Is there a use for linear logic?

Philip Wadler
University of Glasgow*

March 1991

Abstract

Past attempts to apply Girard's linear logic have either had a clear relation to the theory (Lafont, Holmström, Abramsky) or a clear practical value (Guzmán and Hudak, Wadler), but not both. This paper defines a sequence of languages based on linear logic that span the gap between theory and practice. Type reconstruction in a linear type system can derive information about sharing. An approach to linear type reconstruction based on *use types* is presented. Applications to the *array update* problem are considered.

1 Introduction

Storage reuse; single threading; in-place update; sharing analysis; linearity: any problem with so many names must be important. Girard's linear logic has intrigued computer scientists with its promise to focus new light on this old subject. (It also hints at enlightenment with regard to parallelism, but that's a topic for other papers.)

Attempts to apply linear logic fall into two camps, the *theoreticians* and the *practitioners*. On the theoretical side sit Lafont [Laf88], Holmström [Hol88], and Abramsky [Abr90]. Their languages correspond to linear logic in a precise way, via the Curry-Howard isomorphism. On the practical side sit Guzmán and Hudak [GH90] and Wadler [Wad90]. Their languages are inspired by linear logic, but the connection is of a vaguer and looser kind. (Another practioner is Wakeling, who has implemented Wadler's system [Wak90].)

The goal of this paper is to build a bridge between the camps. It begins by defining a language that corresponds closely to linear logic; it is a slight variation of a language described by Abramsky [Abr90]. The suitability of this language for practical purposes is examined, and it is found to be lacking in some respects. Variants of the language are defined to remedy these shortcomings. One variant, motivated by the standard encoding of intuitionistic logic into linear logic, is found to closely resemble aspects of [GH90]. Another variant, motivated by restrictions on the Promotion and Dereliction rules of linear logic, is found to closely resemble aspects of [Wad90].

A desirable property of a type system is the possession of a most general, or principal, type for each typable term. The basic linear type system appears not to have this property. To achieve it, linear types are augmented with a notion of *use* variables, which indicate presence or absence of the *of course* operator of linear logic. A similar, but more general, form of *use* appears in [GH90].

The main application of linear logic that we will consider is the *array update* problem. A value computed in an implementation of a functional language is *linear* if there is exactly one pointer to it. It is safe to deallocate the storage occupied by a linear value as soon as the value has been accessed, or, equivalently, to reuse the storage. This is particularly useful in the case of arrays, which are finite maps from indices to values. A common operation is *update*: given an array, an index, and a value, return an array identical to that given ex-

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

Appears in *ACM/IFIP Symposium on Partial Evaluation and Semantics Based Program Manipulation* (PEPM), Yale University, June 1991.

$$\begin{tikzcd}
\text{Intuitionistic types (§2, Fig. 2)} \arrow[d] & \\
\text{Linear types (§3, Fig. 3)} \arrow[d] \arrow[r] & \text{Steadfast types (§7, Fig. 8)} \arrow[d] \\
\text{Use types (§5, Fig. 5)} \arrow[d] & \text{Steadfast use types (§7)} \arrow[d] \\
\text{Standard types (§6, Fig. 6)} & \text{Steadfast standard types (§7, Fig. 9)}
\end{tikzcd}$$

Figure 1: A road map

cept the given index maps to the given value. If implemented naively, this requires copying the entire array; but if the array is linear, it may be implemented by overwriting the store at the location corresponding to the given index.

Interest in the update problem dates back to the pioneering work of Darlington and Burstall [DB76]. Syntactic constraints that guarantee the safety of in-place update were suggested by Schmidt [Sch85], and a variety of semantic analysis techniques have been proposed, notably by Hudak and Bloss [Hud86, Blo89].

The thrust of this paper, as well as of the other attempts to apply linear logic [GH90, Wad90], is to use type reconstruction rather than semantic analysis to determine when in-place update is safe. The prototypical example of type reconstruction is the Hindley-Milner algorithm [Hin69, Mil78, DM82]; another example from which we will draw inspiration is Mitchell's work on subtypes [Mit84, Mit91]. A related notion is the use of type reconstruction to perform strictness analysis [KM89, Wri89].

To simplify the presentation, "let" terms will not be considered, although they play a key role in polymorphic type inference. A similar simplifying assumption is made by Mitchell [Mit84, Mit91] and by Guzmán and Hudak [GH90]. Adding "let" terms appears to require more bookkeeping, but does not appear to introduce fundamentally new issues.

The treatment given here says nothing about "read only" types, although they are central to the work in [GH90] and [Wad90]. To find a more formal relation between "read only" types and Girard's linear logic remains an interesting task for future work.

The remainder of this paper is organised as follows.

Section 2 presents the usual simply typed lambda calculus and its correspondence to intuitionistic logic. The role of the Weakening and Contraction rules is stressed.

Section 3 modifies this language to instead correspond to linear logic. The Promotion and Dereliction rules are presented, and turn out to have surprising (and not entirely pleasant) consequences. An application to the array update problem is given.

Section 4 describes the standard mapping of intuitionistic logic into linear logic.

Section 5 explains why the simple linear system is unsuited to finding principal types, and introduces a variant that is better suited, based on the notion of *use* types.

Section 6 focuses on a *standard* form for linear types, motivated by the encoding of Section 4. This system possesses principal types, and a type reconstruction algorithm is presented. This variant turns out to correspond closely to [GH90].

Section 7 presents a different branch of development for the linear type system. The use of Promotion and Dereliction is restricted, resulting in a *steadfast* type system. Use types, standard types, and the type reconstruction algorithm adopt straightforwardly to steadfast types. This variant turns out to correspond closely to [Wad90].

A road map through the various type systems appears in Figure 1.

2 Intuitionistic types

We begin with the simply typed lambda calculus, which corresponds to intuitionistic logic. The presentation differs from some others in that it focuses attention on the structural rules, which assume a particular significance for linear logic.

A *type* is a type variable, function, product, or sum:

$$\begin{array}{l} T, U, V \ ::= \\ \quad X \mid (U \to V) \mid (U \times V) \mid (U + V). \end{array}$$

Let X, Y, Z range over type variables, and T, U, V range over types.

A *term* is an individual variable or an introducer or eliminator of a function, product, or sum type:

$$\begin{array}{l} t, u, v \ ::= \\ \quad x \\ \quad \mid (\lambda x.\, v) \mid (t\, u) \\ \quad \mid (u, v) \mid (\text{case } t \text{ of } \{(x, y) \to w\}) \\ \quad \mid (\mathit{inl}\, u) \mid (\mathit{inr}\, v) \\ \quad \mid (\text{case } t \text{ of } \{\mathit{inl}\, x \to u;\ \mathit{inr}\, y \to v\}). \end{array}$$

Let x, y, z range over individual variables and t, u, v range over terms.

An *assumption list* pairs individual variables with types:

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

Here $n \geq 0$ and each of the x_i is distinct. Let A and B range over assumption lists. The order of entries in an assumption list is, as we shall see, irrelevant.

A *typing judgement* takes the form

$$A \vdash t : T.$$

This may be read "Under assumptions A the term t has type T."

