Adding Input-Output to PCF
(Technical Summary)

Jerzy Tiuryn*
Institute of Informatics
Warsaw University
Banacha 2
02-097 Warsaw, POLAND
tiuryn@mimuw.edu.pl

Mitchell Wand†
College of Computer Science
Northeastern University
360 Huntington Avenue, 161CN
Boston, MA 02115, USA
wand@ccs.neu.edu
http://www.ccs.neu.edu/home/wand

Topics: formal semantics, lambda calculus, PCF, operational semantics
Approximate Word Count: 3544

July 13, 1995

Abstract

We extend Plotkin's PCF by adding monadic-style input-output operations. This extension is intended to allow the classification of possibly infinite input-output behaviors, such as those required for servers or distributed

*This work is partly supported by NSF Grants CCR-9417382, CCR-9304144, and by Polish KBN Grant 2 P301 031 06.

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

systems. We define a notion of applicative approximation and show that it coincides with operational equivalence for these new behaviors. Last, we show that the new notion of operational equivalence is a conservative extension of the usual one.

1 Introduction

In the study of languages such as PCF [12], the standard paradigm for studying equivalence of terms begins by setting up some relatively weak notion of behavior (or immediate observation). For PCF, this is generally taken to be convergence to an integer constant. Two terms are said to be *operationally equivalent* iff they have the same behavior in any context. This is a useful notion of equivalence, as it asserts that the two terms can be substituted for each other at will.

A contrasting notion of equivalence is suggested by Abramsky's *applicative bisimulation* [1]. Rather than quantifying over contexts, this approach treats terms as equivalent if they give equivalent results when applied to the same arguments. In an untyped system, this leads naturally to bisimulation; in a typed system, it leads to an induction on types.[1] Work on applicative bisimulation includes [3, 5, 7, 6, 13]. An applicative bisimulation can be considered as a much richer notion of immediate behavior. The goal is then to find an applicative bisimulation which coincides with operational equivalence.

The ordinary notion of behavior as convergence to an integer constant is generally satisfactory and has led to a rich theory. This notion of behavior seems less appropriate, however, if we wish to consider programs with infinite behaviors, such as servers or other components of operating systems or distributed systems. For such systems, it seems more appropriate to adopt a notion of behavior that classifies programs by their possible interactions with other programs.

In this paper, we take a first step in this direction by proposing a notion of behavior for PCF based on monadic input-output, similar to that used in Haskell [11] or by Gordon's CPS input-output system [2]. Our notion of behavior is based on simulation[2]. This is done in Section 2 by adding a

[1] though *not* to the logical relations paradigm; see Definition 3.1.

[2] Because the systems we will consider are deterministic, it is sufficient (and traditional) to consider only simulation (or approximation), rather than bisimulation (or equivalence); see Proposition 2.3.

new type pr of processes and appropriate constants. We give some examples of programs that are equivalent and inequivalent under this new notion of behavior. We then define operational equivalence or operational approximation (the corresponding one-sided notion) by quantifying over contexts in the usual way.

In Section 3, we extend our notion of behavior to a notion of applicative approximation (the one-sided version of applicative bisimulation), defined by an induction on types. As a step toward the main theorem, we show that applicative approximation is preserved by the typed term-construction rules of PCF. In Section 4, we use this result to show that applicative approximation and operational approximation coincide. This is the main theorem of the paper.

Last, in Section 5 we consider the relation between our new notion of operational equivalence and the traditional notion of operational equivalence based on convergence to an integer. We show that the new notion is a conservative extension of the old one: that the additional contexts allowed in our system do not distinguish any pure PCF terms that were not distinguished by integer contexts. This holds even if the terms are allowed to have subterms with types involving pr.

We conclude with discussion of related work and some conclusions and open problems.

2 The Basic System

We begin with Plotkin's PCF [12]. We add to the system a new type pr of processes, and new constants

$$\begin{aligned}&\mathbf{nil} : pr\\&\mathbf{write} : int \to pr \to pr\\&\mathbf{read} : (int \to pr) \to pr\end{aligned}$$

We follow [4] in presenting the integers using constants $\mathbf{0}$ and $\mathbf{succ}$. We write $\lceil n \rceil$ for $\mathbf{succ}^{(n)}(\mathbf{0})$.

