tools used to reason about concurrent processes and languages. Researchers into sequential languages, notably Abramsky [Abr90], Pitts [PS93, RP95] and Gordon [Gor94b, Gor94a, Gor95], have adapted these techniques for sequential, and usually deterministic, settings.

Two states are strongly bisimilar if they can always take the same sequence of labeled transitions. We say that a relation between states is a strong bisimulation if whenever two states are in the relation, they can take the same action (labeled transition) and stay in the relation. In the following definitions, we let α range over the set of actions, $Act = \{?n, !n, \surd\}$, pronounced "read n", "write n", and "halt," for all integers n.

Definition 2.3.1 ($[-]_{bisim}$) *Given R a relation on states, we define a relation $[R]_{bisim}$ on states as follows.*

$$\langle\!\langle M\ ;\Sigma_M\rangle\!\rangle\ [R]_{bisim}\ \langle\!\langle N\ ;\Sigma_N\rangle\!\rangle$$

iff for all $\alpha \in Act$,

1. *if* $\langle\langle M\ ;\Sigma_M\rangle\rangle \xrightarrow{\alpha}_{io} \langle\langle M'\ ;\Sigma_{M'}\rangle\rangle$, *then* $\exists N', \Sigma_{N'}$ s.t. $\langle\langle N\ ;\Sigma_N\rangle\rangle \xrightarrow{\alpha}_{io} \langle\langle N'\ ;\Sigma_{N'}\rangle\rangle$ *and* $\langle\langle M'\ ;\Sigma_{M'}\rangle\rangle\ R\ \langle\langle N'\ ;\Sigma_{N'}\rangle\rangle$

2. *if* $\langle\langle N\ ;\Sigma_N\rangle\rangle \xrightarrow{\alpha}_{io} \langle\langle N'\ ;\Sigma_{N'}\rangle\rangle$, *then* $\exists M', \Sigma_{M'}$ s.t. $\langle\langle M\ ;\Sigma_M\rangle\rangle \xrightarrow{\alpha}_{io} \langle\langle M'\ ;\Sigma_{M'}\rangle\rangle$ *and* $\langle\langle M'\ ;\Sigma_{M'}\rangle\rangle\ R\ \langle\langle N'\ ;\Sigma_{N'}\rangle\rangle$

The symmetry in the above definition is clear, and we will usually focus only on one direction, *simulation*:

Definition 2.3.2 ($[-]_{sim}$ – simulation generator) *Given R a relation on states, we define a relation $[R]_{sim}$ on states as follows.*

$$\langle\langle M\ ;\Sigma_M\rangle\rangle\ [R]_{sim}\ \langle\langle N\ ;\Sigma_N\rangle\rangle$$

iff for all $\alpha \in Act$,

if $\langle\langle M\ ;\Sigma_M\rangle\rangle \xrightarrow{\alpha}_{io} \langle\langle M'\ ;\Sigma_{M'}\rangle\rangle$, *then* $\exists N', \Sigma_{N'}$ s.t.
$\langle\langle N\ ;\Sigma_N\rangle\rangle \xrightarrow{\alpha}_{io} \langle\langle N'\ ;\Sigma_{N'}\rangle\rangle$ *and*
$\langle\langle M'\ ;\Sigma_{M'}\rangle\rangle\ R\ \langle\langle N'\ ;\Sigma_{N'}\rangle\rangle$

We can unwind the definition of the transition relation $\xrightarrow{\alpha}_{io}$ to get an equivalent definition of the simulation generator $[-]_{sim}$ in terms of the store semantics transition relation $\rightarrow_{sto}$, as follows.

Lemma 2.3.3 *Given R a relation on states,*
$\langle\langle M\ ;\Sigma_M\rangle\rangle\ [R]_{sim}\ \langle\langle N\ ;\Sigma_N\rangle\rangle$ *iff*

1. *if* $\langle\langle M\ ;\Sigma_M\rangle\rangle \rightarrow^*_{sto} \langle\langle \mathsf{write}\, nM'\ ;\Sigma'_M\rangle\rangle$, *then* $\exists N', \Sigma'_N$ s.t.
$\langle\langle N\ ;\Sigma_N\rangle\rangle \rightarrow^*_{sto} \langle\langle \mathsf{write}\, nN'\ ;\Sigma'_N\rangle\rangle$ *and* $\langle\langle M'\ ;\Sigma'_M\rangle\rangle\ R\ \langle\langle N'\ ;\Sigma'_N\rangle\rangle$,

2. *if* $\langle\langle M\ ;\Sigma_M\rangle\rangle \rightarrow^*_{sto} \langle\langle \mathsf{read}\, M'\ ;\Sigma'_M\rangle\rangle$, *then* $\exists N', \Sigma'_N$ s.t.
$\langle\langle N\ ;\Sigma_N\rangle\rangle \rightarrow^*_{sto} \langle\langle \mathsf{read}\, N'\ ;\Sigma'_N\rangle\rangle$ *and* $\forall n.\langle\langle M'\, n\ ;\Sigma'_M\rangle\rangle\ R\ \langle\langle N'\, n\ ;\Sigma'_N\rangle\rangle$,

3. *if* $\langle\!\langle M \; ; \Sigma_M \rangle\!\rangle \rightarrow^*_{sto} \langle\!\langle \mathsf{stop} \; ; \Sigma'_M \rangle\!\rangle$, *then* $\exists \Sigma'_N$ s.t.
$\langle\!\langle N \; ; \Sigma_N \rangle\!\rangle \rightarrow^*_{sto} \langle\!\langle \mathsf{stop} \; ; \Sigma'_N \rangle\!\rangle$ *and* $\langle\!\langle \mathsf{stopped} \; ; \Sigma'_M \rangle\!\rangle \; R \; \langle\!\langle \mathsf{stopped} \; ; \Sigma'_N \rangle\!\rangle$.

PROOF:

By unwinding the definition of $\xrightarrow{\alpha}_{io}$.

Most of our proofs of simulation will be based on Lemma 2.3.3. Actually, the cases for each I/O operator are always quite similar, so most proofs will do only one case for reduction under $\rightarrow_{sto}$ to a "generic" I/O operator, io, which is considered to take a single continuation term. The fact that the write operator requires its first argument to reduce to an integer first does not affect any of our proofs, as only PCF reductions are allowed in that position, and PCF reduction preserves simulation.

As with bisimulation relations, we call a binary relation between states a simulation if whenever two states are in the relation, they can take the same action and stay within the relation. More formally, a relation R is called *a simulation relation* iff $R \subseteq [R]_{sim}$.

The largest simulation relation is called, not surprisingly, *simulation*, and it is denoted by $\lesssim_{sim}$. We take membership in $\lesssim_{sim}$ to be our basic notion of (preordered) equivalence between states. The largest bisimulation relation, called *bisimulation*, is denoted $\approx_{sim}$.

Simulation, $\lesssim_{sim}$ is therefore the union of all simulation relations,

$$\lesssim_{sim} = \bigcup \{R \mid R \subseteq [R]_{sim}\}$$

which is easily shown to be the greatest fixed point of $[-]_{sim}$.

From this definition of $\lesssim_{sim}$ as the greatest fixed point of the operator $[-]_{sim}$ on relations, we get the following principle of coinduction:

$$R \subseteq [R]_{sim} \Rightarrow R \subseteq \lesssim_{sim}$$

This principle of coinduction forms the basis for most of our proofs of simulation.

2.3.1 Bisimulation in a Deterministic Setting

The reduction rules for our metalanguage are deterministic up to the choice of location constants returned by the `new` operator. In Section 2.5.4, we prove that metalanguage states that differ only up to a consistent renaming of locations (denoted α_{loc}) are bisimilar. When stating that our metalanguage is deterministic, then, we use the following definition of determinism:

Definition 2.3.4 (Deterministic Transition System) *A transition system* $\xrightarrow{\alpha}$ *is* deterministic *if, for any state* A*,*

$$\begin{aligned}&\text{if } A \xrightarrow{\alpha} B \text{ and } A \xrightarrow{\alpha} B', \text{ then}\\ &\quad B \approx_{sim} B'\end{aligned}$$

Given a deterministic transition system, one can prove bisimulation of two states using symmetric simulation subproofs:

Lemma 2.3.5 *For simulation* $\lesssim_{sim}$ *and bisimulation* $\approx_{sim}$ *defined based on a deterministic transition system,*

$$\begin{aligned}&\text{if } A \lesssim_{sim} B \text{ and } B \lesssim_{sim} A, \text{ then}\\ &\quad A \approx_{sim} B\end{aligned}$$

PROOF:

By coinduction.
LET: $S \stackrel{def}{=} \{(A, B) \mid A \lesssim_{sim} B \text{ and } B \lesssim_{sim} A\}$
We must show that $S \subseteq [S]_{bisim}$. That is, given $(A, B) \in S$, we must show:
1. If $A \xrightarrow{\alpha} A'$, then $\exists B'$ s.t. $B \xrightarrow{\alpha} B'$ and $A' \; S \; B'$, and
2. If $B \xrightarrow{\alpha} B'$, then $\exists A'$ s.t. $A \xrightarrow{\alpha} A'$ and $B' \; S \; A'$.
$\langle 1 \rangle$1. if $A \xrightarrow{\alpha} A'$, then $\exists B'$ s.t. $B \xrightarrow{\alpha} B'$ and $A' \; S \; B'$
PROOF:
  $\langle 2 \rangle$1. $\exists B'$ s.t. $B \xrightarrow{\alpha} B'$ and $A' \lesssim_{sim} B'$

By $(A,B) \in S \Rightarrow A \lesssim_{sim} B$.

$\langle 2 \rangle 2.$ $\exists A''$ s.t. $A \stackrel{\alpha}{\to} A''$ and $B' \lesssim_{sim} A''$

By $(A,B) \in S \Rightarrow B \lesssim_{sim} A$.

$\langle 2 \rangle 3.$ $A' \approx_{sim} A''$

By determinism.

$\langle 2 \rangle 4.$ $B' \lesssim_{sim} A'$

By $B' \lesssim_{sim} A'' \approx_{sim} A'$ and $(\lesssim_{sim} \circ \approx_{sim}) \subseteq \lesssim_{sim}$ (a simple coinduction).

$\langle 1 \rangle 2.$ if $B \stackrel{\alpha}{\to} B'$, then $\exists A'$ s.t. $A \stackrel{\alpha}{\to} A'$ and $B' \; S \; A'$

PROOF:

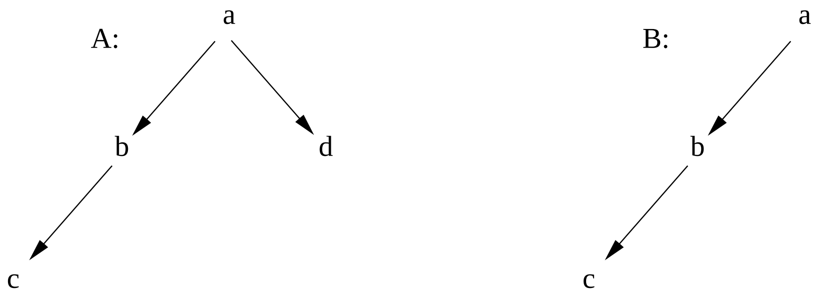
By the same line of reasoning as in the first case.

$\square$

Thus most of our proofs of bisimulation consist of two proofs of simulation. Furthermore, in most cases the two simulation proofs are nearly identical, and we will only present one direction in detail. Whenever the other simulation proof would be different (or impossible), we point that out.

It should be noted that in a **non**deterministic setting the preceding lemma does not hold.

Consider a nondeterministic transition system with a single action (so we omit the arrow labels in the diagrams) and two processes, A and B, with transition diagrams as follows:



The labels $\{a, b, c, d\}$ represent states. We show that $A \lesssim_{sim} B$ by defining the relation R on states,

$$R \stackrel{def}{=} \{(a, a), (b, b), (c, c), (d, b)\}$$

and then proving that $R \subseteq [R]_{sim}$. Likewise, we can show that $B \lesssim_{sim} A$ by defining the relation S on states,

$$S \stackrel{def}{=} \{(a, a), (b, b), (c, c)\}$$

and showing that $S \subseteq [S]_{sim}$.

However, clearly $A \not\approx_{sim} B$, because when A takes a transition to state d, B can only transition to state b. While state b can take all of the actions that d can (none), it is not true that d can take all of the actions that b can.

2.4 Term Equivalence

Bisimulation defines when two states (i.e. program-store pairs) have the same behavior. Often we are interested in comparing two terms for equivalence, and the two terms may not be programs. We use a standard notion of equivalence between terms, *contextual equivalence*, which formalizes when two terms are indistinguishable.

2.4.1 Contextual Equivalence

We say that two terms are *contextually equivalent* if they have the same I/O behavior when inserted into every possible program context. We define contextual equivalence on *open* terms, as contexts may capture free variables.

Definition 2.4.1 (contextual preorder, $\lesssim_{\forall C}$)

If $\Gamma \vdash M, N{:}\tau$*, then*

$$\Gamma \vdash M \lesssim_{\forall C} N{:}\tau$$

iff for all closing contexts $C[-_\tau]$ *of type pr that respect* Γ*, and for all stores* Σ*,*

$$\langle\!\langle C[M]\ ;\ \Sigma \rangle\!\rangle \lesssim_{sim} \langle\!\langle C[N]\ ;\ \Sigma \rangle\!\rangle$$

Contextual equivalence, $\cong_{\forall C}$, is the symmetrization of $\lesssim_{\forall C}$:

Definition 2.4.2 (contextual equivalence, $\cong_{\forall C}$)

If $\Gamma \vdash M, N{:}\tau$, *then*

$$\Gamma \vdash M \cong_{\forall C} N{:}\tau$$

iff for all closing contexts $C[-_\tau]$ *of type pr that respect* Γ, *and for all stores* Σ,

$$\langle\!\langle C[M]\ ;\ \Sigma \rangle\!\rangle \approx_{sim} \langle\!\langle C[N]\ ;\ \Sigma \rangle\!\rangle$$

The contextual preorder and contextual equivalence are, by definition, *congruences* – that is, both relations are preserved by all term constructors of the language.

Contextual equivalence is the relation we are most interested in between terms. However, the quantification over all contexts makes $\cong_{\forall C}$ difficult to establish directly, and we expend considerable effort developing methods to ease this proof burden. The most important tool is a "context lemma" which states that two terms are similar in a limited set of contexts if and only if they are contextually equivalent. We will call equivalence in this restricted class of contexts *extensional equivalence*.

2.4.2 Extensional Equivalence

For each type constructor $(\rightarrow, +, \times)$, there is a natural way of determining whether two terms of such a type "behave the same way". At function types $\alpha{\rightarrow}\beta$, two terms are extensionally equal if they take every term of type α to equivalent terms of type β. For product types, both projections should be equivalent, and for sum types, the `case` operator should produce equivalent terms. We can view these type-based contexts (application, projection, and case) as *type deconstructors*, as they decompose complex types into terms of their component types.

$T_o[-]{:}o ::= [-]$ – base types

$T_{\alpha\to\beta}[-]{:}o ::= T_\beta[(([-]\ M)]{:}o$ – $\forall$ closed $M{:}\alpha$

$T_{\alpha\times\beta}[-]{:}o ::= T_\alpha[\pi_1[-]]{:}o \mid T_\beta[\pi_2[-]]{:}o$

$T_{\alpha+\beta}[-]{:}o ::= T_{(\alpha\to\gamma)\to(\beta\to\gamma)\to\gamma}[\mathsf{case}^{\gamma}_{\alpha,\beta}[-]]{:}o$ – $\forall$ types γ

Figure 6: Grammar for Type Contexts

While contextual equivalence is defined to be equivalence under all term constructors, *extensional equivalence* is defined to be equivalence under all type deconstructors, as described above. Figure 6 gives the grammar for *type contexts*. Type contexts $T_\alpha[-]{:}o$ are contexts with a single hole of type α which produce base type o. The type contexts with holes of base type are simply the empty context. That is,

$$T_{int}[-]{:}int \equiv \ldots \equiv T_{pr}[-]{:}pr \equiv [-]$$

There are no type contexts such as $T_o[-]{:}o'$ for different base types o and o'.

Extensional equivalence for closed terms at base types is defined simply. For base types other than pr, terms must PCF-reduce to the same constant. Equivalent terms of type pr must be in $\approx_{sim}$ for all stores Σ. We use the superscript $\bullet$ to indicate relations on closed terms.

Definition 2.4.3 (extensional equivalence for closed terms at base type, $\cong^\bullet_o$)

For closed terms M and N,

(*int***)** $M \cong^\bullet_{int} N$ *iff* $M \to^*_{pcf} n \Leftrightarrow N \to^*_{pcf} n$.

(*bool***)** $M \cong^\bullet_{bool} N$ *iff* $M \to^*_{pcf} b \Leftrightarrow N \to^*_{pcf} b, \quad b \in \{\mathsf{true}, \mathsf{false}\}$.

(*ref*(τ)**)** $M \cong^\bullet_{ref(\tau)} N$ *iff* $M \to^*_{pcf} l^\tau \Leftrightarrow N \to^*_{pcf} l^\tau$.

(*pr***)** $M \cong^\bullet_{pr} N$ *iff* $\forall\Sigma\,.\,\langle\!\langle M\ ;\ \Sigma\rangle\!\rangle \approx_{sim} \langle\!\langle N\ ;\ \Sigma\rangle\!\rangle$.

Extensional equivalence at complex types is defined in terms of type contexts and equivalence at base types.

Definition 2.4.4 (extensional equivalence, complex types, $\cong^{\bullet}_{ext}$)

$$M \cong^{\bullet}_{ext} N{:}\tau$$

iff $M, N{:}\tau$ *and for all base types* o,

$$\forall T_\tau[-]{:}o \,.\, T_\tau[M] \cong^{\bullet}_{o} T_\tau[N].$$

The definition of the extensional preorder, $\lesssim^{\bullet}_{ext}$ has exactly the same structure as for $\cong^{\bullet}_{ext}$, except that in the previous two definitions, the $\Leftrightarrow$'s are replaced by $\Rightarrow$ and $\lesssim_{sim}$ is used for equivalence at pr type:

Definition 2.4.5 (extensional preorder, base types, $\lesssim^{\bullet}_{o}$)

For closed terms M *and* N,

(*int*) $M \lesssim^{\bullet}_{int} N$ *iff* $M \to^{*}_{pcf} n \Rightarrow N \to^{*}_{pcf} n.$

(*bool*) $M \lesssim^{\bullet}_{bool} N$ *iff* $M \to^{*}_{pcf} b \Rightarrow N \to^{*}_{pcf} b, \quad b \in \{\mathsf{true}, \mathsf{false}\}.$

(*ref*(τ)) $M \lesssim^{\bullet}_{ref(\tau)} N$ *iff* $M \to^{*}_{pcf} l^{\tau} \Rightarrow N \to^{*}_{pcf} l^{\tau}.$

(*pr*) $M \lesssim^{\bullet}_{pr} N$ *iff* $\forall \Sigma \,.\, \langle\!\langle M\ ;\Sigma\rangle\!\rangle \lesssim_{sim} \langle\!\langle N\ ;\Sigma\rangle\!\rangle.$

Definition 2.4.6 (extensional preorder, complex types, $\lesssim^{\bullet}_{ext}$)

$M \lesssim^{\bullet}_{ext} N{:}\tau$ *iff* $M, N{:}\tau$ *and for all base types* o,

$$\forall T_\tau[-]{:}o \,.\, T_\tau[M] \lesssim^{\bullet}_{o} T_\tau[N].$$

The following simple proposition about $\lesssim^{\bullet}_{ext}$ is used frequently:

Proposition 2.4.7 *If $M \lesssim^{\bullet}_{ext} N{:}\tau$ and $(MP_1 \ldots P_k){:}\tau'$ for $k \geq 0$, then $(MP_1 \ldots P_k) \lesssim^{\bullet}_{ext} (NP_1 \ldots P_k){:}\tau'$.*

PROOF: Easy induction on k.

We extend the definitions of extensional equivalence to open terms by quantifying over closing substitutions.

Definition 2.4.8 ($\cong_{ext}$)

If $\Gamma \vdash M, N{:}\tau$, then

$$\Gamma \vdash M \cong_{ext} N{:}\tau$$

iff for all closing substitutions σ that respect Γ,

$$M\sigma \cong^{\bullet}_{ext} N\sigma{:}\tau$$

And similarly for the extensional preorder on open terms, $\lesssim_{ext}$:

Definition 2.4.9 ($\lesssim_{ext}$)

If $\Gamma \vdash M, N{:}\tau$, then

$$\Gamma \vdash M \lesssim_{ext} N{:}\tau$$

iff for all closing substitutions σ that respect Γ,

$$M\sigma \lesssim^{\bullet}_{ext} N\sigma{:}\tau$$

2.4.2.1 Coinductive Definition of Extensional Equivalence

Following is an equivalent, coinductive definition of extensional equivalence. It is this definition that is used to structure proofs of extensional equivalence, because we cannot use induction on recursive types. First we define the extensional simulation generator, $[-]_{ext}$, which operates on type-indexed families of relations between closed well-typed terms.

Definition 2.4.10 ($[-]_{ext}$)

Given R a type-indexed family of relations on closed terms, $R = \{R_\tau \mid \tau \text{ a type}\}$, define

$$[R]_{ext} \stackrel{def}{=} \{[R]^{\tau}_{ext} | \ \tau \text{ a type}\}$$

where

$$\begin{array}{lcl}
M\ [R]^{o}_{ext}\ N & \Leftrightarrow & M \lesssim^{\bullet}_{o} N \\
M\ [R]^{\alpha\to\beta}_{ext}\ N & \Leftrightarrow & \forall P{:}\alpha\ .\ (MP)\ R_\beta\ (NP) \qquad - P \text{ closed} \\
M\ [R]^{\alpha\times\beta}_{ext}\ N & \Leftrightarrow & (\pi_1\, M)\ R_\alpha\ (\pi_1\, N) \text{ and } (\pi_2\, M)\ R_\beta\ (\pi_2\, N) \\
M\ [R]^{\alpha+\beta}_{ext}\ N & \Leftrightarrow & \forall \gamma\ .\ (\mathsf{case}\, M)\ R_{(\alpha\to\gamma)\to(\beta\to\gamma)\to\gamma}\ (\mathsf{case}\, N)
\end{array}$$

For now we denote the greatest fixed point of $[-]_{ext}$ as $\lesssim^{*}_{ext}$, until we prove that in fact $\lesssim^{*}_{ext} = \lesssim^{\bullet}_{ext}$.

Following are properties that the relation $\lesssim^{*}_{ext}$ satisfies as a fixed point of $[-]_{ext}$:

1. $M \lesssim^{*}_{ext} N{:}o \Leftrightarrow M \lesssim^{\bullet}_{o} N$
2. $M \lesssim^{*}_{ext} N{:}\alpha\to\beta \Leftrightarrow \forall P{:}\alpha\ .\ (MP) \lesssim^{*}_{ext} (NP){:}\beta$ – P closed
3. $M \lesssim^{*}_{ext} N{:}\alpha\times\beta \Leftrightarrow (\pi_1\, M) \lesssim^{*}_{ext} (\pi_1\, N){:}\alpha$, and $(\pi_2\, M) \lesssim^{*}_{ext} (\pi_2\, N){:}\beta$
4. $M \lesssim^{*}_{ext} N{:}\alpha+\beta \Leftrightarrow \forall \gamma, P_1{:}\alpha\to\gamma, P_2{:}\beta\to\gamma\ .$ – P_1, P_2 closed
   $(\mathsf{case}\, M P_1 P_2) \lesssim^{*}_{ext} (\mathsf{case}\, N P_1 P_2){:}\gamma$

Proposition 2.4.11

$$\lesssim^{*}_{ext} = \lesssim^{\bullet}_{ext}$$

PROOF:

⟨1⟩1. ($\supseteq$)

PROOF SKETCH: use coinduction – show that $\lesssim^{\bullet}_{ext} \subseteq [\lesssim^{\bullet}_{ext}]_{ext}$

ASSUME: $M \lesssim^{\bullet}_{ext} N{:}\tau$

PROVE: $M\ [\lesssim^{\bullet}_{ext}]_{ext}\ N$

PROOF: Cases of τ:

⟨2⟩1. CASE: base types o

Immediate.

⟨2⟩2. CASE: $\alpha{\to}\beta$

Immediate from Proposition 2.4.7.

⟨2⟩3. CASE: Other type constructions are similar

⟨1⟩2. ($\subseteq$)

ASSUME: $M \lesssim^{*}_{ext} N{:}\tau$

PROVE: $\forall o\ .\ \forall T_\tau[-]{:}o\ .\ T_\tau[M] \lesssim^{\bullet}_{o} T_\tau[N]$

PROOF: Structural induction on $T_\tau[-]{:}o$:

⟨2⟩1. CASE: $T_o = [-]$, $M, N{:}o$

PROOF: Immediate.

⟨2⟩2. CASE: $T = T'_\beta[([-]\ P)]{:}o$, $P{:}\alpha$, with $M, N{:}\alpha{\to}\beta$

PROOF: By $M \lesssim^{*}_{ext} N{:}(\alpha{\to}\beta)$, we have $\forall P{:}\alpha\ .\ (M\ P) \lesssim^{*}_{ext} (N\ P){:}\beta$. By IH($T'$), we have $T'[(M\ P)] \lesssim^{\bullet}_{o} T'[(N\ P)]$, so we are done.

⟨2⟩3. CASE: Other constructions for T are similar.

□

All of the basic relations we use are easily shown to enjoy reflexivity and transitivity:

Lemma 2.4.12 *The relations $\lesssim_{sim}$, $\lesssim_{\forall C}$, $\lesssim_{ext}$, $\lesssim^{\bullet}_{ext}$, and their symmetrizations ($\approx_{sim}$, $\cong_{\forall C}$, $\cong_{ext}$, $\cong^{\bullet}_{ext}$), are preorders.*

Lemma 2.4.13 *Given a preorder S on states, $[S]_{sim}$ is a preorder.*

Lemma 2.4.14 *Given a type-indexed family of relations on terms, R, $[R]_{ext}$ is a preorder.*

2.5 Theory of the Metalanguage

The main theoretical result in this section is an Extensionality Theorem which states that extensional equivalence coincides with contextual equivalence, i.e.,

$$\forall \Gamma, M, N \,.\, \Gamma \vdash M \cong_{ext} N{:}\tau \Leftrightarrow \Gamma \vdash M \cong_{\forall C} N{:}\tau$$

We get this extensionality result by showing that the extensional preorder is a *precongruence*, namely that $\lesssim_{ext}$ is preserved by the term constructors of the metalanguage. To prove the precongruence lemma, we need a substitution lemma, stating that substitution of a variable by $\lesssim^{\bullet}_{ext}$-related terms produces $\lesssim^{\bullet}_{ext}$-related terms. The proof of the substitution lemma is the most involved.

Before we can get to the big, important proofs, we need to establish a number of supporting properties of the metalanguage. The reader is invited to skim this section, noting the lemmas given, and referring back to this section as needed whilst poring over the more interesting proofs later.

2.5.1 Simulations

The typical structure of a proof of simulation in our metalanguage is to use coinduction at the outermost level and to use induction to prove the individual cases. Given a monotonic relational operator such as $[-]_{sim}$, and a candidate simulation relation S, a standard coinductive argument is to show that $S \subseteq [S]_{sim}$ and that therefore $S \subseteq \lesssim_{sim}$. Often, though, it is more straightforward to show something such as $S \subseteq [\lesssim_{sim} \circ S \circ \lesssim_{sim}]_{sim}$, thus requiring a supporting lemma that $[\lesssim_{sim} \circ S \circ \lesssim_{sim}]_{sim} \subseteq \lesssim_{sim}$. These supporting lemmas are usually easy to prove, and are often referred to as "Simulation up to", as presented in [Mil90, Propositions 3.4.2 and 3.4.3]. We give the basic lemmas, mutatis mutandis for our notation, here.

Definition 2.5.1 *A relation S over states is a* simulation up to $\lesssim_{sim}$ *if* $(\lesssim_{sim} \circ S \circ \lesssim_{sim})$ *is a simulation.*

Because $\lesssim_{sim}$ is defined as the greatest fixed point of $[-]_{sim}$, the way we show that $(\lesssim_{sim} \circ S \circ \lesssim_{sim})$ is a simulation (i.e., is contained in $\lesssim_{sim}$) is to show that

$$(\lesssim_{sim} \circ S \circ \lesssim_{sim}) \subseteq [\lesssim_{sim} \circ S \circ \lesssim_{sim}]_{sim}$$

Lemma 2.5.2 (Milner) *If S is a simulation up to $\lesssim_{sim}$, then $S \subseteq \lesssim_{sim}$.*

Proof: Let $P \; S \; Q$. Then by reflexivity of $\lesssim_{sim}$, $P \; (\lesssim_{sim} \circ S \circ \lesssim_{sim}) \; Q$. So $P \lesssim_{sim} Q$ since $\lesssim_{sim} \circ S \circ \lesssim_{sim} \subseteq \lesssim_{sim}$.

□

Lemma 2.5.3 (Milner) *Given a relation S over states,*
if $S \subseteq [\lesssim_{sim} \circ S \circ \lesssim_{sim}]_{sim}$, then $S \subseteq \lesssim_{sim}$.

Proof: By Lemma 2.5.2, it is enough to show that $(\lesssim_{sim} \circ S \circ \lesssim_{sim}) \subseteq [\lesssim_{sim} \circ S \circ \lesssim_{sim}]_{sim}$ (i.e. that S is a simulation up to $\lesssim_{sim}$). For this purpose, let $P \; (\lesssim_{sim} \circ S \circ \lesssim_{sim}) \; Q$ – i.e. there exists P_1, Q_1 such that $P \lesssim_{sim} P_1 \; S \; Q_1 \lesssim_{sim} Q$. Suppose $P \xrightarrow{\alpha}_{io} P'$; we wish to find Q' so that $Q \xrightarrow{\alpha}_{io} Q'$ and $P' \; (\lesssim_{sim} \circ S \circ \lesssim_{sim}) \; Q'$. This is done by filling in the bottom row of the following diagram from left to right, starting with the given P':

$$\begin{array}{ccccccc}
P & \lesssim_{sim} & P_1 & S & Q_1 & \lesssim_{sim} & Q \\
\downarrow & & \downarrow & & \downarrow & & \downarrow \\
P' & \lesssim_{sim} & P'_1 & \lesssim_{sim} \circ S \circ \lesssim_{sim} & Q'_1 & \lesssim_{sim} & Q'
\end{array}$$

□

2.5.2 Properties of the Extended PCF Functional Sublanguage

Several properties of PCF, proved elsewhere in the literature, are used to get results about our metalanguage, which is an extension of PCF. Because PCF has been extensively studied, we present only the statements of the PCF properties we need. For a good introduction to PCF, and for detailed proofs including confluence, subject reduction, and standardization, see [Mit96].

Recursive types are not a standard part of PCF and are more reminiscent of *FPC*, as presented in textbooks by Gunter [Gun92] or Winskel [Win93]. In [Gor95], Andrew Gordon presents a bisimilarity-based theory of *FPC*, including extensionality.

Lemma 2.5.4 (Subject Reduction for PCF) *If* $M{:}\tau$ *and* $M \rightarrow_{pcf} N$, *then* $N{:}\tau$.

Lemma 2.5.5 (Confluence for PCF) *If* $M \rightarrow^*_{pcf} N$ *and* $M \rightarrow^*_{pcf} N'$, *then* $\exists P$ *such that* $N \rightarrow^*_{pcf} P$ *and* $N' \rightarrow^*_{pcf} P$.

The confluence property is also known as the "Church-Rosser" (CR) property or the "Diamond" property, after the commuting diagram used to illustrate confluence:

$$\begin{tikzcd} & M \arrow[dl, "*"'] \arrow[dr, "*"] & \\ N \arrow[dr, dashed, "*"'] & & N' \arrow[dl, dashed, "*"] \\ & P & \end{tikzcd}$$

It is fundamental that if a term PCF-reduces to a constant, then a leftmost reduction strategy will reduce to the same constant. See [JM91] for a detailed proof of standardization for PCF-like reduction systems.

Definition 2.5.6 ($\rightarrow^*_{left,pcf}$) *The transition relation* $\rightarrow_{left,pcf}$ *always chooses the leftmost PCF reduction.*

Lemma 2.5.7 (Completeness of Leftmost Reduction for PCF) *For all constants* c *of type int, bool, or ref*(τ), *If* $M \rightarrow^*_{pcf} c$, *then* $M \rightarrow^*_{left,pcf} c$.

2.5.3 Subject Reduction for Metalanguage

Lemma 2.5.8 (Subject Reduction)

1. *If* $\langle\!\langle M \,;\, \Sigma \rangle\!\rangle \to^*_{sto} \langle\!\langle M' \,;\, \Sigma' \rangle\!\rangle$, *then* $\emptyset \vdash M'{:}pr$.

2. *If* $\langle\!\langle M \,;\, \Sigma \rangle\!\rangle \xrightarrow{a}_{io} \langle\!\langle M' \,;\, \Sigma' \rangle\!\rangle$, *then* $\emptyset \vdash M'{:}pr$.

PROOF:

Easy by inspection of reduction rules.

2.5.4 Simulation up to Location Renaming

A property that is used in nearly every simulation proof is that a consistent renaming of locations between states maintains $\lesssim_{sim}$.

Two states are equal up to a renaming of location constants, denoted α_{loc}, if there is a partial bijection[1] Θ between locations that satisfies several properties:

Definition 2.5.9 (α_{loc})

$$\langle\!\langle M \,;\, \Sigma_M \rangle\!\rangle \; \alpha_{loc} \; \langle\!\langle N \,;\, \Sigma_N \rangle\!\rangle$$

iff $\exists \Theta \subseteq \mathit{Locs} \times \mathit{Locs}$ s.t.

