An Extensional Metalanguage with I/O and a Dynamic Store (Extended Abstract)

Mitchell Wand   Gregory T. Sullivan*
College of Computer Science
Northeastern University
360 Huntington Avenue, 161CN
Boston, MA 02115, USA
voice: (617) 373-2462, fax: (617) 373-5121
{wand,gregs}@ccs.neu.edu
http://www.ccs.neu.edu/home/{wand,gregs}

December 12, 1995

Abstract

We introduce a typed metalanguage based on PCF that includes I/O and store operations. We give an applicative characterization of operational equivalence for this language. Finally, we give an example of a translation from an imperative programming language into our metalanguage, and we demonstrate how to use the theory of the metalanguage to reason about terms in the source programming language.

1 Introduction

There are a variety of reasons for and approaches to formally proving the equivalence of two program fragments. We are interested in proving compiler optimizations correct, and our research has the following requirements:

1. We are interested in proving optimizations for higher-order languages such as ML and Scheme which have imperative features such as destructive update.
2. We would like a framework for reasoning about programs with possibly infinite meaningful behavior, such as I/O or lazy lists. In particular, we cannot simply equate all nonterminating programs.

*Work supported by the National Science Foundation under grants number CCR-9304144 and CCR-9404646.

3. The optimizations we are interested in may alter the runtime storage behavior of the compiled programs, but of course they cannot alter the runtime I/O behavior.

4. We are interested in local, rather than global, optimizations. Therefore the notion of equivalence we are interested in is operational, or contextual, equivalence. Two terms are operationally equivalent iff they have the same behavior in all possible program contexts.

There are three major approaches to proving operational equivalence between programs, of which we have chosen the third:

Denotational: The denotational approach, exemplified by [MS88, HMT84], is arguably the most elegant but has traditionally had trouble with imperative features. While fully abstract models for store semantics exist, they are difficult to construct and reason within. The denotational line of research, and the paper by Meyer and Sieber in particular, has been successful in exploring and elucidating the issues inherent in reasoning about imperative programs.

Operational: Reasoning directly within the operational semantics of a language avoids some of the mathematical complexity of the denotational approach, at the cost of flexibility and generality. In [HMST95], the authors prove a *ciu theorem* for a language with a dynamic store. The ciu theorem states that equivalence in *reduction contexts* coincides with operational equivalence. On the other hand, in [PS93] the authors show that a simple functional language extended with the most basic notion of dynamic state – even without side-effects – lacks extensionality. One drawback of operational methods is that the theory is usually quite syntax-dependent, thus being overly sensitive to changes or additions to the language, and it is not clear that existing methods would scale up to handle "real" imperative programming languages.

Metalanguage: Using a metalanguage is a hybrid approach. The idea is to provide a compositional (syntax-directed) translation from the source language into the metalanguage. One then is able to use the presumably robust theory of the metalanguage to prove equivalence of the translated terms and, by compositionality, of the original terms. The canonical metalanguage is the lambda calculus, which provides the syntactic basis for the denotational approach. In [San94, Wal91], the π-calculus is used as a metalanguage, but this seems too general for compiler optimizations, hiding the sequential structure of source programs. We base our metalanguage on PCF and we use operational techniques to prove properties of the metalanguage. The results are not affected by changes in the source programming language, as only the translation to the metalanguage needs to be changed.

The metalanguage we present in this paper has the following features:

Simple Types: We use PCF [Plo77] as our base language. We discuss in Section 4 why we find types useful.

Dynamic Store: We add to PCF operators for allocating new store locations, dereferencing locations, comparing locations, and updating the store.

Input/Output: There are operations for reading and writing integers to/from some standard output, in the style of [Gor94].

Continuation Passing Style (CPS): The I/O and store operators are in CPS, and the metalanguage is designed to be used as the target of a CPS translation from the source language.

Lazy Evaluation, Internal Reduction: Reasoning within subterms, using the equational theory of pure PCF, preserves operational equivalence in our metalanguage. This improves upon the system in [HMST95], which allows only the weaker β-value rule. Thus we can use the full power of β-conversion to reason about metalanguage terms.

Applicative Extensionality: We show that metalanguage terms are equal in all contexts iff they are equal in all applicative contexts. The extensionality of the metalanguage substantially reduces the burden of proving operational equivalence.

It is initially surprising that a language supporting a dynamic store can also enjoy applicative extensionality. For example, the functions f and g,

$$f(x) = \{x := !x + 1; 2\}, \qquad g(x) = \{2\},$$

where $!x$ dereferences x and $x := !x + 1$ increments x, are both extensionally equal to the integer 2. However, a "non-applicative" program context such as

$$C[] \stackrel{\text{def}}{=} new(z = 0)\{[](z); !z\},$$

which allocates a new reference variable z, initializes z to 0, invokes some function on z and finally returns the value stored at z, will distinguish f and g (assuming call-by-reference): $C[f]$ returns 1 whereas $C[g]$ returns 0.

Our goal has been to provide a suitable framework for proving the correctness of compiler optimizations for imperative interactive languages, and we believe that this metalanguage with its theory provides such a platform. In addition to providing a framework for compiler optimization, the results in this paper answer the challenge, posed in [RP95], of giving an applicative characterization of operational equivalence in a language with a dynamic store.