The typing rules are standard and are omitted here. We name the set of terms of type σ (with the free variables typed by assumptions A) $\Lambda_{A,\sigma}$; the set of closed terms of type σ will be denoted Λ_σ.

We give the system a small-step call-by-name operational semantics, presented using reduction contexts in Figure 1. The only difference between

Reduction contexts:

$$R ::= [] \mid RM \mid \mathbf{pred}\,R \mid \mathbf{succ}\,R \mid \mathbf{zero?}\,R \mid \mathbf{if}\,RMN \mid \mathbf{write}\,R$$

Reduction Rules:

$$\begin{array}{l}
R[\mathbf{pred}\,\mathbf{0}] \to R[\mathbf{0}] \\
R[\mathbf{pred}(\mathbf{succ}\,V)] \to R[V] \\
R[\mathbf{zero?}\,\mathbf{0}] \to R[\mathbf{true}] \\
R[\mathbf{zero?}(\mathbf{succ}\,V)] \to R[\mathbf{false}] \\
R[\mathbf{if}\,\mathbf{true}\,MN] \to R[M] \\
R[\mathbf{if}\,\mathbf{false}\,MN] \to R[N] \\
R[(\lambda x.M)N] \to R[M[N/x]] \\
R[\mathbf{Y}M] \to R[M(\mathbf{Y}M)]
\end{array}$$

Figure 1: Reduction Rules for PCF

this operational semantics and the standard one is the addition of the new reduction context $\mathbf{write}\,R$. We write $\stackrel{*}{\to}$ for the reflexive transitive closure of $\to$. Note that the reduction sticks on terms of the form $R[\mathbf{read}\,M]$ or $R[\mathbf{write}\,V N]$.

We turn the closed terms of type pr into a labelled transition system with actions $\{(a, n) \mid a \in \{!, ?\}, n \in N\}$ by the rules:

$$\frac{M \stackrel{*}{\to} \mathbf{write}\lceil n \rceil M'}{M \stackrel{!n}{\to} M'}$$

$$\frac{M \stackrel{*}{\to} \mathbf{read}\,M'}{M \stackrel{?n}{\to} (M'\ \lceil n \rceil)}$$

We use unlabelled arrows ($\to$, $\stackrel{*}{\to}$) for computation steps and labelled arrows ($\stackrel{!n}{\to}$, $\stackrel{?n}{\to}$) for the action steps of the labelled transition system. We hope this will cause no confusion.

The input-output behavior of these terms is expressed through their behavior in the labelled transition system. **write** writes an integer, **read** reads an integer and passes it to its argument, and **nil** deadlocks– it has no actions. Note that these terms operate only at the top level, so they are effectively in monadic or continuation-passing style.

For example, the term

$$\mathbf{Y}(\lambda p : pr \,.\, \mathbf{read}\,(\lambda x : int \,.\, \mathbf{write}\, x\, p)) \tag{1}$$

echoes its input.

For a more complex example, assume we have a simulation of unbounded queues of integers (coded as integers), with q_0 a coding of the empty queue, $\mathrm{enq}\, q\, x$ returning the queue obtained by adding x to tail of q, and $\mathrm{deq}\, q f$ passing to f the first element on the queue q and the queue obtained by removing that first element. Such terms are of course easy to code. Then here is a term that echoes its input at half-speed by reading two numbers, depositing them in an unbounded queue, and printing one number from the queue:

$$\begin{aligned}&\mathbf{Y}(\lambda f : int \to pr \,.\, \lambda q : int \,.\\ &\quad \mathbf{read}\,(\lambda x : int \,.\, \mathbf{read}\,(\lambda y : int \,.\\ &\quad\quad \mathrm{deq}\,(\mathrm{enq}(\mathrm{enq}\, q\, x) y)\\ &\quad\quad\quad (\lambda v : int \,.\, \lambda q' : int \,.\, \mathbf{write}\, v\,(f q'))))))\, q_0\end{aligned} \tag{2}$$

This calculus inherits the usual notions of simulation and bisimulation from its definition as a labelled transition system [10]. It is straightforward to check that the terms (1) and (2) above are not bisimilar, whereas a term that differed from (2) only by altering the coding of queues would be bisimilar to it.

