Arrays in Imperative Lambda Calculus

Sam Kamin*, Uday S. Reddy
Computer Science Dept.
University of Illinois at Urbana-Champaign
Urbana, IL 61820
{kamin,reddy}@cs.uiuc.edu

June 15, 1992

Abstract

In recent work, Swarup, Reddy, and Ireland defined a formal system called Imperative Lambda Calculus to provide clean integration of functional and imperative programming styles. In this paper, we study the issues of array manipulation in this framework. It is shown that the unique features of the calculus allow one to express array algorithms using high-level abstractions that are not available in purely functional languages.

1 Introduction

One view of functional programming is that it is restricted to "value-oriented" computation. Values are, by definition, static (or eternal) and no dynamic quantities are permissible. This viewpoint makes it difficult to deal with large aggregates such as arrays. It also makes it difficult to model a variety of dynamic objects which are required in programming.

We take a more "positive" view towards functional programming. First of all, we value functional programming for the abstractions it offers in terms of higher-order functions and lazy data structures. Secondly, we value it for its mathematical properties: the naturality of its semantics, confluent reduction system, support for equational reasoning etc. This liberal view point allows us to extend functional programming with dynamic objects (called *references*) and state-dependent values (called *observers*) while preserving all the properties of functional programs mentioned above. A formal system called imperative lambda calculus (ILC) is presented in [13] along these lines. This system has the following important properties:

- It is a conservative extension of the typed λ-calculus, *i.e.*, all the constructs of the latter continue to have their usual meaning in ILC.
- ILC terms can be evaluated using a confluent reduction system which extends the reduction system of typed λ-calculus.

*Partial support received from NASA Grant NAG–1–613.

An important benefit of this approach is that the abstraction mechanisms of functional programming, *viz.*, higher-order functions and lazy evaluation, now become available with references and observers as well. So, imperative programming in ILC is much richer than that in conventional imperative languages like Pascal. It is also richer than value-oriented (purely functional) solutions to in-place updates such as [5, 15, 16].

Our notion of arrays is the following. An array is an indexed collection of references. Since references are dynamic objects, an array becomes a *dynamic structure*. One can define and use various operations to manipulate arrays such as extracting subarrays, transposing matrices *etc.*. All such operations map dynamic structures to other dynamic structures. Secondly, one can define operations to *alter* the structures themselves such as, *e.g.*, multiplying every component value by a scalar. While the structures are being altered, the other structure-level manipulations continue to make sense. The most important benefit of programming in ILC is that problem solutions can be decomposed into the two classes: structure-level manipulations and dynamic manipulations.

This paper is in five sections: (1) an overview of ILC as presented in [13]; (2) explanation of the sugared version of ILC used in Section 3; (3) examples of ILC, showing the power, as well as limitations, of the language in its present state; (4) comparisons of ILC with related work; and (5) conclusions concerning the future of ILC, especially about our current research on it.

2 ILC

ILC is an extension of typed λ-calculus with constructs to declare, dereference, and assign to, references[1]. Thus, its syntax is:

$$
\begin{aligned}
e \;::=\;& \mathtt{k} \mid \mathtt{x} \mid \lambda\mathtt{x}{:}\tau.e \mid e_1(e_2)\\
\mid\;& \mathtt{letref\ r{:}Ref}\ \theta\ \mathtt{:=}\ e_1\ \mathtt{in}\ e_2\\
\mid\;& \mathtt{get\ x{:}}\theta\ \mathtt{<=}\ e_1\ \mathtt{in}\ e_2\\
\mid\;& e_0\ \mathtt{:=}\ e_1\mathtt{;}\ e_2
\end{aligned}
$$

subject to the typing rules presented below. To explain the new constructs: `letref` introduces and initializes a reference variable `r`; `get` dereferences a reference value $\mathbf{e}_1$ and assigns the value to `x`; and assignment modifies the referent of the reference value e_0 before evaluating e_2.

For example,

```
(1)    letref r:Ref int := 1 in
         (r:=2; get x:int <= r in x+1)
```

has value 3.

With no other restrictions, this language would, of course, be a non-confluent mess. For instance,

```
(2)    letref r:Ref int := 0 in
         (r:=1; get x <= r in x) + (get y <= r in y)
```

could have value 1 or 2. This is just the reason that adding assignment to functional languages is difficult.

