Assignments for Applicative Languages[*]

Vipin Swarup[†]
The MITRE Corporation
Burlington Road
Bedford, MA 01730.
E-mail: swarup@mitre.org

Uday S. Reddy[‡]
Dept. of Computer Science
University of Illinois
at Urbana-Champaign
Urbana, IL 61801.
E-mail: reddy@cs.uiuc.edu

Evan Ireland
School of Information Sciences
Massey University
Palmerston North
New Zealand.
E-mail: E.Ireland@massey.ac.nz

Abstract

We propose a theoretical framework for adding assignments and dynamic data to functional languages without violating their semantic properties. This differs from semi-functional languages like Scheme and ML in that values of expressions remain static and side-effect-free. A new form of abstraction called *observer* is designed to encapsulate state-oriented computation from the remaining purely applicative computation. The type system ensures that observers are combined linearly, allowing an implementation in terms of a global store. The utility of this extension is in manipulating *shared dynamic data* embedded in data structures. Evaluation of well-typed programs is Church-Rosser. Thus, programs produce the same results whether an eager or lazy evaluation order is used (assuming termination). A simple, sound logic permits reasoning about well-typed programs. The benefits of this work include greater expressive power and efficiency (compared to applicative languages), while retaining simplicity of reasoning.

Keywords Functional languages, imperative programming, lambda calculus, type systems, strong normalization, Church-Rosser property, referential transparency, continuation-passing style.

1 Introduction

Functional languages are popular among computer scientists because of their strong support of modularity. They possess two powerful glues, higher-order functions and laziness, that permit programs to be modularized in new, useful ways. Hughes [Hug90] convincingly argues that "...lazy evaluation is too important to be relegated to second-class citizenship. It is perhaps the most powerful glue functional programmers possess. One should not obstruct access to such a vital tool." However, side-effects are incompatible with laziness: programming with them requires knowledge of global context, defeating the very modularity that lazy evaluation is designed to enhance.

Pure functional languages have nice properties that make them easy to reason about. For instance, + is commutative, = is reflexive, and most other familiar mathematical properties hold of the computational operators. This is a consequence of expressions representing *static* values: values that do not change over time. Thus, an expression's value is independent of the order in which its sub-expressions are evaluated. Side-effects are incompatible with these properties, as side-effects change the values of other expressions, making the order of evaluation important.

Assignments are a means of describing *dynamic* data : data whose values change over time. In their conventional form, assignments have side-effects on their environment, making their order of evaluation important. Not only are such assignments incompatible with laziness, but they also destroy the nice mathematical properties of pure languages. Hence lazy functional languages shun assignments.

However, since assignments directly model the dynamic behavior of a physical computer's store, they yield efficient implementations of dynamic data. In contrast, one models dynamic data in functional languages by representing the state explicitly or, possibly, by creating streams of states. Compilation techniques and language notations have been proposed to permit explicit state manipulation to be implemented efficiently [HB85, GH90, Wad90b, Wad90a]. Unfortunately, these methods do not achieve all the effects of true dynamic data. For instance, dynamic

[*]To appear in the Proceedings of the Conference on Functional Programming Languages and Computer Architecture, August 1991.

[†]Supported by NASA grant NAG-1-613 (while at the University of Illinois at Urbana-Champaign).

[‡]Supported by a grant from Motorola Corporation.

data may be "shared", i.e., embedded in data structures and accessed via different access paths. When shared dynamic data are updated using assignments, the change is visible to all program points that have access to the data. In contrast, when state is being manipulated explicitly, updating shared data involves constructing a fresh copy of the entire data structure in which the data are embedded, and explicitly passing the copy to all program points that need access to the data. This tends to be tedious and error-prone, and results in poor modularity. One particularly faces this difficulty while encoding graph traversal algorithms such as topological sort, unification and the graph reduction execution model of lazy functional languages.

In this paper, we propose a theoretical framework for extending functional languages with dynamic data and assignments while retaining the desirable properties of static values. The resulting language has the following key properties:

- Expressions have static values.
  State-dependent and state-independent expressions are distinguished via a type system. The former are viewed as functions from states to values and the functions themselves are static. (Such functions are called *observers* and resemble classical continuations [Sto77, SW74]). The type system ensures that this view can be consistently maintained, and limits the interaction between observers in such a way that expressions do not have side-effects [1].

- The language is a strict extension of lambda calculus.
  Function abstraction and application have precisely the same meaning as in lambda calculus. This is a key property that is not respected by call-by-value languages like Scheme (even in the absence of side-effects). The operational semantics is presented as a reduction system that consists of the standard reduction rules of lambda calculus together with a set of additional rules; these rules exhibit symmetries similar to those of lambda calculus. The reduction system is *confluent* (or, equivalently, Church-Rosser), and recursion-free terms are *strongly normalizing*.

[1]In the contemporary functional programming community, the terms "assignment" and "side-effect" are sometimes used synonymously. We use the term "side-effect" in its original meaning: an expression has a side-effect if, in addition to yielding a value, it changes the state in a manner that affects the values of other expressions in the context. Assignments in our proposed language do not have such side-effects. Similar comments apply to terms like "procedure" and "object".

This work can also be given a logical interpretation: it extends the correspondence between logic and programming to include dynamic data. Entities that describe dynamic data, namely references, play a role similar to that of variables in conventional logic. The language described in this paper is the language of constructions for a suitably formulated constructive logic. This dual aspect is treated elsewhere [SR91, Swa91].

The utility of the language is characterized by the following properties:

- Shared dynamic data are available.
  Dynamic data are represented by typed objects called *references*. References can refer to other references as well as to functions. They can be embedded in data structures and used as inputs and outputs of functions.

- Dynamic data may be implemented by a store.
  This is achieved by a type system that sequentializes access to regions of the state, much as in effect systems [GL86] and the languages based on linear logic [GH90, Wad90b, Wad91].

- The language is higher-order.
  References, data structures, functions and observers are all permissible as arguments and results of functions. This permits, for instance, the definition of new control structures, new storage allocation mechanisms, and an object-oriented style of programming.

- The language is integrated symmetrically.
  The applicative sublanguage and the imperative sublanguage are equally powerful and they embed each other. Not only can applicative terms be embedded in imperative terms, but imperative terms can also be embedded in applicative terms. This allows the definition of functions that create and use state internally but are state-independent externally.

The remainder of this paper is organized as follows. Section 2 presents a core formal language called Imperative Lambda Calculus (ILC) that is an extension of the typed lambda calculus. Section 3 studies ILC's use in programming. Section 4 discusses the motivation and design issues behind ILC's type system. Section 5 presents the formal semantics of ILC. This includes a typed denotational semantics and an operational semantics presented as reduction rules. Various formal properties are established, such as type soundness, confluence and strong normalization. Section 6 demonstrates the utility of ILC with an extended example: the unification of first-order terms. Finally, Section 7 compares ILC with related work in the literature.

2 Imperative Lambda Calculus (ILC)

Imperative Lambda Calculus (ILC) is an abstract formal language obtained by extending the typed lambda calculus [Mit90] with imperative programming features. Its main property is that, in spite of this extension, its applicative sublanguage has the same semantic properties as the typed lambda calculus (eg. confluence and strong normalization). Furthermore, these same properties also hold for the entire language of ILC.

2.1 Types

Let β represent the primitive types of ILC. These may include the natural numbers, characters, strings etc. The syntax of ILC types is as follows:

$$
\begin{array}{lll}
\multicolumn{3}{l}{\text{(Applicative types)}} \\
\tau & ::= & \beta \mid \tau_1 \times \tau_2 \mid \tau_1 \to \tau_2 \\
\multicolumn{3}{l}{\text{(Mutable types)}} \\
\theta & ::= & \tau \mid \texttt{Ref}\,\theta \mid \theta_1 \times \theta_2 \mid \theta_1 \to \theta_2 \\
\multicolumn{3}{l}{\text{(Observer types)}} \\
\omega & ::= & \theta \mid \texttt{Obs}\,\tau \mid \omega_1 \times \omega_2 \mid \omega_1 \to \omega_2
\end{array}
$$

The type system is stratified into three layers. The *applicative* layer τ contains the types of the simply typed lambda calculus (extended with pairs). These applicative types include the primitive types β and are closed under product and function space constructions. Note that we use the term "applicative" to refer to the *classical* values manipulated in lambda calculus; semantically, all three layers of ILC are applicative.