The inference rules for assigning types to terms are shown in Figure 2. The rules come in two families. The first four rules are *structural*, and deal with variables. The remaining rules are *logical*, and deal with all other terms; like the terms, the rules come in introduction and elimination pairs.

Concatenation of assumption lists is written with a comma, and implies that the two lists refer to disjoint sets of individual variables. Hence, in the $\to$-I rule, the appearance of $A,\ x : U$ means that x cannot also appear in A; this sometimes appears as an explicit side-condition in other presentations. In the $\to$-E rule, the appearance of A, B means that t and u must have disjoint sets of free variables. This may seem too restrictive, but we shall see that Contraction allows variables to be shared in the usual way.

The structural rules allow fine control over the usage of variables; and we will take advantage of this control shortly, to design a linear type system. The Id rule is like a tautology. The Exchange rule encodes the fact that order of variables in an assumption list is irrelevant.

The Weakening rule allows a variable to be discarded. In other presentations of simply typed lambda calculus, the rule for variables looks like this:

$$\frac{}{A,\ x : U \vdash x : U.}$$

This is mimicked in the system given here as follows: use the Id rule followed by one application of weakening for each assumption in A.

The Contraction rule allows a variable to be shared. The notation $v^{z;z}_{x,y}$ stands for the term v with each free occurrence of x and y replaced by z. Here x and y are introduced for technical convenience, to maintain the invariant that a variable appears in an assumption list only once. In other presentations of simply typed lambda calculus, the rule for function application looks like this:

$$\frac{A \vdash t : (U \to V) \qquad A \vdash u : U}{A \vdash (t\, u) : V}.$$

This is mimicked in the system given here as follows. Let $\mathbf{z}$ be a list of all the variables in the domain of A, and let $\mathbf{x}$ and $\mathbf{y}$ be two lists of fresh variable names of the same length as $\mathbf{z}$. Then we have:

$$\to\text{-E}\ \frac{A^{\mathbf{x}}_{\mathbf{z}} \vdash t^{\mathbf{x}}_{\mathbf{z}} : (U \to V) \qquad A^{\mathbf{y}}_{\mathbf{z}} \vdash u^{\mathbf{y}}_{\mathbf{z}} : U}{\text{Cont}\ \dfrac{A^{\mathbf{x}}_{\mathbf{z}}, A^{\mathbf{y}}_{\mathbf{z}} \vdash (t^{\mathbf{x}}_{\mathbf{z}}\, u^{\mathbf{y}}_{\mathbf{z}}) : V)}{A \vdash (t\, u) : V.}}$$

The last line applies Contraction once for each variable in A; observe that $(t^{\mathbf{x}}_{\mathbf{z}}\, u^{\mathbf{y}}_{\mathbf{z}})^{\mathbf{z},\mathbf{z}}_{\mathbf{x},\mathbf{y}} = (t\, u)$.

$$\text{Id}\ \frac{}{x : U \vdash x : U} \qquad \text{Exchange}\ \frac{A, x : U, y : V, B \vdash w : W}{A, y : V, x : U, B \vdash w : W}$$

$$\text{Weakening}\ \frac{A \vdash v : V}{A,\ x : U \vdash v : V} \qquad \text{Contraction}\ \frac{A,\ x : U,\ y : U \vdash v : V}{A,\ z : U \vdash v^{z,z}_{x,y} : V}$$

$$\to\text{-I}\ \frac{A,\ x : U \vdash v : V}{A \vdash (\lambda x.\, v) : (U \to V)} \qquad \to\text{-E}\ \frac{A \vdash t : (U \to V) \qquad B \vdash u : U}{A,\ B \vdash (t\, u) : V}$$

$$\times\text{-I}\ \frac{A \vdash u : U \qquad B \vdash v : V}{A,\ B \vdash (u, v) : (U \times V)} \qquad \times\text{-E}\ \frac{A \vdash t : (U \times V) \qquad B,\ x : U,\ y : V \vdash w : W}{A, B \vdash (\text{case } t \text{ of } \{(x, y) \to w\}) : W}$$

$$+\text{-I}\ \frac{A \vdash u : U}{A \vdash (\mathit{inl}\, u) : (U + V)} \qquad \frac{A \vdash v : V}{A \vdash (\mathit{inr}\, v) : (U + V)}$$

$$+\text{-E}\ \frac{A \vdash t : (U + V) \qquad B,\ x : U \vdash u : W \qquad B,\ y : V \vdash v : W}{A, B \vdash (\text{case } t \text{ of } \{\mathit{inl}\, x \to u;\ \mathit{inr}\, y \to v\}) : W}$$

Figure 2: Rules for intuitionistic types

The Curry-Howard isomorphism. Take an inference rule for simply typed lambda calculus and erase all the variables and terms: the result is a rule of intuitionistic logic. For instance, the →-I and →-E rules yield the deduction rule and modus ponens:

$$\to\text{-I}\ \frac{A, U \vdash V}{A \vdash (U \to V)},$$

$$\to\text{-E}\ \frac{A \vdash (U \to V) \qquad A \vdash U}{A \vdash V}.$$

One reads → as “implies”, × as “and”, + as “or”. Every typing of a term maps into a proof of the corresponding proposition, and every proof of a proposition maps into a typing of a corresponding term. This remarkable correspondence between types and logic was observed by Curry [Cur58] and refined by Howard [How80].

It is instructive to consider the effect of the Curry-Howard isomorphism on the structural rules:

$$\text{Id}\ \frac{}{U \vdash U}, \qquad \text{Exch}\ \frac{A,\ U,\ V,\ B \vdash W}{A,\ V,\ U,\ B \vdash W},$$

$$\text{Weak}\ \frac{A \vdash W}{A,\ U \vdash W}, \qquad \text{Cont}\ \frac{A,\ U,\ U \vdash W}{A,\ U \vdash W}.$$

Id is an obvious tautology. Exchange says that the the order of hypotheses is irrelevant. Weakening says that a hypothesis can be ignored. Contraction says that a hypothesis can be used twice.

Weakening corresponds to discarding the value of a variable, and Contraction corresponds to using it twice. These are exactly the things we wish to restrict, and linear types provide a way of doing so.

$$\text{Id}\ \frac{}{x : U \vdash x : U} \qquad \text{Exchange}\ \frac{A, x : U, y : V, B \vdash w : W}{A, y : V, x : U, B \vdash w : W}$$

$$\text{Promotion}\ \frac{(!A) \vdash v : V}{(!A) \vdash v : (!V)} \qquad \text{Dereliction}\ \frac{A,\ x : U \vdash v : V}{A,\ x : (!U) \vdash v : V}$$

$$\text{Weakening}\ \frac{A \vdash v : V}{A,\ x : (!U) \vdash v : V} \qquad \text{Contraction}\ \frac{A,\ x : (!U),\ y : (!U) \vdash v : V}{A,\ z : (!U) \vdash v^{z,z}_{x,y} : V}$$

$$\multimap\text{-I}\ \frac{A,\ x : U \vdash v : V}{A \vdash (\lambda x.\, v) : (U \multimap V)} \qquad \multimap\text{-E}\ \frac{A \vdash t : (U \multimap V) \qquad B \vdash u : U}{A,\ B \vdash (t\, u) : V}$$

$$\otimes\text{-I}\ \frac{A \vdash u : U \qquad B \vdash v : V}{A,\ B \vdash (u, v) : (U \otimes V)} \qquad \otimes\text{-E}\ \frac{A \vdash t : (U \otimes V) \qquad B,\ x : U,\ y : V \vdash w : W}{A, B \vdash (\mathsf{case}\ t\ \mathsf{of}\ \{(x, y) \to w\}) : W}$$

$$\oplus\text{-I}\ \frac{A \vdash u : U}{A \vdash (\mathit{inl}\, u) : (U \oplus V)} \qquad \frac{A \vdash v : V}{A \vdash (\mathit{inr}\, v) : (U \oplus V)}$$

$$\oplus\text{-E}\ \frac{A \vdash t : (U \oplus V) \qquad B,\ x : U \vdash u : W \qquad B,\ y : V \vdash v : W}{A, B \vdash (\mathsf{case}\ t\ \mathsf{of}\ \{\mathit{inl}\, x \to u;\ \mathit{inr}\, y \to v\}) : W}$$

