November 1989

Computing With Coercions

Val Tannen
University of Pennsylvania

Carl A. Gunter
University of Pennsylvania

Andre Scedrov
Stanford University

Computing With Coercions

Abstract

This paper relates two views of the operational semantics of a language with multiple inheritance. It is shown that the introduction of explicit coercions as an interpretation for the implicit coercion of inheritance does not affect the evaluation of a program in an essential way. The result is proved by semantic means using a denotational model and a computational adequacy result to relate the operational and denotational semantics.

Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-89-62.

COMPUTING WITH COERCIONS

V. Breazu-Tannen
C.A. Gunter
A. Scedrov

MS-CIS-89-62
LOGIC & COMPUTATION 11

Department of Computer and Information Science
School of Engineering and Applied Science
University of Pennsylvania
Philadelphia, PA 19104

November 1989

COMPUTING WITH COERCIONS

(Extended Abstract)

V. Breazu-Tannen *C. A. Gunter* *A. Scedrov*[1]

University of Pennsylvania
December 4, 1989

Abstract. This paper relates two views of the operational semantics of a language with multiple inheritance. It is shown that the introduction of explicit coercions as an interpretation for the implicit coercion of inheritance does not affect the evaluation of a program in an essential way. The result is proved by semantic means using a denotational model and a computational adequacy result to relate the operational and denotational semantics.

1 Introduction

There have been a number of efforts to understand the denotational semantics of inheritance polymorphism and a variety of mathematical models for languages with subtle semantic features have been discovered. However, as far as the authors of this paper know, no one has attempted to discuss what, if anything, these denotational models have to do with the intended execution of programs in the languages they model. For example, *all* of the published denotational models of the language Fun of Cardelli Wegner [CW85] (including the work of authors of this paper) model this language in way that corresponds to no reasonable interpretation of its operational semantics! No functional programming language in common use diverges when evaluating the program $\lambda x.\ e$, even when the expression e may diverge. Yet the models for Fun which have been studied identify the abstraction $\lambda x.\ \bot$ with the divergent program $\bot$. Besides this problem, all existing models satisfy the unrestricted β rule, which fails to be a legitimate transformation in call-by-value languages. Since call-by-value is the most common form of evaluation, one is led to ask whether this commitment to β was an important feature of the models concerned. In short, very little has been done to close the gap between denotational and operational theories of inheritance. We see two basic things as missing from the current theories: (1) a careful discussion of the structional operational semantics of languages with inheritance type systems and (2) any account of the relationship between the suggested models and a reasonable account of operational semantics.

Our goal in this paper is to attempt an account of problem (1) guided by an approach to (2). We carry out this study in a simple, familiar context by using an extension of Plotkin's illustrative language PCF [Plo77]. We develop a simple structural operational semantics for this language in the spirit of

[1]Author's addresses. Breazu-Tannen (val@cis.upenn.edu) and Gunter (gunter@cis.upenn.edu): Department of Computer and Information Sciences, University of Pennsylvania, Philadelphia PA 19104, USA. Scedrov (andre@csli.stanford.edu) is on leave at: Department of Computer Science, Stanford University, Stanford CA 94305, USA.

the evaluation mechanisms of languages such as LISP and ML in which functions call their parameters by value. Our extension, which we call PCF+, is obtained by adding record and variant types. This language is extended to a new language, PCF++, by permitting the use of a form of inheritance which allows more programs to be viewed as type correct. We then study the question of the proper operational interpretation of PCF++. One possible approach is simple to understand: after a PCF++ program is shown to be type correct, the type information in the term is erased and the resulting term (which lives in an extended untyped lambda-calculus) is evaluated. However, in view of the form of semantics that we have studied in our work on Fun and its relatives [BCGS89, BCGS90] there is another view of the proper operational semantics of PCF++. Under this view, a term of PCF++ is translated into a PCF+ term by inserting explicit coercions which "explain" the inheritance in the original PCF++ program in an intuitive way. If this "explanation" really is intuitive and the first form of evaluation (which is a common from of implementation) is reasonable, then it seems that there must be some relationship between these two views of program evaluation! Moreover, this latter approach is also not uncommon as a form of evaluation, and therefore has independent interest. In this paper we will show that these two forms of evaluation are essentially the same for observable types.

To give the reader an idea of what translation we have in mind let us look at an example of how a simple program would be evaluated. Applying our semantic paradigm to PCF++, we translate its programs into PCF+ programs, essentially by inserting explicit coercion terms wherever inheritance is used in type-checking. In anticipation of an exact definition of this translation (section 2), here is an example. PCF++ type-checks the program $P \equiv G(F)$ where

$$\begin{aligned} G &\equiv \lambda f : \{l : \mathbf{num}\} \to \mathbf{num}.\ \{k_1 = f(\{l = \mathbf{0}, l_1 = \mathbf{1}\}), k_2 = f(\{l = \mathbf{2}, l_2 = \mathbf{false}\})\} \\ F &\equiv \lambda x : \{l : \mathbf{num}\}.\ x.l \end{aligned}$$

Note that G will not type-check in PCF+ because of the different types of the two arguments to which f is applied.

The translation to PCF+ depends on the way P is type-checked. One possible translation is $P' \equiv G'(F)$ where

$$G' \equiv \lambda f : \{l : \mathbf{num}\} \to \mathbf{num}.\ \{k_1 = f(\xi_1(\{l = \mathbf{0}, l_1 = \mathbf{1}\})), k_2 = f(\xi_2(\{l = \mathbf{2}, l_2 = \mathbf{false}\}))\}$$

where ξ_1 and ξ_2 are the following *coercion terms*

$$\xi_1 \equiv \lambda x_1 : \{l : \mathbf{num}, l_1 : \mathbf{num}\}.\ \{l = x_1.l\} \qquad \xi_2 \equiv \lambda x_2 : \{l : \mathbf{num}, l_2 : \mathbf{bool}\}.\ \{l = x_2.l\}\ .$$

Another possible translation is $P'' \equiv G''(F)$ where

$$G'' \equiv \lambda f : \{l : \mathbf{num}\} \to \mathbf{num}.\ \{k_1 = \zeta_1(f)(\{l = \mathbf{0}, l_1 = \mathbf{1}\}), k_2 = \zeta_2(f)(\{l = \mathbf{2}, l_2 = \mathbf{false}\})\}$$

where

$$\zeta_1 \equiv \lambda f_1 : \{l : \mathbf{num}\} \to \mathbf{num}.\ \lambda x_1 : \{l = \mathbf{num}, l_1 = \mathbf{num}\}.\ f_1(\xi_1(x_1))$$

and

$$\zeta_2 \equiv \lambda f_2 : \{l : \mathbf{num}\} \to \mathbf{num}.\ \lambda x_2 : \{l = \mathbf{num}, l_2 = \mathbf{bool}\}.\ f_2(\xi_2(x_2))$$

The fact that the translation (more generally, the meaning) depends on the type-checking derivation entails the need for denotational coherence results [BCGS89]. In this paper, however, we will examine the computational (operational) aspects of this translation. Notice that the "execution" of both P' and P'' yields the same result

$$\{k_1 = \mathbf{0}, k_2 = \mathbf{2}\}$$

More importantly, so does the direct execution of P. The "direct" operational semantics for PCF++ that we have in mind is just the same as that of PCF+. It is a simple but crucial observation that the same evaluation rules work on programs allowed by the more permissive type discipline of PCF++. Not surprisingly, this is the natural way to implement such languages (Cardelli, personal communication about Quest [Car89]). Although it is may not be useful to fully translate a term before executing it, it is reasonable to ask whether translation would affect the evaluation. Since coercions remove the "junk" in a term, they may play a useful role in efficient implementation. However, our primary interest is in the abstract specification of the language and not the details of its efficient implementation.