In Sections 2.1 and 2.2, we give the syntax and semantics of the metalanguage. In Section 2.3 we define simulation between states and applicative and operational simulation between terms. In Section 2.4 we present the main theoretical result of this paper, namely the context lemma for the metalanguage, and we outline the most important proofs. In Sections 3.1 and 3.2, we define a simple imperative source language and give a CPS translation from that language into our metalanguage. In Section 3.3, we give a short example of using the CPS translation to prove equivalence of source program fragments. We conclude in Section 4 with a summary of our work.

In Appendix A, we give the typing and operational semantics of the PCF fragment of our metalanguage, which are standard. In Appendix B, we list some other useful properties of the metalanguage, which are referenced in some of the earlier proof outlines.

2 The Metalanguage

We extend the simply-typed, higher-order functional language PCF [Plo77] with CPS I/O and dynamic store operators. The typing rules and the operational semantics of the pure PCF fragment of our metalanguage are standard and given in the Appendix.

2.1 Syntax of the Metalanguage

The types in our metalanguage, ranged over by τ, are generated by the following grammar:

$$\tau ::= i \mid b \mid pr \mid \mathit{ref}(\tau) \mid \tau \rightarrow \tau$$

with i for integers, b for booleans, $\mathit{ref}(\tau)$ for locations with contents of type τ, and pr for processes, which may do I/O.

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

Constants c are enumerated and typed by the set *Const*:

$$\begin{aligned} \mathit{Const} \stackrel{\text{def}}{=} \{ & \mathsf{pred}{:}i{\rightarrow}i,\ \mathsf{succ}{:}i{\rightarrow}i,\ \mathsf{true}{:}b,\ \mathsf{false}{:}b,\ \mathsf{if}^{\tau}{:}b{\rightarrow}\tau{\rightarrow}\tau{\rightarrow}\tau,\ \mathsf{zero?}{:}i{\rightarrow}b, \\ & \mathsf{Y}^{\tau}{:}(\tau{\rightarrow}\tau){\rightarrow}\tau,\ \mathsf{write}{:}i{\rightarrow}pr{\rightarrow}pr,\ \mathsf{read}{:}(i{\rightarrow}pr){\rightarrow}pr,\ \mathsf{nil}{:}pr,\ \mathsf{O}{:}pr, \\ & \mathsf{new}^{\tau}{:}\tau{\rightarrow}(\mathit{ref}(\tau){\rightarrow}pr){\rightarrow}pr,\ \mathsf{get}^{\tau}{:}\mathit{ref}(\tau){\rightarrow}(\tau{\rightarrow}pr){\rightarrow}pr, \\ & \mathsf{set!}^{\tau}{:}\mathit{ref}(\tau){\rightarrow}\tau{\rightarrow}pr{\rightarrow}pr,\ \mathsf{eq?}^{\tau}{:}\mathit{ref}(\tau){\rightarrow}\mathit{ref}(\tau){\rightarrow}b\} \end{aligned}$$

A *typing judgement* of the form $\Gamma \vdash M{:}\tau$ can be read as saying that type environment Γ, which maps the free variables of M to types, entails that term M has type τ. 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 $\vdash P_i{:}\tau_i$. Thus a closing substitution σ instantiates all free variables: $\mathit{fv}(M\sigma) = \emptyset$.

Contexts are generated by the same grammar as terms, but with an additional production for a typed hole, i.e., $C ::= [-_{\tau}] \mid x \mid n \mid c \mid l^{\tau} \mid \lambda x{:}\tau.C \mid CC$. A context $C[-_{\tau}]$ instantiated with a term M, denoted $C[M]$, *respects* a type assignment $\Gamma \vdash M{:}\tau$ if for every variable $x \in \mathit{fv}(M)$ captured at a hole in C, if the type of x at the hole is τ', then $\Gamma(x) = \tau'$.

A *state*, denoted $[M, \Sigma]$, is a term-store pair where M is a closed term of type pr and store Σ is a finite map from locations to closed terms. A state must be well formed in that all locations l^{τ} in M must be in the domain of Σ and 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 is given in the Appendix and defines

$$
\begin{array}{ll}
M \rightarrow_{pcf} M' \Rightarrow [M,\, \Sigma] \rightarrow_\Sigma [M',\, \Sigma] & \\
[\mathsf{new}^\tau\, MN,\, \Sigma] \rightarrow_\Sigma [N l^\tau,\, \Sigma[l^\tau \mapsto M]] & \text{if } l^\tau \notin dom(\Sigma) \cup locs(rng(\Sigma), M, N) \\
[\mathsf{get}^\tau\, l^\tau M,\, \Sigma] \rightarrow_\Sigma [MN,\, \Sigma] & \text{if } \Sigma(l^\tau) = N \\
[\mathsf{set!}^\tau\, l^\tau MN,\, \Sigma] \rightarrow_\Sigma [N,\, \Sigma[l^\tau \mapsto M]] & \text{if } l^\tau \in dom(\Sigma)
\end{array}
$$

Table 1: Operational Semantics for Store Operators

$$
\frac{[M,\, \Sigma] \rightarrow^*_\Sigma [\mathsf{write}\, n M',\, \Sigma']}{[M,\, \Sigma] \xrightarrow{!n} [M',\, \Sigma']}
\qquad
\frac{[M,\, \Sigma] \rightarrow^*_\Sigma [\mathsf{read}\, M',\, \Sigma']}{[M,\, \Sigma] \xrightarrow{?n} [M'\, n,\, \Sigma']}
\qquad
\frac{[M,\, \Sigma] \rightarrow^*_\Sigma [\mathsf{nil},\, \Sigma']}{[M,\, \Sigma] \xrightarrow{\surd} [\mathsf{0},\, \Sigma']}
$$

Table 2: Semantics for I/O Operators

a relation $\rightarrow_{pcf}$ between closed well-typed terms. The reduction relation $\rightarrow_\Sigma$ 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, $\xrightarrow{\alpha}$, between states which reduce under $\rightarrow^*_\Sigma$ to states headed by I/O operations.

