Linear types can change the world!

Philip Wadler
University of Glasgow*

Abstract

The linear logic of J.-Y. Girard suggests a new type system for functional languages, one which supports operations that "change the world". Values belonging to a linear type must be used exactly once: like the world, they cannot be duplicated or destroyed. Such values require no reference counting or garbage collection, and safely admit destructive array update. Linear types extend Schmidt's notion of single threading; provide an alternative to Hudak and Bloss' update analysis; and offer a practical complement to Lafont and Holmström's elegant linear languages.

An old canard against functional languages is that they cannot change the world: they do not "naturally" cope with changes of state, such as altering a location in memory, changing a pixel on a display, or sensing when a key is pressed.

As a prototypical example of this, consider the world as an array. An array (of type Arr) is a mapping from indices (of type Ix) to values (of type Val). For instance, the world might be a mapping of variable names to values, or file names to contents. At any time, we can do one of two things to the world: find the value associated with an index, or update an index to be associated with a new value.

Of course it is possible to model this functionally; we just use the two operations

$$\begin{aligned} \mathit{lookup} &: Ix \to Arr \to Val, \\ \mathit{update} &: Ix \to Val \to Arr \to Arr. \end{aligned}$$

A program that interacts with the world might have the form

$$\mathit{main} \;:\; Args \to Arr \to Arr,$$

where the first parameter is the list of arguments that make up the command line, the second parameter is the old world, and the result is the new world. An example of a program is

$$\mathit{main}\ \mathit{files}\ a \;=\; \mathit{update}\ \text{“stdout”}\ (\mathit{concat}\ [\mathit{lookup}\ i\ a \mid i \leftarrow \mathit{files}])\ a.$$

*Author's address: Department of Computing Science, University of Glasgow, G12 8QQ, Scotland. Electronic mail: `wadler@cs.glasgow.ac.uk`.

Presented at *IFIP TC 2 Working Conference on Programming Concepts and Methods*, Sea of Galilee, Israel, April 1990; published in M. Broy and C. B. Jones, editors, *Programming Concepts and Methods*, North Holland, 1990.

This performs the same operation as "cat" in Unix: it writes to the file "stdout" the result of concatenating the contents of each file named in the list *files*. (The example uses a list comprehension notation familiar from languages such as Miranda[1] [Tur85] or Haskell [HW88].)

So the canard is indeed a lie. But it is not completely without basis, because there is something unsatisfactory with this way of modelling the world. Namely, it allows operations that duplicate the world:

$$copy\ a = (a, a),$$

or that discard it:

$$kill\ a = ().$$

Neither of these correspond to our intuitive understanding of "the world": there is one world, which (although it may change) is neither duplicated nor discarded.

This paper discusses the use of a linear type system, based on the linear logic of J.-Y. Girard [Gir87, GLT89]. Values belonging to a linear type must be used exactly once: like the world, they can be neither duplicated nor discarded.

"No duplication" helps to guarantee efficient implementation. For example, it guarantees that it is safe to update an array destructively, by overwriting the given index with the given value. Similar efficiencies can be achieved when updating a value that corresponds to a file system. If a value is represented by a linked list, then knowing that one holds the only pointer to the list may enable efficient re-use of the list cells.

"No discarding" is equally important. In most implementations of a functional language it is essential to recover space that is discarded by the use of reference counting or garbage collection. Values of a linear type avoid this overhead. As we shall see, this doesn't mean that one cannot release storage used by a linear value; rather, it means that both allocation and deallocation of linear values are explicitly indicated in the program text.

Linear types enforce useful design constraints. Imagine a denotational semantics of a programming language. We would expect the store not to be duplicated or discarded—there would be something odd about a semantics that did either of these. Giving the store a linear type guarantees these properties.

Similarly, if the file system is to be represented by a single value (along the lines outlined above), then it is useful to give the file system a linear type. Among other things, this would trap some errors that novice programmers might make, such as throwing away the entire file system. (An early proposal for Haskell I/O treated the file system as a value. It was rejected, in part, because no mechanism like linearity was available.)

The system described here divides types into two families. Values of linear type have exactly one reference to them, and so require no garbage collection. Values of nonlinear type may have many pointers to them, or none, and do require garbage collection.

Pure linearity is, in fact, a stronger constraint than necessary. It is ok to have more than one reference to a value, so long as the value is being "read"; only when the value is "written" (e.g., destructively updated) is it necessary to guarantee that a single reference exists.

[1] Miranda is a trademark of Research Software Limited.

For example, the "cat" program given previously is not legal as it stands. It should be written

$$\begin{array}{rcl} \mathit{main\ files\ a} & = & \mathbf{let!}\ (a)v = \mathit{concat}\ [\mathit{lookup}\ i\ a \mid i \leftarrow \mathit{files}]\ \mathbf{in} \\ & & \mathit{update}\ \text{“stdout”}\ v\ . \end{array}$$

This allows multiple read accesses to the file system (one for each file in *files*), all of which may occur in parallel; but all of which must be completed before the "write" access (to "stdout").

The language used in this paper is lazy. The one exception is the "**let!**" construct, which completely evaluates the term after the "=" before commencing evaluation of the term after the "**in**". This sequencing is essential to guarantee that all reads of a structure are completed before it is updated. (In this example, it also has the unfortunate effect of removing the opportunity for lazy evaluation to cause reading the inputs and writing the output to act as coroutines.)

The name "single threading" was coined by Schmidt to describe the situation where the store of a denotational semantics satisfies the "no duplication" property, and he gave syntactic criteria for recognising when this occurs [Sch82, Sch85]. However, Schmidt's criteria say nothing about the "no discarding" property, and are designed for use with a strict, rather than a lazy, language. Further, Schmidt allows only one single-threaded value (i.e., the store) while the linear type system allows any number (e.g., two different destructively updated arrays). Nonetheless, the linear type system was closely inspired by Schmidt's work. In particular, the "**let!**" construct was added so that it would be straightforward to transform any single-threaded semantics (well, any one that doesn't discard its store) into an equivalent program with a linear store type.

A great deal of work has gone into compile-time analysis to determine when destructive updating is safe, notably by Bloss and Hudak [Blo89, BHY89, Hud86]; older analysis techniques for determining when list cells can be reused go back to Darlington and Burstall [DB76]. Analysis techniques have the advantage that destructive updating can be inferred whether the user indicates it explictly or not. With linear types, the user must decide which types are linear, and explicitly use **let!** where it is appropriate. Conversely, linear types have the advantage that the types make the user's intention manifest. With analysis techniques, a small change to a program might (by fooling the analysis) result in an unintended, large change in execution efficiency whose cause is difficult to trace.

The type system used in this paper is monomorphic. It is straightforward to extend it to a polymorphic language with explicit type applications, as in the Girard-Reynolds calculus [Gir72, Gir86, Rey74, Rey83, Rey85]. However, it is not clear whether it can be extended to the more common Hindley-Milner-Damas inference system [Hin69, Mil78, DM82] used in languages such as Miranda and Haskell. This is one area for future research.

Other computer scientists have also been struck by the potential of a type system based on Girard's linear logic.