Our *main result* relates the direct execution of a PCF++ program phrase e to the execution of *any* of its PCF+ translations, e^*. We prove that

e terminates if and only if e^* terminates.

If both e and e^* terminate, what can we say about the relationship between the results of the two computations? Of course, we are able to show that if the type of e is *ground*, (integer or boolean) then the results are the exactly the same. In this language we are also interested in computing with more complex objects, such as records/variants of records/variants of ground data (this is particularly consistent with the way things are viewed in object-oriented database programming applications [OBB89] for example). We call the types of such data *observable types*. Now, the philosophy of PCF++ is that the type of program phrases is part of them, i.e., user-supplied in some sense. (This is in contrast with the approaches based on type inference; see for example [Wan89].) At observable types, we show that the results of the two computations have the same *components* in those record fields which appear in the prescribed type of the program phrase. This is the best we can hope for, since the introduction of coercions yields computations which may remove "junk" fields, namely the fields not occurring in the prescribed type. Moral: if you specify a type for your program, don't expect to observe more than what the type allows. Anyway, our conclusion is that coercions *make no essential difference* to the computation.

While this result only relates our translation to the operational semantics, it can be used for *transfer of computational adequacy*. Consider a denotational semantics $\mathcal{D}^+$ of PCF+ for which our translation is coherent. This yields a denotational semantics $\mathcal{D}^{++}$ for PCF++ where a term is interpreted by first translating it into PCF+ and then taking the $\mathcal{D}^+$-meaning of the translation. Under some reasonable assumptions about $\mathcal{D}^+$, our main result implies that if $\mathcal{D}^+$ is computationally adequate (*i.e.* the meaning of a term e is non-bottom iff the evaluation of e terminates) for the operational semantics of PCF+ then $\mathcal{D}^{++}$ is computationally adequate for the operational semantics of PCF++.

An interesting methodological twist is that our proof of the main result actually uses a specific denotational semantics $[\![\cdot]\!]^+$ which is computationally adequate for PCF+ and for which this transfer can be done! As it is, we show directly that $[\![\cdot]\!]^{++}$ is computationally adequate for PCF++ and we derive our

main result from this. We regard this as a nice example of the use of a domain-theoretic semantics for obtaining an essentially syntactic result.

Another comment on methodology. We have chosen to focus on call-by-value operational semantics since this is the most common style of implementation for the languages we are studying and because it offers a change of pace from our earlier results [BCGS89] where we focused on models in which the unrestricted β axiom holds. We expect that results such as the ones we are proving in this paper could be formulated for a call-by-name operational semantics, although this would call for some changes in our concept of observability.

In section 2 we begin by introducing the syntax of PCF++ as an extension of PCF+. Then we describe the translation back, from PCF++ to PCF+. Finally we give the call-by-value operational semantics and state our main theorem. In section 3 we give a domain-theoretic denotational semantics of PCF+ for which our translation is coherent and for which the operational semantics of PCF+ is sound and computationally adequate. We prove that the operational semantics of PCF++ is sound and computationally adequate for the induced denotational semantics and then we show how to derive from this our main theorem. The paper ends with a section of conclusions and ideas for more work.

2 From PCF+ to PCF++ and back again.

In this section we introduce the two calculi on which the central result of the paper focuses.

2.1 Extending PCF+ to PCF++.

The following grammar defines the syntax of *type expressions* s and *raw terms* e of our calculi. We assume primitive syntax classes of variables and labels:

$$
\begin{array}{lcl}
x & \in & \textbf{Variable} \\
l & \in & \textbf{Label} \\
s & ::= & \textbf{num} \mid \textbf{bool} \mid s \to s \mid \{l_1 : s_1, \ldots, l_n : s_n\} \mid [l_1 : s_1, \ldots, l_n : s_n] \\
e & ::= & \textbf{0} \mid \textbf{Succ}(e) \mid \textbf{Pred}(e) \mid \textbf{true} \mid \textbf{false} \mid \textbf{IsZero}(e) \mid \\
 & & x : s \mid \lambda x : s.\ e \mid e(e) \mid \mu x : s.\ e \mid \textbf{if } e \textbf{ then } e \textbf{ else } e \mid \\
 & & \{l_1 = e, \ldots, l_n = e\} \mid e.l \mid [l = e] \mid \textbf{case } e \textbf{ of } l_1 \Rightarrow e, \ldots, l_n \Rightarrow e
\end{array}
$$

For *records* $\{l_1 = e_1, \ldots, l_n = e_n\}$ and *variants* $[l_1 = e_1, \ldots, l_n = e_n]$, it is assumed that the labels $l_1, \ldots, l_n$ are all distinct. We assume that the reader can infer from our notation what is meant by free and bound variables of raw terms. A raw term is said to be closed if it has no free variables.

A *type context* is a list $x_1 : s_1, \ldots, x_n : s_n$ of pairs of variables and types. We assume that the variables x_i in such a context are distinct. A *typing judgement* is a sequent of the form $H \vdash e : s$ where H is a typing context which includes all of the free variables of the raw term e. A typing judgement is said to be *derivable in PCF+* if it can be proved using the axioms and rules listed in Table 1. It is not hard to see that any derivable sequent has a *unique* derivation. This latter fact will not be true of the calculus PCF++ which we now define. PCF++ is the extension of PCF+ to a calculus with multiple inheritance. First of all, we define a binary relation $s < t$ of subtyping between type expressions s and t using the rules in Table 2. The reader can check that $<$ is a preorder on type expressions. This relation is now incorporated into the typing system of PCF++ by the addition of the *subsumption rule:*

$$\frac{H \vdash e : s \qquad s < t}{H \vdash e : t}$$

2.2 Translation from PCF++ into PCF+.

Definition: Given types s and t such that $s < t$ is provable, we define a PCF+ term $\mathsf{coerce}[s < t]$ of type $s \to t$ by induction on the proof of $s < t$ as follows:

- $\mathsf{coerce}[\textbf{bool} < \textbf{bool}] \equiv \lambda x : \textbf{bool}.\ x$ and $\mathsf{coerce}[\textbf{num} < \textbf{num}] \equiv \lambda x : \textbf{num}.\ x$.
- $\mathsf{coerce}[s \to t < s' \to t'] \equiv \lambda f : s \to t.\ \mathsf{coerce}[t < t'] \circ f \circ \mathsf{coerce}[s' < s]$
- Say $s \equiv \{l_1 : s_1, \ldots, l_n : s_n, \ldots, l_m : s_m\}$ and $t \equiv \{l_1 : t_1, \ldots, l_n : t_n\}$ and $s < t$, then

$$\mathsf{coerce}[s < t] \equiv \lambda x : s.\ \{l_1 = \mathsf{coerce}[s_1 < t_1](x.l_1), \ldots l_n = \mathsf{coerce}[s_1 < t_n](x.l_n)\}$$

$$\mathbf{0} : \mathbf{num} \qquad \mathbf{false} : \mathbf{bool} \qquad \mathbf{true} : \mathbf{bool}$$

$$\frac{H \vdash e : \mathbf{num}}{H \vdash \mathbf{Pred}(e) : \mathbf{num}} \qquad \frac{H \vdash e : \mathbf{num}}{H \vdash \mathbf{Succ}(e) : \mathbf{num}} \qquad \frac{H \vdash e : \mathbf{num}}{H \vdash \mathbf{IsZero}(e) : \mathbf{bool}}$$

