Fully Abstract Translations and Parametric Polymorphism

Peter W. O'Hearn*[1] and Jon G. Riecke[2]

[1] School of Computer & Information Science, Syracuse University
Syracuse NY 13244-4100
ohearn@top.cis.syr.edu
[2] AT&T Bell Laboratories
600 Mountain Avenue, Murray Hill, NJ 07974
riecke@research.att.com

Abstract. We examine three languages: call-by-name PCF; an idealized version of Algol called IA; and a call-by-name version of the functional core of ML with a parallel conditional, called PPCF+XML. Syntactic translations from PCF and IA into PPCF+XML are given and shown to be fully abstract, in the sense that they preserve and reflect observational equivalence. We believe that these results suggest the potential unifying force of Strachey's concept of parametric polymorphism.

1 Introduction

When Strachey first identified the notion of polymorphism, he immediately distinguished between two main species of polymorphic function [28]. In one form, called *ad hoc* polymorphism, a function may be applied to arguments of different types, but the algorithm may differ depending on the type. In the other form, called *parametric* polymorphism (the kind of polymorphism supported by the Girard-Reynolds polymorphic λ-calculus and the programming language Standard ML), the behaviour of a polymorphic function is determined uniformly for each instantiation of type variables. For instance, the *map* function, whose type can be written as $\forall\alpha.\ \forall\beta.\ (\alpha \rightarrow \beta) \rightarrow \mathrm{list}[\alpha] \rightarrow \mathrm{list}[\beta]$, works the same way across different types. Parametric polymorphism captures a form of abstraction (cf. [22, 24]): intuitively, a parametric function works in a way that does not presume knowledge of specific details of types to which it is instantiated.

Here we illustrate another connection between abstraction and parametric polymorphism: that parametric polymorphism can be used to represent in a very precise manner certain programming language features. A simple example crystallizes the general point. Consider a functional language with parametric polymorphism (e.g., the Girard-Reynolds calculus with recursion and basic arithmetic), and the polymorphic function

$$\begin{aligned} Q_P \;=\; & \Lambda \mathit{Counter}.\ \lambda \mathit{new} : (\mathit{Counter} \rightarrow \mathbf{nat}) \rightarrow \mathbf{nat}. \\ & \qquad \lambda \mathit{inc} : \mathit{Counter} \rightarrow \mathit{Counter}.\ \lambda \mathit{val} : \mathit{Counter} \rightarrow \mathbf{nat}.\ P. \end{aligned}$$

* Supported by NSF grant CCR-92110829.

We can apply Q_P to arguments that form the representation of an abstract "counter type," e.g., (Q_P[**nat**] **new inc val**) where

$$\begin{aligned}\mathbf{new} &= \lambda p : \mathbf{nat} \to \mathbf{nat}\,.\,p(0)\\ \mathbf{inc} &= \lambda n : \mathbf{nat}\,.\,n+1\\ \mathbf{val} &= \lambda n : \mathbf{nat}.n.\end{aligned}$$

Here we are utilizing the connection between abstract types and parametric polymorphism proposed by Reynolds [22]. The application binds the type variable *Counter* to the type **nat** of natural numbers, and the formal parameters to representations of the corresponding operations of the counter type; for instance, **new** declares a new counter, initialized to 0, for use inside its argument p. Intuitively, the parametricity of $(\Lambda \mathit{Counter}.\dots P)$ guarantees that the representation of the counter type can only be accessed through the given operations: one cannot, for example, apply the *inc* formal parameter to a number inside the definition of P, since the type of *inc* expects an argument of type *Counter*—a type that could later be bound to a functional type. But even more complex properties of the term P hold. For instance, a subterm of the form $(\mathit{new}\,(\lambda c : \mathit{Counter}.\ C))$ binds a new counter to c for use within C. Furthermore, only *new* can create a new counter, and the counter disappears when the execution of C terminates, because the type of the subterm is **nat**. Notice that this is reminiscent of the stack discipline for local variables, and that it arises *in a purely functional language!* A simple program equivalence makes the connection to local variables even more explicit. Suppose the type of Q_P is

$$\begin{aligned}\forall \mathit{Counter}.\ ((\mathit{Counter} \to \mathbf{nat}) \to \mathbf{nat}) &\to (\mathit{Counter} \to \mathit{Counter})\\ &\to (\mathit{Counter} \to \mathbf{nat}) \to \mathit{Counter}\end{aligned}$$

Then (Q_P[**nat**] **new inc val**) $\equiv \Omega$ where Ω is a divergent term of type **nat**: even though a new non-divergent counter might be used in P (in the context of a *new* declaration), such a counter can never be returned outside the declaration, so the only counter that the application can return is Ω.

This example has to do with the "abstraction barrier" between an abstract description of a programming language and a more detailed implementation. Think of P as a program written in a functional language with two base types: one called **nat** with the usual operations, and one called *Counter* whose only operations are *new*, *inc*, and *val*. Two pieces of code P_1 and P_2 written in this language may only be distinguishable using the arithmetic operations and *new*, *inc*, *val*. A compiler or interpreter hides the implementation of these operations from the programmer. For instance, an interpreter could implement "deallocation" of counters by decrementing a stack pointer: even though the "old" counter may still be held in memory, a program may not access it. This is precisely what parametricity provides: a way to specify that only the operations of the base type—and no other operations derivable from the details of the implementation—may be used to operate on *Counter*.

The example illustrates an important point: reasoning about polymorphic functions in a pure functional language can provide a basis for reasoning about certain features of local state. In this paper we develop this idea and show how to define a translation from an Algol-like language into a purely functional language with ML-style polymorphism. The translation is defined by analogy with the treatment of the "counter" example. Instead of abstracting on a single type variable *Counter*, the translation abstracts on a type variable for *each* primitive type in an Algol-like language, and passes representations of Algol base types and relevant operations (such as assignment) as arguments to a polymorphic function. In essence, the translation treats the base types and constructs of Algol as forming "higher-order" abstract data types. Our main theorem shows that that the translation is **fully abstract**, i.e., it preserves and reflects observational equivalence (cf. [25]).