A very elegant linear language has been developed by Lafont [Laf88], and a variant of it by Holmström [Hol88]. These systems have the amazing property that *every* value has exactly one reference to it, and so garbage collection is not required *at all*. This seems too good to be true, and perhaps it is: since there is never more than one pointer to a value, *the results of computations cannot be shared.* In fact, sharing is allowed (the so-called "of course" types) but shared values are dealt with in two ways, neither of which is as efficient as one would like:

- The first way is to represent the shared value by a pointer to a closure. This is how Lafont's system implements values of functional type, and how Holmström's system implements all values. Evaluating a closure does *not* cause the closure to be overwritten. Hence, this implementation is more like call-by-name than call-by-need: any shared value will be recomputed *each time* it is used, which can be staggeringly inefficient.

- The second way is to completely copy a shared value each time a reference to it is copied. This is how Lafont's system implements values of base type, such as lists. This is much more efficient than the first method, but it is still far less efficient than the usual conception of shared pointers.

While marvelously elegent and worthy of study, these systems seem unsuited for practical use.

The system described here is inspired in part by Lafont and Holmström's work, but it differs in three ways:

- Lafont and Holmström use a single family of types, augmented with the "of course" type constructor. The system given here uses two completely distinct families of types, one linear and one nonlinear. This is less elegant, as it means most of the typing rules appear in two versions, one for each family. But it means one has the advantage of unique reference, for linear types, while retaining the efficiency of sharing, for nonlinear types. In particular, none of the efficiency problems above arise.

- Lafont and Holmström use syntaxes rather different than that of the lambda calculus, and a non-trivial translation is required to transform lambda calculus terms into terms in their languages. In contrast, the traditional lambda calculus is a subset of the language described here.

- Lafont and Holmström have no analogue of the "**let!**" construct, and hence do not support some useful ways of structuring programs. In particular, there is no obvious way to translate Schmidt's single-threaded programs into linear programs in the style of Lafont and Holmström, but there is a straightforward translation into the type system in this paper.

Another type system, more loosely inspired by linear logic, has been developed by Hudak and Guzmán [HG89]. The two were developed concurrently and mostly independently, although the exact form of "**let!**" used here was partly influenced by their work. The goals of their work are more ambitious, and as a result their type system is (perhaps) more complex. Further work is needed to understand the relation between the two systems. One clear difference is that their system guarantees the "no duplication" property, but not the "no discarding" property; hence in their system deallocation of linear values is not always explicit.

The remainder of this paper is organised as follows. Section 1 describes a conventional typed language. Section 2 modifies this language for linear types, and Section 3 adds back in nonlinear types. Section 4 introduces the "**let!**" construct. Section 5 presents an extended example involving arrays, showing how to implement an interpreter for a simple imperative language. Section 6 concludes.

1 Conventional types

This section presents the definition of a conventional typed lambda calculus, that is, one without linear types. The definition will be adapted to linear types in the following sections.

The only novel feature of the calculus in this section is an unusual form of data type declaration. That is, it is unusual for theorists; in practice, a quite similar form of declaration is used in languages such as Miranda and Haskell.

Assume there is a fixed set of base type declarations. Each declaration takes the form

$$K = C_1 \ T_{11} \ \ldots \ T_{1k_1} \mid \cdots \mid C_n \ T_{n1} \ \ldots \ T_{nk_n} \ ,$$

where K is a new base type name, the C_i are new constructor names, and the T_{ij} are types (called the *immediate components* of K). For example, here are declarations for booleans and lists of Val, where Val is another base type:

$$\begin{array}{lcl} \mathit{Bool} & = & \mathit{True} \mid \mathit{False} \ , \\ \mathit{List} & = & \mathit{Nil} \mid \mathit{Cons} \ \mathit{Val} \ \mathit{List} \ . \end{array}$$

The immediate components of $List$ are Val and $List$.

A type is a base type or a function type:

$$\begin{array}{lcl} T & ::= & K \\ & \mid & (U \to V) \ . \end{array}$$

Here K ranges over base types and T, U, V range over types.

A term is variable, abstraction, application, constructor term, case term, or fixpoint:

$$\begin{array}{lcl} t & ::= & x \\ & \mid & (\lambda x : U . \ v) \\ & \mid & (t \ u) \\ & \mid & (C \ t_1 \ \ldots \ t_k) \\ & \mid & (\mathbf{case} \ u \ \mathbf{of} \ C_1 \ x_{11} \ \ldots \ x_{1k_1} \to v_1 \mid \cdots \mid C_n \ x_{n1} \ \ldots \ x_{nk_n} \to v_n) \\ & \mid & (\mathit{fix} \ t) \ . \end{array}$$

Here x ranges over variables and t, u, v range over terms. Following convention, parentheses may be dropped in $(T \to (U \to V))$ and $((t \ u) \ v)$.

Let A, B range over assumption lists. An assumption list associates variables with types:

$$A \ ::= \ x_1 : T_1, \ldots, x_n : T_n \ .$$

Write $x \notin A$ to indicate that x is not one of the variables in A.

A typing is an assertion of the form $A \vdash t : T$. This asserts that if the assumptions in A are satisfied (that is, x_i has type T_i for each $x_i : T_i$ in A) then t has type T. The type system possesses *unicity of type*: for a given A and t there is at most one T such that $A \vdash t : T$.

Typings are derived using the rules shown in Figure 1. As usual, these rules take the form of a number of hypotheses (above the line) and a conclusion (below the line) that can be drawn if all the hypotheses are satisfied. The rules concerning base types include an additional hypothesis displaying the declaration of the type, shown in a box above the rule. The rules come in pairs: the $\rightarrow\mathcal{I}$ rule introduces a function (by a lambda expression), and the $\rightarrow\mathcal{E}$ rule eliminates a function (by applying it); the $K\mathcal{I}$ rule introduces a value of a base type (by constructing it), and the $K\mathcal{E}$ rule eliminates a value of a base type (by a case analysis). There are also rules for variables and fixpoints (VAR and FIX).

As an example, here is a function two append two lists, written out in painful detail:

$$\begin{array}{l} \mathit{fix}\ (\lambda \mathit{append} : \mathit{List} \rightarrow \mathit{List} \rightarrow \mathit{List}. \\ \quad \lambda xs : \mathit{List}.\ \lambda ys : \mathit{List}. \\ \quad\quad \textbf{case}\ xs\ \textbf{of} \\ \quad\quad\quad \mathit{Nil} \rightarrow ys \\ \quad\quad\quad \mathit{Cons}\ x'\ xs' \rightarrow \mathit{Cons}\ x'\ (\mathit{append}\ xs'\ ys)\ )\ . \end{array}$$

This term is well-typed, with type $\mathit{List} \rightarrow \mathit{List} \rightarrow \mathit{List}$.

2 Linear types

In the conventional type system just described, each occurrence of a variable in an assumption list, $x : T$, can be read as permission to use the variable x at type T. Furthermore, it can be used any number of times: zero, one, or many. Hence

$$x : \mathit{Arr} \vdash () : \mathit{Unit}$$

is a valid typing, even though the assumption $x : \mathit{Arr}$ is used zero times. Similarly,

$$x : \mathit{Arr} \vdash (x, x) : \mathit{Arr} \times \mathit{Arr}$$

is also valid, even though the assumption $x : \mathit{Arr}$ is used twice.

The key idea in a linear type system is that *each assumption must be used exactly once*. Thus both of the above typings become illegal. As a first approximation, if $A \vdash t : T$ is a valid typing in a linear type system, then each variable in A appears exactly once in t.