The *mutable* layer θ extends the applicative layer with objects called *references*. References are typed values that refer (i.e. point) to values of a particular type. ($\texttt{Ref}\ \theta$) denotes the type of references that refer to values of type θ. References are used to construct a mutable world (called a *store*) that is used for imperative programming. The world itself is mutable and goes through *states*. The mutable layer includes all applicative types and is closed under the type constructors $\times, \to$ and `Ref`. Note that references can point to other references, thereby permitting linked data structures. Tuples of references denote mutable records, while reference-returning functions denote mutable arrays.

Finally, the world of the mutable layer needs to be manipulated. In ILC, we take the position that the only manipulation needed for states is *observation* (i.e. inspection). Consider the fact that in the typed lambda calculus, environments are implicitly extended and observed (via the use of variables), but are never explicitly manipulated. Similarly, in ILC, states are implicitly extended and observed (via the use of references), but are never explicitly manipulated. Thus, in a sense, *the world exists only to be observed.* A state differs from an environment in that it may be mutated while being observed; the mutation is restricted to the observation and is not visible to expressions outside the observation.

Observation of the state is accommodated in the *observer* layer ω. This layer includes all applicative and mutable types. In addition, it includes a new type constructor denoted "$\texttt{Obs}\ \tau$". A value of type $\texttt{Obs}\ \tau$ is called an *observer.* Such a value observes (i.e. views or inspects) a state and returns a value of type τ. It is significant that the value returned in this fashion is of an applicative type τ. Since a state exists only to be observed, all information about the state is lost when its observation is completed. So, the values observed in this fashion should be meaningful independent of the state, i.e., they should be applicative. An observer type $\texttt{Obs}\ \tau$ may be viewed as an implicit function space from the set of states to the type τ.

The three layers can be characterized as *kinds* and given category-theoretic semantics. The product and function space constructions have the same meaning in all three layers (cf. Section 5). Thus, there is no ambiguity involved in treating τ types as also being θ and ω types. The name "Imperative Lambda Calculus" is justified by the property that the semantics of functions in all three layers is the same as that of lambda calculus.

2.2 Terms

The abstract syntax of unchecked "preterms" is as follows:

$$
\begin{array}{lll}
e & ::= & k \mid x \mid v^* \mid \lambda x{:}\,\omega.e \mid f(e) \mid \langle e_1, e_2 \rangle \mid e.1 \mid e.2 \\
 & & \mid \texttt{letref}\ v^*{:}\,\texttt{Ref}\,\theta := e\ \texttt{in}\ t \\
 & & \mid \texttt{get}\ x{:}\,\theta \Leftarrow l\ \texttt{in}\ t \\
 & & \mid l := e\ ;\ t
\end{array}
$$

where k are constants, x, v^* are variables, e, e_1, e_2, f, l, t are terms and ω, θ are types.

The constants of ILC *are limited to be of applicative type.* Permissible constants include numbers, booleans, characters and primitive functions on these

Constant

$$\Gamma \vdash k\colon \tau$$
(if k is a constant of type τ)

Weakening

$$\frac{\Gamma \vdash e\colon \pi}{\Gamma, x\colon \pi' \vdash e\colon \pi}$$

Variable hypothesis

$$\Gamma, x\colon \pi, \Gamma' \vdash x\colon \pi$$

Reference hypothesis

$$\Gamma, v^*\colon \mathtt{Ref}\ \theta, \Gamma' \vdash v^*\colon \mathtt{Ref}\ \theta$$

→-intro

$$\frac{\Gamma, x\colon \pi_1 \vdash e\colon \pi_2}{\Gamma \vdash (\lambda x\colon \pi_1 . e)\colon \pi_1 \to \pi_2}$$

→-elim

$$\frac{\Gamma \vdash f\colon \pi_1 \to \pi_2 \quad \Gamma \vdash e\colon \pi_1}{\Gamma \vdash f(e)\colon \pi_2}$$

×-intro

$$\frac{\Gamma \vdash e_1\colon \pi_1 \quad \Gamma \vdash e_2\colon \pi_2}{\Gamma \vdash \langle e_1, e_2 \rangle\colon \pi_1 \times \pi_2}$$

×-elim

$$\frac{\Gamma \vdash e\colon \pi_1 \times \pi_2}{\Gamma \vdash e.i\colon \pi_i} \quad \text{for } i = 1, 2$$

Obs-intro

$$\frac{\Gamma \vdash t\colon \tau}{\Gamma \vdash t\colon \mathtt{Obs}\ \tau}$$

Obs-elim

$$\frac{\Gamma \vdash t\colon \mathtt{Obs}\ \tau}{\Gamma \vdash t\colon \tau} \quad \text{(if } \Gamma \text{ has only } \tau \text{ types)}$$

Creation

$$\frac{\Gamma, v^*\colon \mathtt{Ref}\ \theta \vdash e\colon \theta \quad \Gamma, v^*\colon \mathtt{Ref}\ \theta \vdash t\colon \mathtt{Obs}\ \tau}{\Gamma \vdash (\mathtt{letref}\ v^*\colon \mathtt{Ref}\ \theta := e\ \mathtt{in}\ t)\colon \mathtt{Obs}\ \tau}$$

Dereference

$$\frac{\Gamma \vdash l\colon \mathtt{Ref}\ \theta \quad \Gamma, x\colon \theta \vdash t\colon \mathtt{Obs}\ \tau}{\Gamma \vdash (\mathtt{get}\ x\colon \theta \Leftarrow l\ \mathtt{in}\ t)\colon \mathtt{Obs}\ \tau}$$

Assignment

$$\frac{\Gamma \vdash l\colon \mathtt{Ref}\ \theta \quad \Gamma \vdash e\colon \theta \quad \Gamma \vdash t\colon \mathtt{Obs}\ \tau}{\Gamma \vdash (l := e\ ;\ t)\colon \mathtt{Obs}\ \tau}$$

Figure 1: Type inference rules

values. No imperative constants (i.e. no constants involving mutable or observer types) are permitted. This permits us to carefully control the creation and use of the state. We (partially) relax this restriction in section 5.1.

The terms of ILC use two countable sets of variables: *conventional variables* and *reference variables*. Conventional variables are the usual variables of the typed lambda calculus. Reference variables are a new set of variables that share all the properties of conventional variables. Further, distinct reference variables within a term always denote distinct references. References are always introduced by binding them to reference variables; conventional variables can then be bound to such references. This property permits us to reason about the equality of references without recourse to reference constants (which are absent from the language). In the formal presentation, we use an asterisk superscript to distinguish reference variables u^*, v^*, w^* from conventional variables x, y, z. Since the context of a variable determines whether it is a reference or conventional variable, we do not use asterisk superscripts in any of our examples.

Figure 1 presents the context sensitive type syntax of ILC terms. The syntax is expressed as inference rules for judgements of the form $(\Gamma \vdash e\colon \pi)$, where e is a term, π is a type (of any kind τ, θ or ω), and Γ is a sequence of typing assumptions of the forms $(x\colon \pi)$ or $(v^*\colon \texttt{Ref}\ \theta)$. Γ contains typing assumptions for all the free variables in e.

ILC includes the simply-typed lambda calculus extended with pairs. These terms have their usual meaning in all three layers (rules $\rightarrow$-*intro*, $\rightarrow$-*elim*, $\times$-*intro*, and $\times$-*elim*). In addition, ILC contains three new observer terms to create a new reference (*creation*), access a reference's content (*dereference*), and modify a reference's content (*assignment*). We now discuss these terms in more detail.

We have seen that there are no reference constants in the language. All references have to be explicitly allocated and bound to a reference variable. This is done by the `letref` construct:

$$\texttt{letref}\ v^*\colon \texttt{Ref}\ \theta := e\ \texttt{in}\ t$$

Such a term is an observer of the same type as t (rule *Creation*). When used to observe a state, it extends the state by creating a new reference, extends the environment by binding v^* to the reference, initializes the reference to the value of e in the extended environment, and finally observes the value of t in the extended environment and state.