[1]ILC also has pairing, but we omit it in this abstract.

$$
\begin{array}{ll}
\textit{Obs-intro} & \textit{Obs-elim} \\
\dfrac{\Gamma \vdash t : \tau}{\Gamma \vdash t : \mathrm{Obs}\ \tau} & \dfrac{\Gamma \vdash t : \texttt{Obs}\ \tau}{\Gamma \vdash t : \tau}\ \text{(if Γ has only τ types)}
\end{array}
$$

$$
\textit{Creation} \quad \dfrac{\Gamma, r : \texttt{Ref}\ \omega \vdash e : \omega \quad \Gamma, r : \texttt{Ref}\ \omega \vdash t : \texttt{Obs}\ \tau}{\Gamma \vdash (\texttt{letref}\ r\texttt{:Ref}\ \omega\ \texttt{:=}\ e\ \texttt{in}\ t) : \texttt{Obs}\ \tau}
$$

$$
\textit{Dereference} \quad \dfrac{\Gamma \vdash l : \texttt{Ref}\ \omega \quad \Gamma, x : \omega \vdash t : \texttt{Obs}\ \tau}{\Gamma \vdash (\texttt{get}\ x\texttt{:}\omega\ \texttt{<=}\ l\ \texttt{in}\ t) : \texttt{Obs}\ \tau}
$$

$$
\textit{Assignment} \quad \dfrac{\Gamma \vdash l : \texttt{Ref}\ \omega \quad \Gamma \vdash e : \omega \quad \Gamma \vdash t : \texttt{Obs}\ \tau}{\Gamma \vdash (l\ \texttt{:=}\ e\texttt{;}\ t) : \texttt{Obs}\ \tau}
$$

Figure 1: Type inference rules

The type system ensures that such potentially non-deterministic expressions are illegal. (In [13], a confluent and strongly normalizing set of reduction rules is given.) We must first describe the set of types, which is divided into two parts[2], *applicative* types (normal, applicative values) and *observer* types (values that depend on the state):

$$
\begin{array}{ll}
\text{Applicative types:} & \tau ::= \beta \mid \tau_1 \to \tau_2 \\
\text{Observer types:} & \omega ::= \tau \mid \texttt{Obs}\ \tau \mid \texttt{Ref}\ \omega \mid \omega_1 \to \omega_2
\end{array}
$$

β stands for some primtive types, like `int`; thus, τ represents the usual applicative types. The type `Ref` ω is a reference to a value of type ω; note that we can have references to references, so can build arbitrary pointer structures. `Obs` τ is the type of values that observe the state and return a value of (applicative) type τ; you can think of such values as functions in State $\to \tau$.

Figure 1 contains those inference rules that relate to `Obs` and `Ref` types and the new expressions; the typing rules for the λ-calculus fragment of ILC are exactly as in the typed λ-calculus. In Figure 1, r and x are variables, e, l, and t expressions.

Expression (1) can be typed as follows (for type-setting purposes, we omit occurrences of hypothesis `r:Ref int` $\vdash$ `r:Ref int`):

$$
\dfrac{\dfrac{\texttt{r:Ref int} \vdash \texttt{1:int} \quad \dfrac{\texttt{r:Ref int} \vdash \texttt{2:int} \quad \dfrac{\dfrac{\texttt{r:Ref int, x:int} \vdash \texttt{x+1:int}}{\texttt{r:Ref int, x:int} \vdash \texttt{x+1:Obs int}}}{\texttt{r:Ref int} \vdash \texttt{get x<=r in x+1:Obs int}}}{\texttt{r:Ref int} \vdash \texttt{r:=2; get x<=r in x+1:Obs int}}}{\vdash \texttt{letref r:Ref int := 1 in (r:=2; get x <= r in x+1):Obs int}}}{\vdash \texttt{letref r:Ref int := 1 in (r:=2; get x <= r in x+1):int}}
$$

[2] In [13], there were three parts, but we've combined two of them here; in this version, the system does not have the strong normalization property, but it is still confluent.

Notice how this expression observes the state *internally*, but from the outside is not considered an observer.

On the other hand, expression (2) is clearly not typable: the outer `letref` will introduce the assumption `r:Ref int`, so that each of the summands in its body will be typed as: `r:Ref int` $\vdash$ `...:Obs int`. With the "`Ref int`" assumption, *Obs-elim* is not applicable and since `+` takes `int`, rather than `Obs int`, arguments, the sum cannot be typed.

