Fully Abstract Translations between Functional Languages (Preliminary Report)

Jon G. Riecke*
MIT Laboratory for Computer Science

Abstract

We address the problem of finding fully abstract translations between programming languages, *i.e.*, translations that preserve code equivalence and nonequivalence. We give three fully abstract translations between versions of PCF: one from call-by-name [14, 21] to call-by-value [18, 22, 23], one from call-by-value to lazy [4, 5], and one from lazy to call-by-value. The translations yield lower bounds on decision procedures for proving equivalences of code. We finally give a definition of "functional translation" and show that no such translation exists from call-by-value to call-by-name, or from lazy to call-by-name.

1 Introduction

There are many ways to compare the expressive power of programming languages. For instance, the relative time efficiency of compiled programs might be one basis for comparing two languages. Another criterion might be the constructs provided by the programming languages: language A is more powerful than language B if language A can define all of the operators of language B. (This idea of "definable operators" is explored in [6].) We might also say that language A is more powerful than language B if it can type-check more expressions. Of course, we would not want to define expressive power based on recursion theory, since most languages compute precisely the partial recursive functions over, say, integers.

Here we explore another notion of expressiveness: whether a language can be *translated* into another language. A translation is a meaning-preserving, syntactically-defined map that transforms programs of language A into programs of language B. Translations generalize the notion of definable operators: the difference is that entire programs—not just subexpressions—are mapped to programs in the target language in some uniform manner.

To keep the study focused, we consider translations between various versions of the simply-typed, functional language PCF [14, 21]. A full definition of the syntax of PCF appears in Section 2 along with the specification of the three interpreters, call-by-name, call-by-value, and lazy. The word "lazy" does *not* refer to infinite lists; here, a lazy language is a call-by-name language in which the programmer can observe higher-order termination to λ-abstractions as well as evaluation to ground constants [1, 5, 11, 12].

One translation from call-by-value to lazy PCF may found in [12]. The idea behind this translation is familiar and simple: since the call-by-value interpreter evaluates operands in function applications—in essence, forcing all functions to be strict—translated terms use the constants $\mathsf{conv}^{\sigma,\tau}$ of lazy PCF to ensure that the evaluation of an operand terminates. The operator $\mathsf{conv}^{\sigma,\tau}$ of type $\sigma \to \tau \to \tau$ forces reduction of its first argument; if this reduction halts at either a λ-abstraction or constant, the operator returns its second argument, otherwise it diverges [1, 4, 5, 11, 12]. The translation is defined by structural induction:

$$\begin{aligned} \overline{x^\sigma} &= x^\sigma \\ \overline{\lambda x^\sigma.M} &= \lambda x^\sigma.(\mathsf{conv}^{\sigma,\tau}\ x\ \overline{M}), \quad M \text{ has type } \tau \\ \overline{(M\ N)} &= (\overline{M}\ \overline{N}) \end{aligned}$$

The translation of constants and conditionals may be found in Section 3.1.

In what sense should a translation preserve the meaning of code? At the very least, we expect a translated program to produce the same observable outcomes as the original code.

Definition 1 *A translation $M \mapsto \widetilde{M}$ from $\mathcal{L}_1$ to $\mathcal{L}_2$ is* **adequate** *if M produces an observable in $\mathcal{O}$ under $\mathcal{L}_1$ iff $\widetilde{M}$ produces the same observable under $\mathcal{L}_2$.*

*Supported by an NSF Graduate Fellowship, NSF Grant Nos. 8511190-DCR and 8819761-CCR, and ONR grant No. N00014-83-K-0125.

The translation defined above is adequate: M halts under the call-by-value interpreter iff $\overline{M}$ halts under the lazy interpreter, and moreover, M call-by-value evaluates to a numeral n iff $\overline{M}$ lazy evaluates to n [12].

We can ask for a stronger condition and require that a translation preserve equivalences of arbitrary pieces of code (*e.g.*, code in which functions or procedures have not yet been declared). By "equivalence" we mean observational congruence.

Definition 2 *A term M* **observationally approximates** *a term N with respect to observables $\mathcal{O}$ and language $\mathcal{L}$ (written $M \sqsubseteq^{\mathcal{O}}_{\mathcal{L}} N$) if, for any $\mathcal{L}$-context $C[\cdot]$, $C[M]$ yields an observable outcome implies that $C[N]$ yields the same outcome. Two terms M and N are* **observationally congruent** *(written $M \equiv^{\mathcal{O}}_{\mathcal{L}} N$) if both $M \sqsubseteq^{\mathcal{O}}_{\mathcal{L}} N$ and $N \sqsubseteq^{\mathcal{O}}_{\mathcal{L}} M$.*

A translation will be called fully abstract if it preserves observational approximations (*cf.* [14]):

Definition 3 *Let $\mathcal{O}_1$ and $\mathcal{O}_2$ be the observables of $\mathcal{L}_1$ and $\mathcal{L}_2$. A translation $P \mapsto \widetilde{P}$ from $\mathcal{L}_1$ to $\mathcal{L}_2$ is* **fully abstract** *if $\mathcal{O}_1 \subseteq \mathcal{O}_2$ and*

$$M \sqsubseteq^{\mathcal{O}_1}_{\mathcal{L}_1} N \iff \widetilde{M} \sqsubseteq^{\mathcal{O}_2}_{\mathcal{L}_2} \widetilde{N}.$$

A simple example shows that the translation above is *not* fully abstract. Consider the terms $M_1 = \lambda x^{\iota\to\iota}.x$ and $M_2 = \lambda x^{\iota\to\iota}.\lambda y^{\iota}.(x\ y)$, where ι is the base type of numerals. If we observe numerals and termination to λ-abstractions or constants (the observations *termin*) under the call-by-value interpreter, $M_1 \equiv^{termin}_{val} M_2$. But the terms

$$\begin{aligned} \overline{M_1} &= \lambda x^{\iota\to\iota}.\mathsf{conv}\ x\ x \\ \overline{M_2} &= \lambda x^{\iota\to\iota}.\mathsf{conv}\ x\ (\lambda y^{\iota}.\mathsf{conv}\ y\ (x\ y)) \end{aligned}$$

are *not* lazy observationally congruent: the context

$$C[\cdot] = [\cdot]\ (\lambda z.3)\ \Omega$$

(where Ω is any divergent term of type ι) causes the first term to return 3 and the second to diverge under the lazy interpreter. Similar problems arise in other translations, *e.g.*, continuation-passing transforms, while adequate, fail to be fully abstract [7, 13].

