What is a Purely Functional Language?

Amr Sabry[†]

Department of Computer & Information Science
University of Oregon
Eugene, OR 97403
sabry@cs.uoregon.edu

Abstract

Functional programming languages are informally classified into *pure* and *impure* languages. The precise meaning of this distinction has been a matter of controversy. We therefore investigate a formal definition of purity.

We begin by showing that some proposed definitions that rely on confluence, soundness of the beta axiom, preservation of pure observational equivalences, and independence of the order of evaluation, do not withstand close scrutiny. We propose instead a definition based on *parameter-passing independence*. Intuitively, the definition implies that functions are pure mappings from arguments to results; the operational decision of *how* to pass the arguments is irrelevant.

In the context of Haskell, our definition is consistent with the fact that the traditional call-by-name denotational semantics coincides with the traditional call-by-need implementation. Furthermore, our definition is compatible with the stream-based, continuation-based, and monad-based integration of computational effects in Haskell. Finally, we observe that call-by-name reasoning principles are unsound in compilers for monadic Haskell.

1 Functional Languages and Computational Effects

Functional programming languages extend pure λ-calculi with a variety of constructs that are indispensable for programming. Besides simple and functional constants like numbers and addition, all realistic languages include some kind of computational effects. For example, Scheme includes I/O operations, pointers with equality, assignments, and control constructs, SML includes I/O operations, assignments, and exceptions with handlers, and Haskell includes I/O operations.

Despite the presence of computational effects in all realistic functional languages, Scheme and SML have an intuitively different character from Haskell. Indeed, Haskell is referred to as a *purely functional language* (Hudak *et al.*, 1992; Peterson *et al.* , 1996) to distinguish its treatment of computational effects from the Scheme and SML treatment. Attempts to explain why Haskell is purer than Scheme and SML usually lead to confusion and disagreement (as witnessed on the newsgroup

[†] Work started at Chalmers University, 412 96 Göteborg, Sweden.

`comp.lang.functional`). Even in published papers we find varying statements regarding the definition of purity that refer to notions like the soundness of the β-axiom (Odersky *et al.*, 1993), referential transparency (Launchbury & Peyton Jones, 1995), the confluence of a calculus for the language (Swarup *et al.*, 1991), the preservation of pure observational equivalences (O'Hearn, 1995), and the independence of order of evaluation (Launchbury & Peyton Jones, 1995).

The investigation of a formal definition of purity goes beyond settling some differences in opinion. It is crucial at this time when a significant amount of current research aims for efficient realizations of stateful algorithms in functional languages while maintaining the purity of these functional languages. In the absence of a formal definition of purity, we cannot judge the correctness of such extensions. In fact, we cannot even state the correctness properties that need to be proven.

This work is therefore a first step towards reasoning about the imperative extensions of functional languages. Our main point is to propose the following definition of purity:

> A language is purely functional if (i) it includes every simply typed λ-calculus term, and (ii) its call-by-name, call-by-need, and call-by-value implementations are equivalent (modulo divergence and errors).

We will formalize this statement in Section 4.3.

To get to our main result, we proceed according to the following plan:

1. Since the notion of purity is an informal one, we begin with some assumptions. First, we assume that a language is functional (pure or not) if it includes the simply typed λ-calculus. Second, we assume that the following three languages are purely functional: the language Λ that extends the call-by-name λ-calculus with numbers and addition, and the languages PCF and PPCF (Plotkin, 1977). These assumptions may be challenged but they appear to be consistent with the informal practice.
   Finally, two of the informal definitions of purity: referential transparency and independence of order of evaluation, do not have universally agreed-upon definitions and are not considered any further. (See, however, the treatment of referential transparency by Søndergaard and Sestoft (1990).)
2. In Section 2, we specify the semantics of our pure (by assumption) languages and study some of their properties. Any property that holds at this point is potentially relevant to the formal definition of purity.
3. In Section 3, we eliminate many potential definitions of purity using the following strategy.
   On one hand, we extend the pure language Λ to $\Lambda_!$ by adding two expressions, `inc` and `read`, that perform side-effects on an implicit global location that contains a natural number. The language $\Lambda_!$ is reminiscent of canonical impure languages such as Scheme and SML, with the important caveat that it has call-by-name semantics rather than call-by-value semantics. We assume that it should not be classified as purely functional according to any proposed definition. We conclude that any property of Λ that still holds in $\Lambda_!$ is insuf-

ficient to characterize purity. Two such properties are the soundness of the β axiom, and the confluence of the associated calculus.
On the other hand, we extend PCF to PPCF by adding one constant **por** which, by assumption, preserves the purity of the language. Hence, any property of PCF that no longer holds in PPCF is insufficient to characterize purity. In particular, since the addition of **por** breaks some PCF observational equivalences while retaining purity, we conclude that observational equivalence alone cannot characterize purity.

4. The elimination procedure gets rid of most candidate definitions, but it does, however, leave one reasonable alternative: purity means that the language semantics is insensitive to the parameter-passing mechanism. This alternative is explored in Section 4.
5. Our proposed definition has a drawback: it requires the existence of a notion of value, and several evaluation functions (implementations) for the same syntax. Nevertheless, we demonstrate in Section 5 that it is compatible with several designs, some widely used and some less so, for including computational effects in purely functional languages.

Before concluding, we briefly discuss implementations of Haskell with monadic state and their correctness.

2 A Purely Functional Language

We begin our investigation with a canonical, purely functional, call-by-name language Λ.

2.1 Λ *and its Extensions*

The set of Λ-terms extends the language of the λ-calculus (variables, procedures, and applications) with basic and functional constants. For the sake of presentation, we consider a representative set of constants that contains numerals and addition. We do not make any assumptions about the type structure of the language.

Definition 2.1 (Syntax of Λ*)*
Let x, y, z range over an infinite set of variables *Vars*, and n range over the natural numbers:

$$M, N, L \in \mathit{Term} \ ::= \ x \mid \lambda x.M \mid MN \mid n \mid M + N$$

The language has the following context-sensitive properties. In a procedure $(\lambda x.M)$, the variable x is *bound* in the body M. A variable that is not bound is *free*. A term with no free variables is *closed*. Like Barendregt (1984, ch 2,3), we identify terms modulo bound variables and we assume that free and bound variables do not interfere in definitions or theorems. The term $M[N/x]$ is the result of the capture-free substitution of all free occurrences of x in M by N. A *context* C is a term with a *hole* $[\,]$ in the place of one subterm. The operation of *filling* the context C with a term M yields the term $C[M]$, possibly capturing some free variables of M in the process.

The semantics of Λ is a partial function $eval_n$ from programs to observables.

Definition 2.2 (Programs and Observables)
A program is a closed term. An observable B is either a number or the tag proc indicating a procedure. Thus, as usual, the code of a procedure is not observable.

We choose to specify the partial function $eval_n$ using a term rewriting machine since this approach does not require the introduction of many new concepts. The machine states are simply closed terms. To perform a step $\longmapsto$, the machine decomposes the current term into an evaluation context E and a redex, and then rewrites the redex. The full definition follows.