There are no rules specifically for arrays. We can view arrays simply as functions from indices to references:

$$\alpha\,\texttt{Array} \;=\; \texttt{int} \rightarrow \texttt{Ref}\ \alpha$$

As an example, we can write a `swap` routine as:

```
swap! a i j = λk. get x <= a i
                  in get y <= a j
                     in a i := y; a j := x; k
```

`swap! a i j` is a function from observers to observers. Given an observer k, it swaps the ith and jth elements of the array a and invokes k. The effect of swapping is thus only observable inside k. In general, the effects of assignments are localized to specific observers. As discussed in [13], this plays a large role in obtaining a semantically clean language.

Before proceeding to technical matters, we feel a few words are in order concerning the vexed question of "referential transparency." It is often said that functional languages *are* referentially transparent, imperative languages *aren't*, and adding dynamic values such as references to a functional language renders it no longer referentially transparent, a view with which we are not altogether in accord.

Let us stipulate for purposes of this discussion that the term "referentially transparent" has the meaning given by Quine [10]: A language is referentially transparent if for all terms t and u, and all contexts $C[\cdot]$, $t \equiv u$ implies $C[t] \equiv C[u]$. This definition does not seem to provide a basis for a challenge of any kind: if equivalence is given by its denotational semantics, the referential transparency of a language is an immediate consequence of the compositionality of its semantic definition.

Where, then, is the argument? It is here: when functional languages are said to be referentially transparent, while imperative languages are not, there is an implicit assumption about the nature of the semantics of those two classes of languages. Functional languages have a "simple semantics" with no side effects, and thus a simple equality relation; for example,

$$\texttt{let x=e in (x, x)} \equiv \texttt{(e, e)}.$$

Imperative languages have complex semantics with state, rendering some "obvious" equivalences, like the one just given, untrue.

In ILC, the logjam is broken by using types to distinguish between static and state-dependent values. An applicative type such as `int` contains static values and the corresponding observer type `Obs int` contains state-dependent values. No dynamic value can ever "masquerade" as a static one; it is this masquerading that, we believe, is the real source of the argument over referential transparency. The above equivalence holds in ILC just as all other equivalences of lambda calculus. Moreover, a new set of equivalences holds for state-dependent values, as documented in [13]. Thus, ILC is referentially transparent, not only in the trivial sense mentioned above, but also in that its equivalences are what one expects to see.

3 ILC in practice

Pure ILC is somewhat cumbersome to use in practice. In this section, we define some straightforward extensions motivated by practical considerations.

As already pointed out, an observer of type Obsτ can be thought of as a function in State $\rightarrow \tau$. The notion of an "effect" is then as a modifier, *i.e.*, a function, of observers. This is essentially the meaning of a "command" in the continuation semantics of imperative languages [4]. So, we define a new primitive type `cmd` with the (polymorphic) semantics:

$$\texttt{cmd = } \forall\tau.\ \texttt{Obs}\ \tau \rightarrow \texttt{Obs}\ \tau$$

Some primitive operations on commands are:

```
skip : cmd
_:=_ : Ref ω * ω → cmd
_;_ : cmd * Obs τ → Obs τ    /* associtites to right */
_○_ : cmd * cmd → cmd

skip      ≡   λk. k
x := y    ≡   λk. x := y; k
c;k       ≡   c k
c1 ○ c2   ≡   λk. c1 (c2 k)
```

As the name implies, `skip` is the empty command, `x := y` is the assignment command, `c;k` applies the command `c` to observer `k`, and `c1 ○ c2` is the sequential composition of commands. Note that the pure ILC construct $e_1 := e_2; k$ can now be parsed as a command $e_1 := e_2$ applied to an observer k. But, its meaning obtained by this parse as the same as the original meaning. The ";" operator is essentially function application, but it associates to the right. So, $\texttt{c}_1\texttt{;c}_2\texttt{;k}$ may be thought of as executing $\texttt{c}_1$, $\texttt{c}_2$, and `k`, in that order, to produce a result.