Fully abstract translations are important for a number of reasons. First, fully abstract translations can be used to translate questions about code equivalence or nonequivalence from one language to another. For example, if there is an effective means of proving observational congruences in language B and we have an effective, fully abstract translation from language A to language B, then we have an effective proof procedure for observational congruences in language A: first translate terms and then reason about them. Moreover, if the translation is time-bounded, we may be able to deduce lower and upper bounds on decision procedures for proving equivalences.

Second, the concept of fully abstract translations yields a notion of expressiveness: language A is "no more expressive" than B if there is a fully abstract translation from A to B. This idea is not new; Mitchell [9] uses the idea of "well-structured," fully abstract translations to compare languages. Obtaining a definition of "well-structured" translation seems difficult, but we will make an attempt at such a definition in Section 4.

It turns out that the translation from call-by-value PCF to lazy PCF *can* be made fully abstract. The example above provides the clue for fixing the translation. Intuitively, $\overline{M_1}$ and $\overline{M_2}$ are distinguished precisely because variables in lazy terms range over non-strict as well as strict functions. The variable x in $\overline{M_1}$ can be instantiated with *any* lazy function; the term $\overline{M_2}$, on the other hand, forces x to be strict in its first argument.

The key to repairing the translation is to force *all* variables of functional type to be strict. In Section 3 we show how to define a lazy PCF term $\delta^{\sigma\to\tau}$ that, when given a function f of type $\sigma \to \tau$, returns the strict version f. Applying this term to variables will make the translation above fully abstract. The proof of full abstraction relies upon logical relations built between fully abstract models, a technique also developed independently by Sieber for obtaining full abstraction theorems from others [22].

Section 3 describes the translation from call-by-value to lazy PCF in greater detail and sketches the proof of full abstraction. Section 3 also gives fully abstract translations from call-by-name PCF to call-by-value PCF, and from lazy PCF to call-by-value PCF. These translations rely upon definable retractions, and the proofs of full abstraction use the same basic technique as the call-by-value to lazy case. Section 3 finally discusses corollaries to the full abstraction theorems regarding the complexity of proving observational congruences in fragments of the languages.

Other effective, fully abstract translations based on Gödel numberings of terms can be given. Section 4 defines the notion of a **functional translation** (based on logical relations) that eliminates such Gödel numbering translations from consideration. We then show that call-by-value and lazy PCF cannot be translated into call-by-name PCF via a functional translation. This is evidence that the notion of a "functional translation" leads to a nontrivial expressiveness theory. Section 5 concludes the paper with a discussion of some open problems.

2 Preliminaries

2.1 Syntax of Language

Simple types range over ι (the base type of nonnegative integers) and $\sigma \to \tau$ (functions from σ to τ). As is standard, $\to$ associates to the right; *e.g.*, $\iota \to \iota \to \iota$ is shorthand for $\iota \to (\iota \to \iota)$.

The set of PCF terms, with their associated types, is given by the following inductive definition (*cf.* [14, 21]):

- x^σ is a variable and a term of type σ;
- $0, 1, 2, \ldots$ are constants of type ι;
- succ, pred are constants of type $\iota \to \iota$;
- Y^σ is a constant of type $(\sigma \to \sigma) \to \sigma$, where $\sigma \neq \iota$;
- $\lambda x^\sigma.M$ is a term of type $\sigma \to \tau$ if M has type τ;
- $(M\ N)$ is a term of type τ if M has type $\sigma \to \tau$ and N has type σ;
- cond $M\ N\ P$ is a term of type σ if M has type ι and N and P have type σ;
- pcond $M\ N\ P$ is a term of type ι if M, N, and P have type ι.

Here, pcond is parallel conditional, a constructor that is needed to make standard models of the language fully abstract [4, 14, 19, 23]. Also note that the type of Y is restricted; this technical restriction makes the call-by-value interpreter easier to define [18, 23].

A term is a **value** if it is either a variable, constant, or λ-abstraction. To simplify the exposition, values are denoted by V and arbitrary terms are denoted by M, N, P, and Q.

2.2 Three Operational Semantics

We define three interpreters, call-by-name, call-by-value, and lazy, via three binary relations $\Downarrow_n$, $\Downarrow_v$, and $\Downarrow_l$. The (generic) notation $M \Downarrow N$ is read "M halts with answer N." We write $M \Downarrow$ when there is some N with $M \Downarrow N$, and $M \Uparrow$ otherwise.

The three languages exhibit the same behavior when interpreting conditionals and applications of constants (except for Y) to terms. Figure 1 contains rules for reducing constants and conditionals to canonical forms. Additional rules for the three interpreters appear in Figure 2.

The definition of the **call-by-name** interpreter requires only two more rules: the usual β-reduction rule for applying λ-abstractions to terms, and the rule for applying the fixpoint combinator Y to a term. Under call-by-name, numerals are the only observable outcomes of computations [14]:

Definition 4 *$M \sqsubseteq^{num}_{name} N$ if for any PCF context $C[\cdot]$ and numeral k, $C[M] \Downarrow_n k$ implies $C[N] \Downarrow_n k$.*

The **lazy** interpreter also passes arguments by name; the main difference between the call-by-name and lazy interpreters lies in the *observations* one makes about code. In addition to numerals, one may observe in lazy PCF whether a term of functional type halts at a λ-abstraction or constant. These observations are called the *termin* observations.

Lazy PCF also differs from call-by-name PCF in its constants. In order to give the programmer the ability to test the termination behavior, we add convergence-testing constants $\mathsf{conv}^{\sigma,\tau}$ to PCF and call the resulting language LPCF. The observational approximation relation for LPCF is defined as follows:

Definition 5 *$M \sqsubseteq^{termin}_{lazy} N$ if for any LPCF context $C[\cdot]$,*

1. *$C[M] \Downarrow_l$ implies $C[N] \Downarrow_l$; and*
2. *$C[M] \Downarrow_l k$ implies $C[N] \Downarrow_l k$, for k a numeral.*

Call-by-value PCF, in contrast, requires that all arguments in applications be reduced to values. Evaluating applications of Y also differs from the call-by-name and lazy cases for technical reasons. As in the lazy case, we observe both termination and numerals:

Definition 6 *$M \sqsubseteq^{termin}_{val} N$ if for any PCF context $C[\cdot]$,*

1. *$C[M] \Downarrow_v$ implies $C[N] \Downarrow_v$ and*
2. *$C[M] \Downarrow_v k$ implies $C[N] \Downarrow_v k$, for k a numeral.*

We need not add the constants $\mathsf{conv}^{\sigma,\tau}$ to call-by-value, since they are definable by $\lambda x^\sigma.\lambda y^\tau.y$.