1. Θ *is a partial bijection*

2. $M\Theta = N$

3. $\mathit{locs}(M) \subseteq \mathit{dom}(\Theta)$

4. *if* $(l, l') \in \Theta$ *then either*

   (a) $l \notin \mathit{dom}(\Sigma_M)$ *and* $l' \notin \mathit{dom}(\Sigma_N)$ *or*

   (b) $\Sigma_M(l)\Theta = \Sigma_N(l')$ *and* $\mathit{locs}(\Sigma_M(l)) \subseteq \mathit{dom}(\Theta)$

[1] Θ is a partial bijection iff Θ has the property that if $(i, j) \in \Theta$ and $(i', j') \in \Theta$, then $(i = i' \Leftrightarrow j = j')$.

Lemma 2.5.10 *If* $\langle\langle M\ ;\ \Sigma_M\rangle\rangle\ \alpha_{loc}\ \langle\langle N\ ;\ \Sigma_N\rangle\rangle$ *and* $\langle\langle M\ ; \Sigma_M\rangle\rangle \xrightarrow{n}_{sto} \langle\langle M'\ ; \Sigma'_M\rangle\rangle$, *then* $\exists N', \Sigma'_N$ s.t.

$$\langle\langle N\ ; \Sigma_N\rangle\rangle \xrightarrow{n}_{sto} \langle\langle N'\ ; \Sigma'_N\rangle\rangle$$

and $\langle\langle M'\ ; \Sigma'_M\rangle\rangle\ \alpha_{loc}\ \langle\langle N'\ ; \Sigma'_N\rangle\rangle$.

PROOF:

Induction on n.

Lemma 2.5.11 (α_{loc})

$$\alpha_{loc} \subseteq \lesssim_{sim}$$

PROOF:

PROOF SKETCH: Use coinduction.

$\langle 1\rangle 1.$ $\alpha_{loc} \subseteq [\alpha_{loc}]_{sim}$

ASSUME: 1. $\langle\langle M\ ; \Sigma_M\rangle\rangle\ \alpha_{loc}\ \langle\langle N\ ; \Sigma_N\rangle\rangle$, witnessed by

2. $\Theta \subseteq dom(\Sigma_M) \times dom(\Sigma_N)$ a partial bijection

3. $\langle\langle M\ ; \Sigma_M\rangle\rangle \xrightarrow{n}_{sto} \langle\langle \mathsf{io}\, M'\ ; \Sigma'_M\rangle\rangle$

PROVE: $\exists N', \Sigma'_N$ s.t.

$\langle\langle N\ ; \Sigma_N\rangle\rangle \rightarrow^*_{sto} \langle\langle \mathsf{io}\, N'\ ; \Sigma'_N\rangle\rangle$ and

$\langle\langle M'\ ; \Sigma'_M\rangle\rangle\ \alpha_{loc}\ \langle\langle N'\ ; \Sigma'_N\rangle\rangle$.

PROOF: Induction on n.

$\langle 2\rangle 1.$ CASE: $n = 0$

If $n = 0$, it must be that both sides are headed by the same I/O operator and that they stay (after 0 steps) in α_{loc}.

$\langle 2\rangle 2.$ CASE: $n > 0$

Note that the left- and right-hand sides must have the same term structure, differing only in the actual location constants. Therefore, for each step that the left-hand side takes, the right-hand side may take a corresponding step. Consider the first step in the reduction of the left-hand side:

$$\langle\!\langle M \ ; \Sigma_M \rangle\!\rangle \xrightarrow{1}_{sto} \langle\!\langle M'' \ ; \Sigma''_M \rangle\!\rangle \xrightarrow{n-1}_{sto} \langle\!\langle \mathsf{io}\, M' \ ; \Sigma'_M \rangle\!\rangle$$

Either the first step is an internal PCF reduction, a head PCF reduction, or a head store operation reduction.

It is clear that PCF reductions do not affect α_{loc} – the same Θ remains valid after both sides take the same step.

For head store reductions, deref and update both maintain α_{loc} with the same Θ. This leaves new as the only interesting case:

⟨3⟩1. CASE: $M = \mathsf{new}\, M_1 M_2$

⟨4⟩1. $\mathbf{lhs} = \langle\!\langle \mathsf{new}\, M_1\ M_2 \ ; \Sigma_M \rangle\!\rangle$

$\xrightarrow{1}_{sto} \langle\!\langle (M_2\ l) \ ; \Sigma_M[l \mapsto M_1] \rangle\!\rangle \quad l\ \mathit{fresh}$

$\stackrel{def}{=} \mathbf{lhs}'$

$\xrightarrow{j<n}_{left,sto} \langle\!\langle \mathsf{io}\, M'' \ ; \Sigma''_M \rangle\!\rangle$

⟨4⟩2. $\mathbf{rhs} = \langle\!\langle \mathsf{new}\, N_1\ N_2 \ ; \Sigma_N \rangle\!\rangle$

$\xrightarrow{1}_{sto} \langle\!\langle (N_2\ l') \ ; \Sigma_N[l' \mapsto N_1] \rangle\!\rangle \quad l'\mathit{fresh}$

$\stackrel{def}{=} \mathbf{rhs}'$

⟨4⟩3. $\mathbf{lhs}'\ \alpha_{loc}\ \mathbf{rhs}'$

PROOF: Construct new partial bijection $\Theta' = \Theta \cup \{(l, l')\}$.

⟨4⟩4. Q.E.D.

PROOF: by IH(j).

□

2.5.5 PCF Reduction Preserves Simulation (Full Beta)

We can do fully lazy PCF reduction (a.k.a. "full β") without affecting the observable behavior of states. First we define PCF-equality for terms ($=_{pcf}{}^{term}$) and states ($=_{pcf}{}^{state}$). Generally it will be obvious whether we are referring to $=_{pcf}$ between terms or states, and we will usually omit the superscript.

Definition 2.5.12 **($=_{pcf}{}^{term}$)**

$$M =_{pcf}{}^{term} N$$

iff $\exists P$ s.t.

$$M \rightarrow^*_{pcf} P \text{ and } N \rightarrow^*_{pcf} P.$$

Definition 2.5.13 **($=_{pcf}{}^{state}$)**

$$\langle\!\langle M \; ; \Sigma_M \rangle\!\rangle =_{pcf}{}^{state} \langle\!\langle N \; ; \Sigma_N \rangle\!\rangle$$

iff

1. $M =_{pcf}{}^{term} N$, *and*
2. $dom(\Sigma_M) = dom(\Sigma_N)$, *and*
3. $\forall l \in dom(\Sigma_M) \; . \; \Sigma_M(l) =_{pcf}{}^{term} \Sigma_N(l)$

Lemma 2.5.14 ($=_{pcf}{}^{term}$ is a congruence) $=_{pcf}{}^{term}$ *is a congruence. That is, if* $M =_{pcf}{}^{term} N$ *and* $M, N{:}\tau$*, then for all contexts* $C[-_\tau]{:}\tau'$*,*

$$C[M] =_{pcf}{}^{term} C[N]$$

PROOF:

Easy induction on structure of C.

Lemma 2.5.15 (full β for states)

If $\langle\!\langle M \; ; \Sigma_M \rangle\!\rangle =_{pcf}{}^{state} \langle\!\langle N \; ; \Sigma_N \rangle\!\rangle$*, then*

$$\langle\!\langle M \; ; \Sigma_M \rangle\!\rangle \lesssim_{sim} \langle\!\langle N \; ; \Sigma_N \rangle\!\rangle.$$

PROOF:

ASSUME: $\langle\!\langle M \,;\, \Sigma_M \rangle\!\rangle \xrightarrow{n}_{sto} \langle\!\langle \mathsf{io}\, M' \,;\, \Sigma'_M \rangle\!\rangle$

PROVE: $\exists N', \Sigma'_N$ s.t. $\langle\!\langle N \,;\, \Sigma_N \rangle\!\rangle \to^*_{sto} \langle\!\langle \mathsf{io}\, N' \,;\, \Sigma'_N \rangle\!\rangle$ and

$\langle\!\langle M' \,;\, \Sigma'_M \rangle\!\rangle (\lesssim_{sim} \circ =_{pcf}{}^{state} \circ \lesssim_{sim}) \langle\!\langle N' \,;\, \Sigma'_N \rangle\!\rangle$

by induction on n.

$\langle 1 \rangle 1$. CASE: $n = 0$

PROOF: By $M =_{pcf} N$ and $M = \mathsf{io}\, M'$, it must be that $\exists N'$ s.t. $N \to^*_{pcf} \mathsf{io}\, N'$ and $M' =_{pcf} N'$.

$\langle 1 \rangle 2$. CASE: $n > 0$

Consider the first step in the sequence of reductions.

$\langle 2 \rangle 1$. CASE: First step is a PCF reduction

Clearly both sides stay in $=_{pcf}$, so result follows from IH$(n-1)$.

$\langle 2 \rangle 2$. CASE: First step is a store operator

Consider new case:

$\langle 3 \rangle 1$. CASE: $M = \mathsf{new}\, M_1\ M_2$

$\langle 4 \rangle 1$. $\mathbf{lhs} \stackrel{def}{=} \langle\!\langle M \,;\, \Sigma_M \rangle\!\rangle = \langle\!\langle \mathsf{new}\, M_1\ M_2 \,;\, \Sigma_M \rangle\!\rangle$

$\xrightarrow{1}_{sto} \langle\!\langle (M_2\ l) \,;\, \Sigma_M[l \mapsto M_1] \rangle\!\rangle \quad l$ *fresh* w.r.t. $\{M_1, M_2, \Sigma_M\}$

$\stackrel{def}{=} \mathbf{lhs}'$

$\xrightarrow{j<n}_{sto} \langle\!\langle \mathsf{io}\, M' \,;\, \Sigma'_M \rangle\!\rangle$

$\langle 4 \rangle 2$. $\exists N_1, N_2$ s.t. $N \to^*_{pcf} \mathsf{new}\, N_1\ N_2$ and $M_1 =_{pcf} N_1$ and $M_2 =_{pcf} N_2$

PROOF: By $M =_{pcf} N$ and $M = \mathsf{new}\, M_1 M_2$.

$\langle 4 \rangle 3$. $\mathbf{rhs} \stackrel{def}{=} \langle\!\langle N \,;\, \Sigma_N \rangle\!\rangle$

$\to^*_{pcf} \langle\!\langle \mathsf{new}\, N_1\ N_2 \,;\, \Sigma_N \rangle\!\rangle \quad$ by Step $\langle 4 \rangle 2$

$\xrightarrow{1}_{sto} \langle\!\langle (N_2\ l') \,;\, \Sigma_N[l' \mapsto N_1] \rangle\!\rangle \quad l'$ *fresh* w.r.t. $\{N_1, N_2, \Sigma_N\}$

$\stackrel{def}{=} \mathbf{rhs}'$

LET: l'' be fresh w.r.t. $\{M_1, M_2, N_1, N_2, \Sigma_M, \Sigma_N\}$.

$\langle 4 \rangle 4$. $\langle\!\langle (M_2\ l) \,;\, \Sigma_M[l \mapsto M_1] \rangle\!\rangle\ \alpha_{loc}\ \langle\!\langle (M_2\ l'') \,;\, \Sigma_M[l'' \mapsto M_1] \rangle\!\rangle$

$\langle 4 \rangle 5$. $\langle\!\langle (N_2\ l') \,;\, \Sigma_N[l' \mapsto N_1] \rangle\!\rangle\ \alpha_{loc}\ \langle\!\langle (N_2\ l'') \,;\, \Sigma_N[l'' \mapsto N_1] \rangle\!\rangle$

$\langle 4 \rangle 6$. $\exists M'', \Sigma''_M$ s.t. $\langle\!\langle (M_2\ l'') \,;\, \Sigma_M[l'' \mapsto M_1] \rangle\!\rangle \xrightarrow{j<n}_{sto} \langle\!\langle \mathsf{io}\, M'' \,;\, \Sigma''_M \rangle\!\rangle$

s.t. $\langle\!\langle M' \,;\, \Sigma'_M \rangle\!\rangle\ \alpha_{loc}\ \langle\!\langle M'' \,;\, \Sigma''_M \rangle\!\rangle$

By Lemma 2.5.10.

$\langle 4\rangle 7$. $\langle\!\langle (M_2\ l'')\ ;\ \Sigma_M[l'' \mapsto M_1]\rangle\!\rangle =_{pcf} \langle\!\langle (N_2\ l'')\ ;\ \Sigma_N[l'' \mapsto N_1]\rangle\!\rangle$

By Step $\langle 4\rangle 2$.

$\langle 4\rangle 8$. $\exists N'', \Sigma''_N$ s.t. $\langle\!\langle (N_2\ l'')\ ;\ \Sigma_N[l'' \mapsto N_1]\rangle\!\rangle \to^*_{sto} \langle\!\langle \mathsf{io}\, N''\ ;\ \Sigma''_N\rangle\!\rangle$ and $\langle\!\langle M''\ ;\ \Sigma''_M\rangle\!\rangle\ (\lesssim_{sim} \circ =_{pcf} \circ \lesssim_{sim})\ \langle\!\langle N''\ ;\ \Sigma''_N\rangle\!\rangle$

Proof: By steps $\langle 4\rangle 6$ and $\langle 4\rangle 7$ and IH(j).

$\langle 4\rangle 9$. $\exists N', \Sigma'_N$ s.t. $\langle\!\langle (N_2\ l')\ ;\ \Sigma_N[l' \mapsto N_1]\rangle\!\rangle \to^*_{sto} \langle\!\langle N'\ ;\ \Sigma'_N\rangle\!\rangle$ and $\langle\!\langle N''\ ;\ \Sigma''_N\rangle\!\rangle \lesssim_{sim} \langle\!\langle N'\ ;\ \Sigma'_N\rangle\!\rangle$

Proof: By steps $\langle 4\rangle 8$ and $\langle 4\rangle 5$ and Lemma 2.5.11, i.e. $\alpha_{loc} \subseteq \lesssim_{sim}$.

$\langle 4\rangle 10$. Q.E.D.

Proof: We have

$\langle\!\langle M'\ ;\ \Sigma'_M\rangle\!\rangle\ \alpha_{loc}\ \langle\!\langle M''\ ;\ \Sigma''_M\rangle\!\rangle\ (\lesssim_{sim} \circ =_{pcf} \circ \lesssim_{sim})\ \langle\!\langle N''\ ;\ \Sigma''_N\rangle\!\rangle \lesssim_{sim} \langle\!\langle N''\ ;\ \Sigma''_N\rangle\!\rangle$

and $\alpha_{loc} \subseteq \lesssim_{sim}$ so we are done.

$\langle 3\rangle 2$. Case: deref and update cases

are similar to the new case, without the need to use α_{loc}. We do need the property, which follows directly from confluence, that if $M =_{pcf} N$ and $M, N{:}ref(\tau)$, then $M \to^*_{pcf} l \Rightarrow N \to^*_{pcf} l$.

□

Corollary 2.5.16 (full β for terms)

If $M =_{pcf}{}^{term} N$, $M, N{:}\tau$, then $M \cong_{\forall C} N$.

Proof:

Immediate from Lemma 2.5.14.

It is easy to show that PCF reduction preserves extensional equivalence.

Lemma 2.5.17 $=_{pcf}{}^{term} \subseteq \lesssim^{\bullet}_{ext}$

Proof:

An easy coinduction.

2.5.6 Completeness of Leftmost Reduction

We show that if a state can reduce under unrestricted $\rightarrow^*_{sto}$ to an I/O operation, then there is a leftmost reduction sequence which will get us to the same I/O operation, and the continuations will be in $\lesssim_{sim}$.

Definition 2.5.18 **($\rightarrow^*_{left,sto}$)** *The reduction operator $\rightarrow_{left,sto}$ always chooses the leftmost redex, if any. In particular, it will always choose to take a* deref*,* update*, or* new *step if it can.*

We prove a slightly more general lemma, allowing start states to be within a sequence of PCF reductions of each other.

Lemma 2.5.19 *If $\langle\!\langle M\ ;\Sigma_M\rangle\!\rangle\ {=_{pcf}}^{state}\ \langle\!\langle N\ ;\Sigma_N\rangle\!\rangle$ and $\langle\!\langle M\ ;\Sigma_M\rangle\!\rangle \rightarrow^*_{sto} \langle\!\langle \mathsf{io}\, M'\ ;\Sigma'_M\rangle\!\rangle$, then $\exists N', \Sigma'_N$ such that*

$$\langle\!\langle N\ ;\Sigma_N\rangle\!\rangle \rightarrow^*_{left,sto} \langle\!\langle \mathsf{io}\, N'\ ;\Sigma'_N\rangle\!\rangle,\ \text{and}$$
$$\langle\!\langle M'\ ;\Sigma'_M\rangle\!\rangle \lesssim_{sim} \langle\!\langle N'\ ;\Sigma'_N\rangle\!\rangle$$

PROOF: By induction on the number of $\rightarrow_{sto}$ steps in the reduction sequence $\langle\!\langle M\ ;\Sigma_M\rangle\!\rangle \rightarrow^*_{sto} \langle\!\langle \mathsf{io}\, M'\ ;\Sigma'_M\rangle\!\rangle$.

$\langle 1\rangle$1. CASE: $n = 0$

Immediate by Lemma 2.5.15, i.e. ${=_{pcf}}^{state} \subseteq \lesssim_{sim}$.

$\langle 1\rangle$2. CASE: $n > 0$

Consider first step of lhs:

$\langle\!\langle M\ ;\Sigma_M\rangle\!\rangle \xrightarrow{1}_{sto} \langle\!\langle M''\ ;\Sigma''_M\rangle\!\rangle \xrightarrow{n-1}_{sto} \langle\!\langle \mathsf{io}\, M'\ ;\Sigma'_M\rangle\!\rangle$:

$\langle 2\rangle$1. CASE: first step is a PCF reduction

$\langle\!\langle M''\ ;\Sigma''_M\rangle\!\rangle =_{pcf} \langle\!\langle N\ ;\Sigma_N\rangle\!\rangle$, and we get the desired result by IH$(n-1)$.

$\langle 2\rangle$2. CASE: first step is leftmost store op

$\langle 3\rangle 1.$ CASE: $M = \mathsf{deref}\, l\ M_1$

ASSUME: 1. $\Sigma_M(l) = Q$

2. $\Sigma_N(l) = Q'$

$\langle 3\rangle 2.$ $Q =_{pcf} Q'$

By $\langle\!\langle M\ ;\Sigma_M\rangle\!\rangle =_{pcf}{}^{state} \langle\!\langle N\ ;\Sigma_N\rangle\!\rangle$.

$\langle 3\rangle 3.$ $\mathbf{lhs} \stackrel{def}{=} \langle\!\langle \mathsf{deref}\, l\ M_1\ ;\Sigma_M\rangle\!\rangle$

$\stackrel{1}{\rightarrow}_{sto} \langle\!\langle M_1\ Q\ ;\Sigma_M\rangle\!\rangle$

$\stackrel{n-1}{\rightarrow}_{sto} \langle\!\langle \mathsf{io}\, M'\ ;\Sigma'_M\rangle\!\rangle$

$\langle 3\rangle 4.$ $\exists N'_1$ s.t. $\langle\!\langle N\ ;\Sigma_N\rangle\!\rangle \rightarrow^*_{pcf} \langle\!\langle \mathsf{deref}\, l\ N'_1\ ;\Sigma_N\rangle\!\rangle$ and $M_1 =_{pcf} N'_1$

By $M =_{pcf} N$.

$\langle 3\rangle 5.$ $\exists N_1$ s.t. $\langle\!\langle N\ ;\Sigma_N\rangle\!\rangle \rightarrow^*_{left,pcf} \langle\!\langle \mathsf{deref}\, l\ N_1\ ;\Sigma_N\rangle\!\rangle$ and $M_1 =_{pcf} N_1$

By completeness of leftmost reduction for PCF and transitivity of $=_{pcf}$.

$\langle 3\rangle 6.$ $\mathbf{rhs} \stackrel{def}{=} \langle\!\langle N\ ;\Sigma_N\rangle\!\rangle$

$\rightarrow^*_{left,pcf} \langle\!\langle \mathsf{deref}\, l\ N_1\ ;\Sigma_N\rangle\!\rangle$

$\stackrel{1}{\rightarrow}_{left,sto} \langle\!\langle N_1\ Q'\ ;\Sigma_N\rangle\!\rangle$

$\rightarrow^*_{left,sto} \langle\!\langle \mathsf{io}\, N'\ ;\Sigma'_N\rangle\!\rangle$ s.t.

$\langle\!\langle M'\ ;\Sigma'_M\rangle\!\rangle \lesssim_{sim} \langle\!\langle N'\ ;\Sigma'_N\rangle\!\rangle$

By IH$(n-1)$.

$\langle 2\rangle 3.$ CASE: other leftmost store ops

are similar to deref case.

□

Lemma 2.5.20 (completeness of $\rightarrow^*_{left,sto}$ w.r.t. $\lesssim_{sim}$)

If $\langle\!\langle M\ ;\Sigma\rangle\!\rangle \rightarrow^*_{sto} \langle\!\langle \mathsf{io}\, M'\ ;\Sigma'\rangle\!\rangle$, *then* $\exists M'', \Sigma''$ *such that*

$\langle\!\langle M\ ;\Sigma\rangle\!\rangle \rightarrow^*_{left,sto} \langle\!\langle \mathsf{io}\, M''\ ;\Sigma''\rangle\!\rangle$, *and*

$\langle\!\langle M'\ ;\Sigma'\rangle\!\rangle \lesssim_{sim} \langle\!\langle M''\ ;\Sigma''\rangle\!\rangle$

PROOF: immediate from preceding Lemma 2.5.19 and reflexivity of $=_{pcf}$.

□

2.5.7 Substitution Lemma for PCF

A substitution lemma for PCF is used in [Pit99] and proved in [JM91]. Given standardization, the proof of the substitution lemma is a straightforward induction on the number of steps by the left-hand side in a standard reduction to a constant.

Lemma 2.5.21 (Substitution Lemma for PCF and $\lesssim^{\bullet}_{ext}$) *For non-pr base type o, closed terms M, N of type τ and open term P such that $x{:}\tau \vdash P{:}o$, if $M \lesssim^{\bullet}_{ext} N{:}\tau$, then*

$$P[M/x] \lesssim^{\bullet}_{ext} P[N/x]{:}o.$$

PROOF:

Recall that at base types other than pr, two terms are in $\lesssim^{\bullet}_{ext}$ iff they PCF-reduce to the same constant. We use induction on the length of a standard reduction to a constant, by considering all possible forms of P. The induction hypothesis is:

$$\mathrm{IH}(n) = \text{if } P[M/x] \overset{n}{\to}_{left,pcf} c, \text{ then } P[N/x] \to^{*}_{pcf} c$$

We give a few of the cases:

$\langle 1\rangle$1. CASE: $P = c$. Immediate.

$\langle 1\rangle$2. CASE: $P = x$

It must be that $\tau = o$, so by $M \lesssim^{\bullet}_{ext} N{:}\tau$, we have co-PCF-reduction to the same constant.

$\langle 1\rangle$3. CASE: $P = (\lambda y.P_0)\ P_1 \ldots P_k$, $y \neq x$

$\langle 2\rangle$1. $\mathbf{lhs} \to_{\beta} P_0[P_1[M/x]/y]\ P_2[M/x] \ldots P_k[M/x]$

$$= (P_0[P_1/y]\ P_2 \ldots P_k)[M/x]$$
$$\overset{n-1}{\to}_{left,pcf}\ c$$

$\langle 2\rangle$2. $\mathbf{rhs} \to_{\beta} P_0[P_1[N/x]/y]\ P_2[N/x] \ldots P_k[N/x]$

$$= (P_0[P_1/y]\ P_2 \ldots P_k)[N/x]$$
$$\to^{*}_{pcf}\ c$$

by IH$(n-1)$.

Other cases are similarly straightforward.

□

2.5.8 Contextual Equivalence with Empty Stores

If the terms we are considering are location-free, it suffices to show (bi)simulation starting with an empty store.

Lemma 2.5.22 *If $\Gamma \vdash M, N{:}\tau$ and if M and N contain no locations, then $\Gamma \vdash M \cong_{\forall C} N{:}\tau$ iff for all closing contexts $C[-_\tau]$ of type pr that respect Γ,*

$$\langle\!\langle C[M]\ ;\ \emptyset\rangle\!\rangle \approx_{sim} \langle\!\langle C[N]\ ;\ \emptyset\rangle\!\rangle$$

PROOF:

(⇒) Immediate from the definition of $\cong_{\forall C}$.

(⇐) Suppose the lemma is not true. In that case, we have that for all closing contexts $C[-_\tau]$ of type pr that respect Γ,

$$\langle\!\langle C[M]\ ;\ \emptyset\rangle\!\rangle \approx_{sim} \langle\!\langle C[N]\ ;\ \emptyset\rangle\!\rangle$$

yet there must exist some context C' and store Σ such that

$$\langle\!\langle C'[M]\ ;\ \Sigma\rangle\!\rangle \not\approx_{sim} \langle\!\langle C'[N]\ ;\ \Sigma\rangle\!\rangle$$

Let the locations in Σ be $\{l_1, \ldots, l_n\}$ with contents $\{M_1, \ldots, M_n\}$. Let $\{z_1, \ldots, z_n\}$ be n fresh variables of the appropriate types. Define a context C'' as follows:

$$\begin{aligned} C'' \stackrel{def}{=}\ & \mathsf{new}\,\Omega_1\ (\lambda z_1.\ldots \mathsf{new}\,\Omega_n\ (\lambda z_n. \\ & \mathsf{update}\,z_1\ M_1(\ldots \mathsf{update}\,z_n\ M_n \\ & C'''[-_\tau])\ldots)) \end{aligned}$$

where Ω_i has the type of M_i and where $C'''[]$ is obtained from $C'[]$ by replacing every occurrence of l_i in $C'[]$ by z_i.

After about $2n$ steps, $\langle\!\langle C''[M]\ ;\ \emptyset\rangle\!\rangle$ gets into a state which is identical to $\langle\!\langle C'[M]\ ;\ \Sigma\rangle\!\rangle$ (recall that new can choose the location returned, as long as the location is fresh). The same reasoning holds for the right-hand side, $\langle\!\langle C''[N]\ ;\ \emptyset\rangle\!\rangle$. Thus the state context $\langle\!\langle C''[]\ ;\ \emptyset\rangle\!\rangle$ distinguishes M and N – contradiction.

□

Note that if M and N have locations in them, this lemma does not hold. It could be that the only way to distinguish some such M and N is with a store with exactly those locations set which occur in M and N. Attempting to create such a store from the empty store cannot work, because locations created by `new` are guaranteed *not* to occur in the terms.

2.6 Substitution Lemmas for the Metalanguage

We are now ready to prove two substitution lemmas for the metalanguage – one for substitution within states and one for substitution on terms. We present the proofs of these lemmas in quite a lot of detail, for several reasons:

1. The substitution lemmas are important, and we want a high degree of confidence that they have been correctly proven,
2. Both the coinductive and inductive steps in the proofs are very similar to steps used in many other proofs in this thesis. In later sections, we may give proofs in less detail, referring to the following proofs for guidance.
3. This is, after all, a doctoral thesis, and we are more concerned with completeness and correctness than brevity.

Lemma 2.6.1 (Substitution in States) *If $x{:}\tau' \vdash Q{:}pr$ and $M \lesssim^{\bullet}_{ext} N{:}\tau'$, then for all stores Σ,*

$$\langle\!\langle Q[M/x]\ ;\ \Sigma \rangle\!\rangle \lesssim_{sim} \langle\!\langle Q[N/x]\ ;\ \Sigma \rangle\!\rangle.$$

Proof sketch: We define a relation S^*_{pr} on metalanguage states and show that $S^*_{pr} \subseteq \lesssim_{sim}$. What we actually show is that

$$S^*_{pr} \subseteq [\lesssim_{sim} \circ S^*_{pr} \circ \lesssim_{sim}]_{sim},$$

which implies $S^*_{pr} \subseteq \lesssim_{sim}$ by Lemma 2.5.3.

PROOF:

LET:

$$S^*_{pr} \stackrel{def}{=} \{ (\langle\!\langle P \,;\, \Sigma \rangle\!\rangle[M/x], \langle\!\langle P \,;\, \Sigma \rangle\!\rangle[N/x]) \mid x{:}\tau' \vdash P{:}pr \text{ and } fv(rng(\Sigma)) \subseteq \{x\} \} \text{ for all terms } P \text{ and stores } \Sigma$$

where substitution is extended to stores by mapping a substitution over the range of a store.

⟨1⟩1. $S^*_{pr} \subseteq [\lesssim_{sim} \circ S^*_{pr} \circ \lesssim_{sim}]_{sim}$

ASSUME: 1. $(\langle\!\langle P \,;\, \Sigma \rangle\!\rangle[M/x], \langle\!\langle P \,;\, \Sigma \rangle\!\rangle[N/x]) \in S^*_{pr}$

2. $\langle\!\langle P \,;\, \Sigma \rangle\!\rangle[M/x] \to^*_{sto} \langle\!\langle \mathsf{io}\, P' \,;\, \Sigma' \rangle\!\rangle$

PROVE: $\exists P'', \Sigma''$ such that

1. $\langle\!\langle P \,;\, \Sigma \rangle\!\rangle[N/x] \to^*_{sto} \langle\!\langle \mathsf{io}\, P'' \,;\, \Sigma'' \rangle\!\rangle$, and

2. $\langle\!\langle P' \,;\, \Sigma' \rangle\!\rangle \; (\lesssim_{sim} \circ S^*_{pr} \circ \lesssim_{sim}) \; \langle\!\langle P'' \,;\, \Sigma'' \rangle\!\rangle$.

⟨2⟩1. $\exists n, P^3, \Sigma^3$ such that

$$\langle\!\langle P \,;\, \Sigma \rangle\!\rangle[M/x] \xrightarrow{n}_{left,sto} \langle\!\langle \mathsf{io}\, P^3 \,;\, \Sigma^3 \rangle\!\rangle \text{ and}$$
$$\langle\!\langle P' \,;\, \Sigma' \rangle\!\rangle \lesssim_{sim} \langle\!\langle P^3 \,;\, \Sigma^3 \rangle\!\rangle$$

PROOF: by Lemma 2.5.20, completeness of leftmost reduction.

⟨2⟩2. $\exists P'', \Sigma''$ such that

1. $\langle\!\langle P \,;\, \Sigma \rangle\!\rangle[N/x] \to^*_{sto} \langle\!\langle \mathsf{io}\, P'' \,;\, \Sigma'' \rangle\!\rangle$, and

2. $\langle\!\langle P^3 \,;\, \Sigma^3 \rangle\!\rangle \; (\lesssim_{sim} \circ S^*_{pr} \circ \lesssim_{sim}) \; \langle\!\langle P'' \,;\, \Sigma'' \rangle\!\rangle$.

PROOF SKETCH: We construct an induction measure (n, b) where n is the number of steps in the leftmost reduction sequence and $b = 0$ if the head of P is x and $b = 1$ otherwise. $(n, b) < (n', b')$ if either $n < n'$ or ($n = n'$ and $b < b'$).

LET: $\mathrm{IH}(n, b) \stackrel{def}{=}$ if $\langle\!\langle P \,;\, \Sigma \rangle\!\rangle[M/x] \xrightarrow{n,b}_{left,sto} \langle\!\langle \mathsf{io}\, P^3 \,;\, \Sigma^3 \rangle\!\rangle$

then $\exists P'', \Sigma''$ s.t.

$\wedge\ \langle\!\langle P \,;\, \Sigma \rangle\!\rangle[N/x] \to^*_{sto} \langle\!\langle \mathsf{io}\, P'' \,;\, \Sigma'' \rangle\!\rangle$

$\wedge\ \langle\!\langle P^3 \,;\, \Sigma^3 \rangle\!\rangle \; (\lesssim_{sim} \circ S^*_{pr} \circ \lesssim_{sim}) \; \langle\!\langle P'' \,;\, \Sigma'' \rangle\!\rangle$

We prove the induction hypothesis by considering all possible forms of P (which necessarily covers cases where $n = 0, n > 0, b = 0$, and $b = 1$).

The possible forms of P are:

1. $xP_1 \ldots P_k, \quad k >= 0$
2. $(\lambda y.P_0)P_1 \ldots P_k, \quad k > 0$
3. $\mathsf{io}\, P_1$
4. $\mathsf{new}\, P_1 P_2$
5. $\mathsf{deref}\, P_1 P_2$
6. $\mathsf{update}\, P_1 P_2 P_3$
7. other PCF head redexes

$\langle 3\rangle 1.$ CASE: $P = xP_1 \ldots P_k, \quad k >= 0$

GIVEN: $\langle\!\langle xP_1 \ldots P_k \,;\, \Sigma\rangle\!\rangle[M/x] \overset{n,1}{\rightarrow}_{left,sto} \langle\!\langle \mathsf{io}\, P^3 \,;\, \Sigma^3\rangle\!\rangle$

PROVE: $\exists P'', \Sigma''$ s.t. $\langle\!\langle xP_1 \ldots P_k \,;\, \Sigma\rangle\!\rangle[N/x] \rightarrow^*_{sto} \langle\!\langle \mathsf{io}\, P'' \,;\, \Sigma''\rangle\!\rangle$

and $\langle\!\langle P^3 \,;\, \Sigma^3\rangle\!\rangle \; (\lesssim_{sim} \circ S^*_{pr} \circ \lesssim_{sim}) \; \langle\!\langle P'' \,;\, \Sigma''\rangle\!\rangle$

PROOF:

$\langle 4\rangle 1.$ left-hand side reduces:

$\langle\!\langle x\; P_1 \ldots P_k \,;\, \Sigma\rangle\!\rangle[M/x]$

$= \langle\!\langle M\; P_1 \ldots P_k \,;\, \Sigma\rangle\!\rangle[M/x]$

$\overset{n,0}{\rightarrow}_{left,sto} \langle\!\langle \mathsf{io}\, P^3 \,;\, \Sigma^3\rangle\!\rangle$

$\langle 4\rangle 2.$ $\exists P^4, \Sigma^4$ such that $\langle\!\langle MP_1 \ldots P_k \,;\, \Sigma\rangle\!\rangle[N/x] \rightarrow^*_{sto} \langle\!\langle \mathsf{io}\, P^4 \,;\, \Sigma^4\rangle\!\rangle$

and $\langle\!\langle P^3 \,;\, \Sigma^3\rangle\!\rangle \; (\lesssim_{sim} \circ S^*_{pr} \circ \lesssim_{sim}) \; \langle\!\langle P^4 \,;\, \Sigma^4\rangle\!\rangle$

PROOF: by IH$(n, 0)$ and $\langle 4\rangle 1$.

