Operational and Axiomatic Semantics of PCF

Brian T. Howard[*]
John C. Mitchell[†]
Department of Computer Science
Stanford University
{bhoward,jcm}@cs.stanford.edu

Abstract

PCF, as considered in this paper, is a lazy typed lambda calculus with functions, pairing, fixed-point operators and arbitrary algebraic data types. The natural equational axioms for PCF include η-equivalence and the so-called "surjective pairing" axiom for pairs. However, the reduction system $pcf_{\eta,sp}$ defined by directing each equational axiom is not confluent, for virtually any choice of algebraic data types. Moreover, neither η-reduction nor surjective pairing seems to have a counterpart in ordinary execution. Therefore, we consider a smaller reduction system *pcf* without η-reduction or surjective pairing. The system *pcf* is confluent when combined with any linear, confluent algebraic rewrite rules. The system is also computationally adequate, in the sense that whenever a closed term of "observable" type has a $pcf_{\eta,sp}$ normal form, this is also the unique *pcf* normal form. Moreover, the equational axioms for PCF, including (η) and surjective pairing, are sound for *pcf* observational equivalence. These results suggest that if we take the equational axioms as defining the language, the smaller reduction system gives an appropriate operational semantics.

1 Introduction

Most systems of lambda calculus have three parts: an equational proof system, a set of reduction rules, and a model theory. These correspond to the standard programming language notions of axiomatic, operational, and denotational semantics. To a first approximation, the connections between axiomatic and operational semantics are straightforward in basic systems such as the simply-typed lambda calculus. The reduction rules may be derived by orienting each equational axiom in a computationally reasonable way and the resulting system is confluent. As a result, the reduction rules serve simultaneously as a useful characterization of equational provablilty and as a natural model of execution. When we add recursion to simply-typed lambda calculus with cartesian products, this straightforward correspondence breaks down. Since confluence fails [Nes89], the reduction rules do not give a good picture of equational provability. Moreover, upon examining the reduction rules more carefully, many investigators have come to the conclusion that neither η-reduction nor surjective pairing is computationally compelling. In fact, it seems to be a common view that η-reduction and surjective pairing do not have any "computational content." Therefore, for both technical and intuitive reasons, we are led to define evaluation without these reduction rules.

In this paper, we study a simply-typed lambda calculus with functions, pairing, fixed-point operators and arbitrary algebraic data types. We will refer to this language as PCF, since it is based on the calculus considered in Plotkin's seminal paper [Plo77], with pairing and algebraic data types added.[1] If we include algebraic data types of natural numbers and booleans, as in [Plo77, Sco69], then it is easy to program any partial recursive function on the natural numbers. With algebraic data types of trees, lists, stacks, and so on, we may write common functional programs in the style of Miranda or Lazy ML, for example [Tur85].

While most sequential implementations of lazy languages are based on a deterministic (typically "left-most") evaluation order, there are several reasons to study arbitrary order. One motivation is parallel execution. If the result of evaluation does not rely on evaluation order, then many subexpressions may safely be evaluated in parallel (see [Hen80, Pey87], for example, for related discussion). Another reason to consider arbitrary evaluation order is to identify desirable properties of a particular implementation. For example, if a set of reduction (or execution) rules is confluent, then the result of nondeterministic execution is well-defined and we may regard this as the "ideal" implementation. We may then show that a particular evaluation order is satisfactory by comparison with nondeterministic evaluation. While we do not analyze any deterministic evaluation strategies in this paper, our study of nondeterministic reduction sets the stage for such later study.

For any variant of PCF with equationally-axiomatized algebraic data types, there is a traditional and accepted equational proof system. The axioms of this proof system include η-equivalence (extensionality) for functions

$$(\eta)_{eq} \qquad \lambda x{:}\sigma.\, Mx = M, \quad x \text{ not free in } M,$$

[*]This material is based upon work supported under a National Science Foundation Graduate Fellowship

[†]Supported in part by an NSF PYI Award, matching funds from Digital Equipment Corporation, the Powell Foundation, and Xerox Corporation, and NSF grant CCR-8814921.

[1]The language itself seems attributable to Scott, since the basic ideas are presented in the manuscript [Sco69] and Plotkin's name for the calculus is clearly derived from Scott's LCF (*Logic for Computable Functions).*

and the *surjective pairing* axiom

$$(sp)_{eq} \qquad\qquad \langle \pi_1 P, \pi_2 P\rangle = P,$$

where π_1 and π_2 are the first and second projection functions. These seem essential for proving common facts about functions and pairs. For example, both are needed to establish that *currying* and *uncurrying* are inverses (or, equivalently, that types $\sigma\to(\tau\to\rho)$ and $(\sigma\times\tau)\to\rho$ are isomorphic). However, neither seems to be used in standard implementations. If we direct these equational axioms from left to right, we obtain evaluation rules, (η) and (sp), that could be used in program execution but which typically do not have any counterpart in practical implementation. One reason surjective pairing is difficult to implement is that it is *non-linear*: the meta-variable P occurs twice on the left-hand side. In order to apply the reduction, we must therefore test two potentially large subexpressions for syntactic equality. This seems inefficient, in general, and it follows from our results that such a test is unnecessary for execution of complete programs.

In addition to the folkloric view that (η) and (sp) are not needed in computation, which we justify in Corollary 6.6, there are technical problems with these rules. While pure typed lambda calculus with these rules but *without* fixed-point operators is confluent [Pot81], the situation is substantially different in the presence of recursion. Even without any algebraic data types, (sp) and recursion cause confluence to fail. This may be demonstrated by adapting Klop's well-known counterexample[2] to confluence in the untyped lambda calculus with surjective pairing [Nes89]. Similar reasoning also shows that in a language with recursion, confluence may fail when confluent but non-linear algebraic rules are added. Although not as immediately problematic, (η) also interferes with confluence of algebraic rules. For example, the simple rewrite rule $f(x)\to a$, combined with (η), is not confluent. Therefore, considering only the technical property of confluence, both (η) and (sp) seem problematic.

Our first theorem is that without (η) and (sp), PCF reduction is confluent over any algebraic data types, provided that the algebraic rewrite rules are all linear, and confluent when considered apart from PCF. This result, described in Section 5, extends a similar theorem of [BT88] to pairing and fixed-point operators. Our proof technique combines labelled reduction [Bar84] with the method of [BT88], which relies on strong normalization. In combining lambda calculus with additional reduction rules, our theorem is also similar in spirit to the delta reduction theorem of Mitschke (see [Bar84]), although our result neither subsumes nor is subsumed by his. Given the apparent suitability of PCF reduction without (η) and (sp), we proceed to study connections between this limited reduction system and the natural equational axioms.