3 Examples of Fully Abstract Translations

3.1 Call-by-value to Lazy

Figure 3 describes the complete translation of call-by-value to lazy PCF, proceeding by induction on the structure of terms. In order to achieve full abstraction, the translation forces variables—and hence any lazy function that instantiates a variable—to be strict using the auxiliary functions $\delta^{\sigma\to\tau}$. Informally, $\delta^{\sigma\to\tau}$ checks its second argument y to make sure it halts; the strict version of y is passed to x and the result is made strict.

$$V \Downarrow V, \quad V \text{ a value}$$

$$\frac{M \Downarrow \mathsf{succ} \qquad N \Downarrow n}{(M\ N) \Downarrow (n+1)} \qquad \frac{M \Downarrow \mathsf{pred} \qquad N \Downarrow (n+1)}{(M\ N) \Downarrow n}$$

$$\frac{M \Downarrow 0 \qquad N \Downarrow k}{\mathsf{pcond}\ M\ N\ P \Downarrow k} \qquad \frac{M \Downarrow \mathsf{pred} \qquad N \Downarrow 0}{(M\ N) \Downarrow 0}$$

$$\frac{M \Downarrow (n+1) \qquad P \Downarrow k}{\mathsf{pcond}\ M\ N\ P \Downarrow k} \qquad \frac{M \Downarrow 0 \qquad N \Downarrow V}{\mathsf{cond}\ M\ N\ P \Downarrow V}$$

$$\frac{N \Downarrow k \qquad P \Downarrow k}{\mathsf{pcond}\ M\ N\ P \Downarrow k} \qquad \frac{M \Downarrow (n+1) \qquad P \Downarrow V}{\mathsf{cond}\ M\ N\ P \Downarrow V}$$

Figure 1: Structured rules for applying constants and reducing conditionals.

Call-by-name	$\dfrac{M \Downarrow_n \lambda x.M' \qquad M'[x := N] \Downarrow_n V}{(M\ N) \Downarrow_n V} \qquad \dfrac{M \Downarrow_n Y \qquad (N\ (Y\ N)) \Downarrow_n V}{(M\ N) \Downarrow_n V}$
Lazy	$\dfrac{M \Downarrow_l \lambda x.M' \qquad M'[x := N] \Downarrow_l V}{(M\ N) \Downarrow_l V} \qquad \dfrac{M \Downarrow_l Y \qquad (N\ (Y\ N)) \Downarrow_l V}{(M\ N) \Downarrow_l V}$ $\dfrac{M \Downarrow_l \mathsf{conv}^{\sigma,\tau} \qquad N \Downarrow_l V}{(M\ N) \Downarrow_l \lambda x^{\tau}.x}$
Call-by-value	$\dfrac{M \Downarrow_v \lambda x.M' \qquad N \Downarrow_v V \qquad M'[x := V] \Downarrow_v V'}{(M\ N) \Downarrow_v V'} \qquad \dfrac{M \Downarrow_v Y \qquad N \Downarrow_v V}{(M\ N) \Downarrow_v \lambda x.V\ (Y\ V)\ x}$

Figure 2: Additional rules for each of the three interpreters.

Basic Translation	$\begin{aligned} \overline{x^{\sigma}} &= (\delta^{\sigma}\ x^{\sigma}) \\ \overline{k} &= k \\ \overline{\mathsf{succ}} &= \mathsf{succ} \\ \overline{\mathsf{pred}} &= \mathsf{pred} \\ \overline{Y} &= \lambda z^{\sigma\to\sigma}.\mathsf{conv}\ z\ (Y\ (\lambda x^{\sigma}.\lambda y^{\sigma_1}.(\delta^{\sigma\to\sigma}\ z)\ x\ y)), \quad \sigma = \sigma_1 \to \sigma_2 \\ \overline{\lambda x^{\sigma}.M} &= \lambda x^{\sigma}.\mathsf{conv}\ x\ \overline{M} \\ \overline{(M\ N)} &= (\overline{M}\ \overline{N}) \\ \overline{\mathsf{cond}\ M\ N\ P} &= \mathsf{cond}\ \overline{M}\ \overline{N}\ \overline{P} \\ \overline{\mathsf{pcond}\ M\ N\ P} &= \mathsf{pcond}\ \overline{M}\ \overline{N}\ \overline{P} \end{aligned}$
Retractions	$\begin{aligned} \delta^{\iota} &= \lambda x^{\iota}.x \\ \delta^{\sigma\to\tau} &= \lambda x^{\sigma\to\tau}.\mathsf{conv}\ x\ (\lambda y^{\sigma}.\mathsf{conv}\ y\ (\delta^{\tau}\ (x\ (\delta^{\sigma}\ y)))) \end{aligned}$

Figure 3: The complete translation of call-by-value PCF to lazy PCF.

Technically, the functions δ^σ are **retractions**. The fact that δ^σ is a retraction, *i.e.*,

$$(\delta^\sigma\ (\delta^\sigma\ M)) \equiv_{lazy}^{termin} (\delta^\sigma\ M)$$

for any term M of type σ, follows by a simple induction on types. Intuitively, this fact states that making a strict function strict does not change the function.

With the addition of these retractions, the translation briefly outlined in the introduction becomes fully abstract:

Theorem 7 *The translation $M \mapsto \overline{M}$ from call-by-value PCF to lazy PCF is fully abstract. That is, $M \sqsubseteq_{val}^{termin} N \iff \overline{M} \sqsubseteq_{lazy}^{termin} \overline{N}$.*

Proof: (Sketch) We rely on denotational models for call-by-value PCF and lazy PCF. The two models, $\mathcal{C}_{val}$ and $\mathcal{C}_{lazy}$, are built using Scott domains of lifted, continuous functions. In the case of $\mathcal{C}_{val}$, only strict continuous functions are in the model. (Various equivalent definitions of these models may be found in [4, 5, 18, 22, 23].) Importantly, both models are fully abstract, *i.e.*,

$$\begin{array}{lll} M \sqsubseteq_{val}^{termin} N & \text{iff} & \mathcal{C}_{val}[\![M]\!] \sqsubseteq \mathcal{C}_{val}[\![N]\!] \\ M \sqsubseteq_{lazy}^{termin} N & \text{iff} & \mathcal{C}_{lazy}[\![M]\!] \sqsubseteq \mathcal{C}_{lazy}[\![N]\!]. \end{array}$$

The key idea is to show how to relate values in $\mathcal{C}_{val}$ to values in $\mathcal{C}_{lazy}$. We use logical relations to relate these two models. Let $\mathcal{C}_{val}^\sigma$ denote the values in $\mathcal{C}_{val}$ of type σ, and define $\mathcal{C}_{lazy}^\sigma$ similarly. Then define the relations $R^\sigma \subseteq \mathcal{C}_{val}^\sigma \times \mathcal{C}_{lazy}^\sigma$ by induction on types as follows:

1. $d\ R^\iota\ e$ iff $d = e$; and
2. $f\ R^{\sigma\to\tau}\ g$ iff $(f = \bot \iff g = \bot)$, and for any $d\ R^\sigma\ e$, $(f\ d)\ R^\tau\ (g\ e)$.