$\langle 4\rangle 3.$ $\exists P'', \Sigma''$ such that $\langle\!\langle NP_1 \ldots P_k \,;\, \Sigma\rangle\!\rangle[N/x] \rightarrow^*_{sto} \langle\!\langle \mathsf{io}\, P'' \,;\, \Sigma''\rangle\!\rangle$

and $\langle\!\langle P^4 \,;\, \Sigma^4\rangle\!\rangle \lesssim_{sim} \langle\!\langle P'' \,;\, \Sigma''\rangle\!\rangle$

PROOF: By $M \lesssim^{\bullet}_{ext} N{:}\tau'$.

$\langle 4\rangle 4.$ Q.E.D.

Note that $\langle\!\langle xP_1 \ldots P_k \,;\, \Sigma\rangle\!\rangle[N/x] = \langle\!\langle NP_1 \ldots P_k \,;\, \Sigma\rangle\!\rangle[N/x]$.

We have:

$\langle\!\langle P^3 \,;\, \Sigma^3\rangle\!\rangle \; (\lesssim_{sim} \circ S^*_{pr} \circ \lesssim_{sim}) \; \langle\!\langle P^4 \,;\, \Sigma^4\rangle\!\rangle \lesssim_{sim} \langle\!\langle P'' \,;\, \Sigma''\rangle\!\rangle$

$\langle 3\rangle 2.$ CASE: $P = (\lambda y.P_0)P_1 \ldots P_k, \quad k > 0$

GIVEN: $\langle\!\langle (\lambda y.P_0)P_1 \ldots P_k \,;\, \Sigma\rangle\!\rangle[M/x] \overset{n,0}{\rightarrow}_{left,sto} \langle\!\langle \mathsf{io}\, P^3 \,;\, \Sigma^3\rangle\!\rangle$

PROVE: $\exists P'', \Sigma''$ such that

$\langle\!\langle (\lambda y.P_0)P_1 \ldots P_k \,;\, \Sigma\rangle\!\rangle[N/x] \rightarrow^*_{sto} \langle\!\langle \mathsf{io}\, P'' \,;\, \Sigma''\rangle\!\rangle$ and

$\langle\!\langle P^3 \,;\, \Sigma^3\rangle\!\rangle \; (\lesssim_{sim} \circ S^*_{pr} \circ \lesssim_{sim}) \; \langle\!\langle P'' \,;\, \Sigma''\rangle\!\rangle$

PROOF:

⟨4⟩1. Left-hand side reduces:

$\langle\!\langle (\lambda y.P_0)P_1 \ldots P_k \ ; \Sigma \rangle\!\rangle[M/x]$

$= \langle\!\langle (\lambda y.P_0[M/x])P_1[M/x] \ldots P_k[M/x] \ ; \Sigma[M/x] \rangle\!\rangle$

$\xrightarrow{1}_{left,sto} \langle\!\langle (P_0[M/x][P_1/y])P_2[M/x] \ldots P_k[M/x] \ ;$

$\Sigma[M/x] \rangle\!\rangle$

$= \langle\!\langle (P_0[P_1/y])P_2 \ldots P_k \ ; \Sigma \rangle\!\rangle[M/x]$

$\overset{n-1,0}{\rightarrow}_{left,sto} \langle\!\langle \mathsf{io}\, P^3 \ ; \Sigma^3 \rangle\!\rangle$

Note that in the future we will generally omit the distribution and abstraction of substitutions, explicitly shown in the second and fourth lines above, when there is no interesting interaction between substitution and other calculations.

⟨4⟩2. Right-hand side reduces:

$\langle\!\langle (\lambda y.P_0)P_1 \ldots P_k \ ; \Sigma \rangle\!\rangle[N/x]$

$\rightarrow^1_{pcf} \langle\!\langle (P_0[P_1/y])P_2 \ldots P_k \ ; \Sigma \rangle\!\rangle[N/x]$

⟨4⟩3. Q.E.D.

PROOF: by IH$(n-1, 0)$.

⟨3⟩3. CASE: $P = \mathsf{io}\, P_1$

GIVEN: $\langle\!\langle \mathsf{io}\, P_1 \ ; \Sigma \rangle\!\rangle[M/x] \overset{n,0}{\rightarrow}_{left,sto} \langle\!\langle \mathsf{io}\, P^3 \ ; \Sigma^3 \rangle\!\rangle$

PROVE: $\exists P'', \Sigma''$ such that $\langle\!\langle \mathsf{io}\, P_1 \ ; \Sigma \rangle\!\rangle[N/x] \rightarrow^*_{sto} \langle\!\langle \mathsf{io}\, P'' \ ; \Sigma'' \rangle\!\rangle$
and $\langle\!\langle P^3 \ ; \Sigma^3 \rangle\!\rangle \ (\lesssim_{sim} \circ S^*_{pr} \circ \lesssim_{sim}) \ \langle\!\langle P'' \ ; \Sigma'' \rangle\!\rangle$

PROOF: Immediate. The only steps that the left-hand side can take are internal PCF reductions, and by Lemma 2.5.15, $\langle\!\langle \mathsf{io}\, P^3 \ ; \Sigma^3 \rangle\!\rangle \lesssim_{sim} \langle\!\langle \mathsf{io}\, P_1 \ ; \Sigma \rangle\!\rangle[M/x]$. Choose $P'' = P_1$ and we have $\langle\!\langle \mathsf{io}\, P^3 \ ; \Sigma^3 \rangle\!\rangle \lesssim_{sim} \langle\!\langle \mathsf{io}\, P_1 \ ; \Sigma \rangle\!\rangle[M/x] \ S^*_{pr} \ \langle\!\langle \mathsf{io}\, P'' \ ; \Sigma \rangle\!\rangle[N/x]$.

⟨3⟩4. CASE: $P = \mathsf{new}\, P_1 P_2$

GIVEN: $\langle\!\langle \mathsf{new}\, P_1 P_2 \ ; \Sigma \rangle\!\rangle[M/x] \overset{n,0}{\rightarrow}_{left,sto} \langle\!\langle \mathsf{io}\, P^3 \ ; \Sigma^3 \rangle\!\rangle$

PROVE: $\exists P'', \Sigma''$ such that $\langle\!\langle \mathsf{new}\, P_1 P_2 \ ; \Sigma \rangle\!\rangle[N/x] \rightarrow^*_{sto} \langle\!\langle \mathsf{io}\, P'' \ ; \Sigma'' \rangle\!\rangle$
and $\langle\!\langle P^3 \ ; \Sigma^3 \rangle\!\rangle \ (\lesssim_{sim} \circ S^*_{pr} \circ \lesssim_{sim}) \ \langle\!\langle P'' \ ; \Sigma'' \rangle\!\rangle$

PROOF:

⟨4⟩1. Left-hand side reduces:

$\langle\!\langle \mathsf{new}\, P_1 P_2 \ ; \Sigma \rangle\!\rangle [M/x]$

$\xrightarrow{1}_{left,sto} \langle\!\langle (P_2 l) \ ; \Sigma[l \mapsto P_1] \rangle\!\rangle [M/x], \quad l$ *fresh w.r.t.* M, P_1, P_2, Σ

$\overset{j<n,0}{\rightarrow}_{left,sto} \langle\!\langle \mathsf{io}\, P^3 \ ; \Sigma^3 \rangle\!\rangle$

$\langle 4 \rangle 2.$ LET: l' be fresh w.r.t. M, N, P_1, P_2, Σ

$\langle 4 \rangle 3.$ $\langle\!\langle (P_2 l) \ ; \Sigma[l \mapsto P_1] \rangle\!\rangle [M/x]$

$\alpha_{loc} \ \langle\!\langle (P_2 l') \ ; \Sigma[l' \mapsto P_1] \rangle\!\rangle [M/x]$

$\xrightarrow{j}_{left,sto} \langle\!\langle \mathsf{io}\, P^4 \ ; \Sigma^4 \rangle\!\rangle$ s.t.

$\langle\!\langle \mathsf{io}\, P^3 \ ; \Sigma^3 \rangle\!\rangle \ \alpha_{loc} \ \langle\!\langle \mathsf{io}\, P^4 \ ; \Sigma^4 \rangle\!\rangle$

By Lemma 2.5.10.

$\langle 4 \rangle 4.$ Right-hand side reduces:

$\langle\!\langle \mathsf{new}\, P_1 P_2 \ ; \Sigma \rangle\!\rangle [N/x]$

$\rightarrow_{sto} \langle\!\langle (P_2 l') \ ; \Sigma[l' \mapsto P_1] \rangle\!\rangle [N/x]$

$\rightarrow^*_{sto} \langle\!\langle \mathsf{io}\, P'' \ ; \Sigma'' \rangle\!\rangle$ s.t.

$\langle\!\langle P^4 \ ; \Sigma^4 \rangle\!\rangle \ (\lesssim_{sim} \circ S^*_{pr} \circ \lesssim_{sim}) \ \langle\!\langle P'' \ ; \Sigma'' \rangle\!\rangle$

By Step $\langle 4 \rangle 3$ and IH(j).

$\langle 4 \rangle 5.$ Q.E.D.

We have

$$\langle\!\langle P^3 \ ; \Sigma^3 \rangle\!\rangle \lesssim_{sim} \langle\!\langle P^4 \ ; \Sigma^4 \rangle\!\rangle \ (\lesssim_{sim} \circ S^*_{pr} \circ \lesssim_{sim}) \ \langle\!\langle P'' \ ; \Sigma'' \rangle\!\rangle$$

by Lemma 2.5.11 ($\alpha_{loc} \subseteq \lesssim_{sim}$).

$\langle 3 \rangle 5.$ CASE: $P = \mathsf{deref}\, P_1 P_2$

GIVEN: $\langle\!\langle \mathsf{deref}\, P_1 P_2 \ ; \Sigma \rangle\!\rangle [M/x] \overset{n,0}{\rightarrow}_{left,sto} \langle\!\langle \mathsf{io}\, P^3 \ ; \Sigma^3 \rangle\!\rangle$

PROVE: $\exists P'', \Sigma''$ s.t. $\langle\!\langle \mathsf{deref}\, P_1 P_2 \ ; \Sigma \rangle\!\rangle [N/x] \rightarrow^*_{sto} \langle\!\langle \mathsf{io}\, P'' \ ; \Sigma'' \rangle\!\rangle$

and $\langle\!\langle P^3 \ ; \Sigma^3 \rangle\!\rangle \ (\lesssim_{sim} \circ S^*_{pr} \circ \lesssim_{sim}) \ \langle\!\langle P'' \ ; \Sigma'' \rangle\!\rangle$

PROOF:

$\langle 4 \rangle 1.$ Left-hand side reduces:

$\langle\!\langle \mathsf{deref}\, P_1 P_2 \ ; \Sigma \rangle\!\rangle [M/x]$

$\rightarrow^*_{pcf} \langle\!\langle \mathsf{deref}\, l P_2 \ ; \Sigma \rangle\!\rangle [M/x]$ for some l

$\xrightarrow{1}_{left,sto} \langle\!\langle (P_2 Q) \ ; \Sigma \rangle\!\rangle [M/x]$, assuming $\Sigma(l) = Q$

$\overset{j<n,0}{\rightarrow}_{left,sto} \langle\!\langle \mathsf{io}\, P^3 \ ; \Sigma^3 \rangle\!\rangle$

$\langle 4 \rangle 2.$ if $P_1[M/x] \rightarrow^*_{left,pcf} l$, then $P_1[N/x] \rightarrow^*_{pcf} l$

PROOF: By substitution lemma for PCF, Lemma 2.5.21. At ref types, $\lesssim^{\bullet}_{ext}$-related terms coreduce to the same location constant.

⟨4⟩3. Right-hand side reduces:

$\langle\!\langle \mathsf{deref}\, P_1 P_2 \ ; \Sigma\rangle\!\rangle[N/x]$

$\to^*_{pcf} \langle\!\langle \mathsf{deref}\, l P_2 \ ; \Sigma\rangle\!\rangle[N/x]$ by ⟨4⟩2

$\to_{sto} \langle\!\langle (P_2 Q) \ ; \Sigma\rangle\!\rangle[N/x]$

⟨4⟩4. Q.E.D.

PROOF: by IH$(j, 0)$.

⟨3⟩6. CASE: $P = \mathsf{update}\, P_1 P_2 P_3$

GIVEN: $\langle\!\langle \mathsf{update}\, P_1 P_2 P_3 \ ; \Sigma\rangle\!\rangle[M/x] \overset{n,0}{\to}_{left,sto} \langle\!\langle \mathsf{io}\, P^3 \ ; \Sigma^3\rangle\!\rangle$

PROVE: $\exists P'', \Sigma''$ such that

$\langle\!\langle \mathsf{update}\, P_1 P_2 P_3 \ ; \Sigma\rangle\!\rangle[N/x] \to^*_{sto} \langle\!\langle \mathsf{io}\, P'' \ ; \Sigma''\rangle\!\rangle$ and

$\langle\!\langle P^3 \ ; \Sigma^3\rangle\!\rangle \ (\lesssim_{sim} \circ S^*_{pr} \circ \lesssim_{sim}) \ \langle\!\langle P'' \ ; \Sigma''\rangle\!\rangle$

PROOF:

⟨4⟩1. Left-hand side reduces:

$\langle\!\langle \mathsf{update}\, P_1 P_2 P_3 \ ; \Sigma\rangle\!\rangle[M/x]$

$\to^*_{pcf} \langle\!\langle \mathsf{update}\, l P_2 P_3 \ ; \Sigma\rangle\!\rangle[M/x]$

$\overset{1}{\to}_{left,sto} \langle\!\langle P_3 \ ; \Sigma[l \mapsto P_2]\rangle\!\rangle[M/x]$

$\overset{j<n,0}{\to}_{left,sto} \langle\!\langle \mathsf{io}\, P^3 \ ; \Sigma^3\rangle\!\rangle$

⟨4⟩2. if $P_1[M/x] \to^*_{left,pcf} l$, then $P_1[N/x] \to^*_{pcf} l$

PROOF: Again, by the substitution lemma for PCF, Lemma 2.5.21.

⟨4⟩3. Right-hand side reduces:

$\langle\!\langle \mathsf{update}\, P_1 P_2 P_3 \ ; \Sigma\rangle\!\rangle[N/x]$

$\to^*_{pcf} \langle\!\langle \mathsf{update}\, l P_2 P_3 \ ; \Sigma\rangle\!\rangle[N/x]$ by ⟨4⟩2

$\to_{sto} \langle\!\langle P_3 \ ; \Sigma[l \mapsto P_2]\rangle\!\rangle[N/x]$

⟨4⟩4. Q.E.D.

PROOF: by IH$(j, 0)$.

⟨3⟩7. CASE: other PCF head redexes

Are all similar to β case, step ⟨3⟩2.

⟨1⟩2. $(\langle\!\langle Q[M/x] \ ; \Sigma\rangle\!\rangle, \langle\!\langle Q[N/x] \ ; \Sigma\rangle\!\rangle) \in S^*_{pr}$

Because $fv(rng(\Sigma)) = \emptyset$ for any store component Σ of a well-formed state.

□

Lemma 2.6.2 (Substitution on Terms) *If $x{:}\tau' \vdash Q{:}\tau''$ and $M \lesssim^{\bullet}_{ext} N{:}\tau'$, then*

$$Q[M/x] \lesssim^{\bullet}_{ext} Q[N/x]{:}\tau''.$$

PROOF SKETCH: We define a type-indexed family of relations S. We then show that for all types τ, $S_\tau \subseteq [S]_{ext}$ which implies that $S \subseteq \lesssim^{\bullet}_{ext}$. Finally, we show that $(Q[M/x], Q[N/x]) \in S$.

PROOF:

LET: $S_\tau \stackrel{def}{=} \{(P[M/x], P[N/x]) \mid x{:}\tau' \vdash P{:}\tau\}$, for all types τ and terms P
$S \stackrel{def}{=} \bigcup_\tau S_\tau$.

⟨1⟩1. $S_\tau \subseteq [S]_{ext}$, for all τ

PROOF:

Consider the cases of type τ. For each τ, consider any pair,

$$(P[M/x], P[N/x]) \in S_\tau :$$

⟨2⟩1. CASE: $int, bool, ref\,(\tau'')$

PROOF: We can show that $P[M/x] \lesssim^{\bullet}_{ext} P[N/x]{:}\tau$ by appealing to the substitution lemma for PCF, Lemma 2.5.21.

⟨2⟩2. CASE: $\alpha{\times}\beta$, $\alpha{+}\beta$, $\alpha{\to}\beta$

PROOF: All three cases follow immediately from the definition of S. In particular, if $M \lesssim^{\bullet}_{ext} N{:}\tau'$, then it is immediate that:

$$\begin{array}{ll}
(\pi_1\, P[M/x])\ S_\alpha\ (\pi_1\, P[N/x]), & \text{for } x{:}\tau' \vdash P{:}\alpha{\times}\beta \\
(\pi_2\, P[M/x])\ S_\beta\ (\pi_2\, P[N/x]), & \text{for } x{:}\tau' \vdash P{:}\alpha{\times}\beta \\
(\mathsf{case}\, P[M/x]\ P_1\ P_2)\ S_\gamma\ (\mathsf{case}\, P[N/x]\ P_1\ P_2), & \text{for } x{:}\tau' \vdash P{:}\alpha{+}\beta, \text{ and} \\
(P[M/x]\ R)\ S_\beta\ (P[N/x]\ R) & \text{for } x{:}\tau' \vdash P{:}\alpha{\to}\beta
\end{array}$$

for closed terms P_1, P_2, and R.

⟨2⟩3. CASE: pr

By the definitions of S_{pr} and $[-]^{pr}_{ext}$, we must show: for all P s.t. $x{:}\tau' \vdash P{:}pr$, for all stores Σ,

$$\langle\!\langle P[M/x]\ ;\ \Sigma\rangle\!\rangle \lesssim_{sim} \langle\!\langle P[N/x]\ ;\ \Sigma\rangle\!\rangle,$$

which is immediate by Lemma 2.6.1.

$\langle 1\rangle 2.$ $S \subseteq \lesssim^{\bullet}_{ext}$

PROOF: By coinduction.

$\langle 1\rangle 3.$ $(Q[M/x], Q[N/x]) \in S$

$\langle 1\rangle 4.$ Q.E.D.

□

2.7 Precongruence of the Extensional Preorder

Using the substitution lemma, we can prove the following precongruence property:

Lemma 2.7.1 (precongruence of $\lesssim_{ext}$)

$\lesssim_{ext}$ is a precongruence – that is, if $\Gamma \vdash M \lesssim_{ext} N{:}\tau$, then for all contexts $C[-_\tau]$ which respect Γ, if $\exists\Gamma', \tau'$ s.t. $\Gamma' \vdash C[M], C[N]{:}\tau'$, then

$$\Gamma' \vdash C[M] \lesssim_{ext} C[N]{:}\tau'.$$

PROOF:

To prove that $\lesssim_{ext}$ is a precongruence, we need to show that $\lesssim_{ext}$ is a preorder and is preserved by the six productions of the grammar for the metalanguage presented on Page 14:

$\langle 1\rangle 1.$ $\lesssim_{ext}$ is a preorder

By Lemma 2.4.12.

$\langle 1\rangle 2.$ CASE: $\Gamma \vdash n \lesssim_{ext} n{:}int \quad \forall\Gamma, n$

By reflexivity of $\lesssim_{ext}$.

$\langle 1\rangle 3.$ CASE: $\Gamma \vdash l^\tau \lesssim_{ext} l^\tau{:}ref(\tau) \quad \forall\Gamma, l^\tau$

By reflexivity of $\lesssim_{ext}$.

$\langle 1\rangle 4.$ CASE: if $c{:}\tau$, then $\Gamma \vdash c \lesssim_{ext} c{:}\tau \quad \forall\Gamma$

By reflexivity of $\lesssim_{ext}$.

$\langle 1\rangle 5.$ CASE: if $\Gamma \vdash x{:}\tau$, then $\Gamma \vdash x \lesssim_{ext} x{:}\tau$

By reflexivity of $\lesssim_{ext}$.

$\langle 1\rangle 6.$ CASE: if $\Gamma, x{:}\tau' \vdash M \lesssim_{ext} N{:}\tau$, then

$$\Gamma \vdash (\lambda x{:}\tau'.M) \lesssim_{ext} (\lambda x{:}\tau'.N){:}\tau'{\to}\tau$$

From the definition of $\lesssim_{ext}$ and Lemma 2.5.17, namely that PCF reduction preserves extensional equivalence. closing substitutions.

$\langle 1\rangle 7.$ CASE: if $\Gamma \vdash M \lesssim_{ext} M'{:}\tau'{\to}\tau$ and $\Gamma \vdash N \lesssim_{ext} N'{:}\tau'$, then

$$\Gamma \vdash (MN) \lesssim_{ext} (M'N'){:}\tau.$$

We split this case into two subcases:

$\langle 2\rangle 1.$ CASE: if $\Gamma \vdash M \lesssim_{ext} M'{:}\tau'{\to}\tau$, then for all closed N of type τ',

$$\Gamma \vdash (M\ N) \lesssim_{ext} (M'\ N){:}\tau$$

Follows from definition of $\lesssim_{ext}$ at function type.

$\langle 2\rangle 2.$ CASE: if $\Gamma \vdash N \lesssim_{ext} N'{:}\tau'$, then for all closed M' of type $\tau'{\to}\tau$,

$$\Gamma \vdash (M'\ N) \lesssim_{ext} (M'\ N'){:}\tau$$

LET: σ be a closing substitution for both sides.

PROVE: $(M'\ N)\sigma \lesssim^{\bullet}_{ext} (M'\ N')\sigma{:}\tau$

$\langle 3\rangle 1.$ $N\sigma \lesssim^{\bullet}_{ext} N'\sigma{:}\tau$

By $\Gamma \vdash N \lesssim_{ext} N'{:}\tau'$.

$\langle 3\rangle 2.$ $(M'\ x)[(N\sigma)/x] \lesssim^{\bullet}_{ext} (M'\ x)[(N'\sigma)/x]$

By Substitution Lemma for terms, Lemma 2.6.2.

$\langle 3\rangle 3.$ Q.E.D.

Do substitution for x in step $\langle 3\rangle 2$ and lift σ outside application (M' is closed).

$\langle 2\rangle 3.$ Q.E.D.

By transitivity of $\lesssim_{ext}$.

□

2.8 The Extensionality Theorem

Given that $\lesssim_{ext}$ is a precongruence, the extensionality result we have been aiming for follows easily.

Theorem 2.8.1 (Extensionality)

$$\Gamma \vdash M \lesssim_{ext} N{:}\tau \Leftrightarrow \Gamma \vdash M \lesssim_{\forall C} N{:}\tau$$

PROOF:

$\langle 1 \rangle 1.$ $\Rightarrow$

Assume $\Gamma \vdash M \lesssim_{ext} N{:}\tau$. By the precongruence of $\lesssim_{ext}$, Lemma 2.7.1, for all contexts $C[-_\tau]$ of type pr which respect Γ, $\Gamma \vdash C[M] \lesssim_{ext} C[N]{:}pr$. By the definition of $\lesssim_{ext}$ at type pr, the result follows.

$\langle 1 \rangle 2.$ $\Leftarrow$

ASSUME: $\Gamma \vdash M \lesssim_{\forall C} N{:}\tau$

That is, for all contexts $C[-_\tau]{:}pr$ and all stores Σ,

$\langle\!\langle C[M]\ ;\ \Sigma \rangle\!\rangle \lesssim_{sim} \langle\!\langle C[N]\ ;\ \Sigma \rangle\!\rangle$.

PROVE: $\Gamma \vdash M \lesssim_{ext} N{:}\tau$

That is, for all closing substitutions σ,

$M\sigma \lesssim^{\bullet}_{ext} N\sigma{:}\tau$.

That is, for all base types o, and type contexts $T_\tau[-]{:}o$,

$T_\tau[M\sigma] \lesssim^{\bullet}_{o} T_\tau[N\sigma]$.

$\langle 2 \rangle 1.$ For any closing substitution σ, $\exists C_\sigma$ s.t.

$(M\sigma) \lesssim^{\bullet}_{ext} (N\sigma){:}\tau \Leftrightarrow C_\sigma[M] \lesssim^{\bullet}_{ext} C_\sigma[N]{:}\tau$

PROOF: It is easy to construct contexts to capture free variables with the same terms (after β) as σ. By Lemma 2.5.17, β-reduction preserves $\lesssim^{\bullet}_{ext}$.

$\langle 2 \rangle 2.$ $C_\sigma[M] \lesssim_{\forall C} C_\sigma[N]{:}\tau$

PROOF: By $\Gamma \vdash M \lesssim_{\forall C} N{:}\tau$ and the fact that $\lesssim_{\forall C}$ is a precongruence (by definition).

$\langle 2 \rangle 3.$ $C_\sigma[M] \lesssim_{\forall C} C_\sigma[N]{:}\tau \Rightarrow C_\sigma[M] \lesssim^{\bullet}_{ext} C_\sigma[N]{:}\tau$

ASSUME: For all contexts $C[-_\tau]{:}pr$ and stores Σ,

$$\langle\!\langle C[C_\sigma[M]]\ ;\ \Sigma\rangle\!\rangle \lesssim_{sim} \langle\!\langle C[C_\sigma[N]]\ ;\ \Sigma\rangle\!\rangle$$

PROVE: For all base types o and type contexts $T_\tau[-]{:}o$,

$$T_\tau[C_\sigma[M]] \lesssim^\bullet_o T_\tau[C_\sigma[N]]$$

PROOF: For $o = pr$, result is immediate.

For other base types, we must construct contexts of type pr which distinguish PCF reduction to different base values. Consider int type:

LET: $C_{int} \stackrel{def}{=}$

$$\{(\mathsf{if}\ (n =_{int} C_\sigma[-])(\mathsf{write}\,1\,\mathsf{stop})(\mathsf{write}\,0\,\mathsf{stop})) \mid n \text{ an integer}\}$$

$\langle 3\rangle 1.$ For any $T_\tau[-]{:}int$

$$T_\tau[C_\sigma[M]] \lesssim^\bullet_{int} T_\tau[C_\sigma[N]] \Leftrightarrow$$

$$\forall C_n \in C_{int}, \Sigma\ .\ \langle\!\langle C_n[M]\ ;\ \Sigma\rangle\!\rangle \lesssim_{sim} \langle\!\langle C_n[N]\ ;\ \Sigma\rangle\!\rangle$$

PROOF: which is true by assumption.

Other base types are similar to int. For $ref(\tau)$ types, we can distinguish different location constants with $\mathsf{eq?}_{loc}{}^\tau$.

□

We now present a useful property of the metalanguage which follows easily from extensionality and other previous results.

2.8.1 Extensional Equivalence at Sum Types

At sum types, there is an alternative method for showing extensional equivalence, using the injection operators rather than case.

Proposition 2.8.2 *If $M, N{:}\alpha{+}\beta$, then $M \lesssim^\bullet_{ext} N{:}\alpha{+}\beta$ iff*

1. *if $M \rightarrow^*_{pcf} (\mathsf{inl}\,M')$, then $\exists N'$* s.t. *$N \rightarrow^*_{pcf} (\mathsf{inl}\,N')$ and $M' \lesssim^\bullet_{ext} N'{:}\alpha$, and*

2. *if* $M \to^*_{pcf} (\mathsf{inr}\, M')$, *then* $\exists N'$ s.t. $N \to^*_{pcf} (\mathsf{inr}\, N')$ *and* $M' \lesssim^{\bullet}_{ext} N' {:} \beta$.

PROOF:

(⇒) In each case, N must reduce to the correct form, $(\mathsf{inl}\, N')$ or $(\mathsf{inr}\, N')$, as it easy to construct a counterexample to the assumption that the terms are in $\lesssim^{\bullet}_{ext}$ (e.g. at type $\gamma = pr$) otherwise. Result follows from Extensionality (Theorem 2.8.1) and from the fact that PCF reduction preserves $\cong_{\forall C}$ (Lemma 2.5.16).

(⇐) Consider each type γ. Result follows by precongruence of $\lesssim_{ext}$, Lemma 2.7.1. For example, in pr case we end up needing to show $\langle\!\langle P_1 M' \,;\, \Sigma \rangle\!\rangle \lesssim_{sim} \langle\!\langle P_1 N' \,;\, \Sigma \rangle\!\rangle$ with $M' \lesssim_{ext} N'$, which follows from precongruence of $\lesssim_{ext}$. As another example, in the function case, we end up needing to show something like $((P_1 M')P_3) \lesssim_{ext} ((P_1 N')P_3)$, which again follows from the precongruence of $\lesssim_{ext}$.

Chapter 3

Examples from the Literature

In this chapter, we work through some examples from the literature. The exercise of doing these examples serves two main purposes: first, it allows our methods to be compared to others; second, these examples are short and provide a good introduction in preparation for the more substantial applications presented in following chapters.

3.1 Some Meyer-Sieber Examples

The paper by Meyer and Sieber, [MS88], has long been a source of good test cases for proving the observable equivalence of program fragments in an imperative language. While much of the research which follows the Meyer-Sieber paper focuses on formalizing the stack discipline of Algol, e.g. [OT95], the examples given are suitable if only a heap is assumed for storage. We present the proofs of equivalence for two of their examples to demonstrate the application of our techniques.

3.1.1 Meyer-Sieber Example 1

The first example given by Meyer and Sieber is as follows:

The block below is replaceable simply by the call P.

begin

  new x;

  P; *% P is declared elsewhere*

end

A straightforward CPS translation from this fragment of Algol to our metalanguage, where all blocks get translated to terms of type $pr \to pr$, yields the following term for the block above:

$$[\![lhs]\!] = (\lambda\kappa{:}pr.\,\mathsf{new}\,0\ (\lambda x{:}ref(int).P\ \kappa))$$

The term takes a run-time continuation κ, allocates a fresh location, initializes the location to 0, binds the new location to x, and invokes the closed metalanguage term P with continuation κ. P must have type $pr \to pr$.

The translation of a call to P is simply P.

The Extensionality Theorem says that to prove the two translations operationally equivalent we need only show that for all stores Σ and all closed terms K of type pr,

$$\langle\!\langle(\lambda\kappa.\,\mathsf{new}\,0\ (\lambda x.P\ \kappa))\ K\ ;\Sigma\rangle\!\rangle \approx_{sim} \langle\!\langle P\ K\ ;\Sigma\rangle\!\rangle$$

Because $\to^*_{sto}$ preserves $\approx_{sim}$, we may reduce the left-hand side of the above equation:

$$\begin{aligned}[\![lhs]\!] &\to^*_{sto} \langle\!\langle(P\ K)[l/x]\ ;\Sigma[l \mapsto 0]\rangle\!\rangle && \textit{(l fresh)}\\ &= \langle\!\langle(P\ K)\ ;\Sigma[l \mapsto 0]\rangle\!\rangle && \textit{(P and K are closed)}\end{aligned}$$

which is the same as $\langle\!\langle P\ K\ ;\Sigma\rangle\!\rangle$, up to a consistent renaming of locations (i.e. in α_{loc}, with partial bijection $\Theta = \emptyset$). By Lemma 2.5.11, the two blocks are equivalent.

Thus we are able to prove the equivalence of the first Meyer-Sieber example by simple calculation and direct use of a few properties of the metalanguage.

3.1.2 Meyer-Sieber Example 4

Example 4 from [MS88] is given as:

The block below always diverges.

begin

  new x; **new** y;

  procedure *Twice*; **begin** $y := 2 * contents(y)$ **end**;

  $x := 0;\ y := 0;$

  $Q(Twice)$;   % *Q is declared elsewhere*

  if $contents(x) = 0$ **then** *diverge* **fi**

end

In a language with I/O, the above block is *not* necessarily equivalent to *diverge*, as the procedure Q might do I/O. So we will undertake to prove the equivalence of the following two blocks:

---

begin

  new x; **new** y;

  procedure *Twice*; **begin** $y := 2 * contents(y)$ **end**;

  $x := 0;\ y := 0;$

  $Q(Twice)$;   % *Q is declared elsewhere*

  if $contents(x) = 0$ **then** *skip* **else** *diverge* **fi**

end

is operationally equivalent to:

begin

  new x; **new** y;

  procedure *Twice*; **begin** $y := 2 * contents(y)$ **end**;

  $x := 0;\ y := 0;$

  $Q(Twice)$;   % *Q is declared elsewhere*

end

---

where *skip* does nothing.

A translation from Algol to our metalanguage gives us the following metalanguage term for the original block:

$$[\![orig]\!] = (\lambda\kappa{:}pr.\, \mathsf{new}\, 0\ (\lambda x{:}ref\,(int).\, \mathsf{new}\, 0\ (\lambda y{:}ref\,(int).$$
$$f_Q\ Twice\ (\mathsf{deref}\, x\ (\lambda v_x.\, zero?\ v_x\ \kappa\ \Omega)))))$$

where $Twice$ is defined as:

$$Twice \stackrel{def}{=} (\lambda\kappa'{:}pr.\, \mathsf{deref}\, y\ (\lambda v_y.\, \mathsf{update}\, y\ (*\ 2\ v_y)\ \kappa'))$$

and where f_Q is a variable of type $(pr \to pr) \to pr \to pr$.

The $[\![orig]\!]$ term takes a run-time continuation κ, allocates fresh locations for x and y, initializes them to 0, and then continues with x and y bound to the fresh locations. The body of the block invokes whatever procedure is bound to f_Q, sending it $Twice$ and a continuation which checks if x is zero, continues with κ if so and diverges if not. $Twice$, in which y will be bound to a location at run-time, takes a run-time continuation κ', gets the current value of y, binds the value to v_y, then updates y with $(2 * v_y)$ and continues with κ'. The entire block has type $pr \to pr$.

The translation of the replacement block, $[\![repl]\!]$, is exactly the same as $[\![orig]\!]$, except that the $(\mathsf{deref}\, x \ldots)$ continuation is replaced by simply κ.