Linear types are written differently from conventional types. Base types are written $¡K$, and function types are written $U \multimap V$. (In linear logic, by default an assumption T must be used exactly once; if an assumption is to be discarded or duplicated it must be written $!T$. The symbol for linear types types was chosen to reflect this.)

Thus, the new grammar of types is:

$$\begin{array}{rcl} T & ::= & ¡K \\ & | & (U \multimap V)\ . \end{array}$$

The grammar of terms is changed similarly:

$$\begin{array}{rcl} t & ::= & x \\ & | & (¡\lambda x : U.\ v) \\ & | & (¡t\ u) \\ & | & (¡C\ t_1\ \ldots\ t_k) \\ & | & (\textbf{case}\ u\ \textbf{of}\ ¡C_1\ x_{11}\ \ldots\ x_{1k_1} \rightarrow v_1 \mid \cdots \mid ¡C_n\ x_{n1}\ \ldots\ x_{nk_n} \rightarrow v_n)\ . \end{array}$$

$$\textsc{var}\ \frac{}{A,\ x : T \vdash x : T}$$

$$\rightarrow\!\mathcal{I}\ \frac{A,\ x : U \vdash v : V}{A \vdash (\lambda x : U.\ v) : U \rightarrow V}\ x \notin A$$

$$\rightarrow\!\mathcal{E}\ \frac{A \vdash t : U \rightarrow V \qquad A \vdash u : U}{A \vdash (t\ \ u) : V}$$

$$\boxed{K = \cdots \mid C\ \ T_1\ \ldots\ T_k \mid \cdots}$$

$$K\mathcal{I}\ \frac{\begin{array}{c} A \vdash t_1 : T_1 \\ \cdots \\ A \vdash t_k : T_k \end{array}}{A \vdash (C\ \ t_1\ \ldots\ t_k) : K}$$

$$\boxed{K = C_1\ \ T_{11}\ \ldots\ T_{1k_1} \mid \cdots \mid C_n\ \ T_{n1}\ \ldots\ T_{nk_n}}$$

$$K\mathcal{E}\ \frac{\begin{array}{c} A \vdash u : K \\ A,\ x_{11} : T_{11},\ \ldots,\ x_{1k_1} : T_{1k_1} \vdash v_1 : V \\ \cdots \\ A,\ x_{n1} : T_{n1},\ \ldots,\ x_{nk_n} : T_{nk_n} \vdash v_n : V \end{array}}{A \vdash (\textbf{case}\ \ u\ \ \textbf{of}\ \ C_1\ \ x_{11}\ \ldots\ x_{1k_1} \rightarrow v_1 \mid \cdots \mid C_n\ \ x_{n1}\ \ldots\ x_{nk_n} \rightarrow v_n) : V}\ x_{ij} \notin A$$

$$\textsc{fix}\ \frac{A \vdash t : T \rightarrow T}{A \vdash (\mathit{fix}\ \ t) : T}$$

Figure 1: Conventional typing rules

$$\textsc{var}\;\frac{}{x : T \vdash x : T}$$

$$\multimap\mathcal{I}\;\frac{A,\, x : U \vdash v : V}{A \vdash (\text{¡}\lambda x : U.\; v) : U \multimap V}\; x \notin A$$

$$\multimap\mathcal{E}\;\frac{A \vdash t : U \multimap V \qquad B \vdash u : U}{A, B \vdash (\text{¡}t\;\; u) : V}$$

$$\boxed{\text{¡}K = \cdots \mid \text{¡}C\;\; T_1\; \ldots\; T_k \mid \cdots}$$

$$\text{¡}K\mathcal{I}\;\frac{\begin{array}{c} A_1 \vdash t_1 : T_1 \\ \cdots \\ A_k \vdash t_k : T_k \end{array}}{A_1, \ldots, A_k \vdash (\text{¡}C\;\, t_1\; \ldots\; t_k) : K}$$

$$\boxed{\text{¡}K = \text{¡}C_1\;\; T_{11}\; \ldots\; T_{1k_1} \mid \cdots \mid \text{¡}C_n\;\; T_{n1}\; \ldots\; T_{nk_n}}$$

$$\text{¡}K\mathcal{E}\;\frac{\begin{array}{c} A \vdash u : K \\ B,\, x_{11} : T_{11}, \ldots, x_{1k_1} : T_{1k_1} \vdash v_1 : V \\ \cdots \\ B,\, x_{n1} : T_{n1}, \ldots, x_{nk_n} : T_{nk_n} \vdash v_n : V \end{array}}{A, B \vdash (\textbf{case}\;\; u\;\; \textbf{of}\;\; \text{¡}C_1\, x_{11}\; \ldots\; x_{1k_1} \rightarrow v_1 \mid \cdots \mid \text{¡}C_n\, x_{n1}\; \ldots\; x_{nk_n} \rightarrow v_n) : V}\; x_{ij} \notin B$$

Figure 2: Rules for linear types

There are no fixpoints in the linear language. A linear value should be accessed exactly once, but *fix* t is equivalent to $t\;(t\;(t\;\cdots))$.

Each of the typing rules must now be modified accordingly. The new rules are summarised in Figure 2.

Consider the old VAR rule:

$$\textsc{var}\;\frac{}{A,\, x : T \vdash x : T}\;.$$

Any assumption in the list A is unused in the typing, and hence this is no longer legal. The new rule is:

$$\textsc{var}\;\frac{}{x : T \vdash x : T}\;.$$

Here the single assumption on the left is used exactly once on the right.

Next, consider the old $\rightarrow\mathcal{E}$ rule:

$$\rightarrow\mathcal{E}\ \frac{A \vdash t : U \rightarrow V \qquad A \vdash u : U}{A \vdash (t\ u) : V}\ .$$

Any variable that appears in A may appear in both t and u, and this should be prohibited. The new rule is:

$$\multimap\mathcal{E}\ \frac{A \vdash t : U \multimap V \qquad B \vdash u : U}{A, B \vdash (\text{¡}t\ u) : V}\ .$$

Here both A and B range over assumption lists, and A, B stands for the conjunction of the two lists. It is easy to see that if each variable in A occurs exactly once in t, and if each variable in B occurs exactly once in u, then each variable in A, B occurs exactly once in $t\ u$.

The ¡$K\mathcal{I}$ rule is modified to refer to k assumption lists, $A_1, \ldots, A_k$. On the other hand, the ¡$K\mathcal{E}$ rule is modified to refer to only two assumption lists, A and B. The same assumption list B is used for each of the branches of the **case**; this is sensible because exactly one branch will be executed. This, incidentally, is why the claim that each variable appears exactly once is only approximate; in a **case** term a variable may appear exactly once in each branch.