Figure 3: Rules for linear types

3 Linear types

What a simple world it would be, if only it were not for Weakening and Contraction!

Without Weakening, no value could be discarded. Lazy evaluation is no longer required, because no computation can be ignored. Without Contraction, no value could be duplicated. Overwriting (one of the trickiest bits of graph reduction) is no longer required, because no computation can be shared. Each value has exactly one pointer to it. Garbage collection is no longer required, because each value can be deallocated (or reused) at the sole place it is accessed.

The only defect in this marvelously simple world is that it is too simple: it is impossible to write even the function that squares a number.

We cannot afford to get rid of Weakening and Contraction outright, but we can at least bell the cat. The linear type system introduces a new type constructor, written "!" and pronounced "of course!". Only variables with types of the form $(!U)$ may have Weakening or Contraction applied to them. Types in the form $(!U)$ will be called *nonlinear*, and all other types will be called *linear*; but the name *linear type* system encompasses both sorts of types. The intuition is that values of nonlinear type may be shared, while others may not; so the presence of sharing is explicitly indicated by "!". (The real story is a bit more complex than this, as we shall see.)

Linear types. In the linear type system, functions are written with $\multimap$ instead of $\rightarrow$, products with $\otimes$ instead of $\times$, and sums with $\oplus$ instead of $+$. Thus, the full grammar of linear types is given by:

$$\begin{aligned}&T, U, V \; ::= \\ &\quad X \mid (!U) \mid (U \multimap V) \mid (U \otimes V) \mid (U \oplus V).\end{aligned}$$

In Girard's linear logic there is a second form of linear product, written &, but that will not be used here.

The inference rules for assigning types to terms are shown in Figure 3. These rules are identical to those given previously (modulo a change in symbol names), with the exception of the four rules concerned with "!" types: Promotion, Dereliction, Weakening, and Contraction.

Promotion is an !-Introduction rule, while Dereliction, Weakening, and Contraction are !-Elimination rules. The introduction rule (Promotion) states that a value may be shared if all of its free variables may be shared. Here the notation $(!A)$ stands for an assumption list in the form $x_1 : (!U_1), \ldots, x_n : (!U_n)$. A value should not be shared if it has a free variable of linear type, because sharing the value would, indirectly, create shared pointers to the linear value, and this should not happen. The three elimination rules correspond to possible uses of a value of type $(!U)$: it may be used exactly once (Dereliction), not at all (Weakening), or many times (Contraction).

The system described here is nearly identical to that given by Abramsky [Abr90], the only difference being that Abramsky introduces four new term forms that correspond to the Promotion, Dereliction, Weakening, and Contraction rules, just as lambda abstraction and application correspond to the $\rightarrow$-I and $\rightarrow$-E rules. These term forms are omitted here because our interest is in assigning a linear type to a term expressed in the usual lambda calculus. Nonetheless, this type system corresponds precisely to the linear logic of Girard, via the Curry-Howard isomorphism.

Examples. Here is one detailed example to show how the Dereliction and Contraction rules are used. Consider the typing:

$$\lambda f.\, \lambda x.\, f\,(f\,x) : !(X \multimap X) \multimap X \multimap X.$$

(Parentheses are omitted according to convention: $\multimap$ associates to the right, and ! binds more tightly than $\multimap$.) First, use Id and $\multimap$-E to derive a type for the term $f\,(f'\,x)$, where one occurrence of f has been replaced by f'. Second, the criticial part of the derivation. Write F as an abbreviation for the type $(X \multimap X)$. Then use Dereliction (twice) followed by Contraction:

$$\dfrac{\dfrac{f : F,\, f' : F,\, x : X \vdash f\,(f'\,x) : X}{f : !F,\, f' : !F,\, x : X \vdash f\,(f'\,x) : X}\,\text{Der}}{f : !F,\, x : X \vdash f\,(f\,x) : X.}\,\text{Cont}$$

Third, use $\multimap$-I twice to yield the typing given above. Here Contraction is essential because f is used twice, and Dereliction is essential to strip the "!" off the type of f so that the $\multimap$-E rule applies.

The combinators provide an instructive set of examples, as shown in Figure 4. Combinators **I**, **B**, and **C** can be typed without recourse to "!"; the $!Y$ in the type of **K** allows Weakening, required because y is never used; and the $!X$ in the type of **S** allows Contraction, required because x is used twice. A term containing only variables, function abstractions, and function applications can be typed without recourse to "!" exactly when it can be can be compiled into combinators using **I**, **B**, and **C** without recourse to **K** and **S**.

A term can be assigned more than one type. Consider the term

$$\lambda f.\, \lambda x.\, f\,x.$$

Among the types for this term are the following:

$$\begin{gathered}(X \multimap Y) \multimap X \multimap Y, \\ (X \multimap Y) \multimap !X \multimap Y, \\ !(X \multimap Y) \multimap X \multimap Y, \\ !(X \multimap Y) \multimap !X \multimap !Y.\end{gathered}$$

The first typing requires only the Id, $\multimap$-E, and $\multimap$-I rules. The second and third typings use Dereliction on x and f respectively. The fourth typing uses Dereliction on both f and x, and then Promotion on $(f\,x)$. A type that this term does *not* possess is:

$$(X \multimap Y) \multimap !X \multimap !Y.$$

$$\begin{array}{rcccl}
\lambda x.\, x & = & \mathbf{I} & : & X \multimap X, \\
\lambda g.\, \lambda f.\, \lambda x.\, g\,(f\,x) & = & \mathbf{B} & : & (Y \multimap Z) \multimap (X \multimap Y) \multimap X \multimap Z, \\
\lambda f.\, \lambda x.\, \lambda y.\, f\,y\,x & = & \mathbf{C} & : & (Y \multimap X \multimap Z) \multimap X \multimap Y \multimap Z, \\
\lambda x.\, \lambda y.\, x & = & \mathbf{K} & : & X \multimap !Y \multimap X, \\
\lambda f.\, \lambda g.\, \lambda x.\, f\,x\,(g\,x) & = & \mathbf{S} & : & (!X \multimap Y \multimap Z) \multimap (!X \multimap Y) \multimap !X \multimap Z.
\end{array}$$

Figure 4: Example: Combinators

Promotion cannot be used here, because in the assumption list for $f\,x$ there will be a variable (namely f) whose type does not begin with "!".

