Correctness of Monadic State: An Imperative Call-by-Need Calculus

Zena M. Ariola   Amr Sabry
Department of Computer & Information Science
University of Oregon
Eugene, OR 97403
ariola@cs.uoregon.edu   sabry@cs.uoregon.edu

Abstract

The extension of Haskell with a built-in state monad combines mathematical elegance with operational efficiency:

- Semantically, at the source language level, constructs that act on the state are viewed as functions that pass an explicit store data structure around.
- Operationally, at the implementation level, constructs that act on the state are viewed as statements whose evaluation has the side-effect of updating the implicit global store in place.

There are several unproven conjectures that the two views are consistent.

Recently, we have noted that the consistency of the two views is far from obvious: all it takes for the implementation to become unsound is one judiciously-placed beta-step in the optimization phase of the compiler. This discovery motivates the current paper in which we formalize and show the correctness of the implementation of monadic state.

For the proof, we first design a typed call-by-need language that models the intermediate language of the compiler, together with a type-preserving compilation map. Second, we show that the compilation is semantics-preserving by proving that the compilation of every source axiom yields an observational equivalence of the target language. Because of the wide semantic gap between the source and target languages, we perform this last step using an additional intermediate language.

The imperative call-by-need λ-calculus is of independent interest for reasoning about system-level Haskell code providing services such as memo-functions, generation of new names, *etc.*, and is the starting point for reasoning about the space usage of Haskell programs.

1 Monadic State

The use of monadic state in functional languages provides significant advantages to both programmers and compiler writers. Technically, monadic state isolates the imperative sublanguage from the purely functional sublanguage by making the sequencing of assignments explicit using new term and type constructors. This type-based separation leads to elegant ways of stating and inferring invariants about the code. For example, a standard type system for Haskell, extended with one rule, can track the lifetimes of references and guarantee non-interference among references in several threads [19, 20]. Even better, recent results suggest that such an analysis can be performed in the intermediate language of a type-directed SML compiler [24]. By exposing information to the type system, not only can compiler writers perform sophisticated analyses elegantly, but also they can avoid some analyses altogether. For example, before eliminating common subexpressions, the TIL compiler [32] must guarantee that the subexpressions have no side-effects using either a weak syntactic check or a sophisticated and complicated analysis. In the presence of monadic state, the type system provides the advantages of the sophisticated analysis without its complexity as the types automatically distinguish semantic values from computations.

Semantically speaking, a program using monadic state is equivalent to a functional program in which the store data structure is passed around and partially copied to simulate updates. Clearly, this view does not yield an efficient implementation and negates much of the benefits of using monadic state. For an efficient implementation:

- no code should be generated for sequencing assignments, and
- no code should be generated for passing the store.

At first sight, such an efficient implementation appears easy to realize, and is indeed implemented in the Glasgow Haskell compiler (ghc) [19]. In the intermediate language of the compiler, the monadic combinators are inlined to eliminate the overhead of sequencing assignments, and the store is represented using a global implicit data structure.

Unfortunately, the efficient implementation of monadic state is more subtle than it first appears. We have noted [20, 27] that β-steps are generally unsound in the intermediate language of compilers such as ghc. The unsoundness of β has several consequences. First, it raises questions about the correctness of transformations like λ-lifting and analyses like strictness analysis that are usually based on the denotational semantics of Haskell, which validates β. More significantly, having shown that the intermediate language cannot have a call-by-name semantics, we must find an appropriate semantics. In the absence of a semantics for the intermediate language, there is no hope to reason about intermediate

To appear in: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1998.

programs, to develop semantic-based analyses and optimizations, or to build robust compilers.

In this paper, we develop the semantic infrastructure for the intermediate languages used for implementing monadic state and use it to prove the correctness of the efficient implementation of monadic state. The semantics is based on an *imperative call-by-need* λ-calculus that extends the pure call-by-need λ-calculus [1, 2, 3, 4] with references to model the intermediate languages of Haskell compilers. The imperative call-by-need λ-calculus has the following properties. It is expressed using a set of local program transformations in the style of Felleisen and Hieb's λ_v-S-calculus [12] and Crank's theories for parameter-passing in the λ-calculus [10, 11]. As expected some observational equivalences that held between pure call-by-need terms (*e.g.*, β) are no longer valid in the presence of assignments. However, all source axioms are valid observational equivalences in the imperative call-by-need λ-calculus, which is sufficient to show that the implementation is faithful to the semantics of the source language.

The practical significance of the imperative call-by-need λ-calculus is two-fold. First, it precisely explains, for the first time, the operational characteristics of the interaction between laziness and assignments. Functional programming folklore has often stated that, in contrast with the interaction of call-by-value and assignments, the combination of laziness and assignments is too complicated, if at all possible, to understand operationally. This view is what motivated the use of monads in the first place. Second, the calculus opens the way for theories and analyses of Haskell programs for reasoning about intensional properties such as space usage and space leaks. Previous theories did not take the imperative update optimizations into account, which limits their validity, as the updates affect the space complexity of programs.

The remainder of the paper is organized as follows. We begin by reviewing the syntax, types, and semantics of the source language. In Section 3, we gently design the intermediate language based on both practical and theoretical considerations. Section 4 introduces the formal syntax and semantics of the intermediate language. Section 5 formalizes the compilation map and show that it is consistent with the typing of both the source and intermediate languages. Section 6 shows the correctness of the efficient implementation of monadic state. Section 7 illustrates the use of the calculus for reasoning about imperative call-by-need programs. Finally Sections 8 and 9 review related work, conclusions, and ideas for future work.

2 Source Language

Our source language is modeled on Haskell.

2.1 Terms

The core constructs of the language include those of an applied λ-calculus with mutually recursive definitions. In addition, the language includes a built-in state monad with operations on reference cells.

Definition 1 (Source Syntax) *Let x range over a set of variables and ℓ range over a disjoint set of locations. The set of terms is inductively defined as follows:*

$$
\begin{array}{rrcl}
\textit{Expressions} & M, N, L & ::= & x \mid V \mid M\,N \\
 & & \mid & \mathbf{let}\ x_i = M_i\ \mathbf{in}\ M \\
 & & \mid & \mathbf{sto}\ \{D\}\ M \\
\textit{Values} & V & :: & K \mid \lambda x.M \mid \ell \mid S \\
 & & \mid & M \mathrel{\texttt{>>=}} N \mid \mathsf{ret}\ M \\
\textit{Store Operations} & S & ::= & \mathsf{New}\ M \mid \mathsf{Read}\ M \\
 & & \mid & \mathsf{Write}\ M\ N \\
\textit{Store Bindings} & D & ::= & \ell_i = M_i \\
\textit{Constants} & K & ::= & () \mid n \mid + \mid \ldots
\end{array}
$$

We let K range over an unspecified set of simple constants like numbers and addition. We will also use C to range over contexts with one hole.

The constructs New, Read, and Write implement the usual operations on reference cells. The constructs ret and >>= are the *unit* and *bind* operations of the state monad respectively. Expressions built up from these five operations are state-transformers and hence syntactic values. In the expression ($\mathbf{sto}\ \{\ell_i = M_i\}\ N$), the scope of each location ℓ_i includes all the right-hand sides M_i and N. The evaluation of the expression executes N, returns its value, and discards the final state. The **sto** construct also *encapsulates* the state, in the sense that the state is neither accessible nor visible to the outside world [19, 20].

2.2 Types

Both the monadic extension to the pure language and the encapsulation of state rely critically on types. Stateful computations can only be generated using the term constructors: ret, >>=, New, Read, and Write. Furthermore, their types are built using a special type constructor `ST`. The type of a stateful computation is generally of the form ($\texttt{ST}\ \tau_1\ \tau_2$), where τ_1 is the *index* of the state and τ_2 is the type of the result of the stateful computation. The type ($\texttt{MutVar}\ \tau_1\ \tau_2$) is the type of references allocated from a state indexed with τ_1 and containing values of type τ_2.

Definition 2 (Source Types) *Let α range over a set of type variables. The set of types is inductively defined as follows:*

$$
\begin{array}{rrcl}
\textit{Types} & \tau & ::= & \texttt{Unit} \mid \texttt{Int} \mid \alpha \mid \tau \to \tau \\
 & & \mid & \texttt{ST}\ \tau\ \tau \mid \texttt{MutVar}\ \tau\ \tau \\
\textit{Type Schemes} & \sigma & ::= & \forall \alpha.\sigma \mid \tau
\end{array}
$$

The typing rules are in Figure 1. For the constants, we assume that the type of K is given by τ_K. The reasoning behind the rule for **sto** is as follows. Every operation which manipulates a state thread is infected with the index of that state thread: when >>= is used to combine operations, the indices have to be the same (*i.e.*, they become unified); every location returned by New has the same index as the thread that created it; and every time a Read or Write is performed its `MutVar` argument has the same index as the state thread in which the read or write is performed. Then when a state thread is encapsulated by **sto** the type system will only accept the encapsulation if:

1. the index is still a variable; *and*
2. that variable is universally quantifiable.