To prove that the two translations are operationally equivalent, we appeal to the Extensionality Theorem and set out to prove that for all terms K of type $pr \to pr$, all closing substitutions $[Q/f_q]$ and all stores Σ,

$$\langle\!\langle [\![orig]\!][Q/f_q]\ K\ ;\Sigma\rangle\!\rangle \approx_{sim} \langle\!\langle [\![repl]\!][Q/f_q]\ K\ ;\Sigma\rangle\!\rangle$$

The left-hand side of the above equation will take a number of steps to the following state:

$$\langle\!\langle (Q\ Twice\ (\mathsf{deref}\, x\ (\lambda v_x.\, zero?\ v_x\ K\ \Omega)))[l_x/x, l_y/y]\ ;$$
$$\Sigma[l_x \mapsto 0, l_y \mapsto 0]\rangle\!\rangle \qquad (l_x, l_y\ fresh)$$

Similarly, the right-hand side goes to:

$$\langle\!\langle (Q\ \mathit{Twice}\ K)[l_x/x, l_y/y]\ ;$$
$$\Sigma[l_x \mapsto 0, l_y \mapsto 0]\rangle\!\rangle$$

Actually, the locations might be different on the two sides but the metalanguage theory allows us to ignore differences up to α_{loc}.

We now construct a candidate simulation relation S, which includes the two states above, and we prove that S is indeed a simulation.

$$\begin{aligned} S \stackrel{def}{=} \{\ & (\langle\!\langle M\ ;\ \Sigma\rangle\!\rangle\sigma_1,\ \langle\!\langle M\ ;\ \Sigma\rangle\!\rangle\sigma_2) \mid \\ & \mathit{fv}(M, \mathit{rng}(\Sigma)) \subseteq \{z\}, \\ & \sigma_1 = (\mathsf{deref}\, l_x\ (\lambda v_x.\, \mathit{zero?}\ v_x\ K\ \Omega))/z, \\ & \sigma_2 = K/z, \\ & \Sigma(l_x) = 0, \\ & l_x \notin \mathit{locs}(M, K, \mathit{rng}(\Sigma))\ \} \end{aligned}$$

The pair of states in which we are interested is clearly in S, with $M = (Q\ \mathit{Twice}\ z)$. Because l_x is allocated fresh, we know that Q and K contain no references to l_x.

The proof that S is indeed a simulation is a straightforward coinduction – namely, we prove that $S \subseteq [S]_{sim}$. The proof of inclusion for each clause in the definition of $[-]_{sim}$ is by induction on the length of a standard reduction of the left-hand side to an I/O operator. The cases where M is of the form $(\mathsf{update}\, M_1\, M_2\, M_3)$ or $(\mathsf{deref}\, M_1\, M_2)$, and M_1 PCF-reduces to a location constant l, are handled by appealing to the definition of S to show that $l \neq l_x$. In the new and update cases, the $(\mathsf{deref}\, l_x \ldots)$ term on the left-hand side can make its way into the store, which necessitates allowing z in the free variables of the store and applying the substitutions to the store as well as the term. For the case where M is headed by the free variable z, both sides reduce quickly to K.

The simulation S characterizes the relevant aspects of the Meyer-Seiber example being considered – namely that x's location l_x is never updated (or, more exactly, that the only references to l_x are within a term of the form $(\mathsf{deref}\, l_x \ldots)$). The fact that *Twice* sets y to twice its current value is unimportant, as is whether or not Q actually calls *Twice* or whether Q modifies other locations in the store.

3.2 Examples from Ian Stark

In his thesis, [Sta94], Ian Stark outlines a small language called the *nu-calculus* which adds dynamic names to the simply-typed lambda calculus. A rich theory of the nu-calculus is developed, using category theory and logical relations. The methods for reasoning within the nu-calculus are then applied to a more realistic language called *Reduced ML*, which supports integer reference types, and several example equivalences are presented. Many of the Reduced ML examples are inspired by the Meyer-Sieber examples. We work through one of the more interesting examples, which we call "count-up-count-down".

3.2.1 Count-up, Count-down

Example 10 in [Sta94][Ch. 5, Sec. 4] is similar to an example used by O'Hearn and Tennent in [OT95] and demonstrates *representation independence* – in this case of a simple counter.

```
let val r = ref 0
in
  fn(x : int) ⇒ (r :=!r + x; !r)
end
```

is operationally equivalent at type int → int to:

```
let val r = ref 0
in
  fn(x : int) ⇒ (r :=!r − x; −!r)
end
```

Both blocks return a function which maintains a running total of the integers to which it has been applied. The second implementation, though, perversely maintains its counter as a negative value. Barring issues of signed integer representations and overflow, the two implementations should be operationally equivalent.

A straightforward CPS translation from the first Reduced ML fragment above yields the following metalanguage term:

$$\begin{aligned}[\![orig]\!] = &(\lambda\kappa.\,\mathsf{new}\,0\;(\lambda r.\kappa\;(\lambda x\kappa'.\,\mathsf{deref}\,r\\ &\quad(\lambda v_r.\,\mathsf{update}\,r\;(v_r+x)\;(\mathsf{deref}\,r\;\kappa')))))\end{aligned}$$

whereas the translation of the negatively-disposed fragment gives us:

$$\begin{aligned}[\![repl]\!] = &(\lambda\kappa.\,\mathsf{new}\,0\;(\lambda r.\kappa\;(\lambda x\kappa'.\,\mathsf{deref}\,r\\ &\quad(\lambda v_r.\,\mathsf{update}\,r\;(v_r-x)\;(\mathsf{deref}\,r\;(\lambda v_r.\kappa'\;(-v_r)))))))\end{aligned}$$

For all continuation terms K and all stores Σ, the two terms behave as follows:

$$\langle\!\langle [\![orig]\!]\,K\;;\Sigma\rangle\!\rangle \rightarrow^*_{left,sto} \langle\!\langle K\;L\;;\Sigma[l_r\mapsto 0]\rangle\!\rangle,\ \text{where}$$

$$L = (\lambda x\kappa'.\,\mathsf{deref}\,r(\lambda v_r.\,\mathsf{update}\,r\;(v_r+x)\;(\mathsf{deref}\,r\;\kappa')))$$

and

$$\langle\!\langle [\![repl]\!]\,K\;;\Sigma\rangle\!\rangle \rightarrow^*_{sto} \langle\!\langle K\;R\;;\Sigma[l_r\mapsto 0]\rangle\!\rangle,\ \text{where}$$

$$R = (\lambda x\kappa'.\,\mathsf{deref}\,r(\lambda v_r.\,\mathsf{update}\,r\;(v_r-x)(\mathsf{deref}\,r\;(\lambda v_r.\kappa'\;(-v_r)))))$$

The obvious candidate simulation relation which characterizes the relationship maintained between the two sides as they run is as follows:

$$\begin{aligned}S \stackrel{def}{=} \{ (&\langle\!\langle M\;;\Sigma_1\rangle\!\rangle\sigma_1,\ \langle\!\langle M\;;\Sigma_2\rangle\!\rangle\sigma_2) \mid\\ &\sigma_1=[L/z],\quad \sigma_2=[R/z],\quad fv(M, rng(\Sigma_1,\Sigma_2))\subseteq\{z\}\\ &l_r\notin locs(M, rng(\Sigma_1,\Sigma_2)),\\ &\Sigma_1(l_r) = -\Sigma_2(l_r)\}\end{aligned}$$

The states we are interested in are clearly in S, with $M=(Kz)$, and it is a straightforward to prove S a simulation. The only interesting case is when $M = z\,M_1\,M_2$, with $M_1{:}int$, and in this case both sides clearly stay in S. It is also easy to show that all ref-typed subterms never PCF-reduce to l_r, by the constraint on locations in the definition of S.

Chapter 4

Twobit Optimizations: An Exercise in Almost Denotational Semantics

> Many well-known and widely studied transformation methods have not been proved to be correct, and it seems that the correctness problem has received little attention because of an implicit (and incorrect) assumption that it is sufficient to argue the correctness at the level of the basic steps.
>
> – *David Sands* [San96]

In this and the two following chapters, we detail the use of our metalanguage to prove the correctness of several source-to-source transformations for the Scheme programming language. The transformations we consider are representative of a wide class of optimizations used in compilers for Scheme and for other functional programming languages. The Twobit Scheme compiler [CH94] by Will Clinger and Lars Hansen provides the actual code we use as a basis for our definitions of the transformations.

4.1 Motivation

One of Scheme's most notable features is a long-standing and well-developed denotational semantics for the language [Cli84, CR91, IEE91]. The denotational semantics, which we will refer to as the R^{n}RS semantics, has been useful in providing a formal basis to guide implementors of the language. However, many universally accepted contextual equivalences are false in the standard denotational semantics [CR91, Section 7.2]. There are several reasons that the denotational semantics for Scheme cannot equate such contextual equivalences:

1. The initial continuation for evaluation of Scheme programs is not defined,
2. The domain of answers is undefined,
3. The evaluation of constants is not defined.

As a simple example, consider the Scheme expressions

$e_1 = ((\mathsf{lambda}\ (x)\ x)\ e)$

$e_2 = e$

which we would expect to be contextually equivalent.

In the R^{n}RS semantics, a Scheme expression e evaluates to $\mathcal{E}[\![e]\!]$, which is a function from environments ρ, expression continuations κ, and stores σ, to answers. According to the R^{n}RS semantics, $\mathcal{E}[\![e_1]\!]$ evaluates e, getting an expressed value, say v, then puts that v in the store at some location, say α, then retrieves v from the store at address α and sends it to the expression's continuation. $\mathcal{E}[\![e_2]\!]$, on the other hand, evaluates e and sends the value to the continuation, doing no store allocation or access (for x anyway). The important thing to keep in mind is that the continuation of $\mathcal{E}[\![e_1]\!]$ is run in a *different store* than the continuation of $\mathcal{E}[\![e_2]\!]$. A sufficiently nasty continuation can distinguish the two situations:

$$\kappa = (\lambda\epsilon^*\sigma.(\sigma(\alpha \mid \mathrm{L}) \downarrow 2) \rightarrow \mathit{wrong}, \mathit{unspecified})$$

The continuation κ looks at the "initialized" flag of the specific location α, and if it has been initialized it halts. Of course it may be that the evaluation of e to v allocates location

α in both sides, in which case the two sides will behave the same. However, in the case where α is the location allocated for the x in e_1, the two sides will be different and are therefore not equivalent.

4.2 Overview

A notable aspect of the set of optimizations we prove correct is that they are *order-dependent.* That is, later optimizations rely on earlier optimizations having taken place. We factor the suite of optimizations into a series of source-to-source and language-to-language transformations. Each transformation is shown to preserve operational behavior.

The optimizations we prove correct – in particular assignment elimination – translate the source Scheme program into a different intermediate language. This is representative of many, if not all, compiler implementations. Intermediate languages chosen for translation from source languages will typically have the following characteristics:

1. Redundant constructions in the source language will map to the same intermediate language construct, and

2. There will be specialized constructs in the intermediate language whose use represents derived information about the input program. For example, if the source language supports generic arithmetic but the compiler knows that two values will always be integers, the intermediate language might support integer-specific arithmetic functions. In assignment elimination, the intermediate language Scheme$_{cell}$ distinguishes between value-bound and location-bound variables, whereas the source language, Scheme, does not.

The framework presented in this chapter handles this important technique of translation from the source programming language to an intermediate language, and then possibly to another intermediate language, and so on. The task of proving a sequence of transformations and translations correct can be presented abstractly as follows. Suppose that we have a set of languages, $L_1, \ldots, L_n$, with L_1 being the source programming language. We have a corresponding series of language-to-language transformations, $xform_{i,j}$ which transform language L_i into language L_j. Finally, for each language L_i, we have a translation $[\![-]\!]_i$ from L_i into our metalanguage. We must show that the following diagram commutes:

$$\begin{tikzcd}
L_1 \arrow[r, "{[\![-]\!]_1}"] \arrow[d, "{xform_{1,2}}"'] & metalanguage \arrow[d, leftrightarrow, "{\cong_{\forall C}}"] \\
L_2 \arrow[r, "{[\![-]\!]_2}"] \arrow[d, "{xform_{2,3}}"'] & metalanguage \arrow[d, leftrightarrow, "{\cong_{\forall C}}"] \\
L_3 \arrow[r, "{[\![-]\!]_3}"] \arrow[d, dotted, no head] & metalanguage \arrow[d, dotted, no head] \\
{} & {}
\end{tikzcd}$$

Twobit makes clever use of Scheme as its own intermediate code, with successive passes and optimizations producing correct, runnable Scheme. The use of a single syntax for multiple intermediate languages reduces the number of data structures that must be manipulated by a compiler, but has no theoretical significance.

In our case, we are free to use successive intermediate languages, each with its own semantics (i.e. translation to the metalanguage), for successive transformations, so we need no such cleverness.

4.3 Translation from Scheme to Metalanguage

The syntax of the subset of Scheme with which we will be concerned initially is given by the grammar in Figure 7. The language consists of expressions e and programs p.

Figure 8 contains a compositional, call-by-value translation from the Scheme subset into the metalanguage. The resulting metalanguage terms are in *continuation passing style (CPS)*. The translation is a standard CPS translation – see [FWH92, App92, Sto77] for introductions to CPS. We use a "natural" model, without an explicit environment. Free variables in expressions are left free in the translation, with type $ref(Val)$. As demonstrated

$$
\begin{array}{lll}
e ::= x \mid n \mid (\mathsf{lambda}\ (x \ldots)\ e) \mid (e\ e \ldots) & \text{expressions} \\
\quad \mid\ (\mathsf{set!}\ x\ e) \mid (\mathsf{letrec}\ ((x\ e) \ldots)\ e) & \\
p ::= \mathsf{run}\ e & \text{programs}
\end{array}
$$

where x ranges over identifiers and n ranges over integers.

Figure 7: A subset of Scheme

in [Wan90], this setup is equivalent to the more common environment model, such as is used in [Cli84], where the type of an environment would be $\mathit{Locs} \rightarrow \mathit{ref}(\mathit{Val})$.

We use $\langle -, \ldots, - \rangle$ and $\mathit{listref}_i$ for the evident list operations. The helper function $\mathit{checkargs} : \mathit{int} \rightarrow \mathit{Val}^* \rightarrow \mathit{pr} \rightarrow \mathit{pr}$ checks that the list of values is of the correct length and stops if not.

The type Val of expressed values is the recursive sum type of expressed values:

$$
\begin{array}{ll}
\mathit{Val} = N + F & \text{expressed values} \\
N = \mathit{int} & \text{integers} \\
F = \mathit{Val}^* \rightarrow (\mathit{Val} \rightarrow \mathit{pr}) \rightarrow \mathit{pr} & \text{procedures}
\end{array}
$$

The denoted values are of type $\mathit{ref}(\mathit{Val})$.

For each summand S of the Val type, we include CPS injection and extraction combinators, $\mathit{in}_S : S \rightarrow (\mathit{Val} \rightarrow \mathit{pr}) \rightarrow \mathit{pr}$ and $\mathit{out}_S : \mathit{Val} \rightarrow (S \rightarrow \mathit{pr}) \rightarrow \mathit{pr}$. For example, the in_N function can be defined as:

$$\mathit{in}_N \stackrel{\mathit{def}}{=} (\lambda n{:}\mathit{int}.\lambda\kappa{:}\mathit{Val} \rightarrow \mathit{pr}.\kappa\ (\mathsf{inl}_{N+F}\ n))$$

The out_S functions invoke an error (print a number and halt) if the argument is not of the correct type (checked with the case construction). For example, the out_N function can be defined as:

$$\mathit{out}_N \stackrel{\mathit{def}}{=} (\lambda v{:}\mathit{Val}.\lambda\kappa{:}\mathit{int} \rightarrow \mathit{pr}.\ \mathsf{case}\ v\ \kappa\ (\lambda f{:}F.\ \mathit{error}))$$

The rule for integers in Figure 8 simply injects the appropriate integer into the Val type.

The rule for an abstraction (lambda $(x_1 \ldots x_n)$ e) injects into the Val type a closure which takes a list of values v and a continuation κ'. When the closure is invoked at function

$$[\![n]\!]_{scm} = \lambda\kappa\colon Val{\rightarrow}pr.\, in_N\, n\, \kappa$$

$$[\![x]\!]_{scm} = \lambda\kappa\colon Val{\rightarrow}pr.\, \mathsf{deref}\, x\, \kappa$$

$$\begin{array}{l}
[\![(\mathsf{lambda}\ (x_1 \ldots x_n)\ e)]\!]_{scm} = \lambda\kappa\colon Val{\rightarrow}pr. \\
\qquad in_F(\lambda v^*\colon Val^*.\lambda\kappa'\colon Val{\rightarrow}pr.(checkargs\ \ n\, v^*) \\
\qquad\quad \mathsf{new}^{Val}(listref_1\, v^*)\ (\lambda x_1{:}ref(Val).\cdots \\
\qquad\qquad \mathsf{new}^{Val}(listref_n\, v^*)\ (\lambda x_n{:}ref(Val).[\![e]\!]_{scm}\ \kappa')\ldots))\ \kappa
\end{array}$$

$$\begin{array}{l}
[\![(e_0\ e_1 \ldots e_n)]\!]_{scm} = \lambda\kappa\colon Val{\rightarrow}pr.[\![e_0]\!]_{scm}(\lambda f\colon Val. \\
\qquad [\![e_1]\!]_{scm}(\lambda v_1\colon Val.\cdots \\
\qquad\quad [\![e_n]\!]_{scm}(\lambda v_n\colon Val. \\
\qquad\qquad out_F\ f\ (\lambda f'\colon Val^*{\rightarrow}(Val{\rightarrow}pr){\rightarrow}pr. \\
\qquad\qquad\quad f'\ \langle v_1,\ldots,v_n\rangle\ \kappa))\ldots))
\end{array}$$

$$[\![(\mathsf{set!}\ x\ e)]\!]_{scm} = \lambda\kappa\colon Val{\rightarrow}pr.[\![e]\!]_{scm}(\lambda v\colon Val.\, \mathsf{update}\, x\ v\ (\kappa\ \Omega_{Val}))$$

$$\begin{array}{l}
[\![(\mathsf{letrec}\ ((x_1\ e_1)\ldots(x_n\ e_n))\ e)]\!]_{scm} = \lambda\kappa\colon Val{\rightarrow}pr. \\
\qquad \mathsf{new}\,\Omega_{Val}(\lambda x_1{:}ref(Val).\ldots \mathsf{new}\,\Omega_{Val}(\lambda x_n{:}ref(Val). \\
\qquad\quad [\![e_1]\!]_{scm}(\lambda v_1\colon Val.\, \mathsf{update}\, x_1 v_1 \cdots \\
\qquad\qquad [\![e_n]\!]_{scm}(\lambda v_n\colon Val.\, \mathsf{update}\, x_n v_n\ ([\![e]\!]_{scm}\kappa)\cdots))))
\end{array}$$

$$[\![(\mathsf{run}\, e)]\!]_{scm} = \langle\!\langle C_0[[\![e]\!]_{scm}]\ ;\ \emptyset\rangle\!\rangle$$

Figure 8: CPS translation from Scheme subset to metalanguage

application time, n new locations are allocated, each fresh location is initialized to an element of v and bound to the corresponding identifier x_i, and the body of the closure is run with κ' as its continuation. The arguments x_i correspond to locations in memory in order to allow for later assignments to x_i via expressions of the form $\mathsf{set!}\, x_i \ldots$. It is the elimination of this allocation overhead whenever possible which is the goal of the assignment elimination transformation.

The rule for application $(e_0\, e_2 \ldots e_n)$ first extracts the closure from the value of $[\![e_0]\!]_{scm}$ and then sends to that closure the list of values $\langle v_1, \ldots, v_n \rangle$ resulting from $[\![e_1]\!]_{scm}, \ldots, [\![e_n]\!]_{scm}$ as well as the current continuation, κ.

The rule for assignment, $(\mathsf{set!}\; x\; e)$, updates the location bound to x with the value of $[\![e]\!]_{scm}$. The Scheme semantics calls for the continuation to be invoked with an undefined value, for which we use Ω_{Val}, the canonical nonterminating term[1].

The rule for recursive definitions, $(\mathsf{letrec}\; ((x_1\; e_1) \ldots (x_n\; e_n))\; e)$ first allocates all required locations, initializing them to a dummy value, and then updates all of the locations with their contents within a scope which includes bindings from all mutually recursive identifiers $(x_1 \ldots x_n)$ to their newly allocated locations. This definition follows the official semantics, which initializes the variables to undefined values while building up the scope (var/loc pairs) and then destructively updates the variables to their correct values. Because the expressions $e_1, \ldots, e_n$ may have side effects, letrec cannot be implemented using the Y operator.

Evaluation of a program, $(\mathsf{run}\, e)$, runs the expression e with an empty store and in a context, C_0, which loads the values of Scheme primitives into the store, binds the primop identifiers to the fresh locations, and then runs e in an initial continuation, such as $(\lambda v\colon Val.print_{Val} v\; \mathsf{stop})$, for a suitable function $print_{Val}$. This definition is sufficiently general to handle primitives which do I/O or non-local jumps, such as call-cc.

The translation of any expression has type $(Val \to pr) \to pr$, given that all the free variables have type $ref(Val)$:

[1] for example, $(\mathsf{Y}^{Val}(\lambda x\colon Val.x))$

Lemma 4.3.1 *For a Scheme expression e and a type assignment Γ such that if $x \in fv(e)$ then $\Gamma(x) = ref(Val)$,*

$$\Gamma \vdash [\![e]\!]_{scm} : (Val \rightarrow pr) \rightarrow pr$$

PROOF:

Structural induction on e.

□

4.4 Equivalence of Scheme Expressions

For an optimization which replaces Scheme expression e_1 by e_2 to be correct in general, we must show that the two expressions behave the same in all Scheme program contexts. From the definition in Figure 8 of what it means to run a Scheme program, we get the following definition:

Definition 4.4.1 ($\cong^{scm}_{\forall C}$) *Scheme expressions e_1 and e_2 are operationally equivalent, denoted $e_1 \cong^{scm}_{\forall C} e_2$ iff for all Scheme contexts $C_{scm}[-]$, and for all initial contexts C_0.*

$$\langle\!\langle C_0[[\![C_{scm}[e_1]]\!]_{scm}] \ ; \emptyset \rangle\!\rangle \approx_{sim} \langle\!\langle C_0[[\![C_{scm}[e_2]]\!]_{scm}] \ ; \emptyset \rangle\!\rangle \qquad (1)$$

We will generally use the following relation to prove the equivalence of two Scheme expressions:

Definition 4.4.2 ($\cong_{scm}$) *For Scheme expressions e_1 and e_2, $e_1 \cong_{scm} e_2$ iff for all closing substitutions σ, all closed terms K of type $Val \rightarrow pr$, and all stores Σ,*

$$\langle\!\langle ([\![e_1]\!]_{scm}\sigma) \ K \ ; \Sigma \rangle\!\rangle \approx_{sim} \langle\!\langle ([\![e_2]\!]_{scm}\sigma) \ K \ ; \Sigma \rangle\!\rangle$$

It is a straightforward calculation to show that the two relations are equivalent:

Lemma 4.4.3 *For Scheme expressions e_1 and e_2,*

$$e_1 \cong_{\forall C}^{scm} e_2 \Leftrightarrow e_1 \cong_{scm} e_2$$

PROOF:

$$
\begin{aligned}
e_1 \cong_{\forall C}^{scm} e_2 \Leftrightarrow\ & \forall C_0, C_{scm}\ . \\
& \langle\!\langle C_0[\llbracket C_{scm}[e_1]\rrbracket_{scm}]\ ;\ \emptyset\rangle\!\rangle \approx_{sim} \langle\!\langle C_0[\llbracket C_{scm}[e_2]\rrbracket_{scm}]\ ;\ \emptyset\rangle\!\rangle \\
& \qquad \text{by Definition 4.4.1} \\[1ex]
\Leftrightarrow\ & \forall C[-_{(Val\to pr)\to pr}]{:}pr\ . \\
& \langle\!\langle C[\llbracket e_1\rrbracket_{scm}]\ ;\ \emptyset\rangle\!\rangle \approx_{sim} \langle\!\langle C[\llbracket e_2\rrbracket_{scm}]\ ;\ \emptyset\rangle\!\rangle \\
& \qquad \text{because } C_0 \text{ is unspecified} \\[1ex]
\Leftrightarrow\ & \forall C, \Sigma\ .\ \langle\!\langle C[\llbracket e_1\rrbracket_{scm}]\ ;\ \Sigma\rangle\!\rangle \approx_{sim} \langle\!\langle C[\llbracket e_2\rrbracket_{scm}]\ ;\ \Sigma\rangle\!\rangle \\
& \qquad \text{by Lemma 2.5.22} \\[1ex]
\Leftrightarrow\ & \forall \mathit{closing}\ \sigma, K{:}\,Val\to pr,\ \ \mathit{stores}\ \Sigma\ . \\
& \langle\!\langle (\llbracket e_1\rrbracket_{scm}\sigma)\ K\ ;\ \Sigma\rangle\!\rangle \approx_{sim} \langle\!\langle (\llbracket e_2\rrbracket_{scm}\sigma)\ K\ ;\ \Sigma\rangle\!\rangle \\
& \qquad \text{by Extensionality Theorem, Theorem 2.8.1} \\[1ex]
\Leftrightarrow\ & e_1 \cong_{scm} e_2 \\
& \qquad \text{by Definition 4.4.2.}
\end{aligned}
$$

□

4.5 Why Types are Needed in Target Metalanguage

It is instructive to try a translation from Scheme to an untyped version of our metalanguage and see what happens. Scheme is a dynamically-typed language, and the R^{n}RS denotational semantics is given in an untyped lambda calculus, so it would seem natural to use an untyped extended lambda calculus as the target metalanguage.

This is in fact what we did originally, and we were able to prove extensionality for an untyped variant of the metalanguage used in this thesis. We were disappointed, though, to find that some simple desired equivalences did not turn out to be equivalences using our untyped metalanguage.

Following is a simple example of an equivalence which should, but does not, hold after translation to an untyped metalanguage.

The two source expressions we wish to show equal are e_1 and e_2, defined below:

$e_1 \stackrel{def}{=} 42$
$e_2 \stackrel{def}{=} ((\mathsf{lambda}\ (x)\ 42)\ 0)$

The untyped metalanguage has integer constants, and the translation of an integer n is simply $(\lambda\kappa.\kappa\, n)$.

So the translation of e_1 is $(\lambda\kappa.\kappa\, 42)$. The translation of e_2 is β-equal to:

$$(\lambda\kappa.\,\mathsf{new}\,0\,(\lambda x.\kappa\,42))$$

In order to prove operational equivalence, we invoke extensionality and attempt to show both translations equivalent in all applicative contexts.

Consider the applicative context $[-]\,(\lambda v\kappa.\kappa\, v)\,(\lambda v.\,\mathsf{write}\,v\ \mathsf{stop})$:

The translation of e_1, inserted into the above context, proceeds as follows, in a state with store Σ:

$\langle\!\langle (\lambda\kappa.\kappa\,42)\,(\lambda v\kappa.\kappa\,v)\,(\lambda v.\,\mathsf{write}\,v\ \mathsf{stop})\ ;\ \Sigma\rangle\!\rangle$
$\rightarrow^*_{pcf}\ \langle\!\langle \mathsf{write}\,42\ \mathsf{stop}\ ;\ \Sigma\rangle\!\rangle$
$\stackrel{!42,\checkmark}{\rightarrow}_{io}\ \langle\!\langle \mathsf{stopped}\ ;\ \Sigma\rangle\!\rangle$

The translation of e_2 proceeds as follows, starting with store Σ:

$\langle\!\langle (\lambda\kappa.\,\mathsf{new}\,0\,(\lambda x.\kappa\,42))\,(\lambda v\kappa.\kappa\,v)\,(\lambda v.\,\mathsf{write}\,v\ \mathsf{stop})\ ;\ \Sigma\rangle\!\rangle$
$\rightarrow^*_{pcf}\ \langle\!\langle \mathsf{new}\,0\,(\lambda x.(\lambda\kappa.\kappa\,42))\,(\lambda v.\,\mathsf{write}\,v\ \mathsf{stop})\ ;\ \Sigma\rangle\!\rangle$

and there is no transition for states with a term component of the form

$$\mathsf{new}\,n\,M_1\,M_2$$

so the state "hangs".

Thus the two sides are not operationally equivalent. This comes about, in a sense, because the continuation terms are not of the right form for the translation we use. This problem is addressed by adding types to the metalanguage and insisting that all extensional contexts be well-typed.

When we use the typed version of the metalanguage, and the translation $[\![-]\!]_{scm}$ given in Figure 8, the translation of e_1 is:

$$[\![e_1]\!]_{scm} = \lambda\kappa{:}\,Val{\to}pr.\,in_N\; n\; \kappa$$

The translation of e_2 β-reduces to the following:

$$[\![e_2]\!]_{scm} \to^*_{pcf} \lambda\kappa{:}\,Val{\to}pr.\,\mathsf{new}\,0\;(\lambda x{:}ref\,(Val).\kappa\,42)$$

By Extensionality, Theorem 2.8.1, to show $[\![e_1]\!]_{scm} \cong_{\forall C} [\![e_2]\!]_{scm}$ we must show that for all closed metalanguage terms K of type $Val{\to}pr$ and for all stores Σ,

$$\langle\!\langle [\![e_1]\!]_{scm}\,K\;;\,\Sigma\rangle\!\rangle \lesssim_{sim} \langle\!\langle [\![e_2]\!]_{scm}\,K\;;\,\Sigma\rangle\!\rangle$$

Assuming that integers are put into the left summand of the Val type, the translation of e_1 reduces as follows:

$$\langle\!\langle [\![e_1]\!]_{scm}\,K\;;\,\Sigma\rangle\!\rangle \to^*_{pcf} \langle\!\langle K\;(\mathsf{inl}\,42)\;;\,\Sigma\rangle\!\rangle$$

The translation of e_2 does some (useless) store allocation and reduces as follows:

$$\langle\!\langle [\![e_2]\!]_{scm}\,K\;;\,\Sigma\rangle\!\rangle \to^*_{sto} \langle\!\langle K\;(\mathsf{inl}\,42)\;;\,\Sigma[l\mapsto 0]\rangle\!\rangle$$

with location l fresh w.r.t. $[\![e_2]\!]_{scm}$, K, and Σ.

The two states are easily shown to be in α_{loc} (with an empty partial bijection), and by Lemma 2.5.11 ($\alpha_{loc} \subseteq \lesssim_{sim}$), we get the desired equivalence.

Chapter 5

Assignment Elimination – A Language-to-Language Translation

In Scheme, all bound variables may be side-effected, and therefore the Scheme semantics [CR91] allocates storage for all bound variables. Many Scheme compilers, e.g. [KKR+86], perform a pass called *Assignment Elimination* in order to avoid unnecessary allocation of storage for variables which are never mutated. Assignment elimination identifies those variables which may be side-effected, explicitly allocates "cells" for those variables, and replaces all operations on those variables with cell operations. All variables deemed immutable are then bound to expressed values at run-time.

We need a language in which we can distinguish value-bound (immutable) and cell-bound (mutable) variables, and for that purpose we introduce the intermediate language Scheme_{cell}.

5.1 Intermediate Language Scheme_{cell}

The target language of assignment elimination is an intermediate language we call Scheme_{cell}, whose syntax is given in Figure 9. Scheme_{cell} has no set! operation but does have operations for cell creation, reference, and update. This language is similar to the untyped example

$$\begin{array}{rcl} e & ::= & x \mid n \mid (\textsf{lambda}\ (x \ldots)\ e) \mid (e\ e \ldots) \\ & \mid & (\textsf{let}\ ((x\ \mathit{rhs}) \ldots)\ e) \\ & \mid & (\textsf{letrec}\ ((x\ e) \ldots)\ e) \\ & \mid & (\textsf{cell-ref}\ x) \mid (\textsf{cell-set!}\ x\ e) \\ \mathit{rhs} & ::= & x \mid (\textsf{make-cell}\ e) \end{array}$$

Figure 9: Syntax of Scheme_{cell}

language featured in the work by Talcott et al. [HMST95], which has explicit cell operations. The storage model, where location-bound variables are explicit, is similar to that of ML.

There is only one syntactic class of variables, but the variables themselves will be bound to either values or cells. We impose a simple type-checking system to ensure that variables are used consistently – i.e., if a variable is cell-bound, then all references and updates to it are encased in cell-ref and cell-set! constructions. Figure 10 gives the typing rules for Scheme_{cell}. The rule for a lambda-expression shows that we treat lambda-bound variables as immutable values. The last two rules guarantee the consistency of the let-bound variables and their uses in the let-bodies. Note that the let of Figure 9 corresponds to let* in Scheme.

The semantics of Scheme_{cell} are given by a translation into our metalanguage. Figure 11 defines the translation $[\![-]\!]_{cell}$ from Scheme_{cell} to the metalanguage. Expressed values are still of type Val; identifiers denote either values of type Val or cells, which are of type $ref(Val)$.

The rule for variables no longer needs to dereference a location. The rule for abstraction no longer allocates storage for parameters. The rules for let state that binding a variable to another variable is simply renaming, whereas binding a variable to the result of a (make-cell e) operation does allocation at run-time, storing the result of $[\![e]\!]_{cell}$ in the fresh location. The rules for integers, application, and letrec are the same as in $[\![-]\!]_{scm}$.