Definition 2.3 ($eval_n$)
The partial function $eval_n$ from terms to observables is defined as: $eval_n(M) = B$ if $M \longmapsto^* A$ and $obs(A) = B$, where $\longmapsto^*$ is the reflexive transitive closure of $\longmapsto$ and:

Answers

$$A \quad ::= \quad n \mid \lambda x.M$$

Evaluation contexts

$$E \quad ::= \quad [\,] \mid EM \mid E + M \mid n + E$$

State transitions

$$\begin{array}{rcll} E[(\lambda x.M)\ N] & \longmapsto & E[M[N/x]] & \\ E[n_1 + n_2] & \longmapsto & E[n] & \quad \text{where } n = n_1 + n_2 \end{array}$$

Observing answers

$$\begin{array}{rcl} obs(n) & = & n \\ obs(\lambda x.M) & = & \mathsf{proc} \end{array}$$

Example 2.4
The following steps of the machine show that $eval_n((\lambda x.x + x)\ (4+2)) = 12$ and $eval_n((\lambda x.xx)\ (\lambda y.y)) =$ proc:

$$\begin{array}{l} (\lambda x.x + x)\ (4+2) \longmapsto (4+2) + (4+2) \longmapsto 6 + (4+2) \longmapsto 6 + 6 \longmapsto 12 \\ (\lambda x.xx)\ (\lambda y.y) \longmapsto (\lambda y.y)(\lambda y.y) \longmapsto (\lambda y.y) \end{array}$$

Sometimes the term rewriting machine gets stuck and can no longer proceed.

Definition 2.5 (Stuck)
A term M where M is an application or an addition is stuck if the term rewriting machine has no transition from that term.

In the remainder of this paper, we will be study the language Λ and its extensions.

Definition 2.6 (Conservative Extension (Felleisen, 1991))
A language $\mathcal{L}_1$ conservatively extends a language $\mathcal{L}_2$ if:

- the set of $\mathcal{L}_1$-terms (programs) includes the set of $\mathcal{L}_2$-terms (programs),
- the of $\mathcal{L}_1$-observables includes the set of $\mathcal{L}_2$-observables, and
- the semantics of $\mathcal{L}_1$ extends the semantics of $\mathcal{L}_2$, *i.e.*, for all $\mathcal{L}_2$-programs M, we have that $eval_{\mathcal{L}_2}(M) = B$ if and only if $eval_{\mathcal{L}_1}(M) = B$.

2.2 Observational Equivalence

As Example 2.4 shows, our interpreter evaluates the arguments to + from left to right. Clearly this choice is arbitrary and it would have been possible to define another interpreter that evaluated the arguments to + from right to left. In general one would not expect the two evaluators to define the same function. However, in the case of Λ, the two evaluators do indeed realize the same function.

Instead of defining a new evaluator and proving its equivalence to the one in Definition 2.3, we show that $M+N$ is indistinguishable from $N+M$ in any context. Thus any two interpreters that differ only in their order of evaluation of the arguments to + would also be indistinguishable. Formally, the notion of being indistinguishable is called *observational equivalence.*

Definition 2.7 (Observational Equivalence $\cong$)
Two terms M and N are observationally equivalent, $M \cong N$, if for all contexts C such that both $C[M]$ and $C[N]$ are programs:

$$eval(C[M]) = B \qquad \text{iff} \qquad eval(C[N]) = B.$$

For the case of Λ, both the axiom β of the λ-calculus and the commutativity of addition are two observational equivalences.

Proposition 2.8
The following are observational equivalences in Λ:

1. $(\lambda x.M)\ N \cong M[N/x]$
2. $M + N \cong N + M$

The proofs are tedious but straightforward. The idea is to set up a relation between machine states containing the left hand side of the equivalence and machine states containing the right hand side. Then it suffices to show that related states rewrite to related states.

2.3 Calculus

A λ-calculus is an equational theory over Λ with a (finite) number of axiom schemas and inference rules. The inference rules extend the axioms to an equivalence relation compatible with contexts (a congruence). The set of axioms should be rich enough to specify the evaluation function but can otherwise include any equalities that are sound with respect to the observational equivalence relation of the language. We write $\vdash M = N$ when $M = N$ is provable in the calculus. By the congruence rules, if $\vdash M = N$ then $\vdash C[M] = C[N]$ for all contexts C.

Definition 2.9 (Axioms for Λ)

A typical calculus for Λ could include the following axioms:

$$\begin{aligned}
(\lambda x.M)\ N &= M[N/x] && (\beta)\\
(M+N)+L &= M+(N+L) && (S)\\
n+M &= M+n && (C)\\
n_1+n_2 &= n \qquad \text{where } n = n_1+n_2 && (A)
\end{aligned}$$

The axioms are all sound with respect to the observational equivalence relation. This guarantees the consistency and correctness of the system. We do not generalize axiom (C) to $N + M = M + N$ as the future extension of the language with assignments would make the more general axiom unsound. Furthermore, the current axioms are sufficient for evaluation.

Lemma 2.10
If $eval_n(M) = B$ then $\vdash M = A$ and $obs(A) = B$.

Proof
The idea is to prove the following statement: If $M \longmapsto N$ and N is not stuck, then $\vdash M = N$. This latter statement follows because the machine's transitions can be performed using the axioms β and A given the compatibility of the relation $=$. □

It is sound to non-deterministically apply the axioms to a program until it reaches an answer: the order of the reductions has no semantic significance. Another way to state this fact is to reason syntactically about the axioms.

Lemma 2.11
If $\vdash M = A_1$ and $\vdash M = A_2$ then $obs(A_1) = obs(A_2)$.

Proof Sketch
The statement is an immediate consequence of the Church-Rosser theorem. It is elementary to check that the Church-Rosser property holds using the following idea. We direct each axiom from left to right to yield a system of reductions. We then divide the reductions into three groups. The first group G_1 includes C and is Church-Rosser because the reduction forms an orthogonal combinatory reduction system (Klop *et al.*, 1993). The second group G_2 includes β which is Church-Rosser (Barendregt, 1984). The third group G_3 includes the remaining reductions S and A and is Church-Rosser because the reflexive closure of the reductions satisfies the diamond property (Barendregt, 1984, Ch.3). The result follows by the Hindley-Rosen Lemma (Barendregt, 1984) since G_2 commutes with G_3, and the union of G_2 and G_3 commutes with G_1. □

3 An Imperative Extension $\Lambda_!$

To study the impact of computational effects on the properties of purely functional languages, we extend our language with expressions whose evaluation performs global side-effects. Any suggested definition of purity should identify such a language as *not* pure.

Definition 3.1 (Syntax of $\Lambda_!$)

The set of terms extends the set in Definition 2.1:

$$M, N, L \in \mathit{Term} \ ::= \ \ldots \mid \mathsf{inc} \mid \mathsf{read}$$

The two new constructs act on an implicit global location which is initialized to 0. Informally speaking, the evaluation of inc returns the current value of the global location and increments it as a side-effect. The evaluation of read returns the current contents of the global location.

The formal semantics is specified using an extension of the term rewriting machine. States are now pairs whose first component is the current program, and whose second component ℓ is the current value of the global location.

Definition 3.2 ($eval_!$)
The partial function $eval_!$ from terms to observables is defined as: $eval_!(M) = B$ if $\langle M, 0\rangle \longmapsto^* \langle A, \ell\rangle$ and $obs(A) = B$. The definitions of answers, evaluation contexts, and obs are identical to the ones in Definition 2.3. The state transitions are:

$$\begin{array}{rcll}
\langle E[((\lambda x.M)\ N)], \ell\rangle & \longmapsto & \langle E[M[N/x]], \ell\rangle & \\
\langle E[n_1 + n_2], \ell\rangle & \longmapsto & \langle E[n], \ell\rangle & \text{where } n = n_1 + n_2 \\
\langle E[\mathsf{read}], \ell\rangle & \longmapsto & \langle E[\ell], \ell\rangle & \\
\langle E[\mathsf{inc}], \ell\rangle & \longmapsto & \langle E[\ell], \ell + 1\rangle &
\end{array}$$

Example 3.3
The following steps of the machine show that $eval_n((\lambda x.x + x)\ \mathsf{inc}) = 1$:

$$\langle(\lambda x.x + x)\ \mathsf{inc}, 0\rangle \longmapsto \langle \mathsf{inc} + \mathsf{inc}, 0\rangle \longmapsto \langle 0 + \mathsf{inc}, 1\rangle \longmapsto \langle 0 + 1, 2\rangle \longmapsto \langle 1, 2\rangle$$

It is straightforward to verify that $\Lambda_!$ is a conservative extension of Λ. In other words, the result of evaluating a pure term using $eval_!$ coincides with the result of evaluating it using $eval_n$.