The mutable world of references may be inspected by dereferencing a reference, i.e. by inspecting the value that the reference points to, or, using alternate terminology, by inspecting the reference's *content*. If l is a reference-valued expression of type $\texttt{Ref}\ \theta$, then a term of the form

$$\texttt{get}\ x\colon \theta \Leftarrow l\ \texttt{in}\ t$$

binds x to the content of l, and denotes the value of t in the extended environment. Here, t must be an observer of type $\texttt{Obs}\ \tau$, and the entire term is again an observer of type $\texttt{Obs}\ \tau$ (rule *Dereference*).

Finally, the content of a reference may be modified via assignment observers of the form

$$l := e\ ;\ t$$

where l is of type $\texttt{Ref}\ \theta$ and e is of type θ, for some θ (rule *Assignment*). When used to observe a state, an assignment observer modifies the reference l to refer to e, and observes the value of t in the modified state. Note that "$l := e$" is not a term by itself as in conventional languages. The state is modified *for* the observer t, and the entire construct is again an observer.

The lifetime of a mutable world (i.e. a collection of references) is limited to its observation. So, the creation of v^* and the modification of l are observable only within the bodies t of the creation and assignment observers respectively, and there are no side effects produced by the observers. If there are no free occurrences of reference variables or other state-dependent variables in an observer term, then the term is a trivial observer that is independent of any state. Such an observer can be coerced to an applicative term (rule *Obs-elim*). Conversely, every applicative term (every term of a τ type) is trivially an observer (rule *Obs-intro*).

It is important to note that all the primitive constructions on observers (get, letref and assignment) involve exactly one subterm of an observer type. This reflects the requirement that manipulations of state should be performed in a sequential fashion, similar in spirit to the proposal of single-threaded lambda calculus [GH90]. Even though it is possible to express functions which accept more than one observer, the state manipulations of such observers have to be eventually sequentialized because there are no multiary primitives on observers. (Recall that there are no constants of mutable and observer types). This fact has two consequences. First, programming in the imperative sublanguage of ILC requires a continuation-passing style. Second, the state can be implemented efficiently by means of a global store. We return to these issues in section 4.

3 ILC as a Programming Language

ILC can be used as a programming language in different styles. It can be used as a purely applicative language by restricting oneself to applicative types. It can be used as a purely imperative language by mainly using observers (this requires a continuation-passing style of programming). These styles correspond to traditional programming paradigms.

ILC also permits an interesting new style of programming. It permits closed imperative observers to be embedded in applicative terms (via the rule *Obs-elim*). Applicative terms can be freely embedded in imperative observers (via the rule *Obs-intro*). Higher-order functions and laziness can be used to glue together both imperative and applicative subcomputations, though imperative computation is restricted to continuation-passing style.

One extreme of this paradigm is to use ILC with imperative observers at the top level, but with non-trivial applicative subcomputations involving higher-order functions. This use is similar to that of Haskell where state-oriented input/output operations are usually carried out at the top level. More generally, ILC can be used with imperative observers embedded in applicative expressions (via the rule *Obs-elim*). This corresponds to the use of side-effect-free function procedures in Algol-like languages.

The examples in this paper further illustrate this style of programming. Example 1 (factorial) displays how imperative computations can be embedded in applicative terms. Example 2 (a movable point object) exhibits how imperative computations can be encapsulated as closures. The unification example of Section 6 demonstrates how the laziness of observers permits them to be passed to functions and returned as results.

3.1 Syntactic sugar for dereferencing

The need to use get's for dereferencing is rather tedious: it forces us to choose new names, and more importantly, it clutters up the code. The tedium can be alleviated to a large extent through a simple notational mechanism.

Abbreviation: If ($\texttt{get}\ x \Leftarrow l\ \texttt{in}\ t[x]$) is an observer term with no occurrence of x in a proper observer subterm of t, then we allow it to be abbreviated as $t[l\uparrow]$. $l\uparrow$ may be read informally as "the current content of reference l".

Expansion: If t is an observer term with a particular occurrence of $l\uparrow$, then it is expanded by introducing a "get" at the smallest observer subterm of t containing the occurrence of $l\uparrow$.

The intuition behind this abbreviation is that an observer term ($\texttt{get}\ x \Leftarrow l\ \texttt{in}\ t$) is a program point at which the content of l is observed, while occurrences of x in t are program points at which that content is used. If l is never modified between the point of dereference and a point of use, then we can safely view the dereference as taking place at the point of use. For example,

$$\begin{array}{l}(p := n\uparrow * p\uparrow;\ n := n\uparrow -1;\ c)\\ \quad = \texttt{get}\ x \Leftarrow n\ \texttt{in get}\ y \Leftarrow p\ \texttt{in}\\ \qquad\quad p := x * y;\\ \qquad\quad \texttt{get}\ z \Leftarrow n\ \texttt{in}\\ \qquad\qquad n := z - 1;\ c\end{array}$$

$$l\uparrow\uparrow = (\texttt{get}\ x \Leftarrow l\ \texttt{in}\ x\uparrow) = (\texttt{get}\ x \Leftarrow l\ \texttt{in get}\ y \Leftarrow x\ \texttt{in}\ y)$$

$$f(l\uparrow) = (\texttt{get}\ x \Leftarrow l\ \texttt{in}\ f(x)) \qquad \text{if} f : \tau_1 \to \texttt{Obs}\ \tau_2$$

$$f(l\uparrow) = f(\texttt{get}\ x \Leftarrow l\ \texttt{in}\ x) \qquad \text{if} f : \texttt{Obs}\ \tau_1 \to \texttt{Obs}\ \tau_2$$

3.2 Examples

For our examples, we assume that ILC is enhanced with user-defined type constructors and record types (drawn from standard ML [MTH90]). We also assume that ILC is enhanced with *explicit* parametric polymorphism with types ranging over the universe of applicative (τ) types. Implicit polymorphism is problematic in the presence of references and assignments [Tof88] — explicit polymorphism does not suffer from these problems. In our examples, we erase explicit type quantification and type application, and leave it to the reader to fill in the missing information.

We also assume primitives such as `case, let, letrec` and `if-then-else` for all types. Note that these primitives violate our earlier prohibition of primitives over mutable and observer types. In section 5.1, we shall see that such primitives are indeed permissible.

Example 1: Factorial

This trivial example is meant to provide an initial feel for the language, and illustrate how imperative observers can be embedded in applicative expressions. This example is not meant to illustrate the benefits of ILC; indeed, a preferred solution is to write this as a tail-recursive applicative function and have the compiler optimize the code into an iterative loop.

```
factorial =
  λm: nat. letref n: Ref nat := m in
           letref acc: Ref nat := 1 in
           letrec fact: Obs nat =
                 if (n↑ < 2) then acc↑
                 else acc := n↑ * acc↑;
                      n := n↑ −1;
                      fact
           in fact
```

The function `factorial` has no free references or state-dependent variables, and so has the applicative type (`nat` → `nat`). This means that `factorial` can be freely embedded in applicative expressions even though it contains imperative subcomputations.

Example 2: Points

We implement a point object that hides its internal state and exports operations. Let `Point` be the type of objects that represent movable planar points.

```
Point =
  {x_coord : (Real → Obs T) → Obs T,
   y_coord : (Real → Obs T) → Obs T,
   move : (Real × Real) → Obs T → Obs T,
   equal : Point → (Bool → Obs T) → Obs T}
```

The function `mkpoint` implements objects of type `Point`.

```
mkpoint : (Real → Real → (Point → Obs T) → Obs T)
  = λx. λy. λk.
      letref xc: Real := x in
      letref yc: Real := y in
      k({x_coord = λk. k(xc↑),
          y_coord = λk. k(yc↑),
          move = λ(dx, dy). λc.
                      xc := xc↑ +dx; yc := yc↑ +dy; c,
          equal = λ{x_coord,y_coord,move,equal}. λk.
                      x_coord(λx. y_coord(λy.
                             k(x = xc↑ and y = yc↑)))
          })
```

Note, first of all, that the `mkpoint` operation cannot simply yield a value of type `Point` because it is not an applicative value. The extent of xc and yc is limited to the bodies of the `letref`s which allocate these references; hence the entire computation which uses these references must occur in these bodies. Therefore, `mkpoint` is defined to accept a point observer function k and pass it the newly created point. This is similar to the continuation-passing style of programming. Observers here play the role of continuations.[2] Such continuation-passing style functions can be defined more conveniently using Wadler's monad comprehension notation [Wad90a]. Note that each operation in the object is similarly defined in the continuation-passing style.