Since we have dropped two equational rules, it is not immediately clear whether the reduction system is computationally adequate. In other words, do we still have "enough" evaluation rules? While there are provably equal expressions, such as $\lambda x\colon nat.fx$ and f, which do not reduce to a common form, we show that this is not the case for "full programs." In more detail, the programs we execute in practice are closed expressions of basic types such as *nat*, *bool* and *list*, or perhaps products of basic types. We show that for any closed expression M of "observable" type, and possible result N of complete execution, M reduces to N using all reduction rules iff M reduces to N without (η) and (sp). Thus we have not lost anything by dropping these rules. Moreover, and perhaps more importantly, such expressions M and N are provably equal, possibly using $(\eta)_{eq}$ and $(sp)_{eq}$, iff M reduces to N by reduction without (η) and (sp). In standard computational terms, this demonstrates the computational adequacy of our operational semantics with respect to the axiomatic semantics (*c.f.* [HWWW85]). The first of these theorems follows from a postponement property of (η) and (sp), while the second uses a refinement of the postponement proof. Typing is essential here, since (sp) postponement fails for untyped terms.[3]

Our final connection between axiomatic and operational semantics is soundness of the axioms with respect to computation. In general terms, if we begin with an intuitively appealing axiomatic semantics, then it seems fair to base evaluation on any equational principles which follow from the axioms. However, once we have selected an operational semantics, we must also ask whether this semantics justifies the equational prinicples we began with. Put simply, are the equational axioms sound statements about the result of program execution? We answer this question (affirmatively) using the standard notion of *observational congruence*. This is the natural equivalence relation generated by execution of full programs, and as the name implies it is a congruence relation. Briefly, two expressions are observationally congruent iff they are interchangeable in all programs. Put another way, expressions M and N, which may be higher-order functions or other "non-programs," are observationally congruent iff we may replace any occurrence of M in a full program by N without affecting the result of program execution. In Section 7, we show that axioms $(\eta)_{eq}$ and $(sp)_{eq}$, and hence all the equational rules, are sound for observational congruence. This shows that although we have eliminated some equational principles from execution, we have not invalidated our axiomatic semantics. The proof uses postponement of (η) and (sp).

2 Signatures and terms

The language PCF may be defined over any collection of base types (sorts) and constant symbols. The base types might include natural numbers and booleans or atoms, lists, trees and so on. Since PCF is a typed language, each constant symbol must have a type. Typical constants include the number 3, + for natural number addition, and the list operation *cons*. A difference between PCF and the pure typed lambda calculus is that we assume a *fixed-point constant* $fix_\sigma\colon(\sigma\to\sigma)\to\sigma$ for each type σ.

Using b to stand for any base type, the type expressions of PCF are defined by the grammar

$$\sigma ::= b \mid \sigma_1\to\sigma_2 \mid \sigma_1\times\sigma_2.$$

For example, if *nat* is a base type, then $nat\to(nat\times nat)$ is a type. We may avoid writing too many parentheses by adopting the convention that $\to$ associates to the right and $\times$ has a higher precedence than $\to$. Thus $nat\times nat\to nat\to nat$ is the type of functions which, given a pair of natural numbers, return a numeric function.

A PCF *signature* $\Sigma = \langle B, C\rangle$ consists of

[2] A slick presentation is given in [Bar84]. However, the exact counterexample given there cannot be typed. Instead, one must consider a version of the counterexample given in Klop's thesis [Klo80].

[3] The untyped term $\langle\pi_1(\lambda x.\,x), \pi_2(\lambda x.\,x)\rangle(\lambda x.\,x)$ provides a simple counterexample.

- a set B whose elements are called *base types* or *type constants*, and
- a collection C of pairs $\langle c, \sigma\rangle$, where c is called a *term constant* and σ is an *algebraic* type expression over B, of the form $b_1 \to \ldots \to b_n \to b$ for base types $b_1, \ldots, b_n, b$, with $n \geq 0$.

We require that no term constant appear with more than one type, and that the term constants be disjoint from type constants and other syntactic classes of the language. If $\langle c, \sigma\rangle \in C$, then c is said to be a *constant symbol of type* σ, and we sometimes write c^σ when convenient. Note that the base types and term constants must be consistent, in that the type of each constant may only contain the given base types. For example, it only makes sense to have a natural number constant 3^{nat} when we have *nat* as a base type. Note that constants have the curried type $b_1 \to \ldots \to b_n \to b$ instead of the more usual type $b_1 \times \ldots \times b_n \to b$; this is chiefly a matter of preference, since all of the results in this paper hold in either case. The choice of curried functions simplifies some proofs since we do not have to worry about *sp*-redexes in the arguments of functions.

Before defining the syntax of terms over a given signature, we choose some infinite set $\mathcal{V}$ of variables. We will give the well-formed terms and their types using an inference system for typing assertions

$$\Gamma \triangleright M \colon \tau,$$

where Γ is a *type assignment* of the form

$$\Gamma = \{x_1 \colon \sigma_1, \ldots, x_k \colon \sigma_k\},$$

with no x_i occurring twice. Intuitively, the assertion $\Gamma \triangleright M \colon \tau$ means that if variables $x_1, \ldots, x_k$ have types $\sigma_1, \ldots, \sigma_k$ (respectively), then M is a well-formed term of type τ. If Γ is any type assignment, we will write $\Gamma, x \colon \sigma$ for the type assignment

$$\Gamma, x \colon \sigma \;=\; \Gamma \cup \{x \colon \sigma\}.$$

In doing so, we always assume that x does not appear in Γ.

The atomic expressions of PCF over the signature $\Sigma = \langle B, C\rangle$ are given by typing axioms. The typing axiom

$$(cst) \quad \emptyset \triangleright c \colon \sigma, \quad \text{provided } \langle c, \sigma\rangle \in C, \text{ or } c \text{ is } \mathit{fix}_\tau \text{ and } \sigma = (\tau \to \tau) \to \tau$$

says that each constant symbol c^σ is a term of type σ. Variables are given by the axiom

$$(var) \qquad x \colon \sigma \triangleright x \colon \sigma,$$

which says that a variable x has whatever type it is declared to have. The compound expressions and their types are defined by the following inference rules.

$$(\times\ \text{Intro}) \qquad \frac{M \colon \sigma,\ N \colon \tau}{\langle M, N\rangle \colon \sigma \times \tau}$$

$$(\times\ \text{Elim}) \qquad \frac{M \colon \sigma \times \tau}{\pi_1 M \colon \sigma, \quad \pi_2 M \colon \tau}$$

$$(\to\ \text{Intro}) \qquad \frac{\Gamma, x \colon \sigma \triangleright M \colon \tau}{\Gamma \triangleright (\lambda x \colon \sigma . M) \colon \sigma \to \tau}\ .$$

$$(\to\ \text{Elim}) \qquad \frac{\Gamma \triangleright M \colon \sigma \to \tau,\ \Gamma \triangleright N \colon \sigma}{\Gamma \triangleright MN \colon \tau}$$

$$(\mathit{add\ var}) \qquad \frac{\Gamma \triangleright M \colon \sigma}{\Gamma, x \colon \tau \triangleright M \colon \sigma}$$

The final rule allows us to add variables to the type assignment.

We say *M is a PCF term over signature Σ with type τ in context* Γ if $\Gamma \triangleright M \colon \tau$ is either a typing axiom for Σ, or follows from axioms by the typing rules. We often write $\Gamma \triangleright M \colon \tau$ to mean that "$\Gamma \triangleright M \colon \tau$ is derivable," in much the same way as one often writes a formula $\forall x . P(x)$ in logic, as a way of saying "$\forall x . P(x)$ is true." A term M is *algebraic* if it is of base type and is formed from only algebraic constants and variables of base type, using only ($\to$ Elim), *i.e.*, application.