Table 1 defines the transition relation $\rightarrow_\Sigma$ between states. The first rule incorporates PCF reduction into $\rightarrow_\Sigma$; 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 location in the store, initializes the fresh location to its first argument, and invokes its second argument with the new location. The get operator invokes its second argument with the value in the store at the location given by its first argument. The set! destructive update operator 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 $\rightarrow_\Sigma$ is denoted by $\rightarrow^*_\Sigma$.

A labeled transition system $\xrightarrow{\alpha}$ for the I/O operators is defined in Table 2. 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 nil operator takes a $\surd$ transition to stopped state $[\mathsf{0},\, \Sigma']$, from which there are no transitions.

2.3 Simulation

Our basic notion of term equivalence, *simulation*, characterizes when two states have the same I/O behavior. We define the simulation relation, $\lesssim_{sim}$, as the greatest fixed point of a monotonic operator on relations. The simulation relation, and most other relations we define, is a preorder. Because our semantics is deterministic (up to location constants), symmetrization of the relations – e.g., simulation symmetrizes to bisimulation – is straightforward but elided in this abstract.

Definition 2.1 ($[-]_{sim}$)

Given R a relation on states, $[M,\, \Sigma_M]\ [R]_{sim}\ [N,\, \Sigma_N]$ iff

1. *if* $[M,\, \Sigma_M] \rightarrow^*_\Sigma [\mathsf{write}\, n M',\, \Sigma'_M]$, *then* $\exists N', \Sigma'_N$ s.t. $[N,\, \Sigma_N] \rightarrow^*_\Sigma [\mathsf{write}\, n N',\, \Sigma'_N]$ *and* $[M',\, \Sigma'_M]\ R\ [N',\, \Sigma'_N]$,
2. *if* $[M,\, \Sigma_M] \rightarrow^*_\Sigma [\mathsf{read}\, M',\, \Sigma'_M]$, *then* $\exists N', \Sigma'_N$ s.t. $[N,\, \Sigma_N] \rightarrow^*_\Sigma [\mathsf{read}\, N',\, \Sigma'_N]$ *and* $\forall n.[M'\, n,\, \Sigma'_M]\ R\ [N'\, n,\, \Sigma'_N]$,
3. *if* $[M,\, \Sigma_M] \rightarrow^*_\Sigma [\mathsf{nil},\, \Sigma'_M]$, *then* $\exists \Sigma'_N$ s.t. $[N,\, \Sigma_N] \rightarrow^*_\Sigma [\mathsf{nil},\, \Sigma'_N]$ *and* $[\mathsf{0},\, \Sigma'_M]\ R\ [\mathsf{0},\, \Sigma'_N]$,

Simulation is defined to be the greatest fixed point of $[-]_{sim}$:

Definition 2.2 $\lesssim_{sim} = gfp([-]_{sim})$.

The definition of $\lesssim_{sim}$ as a greatest fixed point gives us the following principle of coinduction:

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

Simulation defines when two states (i.e. term-store pairs) have the same behavior. We say that two *terms* are *operationally similar* if they have the same I/O behavior when inserted in every possible state context. We define operational simulation on *open* terms, as contexts may trap free variables. The superscript o in the notation $\lesssim^o$ is a reminder that the terms may be open.

Definition 2.3 (operational simulation, $\Gamma \vdash - \lesssim^o -{:}\tau$)
If $\Gamma \vdash M, N{:}\tau$, then $\Gamma \vdash M \lesssim^o N{:}\tau$ iff for all closing contexts $C[-_\tau]$ of type pr that respect Γ, and for all stores Σ,

$$[C[M],\, \Sigma] \lesssim_{sim} [C[N],\, \Sigma]$$

Operational simulation is defined by quantification over all contexts. We define *applicative simulation* by considering only applicative contexts. We use the subscript *ap* on the $\lesssim^o_{ap}$ symbol as an indicator that the relation is over a restricted set of contexts compared to the "fully quantified" relation $\lesssim^o$.

For convenience, we give the definition of applicative simulation first for closed terms and then extend it to open terms.

Definition 2.4 (closed applicative simulation, $- \lesssim_{ap} -{:}\tau$)
If M and N are closed terms of type τ, then, by induction on τ, $M \lesssim_{ap} N{:}\tau$ iff:

base types: *if $M \rightarrow^*_{pcf} v$, then $N \rightarrow^*_{pcf} v$, where v is a value of the appropriate type.*

ref(τ'): *if $M \rightarrow^*_{pcf} l^{\tau'}$, then $N \rightarrow^*_{pcf} l^{\tau'}$*

$\tau' \rightarrow \tau''$: *for all closed terms P of type τ', $(M\ P) \lesssim_{ap} (M\ P){:}\tau''$*

pr: *for all stores* Σ, $[M,\, \Sigma] \lesssim_{sim} [N,\, \Sigma]$

Note that we can define $\lesssim_{ap}$ equivalently as the greatest fixed point of an operator on type-indexed families of relations between terms, thus giving us a coinductive characterization of applicative simulation. However, in the proofs for this paper we only need induction on types.