If these two conditions hold then the state thread makes no demands on its environment to provide, say, a location to be read or written. If it did, the index would have been unified with the index of the location in the environment, and

$$\overline{\Gamma \cup \{x : \forall \alpha_i.\tau\} \vdash x : \tau[\alpha_i := \tau_i]}$$

$$\overline{\Gamma \vdash K : \tau_K}$$

$$\overline{\Gamma \cup \{\ell : \tau\} \vdash \ell : \tau}$$

$$\frac{\Gamma \vdash M : \tau' \rightarrow \tau \qquad \Gamma \vdash N : \tau'}{\Gamma \vdash M\ N : \tau}$$

$$\frac{\Gamma \cup \{x : \tau\} \vdash M : \tau'}{\Gamma \vdash \lambda x.M : \tau \rightarrow \tau'}$$

$$\frac{\begin{array}{rcl}\forall j.\ \Gamma \cup \{x_i : \tau_i\} & \vdash & M_j : \tau_j \\ \Gamma \cup \{x_i : \forall \alpha_{j_i}.\tau_i\} & \vdash & M : \tau\end{array}}{\Gamma \vdash \mathbf{let}\ x_i = M_i\ \mathbf{in}\ M : \tau}$$

where $\alpha_{j_i} \in FV(\tau_i) \backslash FV(\Gamma)$

$$\frac{\begin{array}{rcl}\forall j.\ \Gamma \cup \{\ell_i : \texttt{MutVar}\ \alpha\ \tau_i\} & \vdash & M_j : \tau_j \\ \Gamma \cup \{\ell_i : \texttt{MutVar}\ \alpha\ \tau_i\} & \vdash & M : \texttt{ST}\ \alpha\ \tau\end{array}}{\Gamma \vdash \mathbf{sto}\ \{\ell_i = M_i\}\ M : \tau}$$

where $\alpha \notin FV(\tau, \Gamma)$

$$\frac{\Gamma \vdash M : \texttt{ST}\ \tau'\ \tau_2 \qquad \Gamma \vdash N : \tau_2 \rightarrow \texttt{ST}\ \tau'\ \tau_1}{\Gamma \vdash M \texttt{ >>= } N : \texttt{ST}\ \tau'\ \tau_1}$$

$$\frac{\Gamma \vdash M : \tau}{\Gamma \vdash \mathsf{ret}\ M : \texttt{ST}\ \tau'\ \tau}$$

$$\frac{\Gamma \vdash M : \tau}{\Gamma \vdash \mathsf{New}\ M : \texttt{ST}\ \tau'\ (\texttt{MutVar}\ \tau'\ \tau)}$$

$$\frac{\Gamma \vdash M : \texttt{MutVar}\ \tau'\ \tau}{\Gamma \vdash \mathsf{Read}\ M : \texttt{ST}\ \tau'\ \tau}$$

$$\frac{\Gamma \vdash M : \texttt{MutVar}\ \tau'\ \tau \qquad \Gamma \vdash N : \tau}{\Gamma \vdash \mathsf{Write}\ M\ N : \texttt{ST}\ \tau'\ \texttt{Unit}}$$

Figure 1: Typing Rules for Source Language

universal quantification could not take place. This intuitive argument can be made precise to provide a formal proof of the type-based encapsulation of monadic state [20].

2.3 Evaluation Contexts

The semantics of the source language can be conveniently specified using a set of reductions. Because of the lazy semantics and lazy store updates [7, 13, 17], the definition of the reductions is intertwined with the definition of evaluation contexts. Therefore, we define evaluation contexts first. These contexts use two auxiliary concepts: return contexts R and dependency chains H.

Definition 3 (Return Contexts R**)** *The contexts are inductively defined as:*

$$R \quad ::= \quad [\,] \mid M \texttt{ >>= } \lambda x.R$$

Return contexts define the position within a state thread from which it is possible to immediately return without performing the rest of the stateful computation. Indeed, none of the computations to the left of >>= is performed unless they are explicitly demanded. Demands occur according to the following definition. The definition uses the yet-to-be-defined evaluation contexts E. At this point the reader may pretend that all evaluation contexts are the empty context to get the intuition behind the concept of dependencies.

Definition 4 (Dependencies H**)** *Chains of dependencies are defined as follows:*

$$\begin{array}{rcl} H & ::= & \lambda x.R[E[x]] \\ & & \mid \lambda x.R[\mathsf{ret}\ E[x]] \\ & & \mid \lambda x.R[M \texttt{ >>= } E[x]] \\ & & \mid \lambda x.R[E[x] \texttt{ >>= } H] \\ & & \mid \lambda x.R[S \texttt{ >>= } H] \end{array}$$

In the first three clauses, the value of x is needed to proceed with the evaluation of the state thread. In the next two clauses, a demand for a stateful computation is propagated to the beginning of the thread. The last clause shows that the operations New, Read, and Write are strict in the state, which demands all previous computations in the thread.

Definition 5 (Evaluation Contexts E**)** *The set of contexts is inductively defined as:*

$$\begin{array}{rcl} E & ::= & [\,] \mid E\ M \mid K\ E \\ & & \mid \mathbf{sto}\ \{D\}\ R[E] \\ & & \mid \mathbf{sto}\ \{D\}\ R[\mathsf{ret}\ E] \\ & & \mid \mathbf{sto}\ \{D\}\ R[M \texttt{ >>= } E] \\ & & \mid \mathbf{sto}\ \{D\}\ R[E \texttt{ >>= } H] \\ & & \mid \mathbf{sto}\ \{D\}\ (\mathsf{Read}\ E \texttt{ >>= } H) \\ & & \mid \mathbf{sto}\ \{D\}\ (\mathsf{Write}\ E\ M \texttt{ >>= } H) \end{array}$$

The first three clauses in the definition of evaluation contexts define the usual contexts for call-by-name languages. The remaining contexts are used when evaluating a stateful computation. The next three contexts combined keep demanding the right argument of >>= until they reach the last state transformer in an R-sequence. If that state transformer is a ret then we demand the value of its subexpression. If on the other hand, the last state transformer demands a variable, then we backtrack following the previously defined chains of dependencies demanding state transformers on the left of >>=. Finally the operations Read and Write are strict in their first argument which is the location to read or write.

2.4 Semantics

Figure 2 presents an axiomatization of the semantics [20]. The first three rules are as expected in an applied λ-calculus. In particular, the semantics of function application is a call-by-name one that admits full β. In the next three rules, each primitive store operation performs its intended operation on the properly initialized store fragment. The structural rules correspond to the three monad laws. Finally the return rules show how to compute the result of a state thread; there is a rule for each kind of syntactic value.

3 Design Decisions for the Intermediate Language

The intermediate language ended up being rather complex, so we spend some time explaining the major decisions involved in its design. Most of the design ideas build upon the ghc implementation. Interestingly, the design can be motivated by technical problems in attempted proofs. For example, a broken consistency theorem indicated that the

Computational Reductions:

$$
\begin{array}{rcl}
(\lambda x.M)N & \rightarrow & M[x := N] \\
\textbf{let } x_i = M_i \textbf{ in } M & \rightarrow & M[x_j := (\textbf{let } x_i = M_i \textbf{ in } M_j)] \\
K\ V & \rightarrow & \delta(K,V) \quad \text{if defined} \\
\textbf{sto } \{D\}\ (\textsf{New } M \texttt{ >>= } H) & \rightarrow & \textbf{sto } \{D, \ell = M\}\ (H\ \ell) \\
\textbf{sto } \{D, \ell = M\}\ (\textsf{Read } \ell \texttt{ >>= } H) & \rightarrow & \textbf{sto } \{D, \ell = M\}\ (H\ M) \\
\textbf{sto } \{D, \ell = M\}\ (\textsf{Write } \ell\ N \texttt{ >>= } H) & \rightarrow & \textbf{sto } \{D, \ell = N\}\ (H\ ())
\end{array}
$$

Structural Reductions:

$$
\begin{array}{rcl}
\textbf{sto } \{D\}\ R[\textsf{ret } M \texttt{ >>= } H] & \rightarrow & \textbf{sto } \{D\}\ R[H\ M] \\
\textbf{sto } \{D\}\ R[(M \texttt{ >>= } N) \texttt{ >>= } H] & \rightarrow & \textbf{sto } \{D\}\ R[M \texttt{ >>= } \lambda x.((N\ x) \texttt{ >>= } H)] \\
\textbf{sto } \{D\}\ R[S] & \rightarrow & \textbf{sto } \{D\}\ R[S \texttt{ >>= } \lambda x.\textsf{ret } x]
\end{array}
$$

Return Reductions:

$$
\begin{array}{rcl}
\textbf{sto } \{D\}\ R[\textsf{ret } K] & \rightarrow & K \\
\textbf{sto } \{D\}\ R[\textsf{ret } (\lambda y.M)] & \rightarrow & \lambda y.\textbf{sto } \{D\}\ R[\textsf{ret } M] \\
\textbf{sto } \{D\}\ R[\textsf{ret } (M \texttt{ >>= } N)] & \rightarrow & (\textbf{sto } \{D\}\ R[\textsf{ret } M]) \texttt{ >>= } (\textbf{sto } \{D\}\ R[\textsf{ret } N]) \\
\textbf{sto } \{D\}\ R[\textsf{ret } (\textsf{ret } M)] & \rightarrow & \textsf{ret } (\textbf{sto } \{D\}\ R[\textsf{ret } M]) \\
\textbf{sto } \{D\}\ R[\textsf{ret } (\textsf{New } M)] & \rightarrow & \textsf{New } (\textbf{sto } \{D\}\ R[\textsf{ret } M]) \\
\textbf{sto } \{D\}\ R[\textsf{ret } (\textsf{Read } M)] & \rightarrow & \textsf{Read } (\textbf{sto } \{D\}\ R[\textsf{ret } M]) \\
\textbf{sto } \{D\}\ R[\textsf{ret } (\textsf{Write } M\ N)] & \rightarrow & \textsf{Write } (\textbf{sto } \{D\}\ R[\textsf{ret } M])\ (\textbf{sto } \{D\}\ R[\textsf{ret } N]) \\
\textbf{sto } \{D\}\ R[\textsf{ret } \ell] & \rightarrow & \ell \quad \text{if } \ell \notin dom(D)
\end{array}
$$

Figure 2: Call-by-Name Semantics of Source Language

parameter-passing mechanism should be call-by-need rather than call-by-name, etc. Furthermore, the intuitive development here is paralleled by the structure of the proof in Section 6.