This example demonstrates that state-encapsulating closures are available in ILC, albeit in the continuation-passing style. Such closures are also representable in semi-functional languages like Scheme and Standard ML, but usually involve side-effects.

4 Discussion of ILC

The motivation behind ILC's type system is threefold. First, we wish to exclude imperative terms that "export" their local effects. Consider the unchecked preterm (`letref` v := 0 `in` v). This term, if well-typed, would export the locally created reference v outside its scope resulting in a dangling pointer. Closures that capture references are prohibited for the same reason — they export state information beyond its local scope. The type system prohibits such terms by requiring the value returned by an observer to be applicative and hence free of state information. (Recall that observer types are of the form `Obs` τ where τ is an applicative type).

Second, we wish to ensure that the imperative sublanguage can be implemented efficiently without causing side-effects. Consider the unchecked preterm

$$v := 0 \; ; \; ((v := 2 \; ; \; \texttt{get}\ x \Leftarrow v\ \texttt{in}\ x) + (\texttt{get}\ x \Leftarrow v\ \texttt{in}\ x))$$

In a language with a global store and global assignments (eg. ML or Scheme), the value of the term depends on the order of evaluation of +'s arguments. Further, the term has the side-effect of changing the value of the global reference v to 2. On the other hand, if assignments are interpreted to have local effects, then the value of the term would be 2 regardless of the order of evaluation, and the term would not have any side-effects. However, the state can no longer be implemented by a (global) store. The state needs to be copied and passed to each argument of +, making the language quite inefficient.

The type system of ILC excludes such terms from the language by requiring that all state-manipulations be performed in a sequential fashion. Well-typed terms of ILC do not require the state to be copied, and hence the state can be implemented

[2]Technically speaking, observers are not continuations because they return values. But, they can be thought of as continuations in the imperative sublanguage so that the "answers" produced can then be consumed in the applicative sublanguage.

by a (global) store. The only legal way to express the above example in ILC, is to sequentialize its assignments. For example,

$$v := 0 \ ; \ \mathtt{get}\ x \Leftarrow v\ \mathtt{in}\ (v := 2 \ ; \ \mathtt{get}\ y \Leftarrow v\ \mathtt{in}\ (y+x))$$

is a well-typed term.

ILC distinguishes between state-dependent observers and applicative values. Both observers and values can be passed to functions and returned as results — it is not necessary to evaluate an observer to a value before passing it. In fact, an observer of the form ($\mathtt{get}\ x \Leftarrow l\ \mathtt{in}\ t$) is in head normal form, just as a lambda expression is in head normal form (see section 5.2). This is a form of laziness and, in fact, directly corresponds to Algol's call by name. However, an observer passed to a function can only be evaluated in a single state due to the single-threaded nature of the type system. So, the ambiguities caused by Algol's call-by-name are not shared by ILC.

Finally, we wish the type system to ensure that all recursion-free terms are strongly normalizable, i.e., their evaluation always terminates. We postpone a discussion of this issue to section 5.2. For now, we merely note that strong normalization is achieved by making observers non-storable values. If strong normalization is not considered critical, the θ and ω layers may be conflated.

The type system described thus far is overly restrictive. It prohibits all nonsequential combinations of observers in order to ensure that the state is never copied. For example, a term of the form

$$(v := 0 \ ; \ (\mathtt{get}\ x \Leftarrow v\ \mathtt{in}\ x) + (\mathtt{get}\ x \Leftarrow v\ \mathtt{in}\ x))$$

is excluded because the observer arguments of + are combined nonsequentially. However, this term does not require the state to be copied since the arguments of + do not locally modify the state. This suggests that the type system could be relaxed by distinguishing between:

- *creators*, that locally extend the state;
- *pure observers*, that observe the state without locally extending or modifying it; and
- *mutators*, that locally modify the state.

The type system could then permit certain state-dependent terms to be combined. For example, two pure observers could be safely combined. To be effective, such a solution would also have to incorporate a comprehensive type system that captures "effects" of expressions on "regions" of references (similar to that of FX [LG88]). This would permit combining mutators that mutate disjoint regions of the state. We do not explore this solution in this paper because it is orthogonal to the issues considered here. It is also clear that such a type system does not completely eliminate the need for sequencing.

5 Semantics of ILC

We present the denotational and operational semantics of ILC, and sketch the proofs of several important properties including soundness, strong normalization and confluence.

5.1 Denotational semantics

The denotational semantics is defined using complete partial orders (cpo's) as domains. For every primitive type β, choose a domain D_β. $D_{\tau\times\tau}$ and $D_{\tau\to\tau}$ are defined by the standard product and continuous function space constructions on cpo's.

For every reference type $\mathtt{Ref}\ \theta$, choose a countable flat domain $D_{\mathtt{Ref}\ \theta}$. The defined elements of a $D_{\mathtt{Ref}\ \theta}$ domain should be disjoint from those of any other such domain. The defined elements of these domains may be thought of as "locations". $\mathtt{State}$ is the set of partial mappings σ from $\bigcup_\theta D_{\mathtt{Ref}\ \theta}$ to $\bigcup_\theta D_\theta$ with the constraint that, whenever $\alpha \in D_{\mathtt{Ref}\ \theta}$, $\sigma(\alpha) \in D_\theta$ and $\sigma(\bot_{\mathtt{Ref}\ \theta}) = \bot_\theta$. The subset of $\bigcup_\theta D_{\mathtt{Ref}\ \theta}$ mapped by σ is denoted $dom(\sigma)$. σ_0 is the "empty" state, i.e., $dom(\sigma_0)$ contains only $\bot_{\mathtt{Ref}\ \theta}$ elements.

The domain for an observation type is $D_{\mathtt{Obs}\ \tau} = [\mathtt{State} \to D_\tau]$.

An environment η is a mapping from variables to $\bigcup_\omega D_\omega$. If Γ is a type assignment, we say that η satisfies Γ if $\eta(x) \in D_\omega$ for every $x{:}\,\omega \in \Gamma$, $\eta(v^*) \in D_{\mathtt{Ref}\ \theta}$ for every $v^*{:}\,\mathtt{Ref}\ \theta \in \Gamma$, and $\eta(v^*) \neq \eta(w^*)$ for every $v^*, w^*{:}\,\mathtt{Ref}\ \theta \in \Gamma$.

The denotational semantics of ILC (see figure 2) is defined by induction on type derivations. The meaning of an expression ($\Gamma \vdash e{:}\,\omega$) is a mapping from environments satisfying Γ to D_ω. (See [Mit90] for a discussion of this notation).

Lemma 1 $[\![\Gamma \vdash e{:}\,\omega]\!]$ *is well-defined.*

This involves showing that continuous functions in the interpretation of λ are unique and that the choice of α in the interpretation of $\mathtt{letref}$ is immaterial.

Proposition 2 $[\![\Gamma \vdash e{:}\,\omega]\!]\eta \in D_\omega$ *whenever η satisfies Γ.*

This is proved by a simple induction on type derivations. The main property to be verified is that η and σ are always extended or modified in a manner type-consistent with Γ. We present the proof case for