We extend applicative simulation to a relation on open terms by quantifying over closing substitutions.

Definition 2.5 (applicative simulation, $\Gamma \vdash - \lesssim^{o}_{ap} -{:}\tau$)

If $\Gamma \vdash M, N{:}\tau$, *then* $\Gamma \vdash M \lesssim^{o}_{ap} N{:}\tau$ *iff for all closing substitutions* σ *which respect* Γ,

$$M\sigma \lesssim_{ap} N\sigma{:}\tau$$

2.4 Theory of the Metalanguage

The main theoretical result in this paper is that applicative simulation coincides with operational simulation, i.e. $\lesssim^{o}_{ap} = \lesssim^{o}$.

We get this extensionality result by showing that applicative simulation is a *precongruence*, namely that $\lesssim^{o}_{ap}$ is preserved by the term constructors of the metalanguage. To prove the precongruence theorem, we need a substitution lemma, stating that substitution of a variable by $\lesssim_{ap}$-related terms produces $\lesssim_{ap}$-related terms. The proof of the substitution lemma is the most involved, and we give an overview of the proof in this abstract. The proof of the substitution lemma in turn utilizes a number of other lemmas, which are stated without proof.

Lemma 2.1 (substitution)

If $x{:}\tau' \vdash P{:}\tau$ *and* $M \lesssim_{ap} N{:}\tau'$, *then*

$$P[M/x] \lesssim_{ap} P[N/x]{:}\tau$$

.

Proof: Proof is by induction on τ, with the *pr* case being the most complicated:

base types: If $x{:}\tau' \vdash P{:}$base and $M \lesssim_{ap} N{:}\tau'$, then if $P[M/x] \to^{*}_{pcf} v$, then $P[N/x] \to^{*}_{pcf} v$.

ref(τ''): If $x{:}\tau' \vdash P{:}ref(\tau'')$ and $M \lesssim_{ap} N{:}\tau'$, then if $P[M/x] \to^{*}_{pcf} l^{\tau''}$, then $P[N/x] \to^{*}_{pcf} l^{\tau''}$. See Lemma B.3.

$\tau_1{\to}\tau_2$: If $x{:}\tau' \vdash P{:}\tau_1{\to}\tau_2$ and $M \lesssim_{ap} N{:}\tau'$, then for all closed terms Q of type τ_1, $(P[M/x]\ Q) \lesssim_{ap} (P[N/x]\ Q){:}\tau_2$. This case follows directly from the induction hypothesis.

pr: If $x{:}\tau' \vdash P{:}pr$ and $M \lesssim_{ap} N{:}\tau'$, then for all stores Σ, $[P[M/x],\ \Sigma] \lesssim_{sim} [P[N/x],\ \Sigma]$.

To prove this case, we define a "candidate" simulation relation,

$$S = \{([P,\ \Sigma]\sigma_1, [P,\ \Sigma]\sigma_2) \mid fv(P, rng(\Sigma)) \subseteq \{x\}\}$$

where $\sigma_1 = [M/x]$, $\sigma_2 = [N/x]$, and the notation $[P,\ \Sigma]\sigma$ means that σ is applied to P and to each term in the range of Σ. We then show that $S \subseteq [\lesssim_{sim} \circ \Sigma \circ \lesssim_{sim}]_{sim}$ which is easily shown to imply that $S \subseteq \lesssim_{sim}$ (this is an instance of "simulation up to" as in [Mil90, Propositions 3.4.1–3.4.3]).

For the write case, we assume that $[P,\ \Sigma]\sigma_1 \rightarrow^*_{\Sigma} [\mathsf{write}\, i P',\ \Sigma']$, and we must show that $\exists P'', \Sigma''$ s.t. $[P,\ \Sigma]\sigma_2 \rightarrow^*_{\Sigma} [\mathsf{write}\, i P'',\ \Sigma'']$ and $[P',\ \Sigma'](\lesssim_{sim} \circ S \circ \lesssim_{sim})[P'',\ \Sigma'']$.

By Corollary B.1 we know that there is a leftmost reduction sequence such that $[P,\ \Sigma]\sigma_1 \rightarrow^*_{l,\Sigma} [\mathsf{write}\, i P''',\ \Sigma''']$ and that $[P',\ \Sigma'] \lesssim_{sim} [P''',\ \Sigma''']$.

We construct an induction measure (n, b) based on the length n of the leftmost reduction and on the form of P: $b = 1$ if the head of P is the variable x and $b = 0$ otherwise. We proceed by considering each possible case of the form of P. For each case of P, we must show:

$$\begin{array}{l} \text{if } [P,\ \Sigma]\sigma_1 \rightarrow^{(n,b)}_{l,\Sigma} [\mathsf{write}\, i P''',\ \Sigma'''],\ \text{then } \exists P'', \Sigma'' \text{ s.t.} \\ \quad [P,\ \Sigma]\sigma_2 \rightarrow^*_{\Sigma} [\mathsf{write}\, i P'',\ \Sigma''] \text{ and } [P''',\ \Sigma'''](S \circ \lesssim_{sim})[P'',\ \Sigma'']. \end{array}$$

The cases where P is headed by x or a PCF redex are straightforward. The case for $P = \mathsf{new}\, P_1 P_2$ relies on Lemma B.1 stating that equivalence up to renaming of locations, denoted α_{loc}, implies $\lesssim_{sim}$. The cases for get and $\mathsf{set!}$ rely on Lemma B.3 to show that the $ref(\tau)$ terms reduce to identical location constants.