3.1 Basic Compilation Scheme

The starting point in the definition of the intermediate language is the compilation strategy adopted by ghc. The first phase of compilation closely follows the denotational semantics of the source language [19]: it reformulates the program in explicit store-passing style. For example, consider the following code fragment:

$$
\begin{array}{l}
\textbf{sto } \{\}\ (\ \textsf{New } 0 \texttt{ >>= } \lambda p. \\
\qquad\quad \textsf{Write } p\ 5 \texttt{ >>= } \lambda\_. \\
\qquad\quad \textsf{Read } p \texttt{ >>= } \lambda v. \\
\qquad\quad \textsf{ret } v)
\end{array}
$$

Intuitively, the evaluation of the term allocates a new location p, initializes p to 0, updates p with the value 5, binds v to the contents of p, discards the state, and returns v. Clearly the result should be 5. Applying the store-passing translation and simplifying the output for readability, we get:

$$
\begin{array}{l}
\textsf{fst } (\ \textbf{let}\ \ (p, s_1) = \textsf{new } \Box\ 0 \\
\qquad\qquad\quad (\_, s_2) = \textsf{write } s_1\ p\ 5 \\
\qquad\qquad\quad (v, s_3) = \textsf{read } s_2\ p \\
\qquad\quad \textbf{in } (v, s_3))
\end{array}
$$

where $\Box$ is the initial empty store. Next, the compiler eliminates the syntactic sugar associated with pattern-matching which produces the term P:

$$
\begin{array}{l}
P \ \equiv \textsf{fst } (\ \textbf{let}\ \ p_1 = \textsf{new } \Box\ 0 \\
\qquad\qquad\qquad\quad p = \textsf{fst } p_1 \\
\qquad\qquad\qquad\quad s_1 = \textsf{snd } p_1 \\
\qquad\qquad\qquad\quad p_2 = \textsf{write } s_1\ p\ 5 \\
\qquad\qquad\qquad\quad s_2 = \textsf{snd } p_2 \\
\qquad\qquad\qquad\quad p_3 = \textsf{read } s_2\ p \\
\qquad\qquad\qquad\quad v = \textsf{fst } p_3 \\
\qquad\qquad\qquad\quad s_3 = \textsf{snd } p_3 \\
\qquad\qquad\quad \textbf{in } (v, s_3))
\end{array}
$$

3.2 Explicit Store vs. Implicit Store

As apparent from above, each of the operations new, read, and write, in the intermediate language takes a store as an argument and returns a store as part of its result. These stores play two independent roles:

1. Modeling the Heap: Semantically stores are finite functions (tables) mapping locations to their contents, and updates are simulated by creating new stores (copying parts of the table in the process).

2. Sequencing Imperative Operations: The explicit store-passing style defines the relative order of imperative operations via the data dependencies among stores. For example, the operation (write s_1 p 5) must happen after the operation that produces store s_1 as its result because write is strict in its store argument.

Taken literally, the above points imply a rather inefficient implementation. To achieve an efficient implementation, we proceed in two steps. First we model the heap with *one* global implicit store. This means that the operations new, read, and write should be interpreted as performing destructive updates on the global store rather than operating on their store arguments. The store arguments are no longer relevant for the implementation of assignments but are maintained to prevent program transformations from accidentally reordering imperative operations.

After all optimizations have been performed, the back end arranges to generate no code at all to pass around the store arguments. If the intermediate language is explicitly typed, then the code generator can easily distinguish arguments representing the store from other arguments. Let $\diamondsuit$ be the special type of stores: an expression like

$(\lambda s_2.\mathsf{read}\ s_2\ p)$ of type $(\Diamond \to \mathtt{Int} \times \Diamond)$ should be compiled as a statement producing an Int as a side-effect and not as a function. This discussion thus motivates the importance of types for the intermediate language.

3.3 Call-by-Name vs. Call-by-Need

The shift of perspective from explicit stores with functional updates to implicit stores with destructive updates appears like a reasonable optimization but it is not evidently correct. In fact, the correctness turns out to be a rather subtle issue. Let's assume that the semantics of the intermediate language is the usual call-by-name semantics for Haskell that validates full β. Now consider the effect of a sequence of β-reductions on the term P from Subsection 3.1:

$$
\begin{array}{l}
\mathsf{fst}\ (\ \mathbf{let}\ p_3 = \mathsf{read}\ (\mathsf{snd}\ (\mathsf{write}\ (\mathsf{snd}\ (\mathsf{new}\ \Box\ 0)) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathsf{fst}\ (\mathsf{new}\ \Box\ 0)) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad 5)) \\
\qquad\qquad\qquad\qquad\quad (\mathsf{fst}\ (\mathsf{new}\ \Box\ 0)) \\
\qquad\quad v = \mathsf{fst}\ p_3 \\
\qquad\quad s_3 = \mathsf{snd}\ p_3 \\
\qquad \mathbf{in}\ (v, s_3))
\end{array}
$$

By inspection of the code, the number 5 only occurs in the first argument to read. Hence if read evaluates the store argument for effect (but does not use the result), the code can never evaluate to 5. This error is due to the duplication of expressions like $(\mathsf{new}\ \Box\ 0)$. Each occurrence of new ignores its store argument, and returns a new *distinct* location relative to the global store. Compare this behavior to the one dictated by the semantics where new is a pure *function* and every occurrence of $(\mathsf{new}\ \Box\ 0)$ returns the same location.

In conclusion, β-steps in particular, and call-by-name reasoning principles in general are incompatible with the implementation strategy described in Subsection 3.2. We were the first to note this problem and have conjectured [20, 27] that optimizations based on call-by-need semantics would be sound in the intermediate language of the compiler. Intuitively, call-by-need transformations would not duplicate non-values such as $(\mathsf{new}\ \Box\ 0)$. Fortunately, even before the monadic extensions, compilers such as ghc were careful not to duplicate work and hence refrained from using β steps for performance reasons [25]. Consequently, the addition of assignments to the back end did not cause any immediate problems. However, this non-duplication is not a mere optimization issue, but an essential correctness issue.

3.4 Demanding Values vs. Effects

Having decided on a call-by-need semantics for the intermediate language, we must resolve the following dilemma. By definition the call-by-need mechanism only evaluates an expression if its *value* is needed. How then are we to evaluate expressions such as write for effect only?

There are several ways to encode this additional demand for effects. For example, by adding a special combinator seq, a strict-**let**, or writing in continuation-passing style, all of which involve a change in the language or compilation scheme. But fortunately we already have a mechanism in place based on the store arguments that are passed around. Since all imperative operations are strict in their store arguments, the result of an expression such as write (which includes a store) is demanded by future imperative operations. However, since the actual value of the store argument is irrelevant, it is sufficient to have just one value of type $\Diamond$ that we name $\Box$. When a state thread is created, it is provided with the value $\Box$. Operations in the thread will only be performed when they receive the "token" $\Box$ and then pass it to the remaining operations. Again the code generator does not actually generate any code to move the value $\Box$.

3.5 Boxed vs. Unboxed Types

Ultimately the code generator treats the type $\Diamond$ specially. This interacts poorly with polymorphism as the following example illustrates. Consider the source fragment:

$$
\begin{array}{l}
\mathbf{let}\ f = \lambda a.\lambda b.(a, b) \\
\mathbf{in}\ \mathsf{ret}\ (f\ 1\ 2)
\end{array}
$$

After translation to the intermediate language and some processing, we get:

$$
\begin{array}{l}
\mathbf{let}\ f = \lambda a.\lambda b.(a, b) \\
\mathbf{in}\ (\lambda v.\lambda s.(v, s))\ (f\ 1\ 2)
\end{array}
$$

Without paying attention to types, a compiler might recognize and lift the common subexpression producing:

$$
\begin{array}{l}
\mathbf{let}\ f = \lambda a.\lambda b.(a, b) \\
\mathbf{in}\ f\ (f\ 1\ 2)
\end{array}
$$

But then, what code should the compiler generate for the polymorphic function f? Both instances of f cannot be compiled in the same way. The inner instance should clearly take two arguments and build a pair data structure. The outer instance should only take *one* argument; the second being the token representing the store that is ignored at code generation time.

Technically, the compiler cannot generate code for a polymorphic function of type $(\alpha \to \beta \times \alpha)$ unless it knows all the uses of the functions: one of them might have $\alpha = \Diamond$ which would require different code generation routines. Another way of looking at the problem is that a function of type $(\alpha \to \beta \times \alpha)$ cannot be uniformly used at types $(\mathtt{Int} \to \mathtt{Int} \times \mathtt{Int})$ and $(\Diamond \to \mathtt{Int} \times \Diamond)$. The solution to the problem is to treat the type $\Diamond$ as an *unboxed* type, and to restrict type variables not to range over unboxed types [23].

3.6 Recursion

Since the use of assignments can easily create cyclic structures, a proper handling of recursion is a prerequisite to handling state. The source language has two constructs for recursion: **let** and **sto**. We here show that the intermediate language must express these two notions of recursion with one construct. Consider the source term:

$$\mathbf{sto}\ \{\ell = \lambda\_.0\}\ \mathbf{let}\ x = \lambda\_.\mathsf{Read}\ \ell\ \mathbf{in}\ \mathsf{Write}\ \ell\ (\lambda\_.((\lambda\_.5)\ x))$$

Irrespective of how we translate **let** and **sto** we will eventually reach a point, after performing the assignment, where x refers to ℓ and ℓ refers to x. So we must have one construct in which recursion via variables and locations can be simultaneously expressed. This behavior does not happen in the source language because we have full β which enables us to eliminate **let**-expressions by substituting the right-hand sides of the definitions in the body (See Figure 2).