3.1 Observational Equivalence

As Example 3.3 suggests, β is still, despite the imperative extensions, an observational equivalence of the language.

Proposition 3.4
$(\lambda x.M)\ N \cong_! M[N/x]$

The proposition motivates the following statement.

Fact 3.5
Without further information about a language, the soundness of the β axiom does not guarantee that the language is purely functional.

However, as generally expected from an imperative extension (Felleisen, 1991), the observational equivalence relation of $\Lambda_!$ differs from the one for Λ.

Proposition 3.6
The observational equivalence relations $\cong$ and $\cong_!$ are different.

Proof

From proposition 2.8, we have $x + y \cong y + x$. In $\Lambda_!$, this equivalence no longer holds as the terms can be distinguished by the context $((\lambda x.\lambda y.[\,])\ \mathsf{inc}\ \mathsf{read})$:

$$\begin{array}{l} \mathit{eval}_!((\lambda x.\lambda y.x + y)\ \mathsf{inc}\ \mathsf{read}) = 1 \\ \mathit{eval}_!((\lambda x.\lambda y.y + x)\ \mathsf{inc}\ \mathsf{read}) = 0 \end{array}$$

□

The relationship between the two observational equivalence relations for Λ and $\Lambda_!$ may suggest that purity requires that the observational equivalence relation of a language coincides with that of an underlying purely functional subset. However, we show that a naïve interpretation of this idea is incorrect. It is possible to break some observational equivalences of a purely functional language by extending it with a pure but *non-expressible* (Felleisen, 1991) construct. The standard illustration of this situation are the two purely functional languages PCF and PPCF (Plotkin, 1977).

The language PCF extends the simply typed λ-calculus with constants for expressing recursion, conditionals, and operations on the natural numbers. The language PPCF extends PCF with a parallel (but deterministic) operator **por**. Consider the PCF terms $M(1)$ and $M(2)$ where Ω is a canonical diverging term:

$$\begin{array}{rl} M(u) = \lambda f.\mathbf{if} & (f\ \mathsf{True}\ \Omega) \\ & (\mathbf{if}\ (f\ \Omega\ \mathsf{True}) \\ & \quad (\mathbf{if}\ (f\ \mathsf{False}\ \mathsf{False})\ \Omega\ u) \\ & \quad \Omega) \\ & \Omega \end{array}$$

It is a standard result that $M(1)$ and $M(2)$ are observationally equivalent in PCF but not in PPCF (Plotkin, 1977). The way to distinguish the terms in PPCF is to apply them to **por**. The latter construct bypasses the first two conditional tests as it returns True if either of its arguments is True *even* if the other argument diverges.

This result motivates the following statement.

Fact 3.7
Without further information about a language, the non-preservation of pure observational equivalences does not imply that the language is not purely functional.

3.2 Calculus

As for the pure language, we can also realize the evaluation function for $\Lambda_!$ using a calculus. Our calculus includes all of the axioms of the pure language and some additional axioms that manipulate the imperative constructs. To conveniently express these imperative axioms, we extend the internal syntax of the language with a new construct $(\mathbf{ref}\ \ell\ M)$. A source program is mapped to the internal term $(\mathbf{ref}\ 0\ M)$ and the axioms apply to that latter term. This trick is only necessary because our source language is not a realistic language. Had the language been richer, for example, as rich as Scheme, then all the axioms would be expressible in the source language itself (Sabry & Field, 1993; Felleisen & Hieb, 1992).

$$\begin{tikzcd}
& \mathbf{ref}\ 0\ ((\mathsf{inc}+\mathsf{read})+(\mathsf{inc}+\mathsf{inc})) \arrow[d] & \\
& \mathbf{ref}\ 0\ (\mathsf{inc}+(\mathsf{read}+(\mathsf{inc}+\mathsf{inc}))) \arrow[d] & \\
& \mathbf{ref}\ 1\ (0+(\mathsf{read}+(\mathsf{inc}+\mathsf{inc}))) \arrow[dl] \arrow[dr] & \\
\mathbf{ref}\ 1\ ((\mathsf{read}+(\mathsf{inc}+\mathsf{inc}))+0) \arrow[d] & & \mathbf{ref}\ 1\ ((0+\mathsf{read})+(\mathsf{inc}+\mathsf{inc})) \arrow[d] \\
\mathbf{ref}\ 1\ (\mathsf{read}+((\mathsf{inc}+\mathsf{inc})+0)) \arrow[d] & & \mathbf{ref}\ 1\ ((\mathsf{read}+0)+(\mathsf{inc}+\mathsf{inc})) \arrow[d] \\
\mathbf{ref}\ 1\ (1+((\mathsf{inc}+\mathsf{inc})+0)) \arrow[dr] & & \mathbf{ref}\ 1\ (\mathsf{read}+(0+(\mathsf{inc}+\mathsf{inc}))) \arrow[dl] \\
& \mathbf{ref}\ 1\ (1+(0+(\mathsf{inc}+\mathsf{inc}))) \arrow[dl] \arrow[dr] & \\
\mathbf{ref}\ 1\ (1+((\mathsf{inc}+\mathsf{inc})+0)) \arrow[d] & & \mathbf{ref}\ 1\ ((1+0)+(\mathsf{inc}+\mathsf{inc})) \arrow[d] \\
\mathbf{ref}\ 1\ (((\mathsf{inc}+\mathsf{inc})+0)+1) \arrow[d] & & \mathbf{ref}\ 1\ (1+(\mathsf{inc}+\mathsf{inc})) \arrow[d] \\
\mathbf{ref}\ 1\ ((\mathsf{inc}+(\mathsf{inc}+0))+1) \arrow[d] & & \mathbf{ref}\ 1\ ((1+\mathsf{inc})+\mathsf{inc}) \arrow[d] \\
\mathbf{ref}\ 1\ (\mathsf{inc}+((\mathsf{inc}+0)+1)) \arrow[d] & & \mathbf{ref}\ 1\ ((\mathsf{inc}+1)+\mathsf{inc}) \arrow[d] \\
\mathbf{ref}\ 2\ (1+((\mathsf{inc}+0)+1)) \arrow[d] & & \mathbf{ref}\ 1\ (\mathsf{inc}+(1+\mathsf{inc})) \arrow[d] \\
\mathbf{ref}\ 2\ (((\mathsf{inc}+0)+1)+1) \arrow[d] & & \mathbf{ref}\ 2\ (1+(1+\mathsf{inc})) \arrow[d] \\
\mathbf{ref}\ 2\ ((\mathsf{inc}+(0+1))+1) \arrow[d] & & \mathbf{ref}\ 2\ ((1+1)+\mathsf{inc}) \arrow[d] \\
\mathbf{ref}\ 2\ (\mathsf{inc}+((0+1)+1)) \arrow[d] & & \mathbf{ref}\ 2\ (2+\mathsf{inc}) \arrow[d] \\
\mathbf{ref}\ 3\ (2+((0+1)+1)) \arrow[d] & & \mathbf{ref}\ 2\ (\mathsf{inc}+2) \arrow[d] \\
\mathbf{ref}\ 3\ (2+(1+1)) \arrow[dr] & & \mathbf{ref}\ 3\ (2+2) \arrow[dl] \\
& \mathbf{ref}\ 3\ (2+2) \arrow[d] & \\
& \mathbf{ref}\ 3\ 4 &
\end{tikzcd}$$

Fig. 1. Rewriting imperative terms

Definition 3.8 (Axioms for $\Lambda_!$)

A possible set of axioms includes the axioms in Definition 2.9 and:

$$\begin{aligned}
\mathbf{ref}\ \ell\ \mathsf{read} &= \mathbf{ref}\ \ell\ \ell && (R_1)\\
\mathbf{ref}\ \ell\ (\mathsf{read}+M) &= \mathbf{ref}\ \ell\ (\ell+M) && (R_2)\\
\mathbf{ref}\ \ell\ \mathsf{inc} &= \mathbf{ref}\ (\ell+1)\ \ell && (I_1)\\
\mathbf{ref}\ \ell\ (\mathsf{inc}+M) &= \mathbf{ref}\ (\ell+1)\ (\ell+M) && (I_2)
\end{aligned}$$

As before the calculus is confluent and the axioms are sufficient for evaluation.