$$
\begin{array}{lcl}
[\![\Gamma \vdash x\colon \omega]\!]\,\eta & = & \eta x \\
[\![\Gamma \vdash (\lambda x\colon \omega_1 . e)\colon \omega_1 \to \omega_2]\!]\,\eta & = & \lambda v \in D_{\omega_1} . [\![\Gamma, x\colon \omega_1 \vdash e\colon \omega_2]\!]\,(\eta[x \to v]) \\
[\![\Gamma \vdash f(e)\colon \omega_2]\!]\,\eta & = & ([\![\Gamma \vdash f\colon \omega_1 \to \omega_2]\!]\,\eta)([\![\Gamma \vdash e\colon \omega_1]\!]\,\eta) \\
[\![\Gamma \vdash \langle e_1, e_2 \rangle\colon \omega_1 \times \omega_2]\!]\,\eta & = & \langle [\![\Gamma \vdash e_1\colon \omega_1]\!]\,\eta, [\![\Gamma \vdash e_2\colon \omega_2]\!]\,\eta \rangle \\
[\![\Gamma \vdash e.1\colon \omega_1]\!]\,\eta & = & \mathtt{fst}([\![\Gamma \vdash e\colon \omega_1 \times \omega_2]\!]\,\eta) \\
[\![\Gamma \vdash e.2\colon \omega_2]\!]\,\eta & = & \mathtt{snd}([\![\Gamma \vdash e\colon \omega_1 \times \omega_2]\!]\,\eta) \\
 & & \\
[\![\Gamma \vdash e\colon \tau]\!]\,\eta & = & [\![\Gamma \vdash e\colon \mathtt{Obs}\ \tau]\!]\,\eta\,\sigma_0 \\
[\![\Gamma \vdash e\colon \mathtt{Obs}\ \tau]\!]\,\eta & = & \lambda\sigma.[\![\Gamma \vdash e\colon \tau]\!]\,\eta \\
 & & \\
[\![\Gamma \vdash (\mathtt{letref}\ v^*\colon \mathtt{Ref}\ \theta := e\ \mathtt{in}\ t)\colon \mathtt{Obs}\ \tau]\!]\,\eta & = & \lambda\sigma.[\![\Gamma, v^*\colon \mathtt{Ref}\ \theta \vdash t\colon \mathtt{Obs}\ \tau]\!]\,(\eta[v^* \to \alpha])\,(\sigma[\alpha \to v_e]) \\
 & & \text{where } \alpha \text{ is any element of } D_{\mathtt{Ref}\ \theta} \text{ not in } \mathit{dom}(\sigma) \\
 & & \text{and } v_e = [\![\Gamma, v^*\colon \mathtt{Ref}\ \theta \vdash e\colon \theta]\!]\,(\eta[v^* \to \alpha]) \\
[\![\Gamma \vdash (\mathtt{get}\ x\colon \theta \Leftarrow l\ \mathtt{in}\ t)\colon \mathtt{Obs}\ \tau]\!]\,\eta & = & \lambda\sigma.[\![\Gamma, x\colon \theta \vdash t\colon \mathtt{Obs}\ \tau]\!](\eta[x \to \sigma([\![\Gamma \vdash l\colon \mathtt{Ref}\ \theta]\!]\eta)])\,\sigma \\
[\![\Gamma \vdash (l\colon = e; t)\colon \mathtt{Obs}\ \tau]\!]\,\eta & = & \lambda\sigma.[\![\Gamma \vdash t\colon \mathtt{Obs}\ \tau]\!]\,\eta\,(\sigma[v_l \to v_e]) \\
 & & \text{where } v_l = [\![\Gamma \vdash l\colon \mathtt{Ref}\ \theta]\!]\eta \text{ and } v_e = [\![\Gamma \vdash e\colon \theta]\!]\eta
\end{array}
$$

Figure 2: Denotational semantics

`letref` terms; other cases can be verified similarly. Assume that η satisfies Γ.

- $[\![\Gamma \vdash (\mathtt{letref}\ v^*\colon \mathtt{Ref}\ \theta := e\ \mathtt{in}\ t)\colon \mathtt{Obs}\ \tau]\!]\,\eta \in D_{\mathtt{Obs}\ \tau}$

  Let $\alpha \in D_{\mathtt{Ref}\ \theta}$. Then, since η satisfies Γ, $(\eta[v^* \to \alpha])$ satisfies $\Gamma, v^*\colon \mathtt{Ref}\ \theta$. Thus, by induction hypothesis, $[\![\Gamma, v^*\colon \mathtt{Ref}\ \theta \vdash t\colon \mathtt{Obs}\ \tau]\!]\,(\eta[v^* \to \alpha]) \in D_{\mathtt{Obs}\ \tau}$ and $v_e = [\![\Gamma, v^*\colon \mathtt{Ref}\ \theta \vdash e\colon \theta]\!]\,(\eta[v^* \to \alpha]) \in D_\theta$. Thus, $(\sigma[\alpha \to v_e])$ is a well-formed state, and hence $\lambda\sigma.[\![\Gamma, v^*\colon \mathtt{Ref}\ \theta \vdash t\colon \mathtt{Obs}\ \tau]\!]\,(\eta[v^* \to \alpha])\,(\sigma[\alpha \to v_e]) \in D_{\mathtt{Obs}\ \tau}$ as desired.

This property ensures that every expression of an applicative type τ is free of state information. It also shows that observers (of type `Obs` τ) do not have any visible side-effects. This proves our claim that ILC is free of side-effects.

We note that the semantics uses the state in a single-threaded fashion [Sch85]. Whenever a state is updated, the old state is discarded. Thus, the semantics can indeed be realized by a global store and no side effects need enter the implementation through the “back door”.

At this stage, we can also point out what kind of primitive constants of mutable and observer types may be added to the language without violating the basic framework. The acceptable constants should be purely *combinatorial*, i.e., they should not use any information about the semantic interpretations of their parameters. For example, the constants $\mathit{if}_{\mathtt{Obs}\ \tau}\colon \mathtt{Bool} \times \mathtt{Obs}\ \tau \times \mathtt{Obs}\ \tau \to \mathtt{Obs}\ \tau$ defined by

$$\mathit{if}_{\mathtt{Obs}\ \tau}(p, t_1, t_2) = \begin{cases} t_1, & \text{if } p = \mathit{true} \\ t_2, & \text{if } p = \mathit{false} \end{cases}$$

are acceptable because they are not dependent on the semantic interpretation of the `Obs` τ parameters. On the other hand, the constant $\mathit{add}\colon \mathtt{Obs}\ \tau \times \mathtt{Obs}\ \tau \to \mathtt{Obs}\ \tau$ defined by

$$\mathit{add}(t_1, t_2) = \lambda\sigma.\ (t_1\sigma + t_2\sigma)$$

is not acceptable as it interprets `Obs` τ parameters to be functions of type $[\mathtt{State} \to D_\tau]$.

5.2 Reduction semantics

We now present reduction rules for terms of ILC. These rules are meant to reduce terms to normal form such that every closed term of a primitive type β reduces to a constant of that type. Let $V(t)$ be the set of free variables of term t.

The reduction rules presented in figure 3 propagate `get` terms outward until they encounter a `letref` or assignment, and then discharge the `get` construct. The `letref` and assignment constructs can be discharged only after the state observation in their body is completed, i.e., after the body reduces to applicative term. At that stage, the body would be a “value term” of the form k, $\lambda x\colon \omega . e_1$ or $\langle e_1, e_2 \rangle$. Rules (3) and (6) handle this situation.

	Let $u ::= k \mid \lambda x{:}\,\omega.e_1 \mid \langle e_1, e_2\rangle$			
(1)	$(\lambda x{:}\,\omega.e_1)(e_2)$	$\longrightarrow$	$e_1[e_2/x]$	
(2)	$\langle e_1, e_2\rangle.i$	$\longrightarrow$	e_i	for $i = 1, 2$
(3)	$\mathtt{letref}\ v^*{:}\,\theta := e\ \mathtt{in}\ u$	$\longrightarrow$	u	if $v^* \notin V(u)$
(4)	$\mathtt{letref}\ v^*{:}\,\theta := e\ \mathtt{in}$ $\mathtt{get}\ x{:}\,\theta' \Leftarrow v^*\ \mathtt{in}\ t$	$\longrightarrow$	$\mathtt{letref}\ v^*{:}\,\theta := e\ \mathtt{in}\ t[e/x]$	
(5)	$\mathtt{letref}\ v^* : \theta := e\ \mathtt{in}$ $\mathtt{get}\ x{:}\,\theta' \Leftarrow w^*\ \mathtt{in}\ t$	$\longrightarrow$	$\mathtt{get}\ x'{:}\,\theta' \Leftarrow w^*\ \mathtt{in}$ $\mathtt{letref}\ v^* : \theta := e\ \mathtt{in}\ t[x'/x]$	if $v^* \neq w^*$ & $x' \notin V(e) \cup V(t)$
(6)	$v^* := e\ ;\ u$	$\longrightarrow$	u	
(7)	$v^* := e\ ;\ \mathtt{get}\ x{:}\,\theta \Leftarrow v^*\ \mathtt{in}\ t$	$\longrightarrow$	$v^* := e\ ;\ t[e/x]$	
(8)	$v^* := e\ ;\ \mathtt{get}\ x{:}\,\theta \Leftarrow w^*\ \mathtt{in}\ t$	$\longrightarrow$	$\mathtt{get}\ x'{:}\,\theta \Leftarrow w^*\ \mathtt{in}$ $v^* := e\ ;\ t[x'/x]$	if $v^* \neq w^*$ & $x' \notin V(e) \cup V(t)$

Figure 3: Reduction rules

Let $\overset{*}{\longrightarrow}$ be the reflexive, transitive closure of $\longrightarrow$. We can easily show that one step reduction preserves types, and by induction, so does $\overset{*}{\longrightarrow}$.