4 Imperative Call-by-Need Calculus

Building on our previous discussion, we formalize the precise syntax and semantics of the intermediate language.

4.1 Terms

The terms of the intermediate language include pairs, simple constants, (destructive) operations on reference cells, and one construct $\langle . \mid . \rangle$, pronounced *box*, for expressing recursion via both variables and locations. In contrast to the syntax of the source language where we used capital letters to range over the meta-variables, we here use lower case letters.

Definition 6 (Target Terms) *Let x range over a set of variables, ℓ range over a disjoint set of locations. The set of terms is inductively defined as follows:*

$$\begin{array}{rrcl}
\textit{Expressions} & e & ::= & x \mid v \mid e_1 e_2 \mid \langle e \mid d \rangle \\
 & & \mid & (e_1, e_2) \mid \mathsf{fst}\ e \mid \mathsf{snd}\ e \\
 & & \mid & \mathsf{new}\ e_1\ e_2 \mid \mathsf{read}\ e_1\ e_2 \\
 & & \mid & \mathsf{write}\ e_1\ e_2\ e_3 \\
\textit{Values} & v & ::= & k \mid \lambda x.e \mid \ell \mid [e_1, e_2] \mid \Box \\
\textit{Bindings} & d & ::= & \vartheta_i = e_i \\
 & \vartheta & ::= & x \mid \ell \\
\textit{Constants} & k & ::= & () \mid n \mid + \mid \ldots
\end{array}$$

As explained in the previous section, the value $\Box$ is the token representing the store. The imperative operations new, read, and write, all expect this token as their first argument. Following the design of the applied call-by-need λ-calculus [1, 2, 3, 4], we distinguish two kinds of pairs. In contrast to (e_1, e_2), pairs of the form $[e_1, e_2]$ are values that can be duplicated. This distinction avoids the duplication of the components of a pair, which is essential to capture the call-by-need semantics.

4.2 Types

The types of the intermediate language are the expected ones: they are built using constructors for primitives, functions, products, and references.

Definition 7 (Target Types) *Let α range over a set of type variables. The set of types is inductively defined as follows:*

$$\begin{array}{rrcl}
\textit{Types} & \tau & ::= & B \mid U \\
\textit{Boxed Types} & B & ::= & \mathtt{Unit} \mid \mathtt{Int} \mid \alpha \mid \tau \rightarrow \tau \\
 & & \mid & \mathtt{Ref}\ \tau \mid \tau \times \diamondsuit \\
\textit{Unboxed Types} & U & ::= & \diamondsuit \\
\textit{Type Schemes} & \sigma & ::= & \forall \alpha.\sigma \mid \tau
\end{array}$$

As implied by the grammar, type variables range only over boxed types. This is necessary because of the interaction of types treated specially by the code generator and polymorphism (see Section 3.5). Modern compilers have optimized unboxed representations of datatypes like integers; we do not handle such an extension in our language. Also, we have restricted the second component of pairs to be of the special store type $\diamondsuit$. This is not necessary but simplifies the presentation and proofs.

Most importantly, the type of references no longer includes a state index. It is conceivable to maintain the state indices in the types of the intermediate language, but since our goal is to prove the correctness of an existing implementation strategy, we stick to the actual ghc language. This decision to "forget" some type information during compilation will have a major impact when we study the correctness of the compiler in Section 6.

The typing rules for the intermediate language are in Figure 3. They should be of no surprise.

$$\frac{}{\Gamma \cup \{x : \forall \alpha_i.\tau\} \vdash x : \tau[\alpha_i := \tau_i]}$$

$$\frac{\Gamma \vdash e_1 : \tau' \rightarrow \tau \qquad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1\ e_2 : \tau}$$

$$\frac{\begin{array}{rcl}\forall j.\ \Gamma \cup \{x_i : \tau_i,\ \ell_i : \mathtt{Ref}\ \tau_i\} & \vdash & e_j : \tau_j \\ \Gamma \cup \{x_i : \forall \alpha_{j_i}.\tau_i,\ \ell_i : \mathtt{Ref}\ \tau_i\} & \vdash & e : \tau\end{array}}{\Gamma \vdash \langle e \mid \vartheta_i = e_i \rangle : \tau}$$

where $\alpha_{j_i} \in FV(\tau_i) \backslash FV(\Gamma)$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \diamondsuit}{\Gamma \vdash (e_1, e_2) : \tau \times \diamondsuit}$$

$$\frac{\Gamma \vdash e : \tau \times \diamondsuit}{\Gamma \vdash \mathsf{fst}\ e : \tau}$$

$$\frac{\Gamma \vdash e : \tau \times \diamondsuit}{\Gamma \vdash \mathsf{snd}\ e : \diamondsuit}$$

$$\frac{\Gamma \vdash e_1 : \diamondsuit \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathsf{new}\ e_1\ e_2 : \mathtt{Ref}\ \tau \times \diamondsuit}$$

$$\frac{\Gamma \vdash e_1 : \diamondsuit \qquad \Gamma \vdash e_2 : \mathtt{Ref}\ \tau}{\Gamma \vdash \mathsf{read}\ e_1\ e_2 : \tau \times \diamondsuit}$$

$$\frac{\Gamma \vdash e_1 : \diamondsuit \qquad \Gamma \vdash e_2 : \mathtt{Ref}\ \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \mathsf{write}\ e_1\ e_2\ e_3 : \mathtt{Unit} \times \diamondsuit}$$

$$\frac{}{\Gamma \vdash k : \tau_k}$$

$$\frac{}{\Gamma \cup \{\ell : \tau\} \vdash \ell : \tau}$$

$$\frac{\Gamma \cup \{x : \tau\} \vdash e : \tau'}{\Gamma \vdash \lambda x.e : \tau \rightarrow \tau'}$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \diamondsuit}{\Gamma \vdash [e_1, e_2] : \tau \times \diamondsuit}$$

$$\frac{}{\Gamma \vdash \Box : \diamondsuit}$$

Figure 3: Typing Rules for Intermediate Language

4.3 Semantics I: Call-by-Need with Recursion

We begin with the semantics of the subset of the language excluding assignments [1, 2]. To reduce the number of reductions we use evaluation contexts to abstract some common patterns.

Definition 8 (Evaluation Contexts E) *The set of pure evaluation contexts is inductively defined as follows:*

$$E \quad ::= \quad [\,] \mid Ee \mid \mathsf{fst}\ E \mid \mathsf{snd}\ E \mid kE$$

The reductions for the pure language are in Figure 4. They are grouped in three categories. The first category includes reductions on pure lambda graphs that implement call-by-need applications. An argument is added to the list of declarations when first encountered. The argument can only be copied once it becomes a value. Next, we have administrative reductions that re-arrange bindings. All the reductions enlarge the scope of the declarations to include

Reductions for Lambda Graphs:

$$\begin{array}{rcl}
(\lambda x.e_1)\ e_2 & \rightarrow & \langle e_1 \mid x = e_2 \rangle \\
\langle C[x] \mid x = v, d \rangle & \rightarrow & \langle C[v] \mid x = v, d \rangle \\
\langle e \mid y = C[x], x = v, d \rangle & \rightarrow & \langle e \mid y = C[v], x = v, d \rangle \\
k\ v & \rightarrow & \delta(k, v) \qquad \text{if defined}
\end{array}$$

Administrative reductions:

$$\begin{array}{rcl}
E[\langle e \mid d \rangle] & \rightarrow & \langle E[e] \mid d \rangle \\
\langle \langle e \mid d_1 \rangle \mid d_2 \rangle & \rightarrow & \langle e \mid d_1, d_2 \rangle \\
\langle E[\overline{\vartheta}] \mid var(\overline{\vartheta}) = E[\overline{\vartheta}_1], \ldots, var(\overline{\vartheta}_n) = \langle e \mid d_1 \rangle, d_2 \rangle & \rightarrow & \langle E[\overline{\vartheta}] \mid var(\overline{\vartheta}) = E[\overline{\vartheta}_1], \ldots, var(\overline{\vartheta}_n) = e, d_1, d_2 \rangle
\end{array}$$

Reductions for Pairs:

$$\begin{array}{rcl}
(e_1, e_2) & \rightarrow & \langle [x, y] \mid x = e_1, y = e_2 \rangle \\
\mathsf{fst}\ [e_1, e_2] & \rightarrow & e_1 \\
\mathsf{snd}\ [e_1, e_2] & \rightarrow & e_2
\end{array}$$

Figure 4: Reductions for Pure Recursive Call-by-Need Language

$$\begin{array}{rcl}
\mathsf{new}\ \Box\ e & \rightarrow & \langle [\ell, \Box] \mid \ell = e \rangle \\
\langle E[\mathsf{read}\ \Box\ \ell] \mid \ell = v, d \rangle & \rightarrow & \langle E[[v, \Box]] \mid \ell = v, d \rangle \\
\langle E[\mathsf{write}\ \Box\ \ell\ e_1] \mid \ell = e_2, d \rangle & \rightarrow & \langle E[[(), \Box]] \mid \ell = e_1, d \rangle
\end{array}$$

$$\langle E[\overline{\vartheta}] \mid var(\overline{\vartheta}) = E_1[\overline{\vartheta}_1], \ldots, var(\overline{\vartheta}_n) = E_n[\mathsf{read}\ \Box\ \ell], \ell = v, d \rangle$$
$$\rightarrow \langle E[\overline{\vartheta}] \mid var(\overline{\vartheta}) = E_1[\overline{\vartheta}_1], \ldots, var(\overline{\vartheta}_n) = E_n[[v, \Box]], \ell = v, d \rangle$$

$$\langle E[\overline{\vartheta}] \mid var(\overline{\vartheta}) = E_1[\overline{\vartheta}_1], \ldots, var(\overline{\vartheta}_n) = E_n[\mathsf{write}\ \Box\ \ell\ e_1], \ell = e_2, d \rangle$$
$$\rightarrow \langle E[\overline{\vartheta}] \mid var(\overline{\vartheta}) = E_1[\overline{\vartheta}_1], \ldots, var(\overline{\vartheta}_n) = E_n[[(), \Box]], \ell = e_1, d \rangle$$

Figure 5: Reductions for Reference Cells

the enclosing context. Note that in the third rule, the internal box is only flattened when is needed. We will discuss the reasons for this restriction after having introduced the operations on reference cells. Finally, we have the reductions on pairs, which rewrite regular pairs to special pairs $[.,.]$ that can be treated as values and safely duplicated.