We can define some familiar combinators by writing I_X, B_{XYZ}, and C_{XYZ} as abbreviations for

$$\begin{array}{lcl}
\mathsf{I}_X & = & \text{¡}\lambda x : X.\ x \\
\mathsf{B}_{XYZ} & = & \text{¡}\lambda f : Y \multimap Z.\ \text{¡}\lambda g : X \multimap Y.\ \text{¡}\lambda x : X.\ \text{¡}f\ (\text{¡}g\ x) \\
\mathsf{C}_{XYZ} & = & \text{¡}\lambda f : X \multimap Y \multimap Z.\ \text{¡}\lambda y : Y.\ \text{¡}\lambda x : X.\ \text{¡}(\text{¡}f\ x)\ y)
\end{array}$$

yielding the well-typings

$$\begin{array}{lcl}
\vdash \mathsf{I}_X & : & X \multimap X \\
\vdash \mathsf{B}_{XYZ} & : & (Y \multimap Z) \multimap (X \multimap Y) \multimap X \multimap Z \\
\vdash \mathsf{C}_{XYZ} & : & (X \multimap Y \multimap Z) \multimap (Y \multimap X \multimap Z)\ .
\end{array}$$

On the other hand, if we define

$$\begin{array}{lcl}
\mathsf{K}_{XY} & = & \text{¡}\lambda x : X.\ \text{¡}\lambda y : Y.\ x \\
\mathsf{S}_{XYZ} & = & \text{¡}\lambda f : X \multimap Y \multimap Z.\ \text{¡}\lambda g : X \multimap Y.\ \text{¡}\lambda x : X.\ \text{¡}(\text{¡}f\ x)\ (\text{¡}g\ x)
\end{array}$$

then K_{XY} and S_{XYZ} have no well-typings in the linear system; K because it discards y, and S because duplicates x. More generally, any term that translates into a combination of the I, B, and C combinators is well-typed in this system, while any term requiring K or S for its translation is not.

In the linear system, each operation that introduces (and allocates) a value is paired with exactly one operation that eliminates (and deallocates) that value. The introduction operations (abstraction and construction) create values to which a single reference exists. This reference may never be duplicated or discarded, so the elimination operations (application and case) act on values to which they hold the sole reference. Hence, after an application the storage occupied by the function may be reclaimed, and after a case analysis the storage occupied by the node analysed may be reclaimed. No reference counting or garbage collection is required.

This efficiency is achieved by restricting the language severely: each variable that is bound must be used exactly once. This is perfect for variables corresponding to, say, a file system or a large array; but it is not reasonable to impose such a restriction on a variable containing, say, an integer. Thus, the type system needs to distinguish two sorts of values: those that may not be duplicated or discarded, and those that may. This extension is the subject of the next section.

3 Nonlinear types

The linear language of the previous section is wonderfully efficient, but woefully lacking in expressiveness. To recover the power of a sensible programming language, we reintroduce the types K and $U \to V$. This yields a language with two families of types: *linear* types, ¡K and $U \multimap V$, and *nonlinear* types, K and $U \to V$. Values of linear types may not be duplicated or discarded; values of nonlinear types may.

If T is a linear type, then an assumption of the form $x : T$ is a permission, and a requirement, to use the variable x exactly once. It is a restrictive assumption. May I discard x? ¡No! May I duplicate x? ¡No!

If T is a nonlinear type, then an assumption of the form $x : T$ is a permission to use x zero, one, or many times. It is a generous assumption. May I discard x? Of course! May I duplicate x? Of course!

(Similarly, in linear logic an assumption may not be discarded or duplicated unless it is of the form $!T$. The ! is pronounced "of course". But be warned: the formula $U \to V = (!U) \multimap V$ that holds in linear logic is *not* appropriate to this paper; a better guide would be $U \to V = !(U \multimap V)$.)

There are now two forms of base type declaration, linear and nonlinear:

$$\begin{aligned} \text{¡}K &= \text{¡}C_1\ T_{11}\ \ldots\ T_{1k_1} \mid \cdots \mid \text{¡}C_n\ T_{n1}\ \ldots\ T_{nk_n}\ , \\ K &= C_1\ T_{11}\ \ldots\ T_{1k_1} \mid \cdots \mid C_n\ T_{n1}\ \ldots\ T_{nk_n}\ . \end{aligned}$$

In the former, the immediate components may be any types, while in the latter case they must be nonlinear. In other words, a nonlinear data structure must not contain any linear components. Thus,

$$\begin{aligned} \text{¡}List &= \text{¡}Nil \mid \text{¡}Cons\ \text{¡}Val\ \text{¡}List, \\ \text{¡}List &= \text{¡}Nil \mid \text{¡}Cons\ Val\ \text{¡}List, \text{ and} \\ List &= Nil \mid Cons\ Val\ List \end{aligned}$$

are all ok, but

$$List = Nil \mid Cons\ \text{¡}Val\ List$$

is not ok.

This restriction is easy to understand. A value of linear type must be accessed exactly once. Say that a value of nonlinear type contained a pointer to it. If the nonlinear value was duplicated, the linear value would be accessed once for each duplication; it would be virtually duplicated. If the nonlinear value was discarded, the linear value would never be accessed; it would be virtually discarded. Hence, no nonlinear value may point to a linear value.

For example, the following fragment should be well-typed if xs has type $List$:

$$
\begin{array}{l}
\textbf{case}\ xs\ \textbf{of}\ \cdots \mid Cons\ x'\ xs' \rightarrow \\
\quad\quad \textbf{case}\ xs\ \textbf{of}\ \cdots \mid Cons\ x''\ xs'' \rightarrow \\
\quad\quad\quad\quad \ldots x' \ldots x'' \ldots
\end{array}
$$

Under the *illegal* declaration of $List$ above, both x' and x'' would have the linear type $¡Val$, even though they are both bound to the same value. Any operation that took advantage of the linear type of x' (to reuse its storage, or to update it destructively) would create a "side effect" on x''. So much for referential transparency!

Fortunately, the linear type system disallows this. With the legal declaration of the type $List$, both x' and x'' have the nonlinear type Val, and there is no problem. On the other hand, if xs has the linear type $¡List$, then the fragment would be illegal because it uses xs twice.

The grammar of types is now

$$
\begin{array}{rcl}
T & ::= & ¡K \\
 & \mid & K \\
 & \mid & (U \multimap V) \\
 & \mid & (U \rightarrow V)\ .
\end{array}
$$

Each type is deemed linear or nonlinear, depending on its topmost constructor. Hence $(T \multimap (U \rightarrow V))$ is linear, while $(T \rightarrow (U \multimap V))$ is nonlinear.

The grammar of terms is similarly extended. To all of the term formers in Section 2 are added all of the term formers in Section 1. Thus $(¡\lambda x : U.\ v)$ is a term of type $U \multimap V$, and $(\lambda x : U.\ v)$ is a term of type $U \rightarrow V$. The typing rules consist of all those in Figure 2 together with the additional rules in Figure 3.

Mathematically, the notion that values of a nonlinear type may be discarded or duplicated is represented by the rules:

$$
\textsc{kill}\ \frac{A \vdash u : U}{A, x : T \vdash u : U}\ \text{nonlinear } T,
$$

$$
\textsc{copy}\ \frac{A, x : T, x : T \vdash u : U}{A, x : T \vdash u : U}\ \text{nonlinear } T.
$$

The KILL rule allows a value of type T to be discarded, while the COPY rule allows a value of type T to be duplicated, so long as T is a nonlinear type. To formulate the COPY rule neatly, assumptions of the form $x : T$ are allowed to appear possibly multiple times in an assumption list when T is a nonlinear type; thus assumption lists are multisets rather than sets. Note that the VAR rule of Figure 2 applies to both linear and nonlinear types.