The existence of multiple typings raises the question: is there a "most general" type that can be assigned to a term? This is the subject of Sections 5 and 6.

What linearity does and doesn't mean. The guiding intuition behind this type system is that sharing is explicitly indicated by the use of nonlinear types. One would expect, therefore, that a value of linear type will have exactly one pointer to it. Unfortunately, this is not quite what happens.

The difficulty is with the Dereliction rule. On the surface of it, Dereliction seems eminently reasonable: it expresses the idea that one thing you may do with a variable of nonlinear type is to use it exactly once. Similarly, Weakening expresses the idea that you may use it not at all; and Contraction expresses the idea that you may use it many times.

Without Dereliction there would be no way to remove "!" from a type, and hence no way to use any of the elimination rules on nonlinear types. Consider again the typing of:

$$\lambda f.\, \lambda x.\, f\,(f\,x) : !(X \multimap X) \multimap X \multimap X.$$

Clearly f is nonlinear. But in order to type each of the applications of f, we must treat f as if it has type $X \multimap X$, not type $!(X \multimap X)$. (The $\multimap$-E rule can be used with the former type but not the latter.) This is what Dereliction enables us to do.

But here's the rub: Dereliction means one cannot guarantee that a value of linear type always has exactly one pointer to it. In the example above, f has linear type at the point of application, but since f "really" has nonlinear type, there may be more than one pointer to it. As a further example, consider the valid typing:

$$\begin{array}{l}\lambda f.\, \lambda g.\, \lambda x.\, (f\,x, g\,x) : \\ \quad (X \multimap Y) \multimap (X \multimap Z) \multimap !X \multimap (Y \otimes Z).\end{array}$$

Here, f and g each take arguments of linear type. But, thanks to Dereliction, they can each be passed the argument x of nonlinear type.

Does this mean that linearity is useless for practical purposes? Not completely. Dereliction means we cannot guarantee *a priori* that a variable of linear type has exactly one pointer to it. But if we know this by other means, then linearity guarantees that the pointer will not be duplicated or discarded.

Arrays. In-place update of arrays can be supported as follows. Assume a linear array type Arr, with indices of type $!Ix$ and values of type $!\,Val$ (the index and value types must be nonlinear, because we can access each several times). The following operations are provided:

$$\begin{array}{ll}
\mathit{block} & : !(!\,\mathit{Val} \multimap (\mathit{Arr} \multimap (X \otimes \mathit{Arr})) \multimap X), \\
\mathit{lookup} & : !(!\mathit{Ix} \multimap \mathit{Arr} \multimap (!\,\mathit{Val} \otimes \mathit{Arr})), \\
\mathit{update} & : !(!\mathit{Ix} \multimap !\,\mathit{Val} \multimap \mathit{Arr} \multimap \mathit{Arr}).
\end{array}$$

The *block* operation creates an array, initialises each location to the given value, and passes it to the given function; this function returns the final array paired with a value of type X; the array is deallocated and the value of type X is returned.

The function is passed the only pointer to the array, and the array has linear type, so it is guaranteed that the pointer is not duplicated and in-place update is safe. Pointers of linear type cannot be discarded, so the *lookup* operation must return, in addition to the value looked up, the pointer to the given array. This is just as well: since there is only one pointer, if it was not returned then one lookup would be all one could do. As it is, passing the array around in this way enforces a very strict form of single threading, where each operation on the array is sequentialised.

In practice one would prefer a system where only update operations are sequentialised but lookups can occur in parallel. The "read only" types of [GH90] and [Wad90] allow this, but such types appear to go beyond the basics of linear logic, and are outside the scope of this paper.

It would *not* work to create arrays with an operation:

$$alloc : !(!\,Val \multimap Arr).$$

The problem is this: Promotion can be applied to create a value of type $!Arr$, pointers to the created array can be copied, and now Dereliction causes difficulties as described previously. In the system of [GH90], the sites at which an array is created are treated as introducing a new bound variable of linear type. This somewhat odd restriction appears to have been introduced in order to prevent a problem analogous to the one described here.

So, with some care, linear types can support in-place update despite the problems introduced by Promotion and Derelication. But one might wonder: is it possible to restrict these rules so as to guarantee that values of linear type really do have exactly one pointer to them? This is the subject of Section 7.

4 Encoding intuitionistic types as linear types

The linear system refines the notion of type, but not of typeability. Every typing judgement in the intuitionistic system (Figure 2) can be mapped into a corresponding judgement in the linear system (Figure 3), and vice versa. A corollary of this is that a term is typable in the intuitionistic system if and only if it is typable in the linear system.

In one direction, the mapping is trivial. A linear judgement maps into a corresponding intuitionistic judgement as follows: remove each occurrence of "!", and replace occurrences of $\multimap$, $\otimes$, $\oplus$ with corresponding occurrences of $\rightarrow$, $\times$, $+$. If the linear judgement is derivable, so is the corresponding intuitionistic one.

In the reverse direction, there are several possible ways to encode an intuitionistic type as a linear type. The simplest just adds "!" symbols everywhere, so that, for instance, $X \rightarrow Y$ encodes as $!((!X) \multimap (!\,Y))$. But there is a standard encoding, due to Girard, that is somewhat more parsimonious in its use of "!". For instance, it encodes $X \rightarrow Y$ as $(!X) \multimap Y$.

If T is an intuitionistic type, write T° for its encoding as a linear type:

$$\begin{array}{rcl} X^\circ & = & X, \\ (U \rightarrow V)^\circ & = & (!\,U^\circ) \multimap V^\circ, \\ (U \times V)^\circ & = & (!\,U^\circ) \otimes (!\,V^\circ), \\ (U + V)^\circ & = & (!\,U^\circ) \oplus (!\,V^\circ). \end{array}$$

Function arguments are given "!" types, but function results are not. For products and sums, the mapping is more straightforward: each component is given a "!" type. (In Girard's original encoding, $\times$ maps into & rather than $\otimes$; that's not done here because we are eschewing the use of &.)

The encoding extends to judgements as follows:

$$\begin{array}{l} (x_1 : U_1, \ldots, x_n : U_n \vdash v : V)^\circ \;= \\ \qquad\qquad x_1 : !\,U_1^\circ, \ldots, x_n : !\,U_n^\circ \vdash v : V^\circ. \end{array}$$

Each variable is given a "!" type, but the term itself is not. When A stands for the assumption list on the left above, write $!A^\circ$ for the assumption list on the right.