By specializing the general definitions of simulation and bisimulation to our system, we can characterize these notions as greatest fixpoints of monotone operators on relations:

Definition 2.1 *If $R \subseteq \Lambda_{pr} \times \Lambda_{pr}$ is any relation on closed terms of type pr, the relation $[R] \subseteq \Lambda_{pr} \times \Lambda_{pr}$ is defined by $M\ [R]\ N$ iff the following properties hold:*

1. *if $M \xrightarrow{*} \mathbf{nil}$, then $N \xrightarrow{*} \mathbf{nil}$.*
2. *if $M \xrightarrow{*} \mathbf{write}\lceil n \rceil M'$, then there exists an N' such that $N \xrightarrow{*} \mathbf{write}\lceil n \rceil N'$ and $M'\ R\ N'$.*
3. *if $M \xrightarrow{*} \mathbf{read}\, M'$, then there exists an N' such $N \xrightarrow{*} \mathbf{read}\, N'$ and for all $n \in N$, $(M' \lceil n \rceil)\ R\ (N' \lceil n \rceil)$.*

Definition 2.2 Simulation $\leq_{pr}$ *is defined to be the greatest fixpoint of* $[-]$. Bisimulation $\cong$ *is defined to be the greatest fixed point of* $[-] \cap ([-])^{op}$.

Proposition 2.3 $\cong = \leq_{pr} \cap (\leq_{pr})^{op}$.

Proof: A routine argument, using the determinacy of PCF evaluation.
∎

Our notion of behavior immediately gives us a notion of operational approximation:

Definition 2.4 *If* $A \vdash M, N : \sigma$, *then we say* M operationally approximates N ($M \sqsubseteq_{A,\tau} N$) *iff for every context* C *of type* pr, *with a hole of type* (A, τ),

$$C[M] \leq_{pr} C[N]$$

3 Applicative Approximation

Our next task is to extend $\leq_{pr}$ to an applicative approximation. We do this by induction on types, using $\leq_{pr}$ and the usual notions on integers and booleans as the basis.

Definition 3.1 *For each type* σ, *we define* $\leq_\sigma \subseteq \Lambda_\sigma \times \Lambda_\sigma$ *as follows:*

1. *If* $M, N : int$, *then* $M \leq_{int} N$ *iff if* $M \xrightarrow{*} \lceil n \rceil$, *then* $N \xrightarrow{*} \lceil n \rceil$.
2. *If* $M, N : bool$, *then* $M \leq_{bool} N$ *iff if* $M \xrightarrow{*} c$, *then* $N \xrightarrow{*} c$ *for* $c \in \{\mathbf{true}, \mathbf{false}\}$.
3. *If* $M, N : pr$, *then* $M \leq_{pr} N$ *as in Definition 2.2*
4. *If* $M, N : \sigma \to \tau$, *then* $M \leq_{\sigma\to\tau} N$ *iff for all* $P \in \Lambda_\sigma$, $(M\ P) \leq_\tau (N\ P)$.

This defines a relation on closed terms of each type. At functional types, the relation is defined as "gives related behavior when applied to the same argument"; the major portion of our proof will be devoted to showing that this implies that related terms have related behavior when applied to related (but not necessarily identical) arguments.

Lemma 3.2 *For any type* τ,

1. *If $M \to M'$, $N \to N'$, and $M' \leq_\tau N'$, then $M \leq_\tau N$.*

2. *$\leq_\tau$ is reflexive and transitive.*

Proof: By induction on τ. ∎

If A is a set of type assumptions, we define define a *closing substitution* μ for A to be a map from $\mathrm{dom}(A)$ to closed terms such that for each $x \in \mathrm{dom}(A)$, $\mu(x)$ is a closed term of type $A(x)$. We write $\mu \models A$. We can then extend applicative approximation to open terms as follows:

Definition 3.3 *If $A \vdash M, N : \sigma$, then we say $M \leq_{A,\sigma} N$ iff $M\mu \leq_\sigma N\mu$ for every substitution μ such that $\mu \models A$.*

The major technical result needed to show that applicative approximation coincides with operational approximation is that applicative approximation be a *precongruence*, that is, a preorder closed under the (typed) term constructors of PCF.[3] More formally:

Definition 3.4 *Let $R = \{R_{A,\tau}\}$, where A ranges over type assumptions and τ ranges over types, be a family of reflexive, transitive relations on $\Lambda_{A,\tau}$. We say R is a* precongruence *iff it satisfies the following conditions:*