$$H,\ x : s,\ H' \vdash x : s \qquad \frac{H,\ x : s \vdash e : t}{H \vdash \lambda x : s.\ e : t} \qquad \frac{H \vdash e : s \to t \qquad H \vdash e' : s}{H \vdash e(e') : t}$$

$$\frac{H,\ x : s \vdash e : s}{H \vdash \mu x : s.\ e : s} \qquad \frac{H \vdash e : \mathbf{bool} \qquad H \vdash e' : s \qquad H \vdash e'' : s}{H \vdash \mathbf{if}\ e\ \mathbf{then}\ e'\ \mathbf{else}\ e'' : s}$$

$$\frac{H \vdash e_1 : s_1 \quad \cdots \quad H \vdash e_n : s_n}{H \vdash \{l_1 = e_1, \ldots, l_n = e_n\} : \{l_1 : s_1, \ldots, l_n = s_n\}} \qquad \frac{H \vdash e : \{l_1 : s_1, \ldots, l_n : s_n\}}{H \vdash e.l_i : s_i}$$

$$\frac{H \vdash e_i : s_i}{H \vdash [l_i = e_i] : [l_1 : s_1, \ldots, l_n = s_n]}$$

$$\frac{H \vdash e : [l_1 : s_1, \ldots, l_n = s_n] \qquad H \vdash f_1 : s_1 \to s \quad \cdots \quad H \vdash e_n : s_n \to s}{H \vdash \mathbf{case}\ e\ \mathbf{of}\ l_1 \Rightarrow f_1 \cdots l_n \Rightarrow f_n : s}$$

Table 1: Typing rules for PCF+.

$$\mathbf{num} < \mathbf{num} \qquad \frac{s' < s \qquad t < t'}{s \to t < s' \to t'}$$

$$\mathbf{bool} < \mathbf{bool}$$

$$\frac{s_1 < t_1 \quad \cdots \quad s_n < t_n}{\{l_1 : s_1, \ldots, l_n : s_n, \ldots, l_m : s_m\} < \{l_1 : t_1, \ldots, l_n : t_n\}}$$

$$\frac{s_1 < t_1 \quad \cdots \quad s_n < t_n}{[l_1 : s_1, \ldots, l_n : s_n] < [l_1 : t_1, \ldots, l_n : t_n, \ldots, l_m : t_m]}$$

Table 2: Inheritance rules.

- Say $s \equiv [l_1 : s_1, \ldots, l_n : s_n]$ and $t \equiv [l_1 : t_1, \ldots, l_n : t_n, \ldots, l_m : t_m]$ and $s < t$, then

$$\mathsf{coerce}[s < t] \equiv \lambda x : s.\ \textbf{case}\ x\ \textbf{of}\ l_1 \Rightarrow f_1, \ldots, l_n \Rightarrow f_n$$

  where $f_i \equiv \lambda y : s_i.\ [l_i = \mathsf{coerce}[s_i < t_i](y)]$ for each $i = 1, \ldots, n$. ∎

Lemma 1 *If $s < t$ is derivable, then so is* $\vdash \mathsf{coerce}[s < t] : s \to t$ ∎

We will now describe how we translate the derivations of typing judgments of PCF++ into derivations of PCF+. The translation is defined by recursion on the structure of the derivation trees. Since these are freely generated by the typing rules, it is sufficient to provide for each rule of PCF++ a corresponding rule on trees of PCF+ judgments. For the correspondence which we describe, it is possible to show that these corresponding rules are *directly derivable* in PCF+, therefore the translation takes derivations in PCF++ into derivations in PCF+.

A PCF++ derivation Δ yielding an inheritance judgment $H \vdash e : s$ is translated as a tree $T\Delta$ of PCF+ judgments yielding a *translation* $T^*\Delta$ of the form $H \vdash e^* : s$. All of the rules of PCF++ except the subsumption rule are translated "without change." For example, the axiom $\mathbf{0} : \mathbf{num}$ is translated as itself, whereas the rule

$$\frac{H \vdash e : \mathbf{num}}{H \vdash \mathbf{Succ}(e) : \mathbf{num}}$$

is translated as

$$\frac{H \vdash e^* : \mathbf{num}}{H \vdash \mathbf{Succ}(e^*) : \mathbf{num}}$$

where $H \vdash e^* : \mathbf{num}$ is the root of the translation of the derivation of $H \vdash e : \mathbf{num}$. Only the subsumption rule is altered by the translation. In particular, the rule

$$\frac{H \vdash e : s \qquad s < t}{H \vdash e : t}$$

is translated by the rule

$$\frac{H \vdash e^* : s \qquad \mathsf{coerce}[s < t] : s \to t}{H \vdash \mathsf{coerce}[s < t](e^*) : t}$$

which "makes the implicit coercion explicit."

It is not hard to see that a PCF++ typing judgement may have many different derivations. The reader may wish to look at different possible derivations for the term in the introduction to get a sense of why this is the case. This presents a problem for the translation: is there any sense in which two translations to PCF+ of a given PCF++ term are related? In particular, this paper's main theorem can be used to demonstrate a close relationship between the *operational* semantics of the two translations.

2.3 Operational semantics and Main Theorem.

The operational semantics of the closed raw terms of is given by the least relation $\Downarrow$ between raw terms and canonical forms which satisfies the rules and axioms in Table 3. Canonical froms are defined as follows: **0**, **true**, and **false** are canonical forms. For any expression e, $\lambda x : s.\ e$ is a canonical form. If $c_1, \ldots c_n$ are canonical forms, then $\{l_1 = c_1, \ldots, l_n = c_n\}$ is a canonical form. If c is a canonical form,