The idea of using the "abstractness" of type variables in a polymorphic language to protect representations of types is applicable to other languages as well. For example, consider a polymorphic extension of parallel PCF, i.e., the typed λ-calculus with recursion, basic arithmetic operations, and a parallel conditional operation. Then for any term p of type

$$\forall\alpha\,.\,\alpha \to (\alpha \to \alpha) \to (\alpha \to \alpha \to \alpha) \to (\alpha \to \alpha \to \alpha \to \alpha) \to \alpha \to \alpha \to \alpha$$

one may obtain a function of type **bool** $\to$ **bool** $\to$ **bool** by instantiating α to the type **bool** of booleans and the first four arguments to the true boolean, negation, conjunction and the sequential conditional:

$$(p\;[\mathbf{bool}]\;\mathbf{true}\;\;\mathbf{not}\;\;\mathbf{and}\;\mathbf{if})\;:\;\mathbf{bool} \to \mathbf{bool} \to \mathbf{bool}.$$

For any function of type **bool** $\to$ **bool** $\to$ **bool** definable in sequential PCF there is a p such that this term denotes the same function. However, even though "parallel or" exists in the language, this term can never be (equivalent to) "parallel or". Intuitively, the parametricity of p means that (p [**bool**] **true not and if**) cannot use parallel facilities, because they are not definable from the given arguments. It is even possible to prove this using a model based on Reynolds's relational approach to parametricity [24].

Based on these ideas, we define a translation from sequential PCF into a polymorphic version of parallel PCF, and again prove that the translation is fully abstract. Roughly speaking, we again treat the type of natural numbers from sequential PCF as an abstract data type. While our main interest in this translation method concerns possible applications to understanding state and related features, the PCF translation illustrates the main ideas in a simpler context: parametric polymorphism is used, as with the Algol translation, to "protect" the sequential source language from the parallelism present in the target language.

2 Sequential PCF, Idealized Algol, and PPCF+XML

In this section we define the three languages considered in this paper. In defining the languages, we often share reduction and typing rules across languages, expecting that no confusion will arise.

2.1 Sequential PCF

Our version of PCF has one base type **nat** of natural numbers. The types are

$$t ::= \mathbf{nat} \,|\, t \rightarrow t.$$

We use s, t to range over types. A typing judgement is a formula of the form $\Gamma \vdash M : t$ where M is a term, t a type, and Γ is a **PCF type environment**, i.e., a finite function from variables to types. Standard rules for deriving typing judgements may be found in Table 1.

The operational semantics is given by a reduction relation $M \rightarrow N$ between terms in Table 2; this is the usual call-by-name, sequential strategy for PCF. In these rules, $M\{N/x\}$ denotes the result of substituting N for x in M with the necessary renaming of bound variables, and n denotes the n-fold application of **succ** to 0. The relation $\rightarrow^*$ denotes the reflexive, transitive closure of $\rightarrow$. We define observational equivalence so that a judgement of equivalence can only be made in the presence of a type assignment. This is reasonable in a language where variables do not come tagged with types.

Definition 1. 1. $C[\cdot]$ is a PCF **Γt-context** if $\vdash C[M] : \mathbf{nat}$ when $\Gamma \vdash M : t$.
2. Suppose $\Gamma \vdash M : t$ and $\Gamma \vdash N : t$. Then $\Gamma \vdash M \equiv N$ if for all PCF Γt-contexts $C[\cdot]$, $C[M] \rightarrow^* n \Leftrightarrow C[N] \rightarrow^* n$.

Table 1 PCF Typing Rules.

$\overline{\Gamma, x : t \vdash x : t}$	$\overline{\Gamma \vdash 0 : \mathbf{nat}}$	
$\dfrac{\Gamma, x : t \vdash M : s}{\Gamma \vdash (\lambda x : t.M) : t \rightarrow s}$	$\dfrac{\Gamma \vdash M : t \rightarrow s \quad \Gamma \vdash N : t}{\Gamma \vdash (M\ N) : s}$	$\dfrac{\Gamma \vdash M : t \rightarrow t}{\Gamma \vdash (\mathbf{Y}_t M) : t}$
$\dfrac{\Gamma \vdash M : \mathbf{nat}}{\Gamma \vdash (\mathbf{succ}\ M) : \mathbf{nat}}$	$\dfrac{\Gamma \vdash M : \mathbf{nat}}{\Gamma \vdash (\mathbf{pred}\ M) : \mathbf{nat}}$	$\dfrac{\Gamma \vdash M_i : \mathbf{nat}}{\Gamma \vdash (\mathbf{ifz}\ M_1\ M_2\ M_3) : \mathbf{nat}}$

2.2 Idealized Algol

The types of our second language, which we call IA (for Idealized Algol), are

$$\theta ::= \mathbf{exp} \mid \mathbf{loc} \mid \mathbf{comm} \mid \theta \rightarrow \theta$$

where **comm** is the type of commands, **exp** the type of expressions, and **loc** is the type of locations (or storage variables, or memory cells). For simplicity, we assume that the only storable values are natural numbers. Expressions are state-dependent values: they denote a value in a given state but do not change the state. Commands are state transformations.

Table 2 PCF Reduction Rules.

$$
\begin{array}{cc}
((\lambda x.\, M)\, N) \rightarrow M\{N/x\} & (\mathbf{ifz}\, 0\, M\, N) \rightarrow M \\
(\mathbf{Y}\, M) \rightarrow (M\, (\mathbf{Y}\, M)) & (\mathbf{ifz}\, (\mathbf{succ}\, n)\, M\, N) \rightarrow N \\
(\mathbf{pred}\, (\mathbf{succ}\, n)) \rightarrow n & \\
\dfrac{M \rightarrow M'}{(\mathbf{succ}\, M) \rightarrow (\mathbf{succ}\, M')} & \dfrac{M \rightarrow M'}{(M\, N) \rightarrow (M'\, N)} \\
\dfrac{M \rightarrow M'}{(\mathbf{pred}\, M) \rightarrow (\mathbf{pred}\, M')} & \dfrac{M \rightarrow M'}{(\mathbf{ifz}\, M\, N\, P) \rightarrow (\mathbf{ifz}\, M'\, N\, P)}
\end{array}
$$

The typing rules for IA appear in Table 3; π is used to denote an **IA type environment**, i.e., a finite map from variables to IA types. IA has constructs for assignment ($V := E$), dereferencing ($\mathbf{contents}\, V$) and sequencing ($C_1; C_2$). Variable declarations are of the form ($\mathbf{new}\ x = E\ \mathbf{in}\ C$) where C is a command, E an expression, and x a variable of location type. Variable declarations are executed by allocating a fresh location and setting the contents to the value of E in the current state, executing C where x is bound to the new location, and deallocating the location upon termination. The "newness" of local variables is the crux of the full abstraction problem for block structure and has been studied extensively (cf. [9, 13, 14]).