If the judgement $A \vdash v : V$ is derivable in the intuitionistic system, then the corresponding judgement $!A^\circ \vdash v : V^\circ$ is derivable in the linear system. This is shown by providing an encoding of each intuitionistic rule into one or more linear rules. The only tricky cases are those for the Id and $\rightarrow$-E rules. The intuitionistic Id rule maps into a combination of linear Id and Dereliction:

$$\dfrac{\dfrac{}{x : U^\circ \vdash x : U^\circ}\,\text{Id}}{x : !U^\circ \vdash x : U^\circ.}\,\text{Der}$$

The $\rightarrow$-E rule maps into a combination of Promotion and the $\multimap$-E rule:

$$\dfrac{!A^\circ \vdash t : (!U^\circ \multimap V^\circ) \qquad \dfrac{!B^\circ \vdash u : U^\circ}{!B^\circ \vdash u : !U^\circ}\,\text{Pro}}{!A^\circ, !B^\circ \vdash (t\, u) : V^\circ.}\,\multimap\text{-E}$$

The $\rightarrow$-I rule is quite straightforward, and maps directly into the $\multimap$-I rule:

$$\dfrac{!A^\circ, x : !U^\circ \vdash v : V^\circ}{!A^\circ \vdash (\lambda x.\, v) : (!U^\circ \multimap V^\circ).}\,\multimap\text{-I}$$

The maps for the remaining rules are equally straightforward.

This encoding does not introduce the minimal number of "!" symbols. For instance, consider the term $(\lambda x.\, x)$. This term has the intuitionsistic type $(X \rightarrow X)$, and the encoding yields the linear type $(!X \multimap X)$. However, this term also has the linear type $(X \multimap X)$, which contains no "!" instead of one.

Although the encoding is not minimal, it does guarantee that every typable term has a typing where "!" does not appear in certain places (namely, at the very outside and in the range of a linear function). This will motivate the definition of *standard* linear types in Section 6.

5 Use types

In the intuitionistic type system, each term can be assigned a most general, or *principal*, typing: one of which all other typings are substitution instances. This is the basis of the Hindley-Milner system.

We might similarly ask: can each term be assigned a most general typing under the linear type system? The answer is no. The identity function $\lambda x.\, x$ has both the type $X \multimap X$ and type $!X \multimap X$, and neither of these is a substitution instance of the other.

That's too bad, but perhaps the situation can be saved by changing the definition of "most general". For instance, we might introduce a notion of subtype. Write $U \leq V$ to indicate that U is a subtype of V,that is, whenever a term has type U it also has type V. Recall that function arguments are antimonotonic with regard to subtyping, so if $U \leq U'$ and $V \leq V'$ then $(U' \multimap V) \leq (U \multimap V')$.

Should one take $U \leq !U$ or $!U \leq U$? There are good arguments in both directions. By Promotion, if $v : V$ then also $v : !V$ (whenever the free variables of v are all nonlinear), which suggests taking $V \leq !V$ (though the constraint on the free variables of v mitigates against this). By Dereliction combined with $\multimap$-I, if $\lambda x.\, v : U \multimap V$ then also $\lambda x.\, v : !U \multimap V$, which by anti-monotonicity suggests taking $!U \leq U$. Thus in some ways U is a subtype of $!U$, and in some ways the opposite is true.

This appears to bode ill for subtyping, but let's try anyway. Provisionally choose $!U \leq U$. Returning to the identity function example, we now have $!X \multimap X \leq X \multimap X$, so we can consider the latter to be the "most general" type of the identity function.

But consider the term $\lambda f.\, \lambda x.\, \lambda y.\, f\,(x, y)$, which corresponds to currying. Depending on whether Promotion and Dereliction are applied, this term has the following types, among others:

$$\begin{gathered}((X \otimes Y) \multimap Z) \multimap X \multimap Y \multimap Z,\\ ((X \otimes Y) \multimap Z) \multimap !X \multimap !Y \multimap Z,\\ (!(X \otimes Y) \multimap Z) \multimap !X \multimap !Y \multimap Z,\\ (!(!X \otimes !Y) \multimap Z) \multimap !X \multimap !Y \multimap Z.\end{gathered}$$

It does *not* have this type:

$$(!(X \otimes Y) \multimap Z) \multimap X \multimap Y \multimap Z.$$

Regardless of whether we choose $!U \leq U$ or $U \leq !U$, there is no type that is larger than all the legal types and not larger than any of the illegal types. Legal typings of this term depend on a complex relation between the arguments: the argument to f must definitely be a pair, but this pair may be a "!" type only if x and y both have "!" types. Any approach based on subtyping appears doomed.

So a different approach is required to defining "most general" types. What we will do is to generalise our notion of type to include not just type variables, but also *use* variables, which indicate whether a value is used in a linear or nonlinear way. For instance, the type of the identity function will be written $!^m X \multimap X$, where m is a use variable. This has both $X \multimap X$ and $!X \multimap X$ as instances (the first by taking $m = 0$ and the second by taking $m = 1$). To indicate relations among arguments, each type is paired with a *context*, which is a set of inequalities its use variables must satisfy. The next section presents an algorithm that derives the following as the most general type of the currying term:

$$!^i(!^j(!^k X \otimes !^l Y) \multimap Z) \multimap !^m X \multimap !^n Y \multimap Z \quad [m \geq j, n \geq j, m \geq k, n \geq l].$$

Here the context consists of four inequalities. As we shall see, the set of instances of this type satisfying the context is equal to the set of legal types of the currying term in the linear type system, when we restrict our attention to types in a standard form.

Use types. More formally, the system of use types is defined as follows.

A *use* is either a *use variable* or the constant value 0 (denoting linear) or 1 (denoting nonlinear):

$$i, j, k \ ::= \ m \mid 0 \mid 1.$$

Let m range over use variables, and i, j, k range over uses.

Types of the form $(!U)$ are generalised to types of the form $(!^i U)$. Thus, a *use type* has one of the forms:

$$\begin{aligned} &T, U, V \ ::= \\ &\quad X \mid (!^i U) \mid (U \multimap V) \mid (U \otimes V) \mid (U \oplus V). \end{aligned}$$

where T, U, V now range over use types. A use type of the form $(!^0 U)$ corresponds to the type U in the linear system, and a use type of the form $(!^1 U)$ corresponds to the type $(!U)$ in the linear system.

A *context* is a set of inequalities between uses:

$$C, D, E \ ::= \ i_1 \geq j_1, \ldots, i_n \geq j_n.$$

Let C, D, E range over contexts. A substitution mapping each use variable to either 0 or 1 will either *satisfy* a given context, or not. Inequalities of the form $i \geq 0$ and $1 \geq j$ are considered tautologous and may be removed from the context; they are always satisfiable. An inequality of the form $i \geq 1$ can be satisfied only if i takes on the value 1. An inequality of the form $0 \geq j$ can be satisfied only if j takes on the value 0. A context is *unsatisfiable* if its transitive closure contains an inequality of the form $0 \geq 1$.

A *use list* pairs individual variables with uses:

$$I, J \ ::= \ x_1 : i_1, \ldots, x_n : i_n.$$

As with assumption lists, this has length $n \geq 0$ and each of the x_i is distinct. Let I and J range over use lists. If I is a use list $x_1 : i_1, \ldots, x_n : i_n$ and A is an assumption list $x_1 : U_1, \ldots, x_n : U_n$, then $!^I A$ denotes the assumption list $x_1 : !^{i_1} U_1, \ldots, x_n : !^{i_n} U_n$, and $I \geq j$ denotes the context $i_1 \geq j, \ldots, i_n \geq j$.

A typing judgement now takes the form:

$$A \vdash t : T \ [C].$$

This may be read "Under assumptions A the term t has type T in context C".