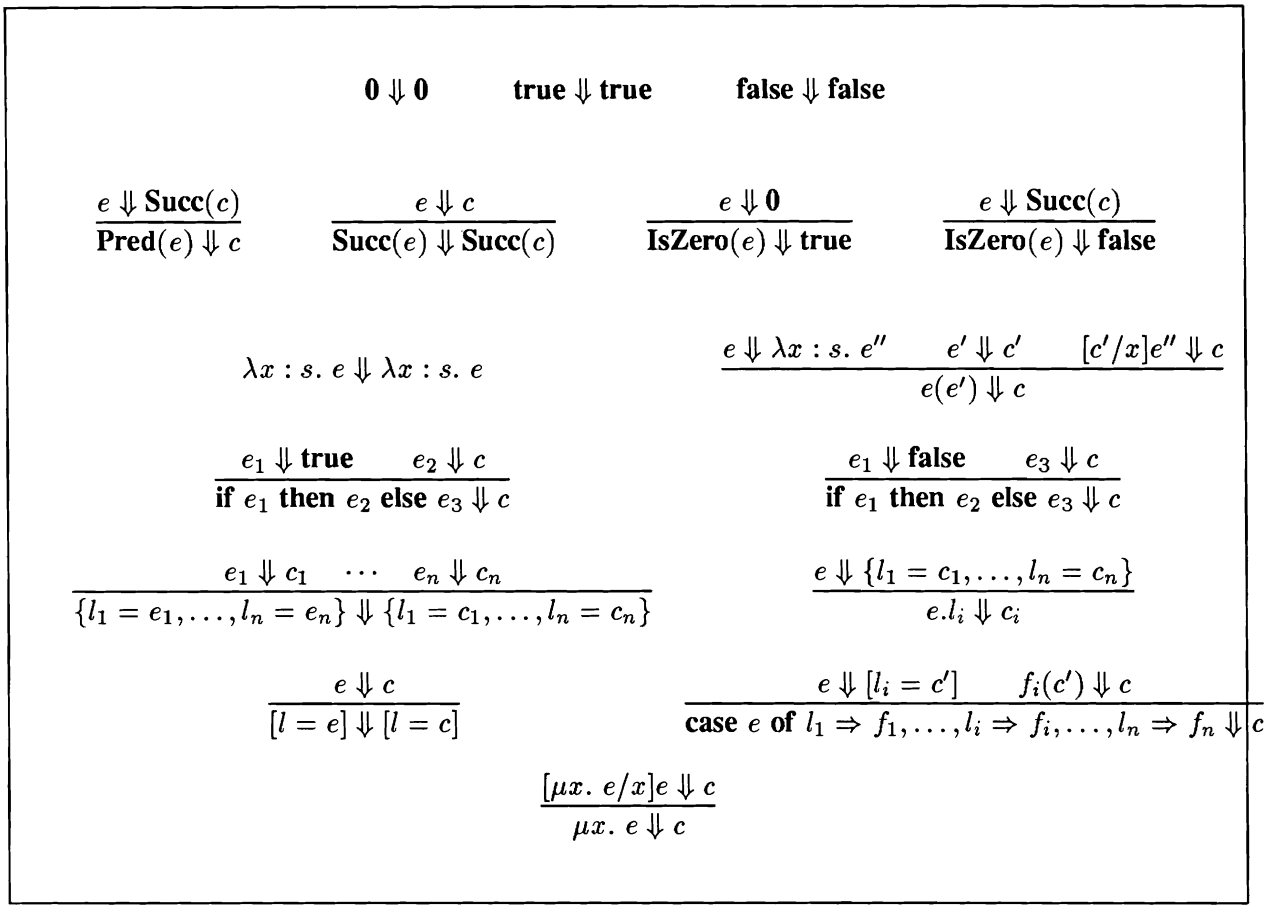

Table 3: Call-by-value evaluation.

then $\mathbf{Succ}(c)$ and $[l = c]$ are canonical forms. We may also write $e \Downarrow$ if there is a canonical form c such that $e \Downarrow c$.

For raw terms e and e' we write $[e'/x]e$ for the result of substituting e' for x in e. We demand all of the usual assumptions about the renaming of bound variables in e to avoid capturing free variables of e'. We assume that the substitution operation associates to the right and we may write $[e_1, \ldots, e_n/x_1, \ldots, x_n]e$ for the simultaneous substitution of $e_1, \ldots, e_n$ for $x_1, \ldots, x_n$ respectively in e. In the event that the terms e_i are closed, note that this is the same as $[e_1/x_1] \cdots [e_n/x_n]e$ and, indeed, the order of the substitutions does not matter.

It is not hard to see that if e is a closed raw term such that $e \Downarrow c$, then c is uniquely determined. This can be proved by showing that, for a given term e, there is at most one axiom or rule from Table 3 which applies to it. Hence the rules define a deterministic evaluation strategy. The evaluation of function application is call-by-value, since the argument to the application is evaluated before being substituted into the body of the applied procedure. There is no evaluation under a lambda-abstraction, but note that records are eagerly evaluated. For example, the evaluation of an expression $\{l = e, l' = e'\}.l$ will result in the evaluation of e' as well as e even though e' is "not needed" in the result. Putting aside efficiency

issues, this is only significant if e' diverges since, in that case, the evaluation of $\{l = e, l' = e'\}.l$ will also diverge. Since evaluation is deterministic, we may define a partial function $\mathcal{E}$ on raw terms as follows

$$\mathcal{E}(e) \simeq \begin{cases} c & e \Downarrow c \text{ if there is such a } c \\ \textit{undefined} & \text{otherwise} \end{cases}$$

We use the symbol $\simeq$ between mathematical expressions to indicate that one of the expressions being related may be undefined. In general, for expressions E and E', $E \simeq E'$ means that if either E or E' is defined, then so is the other and the values are the same.

Let $\vdash e : s$ be a judgement which type-checks in PCF+ and suppose $e \Downarrow c$. It is easy to show that $\vdash c : s$ also type-checks in PCF+. This fact is less obvious for PCF++. We express it in the following:

Lemma 2 *Suppose* $\vdash e : s$ *is derivable in PCF++ and* $e \Downarrow c$, *then* $\vdash c : s$. ∎

This sort of result is closely related to the subject reduction theorems that appear in type theory research.

Let e be raw term such that $\vdash e : s$ is a derivable in PCF++ and suppose e^* is a translation of e into PCF+. Our central question is this: *what, if anything, is the relationship between* $\mathcal{E}(e)$ *and* $\mathcal{E}(e^*)$*?* Naturally, we might start by guessing that $\mathcal{E}(e) \simeq \mathcal{E}(e^*)$ in the sense that when one of them exists, then so does the other, and the results of evaluation are syntactically identical. However, it does not take much looking to see that the syntactic identity may fail in some cases. First of all, if $\mathcal{E}(e)$ is a record, then it may contain some fields which do not appear in the result $\mathcal{E}(e^*)$ of evaluating the coerced term since the latter evaluation will include coercions which may strip various fields in the course of the evaluation of e^*. Moreover, if $\mathcal{E}(e)$ is a lambda term, then $\mathcal{E}(e^*)$ may contain unexecuted coercions in its body which do not appear in $\mathcal{E}(e)$. Worse yet, it seems that even two different translations of $e : s$ may have different canonical forms! Hence we cannot expect a result as simple as the one just proposed and, indeed, we cannot expect a simple-minded statement of an operational coherence result. Nevertheless, there are some obvious counter-observations to the problems just mentioned. In the case of records, the extra fields which appear in $\mathcal{E}(e)$ may be "junk fields" which were not mentioned in the type s. One might argue that it is not even desirable that the result of the evaluation should have fields not included in the specified type s. Could it be that $\mathcal{E}(e)$ and $\mathcal{E}(e^*)$ share "essential" fields in common? Also, the problem with higher types (lambda-abstractions) misses a central point: the "appearance" of a term at non-observable type is not important. Since most interpreters do not display any description of a higher-order procedure, we are interested only in the applicative behavior of such terms in observable contexts. Our goal is therefore to define what we mean by an observable type and define a notion of essential observable equivalence for PCF++ judgements at these types.

Definition: Types **bool** and **num** are *ground* types. A type s is *observable* if

- s is a ground type, or
- $s \equiv \{l_1 : s_1, \ldots, l_n : s_n\}$ where $s_1, \ldots, s_n$ are observable types, or
- $s \equiv [l_1 : s_1, \ldots, l_n : s_n]$ where $s_1, \ldots, s_n$ are observable types. ∎

Definition: The relation $=_s$ between canonical forms of PCF++ observable type s is defined inductively as follows:

- If s is a ground type, then $c =_s c'$ iff $c \equiv c'$.
- Let $s = \{l_1 : s_1, \ldots, l_n : s_n\}$, then

$$\{l_1 = c_1, \ldots, l_n = c_n, \ldots, l_j = c_j\} =_s \{l_1 = c'_1, \ldots, l_n = c'_n, \ldots, l'_k = c'_k\}$$

  iff $c_i =_{s_i} c'_i$ for $i = 1, \ldots, n$.
- Let $s = [l_1 : s_1, \ldots, l_n : s_n]$, then $[l_i = c_i] =_s [l_j = c'_j]$ iff $c_i =_{s_i} c'_j$. ∎

If E and E' are expressions that may be undefined, write $E \simeq_s E'$ to mean that if one expression exists, then so does the other and $E =_s E'$. We may now express the desired result:

> **Main Theorem:** *Suppose* $\vdash e : s$ *is derivable in PCF++ and* e^* *is any PCF+ term which translates this sequent, then* $e \Downarrow$ iff $e^* \Downarrow$. *Moreover, if* s *is observable, then* $\mathcal{E}(e) \simeq_s \mathcal{E}(e')$. ∎

It seems difficult to prove this result directly because of the recursion case. This problem is resolved by appealing to denotational models for PCF+ and PCF++ which we now describe.