The proofs for the read and nil cases are similar to the proof for write.

∎

Using the substitution lemma, we can prove the precongruence theorem:

Theorem 2.1 *$\lesssim^{o}_{ap}$ is a precongruence – that is, $\lesssim^{o}_{ap}$ is a preorder, and*

1. *$n \lesssim^{o}_{ap} n{:}i$,*
2. *if $c{:}\tau$, then $c \lesssim^{o}_{ap} c{:}\tau$,*
3. *if $\Gamma \vdash x{:}\tau$, then $\Gamma \vdash x \lesssim^{o}_{ap} x{:}\tau$,*
4. *if $\Gamma, x{:}\tau' \vdash M \lesssim^{o}_{ap} N{:}\tau$, then $\Gamma \vdash (\lambda x^{\tau'}.M) \lesssim^{o}_{ap} (\lambda x^{\tau'}.N){:}\tau$,*
5. *if $\Gamma \vdash M \lesssim^{o}_{ap} M'{:}\tau' \rightarrow \tau$ and $\Gamma \vdash N \lesssim^{o}_{ap} N'{:}\tau'$, then $\Gamma \vdash (MN) \lesssim^{o}_{ap} (M'N'){:}\tau$.*

Proof: The proofs that $\lesssim^{o}_{ap}$ is reflexive and transitive are straightforward. The cases for type pr require that $\lesssim_{sim}$ be a preorder, which is an easy coinduction.

The first three cases follow by the reflexivity of $\lesssim^{o}_{ap}$. The fourth case is a straightforward calculation. We split the last case into two subcases:

(a) if $\Gamma \vdash M \lesssim^o_{ap} N{:}\tau'{\to}\tau$, then for all closed P of type τ', $\Gamma \vdash (M\ P) \lesssim^o_{ap} (N\ P){:}\tau$, and

(b) if $\Gamma \vdash M \lesssim^o_{ap} N{:}\tau'$, then for all closed P of type $\tau'{\to}\tau$, $\Gamma \vdash (P\ M) \lesssim^o_{ap} (P\ N){:}\tau$.

Case a follows directly from the definitions. For case b, first close the terms with a substitution σ. By $\Gamma \vdash M \lesssim^o_{ap} N{:}\tau'$, we know that $M\sigma \lesssim_{ap} N\sigma{:}\tau$. The substitution lemma gives us $(P\ x)[(M\sigma)/x] \lesssim_{ap} (P\ x)[(N\sigma)/x]$, which equals $(P\ (M\sigma)) \lesssim_{ap} (P\ (N\sigma))$, which is what we need. Case 5 then follows by transitivity. ∎

Given that $\lesssim^o_{ap}$ is a precongruence, the context lemma follows easily.

Theorem 2.2 (extensionality) $\Gamma \vdash M \lesssim^o_{ap} N{:}\tau \Leftrightarrow \Gamma \vdash M \lesssim^o N{:}\tau$

Proof:

($\Rightarrow$) Assume $\Gamma \vdash M \lesssim^o_{ap} N{:}\tau$. By the precongruence of $\lesssim^o_{ap}$, for all contexts $C[-_\tau]$ of type pr, $\Gamma \vdash C[M] \lesssim^o_{ap} C[N]{:}pr$. By the definition of $\lesssim^o_{ap}$ at type pr, the result follows.

($\Leftarrow$) Assume $\Gamma \vdash M \lesssim^o N{:}\tau$ – that is, for all contexts $C[-_\tau]$ of type pr and for all stores Σ, $[C[M], \Sigma] \lesssim_{sim} [C[N], \Sigma]$. We show that $\Gamma \vdash M \lesssim^o_{ap} N{:}\tau$ by induction on τ. For each case, construct a context that captures all free variables of M and N, which is easy. The proof for each case is then straightforward.

∎

3 CPS Translation to the Metalanguage

Given two terms of the metalanguage, we now have a well-structured approach to proving them operationally equivalent. The metalanguage would be tedious to program in directly and is intended to be the target of a translation from a more traditional programming language. Assuming we have a syntax-directed translation, $[\![-]\!]$ from the source language to the metalanguage, we know that if

$$[\![e_1]\!] \lesssim^o [\![e_2]\!]$$

then, by compositionality, we have that

$$[\![C[e_1]]\!] \lesssim^o [\![C[e_2]]\!]$$

for all contexts $C[]$ in the source language. Due to the context lemma, we need only prove the equivalence of the metalanguage terms in all applicative contexts.

In the following section, we define a simply-typed programming language with assignment and call-by-value execution. We then give a CPS translation from expressions in the source language to terms in our metalanguage. A program in the source language translates into a metalanguage state. We then use the translation and the context lemma for the metalanguage to show the equivalence of two source language expressions.

3.1 A Simple Imperative Language

The syntax of the source language is given by the following grammar:

$$\begin{array}{rl} p ::= & \texttt{prog}\ e\ \texttt{end} \\ e ::= & x \mid n \mid \texttt{fn x:}t.e \mid e(e) \mid e;e \\ & \mid x\ \texttt{:=}\ e \mid \texttt{print}\ e \mid \texttt{read} \\ & \mid \texttt{true} \mid \texttt{false} \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \\ t ::= & \texttt{bool} \mid \texttt{int} \mid t\texttt{->}t \end{array}$$

The typing judgements of the source language, $A \vdash_{src} e{:}t$ are standard and omitted. The main expression of a program must be of integer type.