The free and bound occurrences of a variable x in term M have the usual inductive definition. In particular, a variable x occurs free unless it is within the scope of λx, in which case it becomes bound. Since the name of a bound variable is not important, we will generally identify terms that differ only in the names of bound variables. We will write $[N/x]M$ for the result of substituting N for free occurrences of x in M, with renaming of bound variables as usual to avoid capture. We use the notion of a *context* for substitution without renaming: a context $\mathcal{C}[\ ]$ for a type σ and variable assignment Γ is a term with a "hole", such that if $\Gamma \triangleright M \colon \sigma$, then $\mathcal{C}[M]$ will be a well-typed term, formed by "plugging" M into the hole.

Lemma 2.1 *If $\Gamma \triangleright M \colon \sigma$, then every free variable of M appears in Γ.*

Lemma 2.2 *If $\Gamma \triangleright M \colon \sigma$ and $\Gamma' \subseteq \Gamma$ contains all the free variables of M, then $\Gamma' \triangleright M \colon \sigma$.*

Lemma 2.3 *If $\Gamma, x \colon \sigma \triangleright M \colon \tau$ and $\Gamma \triangleright N \colon \sigma$ are well-typed terms, then so is the substitution instance $\Gamma \triangleright [N/x]M \colon \tau$.*

3 Equations and reduction rules

Typed equations have the form

$$\Gamma \triangleright M = N : \tau,$$

where we assume that M and N have type τ in context Γ. Intuitively, the equation

$$\{x_1 \colon \sigma_1, \ldots, x_k \colon \sigma_k\} \triangleright M = N : \tau$$

means that for all type-correct values of the variables $x_1 \colon \sigma_1$ through $x_k \colon \sigma_k$, expressions M and N denote the same element of type τ.

The equational proof system for PCF is standard, with axiom

$$(\mathit{fix})_{eq} \quad \Gamma \triangleright \mathit{fix}_\sigma = \lambda f \colon \sigma \to \sigma . \, f(\mathit{fix}_\sigma\, f) : (\sigma \to \sigma) \to \sigma$$

for each fixed-point operator. The remaining axioms, $(\beta)_{eq}$, $(\eta)_{eq}$, $(\pi)_{eq}$, and $(sp)_{eq}$, resemble the corresponding reduction rules given below. Since we include type assignments in equations, we have an equational version of the structural rule

$$(add\ var) \qquad \frac{\Gamma \triangleright M = N : \sigma}{\Gamma, x\colon \tau \triangleright M = N : \sigma}$$

which lets us add an additional typing hypothesis.

Reduction is a "directed" form of equational reasoning that we will adopt as a form of symbolic evaluation. Technically, reduction is a relation on α-equivalence classes of terms. While we are only interested in reducing typed terms, we will define reduction without mentioning types. Since reduction models program execution, this is a way of emphasizing that execution does not depend on the types of terms. We will formulate reduction so that the type of a term does not change as it is reduced.

The "logical" axioms of reduction are

$$(\mathit{fix}) \qquad \mathit{fix}_\sigma \longrightarrow \lambda f\colon \sigma\to\sigma.\, f(\mathit{fix}_\sigma\, f)$$

for fixed-point operators, and the following standard reduction rules for functions and pairs:

$$(\beta) \qquad (\lambda x\colon\sigma.M)N \longrightarrow [N/x]M$$

$$(\eta) \qquad \lambda x\colon\sigma.Mx \longrightarrow M, \text{ provided } x \text{ not free in } M$$

$$(\pi) \qquad \pi_i\langle M_1,\, M_2\rangle \longrightarrow M_i, \text{ for } i = 1, 2$$

$$(sp) \qquad \langle \pi_1 P,\, \pi_2 P\rangle \longrightarrow P.$$

We also allow any set $\mathcal{R}$ of algebraic rewrite rules of the form $M \to N$, where M and N are algebraic terms over Σ with $\Gamma \triangleright M\colon b$ and $\Gamma \triangleright N\colon b$ for some typing context Γ and basic type b. Additional restrictions on M and N are that M may not be a variable and all the variables in N must occur in M. If no variable occurs twice in M, the rule $M \to N$ is said to be *left-linear*, or simply *linear.* We write $\mathcal{E}_\mathcal{R}$ for the set of all well-typed equations $\Gamma \triangleright M = N : b$ with $(M \to N) \in \mathcal{R}$.

A term of the form $(\lambda x\colon\sigma.M)N$ is a *β-redex*, $\lambda x\colon\sigma.Mx$ is an *η-redex*, and similarly for (π) and (sp). We say M *reduces to N in one step,* written $M \to N$, if N can be obtained by applying a single reduction rule to some subterm of M. To emphasize that a rule r is used, we write $M \xrightarrow{r} N$. As usual, $\twoheadrightarrow$ is the reflexive and transitive closure of one-step reduction. A term M is in *normal form* if there is no N with $M \to N$.

Using Lemma 2.3, inspection of the logical rules, and the constraints on variables in algebraic rules, it is easy to show that one-step reduction preserves type.

Lemma 3.1 *If $\Gamma \triangleright M\colon\sigma$, and $M \to N$, then $\Gamma \triangleright N\colon\sigma$.*

It follows by an easy induction that $\twoheadrightarrow$ also preserves type.

A critical property of reduction systems is *confluence,* which may be drawn graphically as follows.

$$\begin{tikzcd}
 & M \arrow[dl] \arrow[dr] & \\
N_1 \arrow[dr] & & N_2 \arrow[dl] \\
 & P &
\end{tikzcd}$$

In this picture, the top two arrows are universally quantified, and the bottom two existentially, so the picture "says" that whenever $M \twoheadrightarrow N_1$ and $M \twoheadrightarrow N_2$, there exists a term P such that $N_1 \twoheadrightarrow P$ and $N_2 \twoheadrightarrow P$. In lambda calculus, it is traditional to say that a confluent notion of reduction is *Church-Rosser,* since confluence for untyped lambda calculus was first proved by Church and Rosser [Chu41].

The convertibility relation $\leftrightarrow$ on typed terms is the least type-respecting equivalence relation containing reduction. This can be visualized by saying that $\Gamma \triangleright M \leftrightarrow N : \sigma$ iff there is a sequence of terms $M_0, \ldots, M_k$ with $\Gamma \triangleright M_i\colon\sigma$ such that

$$M \equiv M_0 \twoheadrightarrow M_1 \twoheadleftarrow \ldots \twoheadrightarrow M_k \equiv N.$$

In this picture, the directions of $\twoheadrightarrow$ and $\twoheadleftarrow$ should not be regarded as significant. (However, by reflexivity and transitivity of $\twoheadrightarrow$, the order of reduction and "backward reduction" is completely general.) A few words are in order regarding the assumption that $\Gamma \triangleright M_i\colon\sigma$ for each i. For pure typed lambda calculus, this assumption is not necessary; if $\Gamma \triangleright M \leftrightarrow N : \sigma$ and $\Gamma \cap \Gamma'$ mentions all free variables of M and N, then $\Gamma' \triangleright M \leftrightarrow N : \sigma$. However, with algebraic rewrite rules this fails.[4] For conversion as defined here, we have $\Gamma \triangleright M \leftrightarrow N : \sigma$ using rules from $\mathcal{R}$ iff $\mathcal{E}_\mathcal{R} \vdash \Gamma \triangleright M = N : \sigma$, regardless of confluence. Therefore, we only mention convertiblity in the sequel.