3 A computationally adequate denotational semantics.

For technical reasons we have found that it is useful to appeal to some results relating PCF+ and PCF++ to a specific denotational model which we will describe in this section. Although our goal is to prove a purely syntactic result (the Main Theorem at the end of the previous section), the semantic results which we will now establish are of independent interest.

We describe a domain-theoretic model for PCF+. The interpretation of types is as follows:

- $[\![\mathbf{bool}]\!]$ is the flat domain with three distinct elements *tt*, *ff* and least element $\perp$.
- $[\![\mathbf{num}]\!]$ is the flat domain consisting of the numbers $0, 1, 2, \ldots$ together with a least element $\perp$.
- $[\![s \to t]\!] = (s \circ\!\!\to t)_\perp$, the lifted domain of strict (*i.e.* $\perp$-preserving) functions from $[\![s]\!]$ into $[\![t]\!]$.
- $[\![\ \{l_1 : s_1, \ldots, l_n : s_n\}\ ]\!]$ consists of a bottom element $\perp$, together with the set of tuples $\{l_1 = d_1, \ldots, l_n = d_n\}$ where each d_i is a non-bottom element of $[\![s_i]\!]$. The ordering is defined by

$$\{l_1 = d_1, \ldots, l_n = d_n\} \sqsubseteq \{l_1 = d'_1, \ldots, l_n = d'_n\}$$

  iff $d_i \sqsubseteq d'_i$ for each $i = 1, \ldots, n$ and $\perp \sqsubseteq d$ for each record d.
- $[\![\ [l_1 : s_1, \ldots, l_n : s_n]\ ]\!]$ consists of a bottom element $\perp$, together with the set of pairs $[l_i = d_i]$ such that d_i is a non-bottom element of $[\![s_i]\!]$. For two such pairs, $[l_i = d] \sqsubseteq [l_j = d']$ iff $i = j$ and $d \sqsubseteq d'$.

Suppose $H = x_1 : s_1, \ldots x_n : s_n$ is a type context. An *H-environment* is a function which assigns to each variable x_i an element $\rho(x_i)$ of the domain $[\![s_i]\!]$. The PCF+ interpretation of a sequent $H \vdash e : s$ is a function which assigns to each H-environment ρ a value $[\![H \vdash e : s]\!]^{++}\rho$ in $[\![s]\!]$.

We will refrain from writing out all of the semantic equations for the sequents of PCF+. The rules for the introduction and elimination operators for the record and variant types are straightforward, holding in mind that the interpretation of a record with a field which is $\perp$ is itself equal to $\perp$. Recursion is defined in the usual way using least fixedpoints. The function space requires some explanation which we now provide.

The *lift* $D_\perp$ of a domain D is obtained by adding a new bottom element. There is a continuous function $\mathsf{up} : D \to D_\perp$ which sends elements of D to their images in the lifted domain. This function is not strict, since it sends the bottom of D to an element of $D_\perp$ which dominates the "new" bottom element. There is a unique continuous strict function $\mathsf{down} : D_\perp \to D$ such that $(\mathsf{down} \circ \mathsf{up})(x) = x$ for any x and $(\mathsf{up} \circ \mathsf{down})(y) = y$ for any $y \neq \perp$. This equational relationship between the two functions plays an essential role in the computational adequacy result which we will state later. Now, the meaning of a derivable typing judgement of the form $H \vdash \lambda x.\ e : s \to t$ is given as follows:

$$[\![H \vdash \lambda x.\ e : s \to t]\!]^{+}\rho = \mathsf{up}(\mathsf{strict}\lambda d \in [\![s]\!].\ [\![H, x : s \vdash e : t]\!]^{+}\rho[d/x])$$

where $\rho[d/x]$ is the $H, x : s$ environment which is the same as ρ except it sends x to d (we assume that x is a "fresh" variable which does not appear in H) and the second lambda abstraction is the "semantic" notation for a function which takes an argument $d \in [\![s]\!]$. Since the interpretation a function application to a program with value $\perp$ should have value $\perp$ to model call-by-value properly, one must apply the function strict defined as

$$\mathsf{strict}(f)(x) = \begin{cases} f(x) & \text{if } x \neq \perp \\ \perp & \text{if } x = \perp \end{cases}$$

The resulting strict function is lifted by the function up to insure that its value is non-bottom. Again, this will be important later when we prove a correspondence between operational divergence and having $\perp$ as a meaning. Under our intended operational semantics, no lambda-abstraction is a divergent program. The definition of application is given as follows:

$$[\![H \vdash e(e') : t]\!]^{+}\rho = \mathsf{down}([\![H \vdash e : s \to t]\!]^{+}\rho)([\![H \vdash e' : s]\!]\rho)$$

We may now show how our model for PCF+ can be used to construct a model for PCF++. Following ideas from [BCGS89] we use the following Semantic Coherence Theorem due to Rick Blute:

Theorem 3 *(Semantic Coherence) If* Γ *and* Δ *are PCF++ derivations of a sequent* $H \vdash e : s$, *then* $[\![H \vdash T^*(\Gamma) : s]\!]^{++} = [\![H \vdash T^*(\Delta) : s]\!]^{++}$. ∎

A similar result was a central objective of the work in [BCGS89] where the coherence is proved for a class of models of an equational theory. The model here differs from the ones considered there since the unrestricted β rule does not hold in the model we have described in the current paper. The semantic function for sequents of PCF++ is now defined as follows: $[\![H \vdash e : s]\!]^{++}\rho = [\![H \vdash e^* : s]\!]^{+}\rho$ where $H \vdash e^* : s$ is any translation of $H \vdash e : s$. If we note that any PCF+ derivation *is* a PCF++ derivation, then we get the following corollary:

Corollary 4 *If $H \vdash e : s$ is a derivable judgment of PCF+, then $[\![H \vdash e : s]\!]^{+}\rho = [\![H \vdash e : s]\!]^{++}\rho$ for any H-environment ρ.* ∎

As we shall see later, this corollary permits us to transfer some hard-earned results about PCF++ to results about PCF+. In light of the corollary, we may sometimes omit the tags on the semantic brackets for PCF+ derivable typing judgements.