We can be assured that if we apply the translation $[\![-]\!]_{cell}$ to well-typed Scheme_{cell} terms,

$$\Gamma \vdash_{cell} n{:}ok \qquad \frac{\Gamma(x) = ref(Val) \quad \Gamma \vdash_{cell} e{:}ok}{\Gamma \vdash_{cell} (\textsf{cell-set!}\ x\ e){:}ok}$$

$$\frac{\Gamma[x_i{:}\,Val]_{i=1}^{n} \vdash_{cell} e{:}ok}{\Gamma \vdash_{cell} (\textsf{lambda}\ (x_1 \ldots x_n)\ e){:}ok}$$

$$\frac{\Gamma \vdash_{cell} e_1{:}ok \ \ldots\ \Gamma \vdash_{cell} e_n{:}ok}{\Gamma \vdash_{cell} (e_1 \ldots e_n){:}ok}$$

$$\frac{\Gamma(x) = ref(Val)}{\Gamma \vdash_{cell} (\textsf{cell-ref}\ x){:}ok} \qquad \frac{\Gamma(x) = Val}{\Gamma \vdash_{cell} x{:}ok}$$

$$\frac{\Gamma \vdash_{cell} e{:}ok}{\Gamma \vdash_{cell} (\textsf{let}\ ()\ e){:}ok}$$

$$\frac{\Gamma(x) = Val \quad \Gamma[x_1{:}\,Val] \vdash_{cell} (\textsf{let}\ ((x_2\ rhs_2) \ldots (x_n\ rhs_n))\ e){:}ok}{\Gamma \vdash_{cell} (\textsf{let}\ ((x_1\ x)\ (x_2\ rhs_2) \ldots (x_n\ rhs_n))\ e){:}ok}$$

$$\frac{\Gamma \vdash_{cell} e_1{:}ok \quad \Gamma[x_1{:}ref(Val)] \vdash_{cell} (\textsf{let}\ ((x_2\ rhs_2) \ldots (x_n\ rhs_n))\ e){:}ok}{\Gamma \vdash_{cell} (\textsf{let}\ ((x_1\ (\textsf{make-cell}\ e_1))\ (x_2\ rhs_2) \ldots (x_n\ rhs_n))\ e){:}ok}$$

Figure 10: Type Checking for Scheme$_{cell}$

$$[\![x]\!]_{cell} = \lambda\kappa{:}\,Val{\rightarrow}pr.\kappa\,x$$

$$\begin{aligned}[\![(\mathsf{lambda}\ (x_1 \ldots x_n)\ e)]\!]_{cell} = {} & \lambda\kappa{:}\,Val{\rightarrow}pr. \\ & in_F(\lambda v^*{:}\,Val^*.\lambda\kappa'{:}\,Val{\rightarrow}pr.(checkargs\ n\,v^*) \\ & \quad [\![e]\!]_{cell}[(listref_1\,v^*)/x_1,\ldots,(listref_n\,v^*)/x_n]\ \kappa')\ \kappa\end{aligned}$$

$$\begin{aligned}[\![(\mathsf{let}\ ((x_1\ x)\ (x_2\ rhs_2)\ldots(x_n\ rhs_n))\ e)]\!]_{cell} = {} & \\ & [\![(\mathsf{let}\ ((x_2\ rhs_2)\ldots(x_n\ rhs_n))\ e)]\!]_{cell}[x/x_1] \\ [\![(\mathsf{let}\ ((x_1\ (\mathsf{make\text{-}cell}\,e_1))\ (x_2\ rhs_2)\ldots(x_n\ rhs_n))\ e)]\!]_{cell} = {} & \\ & [\![e_1]\!]_{cell}(\lambda v_1{:}\,Val.\,\mathsf{new}\,v_1\ (\lambda x_1{:}ref(Val). \\ & \quad [\![(\mathsf{let}\ ((x_2\ rhs_2)\ldots(x_n\ rhs_n))\ e)]\!]_{cell})) \\ [\![(\mathsf{let}\ ()\ e)]\!]_{cell} = {} & [\![e]\!]_{cell}\end{aligned}$$

$$[\![(\mathsf{cell\text{-}ref}\ x)]\!]_{cell} = \lambda\kappa{:}\,Val{\rightarrow}pr.\,\mathsf{deref}\,x\ \kappa$$

$$[\![(\mathsf{cell\text{-}set!}\ x\ e)]\!]_{cell} = \lambda\kappa{:}\,Val{\rightarrow}pr.[\![e]\!]_{cell}(\lambda v{:}\,Val.\,\mathsf{update}\,x\ v\ (\kappa\,\Omega)))$$

Figure 11: CPS translation from Scheme$_{cell}$ to metalanguage

the resultant metalanguage terms will be well-typed:

Lemma 5.1.1 *For a Scheme$_{cell}$ term e, and a type assumption Γ mapping variables to $\{Val, ref(Val)\}$,*

$$\Gamma \vdash_{cell} e{:}ok \Rightarrow \Gamma \vdash [\![e]\!]_{cell}{:}(Val{\rightarrow}pr){\rightarrow}pr$$

PROOF:

Structural induction on $\vdash_{cell}$.

□

It should be noted that it is straightforward to write implementations of make-cell, cell-ref, and cell-set! in the original Scheme dialect which are operationally equivalent in the original semantics to the same operators in Scheme$_{cell}$ under the second translation. By supplying such function definitions, the Twobit Scheme compiler is able to produce runnable Scheme code as it transforms the original Scheme code into the new Scheme$_{cell}$ language. This trick is useful for debugging, among other things, but is not relevant to our correctness proofs.

5.2 The Assignment Elimination Transformation

Figure 12 defines the assignment-elimination transformation. The transformation $AE(e)$ takes a Scheme expression e and produces a Scheme$_{cell}$ expression. As $AE(e)$ transforms expression e, a set V is maintained of the variables which will correspond to {V}alue-bound (i.e. never side-effected) values in the output Scheme$_{cell}$ term. So $AE(V, e)$ is the translation of Scheme expression e assuming that the variables in V will be treated as immutable values at run-time and do not need to be stored in cells. The translation of a top-level expression assumes no immutable variables have yet been discovered, so $AE(e) = AE(\emptyset, e)$. An invariant of the transformation is that for every $AE(V, e')$ encountered during a transformation $AE(e)$ of a Scheme expression e, $V \subseteq fv(e')$.

The translation of a value-bound (immutable) variable x (i.e., $x \in V$) is simply x. However, if x is not in V, x is assumed mutable and thus a reference to x must be translated to (cell-ref x).

An update of a variable x is translated to (cell-set! $x \ldots$). Because membership in V by x is based on the absence of set! x subexpressions, and because we start the AE transformation with an empty V, we expect that if the rule for $AE(V, (\text{set! } x\ e))$ applies, then $x \notin V$.

The rule for a lambda expression first determines which of its bound variables are updated in its body, using the sv function. For each mutable local variable x_i returned by sv, AE emits a let (x_i (make-cell x_i)) binding. Immutable local variables (i.e. those not returned by sv) are added to the V used to transform the body of the lambda, so that references to those variables will not be surrounded by cell operations.

Finally, the rule for letrec treats all letrec-bound variables as mutable.

Note that in the simple case that $e =$ (lambda (x) e'), $V \supseteq fv(e)$, and there are no set!'s or letrec's in e, then $AE(V, e)$ is the identity transformation (up to some let ((x x)) bindings). If, however, e contains some updates to x, the transformation of e will bind x to (make-cell x) and all references and updates of x will be enclosed in (cell-ref x) and (cell-set! x) constructions. More formally, we can prove that for any Scheme expression e, $AE(e)$ produces a

Definition 5.2.1 (Assignment Elimination (AE)) *Let $sv(e)$ be the set of free variables of e that appear in e as the left-hand side of a* set! *expression. Define $AE(e)$ to be $AE(\emptyset, e)$, where $AE(V, e)$ is defined as follows. Here V is the set of free variables that are to be bound to values rather than locations.*

$$AE(V, x) = \begin{cases} x & \text{if } x \in V \\ (\textsf{cell-ref } x) & \text{if } x \notin V \end{cases}$$

$$AE(V, (\textsf{set! } x\ e)) = (\textsf{cell-set! } x\ AE(V, e))$$

$$AE(V, (e_0\ e_1\ \ldots\ e_n)) = (AE(V, e_0)\ \ AE(V, e_1)\ \ldots\ AE(V, e_n))$$

$$AE(V, (\textsf{lambda } (x_1\ \ldots\ x_n)\ e))$$
$$= (\textsf{lambda } (x_1\ \ldots\ x_n)\ (\textsf{let } ((x_1\ r_1)\ \ldots\ (x_n\ r_n))\ AE(V', e)))$$
$$\text{where } \begin{cases} r_i = \begin{cases} (\textsf{make-cell } x_i) & \text{if } x_i \in sv(e) \\ x_i & \text{if } x_i \notin sv(e) \end{cases} \\ V' = V \cup \{x_i \mid x_i \notin sv(e)\} - \{x_i \mid x_i \in sv(e)\} \end{cases}$$

$$AE(V, (\textsf{letrec } ((x_1\ e_1) \ldots (x_n\ e_n))\ e))$$
$$= (\textsf{letrec } ((x_1\ \ AE(V', e_1)) \ldots (x_n\ \ AE(V', e_n)))$$
$$AE(V', e)) \qquad \text{where } V' = V - \{x_1, \ldots, x_n\}$$

Figure 12: Assignment Elimination

well-typed Scheme$_{cell}$ expression:

Lemma 5.2.2 *For any Scheme expression e, and for a type assignment Γ such that if $x \in fv(e)$ then $\Gamma(x) = ref(Val)$,*

$$\Gamma \vdash_{cell} AE(e){:}ok$$

PROOF:

Structural induction on e.

□

For any given Scheme program, there is some number, say n, of lambda-expressions:

(lambda $(x_{1,1}, \ldots, x_{1,m_1})$ e_1)

$\ldots$

(lambda $(x_{n,1}, \ldots, x_{n,m_n})$ e_n)

In the following sections, we use the following notation to refer to the relevant parts of each lambda-expression:

$$\begin{array}{lcll} X_i & = & \langle x_{i,1}, \ldots, x_{i,m_i} \rangle & \text{(sequence of parameters)} \\ C_i & = & sv(e_i) \cap X_i & \text{(set of \{C\}ell-bound local variables)} \\ V_i & = & X_i - C_i & \text{(set of \{V\}alue-bound local variables)} \end{array}$$

For the length of a sequence X we write $\#X$. For the j^{th} element of a sequence X we write $X_{(j)}$; for a sequence such as X_i, we write $X_{i(j)}$.

5.3 Correctness of Assignment Elimination

We prove the assignment elimination transformation is meaning-preserving. That is, for any Scheme expression e,

$$[\![e]\!]_{scm} \cong_{\forall C} [\![AE(e)]\!]_{cell}$$

Our goal is to build a candidate simulation relation between metalanguage states which characterizes the syntactic differences between (non-transformed) Scheme programs and those programs after assignment elimination, as they reduce under $\stackrel{\alpha}{\rightarrow}_{io}$. In the ensuing analysis, we find that there are several different ways in which corresponding subterms can differ:

1. The translations of Scheme lambda-expressions will differ in the bodies of their closures. The original closure body will contain new operators for all local parameters, whereas the transformed body will only allocate storage for mutable local parameters. In the original, references to local variables always involve store lookup with the deref operator, whereas after transformation, references to immutable variables will not do store lookup, as the variable will get its value via substitution.

2. When subterms corresponding to lambda-expressions are applied to a run-time continuation, they are injected into the F summand of the Val type and sent to the continuation.

3. When the closures are used at run-time (i.e., applied to arguments), they are extracted from the Val type and applied to their application-time argument list and continuation. At this point, the differences between the closure bodies are "exposed", and we see different storage behavior.

4. As the exposed bodies of corresponding closures reduce, the original term allocates storage for all local variables, whereas the transformed term only allocates storage for mutable local variables. For immutable local variables, in the original term we have a substitution $[l_i/x_i]$ for each immutable variable x_i (and mutable too, for that matter). In the transformed term, for the immutable variables x_i, we see value substitutions $[(\mathit{listref}_i\, v^*)/x_i]$, where the variable v^* contains the list of incoming arguments.

Figure 13 presents specifications for, and definitions of, macros to capture the syntactic differences described in items 1-3 above. The macros $\mathcal{L}_1$ and $\mathcal{R}_1$ express the initial differences between the translations of a lambda-expression and of its transformed version.

Initial Translation:

$$
\begin{array}{l}
[\![(\mathsf{lambda}\ (x_1 \ldots x_m)\ e)]\!]_{scm} = \\
\quad \mathcal{L}_1\langle x_1 \ldots x_m\rangle[\![\,[\![e]\!]_{scm}\,]\!]
\end{array}
$$

$$
\begin{array}{l}
\mathcal{L}_1\langle x_1 \ldots x_m\rangle[\![P]\!] \stackrel{def}{=} \\
\quad (\lambda\kappa{:}\,Val{\to}pr.\ in_F(\lambda v^*{:}\,Val^*. \\
\qquad \lambda\kappa'{:}\,Val{\to}pr.(checkargs\ \ m\,v^*) \\
\qquad\quad \mathsf{new}^{Val}(listref_1\,v^*)\ (\lambda x_1{:}ref(Val). \\
\qquad\qquad \ldots \\
\qquad\quad \mathsf{new}^{Val}(listref_m\,v^*)\ (\lambda x_m{:}ref(Val). \\
\qquad\qquad (P\ \kappa'))\ldots))\ \kappa)
\end{array}
$$

$$
\begin{array}{l}
[\![AE(V, (\mathsf{lambda}\ (x_1 \ldots x_m)\ e))]\!]_{cell} = \\
\quad \mathcal{R}_1(V', C)[\![\,[\![AE(V', e)]\!]_{cell}\,]\!],\ \text{with} \\
\quad V' = V \cup \{x_i \mid x_i \notin sv(e)\} - \\
\qquad\qquad \{x_i \mid x_i \in sv(e)\} \\
\quad C = \{x_i \mid x_i \in sv(e)\}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{R}_1(V, C)[\![P]\!] \stackrel{def}{=} \\
\quad (\lambda\kappa{:}\,Val{\to}pr.\ in_F(\lambda v^*{:}\,Val^*. \\
\qquad \lambda\kappa'{:}\,Val{\to}pr.(checkargs\ \ m\,v^*) \\
\qquad\quad \left.\begin{array}{l}\mathsf{new}^{Val}(listref_i\,v^*) \\ \quad(\lambda x_i{:}ref(Val).\end{array}\right\}\begin{array}{l}\text{for each} \\ x_i \in C\end{array} \\
\qquad\qquad ((P\,[(listref_i\,v^*)/x_i]_{x_i \in V})\ \kappa')\ldots))\ \kappa)
\end{array}
$$

Applied to a continuation K:

$$
\begin{array}{l}
\langle\!\langle(\mathcal{L}_1\langle x_1 \ldots x_m\rangle[\![P]\!]\ K)\ ;\ \Sigma\rangle\!\rangle \to^*_{pcf} \\
\quad \langle\!\langle(K\ \mathcal{L}_2\langle x_1 \ldots x_m\rangle[\![P]\!])\ ;\ \Sigma\rangle\!\rangle \\
\mathcal{L}_2\langle x_1 \ldots x_m\rangle[\![P]\!] \stackrel{def}{=} \\
\quad \mathsf{inr}(\lambda v^*{:}\,Val^*.\lambda\kappa'{:}\,Val{\to}pr.(checkargs\ \ m\,v^*) \\
\qquad \mathsf{new}^{Val}(listref_1\,v^*)\ (\lambda x_1{:}ref(Val). \\
\qquad\quad \ldots \\
\qquad \mathsf{new}^{Val}(listref_m\,v^*)\ (\lambda x_m{:}ref(Val). \\
\qquad\quad (P\ \kappa')\ldots)))
\end{array}
$$

$$
\begin{array}{l}
\langle\!\langle(\mathcal{R}_1(V, C)[\![P]\!]\ K)\ ;\ \Sigma\rangle\!\rangle \to^*_{pcf} \\
\quad \langle\!\langle(K\ \mathcal{R}_2(V, C)[\![P]\!])\ ;\ \Sigma\rangle\!\rangle \\
\mathcal{R}_2(V, C)[\![P]\!] \stackrel{def}{=} \\
\qquad \mathsf{inr}(\lambda v^*{:}\,Val^*.\lambda\kappa'{:}\,Val{\to}pr.(checkargs\ \ m\,v^*) \\
\qquad\quad \left.\begin{array}{l}\mathsf{new}^{Val}(listref_i\,v^*) \\ \quad(\lambda x_i{:}ref(Val).\end{array}\right\}\begin{array}{l}\text{for each} \\ x_i \in C\end{array} \\
\qquad\qquad ((P\,[(listref_i\,v^*)/x_i]_{x_i \in V})\ \kappa')\ldots))
\end{array}
$$

Closure Applied to an Argument List Q:

$$
\begin{array}{l}
\langle\!\langle\ out_F(\mathcal{L}_2\langle x_1 \ldots x_m\rangle[\![P]\!]) \\
\quad (\lambda f{:}\,Val^*{\to}(Val{\to}pr){\to}pr. \\
\qquad f\ Q\ K)\ ;\ \Sigma\rangle\!\rangle \\
\quad \text{where } Q{:}\,Val^* \text{ and } K{:}\,Val{\to}pr \\
\quad \to^*_{pcf}\ \langle\!\langle\mathcal{L}_3\langle x_1 \ldots x_m\rangle[\![P,\,Q,\,K]\!]\ ;\ \Sigma\rangle\!\rangle \\
\mathcal{L}_3\langle x_1 \ldots x_m\rangle[\![P_1,\,P_2,\,P_3]\!] \stackrel{def}{=} \\
\quad \mathsf{new}^{Val}(listref_1\,P_2)\ (\lambda x_1{:}ref(Val). \\
\qquad \ldots \\
\qquad \mathsf{new}^{Val}(listref_m\,P_2)\ (\lambda x_m{:}ref(Val). \\
\qquad\quad (P_1\ P_3)\ldots)))
\end{array}
$$

$$
\begin{array}{l}
\langle\!\langle\ out_F(\mathcal{R}_2(V, C)[\![P]\!]) \\
\quad (\lambda f{:}\,Val^*{\to}(Val{\to}pr){\to}pr. \\
\qquad f\ Q\ K)\ ;\ \Sigma\rangle\!\rangle \\
\quad \text{where } Q{:}\,Val^* \text{ and } K{:}\,Val{\to}pr \\
\quad \to^*_{pcf}\ \langle\!\langle\mathcal{R}_3(V, C)[\![P,\,Q,\,K]\!]\ ;\ \Sigma\rangle\!\rangle \\
\mathcal{R}_3(V, C)[\![P_1,\,P_2,\,P_3]\!] \stackrel{def}{=} \\
\qquad \left.\begin{array}{l}\mathsf{new}^{Val}(listref_i\,P_2) \\ \quad(\lambda x_i{:}ref(Val).\end{array}\right\}\begin{array}{l}\text{for each} \\ x_i \in C\end{array} \\
\qquad\quad ((P_1\,[(listref_i\,P_2)/x_i]_{x_i \in V})\ P_3)\ldots)
\end{array}
$$

Figure 13: Definitions of $\mathcal{L}_i$, $\mathcal{R}_i$ macros

The macros $\mathcal{L}_2$ and $\mathcal{R}_2$ express the syntactic forms of the original and transformed terms after having been applied to a continuation term K. The definitions of $\mathcal{L}_2$ and $\mathcal{R}_2$ assume that closures go into the right summand of the *Val* type.

Macros $\mathcal{L}_3$ and $\mathcal{R}_3$ describe the two versions of the lambda-expression after having been applied to an argument list Q of type Val^* and a continuation K of type $Val \rightarrow pr$. For two states where the head redexes of the term components of both states correspond to $\mathcal{L}_3$ and $\mathcal{R}_3$ terms, we can see that the left-hand side will allocate new locations for each local parameter $x_1, \ldots, x_m$. The right-hand side, however, will allocate new storage only for the mutable local variables (listed in C) and will use substitution for the immutable variables (listed in V). Thus, at run-time, the storage behavior of the two sides will differ, with the right-hand side doing potentially less memory allocation than the left-hand side.

All of the macro definitions, $\mathcal{L}_i, \mathcal{R}_i$ $(i = 1, 2, 3)$, are parameterized by the metalanguage terms for the closure bodies. The terms for closure bodies will differ at subterms that correspond to references to immutable parameters. In the original term, all references correspond to a store lookup, whereas in the transformed term, references to immutable variables are translated as variables that are substituted-for at closure application time.

The relation $\overset{AE}{\leadsto}_V$, defined in Figure 14, between metalanguage terms formalizes the differences between translations of immutable variable references. The subscript V is a list of variables that are treated as values in the transformed term. The Γ in the judgement $\Gamma \vdash M \overset{AE}{\leadsto}_V N{:}\tau$ corresponds to the type environment needed to prove that M has type τ. Two metalanguage terms related by $\overset{AE}{\leadsto}_V$ will differ only in their treatment of non-mutated bound variables, specified by the set V, and if every assignment in Γ of $x \in V$ to $ref(Val)$ is replaced by an assignment $x{:}\,Val$, the term N will have type τ as well. In other words:

$\overset{AE}{\leadsto}_V$ is the least relation between metalanguage terms closed under the following rules

$$\Gamma \vdash n \overset{AE}{\leadsto}_V n{:}int \quad (\textsc{cong-int})$$

$$\Gamma \vdash c \overset{AE}{\leadsto}_V c{:}\tau \qquad (\tau \text{ from Figure 1}) \quad (\textsc{cong-const})$$

$$\Gamma \vdash l^\tau \overset{AE}{\leadsto}_V l^\tau{:}ref(\tau) \quad (\textsc{cong-loc})$$

$$\frac{\Gamma[x{:}\tau'] \vdash M \overset{AE}{\leadsto}_V N{:}\tau \quad x \notin V}{\Gamma \vdash (\lambda x{:}\tau'.M) \overset{AE}{\leadsto}_V (\lambda x{:}\tau'.N){:}\tau' \to \tau} \quad (\textsc{cong-abs})$$

$$\frac{\Gamma \vdash M \overset{AE}{\leadsto}_V M'{:}\tau' \to \tau \quad \Gamma \vdash N \overset{AE}{\leadsto}_V N'{:}\tau'}{\Gamma \vdash (M\ N) \overset{AE}{\leadsto}_V (M'\ N'){:}\tau} \quad (\textsc{cong-app})$$

$$\frac{x \notin V \quad \Gamma(x) = \tau}{\Gamma \vdash x \overset{AE}{\leadsto}_V x{:}\tau} \quad (\textsc{mutable-var})$$

$$\frac{\Gamma \vdash M \overset{AE}{\leadsto}_V N{:}\,Val \to pr \quad x \in V \quad \Gamma(x) = ref(Val)}{\Gamma \vdash (\mathsf{deref}\, x\ M) \overset{AE}{\leadsto}_V (N\ x){:}pr} \quad (\textsc{immutable-var})$$

Figure 14: Definition of $\overset{AE}{\leadsto}_V$

Lemma 5.3.1 *If* $V = \{x_1, \ldots, x_n\}$ *and* $\Gamma[x_i{:}\mathit{ref}(\mathit{Val})]_{i=1}^n \vdash M{:}\tau$*, then*

$$\Gamma[x_i{:}\mathit{ref}(\mathit{Val})]_{i=1}^n \vdash M \overset{AE}{\rightsquigarrow}_V N{:}\tau \Rightarrow \Gamma[x_i{:}\,\mathit{Val}]_{i=1}^n \vdash N{:}\tau$$

PROOF:

By induction on the structure of M.

□

The notation $\Gamma[x_i{:}\tau]_{i=1}^n$ is shorthand for the type environment $\Gamma[x_1{:}\tau, \ldots, x_n{:}\tau]$.

The relation $\overset{AE}{\rightsquigarrow}_V$ between terms is preserved by substitution by $(\overset{AE}{\rightsquigarrow}_V)$-related terms for variables not in V:

Lemma 5.3.2 *If* $\Gamma[x{:}\tau'] \vdash M \overset{AE}{\rightsquigarrow}_V N{:}\tau$ *and* $\Gamma \vdash P \overset{AE}{\rightsquigarrow}_V P'{:}\tau'$ *and* $x \notin V$*, then*

$$\Gamma \vdash M[P/x] \overset{AE}{\rightsquigarrow}_V N[P'/x]{:}\tau$$

PROOF:

ASSUME: $\Gamma \vdash P \overset{AE}{\rightsquigarrow}_V P'{:}\tau'$

Proceed by induction on the structure of the proof of

$$\Gamma \vdash M[P/x] \overset{AE}{\rightsquigarrow}_V N[P'/x]{:}\tau.$$

⟨1⟩1. CASE: IMMUTABLE-VAR

ASSUME: 1. $M \equiv \mathsf{deref}\, y\; M'$

2. $N \equiv N'\; y$

3. $y \in V$

4. $\Gamma[x{:}\tau'] \vdash M' \overset{AE}{\rightsquigarrow}_V N'{:}\,\mathit{Val} \to \mathit{pr}$

5. $\Gamma(y) = \mathit{ref}(\mathit{Val})$

⟨2⟩1. $\Gamma[x{:}\tau'] \vdash M'[P/x] \overset{AE}{\rightsquigarrow}_V N'[P'/x]{:}\,\mathit{Val} \to \mathit{pr}$

By IH.

⟨2⟩2. $\Gamma \vdash M[P/x] \overset{AE}{\rightsquigarrow}_V N[P'/x]{:}\mathit{pr}$

By IMMUTABLE-VAR.

⟨1⟩2. CASE: (CONG-APP)

ASSUME: 1. $M \equiv M'\, M''$

2. $N \equiv N'\, N''$

⟨2⟩1. $\Gamma \vdash M'[P/x] \overset{AE}{\leadsto}_V N'[P'/x]{:}\alpha{\to}\beta$

By IH.

⟨2⟩2. $\Gamma \vdash M''[P/x] \overset{AE}{\leadsto}_V N''[P'/x]{:}\alpha$

By IH.

⟨2⟩3. $\Gamma \vdash M[P/x] \overset{AE}{\leadsto}_V N[P'/x]{:}\beta$

By (CONG-APP).

⟨1⟩3. All other cases are similarly straightforward.

□

Consider a pair of states in which the term component of the left-hand side is $\mathcal{L}_3(X)[\![P_1,\, P_2,\, P_3]\!]$, and on the right-hand side we have $\mathcal{R}_3(V', C)[\![P_1',\, P_2',\, P_3']\!]$. Further assume that the corresponding "closure body" terms are related by $\overset{AE}{\leadsto}_V$; that is, $\Gamma \vdash P_i \overset{AE}{\leadsto}_V P_i'{:}pr$ for $1 \leq i \leq 3$, where $V' \subseteq V$ and $C \cap X = \emptyset$. As the states reduce, both sides will do allocation – the left-hand side will allocate storage for all of its local variables, and the right-hand side will only allocate storage for its mutable local variables. The term components start in $\overset{AE}{\leadsto}_V$ but do not stay in $\overset{AE}{\leadsto}_V$. On the left-hand side, the immutable variables x_i will be replaced by location constants, whereas on the right-hand side, the immutable variables will be replaced by actual values. Thus, where we started with $\overset{AE}{\leadsto}_V$-related terms such as $\Gamma \vdash (\mathsf{deref}\, x\ M) \overset{AE}{\leadsto}_V (N\ x){:}pr$, we now have terms such as $(\mathsf{deref}\, l_x\ M) \overset{AE}{\not\leadsto}_V (N\ Q_x)$, where in the store component of the left-hand side, at location l_x, we expect a term related to Q_x.

The relation $\overset{AE}{\leadsto}_{V|Z}$ between metalanguage terms, defined in Figure 15, characterizes the differences between terms which were originally related by $\overset{AE}{\leadsto}_V$ but which have been subjected to run-time substitutions – of locations for variables on the left-hand side and of values for variables on the right-hand side.

Definition 5.3.3 ($\overset{AE}{\leadsto}_{V|Z}$ for metalanguage terms)
For $V = \langle x_1, \ldots, x_n \rangle$, $Z = \langle (l_1, P_1), \ldots, (l_k, P_k) \rangle$,

$$
\begin{aligned}
&\Gamma \vdash M \overset{AE}{\leadsto}_{V|Z} N{:}\tau \Leftrightarrow \\
&\qquad \wedge\ k \leq n \\
&\qquad \wedge\ \Gamma(x_i) = \mathit{ref}(\mathit{Val}), \quad \mathit{for}\ 1 \leq i \leq n \\
&\qquad \wedge\ P_i{:}\mathit{Val},\ \mathit{for}\ 1 \leq i \leq k \\
&\qquad \wedge\ \exists M',\ N'\ \text{s.t.} \\
&\qquad\qquad \wedge\ M = M'[l_1/x_1, \ldots, l_k/x_k] \\
&\qquad\qquad \wedge\ N = N'[P_1/x_1, \ldots, P_k/x_k] \\
&\qquad\qquad \wedge\ \Gamma \vdash M' \overset{AE}{\leadsto}_{V} N'{:}\tau
\end{aligned}
$$

Definition 5.3.4 ($\overset{AE}{\leadsto}_{V|Z}$ for stores)

$$
\begin{aligned}
&\Gamma \vdash \Sigma'_M \overset{AE}{\leadsto}_{V|Z} \Sigma_N \Leftrightarrow \\
&\qquad \wedge\ \mathit{dom}(\Sigma'_M) = \mathit{dom}(\Sigma_N) \\
&\qquad \wedge\ \forall \tau, l^\tau \in \mathit{dom}(\Sigma'_M)\ .\ \Gamma \vdash \Sigma'_M(l) \overset{AE}{\leadsto}_{V|Z} \Sigma_N(l){:}\tau
\end{aligned}
$$

Figure 15: The Relations $\overset{AE}{\leadsto}_{V|Z}$ between Terms and Stores

5.3.1 The Assignment Elimination Simulation Relation

Figure 16 defines a candidate simulation relation, S_{AE}, by abstracting over corresponding pairs of terms which differ as outlined earlier. Each of the distinct ways in which corresponding metalanguage subterms might differ is handled by a substitution pair $(\sigma^i_{\mathcal{L}}, \sigma^i_{\mathcal{R}})$. The substitutions $\sigma^i_{\mathcal{L}}$ and $\sigma^i_{\mathcal{R}}$ have domains $\{u^i_1, \ldots, u^i_{\#\sigma^i}\}$, for $1 \leq i \leq 3$. In $\sigma^1_{\mathcal{L}}$, each u^1_i maps to a metalanguage term of the form $\mathcal{L}_1(X^1_i)[\![M^1_i]\!]$, whereas in $\sigma^1_{\mathcal{R}}$, each u^1_i maps to a metalanguage term of the form $\mathcal{R}_1(V^1_i, C^1_i)[\![N^1_i]\!]$. The terms corresponding to the closure bodies, M^1_i and N^1_i, must be related by $\overset{AE}{\leadsto}_{V|Z}$.

The substitution pairs $(\sigma^2_{\mathcal{L}}, \sigma^2_{\mathcal{R}})$ and $(\sigma^3_{\mathcal{L}}, \sigma^3_{\mathcal{R}})$ are similar, but abstract over pairs of terms of the forms $(\mathcal{L}_2, \mathcal{R}_2), (\mathcal{L}_3, \mathcal{R}_3)$.

The last section of the definition of S_{AE} requires all of the closure bodies to be $\overset{AE}{\leadsto}_{V|Z}$-related. In order to simplify the notation, we assume that all of the bound variables in the states we consider are distinct. This allows us to have a single $\overset{AE}{\leadsto}_{V|Z}$ relation for the whole

$$
\begin{array}{l}
S_{AE} \stackrel{def}{=} \{(\langle\!\langle M_0 \ ; \Sigma_M \rangle\!\rangle \sigma_{\mathcal{L}}, \langle\!\langle N_0 \ ; \Sigma_N \rangle\!\rangle \sigma_{\mathcal{R}}) \mid \ \exists \Gamma, \Sigma'_M, V, Z \text{ s.t.} \\
\qquad \sigma_{\mathcal{L}} = \sigma^1_{\mathcal{L}} \cup \sigma^2_{\mathcal{L}} \cup \sigma^3_{\mathcal{L}} \\
\qquad \sigma_{\mathcal{R}} = \sigma^1_{\mathcal{R}} \cup \sigma^2_{\mathcal{R}} \cup \sigma^3_{\mathcal{R}} \\
\qquad fv(M_0, N_0, rng(\Sigma_M), rng(\Sigma_N)) \subseteq \{u^1_1, \ldots, u^1_{n_1}, u^2_1, \ldots, u^2_{n_2}, u^3_1, \ldots, u^3_{n_3}\}
\end{array}
$$

$$
\begin{array}{c}(\mathcal{L}_1, \mathcal{R}_1) \\ \text{pairs}\end{array}
\left\{
\begin{array}{l}
\sigma^1_{\mathcal{L}} = [u^1_1 \mapsto \mathcal{L}_1(X^1_1)[\![M^1_1]\!], \ldots, u^1_{n_1} \mapsto \mathcal{L}_1(X^1_{n_1})[\![M^1_{n_1}]\!]] \\
\sigma^1_{\mathcal{R}} = [u^1_1 \mapsto \mathcal{R}_1(V^1_1, C^1_1)[\![N^1_1]\!], \ldots, u^1_{n_1} \mapsto \mathcal{R}_1(V^1_{n_1}, C^1_{n_1})[\![N^1_{n_1}]\!]] \\
n_1 = \#\sigma^1_{\mathcal{L}} = \#\sigma^1_{\mathcal{R}}
\end{array}
\right.
$$

$$
\begin{array}{c}(\mathcal{L}_2, \mathcal{R}_2) \\ \text{pairs}\end{array}
\left\{
\begin{array}{l}
\sigma^2_{\mathcal{L}} = [u^2_1 \mapsto \mathcal{L}_2(X^2_1)[\![M^2_1]\!], \ldots, u^2_{n_2} \mapsto \mathcal{L}_2(X^2_{n_2})[\![M^2_{n_2}]\!]] \\
\sigma^2_{\mathcal{R}} = [u^2_1 \mapsto \mathcal{R}_2(V^2_1, C^2_1)[\![N^2_1]\!], \ldots, u^2_{n_2} \mapsto \mathcal{R}_2(V^2_{n_2}, C^2_{n_2})[\![N^2_{n_2}]\!]] \\
n_2 = \#\sigma^2_{\mathcal{L}} = \#\sigma^2_{\mathcal{R}}
\end{array}
\right.
$$

$$
\begin{array}{c}(\mathcal{L}_3, \mathcal{R}_3) \\ \text{pairs}\end{array}
\left\{
\begin{array}{l}
\sigma^3_{\mathcal{L}} = [u^3_1 \mapsto \mathcal{L}_3(X^3_1)[\![M^3_{1,1},\, M^3_{1,2},\, M^3_{1,3}]\!], \ldots, \\
\qquad\quad u^3_{n_3} \mapsto \mathcal{L}_3(X^3_{n_3})[\![M^3_{n_3,1},\, M^3_{n_3,2},\, M^3_{n_3,3}]\!]] \\
\sigma^3_{\mathcal{R}} = [u^3_1 \mapsto \mathcal{R}_3(V^3_1, C^3_1)[\![N^3_{1,1}\ N^3_{1,2}\ N^3_{1,3}]\!], \ldots, \\
\qquad\quad u^3_{n_3} \mapsto \mathcal{R}_3(V^3_{n_3}, C^3_{n_3})[\![N^3_{n_3,1},\, N^3_{n_3,2},\, N^3_{n_3,3}]\!]] \\
n_3 = \#\sigma^3_{\mathcal{L}} = \#\sigma^3_{\mathcal{R}}
\end{array}
\right.
$$

$$
\begin{array}{l}
\Gamma(u^1_i) = (\mathit{Val} \to pr) \to pr,\ \Gamma(u^2_i) = \mathit{Val},\ \Gamma(u^3_i) = pr\ \wedge \\
V = \langle x_1, \ldots, x_n \rangle\ \wedge \\
Z = \langle (l_1, P_1), \ldots, (l_k, P_k) \rangle, k \leq n \text{ s.t.} \\
\quad \text{for each} (M, N) \in \{(M^i_j, N^i_j)\} \cup \{(M_0, N_0)\}, \\
\quad \wedge\ \Gamma \vdash M \stackrel{AE}{\rightsquigarrow}_{V|Z} N{:}\tau \\
\quad \wedge\ \Sigma_M = \Sigma'_M[l_1 \mapsto P'_1, \ldots, l_k \mapsto P'_k] \\
\quad \wedge\ \Gamma \vdash \Sigma'_M \stackrel{AE}{\rightsquigarrow}_{V|Z} \Sigma_N \\
\quad \wedge\ \Gamma \vdash P'_i \stackrel{AE}{\rightsquigarrow}_{V|Z} P_i{:}\mathit{Val}, 1 \leq i \leq k \\
\}
\end{array}
$$

Figure 16: Definition of S_{AE}

simulation relation, rather than different V's and Z's for different closures.