These facilities allow us to use a "combinatorial" style of programming at the level of commands instead of going down to the level of observers. For example, we can define a function denoting "for loops" as follows:

```
(* for : int → int → (int → cmd) → cmd *)
for i j c = if i > j then skip
                else (c i) ○ (for (i+1) j c)
```

We also define the command-level analogues of `letref` and `get` constructs with the following semantics:

```
letref-cmd v:Ref θ := e in c   ≡   λk. letref v:Ref θ := e in c k
get-cmd x:θ <= e in c          ≡   λk. get x:θ <= e in c k
```

The only difference between these and the original constructs is that we have a command `c` rather than an observer as the body. We also allow multiple `letref-cmds` and `get-cmds` to be cascaded. For example, the `swap` routine of the previous section can now be defined as

```
swap! a i j = get-cmd x <= a i
                  and y <= a j
              in (a i := y) ○ (a j := x)
```

4 Examples

This section describes an ILC function for LU-decomposition. Before presenting it, we need to define two data types, `Matrix` and `Vector`[3]:

```
type Matrix = int * int * (int -> int -> real ref)
type Vector = int * (int -> real ref)
```

The matrix (r, c, f) has r rows and c columns, each indexed from 1; fij is the reference at the $(i, j)^{th}$ location. Note that `Matrix` and `Vector` are (indexed) collections of *references*, not of numbers.

The simplest operations on `Matrix` are destructors:

```
(* rows: Matrix -> int
   cols: Matrix -> int
   matrix_ref: Matrix -> int -> int -> real ref *)
fun rows (r, _, _) = r
fun cols (_, c, _) = c
fun matrix_ref (_, _, f) i j = f i j
```

and similarly for `Vector`:

```
(* length: Vector -> int
   vector_ref: Vector -> int -> real ref *)
fun length (r, _) = r
fun vector_ref (_, f) i = f i
```

More interesting are the functions that select rows and submatrices from matrices. `row m i` is the `Vector` that is the $\mathtt{i}^{th}$ row of `m`:

```
(* row: Matrix -> int -> Vector *)
fun row (_, c, f) i = (c, fn j => f i j)
```

and `subVector V lo` is the suffix of vector `V` going from element `lo` to the end:

```
(* subVector: Vector -> int -> Vector *)
subVector (r, f) lo = (r-lo+1, fn i => f (i+lo-1))
```

The most complicated of these operations is `subMatrix`. `subMatrix m (i1,i2) (j1,j2)` represents the rectangular submatrix of `m` whose upper left corner is point $(\mathtt{i1}, \mathtt{i2})$ and whose lower right corner is $(\mathtt{j1}, \mathtt{j2})$:

```
(* subMatrix: Matrix -> (int * int) -> (int * int) -> Matrix *)
fun subMatrix (r, c, f) (i1, i2) (j1, j2)
   = let newr = j1-i1+1 and newc = j2-i2+1
      in (newr, newc, fn (k1, k2) => f (k1+i1-1) (k2+i2-1))
```

[3] In this section, we use ML-like notation to enhance readability.

4.1 LU-decomposition

LU-decomposition is naturally described as a recursive process:

$$LUD(m) = \begin{cases} m, & \text{if } |m| = 1 \\ \begin{array}{|c|c|}\hline \multicolumn{2}{|c|}{} \\ \hline & m'_1 \\ \hline \end{array}, & \text{if } \begin{array}{|c|c|}\hline \multicolumn{2}{|c|}{} \\ \hline & m_1 \\ \hline \end{array} = \mathit{eliminate}(m) \text{ and } m'_1 = LUD(m_1) \end{cases}$$

eliminate is the following function on matrices, where the m_i are the rows of the matrix:

$$\mathit{eliminate}\begin{pmatrix} m_1 \\ m_2 \\ \vdots \\ m_n \end{pmatrix} = \begin{pmatrix} m_1 \\ m_2 \ominus m_1 \\ \vdots \\ m_n \ominus m_1 \end{pmatrix}$$

and $\ominus$ is the following operation on rows:

$$(x_1, x_2, \ldots, x_k) \ominus (y_1, y_2, \ldots, y_k) = (m, y_2 - x_2 m, \ldots, y_k - x_k m), \text{where } m = \frac{x_1}{y_1}$$

Our coding of the problem follows this quite directly, but note that when the sub-matrix m_1 is decomposed *in place*, the recursive structure of LUD becomes *tail*-recursive, so that we can use a loop:

```
(* LUD: Matrix -> cmd *)
fun LUD (m as (r, c, f)) =
   for 1 (r-1) (fn i => eliminate (subMatrix m i i c r))
```