4.4 Semantics II: Assignments

To axiomatize the semantics of the full language including assignments, we extend the sets of evaluation contexts and reductions.

Definition 9 (E) *The set of evaluation contexts extends the one in Definition 8 as follows:*

$$\begin{array}{rcl}
E & ::= & \ldots \mid \mathsf{new}\ E\ e \mid \mathsf{read}\ E\ e \mid \mathsf{read}\ v\ E \\
& \mid & \mathsf{write}\ E\ e_1\ e_2 \mid \mathsf{write}\ v\ E\ e
\end{array}$$

In other words, the imperative operations are strict in their store argument (the token $\Box$). Also read and write are strict in the location to be accessed.

In contrast to the other operations, the operations involving reference cells are partially ordered using evaluation contexts. For example, a read operation is only performed when it occurs in the hole of an evaluation context relative to the location being read. Furthermore, the contents of the location must have already been evaluated before the read can occur. The situation with write is similar but slightly more complicated. An expression ($\mathsf{write}\ \Box\ \ell\ e$) must also occur in the hole of an evaluation context relative to ℓ, but it does not demand the evaluation of the contents of ℓ. Indeed, the old content is about to be overwritten, and hence not needed. In summary, not every expression of the form $E[\ell]$ demands the evaluation of the contents of ℓ. This observation motivates the following definition.

Definition 10 ($\overline{\vartheta}$) *The set of demanding expressions is:*

$$\overline{\vartheta} \quad ::= \quad x \mid \mathsf{read}\ \Box\ \ell$$

The function $var(.)$ maps a demanding expression to the variable or location it demands:

$$\begin{array}{rcl}
var(x) & = & x \\
var(\mathsf{read}\ \Box\ \ell) & = & \ell
\end{array}$$

The complete set of reductions includes the reductions in Figure 4 (with the new definitions of evaluation contexts) in addition to the assignment-specific reductions in Figure 5. In the reductions for reference cells, the store variable is only used for sequencing the evaluation of operations and not for actually modeling the heap.

We can now explain why we need to impose some restrictions on the flattening of internal boxes. A non demanded flattening could disallow a reference operation that would otherwise be possible, as shown in the diagram below:

$$\begin{tikzcd}
\langle \lambda x.y \mid y = \langle \mathsf{read}\ \Box\ \ell \mid \ell = 0 \rangle \rangle \arrow[r, twoheadrightarrow] \arrow[d] & \langle \lambda x.y \mid y = \mathsf{read}\ \Box\ \ell, \ell = 0 \rangle \arrow[d, dashed] \\
\langle \lambda x.y \mid y = [0, \Box], \ell = 0 \rangle \arrow[r, dashed] & ?
\end{tikzcd}$$

The flattening is necessary to allow reductions of the form:

$$\begin{array}{l}
\langle y \mid y = \langle \mathsf{read}\ \Box\ \ell \mid d \rangle, \ell = v \rangle \twoheadrightarrow \\
\langle y \mid y = [v, \Box], d, \ell = v \rangle
\end{array}$$

At this point, the only thing that we require is that the calculus be consistent. Unfortunately, this calculus turns out to be non-confluent [5]. The culprit is the substitution inside the bindings of a box, which is essential in a call-by-need setting. One solution is to restrict this rule to evaluation contexts [3]. That is, we could replace the third lambda graph rule with the rule:

$$\begin{aligned}&\langle E[x_1] \mid x_1 = E[x_2], \ldots, x_n = E[x], x = v, d\rangle \\ &\qquad \to \langle E[x_1] \mid x_1 = E[x_2], \ldots, x_n = E[v], x = v, d\rangle\end{aligned}$$

This restores confluence at the expense of considerably complicating the proof of correctness. Therefore, we prefer to prove consistency by showing unicity of infinite normal forms, also called Böhm trees [21], which are defined as the limit of the chain of observations collected during reduction.

Theorem 11 *The imperative call-by-need lambda-calculus has unique infinite normal forms.*

Proof. The result for the pure part of the calculus is shown in [2]. Since the reference rules commute with the other rules, the result for the full calculus constitutes an easy extension of the previous result. □

5 Compilation

Having set up the source and target languages, we formalize the compilation map. The map has two components: one acting on types and one acting on terms.

Definition 12 *The compilation of types is the following:*

$$\begin{aligned}\texttt{Unit}^* &= \texttt{Unit}\\ \texttt{Int}^* &= \texttt{Int}\\ \alpha^* &= \alpha\\ (\tau_1 \to \tau_2)^* &= {\tau_1}^* \to {\tau_2}^*\\ (\texttt{ST}\ \tau_1\ \tau_2)^* &= \Diamond \to {\tau_2}^* \times \Diamond\\ (\texttt{MutVar}\ \tau_1\ \tau_2)^* &= \texttt{Ref}\ {\tau_2}^*\\ (\forall\alpha.\sigma)^* &= \forall\alpha.\sigma^*\end{aligned}$$

We will use Γ^* to denote the type environment Γ where all the types in the right-hand sides have been translated. As explained in Section 4.2, the compilation map forgets the information about state indices.

The translation on terms is in Figure 6. It is based on a classical store-passing translation but does not actually pass the store around. Instead it passes a token □. Otherwise, the translation is fairly standard except that both recursive constructs in the source language are translated using the same intermediate form.

The translation commutes with substitution.

Lemma 13 (Substitution) *Let M and N be source terms: $M^*[x := N^*] = (M[x := N])^*$.*

More interestingly, the compilation preserves the types of terms. Thus, if a term is typable in the source language, its translation is typable with an equivalent type in the intermediate language. This result holds if we impose the following constraint: all constants K must have types τ_K such that ${\tau_K}^* = \tau_k$. For example, constants of type `Int`, (`Int` $\to$ `Int`) would be acceptable but constants of type (`ST Int Int`) would not.

$$\frac{}{x \hookrightarrow x}$$

$$\frac{M \hookrightarrow M^* \qquad N \hookrightarrow N^*}{M\ N \hookrightarrow M^*\ N^*}$$

$$\frac{M \hookrightarrow M^* \qquad \forall i. M_i \hookrightarrow M_i^*}{\mathbf{let}\ x_i = M_i\ \mathbf{in}\ M \hookrightarrow \langle M^* \mid x_i = M_i^*\rangle}$$

$$\frac{M \hookrightarrow M^* \qquad \forall i. M_i \hookrightarrow M_i^*}{\mathbf{sto}\ \{\ell_i = M_i\}\ M \hookrightarrow \langle \mathsf{fst}\ (M^*\ \square) \mid \ell_i = M_i^*\rangle}$$

$$\frac{}{K \hookrightarrow K}$$

$$\frac{}{\ell \hookrightarrow \ell}$$

$$\frac{M \hookrightarrow M^*}{\lambda x.M \hookrightarrow \lambda x.M^*}$$

$$\frac{M \hookrightarrow M^* \qquad N \hookrightarrow N^*}{M \mathbin{\texttt{>>=}} N \hookrightarrow \lambda x.\langle N^*\ (\mathsf{fst}\ p)\ (\mathsf{snd}\ p) \mid p = M^*\ x\rangle}$$

$$\frac{M \hookrightarrow M^*}{\mathsf{ret}\ M \hookrightarrow \lambda x.(M^*, x)}$$

$$\frac{M \hookrightarrow M^*}{\mathsf{New}\ M \hookrightarrow \lambda x.\mathsf{new}\ x\ M^*}$$

$$\frac{M \hookrightarrow M^*}{\mathsf{Read}\ M \hookrightarrow \lambda x.\mathsf{read}\ x\ M^*}$$

$$\frac{M \hookrightarrow M^* \qquad N \hookrightarrow N^*}{\mathsf{Write}\ M\ N \hookrightarrow \lambda x.\mathsf{write}\ x\ M^*\ N^*}$$

Figure 6: Compiling Terms

Theorem 14 *If $\Gamma \vdash M : \tau$ and $M \hookrightarrow M^*$, then $\Gamma^* \vdash M^* : \tau^*$*

Proof. By induction on the derivation $M \hookrightarrow M^*$. Most cases are routine; the only interesting case involves the type generalization in **let**-expressions as τ^* may have less free type variables than τ. □

The reverse implication does not hold because the compilation loses information about encapsulation of state threads that cannot be recovered from the intermediate program. For example, the source term:

$$\mathbf{sto}\ \{\ell = 0\}\ (\mathsf{ret}\ \ell)$$

does not typecheck since it attempts to export a location. The corresponding target program:

$$\langle \mathsf{fst}\ ((\lambda x.(\ell, x))\ \square) \mid \ell = 0\rangle$$

typechecks with no problems.