1. *$x \; R_{A,\tau} \; x$ whenever $A(x) = \tau$.*

2. *$c \; R_{A,\tau} \; c$ for each constant c of type τ.*

3. *if $M \; R_{A[x:\sigma],\tau} \; M'$, then $(\lambda x : \sigma.M) \; R_{A,(\sigma\to\tau)} \; (\lambda x : \sigma.M')$.*

4. *if $M \; R_{A,(\sigma\to\tau)} \; M'$ and $N \; R_{A,\sigma} \; N'$, then $(MN) \; R_{A,\tau} \; (M'N')$.*

Our goal is to show that $\{\leq_{A,\tau}\}$ is a precongruence. We accomplish this via a series of lemmas, one for each of the rules in the definition of precongruence. The first two are trivial, and the third rule, preservation under abstraction, is straightforward reasoning from the definition.

The hard lemma is showing that $\{\leq_{A,\tau}\}$ is preserved under application. To do this, we introduce an auxiliary notion called *monotonicity*, akin to Plotkin's "computability."

[3] An equivalence relation closed under term constructors is usually called a congruence, so a preorder closed under term constructors is called a precongruence [7].

Definition 3.5 *1. Every closed term of type int or pr is monotone.*

2. $M : \sigma \to \tau$ is monotone iff for all closed and monotone terms $P, Q : \sigma$,

(a) MP and MQ are monotone, and

(b) if $P \leq_\sigma Q$, then $MP \leq_\tau MQ$

3. A substitution is monotone iff for all $x \in \mathrm{dom}(A)$, $\mu(x)$ is monotone.

4. If $A \vdash M : \sigma$, then M is monotone iff for all monotone substitutions μ, $\mu \models A$ implies $M\mu$ is monotone.

Theorem 3.6 *Every term is monotone.*

Proof: By structural induction on terms. All cases are easy except for $\mathbf{Y}$. To establish that $\mathbf{Y}$ is monotone, we first establish that its approximations $\mathbf{Y}^{(n)}$ are monotone. We then use a compactness argument to show that any derivation involving $\mathbf{Y}$ could have been done by a similar derivation in which the $\mathbf{Y}$'s are replaced by some $\mathbf{Y}^{(n)}$'s. The full paper will include the details. ∎

Corollary 3.7 *If $M \leq_{A,\sigma\to\tau} M'$ and $N \leq_{A,\sigma} N'$, then $MN \leq_{A,\tau} (M'N')$.*

Proof: Let $\mu \models A$. Then $M\mu \leq_{\sigma\to\tau} M'\mu$. Therefore $(M\mu\, N\mu) \leq_\tau (M'\mu\, N\mu)$. Since $M'\mu$ is monotone and $N\mu \leq_\sigma N'\mu$, it follows that $(M'\mu\, N\mu) \leq_\tau (M'\mu\, N'\mu)$. Hence, by Lemma 3.2(2), we have $(M\,N)\mu = (M\mu\, N\mu) \leq_\tau (M'\mu\, N\mu) \leq_\tau (M'\mu\, N'\mu) = (M'N')\mu$. ∎

Corollary 3.7, combined with the other three lemmas for the rules of Definition 3.4, yields our first main theorem:

Theorem 3.8 (Precongruence) *$\{\leq_{A,\tau}\}$ is a precongruence.*

4 Equivalence of Operational and Applicative Approximation

Now we can show that applicative and operational approximation coincide. It is easy to show that operational approximation $\{\sqsubseteq_{A,\tau}\}$ is a precongruence. Then we show

Theorem 4.1 *$\{\leq_{A,\tau}\}$ is the largest precongruence R such that $R_{pr} \subseteq \leq_{pr}$.*

Proof: By induction on τ. ∎

This gives us our main theorem:

Theorem 4.2 (Equivalence) *$M \leq_{A,\tau} N$ iff $M \sqsubseteq_{A,\tau} N$.*

Proof: For the forward direction, let C be a context of type pr, with a hole of type (A, τ). If $M \leq_{A,\tau} N$, then $C[M] \leq_{pr} C[N]$, by induction on C and the precongruence of $\{\leq_{A,\tau}\}$. Hence $M \sqsubseteq_{A,t} N$.