Lemma 3 (Type preservation) *If* $(\Gamma \vdash s{:}\,\omega)$ *is a term, and* $s \longrightarrow t$*, then* $(\Gamma \vdash t{:}\,\omega)$ *is a term.*

We can also show that one step reduction preserves meaning.

Proposition 4 (Soundness) *Let* $(\Gamma \vdash s{:}\,\omega)$ *and* $(\Gamma \vdash t{:}\,\omega)$ *be terms, and let* $s \longrightarrow t$*. Then*

$$[\![\Gamma \vdash s{:}\,\omega]\!]\,\eta \equiv [\![\Gamma \vdash t{:}\,\omega]\!]\,\eta$$

Proof: Rules (1) and (2) are classical. For (3) and (6), note that if u is of the stated form, it can only be a trivial observer which is also of an applicative type τ. (Nontrivial observers have a `letref`, `get` or ":=" at their principal position.) Hence, u is state-independent. For example,

$$\begin{aligned}
&[\![\Gamma \vdash (\mathtt{letref}\ v^*{:}\,\mathtt{Ref}\ \theta := e\ \mathtt{in}\ u){:}\,\mathtt{Obs}\ \tau]\!]\,\eta\\
&\quad= \lambda\sigma.[\![\Gamma, v^*{:}\,\mathtt{Ref}\ \theta \vdash u{:}\,\mathtt{Obs}\ \tau]\!](\eta[v^* \to \alpha])(\sigma[\alpha \to v_e])\\
&\qquad\quad \text{where } \alpha \text{ is any element of } D_{\mathtt{Ref}\ \theta} \text{ not in } \mathit{dom}(\sigma)\\
&\qquad\quad \text{and } v_e = [\![\Gamma, v^*{:}\,\mathtt{Ref}\ \theta \vdash e{:}\,\theta]\!]\ (\eta[v^* \to \alpha])\\
&\quad= \lambda\sigma.[\![\Gamma, v^*{:}\,\mathtt{Ref}\ \theta \vdash u{:}\,\tau]\!]\ \eta[v^* \to \alpha]\\
&\quad= \lambda\sigma.[\![\Gamma \vdash u{:}\,\tau]\!]\ \eta \qquad \text{since } v^* \notin V(u)\\
&\quad= [\![\Gamma \vdash u{:}\,\mathtt{Obs}\ \tau]\!]\ \eta
\end{aligned}$$

For (4) and (5), recall that v^* and w^* are "reference variables" which are only bound in letref constructs. By the denotational semantics, any two such variables denote distinct references unless they are syntactically identical. For example,

$$\begin{aligned}
&[\![\Gamma \vdash (\mathtt{letref}\ v^*{:}\,\mathtt{Ref}\ \theta := e\ \mathtt{in}\ (\mathtt{get}\ x{:}\,\theta \Leftarrow v^*\ \mathtt{in}\ t)){:}\,\mathtt{Obs}\ \tau]\!]\ \eta\\
&\quad= \lambda\sigma.[\![\Gamma, v^*{:}\,\mathtt{Ref}\ \theta \vdash (\mathtt{get}\ x{:}\,\theta \Leftarrow v^*\ \mathtt{in}\ t){:}\,\mathtt{Obs}\ \tau]\!](\eta[v^* \to \alpha])\\
&\qquad\qquad (\sigma[\alpha \to v_e])\\
&\qquad\quad \text{where } \alpha \text{ is any element of } D_{\mathtt{Ref}\ \theta} \text{ not in } \mathit{dom}(\sigma)\\
&\qquad\quad \text{and } v_e = [\![\Gamma, v^*{:}\,\mathtt{Ref}\ \theta \vdash e{:}\,\theta]\!]\ (\eta[v^* \to \alpha])\\
&\quad= \lambda\sigma.[\![\Gamma, v^*{:}\,\mathtt{Ref}\ \theta, x{:}\,\theta \vdash t{:}\,\mathtt{Obs}\ \tau]\!]\ \ (\eta[v^* \to \alpha][x \to v_e])\\
&\qquad\qquad (\sigma[\alpha \to v_e])\\
&\quad= \lambda\sigma.[\![\Gamma, v^*{:}\,\mathtt{Ref}\ \theta \vdash t[e/x]{:}\,\mathtt{Obs}\ \tau]\!]\ (\eta[v^* \to \alpha])(\sigma[\alpha \to v_e])\\
&\quad= [\![\Gamma \vdash (\mathtt{letref}\ v^*{:}\,\mathtt{Ref}\ \theta := e\ \mathtt{in}\ t[e/x]){:}\,\mathtt{Obs}\ \tau]\!]\ \eta
\end{aligned}$$

Rules (7) and (8) are similar. □

Strong normalization is considered to be a desirable property of typed programming languages. It asserts that the evaluation of a well-typed recursion-free term *always* terminates. Conceptually, its significance is that all terms are meaningful; there are no undefined terms [Pra71]. Its pragmatic implication is that non-termination is limited to explicit recursion. Strong normalization is ensured in ILC by making observers non-storable. If observers were storable, `Ref Obs nat` would be a well-formed type and the language would contain the following infinite reduction sequence:

$$\begin{aligned}
&(\mathtt{letref}\ u^* : \mathtt{Ref\ Obs}\ T := (u^* \uparrow)\ \mathtt{in}\ u^* \uparrow)\\
&\qquad \longrightarrow (\mathtt{letref}\ u^* : \mathtt{Ref\ Obs}\ T := (u^* \uparrow)\ \mathtt{in}\ u^* \uparrow)\\
&\qquad \longrightarrow \ldots
\end{aligned}$$

Since u^* : `Ref Obs nat`, we can store in it an observation of itself $(u^* \uparrow)$. Indeed, recursion is defined in Scheme by a similar device [RC86].

Proposition 5 (Strong Normalization) *Let* $(\Gamma \vdash t : \omega)$ *be a recursion-free term. Then there is no infinite reduction sequence* $t \longrightarrow t_1 \longrightarrow t_2 \longrightarrow \ldots$ *of well-typed* ILC *terms.*

Proof: The proof of the above proposition is quite elaborate and uses twin induction on types and terms, along the classical lines of [Tai75, GLT89]. The proof may be found in [Swa91].

The Church-Rosser property for the reduction system may be established as follows. Let V be a countably infinite set of reference variables. Treat the reduction rules (4) and (7) as schematic rules representing an infinite set of rules, one for each $v^* \in V$. Similarly, the rules (5) and (8) may be treated as being schematic for an infinite set of rules, one for each distinct pair of $v^*, w^* \in V$. The resulting reduction system has no "critical overlaps", i.e., no left hand side has a common instance with a subterm of another left hand side, unless the subterm is a metavariable. So, it follows that:

Lemma 6 (Local Confluence) *If* $(\Gamma \vdash r : \omega)$ *is a term, and if* $r \longrightarrow s_1$ *and* $r \longrightarrow s_2$*, then there is a term* $(\Gamma \vdash t : \omega)$ *such that* $s_1 \xrightarrow{*} t$ *and* $s_2 \xrightarrow{*} t$*.*

Hence, by Newman's Lemma, we have

Proposition 7 (Confluence) *If* $(\Gamma \vdash r : \omega)$ *is a term, and if* $r \xrightarrow{*} s_1$ *and* $r \xrightarrow{*} s_2$*, then there is a term* $(\Gamma \vdash t : \omega)$ *such that* $s_1 \xrightarrow{*} t$ *and* $s_2 \xrightarrow{*} t$*.*

This result can be extended to the language with recursion as follows. Add the following reduction rule

$$(9) \quad \text{fix}\, e \quad \longrightarrow \quad e(\text{fix}\, e)$$

where *fix* is the least fixed point operator for each type ω. The resulting system still has no critical overlaps. Further, it is left-linear, i.e., there are no repeated occurrences of metavariables on any left hand side. Hence, by Huet [Hue80], Lemma 3.3, we have

Proposition 8 *The reduction system (1-9) is confluent.*

This property, which is equivalent to the Church-Rosser property, gives further evidence of the side--effect-freedom of ILC. If there were side effects, then the evaluation of a subexpression would affect the meaning of its context, and the normal forms would be dependent on the evaluation order.