This definition should be compared to the definitions in [5, 15, 18]. Two crucial facts hold about this logical relation. First, each R^σ is surjective on the range of $\mathcal{C}_{lazy}[\![\delta^\sigma]\!]$. Second, the analog of the Fundamental Theorem of Logical Relations [25] holds: for any closed M, $\mathcal{C}_{val}[\![M]\!]\ R^\sigma\ \mathcal{C}_{lazy}[\![\overline{M}]\!]$.

The ($\Leftarrow$) direction of full abstraction follows somewhat routinely from the second fact above. For the converse, suppose $\overline{M} \not\sqsubseteq_{lazy}^{termin} \overline{N}$ for closed M and N. By the full abstraction theorem, $\mathcal{C}_{lazy}[\![\overline{M}]\!] \not\sqsubseteq \mathcal{C}_{lazy}[\![\overline{N}]\!]$. Thus, there are values d_i for which either

1. $(\mathcal{C}_{lazy}[\![\overline{M}]\!]\ d_1 \cdots d_n) \neq \bot$ & $(\mathcal{C}_{lazy}[\![\overline{N}]\!]\ d_1 \cdots d_n) = \bot$; or
2. $(\mathcal{C}_{lazy}[\![\overline{M}]\!]\ d_1 \cdots d_n) = k$, $(\mathcal{C}_{lazy}[\![\overline{N}]\!]\ d_1 \cdots d_n) = k'$, for k, k' numerals and $k \neq k'$.

Since all variables are translated to variables applied to the retractions, we may assume that the d_i are in the range of $\mathcal{C}_{lazy}[\![\delta^\sigma]\!]$. By the surjectivity of the logical relation, there are d_i' such that $d_i'\ R^{\sigma_i}\ d_i$. By the fact that $\mathcal{C}_{val}[\![M]\!]\ R^\sigma\ \mathcal{C}_{lazy}[\![\overline{M}]\!]$, these d_i' distinguish $\mathcal{C}_{val}[\![M]\!]$ and $\mathcal{C}_{val}[\![N]\!]$, so $\mathcal{C}_{val}[\![M]\!] \not\sqsubseteq \mathcal{C}_{val}[\![N]\!]$. By full abstraction of $\mathcal{C}_{val}$, $M \not\sqsubseteq_{val}^{termin} N$. ■

3.2 Call-by-name to Call-by-value

We need a different idea to translate call-by-name PCF to call-by-value PCF. We adopt the familiar idea of *thunks, viz.*, λ-abstractions that are constant in their first argument, to translate call-by-name PCF to call-by-value PCF. All call-by-name terms are mapped either to thunks or to terms that reduce to thunks. This guarantees that all terms—and hence all operands in applications—terminate, so that the call-by-value interpreter never gets stuck evaluating an operand.

Unlike the previous example, the translation changes the types of terms. For simplicity, we make every term take a dummy argument of type ι—although a dummy argument of any type would suffice. Terms of type σ are translated to terms of type σ', where

$$\begin{array}{rcl} \iota' & = & \iota \to \iota \\ (\sigma \to \tau)' & = & \iota \to \sigma' \to \tau' \end{array}$$

The full translation from call-by-name to call-by-value appears in Figure 4. Again, we need to apply retractions to the variables to make the translation fully abstract. In this case, the retractions γ^σ force terms to be constant functions in their first argument. The definition of the retractions appears in Figure 4, and, as before, proceeds by induction on types.

Like the translation from call-by-value to lazy, this translation is fully abstract:

Theorem 8 *The translation $M \mapsto \widehat{M}$ from call-by-name PCF to call-by-value PCF is fully abstract. That is, $M \sqsubseteq_{name}^{num} N \iff \widehat{M} \sqsubseteq_{val}^{termin} \widehat{N}$.*

The proof of the theorem follows the outline of the proof of Theorem 7, and involves constructing a logical relation from the standard Scott model $\mathcal{C}_{name}$ for call-by-name [14, 21] to the $\mathcal{C}_{val}$ model for call-by-value.

3.3 Lazy to Call-by-value

Figure 4 also gives a translation for lazy to call-by-value PCF similar to the translation for call-by-name to call-by-value. Here, most of the clauses for terms are *identical* to the call-by-name to call-by-value case. The only exceptions are the definition of the retractions χ^σ, the clauses involving these retractions, and the additional clause for translating conv.

Basic Translation	$\widehat{k} = \lambda z^{\iota}.k$ $\widehat{\mathsf{succ}} = \lambda z^{\iota}.\lambda x^{\iota\to\iota}.\lambda z^{\iota}.\mathsf{succ}\,(x\ 3)$ $\widehat{\mathsf{pred}} = \lambda z^{\iota}.\lambda x^{\iota\to\iota}.\lambda z^{\iota}.\mathsf{pred}\,(x\ 3)$ $\widehat{\lambda x^{\sigma}.M} = \lambda z^{\iota}.\lambda x^{\sigma'}.\widehat{M}, \quad z \notin FV(M)$ $\widehat{(M\ N)} = ((\widehat{M}\ 3)\ \widehat{N})$ $\widehat{\mathsf{cond}\,M\,N\,P} = \lambda z^{\iota}.\mathsf{cond}\,(\widehat{M}\ 3)\,(\widehat{N}\ 3)\,(\widehat{P}\ 3)$ $\widehat{\mathsf{pcond}\,M\,N\,P} = \lambda z^{\iota}.\mathsf{pcond}\,(\widehat{M}\ 3)\,(\widehat{N}\ 3)\,(\widehat{P}\ 3)$
Call-by-name to call-by-value	$\widehat{x^{\sigma}} = (\gamma^{\sigma}\ x^{\sigma'})$ $\widehat{Y} = \lambda z^{\iota}.\lambda x^{(\sigma\to\sigma)'}.Y((\gamma^{\sigma\to\sigma}\ x)\ 3)$ $\gamma^{\iota} = \lambda x^{\iota'}.\lambda z^{\iota}.x\ 3$ $\gamma^{\sigma\to\tau} = \lambda x^{(\sigma\to\tau)'}.\lambda z^{\iota}.\lambda y^{\sigma'}.(\gamma^{\tau}\ (\lambda z^{\iota}.x\ 3\ (\gamma^{\sigma}\ y)\ z))$
Lazy to call-by-value	$\widehat{x^{\sigma}} = (\chi^{\sigma}\ x^{\sigma'})$ $\widehat{Y} = \lambda z^{\iota}.\lambda x^{(\sigma\to\sigma)'}.Y((\chi^{\sigma\to\sigma}\ x)\ 3)$ $\widehat{\mathsf{conv}^{\sigma,\tau}} = \lambda z^{\iota}.\lambda x^{\sigma'}.\lambda z^{\iota}.(\lambda w.\lambda u^{\tau'}.(\delta^{\tau}\ u))\ (x\ 3)$ $\chi^{\iota} = \lambda x^{\iota'}.\lambda z^{\iota}.x\ 3$ $\chi^{\sigma\to\tau} = \lambda x^{(\sigma\to\tau)'}.\lambda z^{\iota}.(\lambda w.\lambda y^{\sigma'}.(\chi^{\tau}\ (\lambda z^{\iota}.x\ 3\ (\chi^{\sigma}\ y)\ z)))\ (x\ 3)$