6 Compiler Correctness

Because of the wide semantics gap between the source and target languages, the proof will proceed via an additional intermediate language in store-passing style that we refer to as SPS. The outline of the proof follows:

1. We decompose the compilation map in two maps. The first map converts the source language to SPS, eliminating the monadic combinators in the process. The second map substitutes the store data structures that are being passed around in SPS with the token $\Box$.
2. We then develop two semantics for SPS: a call-by-name one and a call-by-need one and show that they are equivalent.
3. Finally, we relate the source language to the call-by-name SPS semantics, and relate the target language to the call-by-need SPS semantics.

A notable characteristic of our proof technique is that we can only achieve weak correctness in the following sense. If the semantics specifies that a source term evaluates to an observable (such as 5), then we can show that the implementation must also produce the same observable. However we do not preclude the possibility that the implementation produces an observable even if the source semantics specifies that the program diverges.

The weak correctness is achieved by showing the following two statements for each compilation map:

1. translating both sides of each source reduction produces a valid equivalence in the target of the compilation map (either directly because it is a provable equality or because it is an observational equivalence), and
2. showing that the calculus for the target is consistent, *i.e.*, that it does not prove that $M = N$ for all M and N.

For example, if the source semantics specifies that a program should evaluate to 5, then the first fact implies that the target semantics proves that the compiled version of the program also evaluates to 5. The consistency of the semantics ensures that the program cannot evaluate to any other observable like 6.

6.1 SPS-name and SPS-need

We begin by defining the syntax of the store-passing language. This language is still pure: stores are represented as records and are copied to simulate updates. Stores are written in the set notation $\{\ell_i = M_i\}$.

Definition 15 (Syntax of SPS) *Let x range over a set of variables and ℓ range over a disjoint set of locations. The set of terms is inductively defined as follows:*

$$\begin{array}{rcl} M,N,L & ::= & x \mid M\,N \mid \langle M \mid x_i = M_i\rangle \\ & \mid & K \mid \lambda x.M \mid \ell \mid \{D\} \\ & \mid & \mathsf{fst}\ M \mid \mathsf{snd}\ M \mid (M,N) \mid [M,N] \\ & \mid & \mathsf{new}\ L\ M \mid \mathsf{read}\ L\ M \mid \mathsf{write}\ L\ M\ N \\ V & ::= & K \mid \lambda x.M \mid \ell \mid \{\ell_i = W_i\} \mid [M,N] \\ W & ::= & x \mid V \\ D & ::= & \ell_i = M_i \\ K & ::= & () \mid n \mid + \mid \ldots \end{array}$$

Being a pure language, the semantics of SPS should be indifferent to the parameter-passing mechanism [27]. Indeed we will develop a call-by-name semantics and a call-by-need semantics, and show their equivalence.

The call-by-name semantics is in Figure 7. Note that the location returned by the new operator is uniquely determined from D. Since the SPS-name calculus is used as the target of a compilation map, we need to verify its consistency.

Lambda Graphs rules:

$$\begin{array}{rcll} (\lambda x.M)\ N & \rightarrow & M[x := N] & \\ \langle M \mid x_i = M_i\rangle & \rightarrow & M[x_j := \langle M_j \mid x_i = M_i\rangle] & \\ K\ V & \rightarrow & \delta(K,V) & \text{if defined} \end{array}$$

Pairs:

$$\begin{array}{rcl} \mathsf{fst}\ (M,N) & \rightarrow & M \\ \mathsf{snd}\ (M,N) & \rightarrow & N \end{array}$$

References:

$$\begin{array}{rcl} \mathsf{new}\ \{D\}\ M & \rightarrow & (\ell, \{D, \ell = M\}) \\ \mathsf{read}\ \{D, \ell = M\}\ \ell & \rightarrow & (M, \{D, \ell = M\}) \\ \mathsf{write}\ \{D, \ell = M\}\ \ell\ N & \rightarrow & ((), \{D, \ell = N\}) \end{array}$$

Figure 7: Semantics of SPS-name

Lambda Graphs rules:

$$\begin{array}{rcll} (\lambda x.M)\ N & \rightarrow & M[x := N] & \\ \langle C[x] \mid x = V, D\rangle & \rightarrow & \langle C[V] \mid x = V, D\rangle & \\ \langle M \mid y = C[x], x = V, D\rangle & \rightarrow & \langle M \mid y = C[V], x = V, D\rangle & \\ k\ V & \rightarrow & \delta(k,V) & \text{if defined} \end{array}$$

Administrative reductions:

$$\begin{array}{rcl} E[\langle M \mid D\rangle] & \rightarrow & \langle E[M] \mid D\rangle \\ \langle\langle M \mid D_1\rangle \mid D_2\rangle & \rightarrow & \langle M \mid D_1, D_2\rangle \\ \langle M_1 \mid x = \langle M_2 \mid D_2\rangle, D_1\rangle & \rightarrow & \langle M_1 \mid x = M_2, D_2, D_1\rangle \end{array}$$

Reductions for Pairs:

$$\begin{array}{rcl} (M,N) & \rightarrow & \langle [x,y] \mid x = M, y = N\rangle \\ \mathsf{fst}\ [M,N] & \rightarrow & M \\ \mathsf{snd}\ [M,N] & \rightarrow & N \end{array}$$

References:

$$\begin{array}{rcl} \mathsf{new}\ D\ M & \rightarrow & \langle(\ell, \{\ell = x, D\}) \mid x = M\rangle \\ \mathsf{read}\ \{D, \ell = V\}\ \ell & \rightarrow & (V, \{D, \ell = V\}) \\ \mathsf{write}\ \{D, \ell = M\}\ \ell\ N & \rightarrow & \langle((), \{D, \ell = x\}) \mid x = N\rangle \\ \{\ell_i = M_i\} & \rightarrow & \langle\{\ell_i = x_i\} \mid x_i = M_i\rangle \end{array}$$

Figure 8: Semantics of SPS-need

Theorem 16 *The SPS-name calculus is Church-Rosser.*

The call-by-need semantics of SPS is in Figure 8. As in the target language substitution is restricted to values only. In contrast to the target imperative language all the internal boxes can be merged since no restrictions are imposed on where the store operations can be applied.

The two calculi can be proved equivalent using the notion of infinite normal forms. If we let $\mathit{Inf}_{\mathrm{SPS-name}}$ be the infinite normal form computed with respect to SPS-name, and $\mathit{Inf}_{\mathrm{SPS-need}}$ be the the infinite normal form computed with respect to SPS-need, we then have the following theorem.

Theorem 17 *Let M be a term in SPS, then:*

$$\mathit{Inf}_{\mathrm{SPS-name}}(M) = \mathit{Inf}_{\mathrm{SPS-need}}(M).$$

Proof. We use an intermediate calculus SPS*-name obtained by substituting the first two rules of Figure 7 with the fol-

$$\frac{}{x \hookrightarrow x}$$

$$\frac{M \hookrightarrow M^\# \qquad N \hookrightarrow N^\#}{M\ N \hookrightarrow M^\#\ N^\#}$$

$$\frac{M \hookrightarrow M^\# \qquad \forall i. M_i \hookrightarrow M_i^\#}{\textbf{let } x_i = M_i \textbf{ in } M \hookrightarrow \langle M^\# \mid x_i = M_i^\# \rangle}$$

$$\frac{M \hookrightarrow M^\# \qquad \forall i. M_i \hookrightarrow M_i^\#}{\textbf{sto } \{\ell_i = M_i\}\ M \hookrightarrow \mathsf{fst}\ (M^*\ \{\ell_i = M_i^*\})}$$

$$\frac{}{K \hookrightarrow K}$$

$$\frac{}{\ell \hookrightarrow \ell}$$

$$\frac{M \hookrightarrow M^\#}{\lambda x.M \hookrightarrow \lambda x.M^\#}$$

$$\frac{M \hookrightarrow M^\# \qquad N \hookrightarrow N^\#}{M \mathrel{\texttt{>>=}} N \hookrightarrow \lambda x.\langle N^\#\ (\mathsf{fst}\ p)\ (\mathsf{snd}\ p) \mid p = M^\#\ x\rangle}$$

$$\frac{M \hookrightarrow M^\#}{\mathsf{ret}\ M \hookrightarrow \lambda x.(M^\#, x)}$$

$$\frac{M \hookrightarrow M^\#}{\mathsf{New}\ M \hookrightarrow \lambda x.\mathsf{new}\ x\ M^\#}$$

$$\frac{M \hookrightarrow M^\#}{\mathsf{Read}\ M \hookrightarrow \lambda x.\mathsf{read}\ x\ M^\#}$$

$$\frac{M \hookrightarrow M^\# \qquad N \hookrightarrow N^\#}{\mathsf{Write}\ M\ N \hookrightarrow \lambda x.\mathsf{write}\ x\ M^\#\ N^\#}$$

Figure 9: Compiling Source to SPS

lowing rules:

$$\begin{array}{lcl} (\lambda x.M)\ N & \to & \langle M \mid x = N\rangle \\ \langle C[x] \mid x = M, D\rangle & \to & \langle C[M] \mid x = M, D\rangle \\ E[\langle M \mid D\rangle] & \to & \langle E[M] \mid D\rangle \end{array}$$

where the evaluation contexts are the ones of Definition 9. We then show that if $M \longrightarrow\!\!\!\!\!\rightarrow N$ in the SPS-name calculus then $M \longrightarrow\!\!\!\!\!\rightarrow L$ in the new calculus, where all the information contained in N is contained in L. The same holds in the other direction. This guarantees that the infinite normal form does not change. The SPS*-name calculus without reference operations corresponds to the cyclic call-by-name evaluation calculus [2], and hence we can use a technique similar to the one we previously used to prove that sharing preserves infinite normal forms. □

Since the notion of infinite normal form defines a congruence on the set of terms, we have as a corollary of the previous theorem that SPS-need is observationally equivalent to SPS-name.