The main benefits of the functional style we employ lie in the handling of submatrices. The code for the *eliminate* function also illustrates this, here selecting subrows of the rows of m on which to perform the $\ominus$ operation:

```
(* eliminate: Matrix -> cmd *)
fun eliminate m =
     for 2 (rows m)
        (fn j =>
            let row1 = row m 1
            and rowj = row m j
            in get-cmd m11 <= vector_ref row1 1
                   and mj1 <= vector_ref rowj 1
               in let mult = mj1/m11
                   in (vector_ref rowj 1 := mult) ∘
                      (vector-update
                          (fn (x, y) => x-y*mult)
                          (subVector rowj 2)
                          (subVector row1 2)))
```

The call to `vector-update` performs most of the work of $\ominus$ above, but does so destructively. It is a useful function for performing an operation on two vectors with one receiving the result:

```
(* vector-update: (real * real -> real) -> Vector -> Vector -> cmd *)
fun vector-update f v1 v2 =
   for 1 (length v1)
       (fn i => get-cmd v1i <= vector_ref v1 i
                    and v2i <= vector_ref v2 i
                in (vector_ref v1 i) := f (v1i, v2i))
```

The `get-cmd` dereferences `(vector_ref v1 i)` and `(vector_ref v2 i)`, binds the corresponding values to `v1i` and `v2i`, then does the assignment. Notice that, since `vector_ref` returns a reference, which is an assignable object, there is no need for a separate element-modification operation.

4.2 Discussion

The most important difference between the above treatment of LU decomposition and a solution based on a purely functional paradigm is the following: we model arrays as indexed collections of *references* whereas a functional solution models them as indexed collections of *values*. A reference is a dynamic object; more precisely, it is the *name* of a dynamic object. While values referred to by such names change from state to state, the names themselves remain constant. ILC gives us the vocabulary to define a whole new set of operations at the level of such names which are impossible to express in functional languages. For example, `row` and `subVector` extract subregions of a matrix and such regions can be operated upon by the generic operation `vector-update`. All these operations can be used in any state.

In contrast, a functional (or value-oriented) solution would have to deal with each state separately. After extracting parts of a value matrix and operating on them, one would have to put the new values of the parts back to produce a new state of the matrix. For example, here is a definition of `LUD` in Haskell:

```
LUD m = if (bounds m) = (1,1) then m
        else let {m' = eliminate m;
                  m1 = copySubMatrix m' (2,2);
                  m1' = LUD m1}
             in array (bounds m)
                       [(1,j) := m' ! (1,j) | j <- [1..n]]
                    ++ [(i,1) := m' ! (i,1) | i <- [2..n]]
                    ++ [(i,j) := m1' ! (i,j) | i,j <- [2..n]]
```

Thus, the argument can be made that the imperative version of this code is actually simpler than the purely functional.

However, the truer comparison is with a "single-threaded" functional language, like those in [5, 15, 16]. We take the single-threaded lambda calculus, λ_{st} [5], as a typical example.

There are two major differences between the treatment of LUD in λ_{st} versus ILC. First, there is no type "cmd" in λ_{st} — only the normal functional types exist (sometimes decorated to indicate single-threadedness, but not essentially changed). Second, there is no such thing as a subarray, or row, *per se*. Thus, any procedure that operates on part of an array must be given, Fortran-style, the entire array plus the indices descrbing the part of the array that is of interest. Thus, `for` and `LUD` would need to be defined like this:

```
(* for: int -> int -> (int -> α -> α) -> α -> α *)
fun for i n f s = if i>n then s else for (i+1) n f (f i s)

(* LUD: Matrix! -> Matrix! *)
val LUD = for 1 n eliminate
```

`eliminate` is given a large matrix and the indices for the subarray to be eliminated, rather than simply an array. This is unfortunate, but not very compelling. The change in `eliminate` itself is more serious. The obvious transposition of our code into λ_{st} is not legal:

```
(* eliminate: int -> Matrix! -> Matrix! *)
fun eliminate i m =
     for (i+1) (rows m)
        (fn j => fn m =>
           let mj1 = lookup m j i
           and m11 = lookup m i i
           and mult = mj1/m11
           in let* m' = update! m j i mult
              in vector-update (fn (x,y)=> x-y*mult)
                    (subVector-row m' j (i+1))
                    (subVector-row m' i (i+1))
                    (updatable-subVector-row m' j (i+1))
        )
        m
```

(Note that we need to pass to `vector-update` two kinds of arguments, those that index into an array and those that update an array.)