3.2 A CPS Translation into the Metalanguage

We define a translation, $[\![-]\!]$, on types and expressions of the source language into types and terms of the metalanguage. Types are translated according to the following rules:

$$[\![\texttt{int}]\!] = i, \quad [\![\texttt{bool}]\!] = b, \quad [\![t_1\texttt{->}t_2]\!] = [\![t_1]\!]\to([\![t_2]\!]\to pr)\to pr$$

We extend the translation to source language type assumptions A as follows:

$$[\![A]\!](x) = \mathit{ref}([\![A(x)]\!])$$

The translation of source language programs into metalanguage states is given by the following rule:

$$[\![\texttt{prog}\ \ e\ \ \texttt{end}]\!] = [[\![e]\!](\lambda x{:}i.\,\mathsf{write}\,x\,\mathsf{nil}),\ \emptyset]$$

which evaluates the program's body starting with an empty store, writes the result, and halts.

The translation of source language expressions adheres to the following type discipline: if $A \vdash_{src} e : t$, then $[\![e]\!]{:}([\![t]\!]\to pr)\to pr$. Following are the rules for translation of source language expressions.

(int) $[\![n]\!] = \lambda\kappa{:}i\to pr.\kappa\; n$

(var) if $A \vdash_{src} x : t$, then

$$[\![x]\!] = \lambda\kappa{:}[\![t]\!]\to pr.\,\mathsf{get}\;x\;\kappa$$

(fn) if $A, \texttt{x}{:}t_1 \vdash_{src} e{:}t_2$, then

$$\begin{aligned}[\![\texttt{fn x:}t_1.e]\!] = \lambda\kappa{:}&([\![t_1]\!]\to([\![t_2]\!]\to pr)\to pr)\to pr.\\ &\kappa(\lambda v{:}[\![t_1]\!].\lambda k'{:}[\![t_2]\!]\to pr.\,\mathsf{new}^{[\![t_1]\!]}\,v(\lambda x{:}\mathit{ref}([\![t_1]\!]).[\![e]\!]\;\kappa'))\end{aligned}$$

(application) if $A \vdash_{src} e_1{:}t_1\texttt{->}t_2$ and $A \vdash_{src} e_2{:}t_1$, then

$$[\![e1(e2)]\!] = \lambda\kappa{:}[\![t_2]\!]\to pr.[\![e_1]\!](\lambda f{:}[\![t_1]\!]\to([\![t_2]\!]\to pr)\to pr.[\![e_2]\!](\lambda v{:}[\![t_1]\!].f\;v\;\kappa))$$

(assignment) if $A \vdash_{src} e{:}t$ and $A \vdash_{src} x{:}t$, then

$$[\![x\ \ \texttt{:=}\ \ e]\!] = \lambda\kappa{:}[\![t]\!]\to pr.[\![e]\!](\lambda v{:}[\![t]\!].\,\mathsf{set!}\,x\;v\;(\kappa\;v))$$

(sequence) if $A \vdash_{src} e_1{:}t_1$ and $A \vdash_{src} e_2{:}t_2$, then

$$[\![e_1; e_2]\!] = \lambda\kappa{:}[\![t_2]\!]{\rightarrow}pr.[\![e_1]\!](\lambda v_1{:}[\![t_1]\!].[\![e_2]\!]\,\kappa)$$

(print) if $A \vdash_{src} e{:}\mathtt{int}$, then

$$[\![\mathtt{print}\ e]\!] = \lambda\kappa{:}i{\rightarrow}pr.[\![e]\!](\lambda v{:}i.\,\mathsf{write}\,v\,(\kappa\,v))$$

(read) $[\![\mathtt{read}]\!] = \lambda\kappa{:}i{\rightarrow}pr.\,\mathsf{read}\,\kappa$

(if-then-else) if $A \vdash_{src} e_1{:}\mathtt{bool}, e_2{:}t, e_3{:}t$, then

$$[\![\mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3]\!] = \lambda\kappa{:}[\![t]\!]{\rightarrow}pr.[\![e_1]\!](\lambda v{:}b.\,\mathsf{if}\,v\,([\![e_2]\!]\kappa)\,([\![e_3]\!]\kappa))$$

Integers are translated directly. Variables are translated directly, except that if a variable `x` is of type `t` in the source language, the corresponding variable x in the metalanguage will be of type $ref([\![\mathtt{t}]\!])$.

The translation of a function is an abstraction that expects a value v and a runtime continuation κ'. When the abstraction is invoked at procedure call time, a new location is allocated and bound to the variable x. Within the binding of x to the new location, the body of the function, e, is run, with the runtime continuation.

Application evaluates the function and then the argument and then invokes the function abstraction with the value of the argument. Assignment relies on the variable x being within the scope of the assignment statement, and the continuation is invoked with the new value of the variable. The continuation of a sequence will get the value of the second expression of the sequence. A print expression sends the value printed (which must be an integer) to its continuation. Read gets an integer from the standard input and sends the integer to its continuation. The translation of a conditional is what one would expect.

3.3 A Small Example

Because the translation is compositional, we can show operational equivalence of source language expressions by showing the operational equivalence in the metalanguage of the translations of the expressions.