Lemma 3.9

If $\vdash (\text{ref}\ 0\ M) = (\text{ref}\ \ell_1\ A_1)$ and $\vdash (\text{ref}\ 0\ M) = (\text{ref}\ \ell_2\ A_2)$ then $obs(A_1) = obs(A_2)$.

Lemma 3.10

If $eval_!(M) = B$ then $\vdash (\text{ref}\ 0\ M) = (\text{ref}\ \ell\ A)$ and $obs(A) = B$.

In other words, we can still evaluate a program by non-deterministically applying the axioms until we reach an answer: the order of reductions does not affect the relative order of the imperative operations. (See Figure 1 for 4 different proofs that $eval_!((\mathsf{inc} + \mathsf{read}) + (\mathsf{inc} + \mathsf{inc})) = 4$.)

This result motivates the following fact.

Fact 3.11
Without further information about a language, the confluence of a calculus for the language does not guarantee that the language is purely functional.

4 Practical Implementations of Λ and $\Lambda_!$

With all the negative results in the previous section, one might suspect that we have missed some fundamental property of purely functional languages. Indeed, we have not at all considered their implementations in practice and the connection between the semantics and the implementation. We therefore examine a practical implementation of the language Λ.

4.1 Call-by-Need

The call-by-need evaluator achieves an efficient realization of $eval_n$ by *sharing* the evaluation of non-trivial expressions. These expressions are easy to identify from our semantic specifications: any expression that is reduced when in the hole of an evaluation context is non-trivial. For example, in the language Λ, both applications and additions are non-trivial (see Definition 2.3). The other expressions, called *syntactic values*, have a trivial evaluation, and are allowed to be duplicated and hence re-evaluated several times.

Definition 4.1 (Syntactic Value)
In Λ, the following subset of terms are syntactic values (Plotkin, 1975):

$$V \ ::= \ n \mid \lambda x.M$$

Definition 4.2
The call-by-need evaluator is defined (Ariola *et al.*, 1995; Ariola & Felleisen, 1996) as follows: $eval_z(M) = B$ if $M \longmapsto^* A$ and $obs(A) = B$, where:

Answers
$$A \quad ::= \quad V \mid (\lambda x.A)\ M$$

Evaluation contexts
$$E \quad ::= \quad [\,] \mid EM \mid E + M \mid n + E \mid (\lambda x.E)\ M \mid (\lambda x.E[x])\ E$$

State transitions
$$\begin{array}{rcll} E_1[(\lambda x.E_2[x])\ V] & \longmapsto & E_1[(\lambda x.E_2[V])\ V] & \\ E[n_1 + n_2] & \longmapsto & E[n] & \text{where } n = n_1 + n_2 \end{array}$$

$$\begin{array}{rcl} E[(\lambda x.A)\ M\ N] & \longmapsto & E[(\lambda x.AN)\ M] \\ E_1[(\lambda x.E_2[x])\ ((\lambda y.A)\ M)] & \longmapsto & E_1[(\lambda y.(\lambda x.E_2[x])\ A)\ M] \end{array}$$

Observing answers

$$\begin{array}{rcl} obs(n) & = & n \\ obs(\lambda x.M) & = & \mathsf{proc} \\ obs((\lambda x.A)\ M) & = & obs(A) \end{array}$$

The call-by-need implementation is correct since it defines the same partial function as the call-by-name implementation (Ariola *et al.*, 1995; Ariola & Felleisen, 1996).

Theorem 4.3
If $eval_n(M) = B_1$ and $eval_z(M) = B_2$ then $B_1 = B_2$

Furthermore, the call-by-need evaluation is expected to be much more efficient in practice.

To define the call-by-need evaluator for $\Lambda_!$, we must decide whether the new expressions read and inc are syntactic values or not. Again the answer is evident from the reductions in Definition 3.2. Both read and inc are reducible when in the hole of an evaluation context, and hence are not values.

It is now easy to see that a call-by-need evaluator for $\Lambda_!$ would not be observationally equivalent to the call-by-name one. For example, we have:

$$eval_!((\lambda x.x + x)\ \mathsf{inc}) = eval_!(\mathsf{inc} + \mathsf{inc}) = 1$$

But instead attempting to optimize the interpreter $eval_!$ by sharing the evaluation of the non-value inc would produce 0. Thus the equivalence of call-by-name and call-by-need that is crucial for the efficient implementation of Λ does not hold for $\Lambda_!$. Implementations of $\Lambda_!$ cannot rely on laziness to implement the non-strict semantics.

This observation suggests that purity manifests itself in practice when we are trying to combine different parameter-passing mechanisms (in the specification of the semantics and in the implementation). It is therefore reasonable to conjecture that purity implies that these different parameter-passing mechanisms are equivalent.

4.2 Call-by-Value

Having identified a possible connection between purity and parameter-passing, we study the rôle of call-by-value in this context. Consider a (malicious?) implementor who used a call-by-value evaluator to realize the semantic function in Definition 2.3. What would be wrong?

Definition 4.4

The call-by-value evaluator is defined as: $eval_v(M) = B$ if $M \longmapsto^* A$ and $obs(A) = B$. Answers and *obs* are identical to the ones in Definition 2.3:

$$\begin{array}{l}
\textit{Evaluation contexts} \\
E \quad ::= \quad [\,] \mid EM \mid E+M \mid n+E \mid VE \\
\\
\textit{State transitions} \\
\begin{array}{rcll}
E[(\lambda x.M)\ V] & \longmapsto & E[M[V/x]] & \\
E[n_1+n_2] & \longmapsto & E[n] & \text{where } n = n_1+n_2
\end{array}
\end{array}$$

The call-by-value evaluator is not correct in the sense that it defines a different partial function from $eval_n$. But could a user ever observe a difference? If the call-by-name semantics specifies that a program should terminate with an observable answer, then the call-by-value evaluator will either:

1. terminate with the same observable answer, or
2. not terminate.

In the first case, the user observes the same behavior. In the second case, the user does not observe anything and hence cannot ascertain that the evaluator is incorrect: maybe it is just slow.

Proposition 4.5
If $eval_n(M) = B$ then either:

- $eval_v(M) = B$, or
- $eval_v(M)$ is undefined.

Conversely, if $eval_v(M) = B$ then $eval_n(M) = B$.

4.3 Thesis

The previous two subsections motivate the following definition.

Definition 4.6 (Weak Equivalence)
Let P be a set of programs, B be a set of observables, and $eval_1$ and $eval_2$ be two partial functions (implementations) from programs to observables. We say $eval_1$ is weakly equivalent to $eval_2$ when the following conditions hold:

- If $eval_1(P) = B$ then either $eval_2(P) = B$ or $eval_2(P)$ is undefined.
- If $eval_2(P) = B$ then either $eval_1(P) = B$ or $eval_1(P)$ is undefined.