The reductions for IA come in two groups (see Tables 2 and 4). Purely functional reductions, the reductions found in Table 2, do not involve state, while the non-functional reductions, the reductions found in Table 4, are between configurations $[M, s]$ for M a term and s a state, i.e., a finite function from variables to numerals. For example, state is not needed for β-reduction, and so this is given by a functional reduction $(\lambda x.\, P)Q \rightarrow P\{Q/x\}$. In contrast, an assignment changes the state, and so we have a reduction between configurations $[x := n, s] \rightarrow [\mathbf{skip}, s[x \mapsto n]]$, where $s[x \mapsto n]$ is the state that modifies s by mapping x to n. An IA program is a closed term of type **comm**. We observe termination of programs, and write $C \Downarrow$ to mean that $[C, \epsilon] \rightarrow^* [\mathbf{skip}, \epsilon]$, where ϵ is the empty partial function.

Definition 2. 1. $C[\cdot]$ is an IA **$\pi\theta$-context** if $\vdash C[M] : \mathbf{comm}$ when $\pi \vdash M : \theta$.
2. Suppose that $\pi \vdash M : \theta$ and $\pi \vdash N : \theta$. Then $\pi \vdash M \equiv N$ if for all IA $\pi\theta$-contexts $C[\cdot]$, $C[M] \Downarrow \Leftrightarrow C[M] \Downarrow$.

2.3 PPCF+XML

The target language for our translations, called PPCF+XML, is PCF extended with a parallel conditional and with an explicitly-typed version of Milner's polymorphic let [11]. The type system is essentially the XML type system of Mitchell and Harper [12, 5].

Table 3 IA Typing Rules.

$$\frac{}{\pi, x : \theta \vdash x : \theta} \qquad \frac{}{\pi \vdash 0 : \mathbf{exp}} \qquad \frac{}{\pi \vdash \mathbf{skip} : \mathbf{comm}}$$

$$\frac{\pi, x : \theta \vdash M : \theta'}{\pi \vdash (\lambda x : \theta . M) : \theta \to \theta'} \qquad \frac{\pi \vdash M : \theta \to \theta' \quad \pi \vdash N : \theta}{\pi \vdash (M\ N) : \theta'} \qquad \frac{\pi \vdash M : \theta \to \theta}{\pi \vdash (\mathbf{Y}_\theta\, M) : \theta}$$

$$\frac{\pi \vdash M : \mathbf{exp}}{\pi \vdash (\mathbf{succ}\ M) : \mathbf{exp}} \qquad \frac{\pi \vdash M : \mathbf{exp}}{\pi \vdash (\mathbf{pred}\ M) : \mathbf{exp}}$$

$$\frac{\pi \vdash C_1 : \mathbf{comm} \quad \pi \vdash C_2 : \mathbf{comm}}{\pi \vdash C_1; C_2 : \mathbf{comm}} \qquad \frac{\pi \vdash M : \mathbf{loc} \quad \pi \vdash E : \mathbf{exp}}{\pi \vdash M := E : \mathbf{comm}}$$

$$\frac{\pi, z : \mathbf{loc} \vdash C : \mathbf{comm} \quad \pi \vdash E : \mathbf{exp}}{\pi \vdash \mathbf{new}\ z = E\ \mathbf{in}\ C : \mathbf{comm}} \qquad \frac{\pi \vdash M : \mathbf{loc}}{\pi \vdash \mathbf{contents}\, M : \mathbf{exp}}$$

$$\frac{\pi \vdash M : \mathbf{exp} \quad \pi \vdash N : B \quad \pi \vdash P : B}{\pi \vdash (\mathbf{ifz}_B\ M\ N\ P) : B}\ B \text{ a base type}$$

Table 4 Additional Reduction Rules for IA.

$$\frac{M \to N}{[M, s] \to [N, s]}$$

$$[(\mathbf{contents}\, x), s] \to [s(x), s] \qquad [(\mathbf{skip}; C), s] \to [C, s]$$

$$[(x := n), s] \to [\mathbf{skip}, s[x \mapsto n]] \qquad [(\mathbf{new}\, x = m\, \mathbf{in}\ \mathbf{skip}), s] \to [\mathbf{skip}, s]$$

$$\frac{[M, s] \to [M', s]}{[(\mathbf{contents}\ M), s] \to [(\mathbf{contents}\ M'), s]} \qquad \frac{[C_1, s] \to [C_1', s']}{[(C_1; C_2), s] \to [(C_1'; C_2), s']}$$

$$\frac{[E, s] \to [E', s]}{[(\mathbf{new}\, x = E\, \mathbf{in}\, C), s] \to [(\mathbf{new}\, x = E'\, \mathbf{in}\, C), s]} \qquad \frac{[M, s] \to [M', s]}{[(M := E), s] \to [(M' := E), s]}$$

$$\frac{[C, s[x \mapsto m]] \to [C', s'[x \mapsto n]]}{[(\mathbf{new}\, x = m\, \mathbf{in}\, C), s] \to [(\mathbf{new}\, x = n\, \mathbf{in}\, C'), s']} \qquad \frac{[E, s] \to [E', s]}{[(x := E), s] \to [(x := E'), s]}$$

PPCF+XML has two kinds of type, called *types* and *type schemes*:

$$\begin{array}{rll} t & ::= \alpha \mid \mathbf{nat} \mid t \to t & \text{(types)} \\ T & ::= t \mid \forall \alpha.\, T & \text{(type schemes)} \end{array}$$

We use s, t to range over types, S, T to range over type schemes, and α, β to range over type variables (which are distinct from ordinary program variables). Note that types are allowed to contain type variables, though no instances of $\forall$.