The new type inference rules are shown in Figure 5. The changes from the old rules are quite small. First, contexts have been added to all rules. The contexts in the hypothesis of a rule are combined to yield the context of the conclusion. Second, the four rules concerned with "!" types have been reformulated in terms of uses and contexts. The Weakening and Contraction rules are the same as before, except $!U$ has been replaced by $!^1 U$. The Dereliction rule removes a use from a type, and the Promotion rule adds a use to a type. In the Promotion rule, the constraint $I \geq j$ ensures the result type $!^j V$ to correspond to a "!" type only when each type in the assumption $!^I A$ corresponds to a "!" type.

Equivalence to linear types. The use type system of Figure 5 is *sound* and *complete* with regard to the linear type system of Figure 3, in the following sense.

Each judgement in the use type system can be thought of a standing for a set of judgements in the linear type system. To derive this set, replace each use variable consistently by either 0 or 1, and then replace each occurrence of $!^0 U$ with U and each occurrence of $!^1 U$ with $!U$. The resulting contexts will either be tautologous, in which case the judgment is included in the set; or unsatisfiable, in which case it is not.

$$\text{Id}\ \frac{}{x : U \vdash x : U\ [\,]} \qquad \text{Exchange}\ \frac{A, x : U, y : V, B \vdash w : W\ [C]}{A, y : V, x : U, B \vdash w : W\ [C]}$$

$$\text{Promotion}\ \frac{(!^I A) \vdash v : V\ [C]}{(!^I A) \vdash v : (!^j V)\ [C,\ I \geq j]} \qquad \text{Dereliction}\ \frac{A,\ x : U \vdash v : V\ [C]}{A,\ x : (!^i U) \vdash v : V\ [C]}$$

$$\text{Weakening}\ \frac{A \vdash v : V\ [C]}{A,\ x : (!^1 U) \vdash v : V\ [C]} \qquad \text{Contraction}\ \frac{A,\ x : (!^1 U),\ y : (!^1 U) \vdash v : V\ [C]}{A,\ z : (!^1 U) \vdash v^{z,z}_{x,y} : V\ [C]}$$

$$\multimap\text{-I}\ \frac{A,\ x : U \vdash v : V\ [C]}{A \vdash (\lambda x.\, v) : (U \multimap V)\ [C]} \qquad \multimap\text{-E}\ \frac{A \vdash t : (U \multimap V)\ [C] \qquad B \vdash u : U\ [D]}{A,\ B \vdash (t\, u) : V\ [C,\ D]}$$

$$\otimes\text{-I}\ \frac{A \vdash u : U\ [C] \qquad B \vdash v : V\ [D]}{A,\ B \vdash (u, v) : (U \otimes V)\ [C,\ D]}$$

$$\otimes\text{-E}\ \frac{A \vdash t : (U \otimes V)\ [C] \qquad B,\ x : U,\ y : V \vdash w : W\ [D]}{A, B \vdash (\text{case } t \text{ of } \{(x, y) \to w\}) : W\ [C,\ D]}$$

$$\oplus\text{-I}\ \frac{A \vdash u : U\ [C]}{A \vdash (\mathit{inl}\, u) : (U \oplus V)\ [C]} \qquad \frac{A \vdash v : V\ [C]}{A \vdash (\mathit{inr}\, v) : (U \oplus V)\ [C]}$$

$$\oplus\text{-E}\ \frac{A \vdash t : (U \oplus V)\ [C] \qquad B,\ x : U \vdash u : W\ [D] \qquad B,\ y : V \vdash v : W\ [E]}{A, B \vdash (\text{case } t \text{ of } \{\mathit{inl}\, x \to u;\ \mathit{inr}\, y \to v\}) : W\ [C,\ D,\ E]}$$

Figure 5: Rules for use types

It is easy to see that if a typing judgement is derivable in the use type system, then each judgement in the corresponding set is derivable in the linear type system; this is soundness. Further, any derivation in the linear type system trivially has a corresponding derivation in the use type system (just replace each occurrence of $!U$ with $!^1 U$); this is completeness.

Principal types. We now formulate precisely what it means for a term to have a most general, or principal, type.

A *substitution* is a mapping of type variables to use types and use variables to uses. Let S range over substitutions. Substitutions apply to types, assumptions, and contexts in the usual way.

Given a term t and two typing judgements for it,

$$A \vdash t : T\ [C] \qquad \text{and} \qquad A' \vdash t : T'\ [C'],$$

the second judgment is said to be an *instance* of the first if there is a substitution S such that $SA = A'$, $ST = T'$, and SC is a subset of the reflexive transitive closure of C'. A typing judgement is *principal* for t if every other typing judgement for t is an instance of it.

A *syntactic completeness theorem* asserts that every typable term has a principal typing judgment. This is usually proved by giving a *type reconstruction algorithm* that given a t either fails (if t is untypable) or returns a principal typing judgement for t.

It appears straightforward to give a type reconstruction algorithm for the use type system. However, we will see in the next section that it is interesting to restrict our attention to types and judgements in a standard form. Rather than do all the work twice, we will defer consideration of type reconstruction until after this new form has been introduced.

6 Standard types

It will prove interesting to restrict our attention to *standard* types. The grammar of standard types is given by

$$\begin{aligned} &T, U, V \ ::= \\ &\quad X \mid (!^iU \multimap V) \mid (!^iU \otimes !^jV) \mid (!^iU \oplus !^jV), \end{aligned}$$

where T, U, V now range over standard types. A standard typing judgement has the form $!^IA \vdash t : T\ [C]$, where T and all types in A are standard.

The restriction to standard types and judgements is motivated by the standard encoding given in Section 4, which takes intuitionistic types into linear types in standard form. Since every linear typing maps trivially into a corresponding intuitionistic typing, and every intuitionistic typing maps into a standard linear typing, it follows that if a term possesses a linear type judgement then it must posses one in standard form. Alas, it doesn't follow that this judgement will contain a minimal number of "!" symbols. Nonetheless, it is useful because it simplifies the form of principal type judgements slightly, by allowing us to write U instead of $!^iU$ in a few places.

The derivation of a standard judgement may contain sub-derivations of judgements that are not standard. In particular, any use of the Dereliction or Promotion rules will necessarily have a hypothesis or conclusion not in standard form (Dereliction because there is an assumption of the form $x : U$ rather that $x : !^iU$; Promotion because the derived type is of the form $v : !^jV$ rather than $v : V$). Figure 6 gives an equivalent set of inference rules where all the judgements are in standard form. It has no Dereliction and Promotion rules; these have been "built into" the other rules. For example, the new Id rule is a combination of the old Id and Dereliction rules; and the new $\multimap$-E rule is a combination of the old Promotion and $\multimap$-E rules.

The rules for inferring standard types given in Figure 6 are *sound* and *complete* relative to the rules for use types given in Figure 5: a standard judgement can be derived by the new, standard type rules if and only if it can be derived by the old, use type rules. Soundness follows from the fact that each new rule can be expressed as a combination of old rules, and completeness follows from a simple induction over derivations in the old system.