Figure 4: Translations of call-by-name and lazy PCF to call-by-value PCF.

This translation turns out to be fully abstract as well:

Theorem 9 *The translation $M \mapsto \widehat{M}$ from lazy PCF to call-by-value PCF is fully abstract. That is, $M \sqsubseteq^{termin}_{lazy} N \iff \widehat{M} \sqsubseteq^{termin}_{val} \widehat{N}$.*

Again, the proof of full abstraction follows along the same lines, by constructing a logical relation from lazy model $\mathcal{C}_{lazy}$ to the call-by-value model $\mathcal{C}_{val}$.

3.4 Bounds on Decision Procedures

Proving $M \sqsubseteq^{num}_{name} N$ for **pure** terms—*i.e.*, constant and conditional-free terms—coincides with $\beta\eta$-equality for pure terms [15]. Thus, since $\beta\eta$-equality of pure terms is *not* elementary recursive [24], testing to see whether $M \sqsubseteq^{num}_{name} N$ for pure M and N is not elementary recursive. From the fact that the translation from call-by-name to call-by-value PCF works in linear time, we can obtain a lower bound on the decision procedure for call-by-value PCF.

Corollary 10 *The following question cannot be decided in elementary recursive time: given two pure PCF terms M and N, is it the case that $M \sqsubseteq^{termin}_{val} N$?*

Proof: Suppose the question *can* be decided in elementary recursive time. Then one may decide whether $M \sqsubseteq^{num}_{name} N$ for pure terms in elementary recursive time: first translate and check whether $\overline{M} \sqsubseteq^{termin}_{val} \overline{N}$. This is a contradiction, so $M \sqsubseteq^{termin}_{val} N$ cannot be decided in elementary recursive time. ■

This corollary implies that deciding $M \sqsubseteq^{termin}_{val} N$ requires at least iterated exponential time.

Along similar lines, one can show that the problem of deciding $M \sqsubseteq^{termin}_{lazy} N$ for pure conv-terms (those containing only the constant conv) is not elementary recursive. In fact, the decision problems $M \sqsubseteq^{termin}_{lazy} N$ for pure conv-terms, and $M \sqsubseteq^{termin}_{val} N$ for pure terms, are equivalent under polynomial-time reducibility: this follows immediately from the fact that there are linear time reductions—via the translations—between these two problems.

We conjecture the following upper bound:

Conjecture 11 *The decision problem $M \sqsubseteq^{termin}_{val} N$ for pure M and N can be solved in iterated exponential time. Thus, the problem of deciding $M \sqsubseteq^{termin}_{lazy} N$ for M and N pure* conv*-terms can also be solved in iterated exponential time.*

It is already known that the problem of $M \sqsubseteq^{num}_{name} N$ for pure M and N can be decided in iterated exponential time [20, 24].

4 Functional Translations

4.1 Gödel numbering Translations

Not all translations between programming languages share the structure of the two translations above. It is reasonably easy, for instance, to design fully abstract translations that are not effective. But even the condition of effectiveness is not sufficiently strong to rule out "unreasonable" translations based on Gödel numbers. Consider the case of translating lazy into call-by-name.

Theorem 12 *There exists an effective translation of lazy PCF to call-by-name PCF preserving observational congruences. That is, there exists* $M \mapsto \widetilde{M}$ *such that*

$$M \equiv_{lazy}^{termin} N \iff \widetilde{M} \equiv_{name}^{num} \widetilde{N}.$$

Proof: (Sketch) We translate an LPCF term M to $(I\ \#M)$, for some Gödel numbering $\#$ of LPCF terms. Here, the closed term $I : \iota \to \iota \to \iota$ represents a "two-argument interpreter" for lazy PCF written in call-by-name PCF, where the first argument is the term to interpret and the second argument is a Gödel-numbered tuple of arguments to M (possibly an empty tuple). It is not hard to design such an interpreter meeting the following requirements:

1. $(I\ \#M\ \langle n_1, \ldots, n_m\rangle) \Uparrow_n$ if any of $n_1, \ldots, n_m$ is not the Gödel number of a closed term;
2. $(I\ \#M\ \langle \#N_1, \ldots, \#N_m\rangle) \Uparrow_n$ if the lazy term $(M\ N_1 \ldots N_m)$ is not well-typed;
3. $(I\ \#M\ \langle \#N_1, \ldots, \#N_m\rangle) \Downarrow_n$ iff it is the case that $(M\ N_1 \ldots N_m) \Downarrow_l$; and
4. $(I\ \#M\ \langle \#N_1, \ldots, \#N_m\rangle) \Downarrow_n k$ iff it is the case that $(M\ N_1 \ldots N_m) \Downarrow_l k$.

To verify that the translation preserves observational congruences, suppose $M \not\equiv_{lazy}^{termin} N$ with M and N having type $\sigma_1 \to \ldots \to \sigma_n \to \iota$. By the Context Lemma for Lazy PCF (*cf.* [8, 14, 17]), there are terms $P_1, \ldots, P_m$ such that either

1. $(M\ P_1 \ldots P_m) \Downarrow_l$ but $(N\ P_1 \ldots P_m) \Uparrow_l$; or
2. $(M\ P_1 \ldots P_m) \Downarrow_l k$ but $(N\ P_1 \ldots P_m) \Downarrow_l k'$, where $k \neq k'$ and $m = n$.