The grammar of terms is essentially that for PCF extended with four constructs: $(\Lambda\alpha.M)$ for abstraction on a type variable, $(M\,t)$ for application of a polymorphic term to a type (not a type scheme), an explicitly-typed version of Milner's **let** construct for binding variables of polymorphic type, and a parallel conditional of the form (**pifz** $M\,N\,P$) where N and P must be of type **nat**. A typing judgement in PPCF+XML is of the form $\Delta \vdash M : T$, where now Δ is a finite function from variables to type schemes, and T is a type scheme. The typing rules are given in Tables 1 and 5. (Notes: FTV(Δ) is the set of free type variables in type schemes assigned by Δ, and $M\{t/\alpha\}$ denotes the substitution of a type for a type variable in M.) The operational semantics (Tables 2 and 6) extends that of PCF with rules for **pifz**, β-reduction for types, and a β-reduction rule for reducing **let**'s.

To define equivalence in the polymorphic language we must keep track of free type variables in typing judgements. We write $X;\Delta \vdash M : T$ when $\Delta \vdash M : T$ is derivable and X is a set of type variables containing those free in T, M, and Δ. Thus, $\vdash M : T$ means that there are no type variables free in M or T.

Definition 3. 1. $C[\cdot]$ is a PPCF+XML **$X\Delta T$-context** if $\vdash C[M] : \mathbf{nat}$ whenever $X;\Delta \vdash M : T$.

2. Suppose that $X;\Delta \vdash M : T$ and $X;\Delta \vdash N : T$. Then $X;\Delta \vdash M \equiv N$ if for all PPCF+XML $X\Delta T$-contexts $C[\cdot]$, $C[M] \to^* n \Leftrightarrow C[N] \to^* n$.

Table 5 Additional Typing Rules for PPCF+XML.

$$\frac{\Delta \vdash M : T}{\Delta \vdash (\Lambda\alpha\,.\,M) : \forall\alpha.T}\ (\alpha \notin \mathrm{FTV}(\Delta)) \qquad \frac{\Delta \vdash M : \forall\alpha.T}{\Delta \vdash (M\ t) : T\{t/\alpha\}}$$

$$\frac{\Delta \vdash M : \mathbf{nat} \quad \Delta \vdash N : \mathbf{nat} \quad \Delta \vdash P : \mathbf{nat}}{\Delta \vdash (\mathbf{pifz}\ M\ N\ P) : \mathbf{nat}} \qquad \frac{\Delta \vdash M : T \quad \Delta, x : T \vdash N : t}{\Delta \vdash (\mathbf{let}\ x : T = M\ \mathbf{in}\ N) : t}$$

Table 6 Additional Reduction Rules for PPCF+XML.

$$((\Lambda\alpha.\,M)\ s) \to M\{s/\alpha\} \qquad (\mathbf{pifz}\ 0\ M\ N) \to M$$

$$(\mathbf{let}\ x = N\ \mathbf{in}\ M) \to M\{N/x\} \qquad (\mathbf{pifz}\ (\mathbf{succ}\ n)\ M\ N) \to N$$

$$(\mathbf{pifz}\ M\ n\ n) \to n$$

$$\frac{M \to M'}{(\mathbf{pifz}\ M\ N\ P) \to (\mathbf{pifz}\ M'\ N\ P)} \qquad \frac{N \to N'}{(\mathbf{pifz}\ M\ N\ P) \to (\mathbf{pifz}\ M\ N'\ P)}$$

$$\frac{P \to P'}{(\mathbf{pifz}\ M\ N\ P) \to (\mathbf{pifz}\ M\ N\ P')} \qquad \frac{M \to M'}{(M\ t) \to (M'\ t)}$$

3 Translation from PCF to PPCF+XML

Pick a type variable α. Given a PCF type t we obtain a PPCF+XML type t^* by replacing each occurrence of **nat** by α. Assume that the type context of the term to be translated is $\Gamma = x_1 : t_1, \ldots, x_n : t_n$. We build the translation in a few stages:

1. Define the type $\mathrm{Cl}(\Gamma t) = t_1^* \to \cdots \to t_n^* \to t^*$ (Cl here is for "closure"). If Γ is empty then this type is just t^*.
2. For a term M, let $M^*_{\Gamma t} = (\lambda x_1 : t_1 \ldots \lambda x_n : t_n \,.\, M)$. Notice that this depends on the ordering of the $x_i : t_i$'s in Γ. Again, if Γ is empty then $M^*_{\Gamma t}$ is just M. If $\Gamma \vdash M : t$ is a derivable PCF typing judgement then clearly we have $\vdash M^*_{\Gamma t} : \mathrm{Cl}(\Gamma t)\{\mathbf{nat}/\alpha\}$.
3. The type scheme $\mathrm{Con}(\Gamma t)$ (Con is for "context") is

$$\forall \alpha.\, (\alpha \to \alpha) \to (\alpha \to \alpha) \to \alpha \to (\alpha \to \alpha \to \alpha \to \alpha) \to \mathrm{Cl}(\Gamma t) \to \alpha$$

4. The translation of M, with respect to Γt, is the term $[\![M]\!]_{\Gamma t}$ given by

$$[\![M]\!]_{\Gamma t} \;=\; p\, \mathbf{nat}\ \mathit{succ}\ \mathit{pred}\ 0\ \mathit{ifz}\ M^*_{\Gamma t}$$

where $\mathit{succ} = (\lambda x : \mathbf{nat} \,.\, \mathbf{succ}\ x)$ and so on. In $[\![M]\!]$, p is any variable. If $\Gamma \vdash M : t$ is a derivable typing judgement in PCF, then

$$p : \mathrm{Con}(\Gamma t) \vdash [\![M]\!]_{\Gamma t} : \mathbf{nat}$$

is derivable in the polymorphic language.

The main result is that the translation $[\![\cdot]\!]$ preserves and reflects observational equivalence.

Theorem 4 Full Abstraction. *Suppose $\Gamma \vdash M : t$ and $\Gamma \vdash N : t$. Then*

$$\Gamma \vdash M \equiv N \iff p : \mathit{Con}(\Gamma t) \vdash [\![M]\!]_{\Gamma t} \equiv [\![N]\!]_{\Gamma t}$$

Proof. (Sketch) The ($\Leftarrow$) direction is not difficult. The main steps in the ($\Rightarrow$) direction are as follows.