We will write pcf_0 for the fixed-point and lambda calculus reduction rules (fix), (β) and (π), and write pcf for $\mathit{pcf}_0 + \mathcal{R}$, where $\mathcal{R}$ is our chosen set of algebraic rules. We will write $\mathit{pcf}_{\eta,sp}$ for $\mathit{pcf} + (\eta) + (sp)$. Various labelled versions of these reductions will be introduced for technical purposes in the following sections.

4 Example: natural numbers and booleans

An example signature for PCF provides booleans and natural numbers. The basic boolean expressions are the constants *true* and *false*, and boolean-valued conditional

$$\texttt{if } \langle \mathit{bool}\rangle \texttt{ then } \langle \mathit{bool}\rangle \texttt{ else } \langle \mathit{bool}\rangle.$$

The basic natural number expressions include *numerals*

$$0, 1, 2, 3, \ldots,$$

the usual symbols for natural numbers, and addition, written $+$. Thus if M and N are natural number expressions, so is $M + N$.

We can also compute natural numbers using conditional tests,

$$\texttt{if } \langle \mathit{bool}\rangle \texttt{ then } \langle \mathit{nat}\rangle \texttt{ else } \langle \mathit{nat}\rangle,$$

and compare natural numbers for equality. For example, $Eq?\ 3\ 0$ has the boolean value *false*, since 3 is different from 0, but $Eq?\,5\,5 = \mathit{true}$. To summarize, the basic natural number and boolean expressions may be characterized by the following productions.

$$\begin{array}{lcl}
\langle \mathit{bool}\rangle & ::= & \mathit{true} \mid \mathit{false} \mid Eq?\,\langle \mathit{nat}\rangle\,\langle \mathit{nat}\rangle \mid \\
 & & \texttt{if } \langle \mathit{bool}\rangle \texttt{ then } \langle \mathit{bool}\rangle \texttt{ else } \langle \mathit{bool}\rangle \\
 & & \\
\langle \mathit{nat}\rangle & ::= & 0 \mid 1 \mid 2 \mid \ldots \mid \langle \mathit{nat}\rangle + \langle \mathit{nat}\rangle \mid \\
 & & \texttt{if } \langle \mathit{bool}\rangle \texttt{ then } \langle \mathit{nat}\rangle \texttt{ else } \langle \mathit{nat}\rangle
\end{array}$$

[4]Consider the algebraic rules $fx \longrightarrow c$ and $fx \longrightarrow d$, for $f\colon a\to b$. In this case, we do not want $\emptyset \triangleright c \leftrightarrow d : b$, since the equation $c = d$ is not provable without a variable x of type a.

The equational axioms for natural number and boolean expressions are straightforward. We have an infinite collection of basic axioms

$$0+0=0,\ 0+1=1,\ \ldots,\ 1+0=1,\ 1+1=2,\ \ldots$$

for addition, and two axiom schemes for each type of conditional:

$$\begin{array}{rcl}\texttt{if}\ \mathit{true}\ \texttt{then}\ M\ \texttt{else}\ N & = & M,\\ \texttt{if}\ \mathit{false}\ \texttt{then}\ M\ \texttt{else}\ N & = & N.\end{array}$$

There are infinitely many axioms for equality test, determined according to the scheme

$$\begin{array}{rcll}\mathit{Eq?}\,n\,n & = & \mathit{true}, & \text{each numeral } n,\\ \mathit{Eq?}\,m\,n & = & \mathit{false}, & m,n \text{ distinct numerals.}\end{array}$$

Each of these axioms determines a reduction rule, read from left to right. Note that we do *not* have the equational axiom $\mathit{Eq?}\,M\,M = \mathit{true}$, for arbitrary natural number expression M. The reason is that the more general reduction rule $\mathit{Eq?}\,M\,M \rightarrow \mathit{true}$ is non-linear, and confluence fails.

To see how the reduction rules allow us to evaluate basic natural number and boolean expressions, consider the expression

$$\texttt{if}\ \mathit{Eq?}\,(6+5)\,17\ \texttt{then}\ (1+1)\ \texttt{else}\ 27.$$

We cannot simplify the conditional without first producing a boolean constant *true* or *false*. This in turn requires numerals for both arguments to *Eq*?, so we begin by applying the reduction rule $6+5 \rightarrow 11$. This gives us the expression

$$\texttt{if}\ \mathit{Eq?}\,11\,17\ \texttt{then}\ (1+1)\ \texttt{else}\ 27,$$

which is simplified using a reduction rule for *Eq*? to

$$\texttt{if}\ \mathit{false}\ \texttt{then}\ (1+1)\ \texttt{else}\ 27.$$

Finally, one of the rules for conditional applies, and we produce the numeral 27. In order to simplify this expression, we needed to evaluate the test before simplifying the conditional. However, it was not necessary to simplify the number expression $1+1$ since this is discarded by the conditional. Since we may choose to reduce any subterm at any point, we could have simplified $1+1 \rightarrow 2$ between any two of the reduction steps given. With the added flexibility of "nondeterministic choice," the steps involved mimic the action of any ordinary interpreter fairly closely.

5 Confluence of the Reduction System *pcf*

We will now show that *pcf*-reduction is confluent. Standard techniques for showing confluence in the typed lambda calculus will not work directly, because the fixed-point operator allows terms that have no normal form. We will proceed by first considering a related system, $\mathit{pcf}^{\mathcal{N}}$, in which recursion is bounded, restoring strong normalization at the cost of diminished computational power. An argument due to Breazu-Tannen [BT88] will be used to show that $\mathit{pcf}^{\mathcal{N}}$ is confluent, from which we may prove that *pcf* itself is confluent.

The system $\mathit{pcf}^{\mathcal{N}}$ is formed by replacing the (*fix*) rule with a family of labelled rules, one for each type σ and natural number $n > 0$:

$$(\mathit{fix}^n) \qquad \mathit{fix}^n_\sigma \longrightarrow (\lambda f\colon \sigma\to\sigma.\, f(\mathit{fix}^{n-1}_\sigma\ f)).$$

In the absence of algebraic rules, the effect of this is to limit the number of times each fixed-point operator may reduce. The reduction system with no algebraic rules will be denoted $\mathit{pcf}^{\mathcal{N}}_0$.