The reverse direction follows from Theorem 4.1 and the observation that operational approximation is a precongruence. ∎

5 Conservative Extension

Because we now can classify terms with infinite behavior, we can ask whether these additional contexts distinguish terms that were previously operationally equivalent. The answer to this question is negative. This means that all of our previous reasoning about pure PCF terms is still valid in the new context. This is in contrast, for example, to what happens when one adds stores or control operators, which invalidate previous operational equivalences.

To show this conservative extension result, we model processes in pure PCF. Processes will be modelled as trees, which in turn are modelled as partial functions from tree addresses (finite paths) to labels. Hence we model a process as a partial function

$$p : N^* \rightarrow (N \cup \{\mathbf{nil}, \mathbf{read}\})$$

where N denotes the non-negative integers and N^* denotes the set of all sequences over N. We can think of such a function as describing an infinitely-branching tree with nodes labelled by elements of $(N \cup \{\mathbf{nil}, \mathbf{read}\})$. For example, the process described by

$$\mathbf{write}\lceil 5\rceil(\mathbf{read}(\lambda x : \mathit{int} \,.\, \mathbf{if}(\mathbf{zero?}\, x)(\mathbf{write}\lceil 6\rceil\ \mathbf{nil})\,\mathbf{nil}))$$

is modeled by any function p such that

$$
\begin{aligned}
&p(\epsilon) = 5 \\
&p(0) = \mathbf{read} \\
&p(00) = 6 \\
&p(00m) = \mathbf{nil} \\
&p(0m) = \mathbf{nil} \quad (m \neq 0)
\end{aligned}
$$

To model such trees in PCF, assume we have a surjective coding of N^* in N, so that the usual operations on sequences are definable in PCF, and that we also have a coding of $N \cup \{\mathbf{nil}, \mathbf{read}\}$ in N, where $CODENIL : int$ and $CODEREAD : int$ are PCF terms that emit the codes for **nil** and **read**, and $CODE : int \to int$ is a term that emits the code of its integer input. We can then model **nil**, **read**, and **write** by:

$$
\begin{aligned}
&NIL = \lambda w : int \,.\, \mathbf{if}\,(EMPTY?\, w)\ \ CODENIL\ \ DIVERGE \\
&WRITE = \lambda n : int \,.\, \lambda p : int \to int \,.\, \lambda w : int \,. \\
&\qquad \mathbf{if}\,(EMPTY?\, w)\,(CODE\, n) \\
&\qquad\qquad \mathbf{if}\,(\mathbf{zero}?\,(HEAD\, w))\,(p\,(TAIL\, w))\ \ DIVERGE \\
&READ = \lambda f : int \to (int \to int).\, \lambda w : int \,. \\
&\qquad \mathbf{if}\ (EMPTY?\, w)\ \ CODEREAD\ \ (f(HEAD\, w)(TAIL\, w))
\end{aligned}
$$

For any term M in our enriched PCF, let M^* be the term obtained by replacing all instances of the type pr by $int \to int$, and all instances of **nil**, **write**, and **read** by NIL, $WRITE$, and $READ$ respectively. Then we can show:

Theorem 5.1 (Conservative extension) *For all enriched PCF terms, $M \leq_{A,\tau} N$ iff $M^* \sqsubseteq^{\mathrm{pure}}_{A,\tau} N^*$*

Here $\sqsubseteq^{\mathrm{pure}}_{A,\sigma}$ is the "ordinary" operational approximation, using convergence to an integer as the elementary observable (i.e., $M \sqsubseteq^{\mathrm{pure}}_{A,\sigma} N$ iff for every context C of type int with a hole of type (A, σ), if $C[M]$ converges to an integer constant, then so does $C[N]$).

Proof: The proof proceeds by induction on the type τ. If $\tau = pr$, then the proof proceeds by an analysis of the translated terms, using Milner's context lemma and surjectivity of the coding of N^* in N. If $\tau = int$, then the proof proceeds by a direct consideration of the possible computations. At higher types, one proceeds by induction, using Milner's context lemma.
∎

Corollary 5.2 *For all pure PCF terms, $M \leq_{A,\tau} N$ iff $M \sqsubseteq^{\mathrm{pure}}_{A,\tau} N$*

Proof: Pure terms are unchanged by the $(-)^*$ transform. ∎

Furthermore, if M and N contain no instances of the I/O constants (but may contain subterms with types involving pr), then they are operationally approximate in the new system iff they are approximate in the old system with pr replaced by $int \rightarrow int$.