The difficulty is that we can give no useful definition to the function `updatable-subVector-row`. What we would like to say to allow the definition of `vector-update` is:

```
fun updatable-subVector-row m j k = fn i => fn x => update! m j (i+k) x
```

But this can't be type-checked in λ_{st} because "it [is not] permissible for a function to 'capture' a mutation to one of its free variables" [5].

Our only choice is to replace the call to `vector-update` by:

```
for (i+1) (cols m')
  (fn k => fn m =>
      let* v1i = m j k
      and v2i = m i k
      in update! m j k (v1i-v2i*mult))
  m'
```

The use of `vector-update` in ILC gave a pleasing modularity to the code and allowed it to mimic the mathematical function $\ominus$ directly (while updating in place). In contrast, the λ_{st} solution breaks down modularity and distributes the index manipulation throughout the code. Thus, single threading seems to compromise the very values function programming espouses, viz., functional abstraction and lazy evaluation, while ILC preserves them.

5 Related work

Concerning the general problem of destructive update in functional languages, reference [13] contained a number of comparisons of ILC with previous work. Here we recapitulate those comparisons only briefly, while adding some observations specific to the problem of arrays.

The phrase "adding arrays to a functional language" means different things to different people. The basic fault line is this: are we trying to obtain only the *efficiency* of destructively-updated arrays, or do we want their *expressiveness* (with their efficiency presumably following)?

Efficiency. Most work in this area has started from the assumption that the functional programming style is uniformly preferable to the imperative style, and the only problem is how to get the efficiency associated with destructive update. Thus, functional programs are annotated, or type-checked, or statically analyzed, to reveal "single-threadedness," but the programs do not basically differ from what one would write in a language like HASKELL [7], with *non*-destructively updated arrays. It follows that none of these approaches admits programs like our `LUD`, whose tail-recursive structure is possible only because of destructive update.

Implicit update. Much work has been directed at detecting single-threadedness in the absence of any information from the programmer. These efforts include [1, 6, 8, 12]. A serious practical problem is the unreliability of such methods. Sophisticated static analyses are easily fooled, so that a small change in a program can have an unexpectedly dramatic effect on its performance.

Linear logic-inspired systems. Though Wadler [15, 16] does not specifically address arrays, his linear logic type system is obviously applicable to them. (It has been further explored by Wakeling and Runciman [17].) Here, the type checker guarantees single-threadedness. However, the programmer never specifically requests a destructive update, so it is up to the compiler to determine when it is appropriate. For example, in Wadler's destructive `append!` example in [15], *both* arguments of `append!` are single-threaded; it is "obvious" which one should actually be modified, but it is far from clear how intelligent a compiler would be needed to sort this out in general.

In Guzmán and Hudak's single-threaded λ-calculus (λ_{st}) [5], the programmer explicitly requests destructive update of an array by writing `update!` instead of `update` (and `let*` instead of `let`); the type checker checks that the use of these destructive operations is sensible. However, these are still just *annotations* of a functional program: remove the "`!`" and "`*`", and you have a functional program that will give exactly the same answers, albeit more slowly. So, again, our `LUD` and `destructive-op` could not be written in λ_{st}. Furthermore, the type rules of λ_{st} essentially forbid the capture of arrays in closures if the closure can destructively update the array; thus, our entire style of programming by "decomposing in place" is not supported.

Expressiveness. Two paradigms which go beyond the pure functional style are data flow languages [9, 2] and higher-order imperative languages, most notably, Reynolds's Forsythe [11].

I-structures [2] are an array-like data structure used in dataflow languages. They allow *once-only* assignment to each component. In [2], several problems are mentioned whose solution in a purely functional style is difficult. These can be solved in ILC just as they are solved there. The problems mentioned there for which I-structures are not appropriate—histograms, for example—can also be solved in ILC. Recently, the data flow paradigm has been extended with mutable data structures [3]. The resulting language has the expressive power of ILC, but it is nondeterministic and has the flavor of a "concurrent" programming language as opposed to a functional language.

Reynolds's Forsythe [11], apart from its novel conjunctive type system, is very close in spirit to our work. However, it seems that Forsythe is really meant to be an imperative language (with a functional "architecture") whereas ILC is meant to support both imperative and functional paradigms. Thus, many restrictions appearing in Forsythe are removed in ILC. For instance, references are first-class values and functions themselves are storable. More significantly, ILC allows computations which create and use state internally to be viewed as functional computations from the outside. This is an important requirement for a smooth integration of functional and imperative styles.