By the properties of I, $(\widetilde{M}\ \langle \#P_1, \ldots, \#P_m\rangle)$ yields a different observation than $(\widetilde{N}\ \langle \#P_1, \ldots, \#P_m\rangle)$. Thus, $\widetilde{M} \not\equiv_{name}^{num} \widetilde{N}$. The converse follows similarly and is omitted. ■

Similar translations based on Gödel numbers can be found between almost all universal programming languages.

4.2 Definition of Functional Translations

The existence of fully abstract translations becomes a trivial problem unless we limit translations in some way. Mitchell, for example, considers only *compositional* translations [9]. Nevertheless, compositionality alone is too weak to rule out translations based on Gödel numbers. The above translation from lazy to call-by-name can, for instance, be made compositional by explicitly computing the Gödel number and applying the resultant term to the interpreter function I.

The translations of Section 3 enjoy a few other properties. It is instructive to first consider the translation from call-by-value PCF to lazy PCF. This translation not only preserves observational congruences, but also translates call-by-value functions into strict lazy functions. The lazy versions of call-by-value terms are in some sense "functionally equivalent" to their original versions: the translates of terms of type ι produce the same values as the original terms, and the translations of functionally-typed terms, when provided with strict functions, return strict results.

The connection between call-by-name terms and their call-by-value translates is not so clear: translated terms take dummy arguments and hence represent different functions. Nevertheless, some of the behavior of a translated term may be recovered through a call-by-value definable function *proj*. At ground type, $proj^{\iota} : \iota' \to \iota$ is the function that applies a (call-by-value) term of type ι' to a dummy argument (say 3) to obtain a numeric result. An inverse to $proj^{\iota}$, called $inj^{\iota} : \iota \to \iota'$, is also definable in call-by-value PCF; this function maps a number n to the constant function $\lambda z^{\iota}.n$.

Similarly, one can define call-by-value functions

$$proj^{\sigma\to\tau} : (\sigma \to \tau)' \to \sigma' \to \tau'$$
$$inj^{\sigma\to\tau} : (\sigma' \to \tau') \to (\sigma \to \tau)'$$

at higher type. Informally, $(proj^{\sigma\to\tau}\ \widehat{M})$ applies $\widehat{M}$ to 3 to obtain a function from representations of type σ' to results of type τ'. The term $inj^{\sigma\to\tau}$ is the inverse of $proj^{\sigma\to\tau}$: $(proj\ (inj\ N) \equiv_{val}^{termin} N$. In fact, one can define *inj* and *proj* with $(inj\ (proj\ N)) \equiv_{val}^{termin} (\gamma\ N)$.

Finally, the translations each use a uniform representation of source language code. We use a general form of logical relations to capture the idea of a "representation." Let $\mathcal{L}^{\sigma}$ be the set of closed $\mathcal{L}$-terms of type σ. If $M \mapsto \widetilde{M}$ takes $\mathcal{L}_1^{\sigma}$ terms to $\mathcal{L}_2^{\sigma'}$ terms, the relations R^{σ} will relate $\mathcal{L}_1^{\sigma}$ terms to $\mathcal{L}_2^{\sigma'}$ terms.

To simplify the definition, we allow only those languages in which applicative contexts are sufficient to distinguish non-observationally congruent terms. Formally, a simply-typed language $\mathcal{L}_1$ is *operationally ex-*

tensional if $M \not\equiv^{\mathcal{O}_1}_{\mathcal{L}_1} N$ implies that for some terms P_i, $(M\ P_1 \ldots\ P_k)$ and $(N\ P_1 \ldots\ P_k)$ produce different observables [2, 3]. Also, suppose $\mathcal{L}_1$ and $\mathcal{L}_2$ be simply-typed languages with observables $\mathcal{O}_1$ and $\mathcal{O}_2$, where $\mathcal{O}_1 \subseteq \mathcal{O}_2$; then we write $M \rightleftharpoons N$, for M a closed $\mathcal{L}_1$-term and N a closed $\mathcal{L}_2$ term, if M produces an observable in $\mathcal{O}_1$ under $\mathcal{L}_1$ iff N produces the same observable under $\mathcal{L}_2$.

Definition 13 *Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be simply-typed languages with observables $\mathcal{O}_1$ and $\mathcal{O}_2$, where $\mathcal{O}_1 \subseteq \mathcal{O}_2$. A translation $M \mapsto \widetilde{M}$ of $\mathcal{L}_1^{\sigma}$ to $\mathcal{L}_2^{\sigma'}$ is a* **functional translation** *if there exist $\mathcal{L}_2$-definable maps*

$$proj^{\iota} : \iota' \to \iota$$
$$inj^{\iota} : \iota \to \iota'$$

$$proj^{\sigma\to\tau} : (\sigma \to \tau)' \to \sigma' \to \tau'$$
$$inj^{\sigma\to\tau} : (\sigma' \to \tau') \to (\sigma \to \tau)'$$

such that

1. *Logical relation: $M\ R^{\sigma}\ \widetilde{M}$, where*
   (a) *$M\ R^{\iota}\ N$ iff $M \rightleftharpoons (proj^{\iota}\ N)$; and*
   (b) *$M\ R^{\sigma\to\tau}\ N$ iff $M \rightleftharpoons (proj^{\sigma\to\tau}\ N)$, and $P\ R^{\sigma}\ Q$ implies that $(M\ P)\ R^{\tau}\ (N \bullet Q)$, where $N \bullet Q = ((proj^{\sigma\to\tau}\ N)\ Q)$.*
2. *Non-congruent representations are distinguishable by "applicative" contexts: Suppose M and N are in the range of R^{σ} and $M \not\equiv^{\mathcal{O}_2}_{\mathcal{L}_2} N$. Then there are P_i where $(proj\ (M \bullet P_1 \bullet \ldots P_k))$ produces a different $\mathcal{O}_1$-observable than $(proj\ (N \bullet P_1 \bullet \ldots P_k))$.*
3. *Injections are the inverses of projections: For any closed $\mathcal{L}_2$ term N, $(proj\ (inj\ N)) \equiv^{\mathcal{O}_2}_{\mathcal{L}_2} N$.*
4. *Representations can be built by injections: For any closed $\mathcal{L}_2$-term N, $(inj\ (proj\ N))$ is in the range of R^{σ}. Also, if N is in the range of R^{σ}, $(inj\ (proj\ N)) \equiv^{\mathcal{O}_2}_{\mathcal{L}_2} N$.*
5. *Projections force arguments to be in the range of R: For any terms M and N,*

$$M \bullet N \equiv^{\mathcal{O}_2}_{\mathcal{L}_2} inj\ (proj\ (M \bullet (inj\ (proj\ N))))$$

This definition should be compared to the definition of the relations given in the proof of Theorem 7.