The nonlinear version of the function introduction rule is:

$$
\rightarrow\!\mathcal{I}\ \frac{A,\ x : U \vdash v : V}{A \vdash (\lambda x : U.\ v) : U \rightarrow V}\ x \notin A,\ \text{nonlinear } A.
$$

This is identical to the old rule, except for the addition of a side condition requiring that the assumption list A be nonlinear. An assumption list is nonlinear if each assumption

$$\textsc{kill}\ \frac{A \vdash u : U}{A, x : T \vdash u : U}\ \text{nonlinear } T$$

$$\textsc{copy}\ \frac{A, x : T, x : T \vdash u : U}{A, x : T \vdash u : U}\ \text{nonlinear } T$$

$$\rightarrow\mathcal{I}\ \frac{A,\ x : U \vdash v : V}{A \vdash (\lambda x : U.\ v) : U \rightarrow V}\ x \notin A,\ \text{nonlinear } A$$

$$\rightarrow\mathcal{E}\ \frac{A \vdash t : U \rightarrow V \qquad B \vdash u : U}{A, B \vdash (t\ \ u) : V}$$

$$\boxed{K = \cdots \mid C\ \ T_1\ \ldots\ T_k \mid \cdots, \qquad \text{nonlinear } T_i}$$

$$K\mathcal{I}\ \frac{\begin{array}{c} A_1 \vdash t_1 : T_1 \\ \cdots \\ A_k \vdash t_k : T_k \end{array}}{A_1,\ \ldots,\ A_k \vdash (C\ \ t_1\ \ldots\ t_k) : K}$$

$$\boxed{K = C_1\ \ T_{11}\ \ldots\ T_{1k_1} \mid \cdots \mid C_n\ \ T_{n1}\ \ldots\ T_{nk_n}, \qquad \text{nonlinear } T_{ij}}$$

$$K\mathcal{E}\ \frac{\begin{array}{c} A \vdash u : K \\ B,\ x_{11} : T_{11},\ \ldots,\ x_{1k_1} : T_{1k_1} \vdash v_1 : V \\ \cdots \\ B,\ x_{n1} : T_{n1},\ \ldots,\ x_{nk_n} : T_{nk_n} \vdash v_n : V \end{array}}{A, B \vdash (\textbf{case}\ \ u\ \ \textbf{of}\ C_1\ x_{11}\ \ldots\ x_{1k_1} \rightarrow v_1 \mid \cdots \mid C_n\ x_{n1}\ \ldots\ x_{nk_n} \rightarrow v_n) : V}\ x_{ij} \notin B$$

$$\textsc{fix}\ \frac{A \vdash t : T \rightarrow T}{A \vdash (\mathit{fix}\ \ t) : T}$$

Figure 3: Rules for nonlinear types

$x_i : T_i$ in it has a nonlinear T_i. No condition is placed on U or V by this rule; they may be either linear or nonlinear.

This restriction follows from the principle, established above, that a nonlinear value must not contain pointers to linear values. A function value is a closure that contains a pointer to an environment binding each variable in A. Hence a nonlinear function can only be introduced in an environment A that contains only nonlinear types.

For example, if $¡X$ is a linear type and Y is nonlinear, then

$$(\lambda x : ¡X.\ \lambda y : Y.\ x) : ¡X \to Y \to ¡X$$

is not a well-typing. (It is an easy exercise to show that if it were well-typed then any linear value could be duplicated.) On the other hand, both of

$$\begin{aligned} &\vdash (\lambda x : ¡X.\ ¡\lambda y : Y.\ x) : ¡X \to Y \multimap ¡X, \\ &\vdash (\lambda y : Y.\ \lambda x : ¡X.\ x) : Y \to ¡X \to ¡X \end{aligned}$$

are ok, the first because $\multimap\mathcal{I}$ places no constraint on the assumption list, and the second because $\to\mathcal{I}$ places no constraint on the argument or result type.

Returning to a previous example, here is another version of the append function:

$$\begin{array}{l} \mathit{fix}\ \ (\lambda \mathit{append} : ¡\mathit{List} \multimap ¡\mathit{List} \multimap ¡\mathit{List}. \\ \qquad \lambda \mathit{xs} : ¡\mathit{List}.\ \lambda \mathit{ys} : ¡\mathit{List}. \\ \qquad\quad \textbf{case}\ \mathit{xs}\ \textbf{of} \\ \qquad\qquad ¡\mathit{Nil} \to \mathit{ys} \\ \qquad\qquad ¡\mathit{Cons}\ x'\ \mathit{xs}' \to ¡\mathit{Cons}\ x'\ (¡(¡\mathit{append}\ \mathit{xs}')\ \mathit{ys})\ )\ \ . \end{array}$$

This is well-typed under either declaration for $¡List$ given above. Since both arguments are linear, if the first argument is a $¡Cons$ cell then this cell may be deallocated immediately. It requires only a modestly good compiler to notice that the best thing to do with this cell is not to return it to free storage, but to reuse it in building the result. An only moderately better compiler would notice that the head of this cell may be left unchanged. Only the tail of the cell needs to be filled in with the result of the recursive call to *append*. In fact, that recursive call will almost always return the same value that is there already (the exception being when it is $¡Nil$), and a little loop unrolling could result in a very efficient version of append indeed—but it might require a less modestly good compiler writer to notice that.

4 Read-only access

In order for destructive updating of a value to be safe, it is essential that there be only one reference to the value when the update occurs. In the linear type system, this is enforced by guaranteeing that there is always exactly one reference to the value. This restriction is stronger than necessary. It is perfectly safe to have more than one reference to a value temporarily, as long as only one reference exists when the update is performed.

The situation here is similar to the well-known “readers and writers” problem, where access to a resource is to be controlled. Many processes may simultaneously have read access, but a process may have write access only if no other process has access (either read or write) to the resource.

The two sorts of access are modelled using two types. A linear type corresponds to write access, and a nonlinear type corresponds to read access: permission to write must be unique, but permission to read may be freely duplicated.

A new form of term is introduced for granting read-only access:

$$\mathbf{let!}\ (x)\ y = u\ \mathbf{in}\ v\ .$$

This is evaluated similarly to a conventional **let** term: first u is evaluated, then y is bound to the result, then v is evaluated in the extended environment. But there are three differences from a garden-variety **let**.

The first is that within u the variable x has nonlinear type, while within v the variable x has linear type. That is, during evaluation of u, read-only access is allowed to the value in x.

The second is that evaluation of u must be carried out completely before evaluation of v commences—this is sometimes called *hyperstrict* evaluation. For instance, if u returns a list, then all elements of this list must be evaluated. This is required in order to guarantee that all references to x within u are freed before evaluation of v commences.

The third is that there are some constraints on the type of x and u. It must not be possible for u (or any component of u) to be equal to x (or any component of x)—otherwise, read access to x (or a component of x) could be passed outside the scope of u. This condition is made precise below.

Define the *components* of a type to be itself and, if it is a base type, all components of its immediate components. We will restrict our attention to the case where all components of the type of x are base types.