We use the method of logical relations to prove that $\mathit{pcf}^{\mathcal{N}}_0$ is confluent and strongly normalizing. Following [Mit90], if we can show that a property $\mathcal{S}$ of terms (*i.e.*, a type-indexed family of predicates S^σ over terms of type σ) is *type-closed*, then that property holds for all well-formed terms. The appropriate definition of type-closed depends on the types available; for PCF we will need clauses for function and product types. For convenience, we introduce the concept of an *elimination context*, $\mathcal{E}[\ ]$, which in the case of PCF is a context with a single hole at the head of some sequence of applications and projections (no abstractions or pairs are allowed). We write $\mathcal{S}(\mathcal{E}[\ ])$ to mean that $\mathcal{S}$ holds for each application argument in $\mathcal{E}[\ ]$, *e.g.*, if $\mathcal{E}[\ ] \equiv (\pi_1(\cdot\ N_1))N_2$, then $\mathcal{S}(\mathcal{E}[\,])$ is short for $S^{\sigma_1}(N_1) \wedge S^{\sigma_2}(N_2)$. Then a property $\mathcal{S}$ is type-closed for PCF if

- $\mathcal{S}(\mathcal{E}[\ ])$ implies $S^\rho(\mathcal{E}[X])$, where X is any variable or constant of appropriate type
- if $S^\tau(Mx)$ for any variable x of type σ, then $S^{\sigma\to\tau}(M)$
- if $S^\sigma(\pi_1 M)$ and $S^\tau(\pi_2 M)$, then $S^{\sigma\times\tau}(M)$
- if $S^\rho(\mathcal{E}[[N/x]M])$ and $S^\sigma(N)$, then $S^\rho(\mathcal{E}[(\lambda x\colon\sigma.\,M)N])$
- if $S^\rho(\mathcal{E}[M])$ and $S^\sigma(N)$, then $S^\rho(\mathcal{E}[\pi_1\langle M,N\rangle])$ and $S^\rho(\mathcal{E}[\pi_2\langle N,M\rangle])$.

Lemma 5.1 *If a property $\mathcal{S}$ of terms is type-closed, then $S^\sigma(M)$ holds for every well-formed term M of type σ.*

Proof. We may construct another property $\mathcal{P}$ from $\mathcal{S}$ such that $\mathcal{P}$ implies $\mathcal{S}$ and $\mathcal{P}$ is an admissible logical relation; the type-closed conditions on $\mathcal{S}$ are precisely those needed to show this. Then by the Basic Lemma for logical relations, $P^\sigma(M)$, and hence $S^\sigma(M)$, holds for every M; see [Mit90] for details. ∎

Theorem 5.2 *The reduction system $\mathit{pcf}^{\mathcal{N}}_0$ is confluent and strongly normalizing.*

Proof. We need to show that the properties $\mathcal{CR}$ and $\mathcal{SN}$, which assert that reduction from a term is respectively confluent (Church-Rosser) and strongly normalizing, are type-closed. Most of the conditions are easy to establish. The hardest part is to show that each property satisfies the first condition when X is a labelled fixed-point constant; in each case an induction on the label is required. ∎

Next we prove that the system $\mathit{pcf}^{\mathcal{N}}$, obtained by adding a set $\mathcal{R}$ of confluent left-linear algebraic rules to $\mathit{pcf}^{\mathcal{N}}_0$, is also confluent. In [BT88], it is shown that the pure typed lambda calculus (with β-reduction only) remains confluent when any confluent (not necessarily linear) $\mathcal{R}$ is added. While the original proof of one lemma has a subtle but reparable bug when non-linear rules are considered [BTG89], we observe that the proof is correct if all the rules in $\mathcal{R}$ are linear. Therefore, our confluence proof for $\mathit{pcf}^{\mathcal{N}}$ will be based on the original development of [BT88]. We then use the linearity of $\mathcal{R}$ again to prove that *pcf* itself is confluent; for this argument, linearity is essential.

The following lemma allows us to consider algebraic reductions only on terms in $\mathit{pcf}^{\mathcal{N}}_0$ normal form. By Theorem 5.2, every term M in the labelled language has a unique $\mathit{pcf}^{\mathcal{N}}_0$ normal form, which we refer to as $\mathit{pcf}^{\mathcal{N}}_0(M)$.

Lemma 5.3

1. *If* $M \xrightarrow{r} N$ *for* $r \in \mathcal{R}$, *then* $pcf_0^{\mathcal{N}}(M) \xrightarrow{r}\!\!\!\!\rightarrow pcf_0^{\mathcal{N}}(N)$;

2. *If* $M \xrightarrow{pcf^{\mathcal{N}}}\!\!\!\!\rightarrow N$, *then* $pcf_0^{\mathcal{N}}(M) \xrightarrow{\mathcal{R}}\!\!\!\!\rightarrow pcf_0^{\mathcal{N}}(N)$.

Lemma 5.4 *$\mathcal{R}$-reduction is confluent on labelled terms in $pcf_0^{\mathcal{N}}$ normal form.*

These two lemmas now let us prove that $pcf^{\mathcal{N}}$ is confluent. As a corollary, we use the linearity of the rules in $\mathcal{R}$ to show that *pcf* itself is confluent.

Theorem 5.5 *$pcf^{\mathcal{N}}$-reduction is confluent on all labelled PCF terms.*

Proof. For any terms M, N, and P, if $N \xleftarrow{pcf^{\mathcal{N}}}\!\!\!\!\leftarrow M \xrightarrow{pcf^{\mathcal{N}}}\!\!\!\!\rightarrow P$, then by Lemma 5.3 we know that

$$pcf_0^{\mathcal{N}}(N) \xleftarrow{\mathcal{R}}\!\!\!\!\leftarrow pcf_0^{\mathcal{N}}(M) \xrightarrow{\mathcal{R}}\!\!\!\!\rightarrow pcf_0^{\mathcal{N}}(P).$$

Lemma 5.4 then asserts that there is a Q such that

$$pcf_0^{\mathcal{N}}(N) \xrightarrow{\mathcal{R}}\!\!\!\!\rightarrow Q \xleftarrow{\mathcal{R}}\!\!\!\!\leftarrow pcf_0^{\mathcal{N}}(P).$$

Since $N \xrightarrow{pcf^{\mathcal{N}}}\!\!\!\!\rightarrow pcf_0^{\mathcal{N}}(N)$ and $P \xrightarrow{pcf^{\mathcal{N}}}\!\!\!\!\rightarrow pcf_0^{\mathcal{N}}(P)$, we thus have that $N \xrightarrow{pcf^{\mathcal{N}}}\!\!\!\!\rightarrow Q \xleftarrow{pcf^{\mathcal{N}}}\!\!\!\!\leftarrow P$. ■

Corollary 5.6 *pcf-reduction is confluent on all PCF terms.*

Proof. For any terms M, N, and P, if $N \xleftarrow{pcf}\!\!\!\!\leftarrow M \xrightarrow{pcf}\!\!\!\!\rightarrow P$, then there are corresponding labelled terms M^*, N^*, and P^* such that $N^* \xleftarrow{pcf}\!\!\!\!\leftarrow M^* \xrightarrow{pcf}\!\!\!\!\rightarrow P^*$. To see this, let m be the length of the longer of the two *pcf*-reduction sequences from M, and form M^* by labelling each fix_σ in M with m; then mimic the two reductions from M by replacing (fix) steps with (fix^n) steps, for appropriate choices of n. Now, since $pcf^{\mathcal{N}}$ is confluent, N^* and P^* must have a common reduct Q^*; erasing the labels in Q^* gives a term Q such that $N \xrightarrow{pcf}\!\!\!\!\rightarrow Q \xleftarrow{pcf}\!\!\!\!\leftarrow P$, hence *pcf* is confluent. ■