We now wish to show that the semantics for PCF++ which we have just defined is closely related to its operational semantics. Here is our first crucial relationship:

Theorem 5 *(Soundness) If $e : s$ is derivable in PCF++, and $e \Downarrow c$, then $[\![e : s]\!]^{++} = [\![c : s]\!]^{++}$.* ∎

We have omitted the proof, which is straight-forward but tedious. We mention only the following facts which are needed:

Lemma 6 *1. If $r < s < t$, then $[\![\mathsf{coerce}[r < t] : r \to t]\!] = [\![\mathsf{coerce}[s < t] : s \to t]\!] \circ [\![\mathsf{coerce}[r < s] : r \to s]\!]$*

2. If $s < t$, then $[\![\mathsf{coerce}[s < t] : s \to t]\!](d) = \bot$ iff $d = \bot$. ∎

Lemma 7 *If $\vdash c : s$ is a derivable judgement of PCF++ and c is a canonical form, then $[\![c : s]\!]^{++} \neq \bot$.* ∎

Most of the rest of this section is devoted to a proof of a kind of converse to the Soundness Theorem which we will call *computational adequacy* (the term is suggested by Albert Meyer [Mey88], although his definition includes soundness). For PCF++, it can be stated as follows:

Theorem 8 *(Computational Adequacy.) Suppose $e : s$ is derivable in PCF++. If $[\![e : s]\!]^{++} \neq \bot$ then $e \Downarrow c$ for some canonical form c.*

We focus on explaining how the methods that one uses for results such as those above are applied to a calculus with multiple inheritance. We will look at the proof of adequacy in some detail. The proof requires a relation between program meanings and programs sometimes called an "inclusive predicate". We define this relationship as follows:

Definition: Define a relation $\lesssim_s$ between elements of $[\![s]\!]$ on the left and closed raw terms of type s on the right as follows. $d \lesssim_s e$ if $d = \bot$ or $e \Downarrow c$ for some c and $d \lesssim_s c$ where

- $f \lesssim_{s \to t} \lambda x : r.\ e$ iff for each $d \in [\![s]\!]$ and term c, $d \lesssim_s c$ implies $\mathsf{down}(f)(d) \lesssim_t [c/x]e$.
- $\{l_1 = d_1, \ldots, l_n = d_n\} \lesssim_{\{l_1:s_1,\ldots,l_n:s_n\}} \{l_1 = e_1, \ldots, l_m = e_m\}$ iff $m \geq n$ and $d_i \lesssim_{s_i} c_i$ for $i = 1, \ldots n$.
- $[l_i = d] \lesssim_{[l_1:s_1,\ldots,l_n:s_n]} [l_j = c]$ iff $i = j$ and $d \lesssim_{s_i} c$.
- $tt \lesssim_{\mathbf{bool}} \mathbf{true}$ and $ff \lesssim_{\mathbf{bool}} \mathbf{false}$.
- $0 \lesssim_{\mathbf{num}} \mathbf{0}$ and if $n \lesssim_{\mathbf{num}} c$ for a number n, then $n + 1 \lesssim_{\mathbf{num}} \mathbf{Succ}(c)$. ∎

Some of the essential semantic properties of $\precsim$ are given in the following:

Lemma 9 *1. If $a \sqsubseteq b \precsim_s e$, then $a \precsim_s e$.*

2. If $a_0 \sqsubseteq a_1 \sqsubseteq a_2 \sqsubseteq \cdots$ is an ascending chain and $a_n \precsim_s e$ for each n, then $\bigsqcup_{n=0}^{\infty} a_n \precsim_s e$. ∎

We are now ready to sketch the proof of the primary technical lemma which is needed for the proof of PCF++ adequacy.

Lemma 10 *Suppose $H = x_1 : s_1^\dagger \ldots x_n : s_n^\dagger$ and $H \vdash e^\dagger : s^\dagger$ is derivable. If $d_i \in [\![s_i^\dagger]\!]$ and $d_i \precsim_{s_i^\dagger} e_i^\dagger$ for $i = 1, \ldots, k$, then $[\![H \vdash e^\dagger : s^\dagger]\!]^{++}[d_1, \ldots, d_k/x_1, \ldots, x_k] \precsim_{s^\dagger} [e_1^\dagger, \ldots, e_k^\dagger/x_1, \ldots, x_k]e^\dagger$.*

Proof: Let ρ be the environment $[d_1, \ldots, d_n/x_1, \ldots x_n]$ and σ be the substitution $[e_1^\dagger, \ldots, e_n^\dagger/x_1, \ldots, x_n]$. Let Δ be a PCF++ derivation of the typing judgement $H \vdash e^\dagger : s^\dagger$. We prove that $[\![H \vdash e^\dagger : s^\dagger]\!]^{++}\rho \precsim_{s^\dagger} \sigma e^\dagger$ by an induction on Δ. Assume that the Theorem is known for proofs of lesser height. There are eleven possibilities for the last step of Δ. Some of the more interesting cases (subsumption in particular) are written out fully below.

- *Base case:* $H \vdash x_i : s_i^\dagger$.
  Suppose the sequent $H \vdash e^\dagger : s^\dagger$ is an axiom of the form above (*i.e.* $e^\dagger \equiv x_i$). Then we have $[\![H \vdash x_i : s_i^\dagger]\!]^{++}\rho = d_i \precsim_{s_i^\dagger} e_i \equiv \sigma x_i$ by assumption.

- *Lambda abstraction:* $\dfrac{H,\ x : s \vdash e : t}{H \vdash \lambda x : s.\ e : s \to t}$
  Suppose the last inference in Δ has the form above (in particular, $e^\dagger : s^\dagger \equiv \lambda x : s.\ e : s \to t$). Let Δ' be part of the proof Δ which proves $H,\ x : s \vdash e : t$ and suppose $H,\ x : s \vdash e^* : t$ is $T^*\Delta'$. Let $f = [\![H \vdash \lambda x : s.\ e : s \to t]\!]^{++}\rho = [\![H \vdash \lambda x : s.\ e^* : s \to t]\!]^{+}\rho$ and suppose $d \precsim_s c$. We must show that
  $$d' = \mathsf{down}(f)(d) \precsim_t (\sigma\lambda x : r.\ e)(c) \tag{1}$$
  If $d' = \bot$ then there is no problem. Suppose $d' \neq \bot$, then $\mathsf{down}(f) \neq \bot$, so
  $$\begin{aligned} d' &= \mathsf{down}(\mathsf{up}(\mathsf{strict}\lambda d'' \in [\![s]\!].\ [\![H, x : s \vdash e^* : t]\!]^{+}\rho[d''/x]))(d) \\ &= (\mathsf{strict}\lambda d'' \in [\![s]\!].\ [\![H, x : s \vdash e^* : t]\!]^{+}\rho[d''/x])(d) \end{aligned}$$
  and there are two cases. if $d = \bot$, then $d' = \bot \precsim_t \sigma[c/x]e$ as desired. However, if $d \neq \bot$, then
  $$\begin{aligned} d' &= [\![H, x : s \vdash e^* : t]\!]^{+}\rho[d/x] \\ &= [\![H, x : s \vdash e : t]\!]^{++}\rho[d/x] \\ &\precsim_t \sigma[c/x]e \end{aligned}$$
  by the induction hypothesis. Since $d' \neq \bot$, there is a canonical c' such that $\sigma[c/x]e \Downarrow c'$ and $d' \precsim_t c'$. Since $[c/x]\sigma e \equiv \sigma[c/x]e$, we have $(\lambda x : s.\ \sigma e)(c) \equiv (\sigma\lambda x : s.\ e)(c)$ so it must be the case that $(\sigma\lambda x : s.\ e)(c) \Downarrow c'$ too, so 1 holds.