Given a type T, the corresponding nonlinear type $!T$ is derived from it as follows. If $¡K$ is a linear base type defined by

$$¡K = ¡C_1\ \ T_{11}\ \ldots\ T_{1k_1} \mid \cdots \mid ¡C_n\ \ T_{n1}\ \ldots\ T_{nk_n},$$

then $!¡K$ is the nonlinear base type K defined by

$$K = C_1\ \ !T_{11}\ \ldots !T_{1k_1} \mid \cdots \mid C_n\ \ !T_{n1}\ \ldots !T_{nk_n},$$

where the components of K are recursively converted. If K is a nonlinear base type, then $!K$ is identical to K.

Let T be the type of x, and U be the type of u. We must ensure that u cannot "smuggle out" any component of x that is linear. This is guaranteed if no linear component of T has a corresponding nonlinear component in U, and if no component of U is a function type. (Functions are disallowed since a function term may have x or a component of x as a free variable.) If this holds, say that *U is safe for T*.

Finally, the required typing rule is:

$$\textsc{let!}\ \frac{\begin{array}{c} A, x : !T \vdash u : U \\ B, x : T, y : U \vdash v : V \end{array}}{A, B, x : T \vdash (\mathbf{let!}\ (x)\ y = u\ \mathbf{in}\ v) : V}\ U \text{ safe for } T\ .$$

An example of the use of **let!** appears in the next section.

It is easy to allow read-only access to several variables at once; simply take

$$\mathbf{let!}\ (x_1, x_2, \ldots, x_m)\ y = u\ \mathbf{in}\ v$$

as an abbreviation for

$$\mathbf{let!}\ (x_1)\ y = (\mathbf{let!}\ (x_2, \ldots, x_m)\ y' = u\ \mathbf{in}\ y')\ \mathbf{in}\ v\ .$$

5 Arrays

5.1 Conventional arrays

Before considering how to add arrays to a linear type system, let's review the use of arrays in the conventional type system.

An array associates indices with values. We will fix the index and value types, and write Arr for the type of arrays with indices of type Ix and values of type Val. There are three operations of interest on arrays:

$$\begin{array}{lcl} alloc & : & Arr, \\ lookup & : & Ix \to Arr \to Val, \\ update & : & Ix \to Val \to Arr \to Arr. \end{array}$$

Here $alloc$ allocates a new array (with all entries set to some fixed initial value); $lookup\ i\ a$ returns the value at index i in array a; and $update\ i\ v\ a$ returns an array identical to a except that index i is associated with value v. (In practice, the array type may be parameterised on the index and value types, and the new array function may take additional arguments to determine index bounds and initial values; but the simpler version suffices to demonstrate the central ideas.)

This section will use the small interpreter shown in Figure 4 as a running example. This is written in an equational notation familiar from languages such as Miranda and Haskell; it is easy to translate the equations into lambda and case terms.

The interpreter can be read as a denotational semantics for a simple imperative language. Variable names are identified with type Ix, the values stored in variables are identified with type Val, and stores (which map variable names to values) are identified with type Arr.

Three data types correspond to the abstract syntax of expressions, commands, and programs.

- An expression is a variable, a constant, or the sum of two expressions. The semantic function corresponding to an expression takes an array into a value.
- A command is an assignment, a sequence of two commands, or a conditional. The semantic function corresponding to a command takes an array into an array.
- A program consists of a command followed by an expression. The semantics corresponding to a program is a value.

The interpreter satisfies Schmidt's single-threading criteria. Thus, it is safe to implement the *update* operation in a way that re-uses the store allocated for the array. (This is to be expected, since the array represents the store of an imperative language.) However, how is the implementation to determine when it is safe to implement *update* in this way? Succeeding sections will show how linear types can be used for this purpose.

5.2 Linear arrays

Now that the framework of the linear type system is in place, adding primitives for arrays is relatively straightforward. This section will show how to add arrays when

$$
\begin{array}{lcl}
\mathit{Expr} & = & \mathit{Var}\ \mathit{Ix} \mid \mathit{Const}\ \mathit{Val} \mid \mathit{Plus}\ \mathit{Expr}\ \mathit{Expr} \\
\mathit{Com} & = & \mathit{Asgn}\ \mathit{Ix}\ \mathit{Expr} \mid \mathit{Seq}\ \mathit{Com}\ \mathit{Com} \mid \mathit{If}\ \mathit{Expr}\ \mathit{Com}\ \mathit{Com} \\
\mathit{Prog} & = & \mathit{Do}\ \mathit{Com}\ \mathit{Expr} \\
\\
\mathit{expr} & : & \mathit{Expr} \to \mathit{Arr} \to \mathit{Val} \\
\mathit{expr}\ (\mathit{Var}\ i)\ a & = & \mathit{lookup}\ i\ a \\
\mathit{expr}\ (\mathit{Const}\ v)\ a & = & v \\
\mathit{expr}\ (\mathit{Plus}\ e_0\ e_1)\ a & = & \mathit{expr}\ e_0\ a + \mathit{expr}\ e_1\ a \\
\\
\mathit{com} & : & \mathit{Com} \to \mathit{Arr} \to \mathit{Arr} \\
\mathit{com}\ (\mathit{Asgn}\ i\ e)\ a & = & \mathit{update}\ i\ (\mathit{expr}\ e\ a)\ a \\
\mathit{com}\ (\mathit{Seq}\ c_0\ c_1)\ a & = & \mathit{com}\ c_1\ (\mathit{com}\ c_0\ a) \\
\mathit{com}\ (\mathit{If}\ e\ c_0\ c_1)\ a & = & \mathbf{if}\ \mathit{expr}\ e\ a = 0\ \mathbf{then}\ \mathit{com}\ c_0\ a\ \mathbf{else}\ \mathit{com}\ c_1\ a \\
\\
\mathit{prog} & : & \mathit{Prog} \to \mathit{Val} \\
\mathit{prog}\ (\mathit{Do}\ c\ e) & = & \mathit{expr}\ e\ (\mathit{com}\ c\ \mathit{alloc})
\end{array}
$$

Figure 4: A simple interpreter

"read only" access is not used. This leads to a small problem, which will be resolved by the use of "read only" access in the next section.

As before, an array type maps indices into values. The array type is linear, and so is written $¡\mathit{Arr}$. For the simplest version of arrays, the index and value types are nonlinear, and so are written Ix and Val.

We will require a data type that pairs a Val with a $¡\mathit{Arr}$. This may be declared by

$$¡\mathit{ValArr} \quad = \quad ¡\mathit{MkPair}\ \mathit{Val}\ ¡\mathit{Arr},$$

but for readability we will write $\mathit{Val} \otimes ¡\mathit{Arr}$ instead of $¡\mathit{ValArr}$, and (v, a) instead of $¡\mathit{MkPair}\ v\ a$, and $\mathbf{let}\ (v, a) = t\ \mathbf{in}\ u$ instead of $\mathbf{case}\ t\ \mathbf{of}\ ¡\mathit{MkPair}\ v\ a \to u$.

The four operations on linear arrays are:

$$
\begin{array}{lcl}
\mathit{alloc} & : & ¡\mathit{Arr}, \\
\mathit{lookup} & : & \mathit{Ix} \to ¡\mathit{Arr} \to \mathit{Val} \otimes ¡\mathit{Arr}, \\
\mathit{update} & : & \mathit{Ix} \to \mathit{Val} \to ¡\mathit{Arr} \to ¡\mathit{Arr}, \\
\mathit{dealloc} & : & \mathit{Val} \otimes ¡\mathit{Arr} \to \mathit{Val}.
\end{array}
$$

The *alloc* and *update* operations are as before. The new *lookup* operation, being passed the sole reference to an array, must return a reference to the same array when it completes. Otherwise, since arrays cannot be duplicated, an array could only be indexed once, and then would be lost forever! The linear type discipline requires that values of a linear type be explicitly deallocated, and this is the purpose of the *dealloc* operation.