Consider the two source language expressions:

$$e_1 \stackrel{\text{def}}{=} (\mathtt{fn\ x:int}.e)(\mathtt{1})$$
$$e_2 \stackrel{\text{def}}{=} e$$

where the variable `x` is not in the free variables of e. Even though e_1 will cause allocation and update of the store for the unused parameter `x`, we would like to show that the two expressions are operationally equivalent. Suppose e has type `int`.

The translation of e_1 is:

$$[\![e_1]\!] = \lambda\kappa{:}i{\rightarrow}pr.([\![\mathtt{fn\ x:int}.e]\!])(\lambda f{:}i{\rightarrow}(i{\rightarrow}pr){\rightarrow}pr.[\![1]\!](\lambda v{:}i.f\ v\ \kappa))$$

and the translation of `fn x:int.`e is:

$$[\![\mathtt{fn\ x:int}.e]\!] = \lambda\kappa'{:}(i{\rightarrow}(i{\rightarrow}pr){\rightarrow}pr){\rightarrow}pr.\kappa'(\lambda v{:}i.(\lambda\kappa''{:}i{\rightarrow}pr.\,\mathsf{new}^i\,v\,(\lambda x{:}ref(i).[\![e]\!]\,\kappa''))).$$

By doing several beta-reductions, we show that the translation of e_1 is beta-equal to:

$$\lambda\kappa{:}i{\rightarrow}pr.\,\mathsf{new}^i\,1\,(\lambda x{:}ref(i).[\![e]\!]\,\kappa)$$

The translation of e_2 is simply $(\lambda\kappa{:}i{\rightarrow}pr.[\![e]\!]\,\kappa)$.

To prove applicative, and therefore operational, simulation, we must show that for all $\kappa{:}i{\rightarrow}pr$ and for all stores Σ,

$$[\mathsf{new}^i\,1\,(\lambda x{:}ref(i).[\![e]\!]\,\kappa),\,\Sigma] \lesssim_{sim} [[\![e]\!]\,\kappa,\,\Sigma]$$

The left-hand side takes a single step to

$$[([\![e]\!]\,\kappa)[l^i/x],\,\Sigma[l^i \mapsto 1]]$$

with $l^i \notin dom(\Sigma) \cup locs([\![e]\!], \kappa, rng(\Sigma))$ which, because $x \notin fv([\![e]\!])$ and κ is closed, is equal to:

$$[([\![e]\!]\,\kappa),\,\Sigma[l^i \mapsto 1]].$$

It is easy to show that the above state is in α_{loc} with $[[\![e]\!]\,\kappa,\,\Sigma]$, so by Lemma B.1 we have operational simulation. The operational simulation is easily shown to be symmetric, and we have operational equivalence of the translated terms. By compositionality, we conclude that the original source language terms, e_1 and e_2, are operationally equivalent in the source language, as desired.

4 Conclusions

We have presented a metalanguage that directly supports both I/O and imperative operations on a dynamic store. We have demonstrated that our metalanguage enjoys an extensionality property, which greatly simplifies the process of proving operational equivalence of programs. The extensionality of our metalanguage relies crucially on the fact that the store and I/O operators are in CPS and therefore only operate at the outermost level of terms. Using a CPS translation from an imperative programming language into our metalanguage, it is possible to reason about the original source language terms using the robust theory of the metalanguage. In particular, it is possible to prove operational equivalence of expressions in a non-extensional imperative language by first translating the terms into our metalanguage and then proving applicative equivalence of the translated terms.

It is possible to obtain a similar extensionality property for an untyped version of the metalanguage presented in this paper. We found, however, that the untyped applicative contexts were still not restricted enough to prove useful results about translated program fragments. Reasoning about every program context translates in our framework into reasoning about every possible continuation, and in an untyped setting one can still construct malicious continuations which destroy applicative equivalence. The types, therefore, are in the metalanguage not to give us the extensionality, but to further limit the range of applicative contexts which need to be considered for a given pair of translated terms.

This paper presents the first known applicative characterization of operational equivalence for a language with a dynamic store. This improves upon [MT91, HMST95] which require reasoning over the larger set of reduction contexts. We also improve upon [MT91, HMST95] by allowing full β reasoning within subterms rather than relying on the weaker β-value rule.

References

[Gor94] Andrew D. Gordon. *Functional Programming and Input/Output.* Cambridge University Press, Cambridge, 1994.

[HMST95] Furio Honsell, Ian A. Mason, Scott Smith, and Carolyn Talcott. A variable typed logic of effects. *Information and Computation*, 119:55–90, 1995.

[HMT84] J. Y. Halpern, Albert R. Meyer, and Boris A. Trakhtenbrot. The semantics of local storage, or what makes the free-list free? In *Conf. Rec. 11th ACM Symposium on Principles of Programming Languages*, pages 245–257, 1984.

[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.

[Mit95] John C. Mitchell. *Foundations for Programming Languages.* MIT Press, Cambridge, MA, 1995.

[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.

[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.

[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.

[San94] Davide Sangiorgi. The lazy lambda calculus in a concurrency scenario. *Information and Computation*, 111(1):120–153, May 1994.