We show that S_{AE} includes the translations of Scheme expressions before and after assignment elimination:

Lemma 5.3.5

For any Scheme expression e, any type assumption Γ such that $\Gamma \vdash [\![e]\!]_{scm}{:}(\mathit{Val}{\rightarrow}pr){\rightarrow}pr$, any metalanguage context $C[-_{(\mathit{Val}\rightarrow pr)\rightarrow pr}]{:}pr$ which respects Γ, and any store Σ,

$$\langle\!\langle C[[\![e]\!]_{scm}]\ ;\ \Sigma\rangle\!\rangle\ S_{AE}\ \langle\!\langle C[[\![AE(e)]\!]_{cell}]\ ;\ \Sigma\rangle\!\rangle$$

PROOF:

The Z in the $\overset{AE}{\rightsquigarrow}_{V|Z}$ relation will be $\langle\rangle$ (the empty sequence). The σ^2 and σ^3 substitutions will both be empty, and the σ^1 substitutions will have elements for the translation of each lambda expression in e. The V in $\overset{AE}{\rightsquigarrow}_V$ will contain the immutable local variables of the lambda-expressions in e.

□

Before proving that S_{AE} is a simulation, we need a lemma about ref-typed terms:

Lemma 5.3.6 *If $\Gamma \vdash M \overset{AE}{\rightsquigarrow}_{V|Z} N{:}\mathit{ref}(\tau)$, and $(M, N) \notin Z$, then*

$$M \rightarrow^*_{pcf} l \Leftrightarrow N \rightarrow^*_{pcf} l$$

PROOF SKETCH:

LET: 1. $V = \langle x_1 \ldots x_n\rangle$

2. $Z = \langle (l_1, P_1) \ldots (l_k, P_k)\rangle, \quad k \leq n$

3. $\Gamma \vdash M' \overset{AE}{\rightsquigarrow}_V N'{:}\mathit{ref}(\mathit{Val})$ s.t.

4. $M = M'[l_i/x_i]_{i=1}^k,\ N = N'[P_i/x_i]_{i=1}^k$

5. $M \overset{s}{\rightarrow}_{left,pcf} l$ (by PCF standardization, Lemma 2.5.7)

We use induction on the length s of the leftmost reduction of M to a location l, considering all possible forms of M'. Most cases are ruled out:

$\langle 1\rangle 1$. CASE: $M' = x_i$

is ruled out by $(M, N) \notin Z$.

$\langle 1\rangle 2$. CASE: $M' = (\mathsf{deref}\ldots)$, or $(\mathsf{update}\ldots)$ or $(\mathsf{io}\ldots)$

are ruled out by types.

$\langle 1\rangle 3$. CASE: $M' = l$

By $\Gamma \vdash M' \overset{AE}{\leadsto}_V N':\mathit{ref}(\mathit{Val})$, and by (CONG-LOC) rule, we must have $N' = l = N$.

$\langle 1\rangle 4$. CASE: all other cases

The "interesting" rule of Figure 14 is ruled out by the type of $\Gamma \vdash M' \overset{AE}{\leadsto}_V N':\mathit{ref}(\tau)$. Thus M' and N' are in $\overset{AE}{\leadsto}_V$, at the top level, via a congruence rule. Because all of the store and I/O operators have been ruled out, that leaves head PCF redexes. We give the β redex case as an example.

$\langle 2\rangle 1$. CASE: $M' = (\lambda x{:}\tau'.M_0)\ M_1 \ldots M_k$

We know that $\tau' = \tau_1 \to \ldots \to \tau_k$ and that $\Gamma[x{:}\tau'] \vdash M_0{:}\mathit{ref}(\tau)$. By repeated use of types, we argue that only (CONG-APP) and (CONG-ABS) rules apply and that therefore $\Gamma \vdash M' \overset{AE}{\leadsto}_V N':\mathit{ref}(\mathit{Val})$ implies that $N' = (\lambda x{:}\tau'.N_0)N_1 \ldots N_k$, with $\Gamma \vdash M_i \overset{AE}{\leadsto}_V N_i{:}\tau_i$ for $0 \leq i \leq k$. So we have:

$\langle 3\rangle 1$. $\mathbf{lhs} = (\lambda x.M_0)\ M_1 \ldots M_k$

$=_\beta M_0[M_1/x]\ M_2 \ldots M_k$

$\langle 3\rangle 2$. $\mathbf{rhs} = (\lambda x.N_0)\ N_1 \ldots N_k$

$=_\beta N_0[N_1/x]\ N_2 \ldots N_k$

$\langle 3\rangle 3$. Q.E.D.

By Lemma 5.3.2, and by the assumptions $\Gamma \vdash M_i \overset{AE}{\leadsto}_V N_i{:}\tau_i$ for $0 \leq i \leq k$, we have that both sides are in $\overset{AE}{\leadsto}_V$.

□

We are now ready to prove that the S_{AE} relation is contained in $\lesssim_{sim}$. What we actually prove is that

$$S_{AE} \subseteq [\lesssim_{sim} \circ S_{AE} \circ \lesssim_{sim}]_{sim} \tag{2}$$

For example, it is often the case that we argue that two states are in S_{AE} up to α_{loc}. But by Lemma 2.5.11, we know that $\alpha_{loc} \subseteq \lesssim_{sim}$, so we satisfy equation 2.

Lemma 5.3.7

$$S_{AE} \subseteq [\lesssim_{sim} \circ S_{AE} \circ \lesssim_{sim}]_{sim}$$

PROOF:

LET: 1. $(\langle\langle M \,;\, \Sigma_M \rangle\rangle \sigma_{\mathcal{L}},\ \langle\langle N \,;\, \Sigma_N \rangle\rangle \sigma_{\mathcal{R}}) \in S_{AE}$,

2. $\langle\langle M \,;\, \Sigma_M \rangle\rangle \sigma_{\mathcal{L}} \xrightarrow{s}_{left,sto} \langle\langle \mathsf{io}\, M' \,;\, \Sigma'_M \rangle\rangle$

PROVE: $\exists N', \Sigma'_N$ s.t. $\langle\langle N \,;\, \Sigma'_N \rangle\rangle \sigma_{\mathcal{R}} \to^*_{sto} \langle\langle \mathsf{io}\, N' \,;\, \Sigma'_N \rangle\rangle$ and

$\langle\langle M' \,;\, \Sigma'_M \rangle\rangle\ S_{AE}\ \langle\langle N' \,;\, \Sigma'_N \rangle\rangle$

PROOF SKETCH: Use induction on length s of leftmost reduction, by considering all possible cases of (M, N). Most of the interesting cases correspond to M and N headed by a variable u^i_j. The other interesting case corresponds to a reference to an immutable variable.

---

1. $M = (u^1_j\ M_\kappa),\ N = (u^1_j\ N_\kappa)$,
2. $M = (\mathsf{case}\, u^2_j\ M_1\ (\lambda f.f\ M_Q\ M_K)),\ N = (\mathsf{case}\, u^2_j\ N_1(\lambda f.f\ N_Q\ N_K))$
3. $M = u^3_j,\ N = u^3_j$
4. $M = (\mathsf{deref}\, M_1\ M_2),\ N = (N_1\ N_2)$
5. $M = (\mathsf{deref}\, M_1\ M_2),\ N = (\mathsf{deref}\, N_1\ N_2)$
6. $M = (\mathsf{update}\, M_1\ M_2\ M_3),\ N = (\mathsf{update}\, N_1\ N_2\ N_3)$
7. $M = (\mathsf{new}\, M_1\ M_2),\ N = (\mathsf{new}\, N_1\ N_2)$
8. $M = (\mathsf{io}\, M_1), N = (\mathsf{io}\, N_1)$
9. $M = ((\lambda x.M_0)\ M_1 \ldots M_k),\ N = ((\lambda x.N_0)\ N_1 \ldots N_k)$
10. other PCF head redexes

$\langle 1\rangle 1.$ CASE: $M = (u^1_j\ M_\kappa),\ N = (u^1_j\ N_\kappa)$

LET: 1. $\sigma^1_{\mathcal{L}}(j) = \mathcal{L}_1(X^1_j)[\![M^1_j]\!]$

2. $\sigma^1_{\mathcal{R}}(j) = \mathcal{R}_1(V^1_j, C^1_j)[\![N^1_j]\!]$

$\langle 2\rangle 1.$ $\mathbf{lhs} = \langle\!\langle(\mathcal{L}_1(X^1_j)[\![M^1_j]\!]\ M_\kappa)\ ;\ \Sigma_M\rangle\!\rangle\sigma_{\mathcal{L}}$

$\rightarrow^*_{\mathit{left,pcf}}\ \langle\!\langle(M_\kappa\ \mathcal{L}_2(X^1_j)[\![M^1_j]\!])\ ;\ \Sigma_M\rangle\!\rangle\sigma_{\mathcal{L}}$

$\langle 2\rangle 2.$ $\mathbf{rhs} = \langle\!\langle(\mathcal{R}_1(V^1_j, C^1_j)[\![N^1_j]\!]\ N_\kappa)\ ;\ \Sigma_N\rangle\!\rangle\sigma_{\mathcal{R}}$

$\rightarrow^*_{\mathit{pcf}}\ \langle\!\langle(N_\kappa\ \mathcal{R}_2(X^1_j)[\![N^1_j]\!])\ ;\ \Sigma_N\rangle\!\rangle\sigma_{\mathcal{R}}$

$\langle 2\rangle 3.$ Q.E.D.

Both sides stay in S_{AE} by adding corresponding new elements to $\sigma^2_{\mathcal{L}}$ and $\sigma^2_{\mathcal{R}}$. For example, assume the variable u^2_n is fresh. To $\sigma^2_{\mathcal{L}}$, we add the pair $(u^2_n, (\mathcal{L}_2(X^1_j)[\![M^1_j]\!]))$, and to $\sigma^2_{\mathcal{R}}$, we add the pair $(u^2_n, (\mathcal{R}_2(X^1_j)[\![N^1_j]\!]))$. We assume that $\Gamma \vdash M_\kappa \overset{AE}{\rightsquigarrow}_{V|Z} N_\kappa{:}\mathit{Val}{\rightarrow}\mathit{pr}$, and so the pair $(\langle\!\langle(M_\kappa\ u^2_n)\ ;\ \Sigma_M\rangle\!\rangle\sigma_{\mathcal{L}}, \langle\!\langle(N_\kappa\ u^2_n)\ ;\ \Sigma_N\rangle\!\rangle\sigma_{\mathcal{R}})$ is in S_{AE}.

□

$\langle 1\rangle 2.$ CASE: $M = (\mathsf{case}\, u^2_j\ M_1\ (\lambda f.f\ M_Q\ M_K)),\ N = (\mathsf{case}\, u^2_j\ N_1(\lambda f.f\ N_Q\ N_K))$

Same as Step $\langle 1\rangle 1$, in that both sides take β steps to stay in S_{AE}. For this step, we add corresponding new elements to $\sigma^3\mathcal{L}$ and $\sigma^3\mathcal{R}$.

$\langle 1\rangle 3.$ CASE: $M = u^3_j,\ N = u^3_j$

LET: 1. $\sigma^3_{\mathcal{L}}(j) = \mathcal{L}_3(X^3_j)[\![M^3_{j,1},\ M^3_{j,2},\ M^3_{j,3}]\!]$,

2. $\sigma^3_{\mathcal{R}}(j) = \mathcal{R}_3(V^3_j, C^3_j)[\![N^3_{j,1},\ N^3_{j,2},\ N^3_{j,3}]\!]$

ASSUME: $\Gamma \vdash M^3_{j,i} \overset{AE}{\rightsquigarrow}_V N^3_{j,i}{:}\tau,\ 1 \le i \le 3$

$\langle 2\rangle 1.$ $\mathbf{lhs} = \langle\!\langle \mathsf{new}^{\mathit{Val}}(\mathit{listref}_1\ M^3_{j,2})\ (\lambda x_1{:}\mathit{ref}(\mathit{Val}).\cdots$

$\mathsf{new}^{\mathit{Val}}(\mathit{listref}_m\ M^3_{j,2})\ (\lambda x_m{:}\mathit{ref}(\mathit{Val}).$

$(M^3_{j,1}\ M^3_{j,3})\ldots))\ ;\ \Sigma_M\rangle\!\rangle\sigma_{\mathcal{L}}$

if $X^3_j = \langle x_1, \ldots, x_m\rangle$

$\overset{\geq 0}{\rightarrow}_{\mathit{left,sto}}\ \langle\!\langle(M^3_{j,1}\ M^3_{j,3})[l_i/x_i]^m_{i=1}\ ;$

$\Sigma_M[l_i \mapsto (\mathit{listref}_i\ M^3_{j,2})]^m_{i=1}$

$= \mathbf{lhs}'$

$$\langle 2\rangle 2.\ \ \mathbf{rhs} = \left.\begin{array}{l}\langle\!\langle\, \mathsf{new}^{Val}(\mathit{listref}_i\ N^3_{j,2}) \\ (\lambda x_i{:}\mathit{ref}(\mathit{Val}).\end{array}\right\}\ \text{for each } x_i \in C^3_j$$

$$(N^3_{j,1}\,[(\mathit{listref}_i\ N^3_{j,2})/x_i]_{x_i\in V^3_j}\ N^3_{j,3})\ \ldots)\ ;\ \Sigma_N\rangle\!\rangle\sigma_{\mathcal{R}}$$

$$\to^*_{sto}\ \langle\!\langle (N^3_{j,1}\,[(\mathit{listref}_i\ N^3_{j,2})/x_i]_{x_i\in V^3_j}\,[l_i/x_i]_{x_i\in C^3_j}\ N^3_{j,3})\ ;$$
$$\Sigma_N[l_i \mapsto (\mathit{listref}_i\ N^3_{j,2})]_{i\in C^3_j}\rangle\!\rangle\sigma_{\mathcal{R}}$$
$$= \mathbf{rhs}'$$

$\langle 2\rangle 3.$ Q.E.D.

To get $(\mathbf{lhs}', \mathbf{rhs}')$ in S_{AE}, we need to add some pairs to the Z component of the $\overset{AE}{\rightsquigarrow}_{V|Z}$ relation. Specifically, add the following pairs to Z:

$$\{(l_i, (\mathit{listref}\ N^3_{j,2}))\mid x_i \in V^3_j\}$$

$\langle 1\rangle 4.$ Case: $M = (\mathsf{deref}\, M_1\ M_2),\ N = (N_1\ N_2)$

It must be that we have a case of the $([\![x]\!], x$ IMMUTABLE$)$ rule from Figure 14.

Let: 1. $M = M''[l_i/x_i]^k_{i=1},\ N = N''[P_i/x_i]^k_{i=1}$,

for $x_i \in X$ and $(l_i, P_i) \in Z$, $1 \le i \le k$.

2. $M'' = (\mathsf{deref}\, x_i\ M''_2)$ and $N'' = (N''_1\ x_i)$,

3. $\Gamma \vdash M''_2 \overset{AE}{\rightsquigarrow}_V N''_1{:}\mathit{Val}{\to}pr$

4. $\Sigma_M(l_i) = P'_i$ s.t. $P'_i \overset{AE}{\rightsquigarrow}_{V|Z} P_i$

So, we have $\Gamma \vdash M'' \overset{AE}{\rightsquigarrow}_V N''{:}pr$ by rule ($[\![x]\!]$, x IMMUTABLE).

$\langle 2\rangle 1.\ \ \mathbf{lhs} = \langle\!\langle \mathsf{deref}\, l_i\ M_2\ ;\ \Sigma_M\rangle\!\rangle\sigma_{\mathcal{L}}$

$$\overset{1}{\to}_{\mathit{left},sto}\ \langle\!\langle (M_1\ P'_i)\ ;\ \Sigma_M\rangle\!\rangle\sigma_{\mathcal{L}}$$
$$\overset{\le s}{\to}_{\mathit{left},sto}\ \langle\!\langle \mathsf{io}\, M'\ ;\ \Sigma'_M\rangle\!\rangle$$

$\langle 2\rangle 2.\ \ \mathbf{rhs} = \langle\!\langle (N_1\ P_i)\ ;\ \Sigma_N\rangle\!\rangle\sigma_{\mathcal{R}}$

$$\to^*_{sto}\ \langle\!\langle \mathsf{io}\, N'\ ;\ \Sigma'_N\rangle\!\rangle \qquad \text{by IH}(< s) \text{ and (CONG-APP)}$$

$\langle 1\rangle 5.$ Case: $M = (\mathsf{deref}\, M_1\ M_2),\ N = (\mathsf{deref}\, N_1 N_2)$

We have $\Gamma \vdash M' \overset{AE}{\rightsquigarrow}_V N'{:}pr$ by rule (CONG-CONST) for the deref constant, for $M = M'\sigma_{\mathcal{L}}$ and $N = N'\sigma_{\mathcal{R}}$.

Let: 1. $\Gamma \vdash M_1 \overset{AE}{\rightsquigarrow}_{V|Z} N_1{:}\mathit{ref}(\tau)$

2. $\Gamma \vdash M_2 \overset{AE}{\rightsquigarrow}_{V|Z} N_2{:}\tau{\to}pr$

$\langle 2\rangle 1.$ **lhs** $= \langle\!\langle \mathsf{deref}\, M_1\ M_2\ ;\ \Sigma_M \rangle\!\rangle \sigma_{\mathcal{L}}$

$\rightarrow^*_{left,pcf}\ \langle\!\langle \mathsf{deref}\, l\ M_2\ ;\ \Sigma_M \rangle\!\rangle \sigma_{\mathcal{L}}$

$\xrightarrow{1}_{left,sto}\ \langle\!\langle (M_2\ P')\ ;\ \Sigma_M \rangle\!\rangle \sigma_{\mathcal{L}}$ assuming $\Sigma_M(l) = P'$

$\xrightarrow{\leq s}_{left,sto}\ \langle\!\langle \mathsf{io}\, M'\ ;\ \Sigma'_M \rangle\!\rangle$

$\langle 2\rangle 2.$ **rhs** $= \langle\!\langle \mathsf{deref}\, N_1\ N_2\ ;\ \Sigma_N \rangle\!\rangle \sigma_{\mathcal{R}}$

$\rightarrow^*_{pcf}\ \langle\!\langle \mathsf{deref}\, l\ N_2\ ;\ \Sigma_N \rangle\!\rangle \sigma_{\mathcal{R}}$ by Lemma 5.3.6

$\xrightarrow{1}_{sto}\ \langle\!\langle (N_2\ P'')\ ;\ \Sigma_N \rangle\!\rangle \sigma_{\mathcal{R}}$ with $P' \overset{AE}{\rightsquigarrow}_{V|Z} P''$

$\rightarrow^*_{sto}\ \langle\!\langle \mathsf{io}\, N'\ ;\ \Sigma'_N \rangle\!\rangle$ by IH($< s$) and (CONG-APP)

To use Lemma 5.3.6, we must show that $M_1 \notin dom(Z)$. We know this because the deref expression was built with the (CONG-CONST) rule for $c = \mathsf{deref}$, and the locations in Z correspond to those in the first deref case. Furthermore, those variables x which are replaced at runtime by the locations in Z are excluded from the (CONG-VAR) rule.

$\langle 1\rangle 6.$ CASE: $M = (\mathsf{update}\, M_1\ M_2\ M_3)$, $(N = \mathsf{update}\, N_1\ N_2\ N_3)$

We have $\Gamma \vdash M' \overset{AE}{\rightsquigarrow}_V N'{:}pr$ by rule (CONG-CONST) for the update constant.

PROOF: We again want to use Lemma 5.3.6 to show that M_1 and N_1 reduce to the same location, and so we need to show that $(M_1, N_1) \notin Z$. Assume for the moment that (M_1, N_1) is in Z; then M_1 must be a location, l. However, the locations in Z are a record of substitutions for variables in the list V, and V is a list of variables which never occur as the argument to update – contradiction. Thus M_1 cannot be in the domain of Z, and we can use Lemma 5.3.6 to get $M_1 \rightarrow^*_{pcf} l$ and $N_1 \rightarrow^*_{pcf} l$. Now both sides can then take a single step and remain in S_{AE}. We can then invoke the induction hypothesis at $j < s$ to get the result.

$\langle 1\rangle 7.$ CASE: $M = (\mathsf{new}\, M_1\ M_2)$, $N = (\mathsf{new}\, N_1\ N_2)$

We have $\Gamma \vdash M' \overset{AE}{\rightsquigarrow}_V N'{:}pr$ by rule (CONG-CONST) for the new constant.

$\langle 2\rangle 1.$ **lhs** $= \langle\!\langle \mathsf{new}\, M_1\ M_2\ ;\ \Sigma_M \rangle\!\rangle \sigma_{\mathcal{L}}$

$\xrightarrow{1}_{left,sto}\ \langle\!\langle (M_2\ l)\ ;\ \Sigma_M[l \mapsto M_1] \rangle\!\rangle \sigma_{\mathcal{L}}$

$\xrightarrow{j \leq s}_{left,sto}\ \langle\!\langle \mathsf{io}\, M'\ ;\ \Sigma'_M \rangle\!\rangle$

$\langle 2 \rangle 2.$ **rhs** $= \langle\!\langle \mathsf{new}\, N_1\ N_2\ ;\ \Sigma_N \rangle\!\rangle \sigma_{\mathcal{R}}$

$\rightarrow_{sto} \langle\!\langle (N_2\ l')\ ;\ \Sigma_N[l' \mapsto N_1] \rangle\!\rangle \sigma_{\mathcal{R}}$

$\rightarrow^*_{sto} \langle\!\langle \mathsf{io}\, N'\ ;\ \Sigma'_N \rangle\!\rangle$ by α_{loc} and IH(j)

PROOF: We really have to show that rhs is in α_{loc} with $\langle\!\langle (N_2\ l)\ ;\ \Sigma_N[l \mapsto N_1] \rangle\!\rangle$, and we can invoke IH on that term. Lemma 2.5.11 says that α_{loc} preserves $\lesssim_{sim}$, so we are OK. This line of reasoning is carried out more painstakingly in the proof of the new case in the substitution lemma.

$\langle 1 \rangle 8.$ CASE: $M = \mathsf{io}\, M_1$

PROOF: Immediate.

$\langle 1 \rangle 9.$ CASE: $M = (\lambda x.M_0)\ M_1 \ldots M_k$, $N = (\lambda x.N_0)\ N_1 \ldots N_k$

We have $\Gamma \vdash M' \overset{AE}{\rightsquigarrow}_V N' {:} pr$ by rule (CONG-ABS).

PROOF: Both sides can take a β step and stay in S_{AE}. We can then invoke IH$(s-1)$.

$\langle 1 \rangle 10.$ CASE: Other PCF head redexes are all straightforward.

□

Corollary 5.3.8

$$S_{AE} \subseteq\ \lesssim_{sim}$$

□

Now we are ready to use the S_{AE} simulation to prove the correctness of assignment elimination.

Recall that by Lemma 4.3.1 we know that, for Γ mapping free variables of e to $\mathit{ref}(\mathit{Val})$,

$$\Gamma \vdash [\![e]\!]_{scm} {:} (\mathit{Val} \rightarrow pr) \rightarrow pr \tag{3}$$

By Lemma 5.2.2, for Γ mapping e's free variables to $\mathit{ref}(\mathit{Val})$, we know that

$$\Gamma \vdash_{cell} AE(e) {:} ok$$

And by Lemma 5.1.1, we can deduce that

$$\Gamma \vdash [\![AE(e)]\!]_{cell} {:} (\mathit{Val} \rightarrow pr) \rightarrow pr \tag{4}$$

Thus the statement of the correctness of assignment elimination is as follows:

Theorem 5.3.9 (Correctness of Assignment Elimination)
For any Scheme expression e, and any Γ mapping the free variables of $[\![e]\!]_{scm}$ and $[\![AE(e)]\!]_{cell}$ to $ref(Val)$,

$$\Gamma \vdash [\![e]\!]_{scm} \cong_{\forall C} [\![AE(e)]\!]_{cell}{:}(Val{\to}pr){\to}pr$$

PROOF:

1. By the Extensionality Theorem, 2.8.1, it suffices to show that for all closing substitutions σ mapping free variables to closed terms of type $ref(Val)$, all closed terms K of type $Val{\to}pr$, and all stores Σ,
$$\langle\!\langle([\![e]\!]_{scm}\sigma)\ K\ ;\ \Sigma\rangle\!\rangle \lesssim_{sim} \langle\!\langle([\![AE(e)]\!]_{cell}\sigma)\ K\ ;\ \Sigma\rangle\!\rangle$$
2. The effects of any closing substitution σ can be duplicated by using a closing context $C_\sigma[-]$ such that the identifiers in scope at the hole correspond to the domain of σ and such that the binders of those identifiers are applied to the same elements as in the range of σ.
3. By Lemma 5.3.5,
$$\langle\!\langle C_\sigma[[\![e]\!]_{scm}]\ K\ ;\ \Sigma\rangle\!\rangle\ S_{AE}\ \langle\!\langle C_\sigma[[\![AE(e)]\!]_{cell}]\ K\ ;\ \Sigma\rangle\!\rangle$$
for $C = (C_\sigma[-]\ K)$.
4. By Corollary 5.3.8, $S_{AE} \subseteq \lesssim_{sim}$, and we are done.

□

Chapter 6

Lambda Lifting – A Complex Source-to-Source Transformation

Lambda lifting transforms lambda abstractions with free variables into lambda abstractions with fewer free variables by adding those free variables to the parameter lists of the lifted abstractions. Once the free variables of a nested procedure have been converted to arguments, the nested procedure can be lifted out of the scope of the bindings of those variables. Lambda-lifting is done in the hope that lifting a procedure to a more global scope will reduce the number of times a closure will have to be created for that procedure. Further, it is hoped that the savings in reduced closure allocation will make up for the overhead of having to pass additional arguments to the lifted procedure.

Traditional lambda lifting lifts procedures to the outermost scope, adding all free variables of a procedure to its argument list. The Twobit Scheme compiler of Clinger and Hansen [CH94] implements *incremental lambda lifting*, which lifts nested procedures up one level – out of an enclosing abstraction. Only variables which are free in the nested procedure and bound by the enclosing abstraction are added to the procedure's argument list. The process of incremental lambda lifting can be performed repeatedly in order to bring a deeply-nested procedure to the top level, as would be done with traditional lambda lifting. However, being able to lift only one level at a time gives the compiler added flexibility.

In order for the lambda-lifting transformation to be correct, clearly all calls to a lifted procedure must pass any new parameters. This requires that all call sites be known, which rules out lifting procedures that may escape their scope of definition. Also, there can be no side-effects to the once-free-now-bound variables within a lifted procedure, as the side-effects would, as specified in the Scheme semantics, be modifying a different cell in the lambda-lifted body than in the original.

We prove the correctness of incremental lambda lifting as a transformation on Scheme$_{cell}$ terms. Therefore, we may assume that $[\![-]\!]_{cell}$ is the translation into the metalanguage. In particular, no storage is allocated for arguments to an abstraction, so there is no danger of a bad interaction between store updates and store allocation for parameter passing. Thus what we actually prove is the correctness of lambda lifting composed with a meaning-preserving translation from Scheme to Scheme$_{cell}$, such as assignment elimination.

6.1 The Lambda-Lifting Transformation

The lambda-lifting transformation operates on a Scheme$_{cell}$ term of the form

$$
\begin{array}{l}
(\mathsf{lambda}\ (x_1 \ldots x_{m_0})\ (\mathsf{letrec} \\
\qquad ((f_1\ (\mathsf{lambda}\ (y_1 \ldots y_{m_1})\ e_1)) \ldots \\
\qquad\ (f_n\ (\mathsf{lambda}\ (y_1 \ldots y_{m_n})\ e_n))) \\
\quad e_0))
\end{array}
$$

where each of the $f_1 \ldots f_n$ procedures may be lifted out of the scope of the enclosing lambda $(x_1 \ldots x_{m_0})$. Recall that the Scheme letrec is *not* a fixpoint operator – it is a macro for a sequence of set!'s to the identifiers for the procedures being defined. After lambda lifting, each lifted procedure may have additional arguments, taken from the list $x_1 \ldots x_{m_0}$. We use the following notation to denote the different sequences of identifiers:

$$
\begin{aligned}
X_0 &= \langle x_1, \ldots, x_{m_0} \rangle \\
F &= \langle f_1, \ldots, f_n \rangle \\
Y_i &= \langle y_1, \ldots, y_{m_i} \rangle,\ 1 \leq i \leq n \\
X_i &= \{x_j \mid x_j \in X_0 \text{ must be added to the argument list of } f_i\}, \\
&\quad\ 1 \leq i \leq n \\
\mathbb{X} &= \langle X_1, \ldots, X_n \rangle \\
\mathbb{F} &= \langle (f_1, Y_1), \ldots, (f_n, Y_n) \rangle
\end{aligned}
$$

So, X_0 is the list of arguments to the outermost lambda expression; F is the list of local procedure identifiers; Y_i is the list of original arguments to the i^{th} local procedure, f_i; and X_i is the set of identifiers to be added to the argument list of f_i. Each X_i is a subset of the list X_0. We denote the collection of all X_i sequences by the symbol $\mathbb{X}$, and the collection of the local procedure identifiers and their signatures by $\mathbb{F}$.

The length of a sequence Q is denoted by $\#Q$. Sometimes we treat sequences as their underlying sets; allowing, for example, $Q_1 \cup Q_2$. The i^{th} element of a sequence Q is denoted by $Q_{(i)}$; for the i^{th} element of a sequence such as X_k, we write $X_{k(i)}$. To indicate the splicing of a sequence into a larger sequence, we use the notation $[x_i]_{\in Q}$. For example, if $X_5 = \langle x_1, x_3, x_7 \rangle$,

$$(M\ [x_i]_{\in X_5}\ N) \text{ rewrites to } (M\ x_1\ x_3\ x_7\ N)$$

When the splicing operation is obvious, we may simply refer to the sequence. For example, lambda (Y_k) is shorthand for lambda $(y_1 \ldots y_{m_k})$.

The lambda-lifting transformation is defined in several steps. Figure 17 defines the conditions under which lambda lifting may be performed – i.e., when a Scheme$_{cell}$ term e is *lambda liftable* with respect to a list of lists $\mathbb{X}$.