The meaning of these operations can be explained in terms of the old operations as

$$
\begin{array}{lcl}
\mathit{Expr} & = & \mathit{Var}\ \mathit{Ix} \mid \mathit{Const}\ \mathit{Val} \mid \mathit{Plus}\ \mathit{Expr}\ \mathit{Expr} \\
\mathit{Com} & = & \mathit{Asgn}\ \mathit{Ix}\ \mathit{Expr} \mid \mathit{Seq}\ \mathit{Com}\ \mathit{Com} \mid \mathit{If}\ \mathit{Expr}\ \mathit{Com}\ \mathit{Com} \\
\mathit{Prog} & = & \mathit{Do}\ \mathit{Com}\ \mathit{Expr} \\
\\
\mathit{expr} & : & \mathit{Expr} \to \text{¡}\mathit{Arr} \to \mathit{Val} \otimes \text{¡}\mathit{Arr} \\
\mathit{expr}\ (\mathit{Var}\ i)\ a & = & \mathit{lookup}\ i\ a \\
\mathit{expr}\ (\mathit{Const}\ v)\ a & = & (v, a) \\
\mathit{expr}\ (\mathit{Plus}\ e_0\ e_1)\ a & = & \mathbf{let}\ (v_0, a_0) = \mathit{expr}\ e_0\ a\ \mathbf{in} \\
 & & \mathbf{let}\ (v_1, a_1) = \mathit{expr}\ e_1\ a_0\ \mathbf{in} \\
 & & (v_0 + v_1, a_1) \\
\\
\mathit{com} & : & \mathit{Com} \to \text{¡}\mathit{Arr} \to \text{¡}\mathit{Arr} \\
\mathit{com}\ (\mathit{Asgn}\ i\ e)\ a & = & \mathbf{let}\ (v, a') = \mathit{expr}\ e\ a\ \mathbf{in}\ \mathit{update}\ i\ v\ a' \\
\mathit{com}\ (\mathit{Seq}\ c_0\ c_1)\ a & = & \mathit{com}\ c_1\ (\mathit{com}\ c_0\ a) \\
\mathit{com}\ (\mathit{If}\ e\ c_0\ c_1)\ a & = & \mathbf{let}\ (v, a') = \mathit{expr}\ e\ a\ \mathbf{in} \\
 & & \mathbf{if}\ v = 0\ \mathbf{then}\ \mathit{com}\ c_0\ a'\ \mathbf{else}\ \mathit{com}\ c_1\ a' \\
\\
\mathit{prog} & : & \mathit{Prog} \to \mathit{Val} \\
\mathit{prog}\ (\mathit{Do}\ c\ e) & = & \mathit{dealloc}\ (\mathit{expr}\ e\ (\mathit{com}\ c\ \mathit{alloc}))
\end{array}
$$

Figure 5: A simple interpreter, version 2

follows:

$$
\begin{array}{lcl}
\mathit{alloc} & = & \mathit{alloc}_{old}\,, \\
\mathit{lookup}\ i\ a & = & (\mathit{lookup}_{old}\ a\ i, a), \\
\mathit{update}\ i\ v\ a & = & \mathit{update}_{old}\ i\ v\ a, \\
\mathit{dealloc}\ x & = & \mathbf{let}\ (v, a) = x\ \mathbf{in}\ v.
\end{array}
$$

However, we could not write this program to define the new operations; they must be defined as primitives. (The definition of *lookup* is illegal because although a is linear it appears twice on the right-hand side; and the definition of *dealloc* is illegal because although a is linear it appears no times on the right-hand side.)

A new version of the interpreter using the linear array operations is shown in Figure 5. Unfortunately, this second version is a little more elaborate than the first, because each application of *lookup* (and, hence, also each application of *expr*) requires additional plumbing to pass around the unchanged array. This is particularly unfortunate in the *Plus* case of the definition of *expr*. It should be possible to evaluate the two summands, e_0 and e_1, in parallel, but it is necessary here to specify some order, in this case e_0 before e_1.

The linear type system is “too linear” in that it forces a linear order to be specified for operations (like *Plus* in *expr*) that should not require it. The next section shows how “read only” access can solve this problem.

$$
\begin{array}{lcl}
\mathit{Expr} & = & \mathit{Var}\ \mathit{Ix} \mid \mathit{Const}\ \mathit{Val} \mid \mathit{Plus}\ \mathit{Expr}\ \mathit{Expr} \\
\mathit{Com} & = & \mathit{Asgn}\ \mathit{Ix}\ \mathit{Expr} \mid \mathit{Seq}\ \mathit{Com}\ \mathit{Com} \mid \mathit{If}\ \mathit{Expr}\ \mathit{Com}\ \mathit{Com} \\
\mathit{Prog} & = & \mathit{Do}\ \mathit{Com}\ \mathit{Expr} \\
\\
\mathit{expr} & : & \mathit{Expr} \to \mathit{Arr} \to \mathit{Val} \\
\mathit{expr}\ (\mathit{Var}\ i)\ a & = & \mathit{lookup}\ i\ a \\
\mathit{expr}\ (\mathit{Const}\ v)\ a & = & v \\
\mathit{expr}\ (\mathit{Plus}\ e_0\ e_1)\ a & = & \mathit{expr}\ e_0\ a + \mathit{expr}\ e_1\ a \\
\\
\mathit{com} & : & \mathit{Com} \to \text{¡}\mathit{Arr} \to \text{¡}\mathit{Arr} \\
\mathit{com}\ (\mathit{Asgn}\ i\ e)\ a & = & \textbf{let!}\ (a)\ v = \mathit{expr}\ e\ a\ \textbf{in}\ \mathit{update}\ i\ v\ a \\
\mathit{com}\ (\mathit{Seq}\ c_0\ c_1)\ a & = & \mathit{com}\ c_1\ (\mathit{com}\ c_0\ a) \\
\mathit{com}\ (\mathit{If}\ e\ c_0\ c_1)\ a & = & \textbf{let!}\ (a)\ v = \mathit{expr}\ e\ a\ \textbf{in} \\
 & & \textbf{if}\ v = 0\ \textbf{then}\ \mathit{com}\ c_0\ a\ \textbf{else}\ \mathit{com}\ c_1\ a \\
\\
\mathit{prog} & : & \mathit{Prog} \to \mathit{Val} \\
\mathit{prog}\ (\mathit{Do}\ c\ e) & = & \mathit{dealloc}\ (\mathit{expr}'\ e\ (\mathit{com}\ c\ \mathit{alloc})) \\
\\
\mathit{expr}' & : & \mathit{Expr} \to \text{¡}\mathit{Arr} \to \mathit{Val} \otimes \text{¡}\mathit{Arr} \\
\mathit{expr}'\ e\ a & = & \textbf{let!}\ (a)\ v = \mathit{expr}\ e\ a\ \textbf{in}\ (v, a)
\end{array}
$$

Figure 6: A simple interpreter, version 3

5.3 "Read only" arrays

If "read only" access is allowed, then the array operations are as follows:

$$
\begin{array}{lcl}
\mathit{alloc} & : & \text{¡}\mathit{Arr}, \\
\mathit{lookup} & : & \mathit{Ix} \to \mathit{Arr} \to \mathit{Val}, \\
\mathit{update} & : & \mathit{Ix} \to \mathit{Val} \to \text{¡}\mathit{Arr} \to \text{¡}\mathit{Arr}, \\
\mathit{dealloc} & : & \mathit{Val} \otimes \text{¡}\mathit{Arr} \to \mathit{Val}.
\end{array}
$$

The type ¡*Arr* corresponds to write access, and the type *Arr* corresponds to read access. The *alloc*, *update*, and *dealloc* operations are just as in the previous section; while the *lookup* operation is just as it was originally in the conventional type system.