1. Show that if $[\![M]\!]_{\Gamma t}$ and $[\![M]\!]_{\Gamma t}$ are distinguishable, they are distinguishable in a context of the form (**let** $p : \mathrm{Con}(\Gamma t) = P$ **in** $[\cdot]$). This follows from a version of the Context Lemma [10].
2. Show that, for the purpose of making such distinctions, it is sufficient to consider P's defined from the pure simply-typed λ-calculus involving only a divergent term Ω. The polymorphic type of p is essential for this.
3. Prove that for P as in 2, (**let** $p : \mathrm{Con}(\Gamma t) = P$ **in** $[[\![M]\!]]$) reduces to $C[M]$, where $C[\cdot]$ is a sequential PCF context, and similarly for N.

These properties allow us to construct a sequential PCF context that distinguishes M and N when $[\![M]\!]$ and $[\![N]\!]$ are inequivalent in PPCF+XML.

4 Translation from IA to PPCF+XML

The translation from IA to PPCF+XML goes in two stages. We define a denotational semantics-style encoding of IA into PCF called the concrete translation. We then give an abstract translation which uses the concrete translation.

4.1 The Concrete Translation

The concrete translation, on the level of types, goes as follows.

$$\begin{array}{rl@{\qquad}l} \mathcal{C}[\![\mathbf{comm}]\!] &= \mathbf{nat} \to S \to S & L = \mathbf{nat} \\ \mathcal{C}[\![\mathbf{exp}]\!] &= \mathbf{nat} \to S \to V & V = \mathbf{nat} \\ \mathcal{C}[\![\mathbf{loc}]\!] &= \mathbf{nat} \to S \to L & S = L \to V \\ \mathcal{C}[\![\theta \to \theta']\!] &= \mathcal{C}[\![\theta]\!] \to \mathcal{C}[\![\theta']\!] & \end{array}$$

The extra **nat** parameter in the base types is used to keep track of the number of locations that have been allocated. The translation on terms is

$$\begin{array}{ll} \mathcal{C}[\![y]\!] & = y \\ \mathcal{C}[\![0]\!] & = \lambda x : \mathbf{nat}.\, \lambda s : S.\, 0 \\ \mathcal{C}[\![\mathbf{succ}\ M]\!] & = \lambda x : \mathbf{nat}.\, \lambda s : S.\ \mathbf{succ}\ (\mathcal{C}[\![M]\!]\ x\ s) \\ \mathcal{C}[\![\mathbf{pred}\ M]\!] & = \lambda x : \mathbf{nat}.\, \lambda s : S.\ \mathbf{pred}\ (\mathcal{C}[\![M]\!]\ x\ s) \\ \mathcal{C}[\![\mathbf{skip}]\!] & = \lambda x : \mathbf{nat}.\, \lambda s : S.\, s \\ \mathcal{C}[\![(M\, N)]\!] & = (\mathcal{C}[\![M]\!]\ \mathcal{C}[\![N]\!]) \\ \mathcal{C}[\![\lambda z : \theta.\, M]\!] & = \lambda z : \mathcal{C}[\![\theta]\!].\, \mathcal{C}[\![M]\!] \\ \mathcal{C}[\![\mathbf{Y}_\theta\, M]\!] & = \mathbf{Y}_{\mathcal{C}[\![\theta]\!]}\ \mathcal{C}[\![M]\!] \\ \mathcal{C}[\![C_1; C_2]\!] & = \lambda x : \mathbf{nat}.\, \lambda s : S.\, \mathcal{C}[\![C_2]\!]\ x\ (\mathcal{C}[\![C_1]\!]\ x\ s) \\ \mathcal{C}[\![M := E]\!] & = \lambda x : \mathbf{nat}.\, \lambda s : S.\, (s \mid (\mathcal{C}[\![M]\!]\ x\ s) \mapsto (\mathcal{C}[\![N]\!]\ x\ s)) \\ \mathcal{C}[\![\mathbf{contents}\, M]\!] & = \lambda x : \mathbf{nat}.\, \lambda s : S.\, (s\, (\mathcal{C}[\![M]\!]\ x\ s)) \\ \mathcal{C}[\![\mathbf{ifz}\, M\, N\, P]\!] & = \lambda x : \mathbf{nat}.\, \lambda s : S.\, \mathbf{ifz}\, (\mathcal{C}[\![M]\!]\ x\ s)\, (\mathcal{C}[\![N]\!]\ x\ s)\, (\mathcal{C}[\![P]\!]\ x\ s) \\ \mathcal{C}[\![\mathbf{new}\ z = E\ \mathbf{in}\ C]\!] & = \end{array}$$

$$\begin{array}{l} \lambda x : \mathbf{nat}.\ (\lambda z : \mathcal{C}[\![\mathbf{loc}]\!].\lambda s : S.\ (\ (((\mathcal{C}[\![C]\!]\ (\mathbf{succ}\ x)\ \ (s \mid (\mathbf{succ}\ x) \mapsto (\mathcal{C}[\![E]\!]\ x\ s)))) \\ \qquad\qquad\qquad\qquad\qquad\qquad \mid (\mathbf{succ}\ x) \mapsto s(\mathbf{succ}\ x)) \\ \qquad\qquad)\ (\lambda n : \mathbf{nat}.\ \lambda s : S.\ (\mathbf{succ}\ x)) \end{array}$$

There are some obvious provisos here about free variables, e.g., in the equation for $\mathcal{C}[\![\mathbf{succ}\ M]\!]$, x cannot be free in M. The interpretation of $\mathbf{ifz}_{\mathbf{comm}}$ requires a higher-order **ifz** in PCF, but this can be easily encoded. Also, in these definitions $(s \mid a \mapsto b)$ stands for

$$\lambda \ell : L.\ \mathbf{ifz}\ (\mathbf{eq}\ b\ b)\ (\mathbf{ifz}\ (\mathbf{eq}\ \ell\ a)\ b\ s(\ell))\ \Omega$$

where $(\mathbf{eq}\ c\ d)$ is itself sugar for an expression that returns 0 if c and d are defined and equal, 1 if they are defined and unequal, and diverges if either is undefined. The $(\mathbf{eq}\ b\ b)$ test serves merely to make $(s \mid a \mapsto b)$ strict in b. It is not hard to see that the judgement $\pi \vdash M : \theta$ in IA gets translated to $\mathcal{C}[\![\pi]\!] \vdash \mathcal{C}[\![M]\!] : \mathcal{C}[\![\theta]\!]$, where $\mathcal{C}[\![\pi]\!]x = \mathcal{C}[\![\pi(x)]\!]$ for $x \in \mathrm{dom}(\pi)$.