We can now formulate our thesis precisely.

Definition 4.7 (Purely Functional Language)
A language is purely functional if:

1. it is a conservative extension of the simply typed λ-calculus,
2. it has well-defined call-by-value, call-by-need, and call-by-name evaluation functions (implementations), and
3. all three evaluation functions (implementations) are weakly equivalent.

There are several important points to note:

- The first condition in the definition requires that the language be a conservative extension of the simply typed λ-calculus. This condition guards against languages with no functions, and hence that would vacuously satisfy the second and third conditions.
- Among the many parameter-passing mechanisms we have selected call-by-value, call-by-name, and call-by-need as the relevant ones for the thesis. This choice appears to work well as it allows us to verify that the subset of SML (a call-by-value language) without assignments and exceptions is pure, and also that Haskell (a language with a call-by-name denotational semantics and a call-by-need implementation) is pure. It may be the case that the thesis could be formulated with only two of the parameter-passing mechanisms, for example by omitting call-by-value entirely. This new thesis would essentially be about *sharing* of computations since this is the fundamental difference between call-by-name and call-by-need. We leave this point as an open problem.
- A drawback of this definition is that it requires the existence of several evaluation functions (implementations) for the same syntax. Starting from a call-by-value language like Scheme, it is straightforward to devise a call-by-need or call-by-name evaluator. However, starting from a call-by-name language like $\Lambda_!$ or Idealized Algol (Reynolds, 1991; Reynolds, 1981; Reynolds, 1988), the design of the call-by-value or call-by-need variant first requires setting a notion of syntactic value. This latter decision affects the purity of the language. Indeed, as we will see in the next section, by varying the notion of value in $\Lambda_!$, we can design a new variant of the language that is purely functional.
- The thesis follows the convention that non-termination and errors are special kinds of computation whose effects are not observable. Hence expressions that diverge, or evaluate to a black hole, or an error are all considered equivalent. If errors become observable, then not even PCF would be pure (Cartwright & Felleisen, 1991; Cartwright *et al.*, 1993).

5 Case Studies

Using our proposed definition it is straightforward to confirm some common claims. For example, the subsets of Scheme and SML excluding assignments, pointer equality, exceptions, and control operators are purely functional, and their extensions with assignments, *call/cc*, or *eq?* are not purely functional. Also Haskell is pure as long as one observes neither errors nor non-termination (black holes). To show the applicability of our definition beyond these simple examples, we study several extended languages in this section.

5.1 Effects as Values

Given our definition of purity, the design of a purely functional variant of $\Lambda_!$ requires the construction of call-by-value, call-by-need, and call-by-name evaluation functions that behave similarly.

These evaluation functions already differ on simple programs like $((\lambda x.x+x)$ inc) as the program evaluates to 0 using call-by-value or call-by-need but evaluates to 1 using call-by-name. An obvious way of making the evaluation functions agree on the program is to treat the expressions inc and read as values, which we write as **incM** and **readM** for clarity. This implies that the program would be equivalent to $(\texttt{incM} + \texttt{incM})$ using any parameter-passing mechanism.

But this only solves part of the problem. Consider now the term:

$$(\lambda x.x+x)\ (\texttt{incM} + \texttt{readM})$$

The argument is not a value and again we are in a situation where the program evaluates to different results under call-by-value and call-by-name. The solution is however as simple as before: treat the expression $(\texttt{incM} + \texttt{readM})$ as a (constructed) value, which we write as (**Plus incM readM**) for clarity.

We have thus arranged for the evaluation of $((\lambda x.\texttt{Plus}\ x\ x)\ \texttt{incM})$ to produce the value (**Plus incM incM**) as its final answer using any parameter-passing mechanism. The evaluation does not perform any computational effects but just collects the demands for computational effects and propagates them to the top level of the program as the final answer. The mapping of answers to observables would need to perform the computational effects to print the expected answer of 1.

Putting things together the formal syntax and semantics of our language are now as follows.

Definition 5.1 (Syntax of Λ_s)
The set of terms is defined as:

$$\begin{aligned} M, N, L \in \mathit{Term} &::= x \mid \lambda x.M \mid MN \mid n \mid \texttt{Plus}\ M\ N \mid \texttt{incM} \mid \texttt{readM} \\ V \in \mathit{Value} &::= n \mid \lambda x.M \mid \texttt{Plus}\ V_1\ V_2 \mid \texttt{incM} \mid \texttt{readM} \end{aligned}$$

Definition 5.2 ($eval_s$)
The partial function $eval_s$ from terms to observables is defined as: $eval_s(M) = B$ if $M \longmapsto^* A$ and $obs(A) = B$, where:

Answers

$$A \ ::= \ n \mid \lambda x.M \mid \texttt{Plus}\ A_1\ A_2 \mid \texttt{incM} \mid \texttt{readM}$$

Evaluation contexts

$$E \ ::= \ [\,] \mid EM \mid \texttt{Plus}\ E\ M \mid \texttt{Plus}\ V\ E$$

State transitions

$$E[(\lambda x.M)\ N] \ \longmapsto \ E[M[N/x]]$$

The mapping of answers to observables is more complicated than usual since it needs to perform all the effects. We specify this mapping using an abstract machine of its own.

Definition 5.3 (Observing Answers)

We define $obs(A) = B$ if $\langle A, 0\rangle \longmapsto^* \langle B, \ell\rangle$ where:

$$\begin{array}{l}
\textit{Evaluation contexts} \\
\quad E \quad ::= \quad [\,] \mid \texttt{Plus}\ E\ A \mid \texttt{Plus}\ n\ E \\
\\
\textit{State transitions} \\
\begin{array}{rcll}
\langle E[\lambda x.M], \ell\rangle & \longmapsto & \langle E[\mathsf{proc}], \ell\rangle & \\
\langle E[\texttt{readM}], \ell\rangle & \longmapsto & \langle E[\ell], \ell\rangle & \\
\langle E[\texttt{incM}], \ell\rangle & \longmapsto & \langle E[\ell], \ell + 1\rangle & \\
\langle E[\texttt{Plus}\ n_1\ n_2], \ell\rangle & \longmapsto & \langle E[n], \ell\rangle & \qquad \text{where } n = n_1 + n_2
\end{array}
\end{array}$$

Example 5.4
We have $eval_s((\lambda x.\lambda y.\texttt{Plus (Plus } x\ y\texttt{) (Plus } x\ x\texttt{)) incM readM}) = 4$. For clarity we use $\longmapsto_f$ for the reduction steps of the main (functional) evaluator and $\longmapsto_o$ for the reduction steps of the observer:

Functional evaluation

$$\begin{array}{ll}
 & (\lambda x.\lambda y.\texttt{Plus (Plus } x\ y\texttt{) (Plus } x\ x\texttt{)) incM readM} \\
\longmapsto_f & (\lambda y.\texttt{Plus (Plus incM } y\texttt{) (Plus incM incM)) readM} \\
\longmapsto_f & \texttt{Plus (Plus incM readM) (Plus incM incM)}
\end{array}$$

Observing the answer

$$\begin{array}{ll}
 & \langle\texttt{Plus (Plus incM readM) (Plus incM incM)}, 0\rangle \\
\longmapsto_o & \langle\texttt{Plus (Plus } 0\ \texttt{readM) (Plus incM incM)}, 1\rangle \\
\longmapsto_o & \langle\texttt{Plus (Plus } 0\ 1\texttt{) (Plus incM incM)}, 1\rangle \\
\longmapsto_o & \langle\texttt{Plus } 1\ \texttt{(Plus incM incM)}, 1\rangle \\
\longmapsto_o & \langle\texttt{Plus } 1\ \texttt{(Plus } 1\ \texttt{incM)}, 2\rangle \\
\longmapsto_o & \langle\texttt{Plus } 1\ \texttt{(Plus } 1\ 2\texttt{)}, 3\rangle \\
\longmapsto_o & \langle\texttt{Plus } 1\ 3, 3\rangle \\
\longmapsto_o & \langle 4, 3\rangle
\end{array}$$