We disallow any conflicts of bound variables – all binding constructs within the procedure bodies e_i, for $0 \leq i \leq n$, must bind variables distinct from the identifiers

$$x_1 \ldots x_{m_0}, f_1, \ldots f_n, y_1 \ldots y_{m_i}, \text{ for } 1 \leq i \leq n$$

This requirement can be satisfied by alpha-renaming, which can easily be proved correct

Definition 6.1.1 (Lambda Liftable)
Let e be an expression in $Scheme_{cell}$, let Γ be a map from the free variables of e to $\{Val, ref(Val)\}$, and let $\mathbb{X}$ be a list of lists of variables. Then e is lambda-liftable with respect to Γ *and* $\mathbb{X}$ *iff:*

1. $\Gamma \vdash_{cell} e{:}ok$
2. $e =$ (lambda (X_0) e'), $\quad X_0 = x_1 \ldots x_{m_0}$, *and*
3. $e' =$ (letrec
   $\quad$ ((f_1 (lambda (Y_1) e_1)), $\quad Y_1 = y_1 \ldots y_{m_1}$
   $\quad$ ...
   $\quad$ (f_n (lambda (Y_n) e_n)), $\quad Y_n = y_1 \ldots y_{m_n}$
   $\quad$ e_0)
4. *The variables x_i, y_j, f_k are disjoint, for $1 \leq i \leq m_0$, $1 \leq j \leq max(m_k)$, $1 \leq k \leq n$. Also, all binding constructs within e_0 and the e_k bind identifiers distinct from the x_i, y_j, and f_k.*
5. *All occurrences of each f_k in $e_0, e_1, \ldots, e_n$ are in operator position, for $1 \leq k \leq n$. That is, the f_k cannot be assigned to or passed out of their scope.*
6. *The list of lists of variables, $\mathbb{X}$, has n elements, $X_1, \ldots, X_n$. Letting k range from 1 to n, and treating each X_k as a set, we require that for each k,*
   (a) $X_k \subseteq \{x_1, \ldots, x_{m_0}\}$
   (b) $X_k \supseteq (fv(e_k) \cap X_0) \cup (\bigcup\{X_j \mid f_k \text{ calls } f_j\})$

Figure 17: Lambda Liftable

separately. We require the bound variables to be distinct in order to avoid scenarios such as the following, where procedure f_2 shadows the variable x, confounding the transform of a call to f_1:

$$
\begin{array}{l}
(\mathsf{lambda}\ (x) \\
\quad (\mathsf{letrec}\ \ ((f_1\ (\mathsf{lambda}\ (y) \ldots x \ldots)) \\
\qquad\qquad\ \ (f_2\ (\mathsf{lambda}\ (x) \ldots (f_1\ 42) \ldots))) \\
\qquad\qquad \ldots (f_1\ 6) \ldots (f_2\ 7) \ldots))
\end{array}
$$

gets stuck at:

$$
\begin{array}{l}
(\mathsf{letrec}\ \ ((f_1\ (\mathsf{lambda}\ (x\ y) \ldots x \ldots)) \\
\qquad\quad (f_2\ (\mathsf{lambda}\ (x) \ldots (f_1\ ??\ 42) \ldots))) \\
\qquad (\mathsf{lambda}\ (x) \ldots (f_1\ x\ 6) \ldots (f_2\ 7) \ldots))
\end{array}
$$

Clearly there are various workarounds to the above problem, and we choose simply to require that the bound variables all be distinct.

The nested procedures which we are lifting, $f_1 \ldots f_n$, must be *known local procedures.* That is, the procedures are never passed out of the scope of the outer lambda $(x_1 \ldots x_{m_0})$, and the identifiers $f_1 \ldots f_n$ are never assigned to except by the first letrec. We enforce the restriction that the f_k be local by requiring that all occurrences of the f_k be in *operator position.* More sophisticated analyses are possible, but that is another research topic.

We require that, for a given f_k, the arguments to be added are in a solution to (a fixed point of) a set of constraints. Before lifting f_k outside the scope of lambda $(x_1 \ldots x_{m_0})$, any of the x_i referenced in e_k must be added to f_k's parameter list. More subtly, if f_k calls f_j, i.e. if an application $(f_j\ e_1 \ldots e_{m_j})$ is a subterm of e_k, then the new parameters which must be sent to the lifted f_j must be added (if not already there) to the incoming parameters of f_k. Adding these newly required parameters to f_k may have an effect on the parameters added to any f_i's which, in turn, call f_k – and so on. Eventually a fixed point will be reached. The list of arguments to the outer lambda, $\langle x_1, \ldots, x_{m_0} \rangle$ is always a solution to the set of constraints, though not necessarily the smallest solution.

Definition 6.1.2 (Lambda Lifting (LL))
If $Scheme_{cell}$ expression e is lambda-liftable with respect to a list of lists of arguments to be added $\mathbb{X}$, then the Lambda-lift *of e with respect to $\mathbb{X}$ is defined as:*

$$
\begin{array}{ll}
LL(\mathbb{X}, e) \stackrel{def}{=} (\mathsf{letrec} & \\
& ((f_1\ (\mathsf{lambda}\ ([x_i]_{\in X_1}\ y_1 \ldots y_{m_1}) \\
& \qquad LL_{body}(F, \mathbb{X}, e_1))) \\
& \ \ldots \\
& \ (f_n\ (\mathsf{lambda}\ ([x_i]_{\in X_n}\ y_1 \ldots y_{m_n}) \\
& \qquad LL_{body}(F, \mathbb{X}, e_n)))) \\
& (\mathsf{lambda}\ (x_1 \ldots x_{m_0})\ LL_{body}(F, \mathbb{X}, e_0)))
\end{array}
$$

where $F = \{f_1, \ldots, f_n\}$ and where LL_{body} is defined as follows:

Definition 6.1.3 (LL_{body})

$LL_{body}(F, \mathbb{X}, x) = x$

$LL_{body}(F, \mathbb{X}, n) = n$

$LL_{body}(F, \mathbb{X}, (\mathsf{cell\text{-}ref}\ x)) = (\mathsf{cell\text{-}ref}\ x)$

$LL_{body}(F, \mathbb{X}, (\mathsf{cell\text{-}set!}\ x\ e)) = (\mathsf{cell\text{-}set!}\ x\ LL_{body}(F, \mathbb{X}, e))$

$LL_{body}(F, \mathbb{X}, (f_k\ e_1 \ldots e_{m_k})) =$
$\quad (f_k\ [x_i]_{\in X_k}\ LL_{body}(F, \mathbb{X}, e_1) \ldots LL_{body}(F, \mathbb{X}, e_{m_k}))$
$\qquad$ *if* $f_k \in F$

$LL_{body}(F, \mathbb{X}, (e_1\ \ldots\ e_p)) =$
$\quad (LL_{body}(F, \mathbb{X}, e_1) \ldots LL_{body}(F, \mathbb{X}, e_p)) \quad$ *if* $e_1 \notin F$

$LL_{body}(F, \mathbb{X}, (\mathsf{lambda}\ (y_1 \ldots y_p)\ e)) =$
$\quad (\mathsf{lambda}\ (y_1 \ldots y_p)\ LL_{body}(F, \mathbb{X}, e))$

$LL_{body}(F, \mathbb{X}, (\mathsf{let}\ ((y_1\ rhs_1) \ldots (y_p\ rhs_p))\ e)) =$
$\quad (\mathsf{let}\ ((y_1\ LL_{body}(F, \mathbb{X}, rhs_1)) \ldots (y_p\ LL_{body}(F, \mathbb{X}, rhs_p)))\ LL_{body}(F, \mathbb{X}, e))$

$LL_{body}(F, \mathbb{X}, (\mathsf{letrec}\ ((y_1\ e_1) \ldots (y_p\ e_p))\ e)) =$
$\quad (\mathsf{letrec}\ ((y_1\ LL_{body}(F, \mathbb{X}, e_1)) \ldots (y_p\ LL_{body}(F, \mathbb{X}, e_p)))\ LL_{body}(F, \mathbb{X}, e))$

Figure 18: Lambda Lifting

The lambda-lifting transformation, once we have a liftable e and $\mathbb{X}$, is given in Figure 18. Definition 6.1.2 shows how the function definitions are rewritten to take added parameters, and how the various f_k are now defined outside the scope of the $\mathsf{lambda}\ (x_1 \ldots x_{m_0})$ procedure. Definition 6.1.3 defines how the individual call sites are rewritten to be consistent with the revised function definitions. The only interesting rule is the fifth, where an application of an f_k must have the new arguments added. The remaining rules push the transformation down the Scheme$_{cell}$ syntax tree.

6.2 Correctness of Lambda Lifting

Our goal is to prove that incremental lambda lifting preserves the meaning of Scheme$_{cell}$ terms. That is, for any Scheme$_{cell}$ term e and type assumption Γ mapping the free variables of e to $\mathit{ref}(\mathit{Val})$, if $\Gamma \vdash e{:}ok$ and e is lambda-liftable with respect to a list of lists of variables $\mathbb{X}$, then

$$\Gamma \vdash [\![e]\!]_{cell} \cong_{\forall C} [\![LL(\mathbb{X}, e)]\!]_{cell}{:}(\mathit{Val}{\rightarrow}pr){\rightarrow}pr$$

Of course, for lambda-lifting to be a correct transformation for Scheme expressions, there first needs to be a meaning-preserving transformation from Scheme to Scheme$_{cell}$, such as assignment elimination.

It is important that the metalanguage terms we are concerned with are well-typed. Recall that by Lemma 5.1.1, for a Scheme$_{cell}$ term e, and a type assumption Γ,

$$\Gamma \vdash_{cell} e{:}ok \Rightarrow \Gamma \vdash [\![e]\!]_{cell}{:}(\mathit{Val}{\rightarrow}pr){\rightarrow}pr.$$

A similar condition holds after lambda lifting:

Lemma 6.2.1 *For a Scheme$_{cell}$ term e and a type assumption Γ mapping variables to $\{\mathit{Val}, \mathit{ref}(\mathit{Val})\}$, if e is lambda-liftable with respect to a list of variable lists $\mathbb{X}$,*

$$\Gamma \vdash_{cell} e{:}ok \Rightarrow \Gamma \vdash [\![LL(\mathbb{X}, e)]\!]_{cell}{:}(\mathit{Val}{\rightarrow}pr){\rightarrow}pr$$

PROOF:

Structural induction on $\vdash_{cell}$.

The most interesting case is when we have a variable reference x in e. Either x is in the free variables of e, in which case $\Gamma(x) = \mathit{ref}(\mathit{Val})$, or it occurs within some $\mathsf{lambda}\ (\ldots x \ldots)$ binding. We must show that x is still bound by an enclosing $\mathsf{lambda}\ (\ldots x \ldots)$ in $LL(\mathbb{X}, e)$. If $x \in X_0$ and the reference to x is in a lifted local procedure f_i, then criterion (6b) in Figure 17 guarantees that x will be bound by the enclosing $\mathsf{lambda}\ ([x_i]_{\in X_n}\ y_1 \ldots y_{m_n})$

□

The translation of a Scheme$_{cell}$ expression e, when e is lambda-liftable with respect to a list of variable lists $\mathbb{X}$ according to Definition 6.1.1, is as follows:

$$\begin{aligned}\llbracket e \rrbracket_{cell} = \lambda\kappa{:}\, \mathit{Val}{\to}pr.\ in_F(\lambda x^*{:}\,\mathit{Val}^*.\lambda\kappa'{:}\,\mathit{Val}{\to}pr.\\ (\mathit{checkargs}\ \ m_0\ x^*)\llbracket e' \rrbracket_{cell}[(\mathit{listref}_i\, x^*)/x_i]_{i=1}^{m_0}\ \kappa')\ \kappa\end{aligned}$$

where $\llbracket e' \rrbracket_{cell} = \lambda\kappa{:}\,\mathit{Val}{\to}pr.$

$$\begin{aligned}&\mathsf{new}\,\Omega_{\mathit{Val}}(\lambda f_1{:}\mathit{ref}(\mathit{Val}).\cdots \mathsf{new}\,\Omega_{\mathit{Val}}(\lambda f_n{:}\mathit{ref}(\mathit{Val}).\\ &\llbracket(\mathsf{lambda}\ (Y_1)\ e_1)\rrbracket_{cell}(\lambda v_1{:}\,\mathit{Val}.\,\mathsf{update}\,f_1 v_1 \cdots\\ &\llbracket(\mathsf{lambda}\ (Y_n)\ e_n)\rrbracket_{cell}(\lambda v_n{:}\,\mathit{Val}.\,\mathsf{update}\,f_n v_n\\ &(\llbracket e_0 \rrbracket_{cell}\ \kappa))\cdots)))\end{aligned}$$

Recall that

$$\begin{aligned}\llbracket(\mathsf{lambda}\ (Y_k)\ e_k)\rrbracket_{cell} = \ \ &\lambda\kappa{:}\,\mathit{Val}{\to}pr.\ in_F(\lambda y^*{:}\,\mathit{Val}^*.\lambda\kappa'{:}\,\mathit{Val}{\to}pr.\\ &(\mathit{checkargs}\ \ m_k\ y^*)\llbracket e_k \rrbracket_{cell}[(\mathit{listref}_i\, y^*)/y_i]_{i=1}^{m_k}\ \kappa')\ \kappa\end{aligned}$$

We may do a few β-reductions to get the following metalanguage term:

$$\begin{aligned}\mathbf{lhs} \stackrel{\mathit{def}}{=}\ &\lambda\kappa{:}\,\mathit{Val}{\to}pr.\ in_F(\lambda x^*{:}\,\mathit{Val}^*.\lambda\kappa'{:}\,\mathit{Val}{\to}pr.(\mathit{checkargs}\ \ m_0\ x^*)\\ &\mathsf{new}\,\Omega_{\mathit{Val}}(\lambda f_1{:}\mathit{ref}(\mathit{Val}).\cdots \mathsf{new}\,\Omega_{\mathit{Val}}(\lambda f_n{:}\mathit{ref}(\mathit{Val}).\\ &\mathsf{update}\,f_1(\mathsf{inl}(\lambda y^*{:}\,\mathit{Val}^*.\lambda\kappa'{:}\,\mathit{Val}{\to}pr.\\ &(\mathit{checkargs}\ \ m_1\ y^*)\llbracket e_1 \rrbracket_{cell}\sigma_1\ \kappa'))\cdots\\ &\mathsf{update}\,f_n(\mathsf{inl}(\lambda y^*{:}\,\mathit{Val}^*.\lambda\kappa'{:}\,\mathit{Val}{\to}pr.\\ &(\mathit{checkargs}\ \ m_n\ y^*)\llbracket e_n \rrbracket_{cell}\sigma_n\ \kappa'))\\ &(\llbracket e_0 \rrbracket_{cell}\ \kappa'))\cdots)\sigma_0)\ \kappa\end{aligned}$$

where

$$\sigma_0 = [(listref_i\, x^*)/x_i]_{i=1}^{m_0}, \quad \text{and}$$
$$\sigma_k = [(listref_i\, y^*)/y_i]_{i=1}^{m_k}, \quad \text{for } 1 \le k \le n.$$

The translation of $LL(\mathbb{X}, e)$, on the other hand, is β-equal to the following:

$$
\begin{aligned}
\mathbf{rhs} \stackrel{def}{=}\; & \lambda\kappa{:}\,Val{\rightarrow}pr.\\
& \quad \mathsf{new}\,\Omega_{Val}(\lambda f_1{:}ref\,(Val).\cdots \mathsf{new}\,\Omega_{Val}(\lambda f_n{:}ref\,(Val).\\
& \qquad \mathsf{update}\,f_1(\mathsf{inl}(\lambda y^*{:}\,Val^*.\lambda\kappa'{:}\,Val{\rightarrow}pr.\\
& \qquad\quad (checkargs\;\, m_1\, y^*)[\![LL_{body}(F, \mathbb{X}, e_1)]\!]_{cell}\sigma_1'\;\kappa'))\cdots\\
& \qquad \mathsf{update}\,f_n(\mathsf{inl}(\lambda y^*{:}\,Val^*.\lambda\kappa'{:}\,Val{\rightarrow}pr.\\
& \qquad\quad (checkargs\;\, m_n\, y^*)[\![LL_{body}(F, \mathbb{X}, e_n)]\!]_{cell}\sigma_n'\;\kappa'))\\
& \qquad\quad in_F(\lambda x^*{:}\,Val^*.\lambda\kappa'{:}\,Val{\rightarrow}pr.\\
& \qquad\qquad (checkargs\;\, m_0\, x^*)([\![LL_{body}(F, \mathbb{X}, e_0)]\!]_{cell}\;\kappa')\sigma_0)\;\kappa)\cdots)
\end{aligned}
$$

where

$$\sigma_k' = [(listref_i\, y^*)/X_{k(i)}]_{i=1}^{\#X_k}\; [(listref_{\#X_k+j}\, y^*)/y_j]_{j=1}^{m_k}$$
$$\text{for } 1 \le k \le n$$

Recall that each $X_{k(i)} \in \{x_1, \ldots, x_{m_0}\}$.

The pre-lifted translation, **lhs**, returns a closure which will allocate and store closures for $f_1 \ldots f_n$ whenever called. The translation of the lifted term, **rhs**, allocates and stores the $f_1 \ldots f_n$ closures first, then returns a closure with the identifiers $f_1 \ldots f_n$ bound to the new locations and which merely invokes its (lifted) body e_0 whenever called. Note that on the pre-lifted side, the translation of a local procedure f_k is a closure with body $(\lambda y^* \kappa'.[\![e_k]\!]_{cell}\sigma_k\,\kappa')$, which may have free variables $x_i \in \{x_1, \ldots, x_{m_0}\}$. These free variables are substituted-for at runtime by the σ_0 substitution, which binds the variables $x_1 \ldots x_{m_0}$ to the corresponding elements of x^*. On the lambda-lifted side, however, these free variables have been put into the local argument lists y^* and are no longer free in the local procedure's closure. The σ_0 substitution is the same, but the scope of σ_0 covers only the innermost closure – i.e. σ_0 does not apply to any of the $(\lambda y^*\, \kappa'.[\![LL_{body}(F, \mathbb{X}, e_k)]\!]_{cell}\sigma_k'\,\kappa')$ closures.

To reiterate the motivation for the lambda-lifting transformation: lambda lifting is done in the hope that not having to allocate (as much) storage for the $f_1 \ldots f_n$ closures makes up for the overhead of passing additional arguments to the local procedures. Good register allocation may even reduce the overhead of added argument passing to zero.

By the Extensionality Theorem, Theorem 2.8.1, and by the preceding calculations, in order to prove $[\![e]\!]_{cell} \cong_{\forall C} [\![LL(\mathbb{X}, e)]\!]_{cell}$ at type $(Val \to pr) \to pr$, we need to show that the **lhs** and **rhs** terms are bisimilar when applied to a continuation in a state. For a continuation $K{:}\, Val \to pr$ and a store Σ, the original term simply injects the closure into the Val type (we assume the procedure component is the right summand) and invokes the continuation, having done no new allocation:

$\langle\!\langle \mathbf{lhs}\ K\ ;\ \Sigma \rangle\!\rangle \to^*_{pcf} \langle\!\langle K\ \mathcal{L}(X_0, \mathbb{F})[\![\, [\![e_0]\!]_{cell}, \ldots, [\![e_n]\!]_{cell}\,]\!]\ ;\ \Sigma \rangle\!\rangle$, where

$$\begin{aligned} \mathcal{L}(X_0, \mathbb{F})[\![P_0, \ldots, P_n]\!] = \ & (\mathsf{inr}(\lambda x^*{:}\, Val^*. \lambda\kappa'{:}\, Val \to pr.(checkargs\ \ m_0\ x^*) \\ & \quad \mathsf{new}\, \Omega_{Val}(\lambda f_1{:}ref(Val). \cdots \mathsf{new}\, \Omega_{Val}(\lambda f_n{:}ref(Val). \\ & \quad\quad \mathsf{update}\, f_1\, \mathcal{C}(Y_1)[\![P_1]\!] \cdots \mathsf{update}\, f_n\, \mathcal{C}(Y_n)[\![P_n]\!] \\ & \quad\quad\quad (P_0\ \kappa')) \cdots)\sigma_0)), \qquad \text{and} \end{aligned}$$

$$\mathcal{C}(Y_k)[\![P]\!] = (\mathsf{inr}(\lambda y^*{:}\, Val^*. \lambda\kappa'{:}\, Val \to pr.(checkargs\ \ m_k\ y^*)(P\ \sigma_k)\kappa'))$$

The substitutions $\sigma_0, \sigma_1, \ldots, \sigma_n$ are as defined on page 109. Substitution σ_0 is derived from the signature X_0 of the outer `lambda`, and the σ_k are derived from the signatures Y_k contained in $\mathbb{F} = \langle (f_1, Y_1), \ldots, (f_n, Y_n) \rangle$.

The $\mathcal{L}$ macro defines the structure of the metalanguage term corresponding to the translation under $[\![-]\!]_{cell}$ of a lambda-liftable Scheme$_{cell}$ expression. The P_k parameters to $\mathcal{L}$ correspond to the translations of the procedure bodies $e_0, e_1, \ldots, e_n$.

The macro $\mathcal{C}(Y_k)[\![-]\!]$ corresponds to the closure corresponding to the translation under $[\![-]\!]_{cell}$ of the k^{th} local procedure (`lambda` $(Y_k)e_k$).

The lifted term allocates new locations $l_1, \ldots, l_k$ and stores its procedure closures in these locations before returning the closure for the `lambda` (X_0) procedure:

$\langle\!\langle \mathbf{rhs}\ K\ ;\ \Sigma \rangle\!\rangle \rightarrow^*_{sto} \langle\!\langle (K\ \mathcal{R}(X_0)[\![\ [\![LL_{body}(F, \mathbb{X}, e_0)]\!]_{cell}\ ]\!])[l_k/f_k]^n_{k=1}\ ;\ \Sigma' \rangle\!\rangle$, where

$$\mathcal{R}(X_0)[\![P]\!] = (\mathsf{inr}(\lambda x^*\!:\! Val^*.\lambda\kappa'\!:\! Val \rightarrow pr.(checkargs\ \ m_0\ x^*)(P\ \kappa')\sigma_0)),$$

$$\Sigma' = \Sigma[l_1 \mapsto \mathcal{C}'(X_1, Y_1)[\![\ [\![LL_{body}(F, \mathbb{X}, e_1)]\!]_{cell}\sigma'_1\ ]\!][l_k/f_k]^n_{k=1}, \ldots$$
$$l_n \mapsto \mathcal{C}'(X_n, Y_n)[\![\ [\![LL_{body}(F, \mathbb{X}, e_n)]\!]_{cell}\sigma'_n\ ]\!][l_k/f_k]^n_{k=1}],$$

$$\mathcal{C}'(X_k, Y_k)[\![P]\!] = (\mathsf{inr}(\lambda y^*\!:\! Val^*.\lambda\kappa'\!:\! Val \rightarrow pr.(checkargs\ \ m_k\ y^*)(P\ \sigma'_k)\kappa'))$$

Again, $\sigma_0, \sigma'_1, \ldots, \sigma'_n$ are as defined on page 109. The substitutions σ'_k require both the original signatures Y_k and also the added parameters X_k.

The $\mathcal{R}$ macro corresponds to the translation of a lambda-lifted Scheme$_{cell}$ expression. The P parameter to $\mathcal{R}$ corresponds to the translation of the lambda-lifted body of the e_0 procedure body.

We next consider what happens at runtime when the closures returned by the original and lifted terms are actually used. In the original Scheme$_{cell}$ terms, this would correspond to the application of the abstraction returned from evaluation of e to m_0 arguments. The non-lifted metalanguage term allocates n fresh locations, storing new instances of the closures for its local procedures at these locations. The run-time behavior will look like the following:

$$\langle\!\langle (out_F\ \mathcal{L}(X_0, \mathbb{F})[\![[\![e_0]\!]_{cell}, \ldots, [\![e_n]\!]_{cell}]\!](\lambda f\!:\! Val^* \rightarrow (Val \rightarrow pr) \rightarrow pr.$$
$$(f\ \langle M_1, \ldots, M_{m_0}\rangle\ M_\kappa)))$$
$$;\ \Sigma \rangle\!\rangle$$
$$\rightarrow^*_{sto} \langle\!\langle ([\![e_0]\!]_{cell}\ M_\kappa)\sigma^M_0[l_i/f_i]^n_{i=1}\ ;\ \Sigma'' \rangle\!\rangle$$

where

$$\Sigma''(l_k) = \mathcal{C}(Y_k)[\![[\![e_k]\!]_{cell}]\!]\sigma_0[l_i/f_i]^n_{i=1}$$
$$\sigma^M_0 \stackrel{def}{=} [M_i/x_i]^{m_0}_{i=1}$$

The substitution σ^M_0 is the "instantiation" of the σ_0 substitution, with metalanguage terms taking the place of list references.

An important aspect of the preceding calculation is that the σ^M_0 substitution, which substitutes elements of x^* (namely $M_1 \ldots M_{m_0}$) for x_i, $1 \leq i \leq m_0$, is in effect during the allocation for, and storage of, the local procedure closures. Thus any references to the x_i

variables in the procedure bodies e_k are captured by the σ_0^M substitution before the closure is put in the store. Recall that in the lifted metalanguage term the closures for the local procedures are stored earlier. Specifically, the closures for the local procedures are stored when the closure for the $\mathsf{lambda}\ (X_0)$ procedure is created, but before that closure is ever applied.

Thus, in the lambda-lifted term, much less happens when the closure for $\mathsf{lambda}\ (X_0)$ is actually applied. We are assured that there will already be a substitution active, $[l_i/f_i]_{i=1}^n$, mapping the procedure identifiers to locations:

$$\begin{aligned}
&\langle\!\langle (out_F\, \mathcal{R}(X_0)[\![[\![e_0]\!]_{cell}]\!][l_i/f_i]_{i=1}^n\ (\lambda f\colon Val^*\to(Val\to pr)\to pr.\\
&\qquad (f\ \langle N_1,\ldots,N_{m_0}\rangle\ N_\kappa)))\\
&\quad ;\ \Sigma'\rangle\!\rangle\\
&\to_{pcf}^*\ \langle\!\langle ([\![e_0]\!]_{cell}\ N_\kappa)\sigma_0^N[l_i/f_i]_{i=1}^n\ ;\ \Sigma'\rangle\!\rangle
\end{aligned}$$

where

$$\begin{aligned}
\Sigma'(l_k) &= \mathcal{C}'(X_k,Y_k)[\![\,[\![LL_{body}(F,\mathbb{X},e_k)]\!]_{cell}\,]\!][l_i/f_i]_{i=1}^n\\
\sigma_0^N &\stackrel{def}{=} [N_i/x_i]_{i=1}^{m_0}
\end{aligned}$$

Finally, let us consider what happens at runtime when a local procedure f_k is applied. In the original Scheme$_{cell}$ term, an application such as $(f_k\ e_1\ldots e_{m_k})$ will translate into a standard form in the metalanguage, for which we define the $\mathcal{A}$ macro:

$$[\![(f_k\ e_1\ldots e_{m_k})]\!]_{cell} = (\lambda\kappa\colon Val\to pr.\mathcal{A}[\![\,f_k,[\![e_1]\!]_{cell},\ldots[\![e_{m_k}]\!]_{cell},\kappa\,]\!])$$

where

$$\begin{aligned}
&\mathcal{A}[\![f,P_1,\ldots,P_n,P_\kappa]\!] =\\
&\qquad (\mathsf{deref}\ f\ (\lambda v_f\colon Val.P_1\ (\lambda v_1\colon Val\ldots P_n\ (\lambda v_n.\\
&\qquad\quad out_F\ v_f\ (\lambda f'\colon Val^*\to(Val\to pr)\to pr.\\
&\qquad\qquad f'\ \langle v_1,\ldots,v_n\rangle\ P_\kappa)))))
\end{aligned}$$

That is, the closure for the procedure is retrieved from the store (at run-time, the identifier f_k will be bound to a location), each of the arguments to the procedure is evaluated, and finally the body of the closure is invoked with a tuple of arguments and a continuation.

On the lambda-lifted side, the added arguments must be passed, and so we will see a term such as:

$\mathcal{A}'[\![f_k, X_k, [\![e_1]\!]_{cell}, \ldots [\![e_{m_k}]\!]_{cell}, \kappa]\!]$

where

$$\begin{array}{l}\mathcal{A}'[\![f, X_k, P_1, \ldots, P_n, P_\kappa]\!] = \\ \qquad (\mathsf{deref}\, f\ (\lambda v_f{:}\,Val.P_1\ (\lambda v_1{:}\,Val \ldots P_n\ (\lambda v_n. \\ \qquad\quad out_F\, v_f\ (\lambda f'{:}\,Val^{*}{\rightarrow}(Val{\rightarrow}pr){\rightarrow}pr. \\ \qquad\qquad f'\ \langle\, [x_i]_{\in X_k}\, v_1, \ldots, v_n\rangle\ P_\kappa)))))\end{array}$$

The differences between $[\![e]\!]_{cell}$ and $[\![LL_{body}(F, \mathbb{X}, e)]\!]_{cell}$ are characterized by the relation $\stackrel{LL_{body}}{\rightsquigarrow}$, defined in Figure 19. The only interesting rule, (APP-f_k), corresponds to the translation of an application of a known local procedure, where the additional parameters, X_k, are inserted into the call. The rule CONG-VAR rules out the special case which is handled by the (APP-f_k) rule. $\mathbb{F}$ is used not only to match the procedure identifiers f_k but also to specify the m_k's (the number of parameters for each local procedure).

We can prove that the $\stackrel{LL_{body}}{\rightsquigarrow}$ relation captures the effect of the LL_{body} transformation on the translations into the metalanguage. In the following, after stating that a Scheme expression e is lambda-liftable with respect to a list of variable lists $\mathbb{X}$, references to metavariables F, e_k, e' and so forth refer to the sub-parts of e as in Figure 18. Also, the $\subset$ operator denotes the subterm relation: $e' \subset e$ means that e' is a subterm of e.

Lemma 6.2.2 *If e is lambda-liftable with respect to $\mathbb{X}$, and if $e' \subset e$ such that $\Gamma \vdash [\![e']\!]_{cell}{:}\tau$, then*

$$\Gamma, \mathbb{X}, \mathbb{F} \vdash [\![e']\!]_{cell} \stackrel{LL_{body}}{\rightsquigarrow} [\![LL_{body}(F, \mathbb{X}, e')]\!]_{cell}{:}\tau$$

PROOF:

Structural induction on e'.

□

Given $\mathbb{X} = \langle X_1, \ldots, X_n \rangle$ and $\mathbb{F} = \langle (f_1, Y_1), \ldots, (f_n, Y_n) \rangle$, $\overset{LL_{body}}{\rightsquigarrow}$ is the least relation closed under the following rules:

$$\Gamma[x{:}\tau], \mathbb{X}, \mathbb{F} \vdash x \overset{LL_{body}}{\rightsquigarrow} x{:}\tau, \quad x \notin dom(\mathbb{F}) \quad (\textsc{cong-var})$$

$$\Gamma, \mathbb{X}, \mathbb{F} \vdash c \overset{LL_{body}}{\rightsquigarrow} c{:}\tau \qquad (\tau \text{ from Figure 1}) \quad (\textsc{cong-const})$$

$$\Gamma, \mathbb{X}, \mathbb{F} \vdash l^\tau \overset{LL_{body}}{\rightsquigarrow} l^\tau{:}\mathit{ref}(\tau) \quad (\textsc{cong-loc})$$

$$\frac{\Gamma, \mathbb{X}, \mathbb{F} \vdash M \overset{LL_{body}}{\rightsquigarrow} N{:}\tau' \to \tau \quad \Gamma \vdash M' \overset{LL_{body}}{\rightsquigarrow} N'{:}\tau'}{\Gamma, \mathbb{X}, \mathbb{F} \vdash (M\ M') \overset{LL_{body}}{\rightsquigarrow} (N\ N'){:}\tau} \quad (\textsc{cong-app})$$

$$\frac{\begin{array}{c}\Gamma, \mathbb{X}, \mathbb{F} \vdash M_i \overset{LL_{body}}{\rightsquigarrow} N_i{:}(\mathit{Val} \to pr) \to pr, \quad 1 \leq i \leq m_k \\ \Gamma, \mathbb{X}, \mathbb{F} \vdash M_\kappa \overset{LL_{body}}{\rightsquigarrow} N_\kappa{:}\,\mathit{Val} \to pr \\ M = \mathcal{A}[\![f_k, M_1, \ldots, M_{m_k}, M_\kappa]\!] \\ N = \mathcal{A}'[\![f_k, X_k, N_1, \ldots, N_{m_k}, N_\kappa]\!]\end{array}}{\Gamma, \mathbb{X}, \mathbb{F} \vdash M \overset{LL_{body}}{\rightsquigarrow} N{:}pr} \quad (\textsc{app-}f_k)$$

Figure 19: Definition of $\overset{LL_{body}}{\rightsquigarrow}$ on Terms

The relation $\overset{LL_{body}}{\rightsquigarrow}$ is closed under substitution by $\overset{LL_{body}}{\rightsquigarrow}$-related terms:

Lemma 6.2.3 *If* $\Gamma[x{:}\tau'], \mathbb{X}, \mathbb{F} \vdash M \overset{LL_{body}}{\rightsquigarrow} N{:}\tau$ *and if* $\Gamma, \mathbb{X}, \mathbb{F} \vdash M' \overset{LL_{body}}{\rightsquigarrow} N'{:}\tau'$, *then*

$$\Gamma, \mathbb{X}, \mathbb{F} \vdash M[M'/x] \overset{LL_{body}}{\rightsquigarrow} N[N'/x]{:}\tau$$

Proof:

Structural induction on M.

□

We are now ready to prove the correctness of the incremental lambda lifting transformation on Scheme$_{cell}$ terms. We have already started the process by appealing to the Extensionality Theorem 2.8.1 and reducing both sides in a state with a continuation of the correct type. We now construct a candidate simulation and show that the two sides always stay in the simulation relation (up to other simulation relations, as usual).

Figure 20 defines the candidate simulation S_{LL}. The substitution pairs (σ_u, σ'_u) and (σ_v, σ'_v) abstract out the "interesting" subterms using placeholders $u_1 \ldots u_{\#u}$ and $v_1 \ldots v_{\#v}$.

The terms substituted for the u_i variables correspond to **u**nevaluated translations of the (lambda (X_0) $\ldots$) expressions – on the left-hand side the local procedure bodies remain in the term component of the state, whereas on the right-hand side the procedure bodies have already been put in the store. The corresponding procedure bodies are related by $\overset{LL_{body}}{\rightsquigarrow}$.

The terms substituted for the v_i variables correspond to e**v**aluated $[\![$(lambda (X_0) $\ldots$)$]\!]_{cell}$ terms. For each side, this means an active substitution for the $x_i \in X_0$ parameters, and on the left-hand side, there is also a new set of bindings for newly-stored closures, $[l^i_k/f_k]^n_{k=1}$. Again, all of the corresponding procedure bodies are related by $\overset{LL_{body}}{\rightsquigarrow}$.