Most of the valuations are self-explanatory except for **new**. Suppose we are evaluating $(\mathbf{new}\, z = E\ \mathbf{in}\, C)$ in a state s where there are x active locations.

Then we evaluate the body C in a state where there are $x+1$ locations, and where the extra location (which is itself simply **succ** x) has contents $\mathcal{C}[\![E]\!]\,x\,s$. This is the intuition behind $(\mathcal{C}[\![C]\!]\,(\mathbf{succ}\ x)\,(s \mid (\mathbf{succ}\ x) \mapsto (\mathcal{C}[\![E]\!]\,x\,s)))$. The $\mid (\mathbf{succ}\ x) \mapsto s(\mathbf{succ}\ x)$ part restores **succ** x to its old value on termination of the block. Finally, the argument that is passed to z simply serves to bind z to (an expression for) the new location.

The concrete translation yields a semantics that is very poor in many respects. For example, locations are represented as the type **nat**; this has the disadvantage of there being an "undefined" location. Despite this extra baggage the concrete translation is still adequate.

Theorem 5 Adequacy. $C\Downarrow \iff (\mathcal{C}[\![C]\!]\ 0\ (\lambda x : \mathbf{nat}.x)\ 0)\Downarrow$.

Proof. (Sketch) If we compose the translation $\mathcal{C}[\![\cdot]\!]$ with the usual continuous function model of PCF, then we obtain a denotational semantics of IA. The adequacy of this denotational model for IA, together with the adequacy of the continuous function model for PCF, yields the result. The adequacy of this model of IA can be shown using standard methods (as in [7]). The only subtlety involves dealing with the "extra baggage," such as non-definable commands in the semantics that are non-strict in their state argument; this requires some care when proving, using a computability argument as in [19], that operational termination implies semantic termination.

The three arguments to $\mathcal{C}[\![C]\!]$ in this result specify a context of evaluation where there are 0 initial locations and $(\lambda x : \mathbf{nat}.\,x)$ is the initial state. The final argument is a location whose contents we look up to get a **nat**.

4.2 The Abstract Translation

We translate a judgement $\pi \vdash M : \theta$ in IA to a judgement in PPCF+XML, using the translation $\mathcal{C}[\![\cdot]\!]$ as an intermediary. Assume that $\pi = x_1 : \theta_1, ..., x_n : \theta_n$, and let *comm*, *exp* and *loc* be distinct type variables. If θ is an IA type, let θ^* is the PPCF+XML type obtained by replacing occurrences of **comm** by *comm*, **exp** by *exp*, and **loc** by *loc*. The translation is defined in a few stages:

1. Define the type $\mathrm{Cl}(\pi\theta) = \theta_1^* \to \cdots \to \theta_n^* \to \theta^*$.
2. For a term M, let $M^*_{\pi\theta} = (\lambda x_1 : \mathcal{C}[\![\theta_1]\!], \ldots \lambda x_n : \mathcal{C}[\![\theta_n]\!]\,.\mathcal{C}[\![M]\!])$. If π is empty, then $M^*_{\pi\theta}$ is just $\mathcal{C}[\![M]\!]$. If $\pi \vdash M : \theta$ is a derivable IA typing judgement,
$$\vdash M^*_{\pi\theta} : \mathrm{Cl}(\pi\theta)\{\mathcal{C}[\![\mathbf{comm}]\!]/\mathit{comm}, \mathcal{C}[\![\mathbf{exp}]\!]/\mathit{exp}, \mathcal{C}[\![\mathbf{loc}]\!]/\mathit{loc}\}.$$
3. Define $\mathrm{Con}(\pi\theta)$ to be
$$\begin{aligned}\forall \mathit{comm}.\,\forall \mathit{exp}.\,\forall \mathit{loc}.\ &(\mathit{exp} \to \mathit{exp}) \to (\mathit{exp} \to \mathit{exp}) \to \mathit{exp}\\ &\to (\mathit{exp} \to \mathit{exp} \to \mathit{exp} \to \mathit{exp}) \to (\mathit{exp} \to \mathit{loc} \to \mathit{loc} \to \mathit{loc})\\ &\to (\mathit{exp} \to \mathit{comm} \to \mathit{comm} \to \mathit{comm})\\ &\to (\mathit{loc} \to \mathit{exp} \to \mathit{comm}) \to (\mathit{comm} \to \mathit{comm} \to \mathit{comm})\\ &\to (\mathit{loc} \to \mathit{exp}) \to (\mathit{exp} \to (\mathit{loc} \to \mathit{comm}) \to \mathit{comm}) \to \mathit{comm}\\ &\to \mathrm{Cl}(\pi\theta) \to \mathit{comm}.\end{aligned}$$

The types on the first line are for successor, predecessor and zero, those on the second and third lines are for conditionals, and those on the fourth and fifth lines are for assignment, sequencing, dereferencing, variable declarations, and skip.

4. The translation of M, with respect to $\pi\theta$, is the term $\mathcal{A}[\![M]\!]_{\pi\theta}$ given by

$$\begin{aligned}\mathcal{A}[\![M]\!]_{\pi\theta} = p\, \mathcal{C}[\![\mathbf{comm}]\!]\ \mathcal{C}[\![\mathbf{exp}]\!]\ \mathcal{C}[\![\mathbf{loc}]\!]\ & \textit{succ pred zero ifz}_{\mathbf{exp}}\ \textit{ifz}_{\mathbf{loc}}\ \textit{ifz}_{\mathbf{comm}} \\ & \textit{assign seq contents new skip}\ M^*_{\pi\theta} \\ & 0\ \ (\lambda x : \mathbf{nat}.x)\ \ 0\end{aligned}$$

where $\textit{succ} = \mathcal{C}[\![\lambda x : \mathbf{exp}.\ \mathbf{succ}\ x]\!]$, and so on. In $\mathcal{A}[\![M]\!]$, p is some "fresh" variable. If $\pi \vdash M : \theta$ is a derivable typing judgement in IA, then

$$p : \mathrm{Con}(\pi\theta) \vdash \mathcal{A}[\![M]\!]_{\pi\theta} : \mathbf{nat}$$

is derivable in the polymorphic language.