The proof of this corollary uses linearity in asserting that a single labelled term M^* will permit both reductions from M to be mimicked. This property fails in the presence of non-linear rules because a reduction step may then require that two subterms have identical labels. An example of the problem, using (sp), is the term $M^* \equiv \langle \pi_1 fix_\sigma^m\ I, \pi_2 fix_\sigma^n\ I\rangle$, where $I \equiv (\lambda x{:}\,\sigma.\,x)$, and consider the following two reduction sequences:

$$M \quad \xrightarrow{sp} \quad fix_\sigma^m\ I$$

and

$$\begin{array}{lll} M & \xrightarrow{fix^n} & \langle \pi_1 fix_\sigma^m\ I, \pi_2(\lambda f{:}\,\sigma\to\sigma.\, f(fix_\sigma^{n-1}\ f))I\rangle \\ & \xrightarrow{\beta}\!\!\!\!\rightarrow & \langle \pi_1 fix_\sigma^m\ I, \pi_2 fix_\sigma^{n-1}\ I\rangle \\ & \xrightarrow{sp} & fix_\sigma^m\ I. \end{array}$$

In the first case, the (sp) step requires that $m = n$, while in the second case we must have $m = n-1$, hence there is no such M^*.

6 Postponement

In this section, we analyze the reduction system $pcf_{\eta,sp}$. Our main technical tool is a postponement theorem for (η) and (sp), which is proved following the pattern for postponement of (η) in the untyped lambda calculus [Bar84]. While (sp) postponement fails for untyped lambda terms (as noted in the introduction), we are able to prove postponement for typed PCF terms. The main "trick" in the proof is to find the correct analogy between *sp*-reduction and η-reduction. The postponement theorem will be used in the next section to show that *pcf* is sufficient to compute the $pcf_{\eta,sp}$ normal form of any *program*, *i.e.*, any closed term of base type (in fact, programs may have free variables as long as they are of algebraic type, since they act like algebraic constants with no associated reductions).

To show postponement of (η) and (sp) for $pcf_{\eta,sp}$, we will use the related system pcf^{lab}, in which η- and *sp*-redexes are represented as labels. We add term formation rules to allow M^η whenever M is a term of function type, and M^{sp} whenever M is of product type; substitution is then extended in the natural way. We also define two functions from labelled to unlabelled terms: $|\cdot|$ and φ. The action of $|\cdot|$ is simply to erase all the labels, while φ replaces labelled subterms with the corresponding redexes: $\varphi(M^\eta) = (\lambda x{:}\,\sigma.\,\varphi(M)x)$ if $M{:}\,\sigma\to\tau$, and $\varphi(M^{sp}) = \langle \pi_1\varphi(M), \pi_2\varphi(M)\rangle$. Finally, the reduction system pcf^{lab} is defined by lifting *pcf* to labelled terms and adding the following contractions:

$$\begin{array}{lll} (act\ \eta) & M^\eta N \longrightarrow MN & \\ (act\ sp) & \pi_i M^{sp} \longrightarrow \pi_i M, & \text{for } i = 1,2 \\ (int\ \eta) & (\lambda x{:}\,\sigma.\,M)^\eta \longrightarrow (\lambda x{:}\,\sigma.\,M) & \\ (int\ sp) & \langle M,N\rangle^{sp} \longrightarrow \langle M,N\rangle. & \end{array}$$

The effect of the first two rules is to simulate the reduction of "active" η- and *sp*-redexes, *i.e.*, those that are also top-level constituents of β- or π-redexes. The other two rules mimic the situation where an η- or *sp*-redex is reduced internally by (β) or (π), *e.g.*,

$$\lambda x{:}\,\sigma.\,(\lambda x{:}\,\sigma.\,M)x \xrightarrow{\beta} \lambda x{:}\,\sigma.\,M;$$

these two rules are not really necessary for the postponement proof, but they will be needed in the next section and there is no harm in adding them here.

We will now prove a series of lemmas relating pcf^{lab} to the unlabelled systems. The first three are easy inductions, either on the length of a reduction or on the structure of a term.

Lemma 6.1 *If* $P \xrightarrow{pcf^{lab}}\!\!\!\!\rightarrow P'$, *then* $\varphi(P) \xrightarrow{pcf}\!\!\!\!\rightarrow \varphi(P')$.

Lemma 6.2 $\varphi(P') \xrightarrow{\eta,sp}\!\!\!\!\rightarrow |P'|$.

Lemma 6.3 *If* $|P| \xrightarrow{pcf}\!\!\!\!\rightarrow N$, *then there exists a term P' such that* $P \xrightarrow{pcf^{lab}}\!\!\!\!\rightarrow P'$ *and* $|P'| \equiv N$.

Lemma 6.4 *If* $M \xrightarrow{\eta,sp}\!\!\!\!\rightarrow M' \xrightarrow{pcf}\!\!\!\!\rightarrow N$, *then there exists a term Q such that* $M \xrightarrow{pcf}\!\!\!\!\rightarrow Q \xrightarrow{\eta,sp}\!\!\!\!\rightarrow N$.

Proof. Either $M \equiv \mathcal{C}[(\lambda x{:}\sigma.\, Lx)]$ and $M' \equiv \mathcal{C}[L]$, or $M \equiv \mathcal{C}[\langle \pi_1 L, \pi_2 L\rangle]$ and $M' \equiv \mathcal{C}[L]$, for some context $\mathcal{C}[\ ]$ and term L. In the first case, take $P \equiv \mathcal{C}[L^{\eta}]$; in the second case, take $P \equiv \mathcal{C}[L^{sp}]$. Then $M \equiv \varphi(P)$ and $M' \equiv |P|$. By Lemma 6.3, there is a term P' such that $P \xrightarrow{pcf^{lab}}\!\!\!\rightarrow P'$ and $|P'| \equiv N$. Then by Lemma 6.1 we find that $M \equiv \varphi(P) \xrightarrow{pcf}\!\!\!\rightarrow \varphi(P')$. Since by Lemma 6.2, $\varphi(P') \xrightarrow{\eta,sp}\!\!\!\rightarrow |P'| \equiv N$, we may take $Q \equiv \varphi(P')$. ■

From this lemma we may now prove the postponement of (η) and (sp) in $pcf_{\eta,sp}$.