Lemma 6.2.5 S_{LL} *is a simulation.*

Proof:

Definition 6.2.4 (S_{LL})
Given $\mathbb{X} = \langle X_1, \ldots, X_n\rangle$ *and* $\mathbb{F} = \langle (f_1, Y_1), \ldots, (f_n, Y_n)\rangle$,

$$S_{LL} \stackrel{def}{=} \{(\langle\!\langle M\ ;\Sigma\rangle\!\rangle\sigma, \langle\!\langle M\ ;\Sigma'\rangle\!\rangle\sigma') \mid$$

$$\text{preamble}\left\{\begin{array}{l}
\exists u, v \in Var^* \text{ s.t.}\\
fv(M, rng(\Sigma), rng(\Sigma')) \subseteq u \cup v\\
\sigma = \sigma_u \cup \sigma_v\\
\sigma' = \sigma'_u \cup \sigma'_v\\
\exists \Sigma_0 \text{ s.t. } dom(\Sigma_0) \subseteq dom(\Sigma),\ \text{and}\ dom(\Sigma_0) \subseteq dom(\Sigma')\\
(\Gamma \vdash M{:}pr) \Rightarrow (\Gamma(u_i) = Val,\ \Gamma(v_i) = pr)
\end{array}\right.$$

$$\begin{array}{c}\text{unevaluated}\\ \mathsf{lambda}\ (X)\\ \text{closures}\end{array}\left\{\begin{array}{l}
\exists M^i_0, M^i_1, \ldots, M^i_n, N^i_0, N_1, \ldots, N_n, \ \text{ for } 1 \le i \le \#u \ \text{ s.t.}\\
\sigma_u = [\,u_i \mapsto (\mathcal{L}(X_0, \mathbb{F})[\![M^i_0, \ldots, M^i_n]\!])\,]^{\#u}_{i=1}\\
\sigma'_u = [u_i \mapsto (\mathcal{R}(X_0)[\![N^i_0]\!][l_k/f_k]^n_{k=1})\,]^{\#u}_{i=1}\\
\Sigma' = \Sigma_0[l_j \mapsto (\mathcal{C}'(X_j, Y_j)[\![N_j]\!][l_k/f_k]^n_{k=1})\,]^n_{j=1}\\
\left.\begin{array}{l}
\Gamma, \mathbb{X}, \mathbb{F} \vdash M^i_0 \stackrel{LL_{body}}{\leadsto} N^i_0{:}(Val{\to}pr){\to}pr\\
\Gamma, \mathbb{X}, \mathbb{F} \vdash M^i_k \stackrel{LL_{body}}{\leadsto} N_k{:}(Val{\to}pr){\to}pr
\end{array}\right\} \begin{array}{c}1 \le i \le \#u\\ 1 \le k \le n\end{array}
\end{array}\right.$$

$$\begin{array}{c}\text{evaluated}\\ \mathsf{lambda}\ (X)\\ \text{closures}\end{array}\left\{\begin{array}{l}
\exists M_i, M^i_{x_1}, \ldots, M^i_{x_{m_0}}, N_i, N^i_{x_1}, \ldots, N^i_{x_{m_0}}, \ \text{ for } 1 \le i \le \#v \ \text{ s.t.}\\
\sigma_v = [v_i \mapsto M_i[l^i_k/f_k]^n_{k=1}[M^i_{x_j}/x_j]^{m_0}_{j=1}\,]^{\#v}_{i=1}\\
\sigma'_v = [v_i \mapsto N_i[N^i_{x_j}/x_j]^{m_0}_{j=1}\,[l_k/f_k]^n_{k=1}\,]^{\#v}_{i=1}\\
\Sigma = \Sigma_0[l^i_h \mapsto (\mathcal{C}(Y_h)[\![M^i_h]\!][l_k/f_k]^n_{k=1}[M^i_{x_j}/x_j]^{m_0}_{j=1})\,]^{\#v,\,n}_{i=1,h=1}\\
\Gamma, \mathbb{X}, \mathbb{F} \vdash M_i \stackrel{LL_{body}}{\leadsto} N_i{:}pr,\ \text{for } 1 \le i \le \#v\\
\Gamma, \mathbb{X}, \mathbb{F} \vdash M^i_{x_j} \stackrel{LL_{body}}{\leadsto} N^i_{x_j}{:}Val,\ \text{for } 1 \le i \le \#v, 1 \le j \le m_0
\end{array}\right.$$

$$\}$$

Figure 20: The S_{LL} Candidate Simulation

$\langle 1 \rangle 1.$ $S_{LL} \subseteq [\lesssim_{sim} \circ S_{LL} \circ \lesssim_{sim}]_{sim}$

PROOF SKETCH: By induction on the number of steps in the leftmost reduction of the left-hand side to an I/O operator, by considering the possible forms of the term component M:

1. $M = (\mathsf{case}\, u_i\ (\lambda f.f\ M_Q\ M_K)\ M_{int})$
2. $M = v_i$
3. $M = (\mathsf{deref}\, M_1\ M_2)$
4. $M = (\mathsf{update}\, M_1\ M_2\ M_3)$
5. $M = (\mathsf{new}\, M_1\ M_2)$
6. $M = (\mathsf{io}\, M_1)$
7. $M = ((\lambda x.M_0)\ M_1 \ldots M_k)$
8. other PCF head redexes

The first two cases are the most interesting.

$\langle 2 \rangle 1.$ CASE: $M = (\mathsf{case}\, u_i\ (\lambda f.f\ M_Q\ M_K)\ M_{int})$

ASSUME: $\Gamma \vdash M_Q{:}\mathit{Val}^*,\ M_K{:}\mathit{Val}{\rightarrow}\mathit{pr},\ M_{int}{:}\mathit{int}{\rightarrow}\mathit{pr}$

$M_Q = \langle M_1, \ldots, M_{m_0} \rangle$

This case corresponds to the $(\mathsf{lambda}\ (X_0)\ \ldots)$ in question having been applied to a set of arguments, $e_1 \ldots e_{m_0}$. The translations of the argument expressions correspond to the M_j metalanguage terms, for $1 \leq j \leq m_0$. We can expect the left-hand side to store a new set of closures for its local procedures before evaluating its body, whereas the right-hand side gets right to its body:

$\langle 3 \rangle 1.$ Left-hand side reduces:

$$
\begin{aligned}
&\langle\!\langle(\mathsf{case}\, u_i\ (\lambda f.f\ M_Q\ M_K)\ M_{int})\ ;\ \Sigma\rangle\!\rangle\sigma \\
&\quad = \langle\!\langle(\mathsf{case}\ (\mathcal{L}(X_0,\mathbb{F})[\![M_0^i,\ldots,M_n^i]\!])(\lambda f.f\ M_Q\ M_K)\ M_{int}) \\
&\qquad\quad ;\ \Sigma\rangle\!\rangle\sigma \\
&\quad = \langle\!\langle(\mathsf{case}\ (\mathsf{inr}(\lambda x^*{:}\,Val^*.\lambda\kappa'{:}\,Val{\to}pr. \\
&\qquad\quad \mathsf{new}\,\Omega_{Val}(\lambda f_1{:}ref(Val).\cdots\mathsf{new}\,\Omega_{Val}(\lambda f_n{:}ref(Val). \\
&\qquad\qquad \mathsf{update}\, f_1\ \mathcal{C}(Y_1)[\![M_1^i]\!]\cdots\mathsf{update}\, f_n\ \mathcal{C}(Y_n)[\![M_n^i]\!] \\
&\qquad\qquad\quad (checkargs\ \ m_0\ x^*)(M_0^i\ \kappa'))\cdots)\sigma_0)) \\
&\qquad\qquad (\lambda f.\ f\ \langle M_1\ldots M_{m_0}\rangle\ M_K)\ M_{int}) \\
&\qquad\quad ;\ \Sigma\rangle\!\rangle\sigma \\
&\quad =_{pcf}\ \langle\!\langle(\mathsf{new}\,\Omega\ (\lambda f_1.\cdots\mathsf{new}\,\Omega\ (\lambda f_n. \\
&\qquad\quad \mathsf{update}\, f_1\ \mathcal{C}(Y_1)[\![(M_1^i\sigma)]\!]\cdots\mathsf{update}\, f_n\ \mathcal{C}(Y_n)[\![M_n^i\sigma]\!] \\
&\qquad\quad (M_0^i\ M_K)\sigma)\cdots)[(M_j\sigma)/x_j]_{j=1}^{m_0})\ ;\ \Sigma\rangle\!\rangle\sigma \\
&\quad \to^*_{left,sto}\ \langle\!\langle(M_0^i[(M_j\sigma)/x_j]_{j=1}^{m_0}\ [l_k^i/f_k]_{k=1}^n\ M_K)\sigma\ ; \\
&\qquad\qquad \Sigma[l_k^i\mapsto(\mathcal{C}(Y_k)[\![M_k^i]\!][(M_j\sigma)/x_j]_{j=1}^{m_0}\ [l_k^i/f_k]_{k=1}^n)\ ]_{k=1}^n\sigma\rangle\!\rangle \\
&\quad = \langle\!\langle(M_0^i[M_j/x_j]_{j=1}^{m_0}\ [l_k^i/f_k]_{k=1}^n\ M_K)\ ; \\
&\qquad\quad \Sigma[l_k^i\mapsto(\mathcal{C}(Y_k)[\![M_k^i]\!][M_j/x_j]_{j=1}^{m_0}\ [l_k^i/f_k]_{k=1}^n)\ ]_{k=1}^n\rangle\!\rangle\sigma \\
&\quad \stackrel{def}{=}\ \mathbf{lhs}'\ \stackrel{\leq n}{\to}_{left,sto}\ \langle\!\langle\mathsf{io}\, M'\ ;\ \Sigma'_M\rangle\!\rangle
\end{aligned}
$$

Note the substitution of M_j terms for x_j variables in the range of Σ. It is this substitution that is replaced by passing x_j arguments explicitly in the lambda-lifted term on the right-hand side.

⟨3⟩2. Right-hand side reduces:

$$
\begin{aligned}
&\langle\!\langle(\mathsf{case}\, u_i\ (\lambda f.f\ M_Q\ M_K)\ M_{int})\ ;\ \Sigma'\rangle\!\rangle\sigma' \\
&\quad = \langle\!\langle(\mathsf{case}(\mathcal{R}(X_0)[\![N_0^i]\!][l_i/f_i]_{i=1}^n)\ (\lambda f.f\ M_Q\ M_K)\ M_{int}) \\
&\qquad\quad ;\ \Sigma'\rangle\!\rangle\sigma' \\
&\quad = \langle\!\langle(\mathsf{case}(\mathsf{inr}(\lambda x^*{:}\,Val^*.\lambda\kappa'{:}\,Val{\to}pr.(checkargs\ m_0\ x^*)(N_0^i\ \kappa')\sigma_0)[l_i/f_i]_{i=1}^n) \\
&\qquad\quad (\lambda f.f\ \langle M_1\ldots M_{m_0}\rangle\ M_K)\ M_{int}) \\
&\qquad\quad ;\ \Sigma'\rangle\!\rangle\sigma' \\
&\quad =_{pcf}\ \langle\!\langle(N_0^i\ M_K)[(M_j\sigma')/x_j]_{j=1}^{m_0}[l_i/f_i]_{i=1}^n\ ;\ \Sigma'\rangle\!\rangle\sigma' \\
&\quad = \langle\!\langle(N_0^i[M_j/x_j]_{j=1}^{m_0}[l_i/f_i]_{i=1}^n\ M_K)\ ;\ \Sigma'\rangle\!\rangle\sigma' \\
&\quad \stackrel{def}{=}\ \mathbf{rhs}'
\end{aligned}
$$

$\langle 3\rangle$3. Q.E.D.

To get $(\mathbf{lhs}', \mathbf{rhs}')$ into S_{LL}, we have to add new entries to the σ_v and σ_v' substitutions (which track of the applications of the closures for the outer $(\mathsf{lambda}\ (X_0)\ldots)$).

We use fresh variable $v_{\#v+1}$, and note that no x_j is in the domain of σ or σ', so we have:

$$
\begin{aligned}
\mathbf{lhs}' =& \langle\!\langle(v_{\#v+1}\ M_K)\ ;\ \ldots\rangle\!\rangle \\
& \quad \sigma[v_{\#v+1} \mapsto M_0^i[M_j/x_j]_{j=1}^{m_0}[l_k^i/f_k]_{k=1}^n]
\end{aligned}
$$

and similarly with the right-hand side. The two sides are easily seen to be in S_{LL}, with augmented σ_v and σ_v''s, because we assume that $\Gamma, \mathbb{X}, \mathbb{F} \vdash M_0^i \stackrel{LL_{body}}{\rightsquigarrow} N_0^i{:}(Val{\to}pr){\to}pr$, and Lemma 6.2.3 allows us to substitute $\stackrel{LL_{body}}{\rightsquigarrow}$-related terms and remain $\stackrel{LL_{body}}{\rightsquigarrow}$-related.

$\langle 2\rangle$2. Case: $M = v_i$

Assume:
$$
\begin{aligned}
&\wedge\ \sigma_v(v_i) = M_i[l_k^i/f_k]_{k=1}^n[M_{x_j}^i/x_j]_{j=1}^{m_0} \\
&\wedge\ \sigma_v'(v_i) = N_i[N_{x_j}^i/x_j]_{j=1}^{m_0}\,[l_i/f_i]_{i=1}^n \\
&\wedge\ \Gamma, \mathbb{X}, \mathbb{F} \vdash M_i \stackrel{LL_{body}}{\rightsquigarrow} N_i{:}pr, \\
&\wedge\ \Sigma(l_k^i) = (\mathcal{C}(Y_k)[\![M_k^i]\!][l_k^i/f_k]_{k=1}^n[M_{x_j}^i/x_j]_{j=1}^{m_0}),\ \text{for } 1 \le k \le n \\
&\wedge\ \Gamma, \mathbb{X}, \mathbb{F} \vdash M_{x_j}^i \stackrel{LL_{body}}{\rightsquigarrow} N_{x_j}^i{:}\,Val,\ \text{for } 1 \le j \le m_0
\end{aligned}
$$

For this substep, we use induction on the derivation of $\Gamma, \mathbb{X}, \mathbb{F} \vdash M_i \stackrel{LL_{body}}{\rightsquigarrow} N_i{:}pr$. The only interesting case is when the last step in the derivation is a use of the (app-f_k) rule:

⟨3⟩1. CASE: $M_i = \mathcal{A}[\![f_k, M_{y_1}, \ldots, M_{y_{m_k}}, M_\kappa]\!]$

ASSUME: $\wedge\ N_i = \mathcal{A}'[\![f_k, X_k, N_{y_1}, \ldots, N_{y_{m_k}}, N_\kappa]\!]$

$\wedge\ \Gamma, \mathbb{X}, \mathbb{F} \vdash M_{y_j} \overset{LL_{body}}{\rightsquigarrow} N_{y_j}{:}(\mathit{Val}{\to}pr){\to}pr,\ 1 \le j \le m_k$

$\wedge\ \Gamma, \mathbb{X}, \mathbb{F} \vdash M_\kappa \overset{LL_{body}}{\rightsquigarrow} N_\kappa{:}\mathit{Val}{\to}pr$

⟨4⟩1. Left-hand side reduces:

$$
\begin{aligned}
&\langle\!\langle (\mathsf{deref}\, l_k^i\ (\lambda v_{f_k}{:}\mathit{Val}.M_{y_1}\ (\lambda v_1{:}\mathit{Val}.\ldots M_{y_{m_k}}\ (\lambda v_{m_k}{:}\mathit{Val}.\\
&\quad out_F\ v_{f_k}\ (\lambda f'{:}\mathit{Val}^*{\to}(\mathit{Val}{\to}pr){\to}pr.\\
&\qquad f'\ \langle v_1 \ldots v_{m_k}\rangle\ M_\kappa)\ldots)))[l_k^i/f_k]_{k=1}^n[M_{x_j}^i/x_j]_{j=1}^{m_0})\ ;\ \Sigma\rangle\!\rangle\sigma\\
&\overset{\ge 0}{\to}_{left,sto}\ \langle\!\langle M_{y_1}\ (\lambda v_1.\ldots M_{y_{m_k}}\ (\lambda v_{m_k}.\\
&\quad out_F(\mathcal{C}(Y_k)[\![M_k^i]\!][l_k^i/f_k]_{k=1}^n[M_{x_j}^i/x_j]_{j=1}^{m_0})\\
&\quad (\lambda f'.f'\ \langle v_1 \ldots v_{m_k}\rangle\ M_\kappa)\ldots))\\
&\qquad [l_k^i/f_k]_{k=1}^n[M_{x_j}^i/x_j]_{j=1}^{m_0}\ ;\ \Sigma\rangle\!\rangle\sigma\\
&=_{pcf}\ \langle\!\langle M_1\ (\lambda v_1.\ldots M_{m_k}\ (\lambda v_{m_k}.\\
&\quad (\lambda y^*{:}\mathit{Val}^*.\lambda\kappa'{:}\mathit{Val}{\to}pr.(checkargs\ \ m_k\ y^*)(M_k^i[l_k^i/f_k]_{k=1}^n[M_{x_j}^i/x_j]_{j=1}^{m_0})\,\sigma_k\ \kappa')\\
&\qquad \langle v_1 \ldots v_{m_k}\rangle\ M_\kappa)\ldots)\\
&\qquad [l_k^i/f_k]_{k=1}^n[M_{x_j}^i/x_j]_{j=1}^{m_0}\ ;\ \Sigma\rangle\!\rangle\sigma\\
&=_{pcf}\ \langle\!\langle M_1\ (\lambda y_1.\ldots M_{m_k}\ (\lambda y_{m_k}.\\
&\quad (M_k^i[l_k^i/f_k]_{k=1}^n[M_{x_j}^i/x_j]_{j=1}^{m_0})\ M_\kappa)\cdots)\\
&\qquad [l_k^i/f_k]_{k=1}^n[M_{x_j}^i/x_j]_{j=1}^{m_0}\ ;\ \Sigma\rangle\!\rangle\sigma\\
&= \langle\!\langle M_1\ (\lambda y_1.\ldots M_{m_k}\ (\lambda y_{m_k}.\\
&\quad (M_k^i[l_k^i/f_k]_{k=1}^n[M_{x_j}^i/x_j]_{j=1}^{m_0})\ M_\kappa)\cdots)\ ;\ \Sigma\rangle\!\rangle\sigma\\
&\overset{def}{=}\ \mathbf{lhs}'\ \overset{\le n}{\to}_{left,sto}\ \langle\!\langle \mathsf{io}\, M'\ ;\ \Sigma_M'\rangle\!\rangle
\end{aligned}
$$

⟨4⟩2. Right-hand side reduces:

$$
\begin{aligned}
&= \langle\!\langle(\mathsf{deref}\, l_k\ (\lambda v_{f_k}\!:\!Val.N_{y_1}\ (\lambda v_1\!:\!Val.\ldots N_{y_{m_k}}\ (\lambda v_{m_k}\!:\!Val.\\
&\qquad out_F\ v_{f_k}\ (\lambda f'\!:\!Val^*\!\to\!(Val\!\to\!pr)\!\to\!pr.\\
&\qquad\quad f'\ \langle[x_i]_{\in X_k}\ v_1\ldots v_{m_k}\rangle\ N_\kappa)\ldots))))[N^i_{x_j}/x_j]^{m_0}_{j=1}\ ;\ \Sigma'\rangle\!\rangle\sigma'\\
&\to^*_{sto} \langle\!\langle N_{y_1}\ (\lambda v_1.\ldots N_{y_{m_k}}\ (\lambda v_{m_k}.\\
&\qquad out_F\ \mathcal{C}'(X_k,Y_k)[\![N_k]\!][l_i/f_i]^n_{i=1}\\
&\qquad\quad (\lambda f'.f'\ \langle[x_i]_{\in X_k}\ v_1\ldots v_{m_k}\rangle\ N_\kappa)\ldots))[N^i_{x_j}/x_j]^{m_0}_{j=1}\ ;\ \Sigma'\rangle\!\rangle\sigma'\\
&=_{pcf}\ \langle\!\langle N_1\ (\lambda v_1.\ldots N_{m_k}\ (\lambda v_{m_k}.\\
&\qquad (\lambda y^*\!:\!Val^*.\lambda\kappa'\!:\!Val\!\to\!pr.(checkargs\ \ m_k\ y^*)N_k\sigma'_k\ \kappa')[l_i/f_i]^n_{i=1}\\
&\qquad\quad \langle[x_i]_{\in X_k}\ v_1\ldots v_{m_k}\rangle\ N_\kappa)\ldots)[N^i_{x_j}/x_j]^{m_0}_{j=1}\ ;\ \Sigma'\rangle\!\rangle\sigma'\\
&=_{pcf}\ \langle\!\langle N_1\ (\lambda y_1.\ldots N_{m_k}\ (\lambda y_{m_k}.\\
&\qquad N_k\ N_\kappa)\ldots)[l_i/f_i]^n_{i=1}[N^i_{x_j}/x_j]_{x_j\in X_k}\ ;\ \Sigma'\rangle\!\rangle\sigma'\\
&\stackrel{def}{=} \mathbf{rhs}'
\end{aligned}
$$

The last $=_{pcf}$ equality depends on the fact that the vector of values passed in matches the σ'_k substitution applied to the body of the closure. In particular, the first $\#X_k$ elements of the value vector correspond to the x_j's that were added by lambda-lifting, and the values given by $[N^i_{x_j}/x_j]_{x_j\in X_k}$ make their way into the closure.

⟨4⟩3. Q.E.D.

$(\mathbf{lhs}',\mathbf{rhs}')\in S_{LL}$ by adding elements to σ_v and σ'_v. To make this clear, we rewrite $\mathbf{lhs}'$ as follows:

$$
\begin{aligned}
\mathbf{lhs}' &= \langle\!\langle M_1\ (\lambda y_1.\ldots M_{m_k}\ (\lambda y_{m_k}.\\
&\qquad (v_{\#v+1}\ M_\kappa)\cdots)\ ;\ \Sigma\rangle\!\rangle\sigma''
\end{aligned}
$$

Where $\sigma''=\sigma[v_{\#v+1}\mapsto(M^i_k[l^i_k/f_k]^n_{k=1}[M^i_{x_j}/x_j]^{m_0}_{j=1})]$

and likewise for the right-hand side.

⟨2⟩3. All other cases

Are easy, as the locations involved will never be the crucial l_k constants.

□

The correctness result we want follows easily:

Theorem 6.2.6 (Correctness of Incremental Lambda Lifting)

For any $Scheme_{cell}$ expression e and list of lists of variables $\mathbb{X}$ such that e is lambda-liftable with respect to $\mathbb{X}$, and any Γ mapping the free variables of e to $ref(Val)$, if $\Gamma \vdash_{cell} e{:}ok$, then

$$\Gamma \vdash [\![e]\!]_{cell} \cong_{\forall C} [\![LL(\mathbb{X}, e)]\!]_{cell}{:}(Val \rightarrow pr) \rightarrow pr$$

PROOF:

1. By the Extensionality Theorem, Theorem 2.8.1, and the definition of $\cong_{ext}$, it suffices to show that for all closing substitutions σ mapping free variables to closed terms of type $ref(Val)$, all closed terms K of type $Val \rightarrow pr$, and all stores Σ,
$$\langle\!\langle ([\![e]\!]_{cell}\sigma)\ K\ ;\ \Sigma\rangle\!\rangle \approx_{sim} \langle\!\langle ([\![LL(\mathbb{X}, e)]\!]_{cell}\sigma)\ K\ ;\ \Sigma\rangle\!\rangle$$
2. It is easy to show that for all K and Σ,
$$(\langle\!\langle ([\![e]\!]_{cell}\sigma)\ K\ ;\ \Sigma\rangle\!\rangle, \langle\!\langle ([\![LL(\mathbb{X}, e)]\!]_{cell}\sigma)\ K\ ;\ \Sigma\rangle\!\rangle) \in S_{LL}$$
as S_{LL} was defined exactly for this purpose.
3. By Lemma 6.2.5 and steps 1 and 2, the correctness result follows.

□

Chapter 7

Conclusions and Open Problems

In [RP95], Ritter and Pitts write:

> It remains an open problem to find a co-inductive characterization of observational equivalence for languages like SML that combine higher order functions and local state.

We have solved this open problem with the metalanguage defined in this thesis. As such, the results obtained so far are of independent interest. Additionally, we have applied the metalanguage and its theory to substantial applications from an actual compiler for the programming language Scheme, producing the first known proofs of correctness of two subtle transformations.

7.1 Open Problems – Future Work

The actual proofs given in Chapters 5 and 6 might appear somewhat inscrutable to some readers[1]. It has been our experience that, despite the syntactic complexity of the proofs, the process of (1) devising an appropriate candidate simulation relation, and (2) doing the ensuing coinductive proof, is fairly predictable. As such, it seems that the methods we use

[1]or, more likely, complete gibberish to all readers

might be amenable to automation. With a good theorem prover and a well-developed set of examples and previous proofs, it may be possible to hide much of the detail and make proofs of correctness such as those presented in this thesis accessible to a wider audience of programmers and compiler writers.

Similarly, we believe that some of the predictability and repetitiveness of the proofs can be encompassed by a more robust metatheory for the metalanguage.

It would be interesting to add polymorphism to the type system in order to apply our methods to languages such as ML and object-oriented languages. A first-order exploration of bisimilarity in an object calculus is presented in [GR96].

Generalizing our method to be parameterized by the notion of observation, and extending the framework to handle concurrency, would also be fruitful areas of research. A starting point for this avenue of research is [San94].

Bibliography

[Abr90] Samson Abramsky. The lazy lambda calculus. In David A. Turner, editor, *Research Topics in Functional Programming*, pages 65–116. Addison-Wesley, 1990.

[App92] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, Cambridge, 1992.

[Bar81] Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* North-Holland, Amsterdam, 1981.

[CH94] William D. Clinger and Lars Thomas Hansen. Lambda, the ultimate label, or a simple optimizing compiler for scheme. In *Proc. 1994 ACM Symposium on Lisp and Functional Programming*, pages 128–139, 1994.

[Cli84] William Clinger. The scheme 311 compiler: An exercise in denotational semantics. In *Proc. 1984 ACM Symposium on Lisp and Functional Programming*, pages 356–364, August 1984.

[CR91] William Clinger and Jonathan A. Rees. The revised^4 report on the algorithmic language scheme. *ACM LISP Pointers*, 4(3), 1991.

[EHR91] L. Egidi, F. Honsell, and S. Ronchi della Rocca. The lazy call-by-value λ-calculus. In *proceedings of MFCS '91 Lecture Notes in Computer Science*, 1991.

[FF86] M. Felleisen and D. Friedman. Control operators, the SECD machine and the λ-calculus. In *Formal Description of Programming Concepts III*, pages 131–141. North-Holland, 1986.

[FH92] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103:235–271, 1992.

[FWH92] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages*. MIT Press, Cambridge, MA, 1992. Also McGraw-Hill, Chicago, 1992.

[Gor94a] Andrew Gordon. A tutorial on co-induction and functional programming. In *Proceedings of the 1994 Glasgow Workshop on Functional Programming*, pages 78–95. Springer Workshops in Computing, 1994.

[Gor94b] Andrew D. Gordon. *Functional Programming and Input/Output*. Cambridge University Press, Cambridge, 1994.

[Gor95] Andrew D. Gordon. Bisimilarity as a theory of functional programming. In *Proceedings of 11th Conference on Mathematical Foundations of Programming Semantics*, 1995.

[GR96] Andrew D. Gordon and Gareth D. Rees. Bisimilarity for a first-order calculus of objects with subtyping. In *Conf. Rec. 23rd ACM Symposium on Principles of Programming Languages*, pages 386–395, 1996.

[GSR95] J. D. Guttman, V. Swarup, and J. Ramsdell. The VLISP verified scheme system. *Lisp and Symbolic Computation*, 8(1/2):33–110, 1995.

[Gun92] Carl A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. MIT Press, Cambridge, MA, 1992.

[HM85] Matthew Hennessy and Robin Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32:137–161, 1985.

[HMST95] Furio Honsell, Ian A. Mason, Scott Smith, and Carolyn Talcott. A variable typed logic of effects. *Information and Computation*, 119:55–90, 1995.

[How89] Douglas J. Howe. Equality in lazy computation systems. In *Proc. 4th IEEE Symposium on Logic in Computer Science*, pages 198–203, 1989.

[How95] Douglas J. Howe. A note on proving congruence of bisimulation in a generalized lambda calculus. unpublished manuscript, 1995.

[How96] Douglas J. Howe. Proving congruence of bisimulation in functional programming languages. to appear in *Information and Computation*, 1996.

[IEE91] IEEE Computer Society, New York. *IEEE Standard for the Scheme Programming Language*, IEEE standard 1178-1990 edition, 1991.

[JM91] Trevor Jim and Albert R. Meyer. Full abstraction and the context lemma. Technical Report MIT/LCS/TR-524, MIT Laboratory for Computer Science, December 1991.

[KH89] Richard Kelsey and Paul Hudak. Realistic compilation by program transformation. In *Conf. Rec. 16th ACM Symposium on Principles of Programming Languages*, pages 281–292, 1989.

[KKR+86] David A. Kranz, Richard Kelsey, Jonathan A. Rees, Paul Hudak, James Philbin, and Norman I. Adams. Orbit: An optimizing compiler for scheme. In *Proceedings SIGPLAN '86 Symposium on Compiler Construction*, 1986. *SIGPLAN Notices 21*(7), July, 1986, 219-223.

[Lam93] Leslie Lamport. How to write a proof. Technical Report SRC-094, Digital Systems Research Center, feb 1993. Available by FTP from `gatekeeper.dec.com` in the directory `/archive/pub/DEC/SRC/research-reports`. It is file `SRC-094.ps.Z`.

[Mil77] Robin Milner. Fully abstract models of typed lambda-calculi. *Theoretical Computer Science*, 4:1–22, 1977.

[Mil90] Robin Milner. Operational and algebraic semantics of concurrent processes. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 1201–1242. MIT Press/Elsevier, 1990.

[Mit96] John C. Mitchell. *Foundations for Programming Languages.* MIT Press, Cambridge, MA, 1996.

[Mor68] James H. Morris, Jr. *Lambda Calculus Models of Programming Languages.* PhD thesis, MIT, Cambridge, MA, 1968.

[MS88] Albert R. Meyer and Kurt Sieber. Towards fully abstract semantics for local variables: Preliminary report. In *Conf. Rec. 15th ACM Symposium on Principles of Programming Languages*, pages 191–203, 1988.

[MT91] Ian A. Mason and Carolyn L. Talcott. Equivalence in functional languages with effects. *Journal of Functional Programming*, 1:287–327, 1991.

[MTH89] Robin Milner, Mads Tofte, and Robert Harper. *The Definitiion of Standard ML.* MIT Press, Cambridge, MA, 1989.

[Ode94] Martin Odersky. A functional theory of local names. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 48–59, 1994.

[ORW95] Dino P. Oliva, John D. Ramsdell, and Mitchell Wand. The VLISP verified prescheme compiler. *Lisp and Symbolic Computation*, 8(1/2):111–182, 1995.

[OT95] P. W. O'Hearn and R. D. Tennent. Parametricity and local variables. *Journal of the ACM*, 42(3):658–709, May 1995.

[Pit99] A. M. Pitts. Operationally-based theories of program equivalence. In P. Dybjer and A. M. Pitts, editors, *Semantics and Logics of Computation.* Cambridge University Press, 199? Based on lectures given at the CLICS-II Summer School on Semantics and Logics of Computation, Isaac Newton Institute for Mathematical Sciences, Cambridge UK, September 1995.

[PJ87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall International, 1987.

[PJW93] Simon L. Peyton Jones and Philip L. Wadler. Imperative functional programming. In *Proceedings 20th Annual ACM Symposium on Programming Languages*, pages 71–84, 1993.

[Plo75] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[Plo77] Gordon D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

[PS93] A. M. Pitts and I. D. B. Stark. Observable properties of higher order functions that dynamically create local names, or: What's new? In *Mathematical Foundations of Computer Science, Proc. 18th Int. Symp., Gdańsk, 1993*, volume 711 of *Lecture Notes in Computer Science*, pages 122–141. Springer-Verlag, Berlin, 1993.

[Ram96] John D. Ramsdell. personal communication, July 1996.

[Rey81] John C. Reynolds. The essence of Algol. In J. W. deBakker and J. C. van Vliet, editors, *Algorithmic Languages*, pages 345–372. North-Holland, Amsterdam, 1981.

[Rey83] John C. Reynolds. Types, abstractions, and parametric polymorphism. In *Proceedings IFIP 83*, 1983.

[RP95] E. Ritter and A. M. Pitts. A fully abstract translation between a λ-calculus with reference types and Standard ML. In *2nd Int. Conf. on Typed Lambda Calculus and Applications, Edinburgh, 1995*, volume 902 of *Lecture Notes in Computer Science*, pages 397–413. Springer-Verlag, Berlin, 1995.

[RV95] Jon G. Riecke and Ramesh Viswanathan. Isolating side effects in sequential languages. In *Conf. Rec. 22nd ACM Symposium on Principles of Programming Languages*, pages 1–12, 1995.

[San94] Davide Sangiorgi. The lazy lambda calculus in a concurrency scenario. *Information and Computation*, 111(1):120–153, May 1994.

[San96] David Sands. Total correctness by local improvement in the transformation of functional programs. *ACM Transactions on Programming Languages and Systems*, 18:175–234, march 1996.

[SRI91] Vipin Swarup, Uday S. Reddy, and Evan Ireland. Assignments for applicative languages. *Lecture Notes in Computer Science*, 523:192–??, 1991.

[Sta94] Ian Stark. *Names and Higher-Order Functions.* PhD thesis, University of Cambridge, December 1994. Also published as Technical Report 363, University of Cambridge Computer Laboratory.

[Sto77] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, Cambridge, MA, 1977.

[TW96] Jerzy Tiuryn and Mitchell Wand. Untyped lambda-calculus with input-output. In H. Kirchner, editor, *Trees in Algebra and Programming: CAAP'96, Proc. 21st International Colloquium*, volume 1059 of *Lecture Notes in Computer Science*, pages 317–329, Berlin, Heidelberg, and New York, April 1996. Springer-Verlag.

[Wad92] Philip Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2(4):461–493, December 1992.

[Wan90] Mitchell Wand. A short proof of the lexical addressing algorithm. *Information Processing Letters*, 35:1–5, 1990.

[Win93] Glynn Winskel. *The Formal Semantics of Programming Languages.* MIT Press, Cambridge, MA, 1993.

[WO92] Mitchell Wand and Dino P. Oliva. Proving the correctness of storage representations. In *Proc. 1992 ACM Symposium on Lisp and Functional Programming*, pages 151–160, 1992.