To justify our claim that the above language is purely functional, we should define a call-by-value and call-by-need evaluation functions and show their weak equivalence of $eval_s$. Both variants of the semantics are as expected. Much like the pure call-by-value semantics (Definition 4.4), the call-by-value variant has one additional kind of evaluation context, VE, and it replaces the state transitions of Definition 5.2 with:

$$E[(\lambda x.M)\ V] \quad \longmapsto \quad E[M[V/x]]$$

The call-by-need variant is similarly defined following Definition 4.2. It is almost evident that both variants of the semantics are weakly equivalent to $eval_s$. Indeed, ignoring the mapping from answers to observables which does not involve any procedure calls (and hence does not depend on our notion of parameter-passing), the language is just an applied λ-calculus that includes simple constants and datatypes. The call-by-value, call-by-need, and call-by-name evaluation functions are known to be weakly equivalent for this language.

Proposition 5.5

The language Λ_s is purely functional.

In summary, the idea of the language Λ_s is to treat all expressions that perform effects as values, collect these expressions in some data structure as part of the answer, and perform the effects by a conceptually separate evaluator after all functions have disappeared. This idea originates with the design of Idealized Algol (Reynolds, 1988; Reynolds, 1991; Reynolds, 1981) where the evaluation proceeds as follows: in a first phase, perform all β-steps producing an imperative program, and in a second phase performs all the imperative operations. The only catch is that the imperative program resulting from an Idealized Algol program may be infinite, so this view is only conceptual (Weeks & Felleisen, 1993). In practice the two evaluators would be implemented as coroutines. It is interesting to note that O'Hearn (1995) shows that the observational equivalence of the full Idealized Algol language conservatively extends the observational equivalence of the functional sublanguage, which might be interpreted as evidence for the purity of Idealized Algol.

The idea is also reminiscent of the stream I/O model in Haskell (Hudak *et al.*, 1992) where for example, instead of having side-effecting expressions like writeFile, we have a datatype of `Request` that includes a data constructor (*i.e.*, a value) `WriteFile`. These values that refer to I/O operations are accumulated in a stream and performed at the top level. Again the number of I/O operations in the stream is unbounded, so the phase separation is only conceptual.

5.2 Effects as Monadic Operations

One of the fundamental properties of continuation-passing style (CPS) terms is that they are independent of the parameter-passing technique (Plotkin, 1975; Reynolds, 1972). This suggests a way to embed computational effects in a purely functional language: force all the imperative parts of the program to be written in CPS or the closely related effect-passing style (EPS) based on monads (Filinski, 1994; Filinski, 1996; Wadler, 1990). Intuitively, both CPS and EPS require programmers to explicitly sequence the imperative operations, and hence remove any ambiguity associated with the parameter-passing mechanism.

A naïve implementation of this idea would simply restrict all parts of a program (pure and impure) to be written in EPS. Unfortunately, this would *not* yield a conservative extension of the simply typed λ-calculus (see Definition 2.6). What we need instead is the ability to write the imperative parts of the program in EPS, and the pure parts as before. To implement this idea correctly, we use the state monad to explicitly sequence the imperative operations, leaving the pure sublanguage alone.

For example, consider the $\Lambda_!$ term $((\lambda x.x+x)\ \mathsf{inc})$, where the increment operation is performed once or twice depending on the semantics of function application. This term is now illegal. Instead we extend our language with `return` and `>>=`: the *unit* and *bind* operations of the state monad. If we call `incM` the variant of inc acting on monadic state, then we might write the term as:

$$\texttt{incM >>= }\lambda v_1\texttt{.incM >>= }\lambda v_2\texttt{.return}\ (v_1 + v_2), \quad \text{or}$$
$$\texttt{incM >>= }\lambda v\texttt{.return}\ (v + v)$$

depending on our interpretation. Note that the evaluation of each of the latter two terms is insensitive to the parameter-passing mechanism.

To formulate the evaluation function, we need an additional construct ($\texttt{run}\ M$) that marks the top level of a program. The reason for this additional construct is that monadic operations on the state are *only* performed at top level. Like in the previous section, imperative operations embedded deep inside the program are not performed there but are propagated to the top level using the monadic combinator `>>=`, and only performed during a conceptually second phase of evaluation. This intuition is made precise in the definitions of evaluation contexts and standard reductions below.

Definition 5.6 (Syntax of Λ_m)
The set of terms is defined as:

$$\begin{array}{rcl}
T \in \mathit{TopTerm} & ::= & M \mid \texttt{run}\ M \\
M, N, L \in \mathit{Term} & ::= & x \mid \lambda x.M \mid MN \mid n \mid M + N \\
 & & \mid \texttt{return}\ M \mid M \ \texttt{>>=}\ N \mid \texttt{readM} \mid \texttt{incM} \\
V \in \mathit{Value} & ::= & n \mid \lambda x.M \mid \texttt{return}\ M \mid M \ \texttt{>>=}\ N \mid \texttt{readM} \mid \texttt{incM}
\end{array}$$

As explained above, if the construct `run` occurs in a term, it must occur at the top level. The (implicit) monadic state is just the value of the global location manipulated by `readM` and `incM`.

Definition 5.7 (eval_m)
The partial function eval_m from terms to observables is defined as: $\mathit{eval}_m(T) = B$ if $T \longmapsto^* A$ and $\mathit{obs}(A) = B$, where:

Answers

$$\begin{array}{rcl}
A & ::= & V \mid \texttt{run}\ W \\
V & ::= & n \mid \lambda x.M \mid \texttt{return}\ M \mid M \ \texttt{>>=}\ N \mid \texttt{readM} \mid \texttt{incM} \\
W & ::= & \texttt{readM} \mid \texttt{incM} \mid \texttt{return}\ I \mid W \ \texttt{>>=}\ \lambda x.W \\
I & ::= & n \mid x \mid I + I
\end{array}$$

Evaluation contexts

$$\begin{array}{rcl}
G & ::= & E \mid \texttt{run}\ F \\
E & ::= & [\,] \mid EM \mid E + M \mid I + E \\
F & ::= & E \mid \texttt{return}\ E \mid F \ \texttt{>>=}\ M \mid W \ \texttt{>>=}\ E \mid W \ \texttt{>>=}\ \lambda x.F
\end{array}$$

State transitions

$$\begin{array}{rcll}
G[(\lambda x.M)N] & \longmapsto & [M[N/x]] & \\
G[n_1 + n_2] & \longmapsto & G[n] & \text{where } n = n_1 + n_2
\end{array}$$

As in the previous case, the mapping of answers to observables is complicated since it performs all the effects. We specify this mapping using an abstract machine of its own.