The term $\mathcal{A}[\![M]\!]$, minus its last three arguments, is of type $\mathcal{C}[\![\mathbf{comm}]\!]$. The final three arguments play the same role as in the statement of the adequacy of the concrete translation.

Theorem 6 Full Abstraction. *Suppose $\pi \vdash M : \theta$ and $\pi \vdash N : \theta$. Then*

$$\pi \vdash M \equiv N \quad \Longleftrightarrow \quad p : \mathit{Con}(\pi\theta) \vdash \mathcal{A}[\![M]\!] \equiv \mathcal{A}[\![N]\!]$$

Proof. The proof runs along the same lines as for Theorem 4, with the exception that step 3 needs to be modified as follows.

3. (**let** $p : \mathrm{Con}(\pi\theta) = P$ **in** $[\mathcal{A}[\![M]\!]]$) reduces to $C[\mathcal{C}[\![M]\!]]$, where $C[\cdot]$ is in the image of $\mathcal{C}[\![\cdot]\!]$, and similarly for N.

The adequacy of $\mathcal{C}[\![\cdot]\!]$ is then used to build a distinguishing context.

5 Conclusion

We have given fully abstract translations from sequential PCF and an idealized Algol into a parallel extension of PCF with ML-style polymorphism. Our translation method appears to be fairly general, and it would be interesting to attempt to apply it to other languages. One particularly important area concerns dynamic allocation [18], or a combination of dynamic allocation and reference types like that found in Standard ML. We expect that a stronger type theory than XML—probably involving the addition of recursive types—would be needed to treat references that store stateful objects, such as other references. Our translation techniques also might be applicable to other languages, such as a language with concurrency (e.g., Reppy's Concurrent ML [21]), a language with exceptions, or a language with continuations. While this discussion is speculative, more examples would lend support to the idea of parametricity as a unifying concept, a proper understanding of which may shed light on diverse problems in programming theory.

Proving that a denotational model of a programming language is fully abstract boils down to showing that the "abstraction barrier" between the abstract description of a programming language and a denotational semantics "implementation" is maintained. For instance, the standard continuous function model of PCF is *not* fully abstract because the model contains certain "parallel" functions that can be used to distinguish terms that are not distinguishable in the sequential setting [19, 26]. Similarly, standard models of Algol are not fully abstract [9]: there are functions in these model that violate the "stack discipline" of local variables and that can be used to distinguish observationally indistinguishable terms. In these models, a "denotational program" is allowed access to operations that are not provided directly by the language. Our translations into a PPCF+XML show that parametricity may be used to prevent such unauthorized access, and that a fully abstract model of PPCF+XML would provide a basis for reasoning about PCF and IA.

Nevertheless, there are two potential limitations to our results. Firstly, our translations probably do not yield "practical" methods for reasoning about PCF or IA. Reasoning about a PCF or IA term directly from the translation would involve reasoning about a large polymorphic application. Given a suitable model of PPCF+XML, it would be preferable to bypass the application of a polymorphic function and consider directly the meanings determined by such an application. One might expect that a more direct characterization of such meanings might be possible with a little work, but this is difficult to predict in the absence of a fully abstract model of PPCF+XML.

Secondly, our results do not definitively establish that a solution to the full abstraction problem for PPCF+XML entails a solution to the problems for PCF and IA. One difficulty, of course, is that the word "solution" here is ill-defined. In the case of PCF, one would at least want a characterization of Milner's unique model [10] (which we believe would be possible) together with a characterization of finite elements and some "semantic" conditions explaining the "sequential" nature of the function type (which is not immediate from our translation). In the case of IA, one would perhaps like to see structure, such as that in functor category models [23, 15, 13], explaining the *variance* in the concept of state that is caused by variable declarations. Again, though we believe that there may be reasonable possibilities in this direction, without a fully abstract model of PPCF+XML it is not possible to predict with assurance that "good" models of PCF or IA would be obtained.

In this paper we purposely restricted our attention to ML-style polymorphism in order to emphasize the point that issues of parametricity and full abstraction are interesting even for this limited variety of polymorphism. Indeed, most studies of parametricity have taken place in the context of the second-order polymorphic λ-calculus (*e.g.* [2, 4, 8, 24]). ML's polymorphism is of a comparatively simple form, where type variables never themselves get instantiated to polymorphic types. This *predicative* flavour would allow parametricity to be examined in isolation from the foundational issues raised by the impredicative polymorphism of the second-order λ-calculus. Thus, ML's polymorphism might serve as a useful test bed for examining rigorous explanations of parametricity, as an intermediate step on the way to a more general understanding.

ML-style polymorphic types can be interpreted quite straightforwardly as indexed products of domains, trimmed by Reynolds's relational-parametricity conditions [24]; using the Kripke logical relations of [6] may work especially well. There are also a number of PER models that possess appropriate domain-theoretic structure (*e.g.* [3, 17]). Some of these models perform quite well at certain "low-order" types, but little is known at higher types, and it is not known if any of them is fully abstract. The presence of **pif** in PPCF+XML may help in providing such a semantic model (cf. [19]), e.g., in considering definability results for polymorphic types with embedded occurrences of **nat**. Of course, full abstraction for the polymorphic language without **pif** (not so far-fetched a possibility in light of some recent preliminary announcements [1, 16]) would also be interesting in connection to the results presented here for the purpose of studying Algol (our translation result for IA does not *require* **pif**).

One aspect of state that we have not emphasized—but which is at least as fundamental as local variables—is what is often called the "single threaded" nature of state: when a state change occurs, the old state disappears. Even the most advanced models of Algol—e.g., the ones described in [14, 27]—are known not to be fully abstract without what Reynolds calls "snap-back" operations. These operations violate the single-threaded nature of Algol by allowing backtracking of state changes during evaluation, requiring the maintenance of a stack of (temporary) states instead of only one. It is interesting, then, that our translation results do not require the presence of snap-back operations. This is very similar to the connection between single-threading and abstract types suggested by Wadler [29]. Wadler's observations were made in the context of functional programming, where single-threadedness is needed for the safety of in-place array update, but they also appear to be relevant to the understanding of semantic aspects of single-threading. Reddy [20] has developed a different approach to single-threading by applying and extending some ideas from linear logic.