- *Application:* $\dfrac{H \vdash e_1 : s \to t \qquad H \vdash e_2 : s}{H \vdash e_1(e_2) : t}$

  Let $H \vdash e_1^* : s \to t$ and $H \vdash e_2^* : s$ be translations dictated by Δ. If $d' = [\![H \vdash e_1(e_2) : t]\!]^{++}\rho = [\![H \vdash e_1^*(e_2^*) : t]\!]^{+}\rho \neq \bot$, then $f = [\![H \vdash e_1 : s \to t]\!]^{++}\rho = [\![H \vdash e_1^* : s \to t]\!]^{+}\rho \neq \bot$ and $d = [\![H \vdash e_2 : s]\!]^{++}\rho = [\![H \vdash e_2^* : s]\!]^{+} \neq \bot$ (using the fact that all our functions are strict!). By the induction hypothesis, $f \lesssim_{s\to t} \sigma e_1 : s \to t$ and $d \lesssim_s \sigma e_2 : s$, so there is a term e_3 and a canonical form c such that

  $$\sigma e_1 \Downarrow \lambda x : s.\ e_3 \text{ and } f \lesssim_{s\to t} \lambda x : s.\ e_3$$
  $$\sigma e_2 \Downarrow c \text{ and } d \lesssim_s c$$

  Now $d' = \mathsf{down}(f)(d) \lesssim_t [c/x]e_3$ by the definition of $\lesssim_{s\to t}$, so there is a canonical c' such that $[c/x]e_3 \Downarrow c'$ and $d' \lesssim_t c'$. But $[c/x]e_3 \Downarrow c'$ means $(\sigma e_1)(\sigma e_2) \Downarrow c'$. Since $(\sigma e_1)(\sigma e_2) \equiv \sigma(e_1(e_2))$, we have $d' \lesssim_t \sigma(e_1(e_2))$ as desired.

- *Recursion:* $\dfrac{H,\ x : s \vdash e : s}{H \vdash \mu x : s.\ e : s}$.

  Let $H,\ x : s \vdash e^* : s$ be the translation dictated by Δ. Let $d_0 = \bot$ and $d_{i+1} = [\![H, x : s \vdash e : s]\!]^{++}\rho[d_i/x] = [\![H, x : s \vdash e^* : s]\!]^{+}\rho[d_i/x]$. We show that $d_i \lesssim_s \sigma\mu x : s.\ e$ for each i. This is immediate for $d_0 = \bot$. Suppose $d_i \lesssim_s \sigma\mu x : s.\ e$. By induction hypothesis, $d_{i+1} = [\![H, x : s \vdash e : s]\!]^{++}\rho[d_i/x] \lesssim_s \sigma[\mu x : s.e/x]e$. If $d_{i+1} \neq \bot$, then $\sigma[\mu x : s.\ e/x]e \Downarrow c$ for some c such that $d_{i+1} \lesssim_s c$. Now $\sigma[\mu x : s.\ e/x]e \equiv [\sigma\mu x : s.\ e/x]\sigma e \equiv [\mu x : s.\ \sigma e/x]\sigma e$. Hence $\sigma\mu x : s.\ e \equiv \mu x : s.\ \sigma e \Downarrow c$ as well. Since $[\![H \vdash \mu x : s.\ e : s]\!]^{++}\rho = [\![H \vdash \mu x : s.\ e^* : s]\!]^{+}\rho = \bigsqcup_{i=0}^{\infty} d_i$, the desired result follows.

- *Subsumption rule:* $\dfrac{H \vdash e : s \qquad s < t}{H \vdash e : t}$.

  The proof for this case is by induction on the height of the proof that $s < t$. Assume that we know that the theorem holds for $H \vdash e : s$ and let $H \vdash e^* : s$ be any translation of this sequent to PCF+. There are four subcases:

  - *Base types:* These are both obvious since the coercion is the identity map.
  - *Functions:* $\dfrac{u' < u \qquad v < v'}{u \to v < u' \to v'}$.

    Suppose $s \equiv u \to v$ and $t \equiv u' \to v'$. Let $\xi_1 = \mathsf{down}[\![\mathsf{coerce}[u' < u]]\!]$ and $\xi_2 = \mathsf{down}[\![\mathsf{coerce}[v < v']]\!]$. Then $\xi = \mathsf{down}[\![\mathsf{coerce}[u \to v < u' \to v']]\!]$ satisfies $\xi(f) = \xi_2 \circ f \circ \xi_1$ for $f : [\![u]\!] \circ\!\!\to [\![v]\!]$. Set $f = \mathsf{down}[\![H \vdash e : s]\!]^{++}\rho$. If $d \lesssim_{u'} c$, then $\xi_2(d) \lesssim_u c$ by induction hypothesis on $u' < u$. Thus $f(\xi_2(d)) \lesssim_v (\sigma e)(c)$ by induction hypothesis on $H \vdash e : s$. We may now apply the induction hypothesis on $v < v'$ to conclude that $\xi(f) = \xi_1(f(\xi_2(d))) \lesssim_{v'} (\sigma e)(c)$. Since $\xi(f) = [\![H \vdash e : t]\!]^{++}\rho$ we conclude that $[\![H \vdash e : t]\!]^{++}\rho \lesssim_t \sigma e$.
  - *Records:* $\dfrac{s_1 < t_1 \quad \cdots \quad s_n < t_n}{\{l_1 : s_1, \ldots, l_n : s_n, \ldots, l_m : s_m\} < \{l_1 : t_1, \ldots, l_n : t_n\}}$.

    Let $\xi_i = \mathsf{down}[\![\mathsf{coerce}[s_i < t_i]]\!]$ for $i = 1, \ldots n$ and let $\xi = \mathsf{down}[\![\mathsf{coerce}[s < t]]\!]$. By induction hypothesis, we have $d = [\![H \vdash e : s]\!]^{++}\rho \lesssim_s \sigma e$. If $d = \bot$, then $\xi(d) = [\![H \vdash e : t]\!]^{++}\rho = \bot$ and we are done. If $d \neq \bot$, then $d = \{l_1 = d_1, \ldots, l_m = d_m\}$ where $d_1, \ldots, d_m \neq \bot$ and $\sigma e \Downarrow c$ for some canonical c of the form $c \equiv \{l_1 = c_1, \ldots, l_j = c_j\}$

such that $j \geq m$ and $d_i \lesssim_{s_i} c_i$ for $i = 1, \ldots m$. By the induction hypothesis on inheritance judgements, we must therefore have $\xi_i(d_i) \lesssim_{t_i} c_i$ for each $i = 1, \ldots, n$. Hence $\xi(d) = \{l_1 = \xi_1(d_1), \ldots, l_n = \xi_n(d_n)\} \lesssim_t \{l_1 = c_1, \ldots, l_j = c_j\}$ by the definition of $\lesssim_t$ and we are done.

- *Variants:* $\dfrac{s_1 < t_1 \quad \cdots \quad s_n < t_n}{[l_1 : s_1, \ldots, l_n : s_n] < [l_1 : t_1, \ldots, l_n : t_n, \ldots, l_m : t_m]}$.

  Let $\xi_i = \mathsf{down}[\![\mathsf{coerce}[s_i < t_i]]\!]$ for $i = 1, \ldots n$ and let $\xi = \mathsf{down}[\![\mathsf{coerce}[s < t]]\!]$. By induction hypothesis, we have $d = [\![H \vdash e : s]\!]^{++}\rho \lesssim_s \sigma e$. If $d = \perp$, then $\xi(d) = [\![H \vdash e : t]\!]^{++}\rho = \perp$ and we are done. If $d \neq \perp$, then $d = [l_i = d_i]$ where $d_i \neq \perp$ and $\sigma e \Downarrow c$ where $\sigma e \Downarrow c$ and $d \lesssim_s c$. By the definition of $\lesssim_s$, the term c has the form $[l_i = c_i]$ and $d_i \lesssim_{s_i} c_i$. By induction hypothesis on $s_i < t_i$, we know that $\xi_i(d_i) \lesssim_{t_i} c_i$ so $\xi(d) = [l_i = \xi_i(d)] \lesssim_t [l_i = c_i]$. ∎

We may now express the desired proof of Computational Adequacy for PCF++.

Proof: (of Theorem 8) By Lemma 10 we know that $[\![e : s]\!]^{++} \lesssim_s e$. Since the value on the left is assumed to differ from $\perp$, the Theorem follows immediately from the definition of $\lesssim_s$. ∎

The following theorem follows immediately from Soundness and Computational Adequacy for PCF++ together with Corollary 4 of the Semantic Coherence Theorem for PCF++.