The independence of results on the evaluation order means, in particular, that call-by-value and call-by-name evaluations produce the same results. This observation must be interpreted carefully. In the lambda calculus setting, the distinction between call-by-value and call-by-name refers to when the arguments to a function are *evaluated.* We are using these terms in the same sense. However, in the imperative programming framework, the terms call-by-value and call-by-name are used to make a different distinction — the question of when the arguments to a function are *observed.* In the terminology of ILC, this involves a coercion from a type `Obs` τ to a type τ. Since `Obs` τ represents the function space [`State` $\rightarrow D_\tau$], such a coercion involves change of semantics. ILC permits no such coercion. Thus, in the imperative programming sense, ILC's parameter passing is call-by-name. However, the linearity of observer constructions means that a function accepting an observer can use it to observe at most one state. This contrasts with Algol 60, where a call-by-name parameter can be used to observe many states with quite unpredictable effects.

6 Extended Example : Unification

To illustrate the expressive power and usability of the language, we implement unification by an algorithm that performs *shared updates* on a data structure. Unification [Rob65] is a significant problem that finds applications in diverse problems including type inference, implementation of Prolog and theorem provers, and natural semantics.

Figure 4 contains an ILC program that computes the most general common instance of two terms t_1 and t_2. A *term* is either a variable or a pair denoting the application of a function symbol to a list of subterms. A variable is represented by a *reference.* The reference may contain either a term (if the variable is already bound), or the special value `Unbound`. This representation of terms illustrates the notion of *shared dynamic data* mentioned in Section 1; a function accepting a term has indirect access to all the references embedded in the term.

We assume the existence of a global reference called `sigma` that accumulates the list of references bound during an attempt at unification. If the unification is successful, this yields the most general unifier, while the representations of the terms t_1 and t_2 correspond to the most general common instance. On failure, this list is used to reset the values of the references to `Unbound`.

The function `unify` attempts to compute the most general common instance of two terms `t` and `u`. If the unification is successful, it instantiates both `t` and `u` to their most general common instance (by updating the references embedded in them), and evaluates the success continuation `sc`. If the unification is unsuccessful, it leaves the terms unchanged and evaluates the failure continuation `fc`. Internally, it uses the auxiliary function `unify-aux` which updates the

```
datatype  term  =  Var of Ref var
                |  Apply of (symbol × List term)

    and   var   =  Unbound | Bound of term

unify: term × term × Obs $T$ × Obs $T$ → Obs $T$
    = $\lambda(t, u, sc, fc)$. unify-aux($t, u, sc$, undo($fc$))

unify-aux: term × term × Obs $T$ × Obs $T$ → Obs $T$
    = $\lambda(t, u, sc, fc)$.
        case $(t, u)$ of
            (Var($v1$), Var($v1$)) ⇒ $sc$
          | (Var($v1$), Apply($f, ts$)) ⇒ bind($v1, u, sc, fc$)
          | (Apply($f, ts$), Var($v2$)) ⇒ bind($v2, t, sc, fc$)
          | (Apply($f, ts$), Apply($g, us$)) ⇒
                if $(f = g)$ then unify-lists($ts, us, sc, fc$) else $fc$

unify-lists: List term × List term × Obs $T$ × Obs $T$ → Obs $T$
    = $\lambda(lt, lu, sc, fc)$. case $(lt, lu)$ of
                                 ([], []) ⇒ $sc$
                               | ($t :: ts, u :: us$) ⇒
                                     unify-aux($t, u$, unify-lists($ts, us, sc, fc$), $fc$)
                               | (_, _) ⇒ $fc$

bind: Ref var × term × Obs $T$ × Obs $T$ → Obs $T$
    = $\lambda(v, u, sc, fc)$. case $v\uparrow$ of
                                Unbound ⇒ occurs($v, u, fc$, ($v$ := Bound($u$);
                                                              $sigma := v :: sigma\uparrow$;
                                                              $sc$))
                              | Bound($t$) ⇒ unify-aux($t, u, sc, fc$)

undo: Obs $T$ → Obs $T$
= $\lambda fc$. case $sigma\uparrow$ of
            [] ⇒ $fc$
          | $v :: vs$ ⇒  $v$ := Unbound;
                        $sigma := vs$;
                        undo($fc$)

occurs: Ref var × term × Obs $T$ × Obs $T$ → Obs $T$
    = ⋯
```

Figure 4: Unification of first-order terms

terms in both cases. By providing the failure continuation `(undo fc)` to this function, terms are restored to their original values upon failure. The function `unify-lists` unifies two lists of terms (`lt, lu`), `bind` unifies a variable `v` with a term `u`, `occurs` checks whether a variable `v` occurs free in a term `u`, and `undo` resets the values of variables that have been bound during a failed attempt at unification. The definition of `occurs` is straightforward and has been omitted.

The significant aspect of this program is that when an unbound variable is unified with a term that does not contain any free occurrence of the variable, unification succeeds by *assigning* the term to the reference that represents the variable (see function `bind`). This modification is visible via other access paths to the reference. It is this information sharing that affects the unification of subsequent subterms, even though no values are *passed* between these program points. In contrast, in a pure functional language, every computed value needs to be passed explicitly to all program points which need it. In the unification example, this means that whenever a variable is modified, the modified value needs to be passed to all other subterms that are yet to be unified, an expensive proposition indeed.

7 Related Work

In this section, we compare our research with related work. We organize this comparison based on the broad approach taken by the related work.

Linearity Substantial research has been devoted to determining when values of pure functional languages can be modified destructively rather than by copying. Guzman and Hudak [GH90] propose a typed extension of functional languages called *single threaded lambda calculus* that can express the sequencing constraints required for the in-place update of array-like data structures. Wadler [Wad90b] proposes a similar solution using types motivated by Girard's Linear Logic and, later, shows the two approaches to be equivalent [Wad91]. He also proposes an alternate solution inspired by monad comprehensions [Wad90a].

These approaches differ radically from ours in that they do not treat references as values. Programming is still done in the functional style (that is, using our τ types). Shared updates cannot be expressed, and *pointers* (references to references) and *objects* (mutable data structures with function components) are absent. Although it is possible to represent references as indices into an array called the store, the result is a low-level "Fortran-style" of programming, and it is not apparent how references of *different types* can be accommodated.

Continuation-based effects Our approach to incorporating state changes is closely related to (and inspired by) continuation-based input/output methods used in functional languages [HW90, Kar81, Per90, MH]. The early proposal of Haskell incorporated continuation-based I/O as a primitive mechanism, but Haskell version 1.0 defines it in terms of stream-based I/O [HW90, HS88]. Our `Obs` types are a generalization of the Haskell type `Dialog`. In ILC, `Dialog` can be defined as `Obs Unit` where `Unit` is a one-element type.

Effect systems An effect system of Gifford and Lucassen [GL86] is a type system that describes the side-effects that expressions can have. A compiler can then use this information to determine when expressions can be evaluated in parallel, or when they may be memoized without altering the meaning of the program. The side-effect information computed by Gifford and Lucassen assumes an eager order of evaluation; this contrasts with our goal of handling assignments in lazy languages.

Equational axiomatizations Felleisen [Fel88, FF87, FH89], Mason and Talcott [MT89a, MT89b] give equational calculi for untyped Scheme-like languages with side effects. The calculi are based on the notion of *observational equivalence*: two terms are equivalent if they yield the same result in all contexts of atomic type. Our reduction system bears some degree of similarity to these calculi. However, the calculi are considerably more complex than our reduction system because of the possibility of side effects. We are investigating the formal relationships between the different approaches.

Laws of programming In a recent paper, Hoare et. al. [HHJ+87] present an equational calculus for a simple imperative language without procedures. The equations can be oriented as reduction rules and used to normalize recursion-free command phrases. Our work is inspired, in part, by this equational calculus.