[Wal91] David Walker. π-calculus semantics of object-oriented programming languages. In *Proc. Conference on Theoretical Aspects of Computer Software*, Lecture Notes in Computer Science, Berlin, Heidelberg, and New York, 1991. Springer-Verlag.

$$\Gamma \vdash x{:}\Gamma(x) \quad \Gamma \vdash l^{\tau}{:}\mathit{ref}(\tau) \quad \Gamma \vdash c{:}\tau \text{ if } c{:}\tau \in \mathit{Const}$$

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

$$\frac{\Gamma, x{:}\tau' \vdash M{:}\tau}{\Gamma \vdash \lambda x{:}\tau'.M{:}\tau'{\to}\tau}$$

Table 3: Typing Rules

$$R ::= [] \mid x \mid n \mid c \mid l^{\tau} \mid \lambda x{:}\tau.R \mid RM \mid MR$$

$$\begin{array}{l}
R[(\lambda x.M)N] \to_{pcf} R[M[N/x]] \\
R[\mathsf{Y}M] \to_{pcf} R[M(\mathsf{Y}M)] \\
R[\mathsf{pred}\,0] \to_{pcf} R[0] \\
R[\mathsf{pred}(\mathsf{succ}\,M)] \to_{pcf} R[M] \\
R[\mathsf{zero?}\,0] \to_{pcf} R[\mathsf{true}] \\
R[\mathsf{zero?}(\mathsf{succ}\,M)] \to_{pcf} R[\mathsf{false}] \\
R[\mathsf{if}\,\mathsf{true}\,MN] \to_{pcf} R[M] \\
R[\mathsf{if}\,\mathsf{false}\,MN] \to_{pcf} R[N] \\
R[\mathsf{eq?}\,l_1 l_2] \to_{pcf} R[\mathsf{true}],\ \text{if } l_1 = l_2 \\
R[\mathsf{eq?}\,l_1 l_2] \to_{pcf} R[\mathsf{false}],\ \text{if } l_1 \neq l_2
\end{array}$$

Table 4: PCF Reduction Rules

APPENDIX

A PCF Typing and Semantics

Table 3 gives the typing rules for the metalanguage, which are consistent with the standard typing judgements of PCF.

Table 4 gives the the operational semantics of the PCF fragment of the metalanguage. Reduction takes place within *reduction contexts*, $R[]$.

B More Theory of the Metalanguage

This section contains several of the more important lemmas involved in the proofs of the main results in this paper. As mentioned earlier, these lemmas are often useful in proofs of operational equivalence of metalanguage terms.

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:

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

Definition B.1 $[M,\, \Sigma_M]\ \alpha_{loc}\ [N,\, \Sigma_N]$ *iff* $\exists\Theta{:}loc \leftrightarrow loc$ s.t.

1. $N = M\Theta$
2. $locs(M) \subseteq dom(\Theta)$
3. *if* $(l, l') \in \Theta$ *then either*
   (a) $l \notin dom(\Sigma_M)$ *and* $l \notin dom(\Sigma_N)$ *or*
   (b) $\Sigma_M(l)\Theta = \Sigma_N(l')$ *and* $locs(\Sigma_M(l)) \subseteq dom(\Theta)$

Lemma B.1 $\alpha_{loc} \subseteq \lesssim_{sim}$.

Proof: Proof is by induction on the length of leftmost reduction to an I/O operator, as in the proof of the substitution lemma. ∎

We show that PCF reductions preserve applicative similarity. Equality up to PCF reduction is denoted by $=_{pcf}$.

Lemma B.2 *If* $\Gamma \vdash M =_{pcf} N{:}\tau$, *then* $\Gamma \vdash M \lesssim^{o}_{ap} N{:}\tau$.

As mentioned earlier, the $ref(\tau)$ case of the substitution lemma is a standalone lemma.

Lemma B.3 *If* $x{:}\tau \vdash P{:}ref(\tau')$ *and* $M \lesssim_{ap} N{:}\tau$, *then if* $P[M/x] \rightarrow^{*}_{pcf} l^{\tau'}$, *then* $P[N/x] \rightarrow^{*}_{pcf} l^{\tau'}$.

Finally, there are a number of lemmas related to the completeness up to $\lesssim_{sim}$ of leftmost reduction, denoted by $\rightarrow^{*}_{l,\Sigma}$.

Lemma B.4 *If* $[M,\, \Sigma_M] =_{pcf} [N,\, \Sigma_N]$ *and* $[M,\, \Sigma_M] \rightarrow^{*}_{\Sigma} [\mathsf{write}\, iM',\, \Sigma'_M]$, *then* $\exists N', \Sigma'_N$ *such that* $[N,\, \Sigma_N] \rightarrow^{*}_{l,\Sigma} [\mathsf{write}\, iN',\, \Sigma'_N]$ *and* $[M',\, \Sigma'_M] \lesssim_{sim} [N',\, \Sigma'_N]$.

Proof: We need a number of basic properties of PCF reduction, including confluence (CR) and strong normalization when the Y's are labeled. These results are standard and can be obtained following the outline of [Mit95]. ∎

Corollary B.1 *If* $[M,\, \Sigma_M] \rightarrow^{*}_{\Sigma} [\mathsf{write}\, iM',\, \Sigma'_M]$, *then* $\exists M'', \Sigma''_M$ *such that* $[M,\, \Sigma_M] \rightarrow^{*}_{l,\Sigma} [\mathsf{write}\, iM'',\, \Sigma''_M]$ *and* $[M',\, \Sigma'_M] \lesssim_{sim} [M'',\, \Sigma''_M]$.