Theorem 6.5 *If $L \xrightarrow{pcf_{\eta,sp}}\!\!\!\rightarrow N$, then there is a term M such that $L \xrightarrow{pcf}\!\!\!\rightarrow M$ and $M \xrightarrow{\eta,sp}\!\!\!\rightarrow N$.*

Proof. Given a reduction sequence from L to N, we may use the previous lemma to push all the (η) and (sp) steps to the end. ■

In the special case that N is a program in $pcf_{\eta,sp}$ normal form, which we refer to as a *result*, we find that the reductions (η) and (sp) are unnecessary:

Corollary 6.6 *If $L \xrightarrow{pcf_{\eta,sp}}\!\!\!\rightarrow R$ and R is a result, then $L \xrightarrow{pcf}\!\!\!\rightarrow R$.*

Proof. By Theorem 6.5 there is an M such that $L \xrightarrow{pcf}\!\!\!\rightarrow M \xrightarrow{\eta,sp}\!\!\!\rightarrow R$. If the last step in this reduction is $P \xrightarrow{\eta,sp} R$, with Q the redex in P and Q' its contractum in R, then assume first that Q is passive. Thus Q' is of non-base type and it cannot be the head of an application or the subject of a projection; if it is in the body of an abstraction or a pair, then there is a larger subterm of R that is of non-base type. Consider the largest such enclosing subterm (possibly Q' itself). Since R is a normal form of base type, this subterm must be in the argument of an application. But algebraic constants only take base type arguments, and the presence of a *fix* contradicts R being in normal form, so the head of the application must be a variable. This is also impossible, because there are no free variables in R (except perhaps of algebraic type), and a surrounding abstraction would give yet a larger subterm of non-base type. Hence Q must be active and we may use Lemma 6.4 to push this step in front of the (η) and (sp) steps; this may be repeated to eliminate all of the non-*pcf* reduction steps. ■

7 The Result Property

Although $pcf_{\eta,sp}$ is not confluent, there are other properties of reduction that might be considered as plausible substitutes. In this section, we will show that $pcf_{\eta,sp}$ has a weaker *result property.* To put this property in perspective, we might consider three general properties of reduction, in order of decreasing strength. These are confluence, the so-called *normal form property* of [Kd89], which says that if $\Gamma \triangleright M \leftrightarrow N : \sigma$ and N is a normal form, then $M \twoheadrightarrow N$, and the uniqueness of normal forms. It is not hard to see that confluence implies the normal form property, and the normal form property implies that each term has at most one normal form. While Klop and de Vrijer have shown that the normal form property fails in untyped lambda calculus with surjective pairing, we will show that typed PCF has a modified version of this property, which we call the *result property.*

The result property, proved in Theorem 7.4 below, states that if M is $pcf_{\eta,sp}$ convertible to a result N (see Section 6), then M is $pcf_{\eta,sp}$ reducible to N. Since conversion is equivalent to provable equality in the full PCF proof system, it follows by Corollary 6.6 that if a term M is provably equal to a result, then $M \xrightarrow{pcf}\!\!\!\rightarrow N$. Furthermore, it follows from the result property and postponement that (η) and (sp) are sound equational rules for reasoning about *pcf* observational congruence. Thus, when combined with other properties of $pcf_{\eta,sp}$ and *pcf* reduction, the result property not only gives a certain coherence to $pcf_{\eta,sp}$ reduction, but relates provable equality using (η) and (sp) to program execution without these "non-computational" rules.

We will prove the result property using a series of lemmas similar to those used to show postponement in Section 6. The same labelled system pcf^{lab} will be used, and here the internal rules will be important.

Lemma 7.1 *If $P \xrightarrow{pcf^{lab}}\!\!\!\rightarrow P'$, then $|P| \xrightarrow{pcf}\!\!\!\rightarrow |P'|$.*

Lemma 7.2 *If $\varphi(P) \xrightarrow{pcf}\!\!\!\rightarrow R$ and R is a result, then there exists a term P' such that $P \xrightarrow{pcf^{lab}}\!\!\!\rightarrow P'$ and $\varphi(P') \equiv |P'| \equiv R$.*

Proof. Since there are no η- or sp-redexes in R, there will be no labels left in P'. Therefore, we will have $\varphi(P') \equiv |P'|$, and every descendent of the redex in $\varphi(P)$ corresponding to a label in P must eventually either *pcf*-reduce or be erased. The *pcf*-reductions will correspond to cases where a redex either is active or reduces internally; they may thus be simulated by the labelled reduction. One complication comes in simulating the reduction step $\langle \pi_1 M, \pi_2 M\rangle \rightarrow \langle \pi_1 M', \pi_2 M\rangle$; there is no reduction on the corresponding labelled term M^{sp} that matches this, but since *pcf* is confluent and R is in normal form, we know that the two components of the pair will eventually reduce to the same normal form (or the entire pair will be erased), so we may arbitrarily choose to follow reductions to the first component.

The other situation where we must be careful is when there are rules in $\mathcal{R}$ of the form $(fM_1 \ldots M_{k-1}x) \rightarrow N$; *i.e.*, rules that accept an arbitrary rightmost argument. If $P \equiv (fM_1 \ldots M_{k-1})^{\eta}$, then $\varphi(P) \rightarrow (\lambda x{:}b.\, N)$, but P does not pcf^{lab}-reduce. This case is ruled out by the condition that R be a result, however, because reasoning similar to that in the proof of Corollary 6.6 shows that algebraic constants in R must always be at the head of subterms of base type. Hence we may simulate the *pcf* reduction in the labelled system. ■

Lemma 7.3 *If $M \xrightarrow{\eta,sp} M'$ and $M \xrightarrow{pcf}\!\!\!\rightarrow R$, where R is a result, then $M' \xrightarrow{pcf}\!\!\!\rightarrow R$.*

Proof. Take P as in Lemma 6.4, so that $M \equiv \varphi(P)$ and $M' \equiv |P|$. By Lemma 7.2, there is a term P' such that $P \xrightarrow{pcf^{lab}}\!\!\!\rightarrow P'$ and $\varphi(P') \equiv |P'| \equiv R$. By Lemma 7.1 we find that $M' \equiv |P| \xrightarrow{pcf}\!\!\!\rightarrow |P'|$, so we are done. ■

Now we may prove the result property for *pcf*.

Theorem 7.4 *If $\Gamma \triangleright N \overset{pcf_{\eta,sp}}{\longleftrightarrow} R : \sigma$ and R is a result, then $N \xrightarrow{pcf}\!\!\!\rightarrow R$.*

Proof. We will prove that if $L \overset{pcf_{\eta,sp}}{\twoheadrightarrow} N$ and $L \overset{pcf_{\eta,sp}}{\twoheadrightarrow} R$, where R is a result, then $N \overset{pcf}{\twoheadrightarrow} R$; this is easily seen to be equivalent. By the postponement theorem there is a term M such that $L \overset{pcf}{\twoheadrightarrow} M \overset{\eta,sp}{\twoheadrightarrow} N$; by the corollary to postponement, $L \overset{pcf}{\twoheadrightarrow} R$. Now, since *pcf* is confluent and R is in normal form, we know that $M \overset{pcf}{\twoheadrightarrow} R$. Using Lemma 7.3 once for each reduction step from M to N, we find that $N \overset{pcf}{\twoheadrightarrow} R$. ■

As a corollary to this theorem, we will show that the equational forms of (η) and (sp) are sound for reasoning about observational congruence. First we define a *program context* for a given type σ and variable assignment Γ to be a context $\mathcal{P}[\ ]$ such that $\mathcal{P}[M]$ is a program whenever $\Gamma \triangleright M\colon \sigma$. We say that two terms M and N of the same type σ and variable assignment Γ are *observationally congruent*, written $\Gamma \triangleright M \simeq N\colon \sigma$, if for every program context $\mathcal{P}[\ ]$ for σ and Γ, $\mathcal{P}[M]$ *pcf*-reduces to a result R iff $\mathcal{P}[N] \overset{pcf}{\twoheadrightarrow} R$. In other words, M and N are completely interchangeable when computing the result of a program.