6 Conclusions

The systems of Wadler [15, 16] and Guzmán and Hudak [5] do one thing that ILC doesn't: they distinguish between *reading from* and *writing to* the state. In ILC, a state "observer" may do either, and no distinction is made. Thus, two dereferencing operations must be sequentialized, which obviously should not be necessary. If ILC were developed to include the notion of "pure observers," programming in it would be more convenient.

The uses of monads, as advocated by Wadler [14]. are really orthogonal to the problem of side effects, as can be seen from the wide variety of applications displayed in [14]. They provide notational convenience, whereby an array can be implicitly manipulated by a program, or one can implicitly use continuation-passing style. It is especially from this latter use that ILC might benefit. Indeed, our syntactic sugar of section 3 provides precisely the same advantages, but there are cases where it breaks down and monads would allow the programmer to avoid some of the "continuation hacking."

Finally, the implementation of ILC, even for sequential machines, is far from a trivial matter, for several reasons. As with any lazy evaluation language, strictness analysis is required for a good implementation. However, it is more difficult because of the presence of references.

For the parallel case, there would seem to be reason for optimism about the ability of ILC to produce efficient code. The argument usually made for functional languages as good parallel languages is that programs are free of spurious control dependencies. Though ILC appears to reintroduce control operations, the new dependencies are not necessarily "spurious"— they are introduced to allow for destructive updates which the programmer considers worthwhile. The integration of functional and imperative styles in ILC allows us to use state-oriented sequential computation in a controlled way. On the other hand, ILC allows for excellent control over *data placement*, a critically important issue in parallel programming. For this, additional primitives must be provided for array allocation in multiple memories. As with all the topics mentioned in this section, this is something we are studying at present.

References

[1] S. Anderson and P. Hudak. Compilation of haskell array comprehensions for scientific computing. In *SIGPLAN Conference on Design and Implementation of Programming Languages*, 1990.

[2] Arvind, R.S. Nikhil, and K.K. Pingali. I-structures: Data structures for parallel computing. *ACM Transactions on Programming Languages and Systems*, 11(4):598–632, October 1989.

[3] P. S. Barth, R. S. Nikhil, and Arvind. M-structures: Extending a parallel, non-strict, functional language with state. In R. J. M. Hughes, editor, *Conf. on Functional Program. Lang. and Comput. Arch.*, pages 538–568. Springer-Verlag, Berlin, 1991. (LNCS Vol. 523).

[4] M. J. C. Gordon. *The Denotational Description of programming languages.* Springer-Verlag, New York, 1979.

[5] J.C. Guzman and P. Hudak. Single-threaded polymorphic lambda calculus. In *Fifth Ann. Symp. on Logic in Comp. Science.* IEEE Computer Society, 1990.

[6] P. Hudak and A. Bloss. The aggregate update problem in functional programming systems. In *ACM Symp. on Princ. of Program. Lang.*, pages 300–314, 1985.

[7] P. Hudak and P. Wadler (eds). Report on programming language Haskell, A non-strict purely functional language (Version 1.0). Technical Report YALEU/DCS/RR777, Yale University, Apr 1990.

[8] Paul Hudak. A semantics model of reference counting and its abstraction. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, pages 45–62. Ellis Horwood Ltd., London, 1987.

[9] R.S. Nikhil, K. Pingali, and Arvind. Id nouveau. Technical Report CSG 265, MIT, 1986.

[10] W. V. O. Quine. *Word and Object.* MIT Press, 1960.

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

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

[13] V. Swarup, U. S. Reddy, and E. Ireland. Assignments for applicative languages. In R. J. M. Hughes, editor, *Conf. on Functional Program. Lang. and Comput. Arch.*, pages 192–214. Springer-Verlag, Berlin, 1991. (LNCS Vol. 523).

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

[15] P. Wadler. Linear types can change the world. In M. Broy and C. B. Jones, editors, *Programming Concepts and Methods*. North-Holland, Amsterdam, 1990. (Proc. IFIP TC 2 Working Conf., Sea of Galilee, Israel).

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

[17] D. Wakeling and C. Runciman. Linearity and laziness. In R. J. M. Hughes, editor, *Conf. on Functional Program. Lang. and Comput. Arch.*, pages 215–240. Springer-Verlag, Berlin, 1991. (LNCS Vol. 523).