Type reconstruction. We now turn our attention to a type reconstruction algorithm L. Given a term t, if any typing judgement for t exists then the call $L(t)$ will return a triple $(!^IA;\ T;\ C)$ such that $!^IA \vdash t : T\ [C]$ is a principle standard judgement for t (i.e., one that has all standard judgements for t as instances), and fails otherwise. Our algorithm will be similar in style to the algorithm given by Mitchell for subtyping [Mit84, Mit91].

The rules in Figure 6 are already quite close to a type reconstruction algorithm. Since Dereliction and Promotion have been eliminated, the only inferences that are not dictated by the form a term are the Exchange, Weakening, and Contraction rules. Exchange is trivial: simply disregard the order of the assumptions, treating the assumption list as a bag. Apply Contraction whenever a variable shows up in two assumption lists that are to be combined (this may happen in the $\multimap$-E, $\otimes$-I, $\otimes$-E, and $\oplus$-E rules). Apply Weakening whenever a variable shows up in one but not the other of two assumption lists that should be identical (this may happen in the $\oplus$-E rule). Also apply Weakening whenenever a bound variable doesn't appear in its scope (this may happen in the $\multimap$-I, $\otimes$-E, and $\oplus$-E

$$\text{Id}\ \frac{}{x : {!^i}U \vdash x : U\ [\,]} \qquad \text{Exchange}\ \frac{{!^I}A,\ x : {!^i}U,\ y : {!^j}V,\ {!^J}B \vdash w : W\ [C]}{{!^I}A,\ y : {!^j}V,\ x : {!^i}U,\ {!^J}B \vdash w : W\ [C]}$$

$$\text{Weakening}\ \frac{{!^I}A \vdash v : V\ [C]}{{!^I}A,\ x : {!^{\mathit{1}}}U \vdash v : V\ [C]} \qquad \text{Contraction}\ \frac{{!^I}A,\ x : {!^{\mathit{1}}}U,\ y : {!^{\mathit{1}}}U \vdash v : V\ [C]}{{!^I}A,\ z : {!^{\mathit{1}}}U \vdash v^{z,z}_{x,y} : V\ [C]}$$

$$\multimap\text{-I}\ \frac{{!^I}A,\ x : {!^i}U \vdash v : V\ [C]}{{!^I}A \vdash (\lambda x.\, v) : ({!^i}U \multimap V)\ [C]}$$

$$\multimap\text{-E}\ \frac{{!^I}A \vdash t : ({!^i}U \multimap V)\ [C] \qquad {!^J}B \vdash u : U\ [D]}{{!^I}A,\ {!^J}B \vdash (t\, u) : V\ [C,\ D,\ J \geq i]}$$

$$\otimes\text{-I}\ \frac{{!^I}A \vdash u : U\ [C] \qquad {!^J}B \vdash v : V\ [D]}{{!^I}A,\ {!^J}B \vdash (u, v) : ({!^i}U \otimes {!^j}V)\ [C,\ D,\ I \geq i,\ J \geq j]}$$

$$\otimes\text{-E}\ \frac{{!^I}A \vdash t : ({!^i}U \otimes {!^j}V)\ [C] \qquad {!^J}B,\ x : {!^i}U,\ y : {!^j}V \vdash w : W\ [D]}{{!^I}A,\ {!^J}B \vdash (\text{case } t \text{ of } \{(x, y) \to w\}) : W\ [C,\ D]}$$

$$\oplus\text{-I}\ \frac{{!^I}A \vdash u : U\ [C]}{{!^I}A \vdash (\mathit{inl}\, u) : ({!^i}U \oplus {!^j}V)\ [C,\ I \geq i]} \qquad \frac{{!^J}B \vdash v : V\ [D]}{{!^J}B \vdash (\mathit{inr}\, v) : ({!^i}U \oplus {!^j}V)\ [D,\ J \geq j]}$$

$$\oplus\text{-E}\ \frac{{!^I}A \vdash t : ({!^i}U \oplus {!^j}V)\ [C] \qquad {!^J}B,\ x : {!^i}U \vdash u : W\ [D] \qquad {!^J}B,\ y : {!^j}U \vdash v : W\ [E]}{{!^I}A,\ {!^J}B \vdash (\text{case } t \text{ of } \{\mathit{inl}\, x \to u;\ \mathit{inr}\, y \to v\}) : W\ [C,\ D,\ E]}$$

Figure 6: Rules for standard types

rules). Finally, apply unification whenever a type or use appears twice in one of the rules.

Given this analysis, the inference rules in Figure 6 can be read off directly to yield the algorithm shown in Figure 7.

The auxiliary function *unify* takes a set of equations between types and uses and returns the most general unifier of all the equations, if a unifier exists. If no unifier exists, the call to *unify* (and hence the call to L) fails. Equations are presented in three forms. First, in the simple form $U = V$, where U and V are types. Second, in the form ${!^I}A = {!^J}B$, which corresponds to Contraction. This stands for the set of equations containing $U = V$, $i = \mathit{1}$, $j = \mathit{1}$ for each x such that $x : {!^i}U$ is in ${!^I}A$ and $x : {!^j}V$ is in ${!^J}B$. Third, in the form $I \setminus J$, which corresponds to Weakening. This stands for the set of equations containing $i = \mathit{1}$ for each x such that $x : i$ is in I and x is not in the domain of J.

$$\begin{array}{lcll}
L(x) & = & \text{let} & m, X \text{ be fresh} \\
 & & \text{in} & (x : !^m X;\ X;\ )
\end{array}$$

$$\begin{array}{lcll}
L(\lambda x.\, v) & = & \text{let} & (!^I A,\ x : !^i U;\ V;\ C) = L(v) \\
 & & \text{in} & (!^I A;\ (!^i U \multimap V);\ C)
\end{array}$$

$$\begin{array}{lcll}
L(t\, u) & = & \text{let} & m, Y \text{ be fresh} \\
 & & & (!^I A;\ T;\ C) = L(t) \\
 & & & (!^J B;\ U;\ D) = L(u) \\
 & & & S = \text{unify}\,(T = (!^m U \multimap Y), !^I A = !^J B) \\
 & & \text{in} & (S!^I A,\ S!^J B;\ SY;\ SC,\ SD,\ SJ \geq Sm)
\end{array}$$

$$\begin{array}{lcll}
L((u, v)) & = & \text{let} & m, n \text{ be fresh} \\
 & & & (!^I A;\ U;\ C) = L(u) \\
 & & & (!^J B;\ V;\ D) = L(v) \\
 & & & S = \text{unify}\,(!^I A = !^J B) \\
 & & \text{in} & (S!^I A,\ S!^J B;\ (!^m SU \otimes !^n SV);\ SC,\ SD,\ SI \geq m,\ SJ \geq n)
\end{array}$$

$$\begin{array}{lcll}
\multicolumn{4}{l}{L(\text{case } t \text{ of } \{(x, y) \rightarrow w\})} \\
 & = & \text{let} & (!^I A;\ T;\ C) = L(t) \\
 & & & (!^J B,\ x : !^i U,\ y : !^j V;\ W;\ D) = L(w) \\
 & & & S = \text{unify}\,(T = (!^i U \otimes !^j V), !^I A = !^J B) \\
 & & \text{in} & (S!^I A, S!^J B;\ SW;\ SC,\ SD)
\end{array}$$