6 Related Work

The idea of applicative approximation is due to Abramsky [1]. Abramsky pointed out that the existing theory of Böhm trees can be viewed as an applicative bisimulation in which the existence of a head normal form is an observable event (the basic component of behavior), and he formulated a new bisimulation in which the existence of a weak head normal form is the basic observable. He also emphasized the importance of showing that applicative approximation was a precongruence. In all of these systems, showing closure under application is the most difficult step.

Howe [5, 6, 7] proposed a general framework for showing that applicative approximations were precongruences. We have elsewhere used this method to show congruence for an untyped variant of the language in this paper [13], but we were unable to apply it to the PCF-based language considered here.

Andrew Gordon's thesis [2] presents four different systems for input-output in a functional language and gives translations between them, but does not consider applicative approximations. Our system is a variant of his CPS I/O system. Gordon's more recent paper [3] uses Howe's technique to study applicative approximation for an extension of PCF with a far more elaborate type structure than ours. However, his notion of immediate behavior is also based on convergence and does not deal with potentially infinite interactions such as ours.

The definition of applicative approximation by application to identical arguments (as in Definition 3.1.4) can be seen as requiring that terms have the same behavior in some restricted set of contexts, and the equivalence theorem as asserting that if terms have the same behavior in these contexts, then they have the same behavior in all contexts. The oldest such result is Milner's context lemma [9]. Mason and Talcott's ciu theorem [8] provides a similar result for an untyped system with side-effects. Both these systems use convergence as a notion of immediate behavior.

7 Conclusions and Open Problems

We have formulated a notion of behavior for PCF based on potentially infinite interactions. We have defined an applicative approximation for this notion of behavior, and showed that it is equivalent to the corresponding notion of operational approximation. We also have shown that the new approximation is conservative over the usual notion of operational approximation. The new notion should still be useful, however, because it allows us to classify programs with infinite input-output behaviors, such as servers.

We were somewhat surprised to discover that the type system is crucial to the results we have obtained. We have elsewhere considered applicative approximation for a similar, but untyped, system [13]. For the untyped system, Howe's method could be applied (though in an indirect manner) to show that applicative approximation was a precongruence, but the resulting precongruence was smaller than the operational approximation. We were able to show, however, that the applicative approximation was the largest precongruence contained in operational approximation and satisfying a new property that we called "losslessness."

We hope to extend the current work to consider not only single-process input-output but also CCS-style interacting processes.

References

[1] Samason Abramsky. The lazy lambda calculus. In David A. Turner, editor, *Reearch Topics in Functional Programming*, pages 65–116. Addison-Wesley, 1990.

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

[3] Andrew D. Gordon. Bisimilarity as a Theory of Functional Programming. In *Proceedings of 11th Conference on Mathematical Foundations of Programming Semantics*, 1995.

[4] Carl A. Gunter. *Semantics of Programming Languages: Structures and Techniques.* MIT Press, Cambridge, MA, 1992.

[5] Douglas J. Howe. Equality in Lazy Computation Systems. In *Proc. 4th IEEE Symposium on Logic in Computer Science*, pages 198–203, 1989.

[6] Douglas J. Howe. A Note on Proving Congruence of Bisimulation in a Generalized Lambda Calculus. unpublished manuscript, 1995.

[7] Douglas J. Howe. Proving Congruence of Bisimulation in Functional Programming Languages. to appear in *Information and Computation.*, 1996?

[8] Ian A. Mason and Carolyn L. Talcott. Equivalence in Functional Languages with Effects. *Journal of Functional Programming*, 1:287–327, 1991.

[9] Robin Milner. Fully abstract models of typed lambda-calculi. *Theoretical Computer Science*, 4:1–22, 1977.

[10] Robin Milner. Operational and Algebraic Semantics of Concurrent Processes. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 1201–1242. MIT Press/Elsevier, 1990.

[11] Simon L. Peyton Jones and Philip L. Wadler. Imperative Functional Programmiong. In *Proceedings 20th Annual ACM Symposium on Programming Languages*, pages 71–84, 1993.

[12] Gordon D. Plotkin. LCF Considered as a Programming Language. *Theoretical Computer Science*, 5:223–255, 1977.

[13] Jerzy Tiuryn and Mitchell Wand. Untyped Lambda-Calculus with Input-Output (Progress Report). presented at Atlantique meeting, La Jolla, June 1995.