Every functional translation is fully abstract:

Theorem 14 *Let $\mathcal{L}_1$ be any simply-typed, operationally extensional language, and suppose $M \mapsto \widetilde{M}$ is a functional translation from $\mathcal{L}_1$ to $\mathcal{L}_2$ with projections $proj^{\sigma}$ and injections inj^{σ}. Then $M \mapsto \widetilde{M}$ is fully abstract.*

Proof: Suppose first that $M \not\equiv^{\mathcal{O}_1}_{\mathcal{L}_1} N$. Then by operational extensionality, there exist terms P_i and $k \geq 0$ with $(M\ P_1 \ldots P_k)$ producing a different observable than $(N\ P_1 \ldots P_k)$. By the definition of logical translation, $(proj\ (\widetilde{M} \bullet \widetilde{P_1} \bullet \ldots \bullet \widetilde{P_k}))$ produces a different observable than $(proj\ (\widetilde{N} \bullet \widetilde{P_1} \bullet \ldots \bullet \widetilde{P_k}))$. Thus, $\widetilde{M} \not\equiv^{\mathcal{O}_2}_{\mathcal{L}_2} \widetilde{N}$.

Conversely, suppose $\widetilde{M} \not\equiv^{\mathcal{O}_2}_{\mathcal{L}_2} \widetilde{N}$. As $\widetilde{M}$ and $\widetilde{N}$ are in the range of R^{σ}, there exist closed $\mathcal{L}_2$-terms P_i with $(proj\ (\widetilde{M} \bullet P_1 \bullet \ldots \bullet P_k))$ producing a different $\mathcal{O}_1$-observable than $(proj\ (\widetilde{N} \bullet P_1 \bullet \ldots \bullet P_k))$. Without loss of generality, the definition implies that we may choose P_i in the range of R^{σ_i}. Thus, let $(P_i'\ R\ P_i)$; then by the definition of the relations R, $(M\ P_1' \ldots P_k')$ produces a different observable than $(N\ P_1' \ldots P_k')$. It follows that $M \not\equiv^{\mathcal{O}_1}_{\mathcal{L}_1} N$. ■

4.3 Distinctions Made by Functional Translations

The translations of Section 3 demonstrate that, in some sense, call-by-value PCF and lazy PCF are "equivalent" under the notion of functional translation: each can be translated into the other. Call-by-name PCF can be translated into call-by-value as well, but call-by-name PCF is strictly less expressive than call-by-value PCF under the notion of functional translation. To keep the set of observations uniform between the two languages, we say that a closed call-by-name term M of type $\sigma \to \tau$ *converges* iff $\Omega^{\sigma\to\tau} \not\equiv^{num}_{name} M$. (Note that this is an r.e. property of terms, and adding this observation does not refine the relation $\equiv^{num}_{name}$.) Then

Theorem 15 *There is no functional translation from call-by-value PCF to call-by-name PCF.*

Proof: Suppose $M \mapsto \widetilde{M}$ is a functional translation with projections $proj^{\sigma}$ and injections inj^{σ}. Let Ω_1 and Ω_2 be divergent lazy PCF terms of types $\iota \to \iota$ and ι respectively. By the definition of functional translation, $\Omega_1\ R^{\iota\to\iota}\ \widetilde{\Omega_1}$ and $\lambda x.\Omega_2\ R^{\iota\to\iota}\ \widetilde{\lambda x.\Omega_2}$. Thus, $\Omega_1 \rightleftharpoons (proj^{\iota\to\iota}\ \widetilde{\Omega_1})$ and $\lambda x.\Omega_2 \rightleftharpoons (proj^{\iota\to\iota}\ \widetilde{\lambda x.\Omega_2})$. As Ω_1 and $\lambda x.\Omega_2$ produce different observable outcomes under the call-by-value PCF interpreter,

$$(proj^{\iota\to\iota}\ \widetilde{\Omega_1}) \not\equiv^{num}_{name} (proj^{\iota\to\iota}\ \widetilde{\lambda x.\Omega_2})$$

since they must produce different observables under the call-by-name PCF interpreter.

However, by the definition of functional translation, $((proj^{\iota\to\iota}\ \widetilde{\lambda x.\Omega_2})\ N)$ for any closed N diverges. Similarly, $((proj^{\iota\to\iota}\ \widetilde{\Omega_1})\ N)$ diverges. As we are dealing with the call-by-name theory,

$$(proj^{\iota\to\iota}\ \widetilde{\Omega_1}) \equiv^{num}_{name} (proj^{\iota\to\iota}\ \widetilde{\lambda x.\Omega_2})$$

This is a contradiction, so there can be no functional translation from call-by-value PCF to call-by-name PCF. ■

A similar proof shows that there is no functional translation from lazy PCF to call-by-name PCF.

The notion of a functional translation can also be used to compare languages with more features to those with fewer features. For example, let "sequential" call-by-name PCF be call-by-name PCF without parallel conditional. Then

Theorem 16 *There is no functional translation from call-by-name PCF to sequential call-by-name PCF.*

Proof: Suppose, by way of contradiction, $M \mapsto \widetilde{M}$ is a functional translation with projections $proj^\sigma$ and injections inj^σ. Let $\mathsf{por} : \iota \to \iota \to \iota$ be a term of call-by-name PCF with the following behavior:

$$\begin{array}{lll} (\mathsf{por}\ 0\ N) & \equiv^{num}_{name} & 0 \\ (\mathsf{por}\ N\ 0) & \equiv^{num}_{name} & 0 \\ (\mathsf{por}\ m\ n) & \equiv^{num}_{name} & 1 \text{ if } m, n \text{ are nonzero numerals} \end{array}$$

where N is any closed term of type ι. Note that por is definable in call-by-name PCF but *not* in sequential call-by-name PCF [14, 19]. However, the term

$$P = \lambda x^\iota.\lambda y^\iota.proj^\iota\ (\widetilde{\mathsf{por}} \bullet (inj^\iota\ x) \bullet (inj^\iota\ y))$$

defines por in sequential call-by-name PCF. This is a contradiction, so there can be no functional translation from (parallel) call-by-name PCF to sequential call-by-name PCF. ■

5 Open Problems

We have explored situations in which well-structured, fully abstract translations exist. The three fully abstract translations of Section 3 differ in one respect from the well-known translations: retractions are used on variables to force target language terms to be legal representations. We have also given a definition of "good" translations that rules out Gödel numbering translations, and shown that certain languages are strictly more powerful than other programming languages.