Definition 5.8 (Observing answers)

We define $obs(A) = B$ if $\langle A, 0\rangle \longmapsto^* \langle B, \ell\rangle$ where:

$$\begin{array}{l}
\textit{Evaluation contexts} \\
E \quad ::= \quad [\,] \mid E + M \mid n + E
\end{array}$$

$$\begin{array}{rcl}
\multicolumn{3}{l}{\textit{State transitions}} \\
\langle \lambda x.M, \ell\rangle & \longmapsto & \langle \mathsf{proc}, \ell\rangle \\
\langle \texttt{return}\ M, \ell\rangle & \longmapsto & \langle \mathsf{proc}, \ell\rangle \\
\langle M\ \texttt{>>=}\ N, \ell\rangle & \longmapsto & \langle \mathsf{proc}, \ell\rangle \\
\langle \texttt{readM}, \ell\rangle & \longmapsto & \langle \mathsf{proc}, \ell\rangle \\
\langle \texttt{incM}, \ell\rangle & \longmapsto & \langle \mathsf{proc}, \ell\rangle \\
\\
\langle \texttt{run readM}, \ell\rangle & \longmapsto & \langle \ell, \ell\rangle \\
\langle \texttt{run incM}, \ell\rangle & \longmapsto & \langle \ell, \ell + 1\rangle \\
\langle \texttt{run}\ (\texttt{return}\ n), \ell\rangle & \longmapsto & \langle n, \ell\rangle
\end{array}$$

$$\begin{array}{rcl}
\langle \texttt{run}\ (\texttt{return}\ E[n_1 + n_2]), \ell\rangle & \longmapsto & \langle \texttt{run}\ (\texttt{return}\ E[n]), \ell\rangle \\
 & & \text{where } n = n_1 + n_2 \\
\langle \texttt{run}\ (\texttt{readM}\ \texttt{>>=}\ \lambda x.W), \ell\rangle & \longmapsto & \langle \texttt{run}\ (W[x := \ell]), \ell\rangle \\
\langle \texttt{run}\ (\texttt{incM}\ \texttt{>>=}\ \lambda x.W), \ell\rangle & \longmapsto & \langle \texttt{run}\ (W[x := \ell]), \ell + 1\rangle \\
\langle \texttt{run}\ ((\texttt{return}\ n)\ \texttt{>>=}\ \lambda x.W), \ell\rangle & \longmapsto & \langle \texttt{run}\ (W[x := n]), \ell\rangle \\
\langle \texttt{run}\ ((\texttt{return}\ E[n_1 + n_2])\ \texttt{>>=}\ \lambda x.W), \ell\rangle & \longmapsto & \langle \texttt{run}\ ((\texttt{return}\ E[n])\ \texttt{>>=}\ \lambda x. \\
 & & \qquad W), \\
 & & \ \ell\rangle \\
 & & \text{where } n = n_1 + n_2 \\
\langle \texttt{run}\ ((W\ \texttt{>>=}\ \lambda x.W_1)\ \texttt{>>=}\ \lambda y.W_2), \ell\rangle & \longmapsto & \langle \texttt{run}\ (W\ \texttt{>>=}\ \lambda x. \\
 & & \qquad W_1\ \texttt{>>=}\ \lambda y.W_2)), \\
 & & \ \ell\rangle
\end{array}$$

Example 5.9

For example, the term $(\texttt{run}\ (\texttt{incM}\ \texttt{>>=}\ \lambda v_1.\texttt{incM}\ \texttt{>>=}\ \lambda v_2.\texttt{return}\ (v_1 + v_2)))$ evaluates to 1. The functional evaluation terminates immediately and all the computation happens during the observation part:

$$\begin{array}{rl}
 & \langle \texttt{run}\ (\texttt{incM}\ \texttt{>>=}\ \lambda v_1.\texttt{incM}\ \texttt{>>=}\ \lambda v_2.\texttt{return}\ (v_1 + v_2)), 0\rangle \\
\longmapsto & \langle \texttt{run}\ (\texttt{incM}\ \texttt{>>=}\ \lambda v_2.\texttt{return}\ (0 + v_2)), 1\rangle \\
\longmapsto & \langle \texttt{run}\ (\texttt{return}\ (0 + 1)), 2\rangle \\
\longmapsto & \langle \texttt{run}\ (\texttt{return}\ 1), 2\rangle \\
\longmapsto & \langle 1, 2\rangle
\end{array}$$

Why is the above language purely functional? The argument is similar to the one in the previous section. The evaluation is clearly divided into two separate phases. In the mapping from answers to observables, all substitutions involve values (and hence are valid in call-by-value, call-by-need, and call-by-name semantics). Abstracting from the way answers are observed, the language is just an applied λ-calculus in which answers are trees. Changing the evaluation contexts and standard reductions in Definition 5.7 to either call-by-value or call-by-need will either produce the same tree as the call-by-name semantics or diverge. The analogy to the previous case is not surprising since Peyton Jones and Wadler (1993) demonstrate that there is a close relationship among the stream-based, monad-based, and continuation-based integration of computational effects in Haskell.

Proposition 5.10
The language Λ_m is purely functional.

The language Λ_m is a miniature version of the State in Haskell language (Launchbury & Peyton Jones, 1995), which Launchbury and Peyton Jones informally argue is pure:

> A formal proof would necessarily involve some operational semantics, and a proof that no evaluation order could change the behaviour of the program. We have not yet undertaken such a proof (Launchbury & Peyton Jones, 1995, p.322).

We have already developed a call-by-name operational semantics for the full State in Haskell (Launchbury & Sabry, 1997) language. To prove that the language is pure according to our definition, it remains to develop call-by-value and call-by-need variants of the semantics and show their weak equivalence.

5.3 Implementation

The language Λ_m can by implemented with the same tradeoffs as the language of State in Haskell (Launchbury & Peyton Jones, 1995). We describe two possible implementations: a functional one and an imperative one. The first phase of both implementations translates the source programs by expressing `return`, `>>=`, and `run` in store-passing style. For convenience, the target language of this translation includes, like Haskell, pairs, let-expressions, and pattern-matching with the usual semantics.

Definition 5.11
The translation of Λ_m is defined as follows:

$$\begin{array}{rcl} x^* & = & x \\ (\lambda x.M)^* & = & \lambda x.M^* \\ (MN)^* & = & M^*\ N^* \\ n^* & = & n \\ (M+N)^* & = & M^* + N^* \end{array} \qquad \begin{array}{rcl} (\texttt{return}\ M)^* & = & \lambda \ell.\langle M^*, \ell\rangle \\ (M\ \texttt{>>=}\ N)^* & = & \lambda \ell.\texttt{let}\ \langle x, \ell'\rangle = M^*\ell \\ & & \quad \texttt{in}\ N^* x \ell' \\ (\texttt{run}\ M)^* & = & \texttt{fst}\ (M^*\ 0) \\ \texttt{readM}^* & = & \lambda \ell.\texttt{readM}\ \ell \\ \texttt{incM}^* & = & \lambda \ell.\texttt{incM}\ \ell \end{array}$$

The second phases of the two implementations differ as follows: the functional implementation treats the operations `incM` and `readM` as state transformers, *i.e.*, functions that take an input store as one of their arguments and return an output store as part of their result:

$$\begin{array}{rcl} \texttt{readM} & = & \lambda \ell.\langle \ell, \ell\rangle \\ \texttt{incM} & = & \lambda \ell.\langle \ell, \ell+1\rangle \end{array}$$

This implementation is close to the semantics of the language but would be rather inefficient in practice as it implements updates by copying (parts of) the store data structure. The intermediate language of the functional implementation is clearly pure but not interesting as the basis for a compiler.