An important precursor to this work is the paper of O'Hearn and Tennent [14], where a connection between block structure and parametricity was first proposed (a related work is [27]). The information-hiding that is obtained through encapsulating pieces of local state was explained there using a denotational model that borrowed ideas from Reynolds's relational approach to parametricity. They obtain good characterizations of the structure of "low order" Algol types (related to initial algebra results for polymorphism, e.g. [24, 4]), but full abstraction issues have not been resolved for their models. In contrast, we obtain a (syntactic) full abstraction result, but not a direct characterization of "low order" types. Another difference is that our translation only requires ML-style polymorphism, whereas their semantics contains parametric functions as arguments to other functions. While neither our translation nor our results are directly analogous to [14], we consider our results as providing further evidence in favour of a connection between local state and parametricity.

References

1. S. Abramsky, R. Jagadeesan, and P. Malacaria. Games and full abstraction for PCF: preliminary announcement. Unpublished, 1993.
2. S. Bainbridge, P. J. Freyd, A. Scedrov, and P. J. Scott. Functorial polymorphism. *Theoretical Computer Science*, 70(10):35–64, 1990. *Corrigendum* in 71(3):431, 1990.
3. P. J. Freyd, P. Mulry, G. Rosolini, and D. S. Scott. Extensional PERs. In *Proceedings, Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 346–354, Philadelphia, PA, 1990. IEEE Computer Society Press, Los Alamitos, California.
4. P. J. Freyd, E. P. Robinson, and G. Rosolini. Functorial parametricity. In *Proceedings, 7th Annual IEEE Symposium on Logic in Computer Science*, pages 444–452, Santa Cruz, California, 1992. IEEE Computer Society Press, Los Alamitos, California.
5. R. Harper and J. C. Mitchell. On the type structure of standard ML. *ACM Trans. Programming Languages and Systems*, 15:211–252, 1993.
6. A. Jung and J. Tiuryn. A new characterization of lambda definability. In *Typed Lambda Calculi and Applications*, volume 664 of *Lect. Notes in Computer Sci.*, pages 245–257. Springer-Verlag, 1993.
7. A. F. Lent. The category of functors from state shapes to bottomless CPOs is adequate for block structure. Master's thesis, Massachusetts Institute of Technology, 1992.
8. Q. Ma and J. C. Reynolds. Types, abstraction, and parametric polymorphism, part 2. In S. Brookes et al., editors, *Mathematical Foundations of Programming Semantics*, volume 598 of *Lecture Notes in Computer Science*, pages 1–40. Springer-Verlag, Berlin, 1992. Proceedings of the 1991 Conference.
9. A. R. Meyer and K. Sieber. Towards fully abstract semantics for local variables: preliminary report. In *Conf. Record 15th ACM Symp. on Principles of Programming Languages*, pages 191–203. ACM, New York, 1988.
10. R. Milner. Fully abstract models of typed λ-calculi. *Theoretical Computer Science*, 4:1–22, 1977.
11. R. Milner. A theory of type polymorphism in programming. *J. of Computer and System Sciences*, 17:348–75, 1978.
12. J. C. Mitchell and R. Harper. The essence of ML. In *Conf. Record 15th ACM Symp. on Principles of Programming Languages*, pages 28–46. ACM, New York, 1988.
13. P. W. O'Hearn and R. D. Tennent. Semantics of local variables. In M. P. Fourman, P. T. Johnstone, and A. M. Pitts, editors, *Applications of Categories in Computer Science*, volume 177 of *London Mathematical Society Lecture Note Series*, pages 217–238. Cambridge University Press, Cambridge, England, 1992.
14. P. W. O'Hearn and R. D. Tennent. Parametricity and local variables. Technical Report SU-CIS-93-30, Syracuse University, 1993. Preliminary version appeared in *Conf. Record 20th ACM Symp. on Principles of Programming Languages*, pages 171–184, Charleston, South Carolina, 1993. ACM, New York.
15. F. J. Oles. *A Category-Theoretic Approach to the Semantics of Programming Languages*. Ph.D. thesis, Syracuse University, Syracuse, N.Y., 1982.
16. L. Ong and M. Hyland. Dialogue games and innocent strategies: An approach to intensional full abstraction to PCF (preliminary announcement). Unpublished, 1993.

17. W. K. Phoa. Effective domains and intrinsic structure. In *Proceedings, Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 366–379, Philadelphia, PA, 1990. IEEE Computer Society Press, Los Alamitos, California.
18. A. Pitts and I. Stark. On the observable properties of higher-order functions that dynamically create local names (preliminary report). In *ACM SIGLPLAN Workshop on State in Programming Languages*, pages 31–45, 1993. Available as Yale Technical Report YALEU/DCS/RR-968.
19. G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.
20. U.S. Reddy. Global states considered unnecessary. In *ACM SIGLPLAN Workshop on State in Programming Languages*, 1993. Available as Yale Technical Report YALEU/DCS/RR-968.
21. J. H. Reppy. CML: A higher-order concurrent language. In *Proceedings of the SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 293–305, June 1991.
22. J. C. Reynolds. Towards a theory of type structure. In *Proc. Colloque sur la Programmation*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425, Berlin, 1974. Springer-Verlag.
23. J. C. Reynolds. The essence of Algol. In J. W. de Bakker and J. C. van Vliet, editors, *Algorithmic Languages*, pages 345–372. North-Holland, Amsterdam, 1981.
24. J. C. Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, pages 513–523. North Holland, Amsterdam, 1983.
25. J. G. Riecke. Fully abstract translations between functional languages. *Mathematical Structures in Computer Science*, 1993. To appear.
26. V. Sazonov. Expressibility of functions in D. Scott's LCF language. *Algebra i Logika*, 15:308–330, 1976. Russian.
27. K. Sieber. New steps towards full abstraction for local variables. In *ACM SIGLPLAN Workshop on State in Programming Languages*, pages 88–100, 1993. Available as Yale Technical Report YALEU/DCS/RR-968.
28. C. Strachey. *Fundamental Concepts in Programming Languages*. Unpublished lecture notes, International Summer School in Computer Programming, Copenhagen, August 1967.
29. P. Wadler. Comprehending monads. In *Proceedings of the ACM Conference on* LISP *and Functional Programming*, pages 61–78, Nice, 1990.