It seems quite likely that other structured, fully abstract translations exist between other functional languages. For instance, the examples above deal only with simply-typed languages. Using a fairly natural modification of the retractions in the call-by-value to lazy case, there is a well-structured translation from the untyped call-by-value λ-calculus to the untyped lazy λ-calculus. The proof relies on two models: the fully abstract model for the lazy λ-calculus [1, 11, 12], and the fully abstract model for the call-by-value λ-calculus composed of lifted, strict continuous functions (Felleisen and Sitaram, personal communication). Instead of logical relations, we use inclusive predicates [16]. This example should provide clues for adding general recursive types, as well as sums and products.

All three of the languages we have explored incorporate parallel conditional. Of course, we would like sequential fully abstract translations as well, *e.g.*, from sequential call-by-value PCF to sequential lazy PCF. We believe our methods will carry over to this problem with little modification.

Extending the languages with other features, such as those captured by monads [10] (*e.g.*, exceptions), also would be useful. We expect that the definition of logical relations to be different in these cases, and hence an expanded definition of "functional translation" may be necessary.

We have also only briefly discussed how the notion of "functional translations" leads to a definition of expressiveness. Our translations show that, in some sense, call-by-value and lazy PCF are equally expressive whereas call-by-name PCF is strictly *less* expressive than either of these languages. But there is much more work to be done in exploring this definition of expressiveness. Examining the algebraic properties of functional translations would be a good start; for example, functional translations should be closed under composition. We leave this investigation open as well.

Acknowledgments

I especially thank Albert Meyer for the suggestion of this problem and many productive conversations. I also thank Samson Abramsky, Val Breazu-Tannen, Stavros Cosmadakis, Matthias Felleisen, Carl Gunter, Eugenio Moggi, and Gordon Plotkin for helpful discussions, and Michael Ernst, Lalita Jategaonkar, Trevor Jim, Arthur Lent, and David Wald for comments on drafts of this paper.

References

[1] Samson Abramsky. The lazy lambda calculus. In David A. Turner, editor, *Research Topics in Functional Programming*, pages 65–117. Addison-Wesley, 1990.

[2] Bard Bloom. Can LCF be topped? In 3rd *Symp. Logic in Computer Science*, pages 282–295. IEEE, 1988.

[3] Bard Bloom. Can LCF be topped? flat lattice models of typed λ-calculus. *Information and Computation*, 87:264–301, 1990.

[4] Bard Bloom and Jon G. Riecke. LCF should be lifted. In Teodor Rus, editor, *Proc. Conf. Algebraic Methodology and Software Technology*, pages 133–136. Department of Computer Science, University of Iowa, 1989.

[5] Stavros S. Cosmadakis, Albert R. Meyer, and Jon G. Riecke. Completeness for typed lazy inequalities (preliminary report). In 5^{th} *Symp. Logic in Computer Science*, pages 312–320. IEEE, 1990.

[6] Matthias Felleisen. On the expressive power of programming languages. In *Proc. of European Symp. on Programming*, Lect. Notes in Computer Sci., pages 134–151. Springer-Verlag, 1990.

[7] Albert R. Meyer and Jon G. Riecke. Continuations may be unreasonable. In *Proc. Conf. LISP and Functional Programming*, pages 63–71. ACM, 1988.

[8] Robin Milner. Fully abstract models of the typed lambda calculus. *Theoretical Computer Sci.*, 4:1–22, 1977.

[9] John C. Mitchell. Lisp is not universal (summary). Unpublished manuscript, August 1986.

[10] Eugenio Moggi. Computational lambda-calculus and monads. In 4^{th} *Symp. Logic in Computer Science*, pages 14–23. IEEE, 1989.

[11] Chih-Hao Luke Ong. Fully abstract models of the lazy lambda calculus. In 29^{th} *Symp. Foundations of Computer Science*, pages 368–376. IEEE, 1988.

[12] Chih-Hao Luke Ong. *The Lazy Lambda Calculus: An Investigation into the Foundations of Functional Programming.* PhD thesis, Imperial College, University of London, 1988.

[13] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Sci.*, 1:125–159, 1975.

[14] Gordon D. Plotkin. LCF considered as a programming language. *Theoretical Computer Sci.*, 5:223–257, 1977.

[15] Gordon D. Plotkin. Notes on completeness of the full continuous type hierarchy. Unpublished manuscript, MIT, November 1982.

[16] John C. Reynolds. On the relation between direct and continuation semantics. In *Proceedings of the Second Colloquium on Automata, Languages, and Programming, Lecture Notes in Computer Science 14*, pages 141–156. Springer-Verlag, 1974.

[17] Jon G. Riecke. Should a function continue? Master's thesis, Dept. Electrical Engineering & Computer Sci., Massachusetts Institute of Technology, January 1989. Supervised by A.R. Meyer.

[18] Jon G. Riecke. A complete and decidable proof system for call-by-value equalities (preliminary report). In 17^{th} *ICALP*, volume 443 of *Lect. Notes in Computer Sci.*, pages 20–31. Springer-Verlag, 1990.

[19] V.Yu. Sazonov. Expressibility of functions in D. Scott's LCF language. *Algebra i Logika*, 15:308–330, 1976. (Russian).

[20] Helmut Schwichtenberg. Complexity of normalization in the pure typed lambda-calculus. In A.S. Troelstra and D. van Dalen, editors, *The L.E.J. Brouwer Centenary Symposium*, pages 453–457. North Holland, 1982.

[21] Dana Scott. A type theoretical alternative to CUCH, ISWIM, OWHY. Oxford University, unpublished manuscript, 1969.

[22] Kurt Sieber. Relating full abstraction results for different programming languages. In *Foundations of Software Technology and Theoretical Computer Science, Bangalore, India*, December 1990. To appear.

[23] Dorai Sitaram and Matthias Felleisen. Reasoning with continuations II: Full abstraction for models of control. In *Proc. Conf. LISP and Functional Programming*, pages 161–175. ACM, 1990.

[24] Richard Statman. The typed λ-calculus is not elementary recursive. *Theoretical Computer Sci.*, 9:73–81, 1979.

[25] Richard Statman. Logical relations in the typed λ-calculus. *Information and Control*, 65:86–97, 1985.

–Cambridge, MA
October 22, 1990