The imperative implementation generates code for `incM` and `readM` that *ignores* the store argument and performs *destructive* updates that operate on a *global* location. This is clearly more efficient but is not evidently correct. Indeed, we show that if the semantics of the intermediate language is call-by-name then the implementation strategy based on destructive updates is incorrect. Consider the following term:

$$\texttt{run}\ (\texttt{incM}\ \texttt{>>=}\ \lambda x.\texttt{return}\ (x+x))$$

whose value according to the semantics is 0. The translation of the term into the intermediate language produces:

$$\texttt{fst}\ (\lambda\ell.(\texttt{let}\ \langle a,\ell\rangle = \texttt{incM}\ \ell\ \texttt{in}\ (\lambda x.\lambda\ell.\langle(x+x),\ell\rangle)\ a\ \ell)\ 0)$$

which, if the intermediate language has call-by-name semantics could be simplified as follows:

$$\begin{aligned}
&= \texttt{fst}\ (\texttt{let}\ \langle a,\ell\rangle = \texttt{incM}\ 0\ \texttt{in}\ \langle(a+a),\ell\rangle)\\
&= \texttt{fst}\ (\texttt{let}\ p = \texttt{incM}\ 0\ \texttt{in}\ \langle(\texttt{fst}\ p+\texttt{fst}\ p),\texttt{snd}\ p\rangle)\\
&= \texttt{fst}\ \langle(\texttt{fst}\ (\texttt{incM}\ 0)+\texttt{fst}\ (\texttt{incM}\ 0)),\texttt{snd}\ (\texttt{incM}\ 0)\rangle\\
&= (\texttt{fst}\ (\texttt{incM}\ 0)+\texttt{fst}\ (\texttt{incM}\ 0))
\end{aligned}$$

If `incM` is implemented as an expression that ignores its state argument and instead performs a global side effect, then the above term evaluates to 1 instead of 0.

Launchbury and Peyton Jones (1995) informally argue that the above evaluation strategy as realized in the Glasgow Haskell compiler (ghc) is correct. Clearly, as we demonstrate above, ghc cannot use arbitrary β-reductions on the intermediate representation of the program. Fortunately, even before the monadic extensions, ghc was careful not to duplicate work and hence refrained from using β steps for performance reasons (Ariola *et al.*, 1995). Consequently, the addition of assignments to the back end did not cause any immediate problems. The correctness of the destructive implementation of monadic state is however still an open problem.

6 Conclusion

The paper proposes a framework for reasoning about purely functional languages and their extensions with computational effects. We have put forward the thesis that purity can be determined by the (weak) equivalence of call-by-name, call-by-value, and call-by-need. This definition of purity naturally motivates and explains the various strategies used to integrate computational effects with purely functional languages.

Building on the thesis, we propose a way to formally reason about the correctness of the destructive implementation of monadic operations. We also reveal the unsoundness of call-by-name reasoning principles in compilers for monadic Haskell and hence the importance of call-by-need theories that are rich enough to express imperative operations.

Acknowledgments

I would like to thank Lennart Augustsson, Matthias Felleisen, Robert Harper, Peter O'Hearn, and Uday Reddy for criticism and comments on an early (and rough) draft. I have also benefited from interesting discussions with Zena Ariola, Magnus Carlsson, Thomas Hallgren, John Hughes, John Launchbury, Johan Nordlander, Lars Pareto, Simon Peyton Jones, Miley Semmelroth, Thomas Streicher, and Walid Taha. The referees as well as the editor Philip Wadler offered valuable advice that sharpened both the ideas and the presentation.

References

Ariola, Zena M., & Felleisen, Matthias. (1996). *The call-by-need lambda calculus*. To appear in the *Journal of Functional Programming*.

Ariola, Zena M., Felleisen, Matthias, Maraist, John, Odersky, Martin, & Wadler, Philip. (1995). A call-by-need lambda calculus. *Pages 233–246 of: ACM Symposium on Principles of Programming Languages.*

Barendregt, H. P. (1984). *The lambda calculus: Its syntax and semantics*. Revised edn. Studies in Logic and the Foundations of Mathematics, vol. 103. North-Holland.

Cartwright, R., & Felleisen, Matthias. 1991 (August). *Observable sequentiality and full abstraction*. Tech. rept. 91-167. Rice University. Preliminary version in: *Proc. 19th ACM Symposium on Principles of Programming Languages* (1992), pp. 328–342.

Cartwright, R., Curien, P.-L., & Felleisen, Matthias. 1993 (December). *Fully abstract semantics for observably sequential languages*. Tech. rept. 93-219. Rice University. Also appears in *Information and Computation* **111** (2), 1994, 297–401.

Felleisen, Matthias. (1991). On the expressive power of programming languages. *Pages 35–75 of: Science of Computer Programming*, vol. 17. Preliminary version in: *Proc. European Symposium on Programming*, Lecture Notes in Computer Science, 432. Springer-Verlag (1990), 134–151.

Felleisen, Matthias, & Hieb, R. (1992). The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, **102**, 235–271. Technical Report 89-100, Rice University.

Filinski, Andrzej. (1994). Representing monads. *Pages 446–457 of: ACM Symposium on Principles of Programming Languages.*

Filinski, Andrzej. (1996). *Controlling effects*. Ph.D. thesis, Carnegie Mellon University. Available as Technical Report CS-96-119.

Hudak, Paul, Peyton Jones, Simon L., & Wadler, Philip. (1992). Report on the programming language Haskell, a non-strict purely functional language (version 1.2). *SIGPLAN notices*, **27**(5).

Klop, J. W., van Oostrom, V., & van Raamsdon, F. 1993 (June). *Combinatory reduction systems: Introduction and survey*. Tech. rept. IR-327. Vrije Universiteit Amsterdam.

Launchbury, John, & Peyton Jones, Simon L. (1995). State in Haskell. *Lisp and Symbolic Computation*, **8**, 193–341.

Launchbury, John, & Sabry, Amr. (1997). Monadic state: Axiomatization and type safety. *ACM SIGPLAN International Conference on Functional Programming.*

Odersky, Martin, Rabin, Dan, & Hudak, Paul. 1993 (Jan.). Call by name, assignment, and the lambda calculus. *Pages 43–56 of: ACM Symposium on Principles of Programming Languages.*

O'Hearn, Peter W. (1995). Note on Algol and conservatively extending functional programming. *Journal of Functional Programming*. To appear.

Peterson, J., *et al.* . (1996). *Report on the programming language Haskell (version 1.3).* Tech. rept. YALEU/DCS/RR-1106. Yale University.

Peyton Jones, Simon L., & Wadler, Philip. (1993). Imperative functional programming. *Pages 71–84 of: ACM Symposium on Principles of Programming Languages.*

Plotkin, Gordon D. (1975). Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science*, **1**, 125–159.

Plotkin, Gordon D. (1977). LCF considered as a programming language. *Theoretical Computer Science*, **5**, 223–255.

Reynolds, John C. (1972). Definitional interpreters for higher-order programming languages. *Pages 717–740 of: Proceedings of the ACM annual conference.*

Reynolds, John C. (1981). The essence of Algol. *Pages 345–372 of:* de Bakker, & van Vliet (eds), *Algorithmic languages.* Amsterdam: North-Holland.

Reynolds, John C. (1988). *Preliminary design of the programming language Forsythe.* Tech. rept. CMU-CS-88-159. Carnegie Mellon University.

Reynolds, John C. (1991). *Replacing complexity with generality: The programming language Forsythe.* Unpublished manuscript, Carnegie Mellon University.

Sabry, Amr, & Field, John. (1993). *Reasoning about explicit and implicit representations of state.* Tech. rept. YALEU/DCS/RR-968. Yale University. ACM SIGPLAN Workshop on State in Programming Languages, pages 17-30.

Søndergaard, H., & Sestoft, P. (1990). Referential transparency, definiteness and unfoldability. *Acta Informatica*, **27**(6), 505–517.

Swarup, V., Reddy, Uday, & Ireland, E. (1991). Assignments for applicative languages. *Pages 192–214 of: Conference on Functional Programming and Computer Architecture.*

Wadler, Philip. (1990). Comprehending monads. *Pages 61–78 of: ACM conference on Lisp and Functional Programming.*

Weeks, Stephen, & Felleisen, Matthias. (1993). On the orthogonality of assignments and procedures in Algol. *Pages 57–70 of: ACM Symposium on Principles of Programming Languages.*