Algol-like languages In a series of papers [Rey81, Rey82], Reynolds describes a language framework called *Idealized Algol* which is later developed into the programming language *Forsythe* [Rey88]. Forsythe has a two-layered operational semantics: the reduction semantics of the typed lambda calculus, and a state transition semantics. The former expands procedure calls to (potentially infinite) normal forms, while the latter executes the commands that occur in the normal forms. Forsythe is based on the principle that the lambda calculus layer is independent of the state transition layer. In particular, references to functions are not permitted because assignments to such references would affect β-expansion.

In contrast, our operational semantics involves a single *unified* reduction system that includes both β-expansion and command execution. Therefore, Forsythe's restrictions do not appear in our formulation. At the level of terms, ILC contains an applicative sublanguage (in terms of τ types) which is absent in Forsythe. Further, ILC permits state-independent imperative terms to be coerced to applicative types thereby allowing functions that create and use local state. No similar coercion is available in Forsythe.

8 Conclusion

We have presented a formal basis for adding mutable references and assignments to applicative languages without violating the principle of referential transparency. This is achieved through a rich type system that distinguishes between state-dependent and state-independent expressions and sequentializes modifications to the state. The language possesses the desired properties of applicative languages such as strong normalization and confluence. At the same time, it allows the efficient encoding of state-oriented algorithms and linked data structures.

We hope this work forms the beginning of a systematic and disciplined integration of functional and imperative programming paradigms. Their differing strengths are orthogonal, but not conflicting. Much further work remains to be done regarding the approach presented here. The issues of polymorphism over mutable and observer types must be investigated. A complete equational calculus must be found for supporting formal reasoning. This, in turn, requires a formalization of the models of ILC and the development of proof methods like logical relations. The incorporation of an effect system and use of the monad comprehension notation would make the language more flexible and convenient to use. Finally, the issues of implementation need to be addressed.

Acknowledgements We thank John Gray, Jim Hook, Matthias Felleisen, Sam Kamin, Dave MacQueen, Ian Mason, John Ramsdell, John Reynolds, Peter Sestoft, Harald Sondergard, Carolyn Talcott, Satish Thatte and Phil Wadler for numerous discussions which led to vast improvements in our presentation.

References

[Fel88] M. Felleisen. lambda-v-cs: An extended lambda-calculus for scheme. In *ACM Symp. on LISP and Functional Programming*, 1988.

[FF87] M. Felleisen and D. P. Friedman. A calculus for assignments in higher-order languages. In *ACM Symp. on Principles of Programming Languages*, pages 314–325, 1987.

[FH89] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. Technical Report COMP TR89-100, Rice University, 1989.

[GH90] J.C. Guzman and P. Hudak. Single-threaded polymorphic lambda calculus. In *IEEE Symp. on Logic in Computer Science*, pages 333–343, 1990.

[GL86] D.K. Gifford and J.M. Lucassen. Integrating functional and imperative programming. In *ACM Symp. on LISP and Functional Programming*, pages 28–38, 1986.

[GLT89] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*. Cambridge University Press, 1989.

[HB85] P. Hudak and A. Bloss. The aggregate update problem in functional programming systems. In *ACM Symp. on Principles of Programming Languages*, pages 300–314, 1985.

[HHJ$^+$87] C. A. R. Hoare, I. J. Hayes, He Jifeng, C. C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sorensen, J. M. Spivey, and B. A. Sufrin. Laws of programming. *Communications of the ACM*, 30(8):672–686, August 1987.

[HS88] P. Hudak and R. Sundaresh. On the expressiveness of purely functional I/O systems. Technical Report YALEU/DCS/-RR665, Yale University, Dec 1988.

[Hue80] G. Huet. Confluent reductions: abstract properties and applications to term rewriting systems. *Journal of the ACM*, 27(4):797–821, October 1980. (Previous version in *Proc. Symp. Foundations of Computer Science*, Oct 1977).

[Hug90] J. Hughes. Why functional programming matters. In *Research Topics in Functional Programming*, Univ. of Texas at Austin Year of Programming Series, chapter 2, pages 17–42. Addison-Wesley, 1990.

[HW90] P. Hudak and P. Wadler (editors). Report on the programming language Haskell, A non-strict purely functional language (Version 1.0). Technical Report YALEU/DCS/RR-777, Dep. of Computer Sc., Yale University, Apr 1990.

[Kar81] K. Karlsson. Nebula, A functional operating system. Tech. report, Chalmers University, 1981.

[LG88] J.M. Lucassen and D.K. Gifford. Polymorphic effect systems. In *ACM Symp. on Principles of Programming Languages*, pages 47–57, 1988.

[MH] L. M. McLoughlin and S. Hayes. Interlanguage working from a pure functional language. Functional Programming mailing list, Nov 1988.

[Mit90] J. C. Mitchell. Type systems for programming languages. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*. North-Holland, Amsterdam, 1990. (also Report No. STAN-CS-89-1277, Department of Computer Science, Stanford University).

[MT89a] I. A. Mason and C. Talcott. Axiomatizing operational equivalence in the presence of side effects. In *IEEE Symp. on Logic in Computer Science*, pages 284–293. IEEE, 1989.

[MT89b] I. A. Mason and C. Talcott. A sound and complete axiomatization of operational equivalence between programs with memory. Technical Report STAN-CS-89-1250, Stanford University, 1989. (to appear in *Theoretical Computer Science*).

[MTH90] R. Milner, M. Tofte, and R. Harper. *The definition of Standard ML*. The MIT Press, Cambridge, Massachusetts, 1990.

[Per90] N. Perry. *The Implementation of Practical Functional Programming Languages*. PhD thesis, Imperial College of Science, Technology and Medicine, University of London, 1990.

[Pra71] D. Prawitz. Ideas and results in proof theory. In *Proc. Second Scandinavian Logic Symposium*, 1971.

[RC86] J. Rees and W. Clinger (editors). Revised^3 report on the algorithmic language scheme. *ACM SIGPLAN Notices*, 21(12):37–79, Dec 1986.

[Rey81] J. C. Reynolds. The essence of Algol. In J. W. de Bakker and J. C. van Vliet, editors, *Algorithmic Languages*, pages 345–372. North-Holland, 1981.

[Rey82] J. C. Reynolds. Idealized Algol and its specification logic. In Neel. D., editor, *Tools and Notions for Program Construction*, pages 121–161. Cambridge Univ. Press, 1982.

[Rey88] J.C. Reynolds. Preliminary design of the programming language Forsythe. Technical Report CMU-CS-88-159, Carnegie Mellon University, June 1988.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, 1965.

[Sch85] D. A. Schmidt. Detecting global variables in denotational specifications. *ACM Transactions on Programming Languages and Systems*, 7(2):299–310, Apr 1985.

[SR91] V. Swarup and U.S. Reddy. A logical view of assignments. In *Conf. on Constructivity in Computer Science*, 1991. (To appear).

[Sto77] J. E. Stoy. *Denotational Semantics: The Scott–Strachey Approach to Programming Language Theory*. MIT Press, 1977.

[SW74] C. Strachey and C. P. Wadsworth. Continuations - a mathematical semantics for handling full jumps. Tech. Monograph PRG-11, Programming Research Group, University of Oxford, 1974.

[Swa91] V. Swarup. *Type theoretic properties of assignments*. PhD thesis, University of Illinois at Urbana-Champaign, 1991. (To appear).

[Tai75] W. W. Tait. A realizability interpretation of the theory of species. In R. Parikh, editor, *Proceedings of Logic Colloquium*, volume 453 of *Lecture Notes in Mathematics*, pages 240–251. Springer, Berlin, 1975.

[Tof88] M. Tofte. *Operational semantics and polymorphic type inference.* PhD thesis, Edinburgh University, 1988. Available as Edinburgh Univ. Lab. for Foundations of Computer Science Technical Report ECS-LFCS-88-54.

[Wad90a] P. Wadler. Comprehending monads. In *ACM Symp. on LISP and Functional Programming*, 1990.

[Wad90b] P. Wadler. Linear types can change the world. In *IFIP Working Conf. on Programming Concepts and Methods*, Sea of Gallilee, Israel, Apr 1990.

[Wad91] P. Wadler. Is there a use for linear logic? In *Proc. ACM SIGPLAN Conf. on Partial Evaluation and Semantics-Based Program Manipulation*, New York, 1991. ACM. (SIGPLAN Notices, to appear).