Corollary 7.5 *The equational axioms* $(\eta)_{eq}$ *and* $(sp)_{eq}$ *are sound for* $\simeq$.

Proof. To show that (η) is sound, we need to show that for any term M of type $\sigma \rightarrow \tau$ over a variable assignment Γ, with x a variable not free in M, we have $\Gamma \triangleright (\lambda x\colon \sigma.\, Mx) \simeq M\colon \sigma \rightarrow \tau$. If $\mathcal{P}[\ ]$ is a program context for $\sigma \rightarrow \tau$ and Γ, then obviously $\Gamma \triangleright \mathcal{P}[(\lambda x\colon \sigma.\, Mx)] \overset{pcf_{\eta,sp}}{\longleftrightarrow} \mathcal{P}[M] : \sigma$; if $\mathcal{P}[(\lambda x\colon \sigma.\, Mx)]$ *pcf*-reduces to a result R, then we also have that $\Gamma \triangleright \mathcal{P}[M] \overset{pcf_{\eta,sp}}{\longleftrightarrow} R : \sigma$, and by the previous theorem we find that $\mathcal{P}[M] \overset{pcf}{\twoheadrightarrow} R$. The same argument in reverse shows that if $\mathcal{P}[M] \overset{pcf}{\twoheadrightarrow} R$, then $\mathcal{P}[(\lambda x\colon \sigma.\, Mx)] \overset{pcf}{\twoheadrightarrow} R$. Similar reasoning shows that (sp) is sound. ■

8 Conclusion

Since the full reduction system $pcf_{\eta,sp}$ with (η) and (sp) is not confluent, we consider the more limited system *pcf* more appropriate for PCF execution. The system *pcf* includes fixed-point rules (*fix*), lambda calculus rules (β) for function calls and (π) for pairs, and any set $\mathcal{R}$ of left-linear, confluent algebraic rules. We have shown that *pcf* reduction is confluent and demonstrated several connections between *pcf* and $pcf_{\eta,sp}$. The first is that (η) and (sp) rules may be postponed in $pcf_{\eta,sp}$ reduction, and so whenever a closed term of base type $pcf_{\eta,sp}$ reduces to normal form, this reduction may be accomplished without (η) or (sp). Thus, if we consider programs to be closed terms of base type (or any product of such types), *pcf* is equivalent for the purpose of program execution to the apparently stronger but non-confluent $pcf_{\eta,sp}$. We also prove a *result property* of $pcf_{\eta,sp}$ in Section 7, from which it follows that *(i)* whenever a term is provably equal (in the full system) to a result, the term reduces to this result by *pcf* reduction, and *(ii)* all equational rules, including $(\eta)_{eq}$ and $(sp)_{eq}$, are sound for *pcf* observational congruence. In summary, these technical results suggest that while the full equational proof system is a natural "axiomatic semantics" for PCF, a more limited reduction system has more desirable technical properties and seems suitable as a corresponding operational semantics.

One open problem is to extend our confluence theorem to include reduction rules for non-algebraic terms. For example, we might like to give reduction rules for the evaluation of some higher-order function, and be sure that the resulting extension of *pcf* is confluent. Presumably our current proof already applies to some cases of this form, but we have not yet done a careful analysis. Another problem is to show that some deterministic strategy is sufficient for PCF evaluation. "Left-most" reduction, which is adequate for lambda calculus, does not make sense for algebraic terms. The reason is that there is no essential difference between *cons*(*atom*, *list*) and *cons*(*list*, *atom*), for example. However, we conjecture that a suitable adaptation of "left-most" to algebraic terms will prove satisfactory, at least for the simple case of non-overlapping algebraic rules. This would give a correspondence between nondeterministic reduction and the deterministic evaluation strategy used in the semantic study of [Plo77], providing a full correspondence between axiomatic, operational and denotational semantics of PCF.

Another possible extension is to exploit the analogies between function and product types, as for example in the postponement proof, to add other type constructors such as sums (which would introduce case statements and, as a special case, conditionals) or streams. In categorical terms, each of these type constructors corresponds to some adjunction, and the extensional rules may be derived from one direction of the conditions for being an adjunction. This is most easily expressed in the context of PCF by observing that (β) and (π) give the result of composing the type elimination rules ($\rightarrow$ Elim) and ($\times$ Elim) with the respective introduction rules ($\rightarrow$ Intro) and ($\times$ Intro). The extensional rules do the converse, first eliminating and then re-introducing a type constructor; as such, they produce no observable effect. We expect there to be similar results about the adequacy of an operational semantics excluding all such extensional rules for PCF when augmented with other suitable type constructors.

Acknowledgement: We thank P.-L. Curien and R. Statman for helpful discussions.

References

[Bar84] H.P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* North Holland, 1984.

[BT88] V. Breazu-Tannen. Combining algebra and higher-order types. In *Third IEEE Symp. Logic in Computer Science*, pages 82–90, 1988.

[BTG89] V. Breazu-Tannen and J.H. Gallier. Polymorphic rewriting conserves algebraic strong normalization and confluence. In *16th Int'l Colloq. on Automata, Languages and Programming*, pages 137–159. Springer Verlag, LNCS 37, 1989.

[Chu41] A. Church. *The Calculi of Lambda Conversion.* Princeton Univ. Press, 1941. Reprinted 1963 by University Microfilms Inc., Ann Arbor, MI.

[Hen80] P. Henderson. *Functional Programming.* Prentice–Hall, 1980.

[HWWW85] J.Y. Halpern, J.H. Williams, E.L. Wimmers, and T.C. Winkler. Denotational semantics and rewrite rules for FP. In *Proc. 12-th*

ACM Symp. on Principles of Programming Languages, pages 108–120, January 1985.

[Kd89] J.W. Klop and R.C. de Vrijer. Unique normal forms for lambda calculus with surjective pairing. *Information and Computation*, 80(2):97–113, 1989.

[Klo80] J.W. Klop. *Combinatory Reduction Systems*. PhD thesis, University of Utrecht, 1980. Published as Mathematical Center Tract 129.

[Mit90] J.C. Mitchell. Type systems for programming languages. In J. van Leeuwen *et al.*, editor, *Handbook of Theoretical Computer Science*. North-Holland, 1990. (To appear.).

[Nes89] D. Nesmith. The Church-Rosser property in higher-order rewrite systems. Manuscript, 1989.

[Pey87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice–Hall, 1987.

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

[Pot81] G. Pottinger. The Church–Rosser theorem for typed λ-calculus with surjective pairing. *Notre Dame Journal of Formal Logic*, 22(3):264–268, 1981.

[Sco69] D.S. Scott. A type–theoretic alternative to CUCH, ISWIM, OWHY. Manuscript, 1969.

[Tur85] D.A. Turner. Miranda: a non-strict functional language with polymorphic types. In *IFIP Int'l Conf. on Functional Programming and Computer Architecture, Nancy, Lecture Notes in Computer Science 201*, New York, 1985. Springer-Verlag.