6.2 From Source to SPS-name

The translation from the source language to SPS is in Figure 9. It is a conventional store-passing translation.

As explained at the beginning of this section, it remains to establish that the compilation of a source reduction yields a valid SPS-name equivalence.

Theorem 18 (Soundness) *Let M, N be source terms. If $M \to N$ in the source language, then $M^\# = N^\#$ in SPS-name.*

Proof. It suffices to show that compiling both sides of every reduction in Figure 2 yields an equation of SPS-name. We show some cases in detail:

Case $(\lambda x.M)N \to M[x := N]$. Compiling the left hand side yields $(\lambda x.M^\#)N^\#$ which is equal to $M^\#[x := N^\#]$ since β is an axiom of SPS-name. The result follows by a straightforward substitution lemma.

Case $\textbf{sto}\ \{D\}\ (\mathsf{New}\ M \mathrel{\texttt{>>=}} H) \to \textbf{sto}\ \{D, \ell = M\}\ (H\ \ell)$. Compiling the left hand side yields:

$$\begin{array}{cl} & \mathsf{fst}\ ((\lambda x.\langle H^\#\ (\mathsf{fst}\ p)\ (\mathsf{snd}\ p) \mid p = (\lambda y.\mathsf{new}\ y\ M^\#)x\rangle)\ D^\#) \\ = & \mathsf{fst}\ \langle H^\#\ (\mathsf{fst}\ p)\ (\mathsf{snd}\ p) \mid p = (\lambda y.\mathsf{new}\ y\ M^\#)\ D^\#\rangle \\ = & \mathsf{fst}\ \langle H^\#\ (\mathsf{fst}\ p)\ (\mathsf{snd}\ p) \mid p = \mathsf{new}\ D^\#\ M^\#\rangle \\ = & \mathsf{fst}\ \langle H^\#\ (\mathsf{fst}\ p)\ (\mathsf{snd}\ p) \mid p = (\ell, \{D^\#, \ell = M^\#\})\rangle \\ = & \mathsf{fst}\ (H^\#\ (\mathsf{fst}\ (\ell, \{D^\#, \ell = M^\#\}))\ (\mathsf{snd}\ (\ell, \{D^\#, \ell = M^\#\}))) \\ = & \mathsf{fst}\ (H^\#\ \ell\ \{D^\#, \ell = M^\#\}) \end{array}$$

which is the compilation of the right hand side.

Case $\textbf{sto}\ \{D\}\ R[\mathsf{ret}\ (\lambda y.M)] \to \lambda y.\textbf{sto}\ \{D\}\ R[\mathsf{ret}\ M]$. We proceed by induction on the structure of R. We only show the base case when R is the empty context. Compiling the left hand side yields:

$$\begin{array}{cl} & \mathsf{fst}\ ((\lambda x.(\lambda y.M^\#, x))\ D^\#) \\ = & \mathsf{fst}\ (\lambda y.M^\#, D^\#) \\ = & \lambda y.M^\# \end{array}$$

Compiling the right hand side also yields:

$$\begin{array}{cl} & \lambda y.\mathsf{fst}\ ((\lambda x.(M^\#, x))\ D^\#) \\ = & \lambda y.\mathsf{fst}\ (M^\#, D^\#) \\ = & \lambda y.M^\# \end{array}$$

The remaining cases proceed in a similar fashion. □

6.3 From SPS-need to Target

The translation from SPS to the target imperative language is in Figure 10. This translation simply moves the records representing the stores from the argument positions of new, read, and write, into the lists of mutually recursive declarations.

For the correctness of this compilation map, we augment the imperative call-by-need calculus with store records and one additional rule:

$$\{\ell_i = M_i\} \to \langle \Box \mid \ell_i = M_i\rangle$$

The rule states then once some expression evaluates to a well-defined store, *i.e.*, a record containing pairs of locations and expressions, then we can replace that expression with the token □. This is consistent with our intuitive understanding of the role of the token □ and formalizes the move from local stores to a global one.

Theorem 19 (Soundness) *Let M, N be SPS terms. If $M \to N$ in the SPS-need calculus, then $M^\dagger$ is observationally equivalent to $N^\dagger$ in the augmented imperative call-by-need semantics.*

$$\frac{}{x \hookrightarrow x}$$

$$\frac{M \hookrightarrow M^\dagger \qquad N \hookrightarrow N^\dagger}{M\ N \hookrightarrow M^\dagger\ N^\dagger}$$

$$\frac{M \hookrightarrow M^\dagger \qquad \forall i. M_i \hookrightarrow M_i^\dagger}{\langle M \mid x_i = M_i \rangle \hookrightarrow \langle M^\dagger \mid x_i = M_i^\dagger \rangle}$$

$$\frac{M \hookrightarrow M^*}{\mathsf{fst}\ M \hookrightarrow \mathsf{fst}\ M^\dagger}$$

$$\frac{M \hookrightarrow M^*}{\mathsf{snd}\ M \hookrightarrow \mathsf{snd}\ M^\dagger}$$

$$\frac{L \hookrightarrow L^\dagger \qquad M \hookrightarrow M^\dagger}{\mathsf{new}\ L\ M \hookrightarrow \langle \mathsf{new}\ x\ M^\dagger \mid x = L^\dagger \rangle}$$

$$\frac{L \hookrightarrow L^\dagger \qquad M \hookrightarrow M^\dagger}{\mathsf{read}\ L\ M \hookrightarrow \langle \mathsf{read}\ x\ M^\dagger \mid x = L^\dagger \rangle}$$

$$\frac{L \hookrightarrow L^\dagger \qquad M \hookrightarrow M^\dagger \qquad N \hookrightarrow N^\dagger}{\mathsf{write}\ L\ M\ N \hookrightarrow \langle \mathsf{write}\ x\ M^\dagger N^\dagger \mid x = L^\dagger \rangle}$$

$$\frac{}{K \hookrightarrow K}$$

$$\frac{}{\ell \hookrightarrow \ell}$$

$$\frac{M \hookrightarrow M^\dagger}{\lambda x.M \hookrightarrow \lambda x.M^\dagger}$$

$$\frac{\forall i. M_i \hookrightarrow M_i^\dagger}{\{\ell_i = M_i\} \hookrightarrow \{\ell_i = M_i^\dagger\}}$$

$$\frac{M \hookrightarrow M^\dagger \qquad N \hookrightarrow N^\dagger}{(M, N) \hookrightarrow (M^\dagger, N^\dagger)}$$

$$\frac{M \hookrightarrow M^\dagger \qquad N \hookrightarrow N^\dagger}{[M, N] \hookrightarrow [M^\dagger, N^\dagger]}$$

Figure 10: Compiling SPS to Target

Proof. We show one of the interesting cases:

$$\begin{array}{cl}
 & (\mathsf{read}\ \{D, \ell = V\}\ \ell)^\dagger \\
= & \langle \mathsf{read}\ x\ \ell \mid x = \{D^\dagger, \ell = V^\dagger\} \rangle \\
= & \langle \mathsf{read}\ x\ \ell \mid x = \langle \Box \mid D^\dagger, \ell = V^\dagger \rangle \rangle \\
= & \langle \mathsf{read}\ x\ \ell \mid x = \Box, D^\dagger, \ell = V^\dagger \rangle \\
= & \langle \mathsf{read}\ \Box\ \ell \mid x = \Box, D^\dagger, \ell = V^\dagger \rangle \\
= & \langle \mathsf{read}\ \Box\ \ell \mid D^\dagger, \ell = V^\dagger \rangle \\
= & \langle [V^\dagger, \Box] \mid D^\dagger, \ell = V^\dagger \rangle \\
= & \langle (V^\dagger, \Box) \mid D^\dagger, \ell = V^\dagger \rangle \\
= & (V^\dagger, \langle \Box \mid D^\dagger, \ell = V^\dagger \rangle) \\
= & (V^\dagger, \{D^\dagger, \ell = V^\dagger\}) \\
= & ((V, \{D, \ell = V\}))^\dagger
\end{array}$$

Some of the steps use semantic equalities that are not part of the calculus like the collection of unused bindings, and the identification of different representations of the same underlying graph [1]. The cases for new and write proceed similarly. □

7 Imperative Lazy Programming

In addition to being the main technical device for proving the correctness of the implementation for monadic state, the imperative call-by-need λ-calculus is useful in its own right. We briefly motivate the additional expressive power of the calculus over pure call-by-need calculi with one example.

Like many system-level Haskell programs, the implementation of *memo* functions relies on expressions with side-effects [9, 16]. Since Haskell bans such expressions, programmers that need this functionality either write low-level routines in C or in an *ad hoc* variant of Haskell that integrates computational effects with the call-by-need semantics [6].

Not only does the imperative call-by-need λ-calculus let programmers express such services as suggested above, but also it provides the infrastructure to reason about the correctness and properties of such services.