A third version of the interpreter using "read only" access is shown in Figure 6. Here the *com* semantic function has the same structure it had in version 2, while the *expr* semantic function has the same structure it had in version 1. In particular, the spurious imposition of an evaluation order on summands, present in version 2, has gone away. Note that the type of *expr* refers to *Arr*, not ¡*Arr*, making it explicit that *expr* never changes the array that it is passed.

One unusual feature of this simple semantics is that it does, in effect, discard the store (the command is executed; the expression is evaluated in the resulting store; and then the store is discarded and only the value of the expression is kept). Note that the "no discard" rule does not mean that the store cannot be discarded, simply that the store must be discarded *explicitly*. As a result, the definition of *prog* in the final program (Figure 6) is more longwinded than the equivalent definition in the original (Figure 4). Depending on your point of view, this difference may be regarded as a drawback or as a benefit.

5.4 Arrays of arrays

Finally, we briefly consider how to handle arrays of arrays. Let $¡Arr$ and Arr be as in the previous section: arrays taking indexes of type Ix into values of type Val. Let $¡Arr2$ and $Arr2$ be arrays taking indexes of type Ix into values of type $¡Arr$. The new feature here is that the values of $Arr2$ are linear rather than nonlinear.

Let the operations on the type Arr be as in the previous section. In addition, the operations on $Arr2$ are:

$$\begin{array}{lcl} alloc2 & : & ¡Arr2 \\ lookup2 & : & Ix \to Arr2 \to Arr \\ update2 & : & Ix \to (¡Arr \to ¡Arr) \to ¡Arr2 \to ¡Arr2 \\ dealloc2 & : & Val \otimes ¡Arr2 \to Val \end{array}$$

To compute the value of $a2[k][l]$ and store this in location $a2[i][j]$ one writes:

$$\begin{array}{l} \mathbf{let!}\ (a2)\ v = lookup\ l\ (lookup2\ k\ a2)\ \mathbf{in} \\ update2\ i\ (update\ j\ v)\ a2\ . \end{array}$$

6 Conclusions

Much further work remains to be done. Among the important questions are the following:

- How do linear types fit in with the Hindley-Milner-Damas approach to polymorphism and type inference?
- How can linear types best support i/o and interactive functional programs?
- Girard's Linear Logic contains nothing like the "**let!**" construct. Is there a nice theoretical justification for this construct?
- Deforestation [Wad88] has a linearity constraint, and the "blazed" types described there might be subsumed by "of course" types. Can linear types aid program transformation?

Finally, more practical experience is required before we can evaluate the likelihood that linear types *will* change the world.

Acknowledgements. For discussions relating to this work, and comments on it, I am grateful to Steve Blott, Kei Davis, Sören Holmström, Paul Hudak, Simon Jones, Yves Lafont, Simon Peyton Jones, David Schmidt, the Glasgow FP Group, and the members of IFIP WG 2.8.

References

[Blo89] A. Bloss, Path analysis: using order-of-evaluation information to optimize lazy functional languages. Ph.D. thesis, Yale University, Department of Computer Science, 1989.

[BHY89] A. Bloss, P. Hudak, and J. Young, An optimising compiler for a modern functional language. *Computer Journal*, **32**(2):152–161, April 1989.

[DB76] J. Darlington and R. M. Burstall, A system which automatically improves programs. *Acta Informatica*, **6**:41–60, 1976.

[DM82] L. Damas and R. Milner, Principal type schemes for functional programs. In *Proceedings of the 9'th Annual Symposium on Principles of Programming Languages*, Albuquerque, N.M., January 1982.

[Gir72] J.-Y. Girard, *Interprétation functionelle et élimination des coupures dans l'arithmétique d'ordre supérieure*. Ph.D. thesis, Université Paris VII, 1972.

[Gir86] J.-Y. Girard, The system F of variable types, fifteen years later. *Theoretical Computer Science*, **45**:159–192, 1986.

[Gir87] J.-Y. Girard, Linear logic. *Theoretical Computer Science*, **50**:1–102, 1987.

[GLT89] J.-Y. Girard, Y. Lafont, and P. Taylor, *Proofs and Types*. Cambridge University Press, 1989.

[Hin69] R. Hindley, The principal type scheme of an object in combinatory logic. *Trans. Am. Math. Soc.*, **146**:29–60, December 1969.

[Hol88] S. Holmström, A linear functional language. Draft paper, Chalmers University of Technology, 1988.

[Hud86] P. Hudak, A semantic model of reference counting and its abstraction. In *Proceedings ACM Conference on Lisp and Functional Programming*, August 1986.

[HG89] P. Hudak and J. Guzmán, Taming side effects with a single-threaded type system. Draft paper, Yale University, December 1989.

[HW88] P. Hudak and P. Wadler, editors, *Report on the Functional Programming Language Haskell*. Technical Report YALEU/DCS/RR656, Yale University, Department of Computer Science, December 1988; also Technical Report, Glasgow University, Department of Computer Science, December 1988.

[Laf88] Y. Lafont, The linear abstract machine. *Theoretical Computer Science*, **59**:157–180, 1988.

[Mil78] R. Milner, A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, **17**:348–375, 1978.

[Rey74] J. C. Reynolds, Towards a theory of type structure. In B. Robinet, editor, *Proc. Colloque sur la Programmation*, LNCS 19, Springer-Verlag.

[Rey83] J. C. Reynolds, Types, abstraction, and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, 513–523, North-Holland, Amsterdam.

[Rey85] J. C. Reynolds, Three approaches to type structure. In *Mathematical Foundations of Software Development*, LNCS 185, Springer-Verlag, 1985.

[Sch82] D. A. Schmidt, Denotational semantics as a programming language. Internal report CSR-100, Computer Science Department, University of Edinburgh, 1982.

[Sch85] D. A. Schmidt, Detecting global variables in denotational specifications. *ACM Trans. on Programming Languages and Systems*, **7**:299–310, 1985. (Also Internal Report CSR-143, Computer Science Department, University of Edinburgh, September 1983.)

[Tur85] D. A. Turner, Miranda: A non-strict functional language with polymorphic types. In *Proceedings of the 2'nd International Conference on Functional Programming Languages and Computer Architecture*, Nancy, France, September 1985. LNCS 201, Springer-Verlag, 1985.

[Wad88] P. Wadler, Deforestation: transforming programs to eliminate trees. In *European Symposium on Programming*, LNCS 300, Springer-Verlag, 1988.