Theorem 11 *(Soundness and Adequacy for PCF+) If* $\vdash e : s$ *is derivable in PCF+, then*

1. *(Soundness)* $e \Downarrow c$ *implies* $[\![e : s]\!] = [\![c : s]\!]$.
2. *(Computational Adequacy)* $[\![e : s]\!]^{+} \neq \perp$ *implies* $e \Downarrow c$ *for some canonical form* c. ∎

The following lemma is needed for the proof of the Main Theorem:

Lemma 12 *Let* c *and* c' *be canonical forms such that* $\vdash c : s$ *and* $\vdash c' : s$ *are derivable in PCF++ for an observable type* s. *Then* $[\![c : s]\!]^{++} = [\![c' : s]\!]^{++}$ *iff* $c =_s c'$. ∎

Corollary 13 *Let* e *and* e' *be raw terms such that* $\vdash e : s$ *and* $\vdash e' : s$ *are derivable for an observable type* s. *Then* $[\![e : s]\!]^{++} = [\![e' : s]\!]^{++}$ *iff* $\mathcal{E}(e) \simeq_s \mathcal{E}(e')$.

Proof: This follows from Adequacy, Soundness and Lemma 12. ∎

Main Theorem: Suppose $\vdash e : s$ is derivable in PCF++ and e^* is any PCF+ term which translates this sequent, then $e \Downarrow$ iff $e^* \Downarrow$. Moreover, if s is observable, then $\mathcal{E}(e) \simeq_s \mathcal{E}(e')$. ∎

Proof: Suppose $e \Downarrow c$. Then $[\![e : s]\!]^{++} = [\![c : s]\!]^{++} \neq \perp$ by the Soundness Theorem for PCF++ and Lemma 7. Since $[\![e : s]\!]^{++} = [\![e^* : s]\!]^{+}$, we may conclude from Soundness and Adequacy for PCF+ that there is a canonical form c' such that $e^* \Downarrow c'$ and $[\![c' : s]\!]^{++} = [\![c : s]\!]^{++}$. If s is an observable type then $c =_s c'$ by Lemma 12.

Suppose conversely that $e^* \Downarrow c'$ for some canonical c'. By the Soundness for PCF+, $[\![e^* : s]\!]^{+} = [\![c' : s]\!]^{+} \neq \perp$. Hence $[\![e : s]\!]^{++} = [\![e^* : s]\!]^{+} \neq \perp$. By Adequacy and Soundness for PCF++, there is a canonical form c such that $e \Downarrow c$ and $[\![c : s]\!]^{++} = [\![c' : s]\!]^{++}$. Thus $c =_s c'$ by Lemma 12. ∎

4 Conclusions and directions for further research.

We have shown that inheritance-interpreted-as-definable-coercion semantic paradigm behaves well with respect to operational semantics. More specifically, we have shown that the coercion terms that we introduce in this interpretation, while possibly generating more computation, will only generate "harmless" computation, in particular that no unexpected divergence can be introduced, nor can expected divergence be lost. (In the process, we actually exhibited a nice domain-theoretic model which is sound and computationally adequate for PCF++'s straightforward operational semantics.)

There are at least two points where we can see improvements to our results. One problem is that we would like to strenghthen the main theorem so as to say something interesting about the relationship between $\mathcal{E}(e)$ and $\mathcal{E}(e^*)$ when their type is not necessarily observable. The other problem is that the proof of Theorem 5 does not really use the particularities of the denotational semantics but rather the fact that certain identities between PCF+ terms hold in it. These two points are related and here is a conjectured improvement which would solve both problems.

Suppose $\vdash e : s$ type-checks in PCF++ and let e^* be any PCF+ translation of it. Further suppose that $e \Downarrow c$ for some canonical form c. By our main theorem, there is a canonical form c' such that $e^* \Downarrow c'$. We would like to relate c and c' as PCF+ terms, but $\vdash c : s$ may type-check in PCF++ only. So, let c^* be any PCF+ translation of it. What do we know about the relationship between c^* and c'? It is a consequence of the soundness results that the model we introduce in section 3 equates them. But equality in this model is Π^0_2-hard. Surely the relationship between c^* and c' is much simpler...

We believe that it is possible to formulate a reasonable logical theory about PCF+ terms, call it $\mathcal{T}$ in which c^* and c' can be shown to be *provably equal*. In fact, we believe that such a theory would be closely related to the call-by-value lambda-calculi studied by Plotkin and Moggi [Mog88]. This result would have the following pleasant corollaries. Let $\mathcal{D}^+$ be any denotational model of PCF+ in which the operational semantics and the axiomatization of $\mathcal{T}$ are sound (actually, we expect that the soundness of the later will imply that of the former). One immediately concludes that our translation is denotationally coherent with respect to $\mathcal{D}^+$, which induces a model $\mathcal{D}^{++}$ of PCF++, and that the operational semantics of PCF++ terms is sound in $\mathcal{D}^{++}$. Of course, by the main theorem of this paper, we can also get transfer of computational adequacy. Therefore, we would be able to neatly concentrate in the axiomatization of $\mathcal{T}$ all the conditions needed by a "good" model of PCF+ in order to become a model of PCF++ in accordance to our paradigm.

An intriguing question is whether $c^* = c'$ will turn out to be more than an r.e. statement, whether it is actually decidable? In other words, is full PCF+ computation required in order to systematically disentangle the coercions we introduce?

Finally, we should restate that we expect that the results of this paper generalize to more complicated type disciplines (Fun, Quest, *etc.*) and that analogs can be shown for call-by-name operational semantics.

5 Acknowledgements.

Breazu-Tannen's research was partially supported by ONR Grant NOOO14-88-K-0634. Gunter's research was partially supported by ONR Grant NOOO14-88-K-0557. Breazu-Tannen and Gunter were also partially supported by ARO Grant DAAG29-84-K-0061. Scedrov's research was partially supported by NSF Grant CCR-87-05596, by ONR Grant NOOO14-88-K-0635, and by the 1987 Young Scientist Award from the Natural Sciences Association of the University of Pennsylvania.

References

[BCGS89] V. Breazu-Tannen, T. Coquand, C. A. Gunter, and A. Scedrov. Inheritance and explict coercion (preliminary report). In R. Parikh, editor, *Logic in Computer Science*, pages 112–134, IEEE Computer Society, June 1989.

[BCGS90] V. Breazu-Tannen, T. Coquand, C. A. Gunter, and A. Scedrov. Inheritance as implicit coercion. *Information and Computation.*, ??:??–??, 1990. To appear.

[Car89] L. Cardelli. *Typeful programming.* Research Report 45, DEC Systems, Palo Alto, May 1989.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

[Mey88] A. R. Meyer. Semantical paradigms: notes for an invited lecture. In Y. Gurevich, editor, *Logic in Computer Science*, pages 236–253, IEEE Computer Society, July 1988.

[Mog88] E. Moggi. *The Partial Lambda-Calculus.* PhD thesis, University of Edinburgh, 1988.

[OBB89] A. Ohori, P. Buneman, and V. Breazu-Tannen. Datbase programming in Machiavelli—a polymorphic language with static type inference. In *SIGMOD Conference on the Management of Data*, pages 46–57, ACM, 1989.

[Plo77] G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

[Wan89] M. Wand. Type inference for record concatenation and multiple inheritance. In *Proceedings of the Symposium on Logic in Computer Science*, pages 92–97, IEEE, June 1989.