Example 20 (Invocation Counter) *It is easy to use the imperative call-by-need λ-calculus to write a procedure that keeps track of how many times it is called:*

$$\begin{array}{l}
f = \langle \lambda_.\langle \mathsf{fst}\ p_2 \mid p_1 = \mathsf{read}\ \Box\ \ell \\
\qquad\qquad\qquad v_1 = \mathsf{fst}\ p_1 \\
\qquad\qquad\qquad s_1 = \mathsf{snd}\ p_1 \\
\qquad\qquad\qquad p_2 = \mathsf{read}\ (\mathsf{snd}\ (\mathsf{write}\ s_1\ \ell\ (v_1 + 1)))\ \ell \rangle \\
\qquad \mid \ell = 0 \rangle
\end{array}$$

The procedure refers to a location initialized to 0*. Every invocation reads the current value of the location in v_1, writes $(v_1 + 1)$ in the contents of the location, and returns the last value. The last* read *operation forces the* write *to occur since it is strict in its store argument.*

It is now easy to use the axioms to calculate that the expression $(f\ () + f\ ())$ reduces to 1+2=3.

8 Related Work

Our work builds directly on Ariola *et al.*'s call-by-need calculi [1, 4], the encapsulation and implementation of monadic state by Launchbury and Peyton Jones [18, 19], and the axiomatization of monadic state by Launchbury and Sabry [20]. The imperative call-by-need calculus uses ideas developed in the context of imperative call-by-value calculi [11, 12, 28] as well as in the conservative extensions of functional languages with assignments [22, 31].

The correctness of "updates-in-place" is a long standing problem that has been studied extensively. In the context of monads, a number of researchers [8, 15, 26, 33, 35] noted that when the state monad is treated as an abstract data type, it ought to be possible to implement the state operations more efficiently using *destructive updates*. For example, Wadler states:

> The abstract data type for ST guarantees that it is safe to perform updates (indicated by *assign*) in place - no special analysis technique is required [33, p.70].

However, in an implementation like ghc, the monad combinators are eventually expressed using more primitive operations (see the compilation map in Figure 6). Once the monad is "open" (*i.e.*, no longer viewed as an abstract data type), the above argument used to state the correctness of destructive updates is no longer valid. Indeed, *other* constructs in the language may now interact with the monadic effects in unpredictable ways and we can no longer reason about the implementation of the monadic effects in isolation.

The problem of the correctness of "updates-in-place" has been studied in other contexts [14, 29, 30, 34, 36]. We have not yet explored the detailed connections with linear logic but it appears that the path semantics used by Sestoft [30] in a call-by-value context does not easily extend to the call-by-need world.

9 Conclusion and Future Work

We have formalized the implementation of monadic state using a typed imperative call-by-need language and a type-preserving compilation map. The formal treatment supports the first proof of correctness for efficient implementations of monadic state, and resulted in the development of an imperative extension of the call-by-need λ-calculus. Since assignments change the space complexity of programs, we envision such an extension as a foundation for reasoning about the space properties of call-by-need programs.

Acknowledgments

We would like to thank Stefan Blom, Yong Xiao and the anonymous reviewers for their helpful comments. The research has been supported by the National Science Foundation under grant CCR-9624711.

References

[1] Ariola, Z. M., and Blom, S. Cyclic lambda calculi. In the *International Symposium on Theoretical Aspects of Computer Software, Sendai, Japan* (September 1997).

[2] Ariola, Z. M., and Blom, S. Lambda calculi plus letrec. Tech. Rep. CIS-TR-97-05, Dept. of Computer and Information Science, University of Oregon, 1997.

[3] Ariola, Z. M., and Felleisen, M. The call-by-need lambda calculus. *Journal of Functional Programming 7*, 3 (1997).

[4] Ariola, Z. M., Felleisen, M., Maraist, J., Odersky, M., and Wadler, P. A call-by-need lambda calculus. In the *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1995), ACM Press, New York, pp. 233–246.

[5] Ariola, Z. M., and Klop, J. W. Lambda calculus with explicit recursion. Tech. Rep. CIS-TR-96-04, Dept. of Computer and Information Science, University of Oregon, 1996. To appear in *Inf. Comp.*.

[6] Augustsson, L., Rittri, M., and Synek, D. Functional pearl: On generating unique names. *Journal of Functional Programming 4*, 1 (Jan. 1994), 117–123. Original version: Splitting Infinite Sets of Unique Names by Hidden State Changes, Tech. Rep. 67, Programming Methodology Group, Chalmers University of Technology, 1992.

[7] Cartwright, R., and Felleisen, M. The semantics of program dependence. In the *ACM SIGPLAN Conference on Programming Language Design and Implementation* (1989), ACM Press, New York, pp. 13–27.

[8] Chen, C.-P., and Hudak, P. Rolling your own mutable ADT—A connection between linear types and monads. In the *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (15–17 Jan. 1997), ACM Press, New York, pp. 54–66.

[9] Cook, B., and Launchbury, J. Disposable memo functions. In *the ACM SIGPLAN Haskell Workshop* (1997).

[10] Crank, E. Parameter-passing and the lambda calculus. Master's thesis, Dept. of Computer Science, Rice University, 1990.

[11] Crank, E., and Felleisen, M. Parameter-passing and the lambda calculus. In the *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1991), ACM Press, New York, pp. 233–244.

[12] Felleisen, M., and Hieb, R. The revised report on the syntactic theories of sequential control and state. *Theoret. Comput. Sci. 102* (1992), 235–271. Tech. Rep. 89-100, Rice University.

[13] Field, J. A simple rewriting semantics for realistic imperative programs and its application to program analysis (preliminary report). Tech. Rep. YALE/DCS/RR-909, Dept. of Computer Science, Yale University, 1992. ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, pages 98-107.

[14] Guzmán, J. C., and Hudak, P. Single-threaded polymorphic lambda calculus. In the *IEEE Symposium on Logic in Computer Science* (1990), IEEE Computer Society Press, Los Alamitos, Calif., pp. 333–343.

[15] Hudak, P. Mutable abstract datatypes –or– how to have your state and munge it too. Tech. Rep. YALEU/DCS/RR-914, Dept. of Computer Science, Yale University, December 1992. Revised May 1993.

[16] Hughes, J. Lazy memo-functions. Tech. Rep. 21, Programming Methodology Group, Chalmers University of Technology, 1985.

[17] Launchbury, J. Lazy imperative programming. Tech. Rep. YALEU/DCS/RR-968, Dept. of Computer Science, Yale University, 1993. ACM SIGPLAN Workshop on State in Programming Languages.

[18] Launchbury, J., and Peyton Jones, S. L. Lazy functional state threads. In the *ACM SIGPLAN Conference on Programming Language Design and Implementation* (1994), pp. 24–35.

[19] Launchbury, J., and Peyton Jones, S. L. State in Haskell. *Lisp Symbol. Comput. 8* (1995), 193–341.

[20] Launchbury, J., and Sabry, A. Monadic state: Axiomatization and type safety. In the *ACM SIGPLAN International Conference on Functional Programming* (1997), ACM Press, New York, pp. 227–238.

[21] Lévy, J.-J. Optimal reductions in the lambda-calculus. In *To H.-B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalism* (1980), Academic Press, pp. 159–291.

[22] Odersky, M., Rabin, D., and Hudak, P. Call by name, assignment, and the lambda calculus. In the *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Jan. 1993), ACM Press, New York, pp. 43–56.

[23] Peyton Jones, S., and Launchbury, J. Unboxed values as first class citizens. In the *Conference on Functional Programming and Computer Architecture* (Sept. 1991), Hughes, Ed., vol. 523 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, pp. 636–646.

[24] Peyton Jones, S., Launchbury, J., Shields, M., and Tolmach, A. Bridging the gulf: A common intermediate language for ML and Haskell. In the *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1998), ACM Press, New York. To appear.

[25] Peyton Jones, S. L. *The Implementation of Functional Programming Languages*. International Series in Computer Science. Prentice-Hall, 1987.

[26] Peyton Jones, S. L., and Wadler, P. Imperative functional programming. In the *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1993), ACM Press, New York, pp. 71–84.

[27] Sabry, A. What is a purely functional language? *Journal of Functional Programming* (1997). To appear.

[28] Sabry, A., and Field, J. Reasoning about explicit and implicit representations of state. Tech. Rep. YALEU/DCS/RR-968, Dept. of Computer Science, Yale University, 1993. *ACM SIGPLAN Workshop on State in Programming Languages*, pages 17-30.

[29] Schmidt, D. A. Detecting global variables in denotational specifications. *ACM Trans. Program. Lang. Syst. 7*, 2 (Apr. 1985), 299–310.

[30] Sestoft, P. Replacing function parameters by global variables. In the *Conference on Functional Programming and Computer Architecture* (1989).

[31] Swarup, V., Reddy, U., and Ireland, E. Assignments for applicative languages. In the *Conference on Functional Programming and Computer Architecture* (1991), pp. 192–214.

[32] Tarditi, D., Morrisett, G., Cheng, P., Stone, C., Harper, R., and Lee, P. TIL: A type-directed optimizing compiler for ML. In the *ACM SIGPLAN Conference on Programming Language Design and Implementation* (1996), ACM Press, New York, pp. 181–192.

[33] Wadler, P. Comprehending monads. In the *ACM Conference on Lisp and Functional Programming* (1990), pp. 61–78.

[34] Wadler, P. Is there a use for linear logic. In the *Symposium on Partial Evaluation and Semantics-Based Program Manipulation* (1991), ACM Press, New York.

[35] Wadler, P. The essence of functional programming (invited talk). In the *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1992), ACM Press, New York, pp. 1–14.

[36] Wakeling, D., and Runciman, C. Linearity and laziness. In the *Conference on Functional Programming and Computer Architecture* (Sept. 1991), Hughes, Ed., vol. 523 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, pp. 215–240.