$$\begin{array}{lcll}
L(\mathit{inl}\, u) & = & \text{let} & m, n, Y \text{ be fresh} \\
 & & & (!^I A;\ U;\ C) = L(u) \\
 & & \text{in} & (!^I A;\ !^m U \oplus !^b Y;\ C,\ I \geq m)
\end{array}$$

$$\begin{array}{lcll}
L(\mathit{inr}\, v) & = & \text{let} & m, n, X \text{ be fresh} \\
 & & & (!^J B;\ V;\ D) = L(v) \\
 & & \text{in} & (!^J B;\ !^m X \oplus !^n V;\ D,\ J \geq n)
\end{array}$$

$$\begin{array}{lcll}
\multicolumn{4}{l}{L(\text{case } t \text{ of } \{\mathit{inl}\, x \rightarrow u;\ \mathit{inr}\, y \rightarrow v\})} \\
 & = & \text{let} & (!^I A;\ T;\ C) = L(t) \\
 & & & (!^J B,\ x : !^i U;\ W;\ D) = L(u) \\
 & & & (!^{J'} B',\ y : !^j V;\ W';\ D') = L(v) \\
 & & & S = \text{unify}(T = (!^i U \oplus !^j V),\ W = W', \\
 & & & \qquad\qquad !^I A = !^J B,\ !^I A = !^{J'} B',\ J \setminus J',\ J' \setminus J) \\
 & & \text{in} & (S!^I A, S!^J B, S!^{J'} B';\ SW;\ SC,\ SD,\ SD')
\end{array}$$

Figure 7: Reconstruction algorithm for standard types

The algorithm is expressed using a form of pattern matching against assumption lists. The pattern $!^I A,\ x : !^i U$ binds against an assumption list as follows. If the assumption list contains x, then $!^i U$ is bound to the type associated with x, and $!^I A$ is bound to the list with the entry for x removed (this corresponds to Exchange). If x is not in the list, then $!^I A$ is bound to the entire list, and i is bound to 1, and U is bound to a fresh type variable (this corresponds to Weakening).

The correctness of the algorithm should be apparent from the above description. A formal proof would resemble that for the Hindley-Milner algorithm [Hin69, Mil78, DM82] or Mitchell's algorithm for subtypes [Mit84, Mit91], and appears straightforward.

Relation to Guzmán and Hudak. The type inference rules given here are strongly reminiscent of those in [GH90]. Both papers have similar notions of "type", "use", "assumptions", and "context". The "use lists" of this paper correspond directly to the "liabilities" of that paper. Types written in the form $!^i U \multimap V$ here correspond to types written in the form $U \xrightarrow{i} V$ there. One interesting difference is that here contexts consist only of inequalities between uses, while there contexts also contain inequalities between types. Hence the reconstruction algorithm there needs auxiliary routines for matching and simplifying types in contexts, which are unnecessary here.

The work reported in [GH90] goes considerably beyond the work reported here in that it has a much richer family of uses, in the technical sense: seven there as compared to two here. As a result, it is also of greater use, in the practical sense: a far wider range of programs will pass the type checker. In particular, unlike the work here, it attempts to cope with the important notion of "read only" uses. Nonetheless, the work described here seems to form a valuable bridge between the theory of linear logic on the one side, and the practical concerns of Guzmán and Hudak on the other.

7 Steadfast types

Now return to the end of Section 3 and consider a different path of development. Recall that Dereliction allows circumstances to arise in which a value of linear type may have more than one pointer to it. This section considers a variant of the linear system that restricts the use of Dereliction and Promotion so that every value of linear type is guaranteed to have exactly one pointer to it. The resulting type system is called *steadfast*. Steadfast types are never worthy of Promotion, nor guilty of Dereliction.

It won't do to simply get rid of Promotion and Dereliction, since this would provide no way to introduce or eliminate terms of nonlinear type. Instead, Promotion and Dereliction are replaced by new introduction and elimination rules for the other type formers. The new system consists of all the rules in Figure 3 except for Promotion and Dereliction, together with the new rules shown in Figure 8.

For each type former there are now two sets of rules, the linear rules (in Figure 3) and the nonlinear rules (in Figure 8). For instance, the linear rules $\multimap$-I and $\multimap$-E introduce and eliminate types of the form $(U \multimap V)$, while the nonlinear rules $!\multimap$-I and $!\multimap$-E introduce and eliminate types of the form $!(U \multimap V)$. Since Promotion and Dereliction provide the only way to add or remove "!" from a type, and these rules are now gone, $(U \multimap V)$ and $!(U \multimap V)$ may be regarded as completely distinct types.

A nonlinear data structure must not contain any pointers to components of linear type. Thus, linear products have the form $(U \otimes V)$, where U and V may be either linear or nonlinear; but nonlinear products must have the form $!(!U \otimes !V)$, where the components are constrained to be nonlinear. Similarly for sums. It's easy to see why this constraint is neccessary. Consider the term

$$\begin{array}{l} \text{case } z \text{ of } \{ \\ \quad \mathit{inl}\, x \to \text{case } z \text{ of } \{ \\ \qquad\qquad \mathit{inl}\, x' \to \ldots x \ldots x' \ldots \}\}. \end{array}$$

If z has type $!(!U \oplus !V)$ then this is legal. But if the nonlinear type $!U$ is replaced by the linear type U, then there would be two variables, x and x', containing a pointer to the same linear value—a disaster! Hence the requirement that components of products and sums be nonlinear.

For functions, the situation is a little different. Functions of the form $!(U \multimap V)$ are allowed for U

$$!\multimap\text{-I}\ \frac{!A,\ x : U \vdash v : V}{!A \vdash (\lambda x.\, v) : !(U \multimap V)} \qquad !\multimap\text{-E}\ \frac{A \vdash t : !(U \multimap V) \quad B \vdash u : U}{A,\ B \vdash (t\, u) : V}$$

$$!\otimes\text{-I}\ \frac{A \vdash u : !U \quad B \vdash v : !V}{A,\ B \vdash (u, v) : !(!U \otimes !V)} \qquad !\otimes\text{-E}\ \frac{A \vdash t : !(!U \otimes !V) \quad B,\ x : !U,\ y : !V \vdash w : W}{A, B \vdash (\text{case } t \text{ of } \{(x, y) \to w\}) : W}$$

$$!\oplus\text{-I}\ \frac{A \vdash u : !U}{A \vdash (\mathit{inl}\, u) : !(!U \oplus !V)} \qquad \frac{A \vdash v : !V}{A \vdash (\mathit{inr}\ v) : !(!U \oplus !V)}$$

$$!\oplus\text{-E}\ \frac{A \vdash t : !(!U \oplus !V) \quad B,\ x : !U \vdash u : W \quad B,\ y : !V \vdash v : W}{A, B \vdash (\text{case } t \text{ of } \{\mathit{inl}\, x \to u;\ \mathit{inr}\, y \to v\}) : W}$$

Plus all rules in Figure 3 except Promotion and Dereliction.

Figure 8: Rules for steadfast types

and V either linear or nonlinear. For instance,

$$\vdash (\lambda x.\, x) : !(X \multimap X)$$

is a valid typing by the $!\multimap$-I rule, indicating that it is perfectly alright to use the identity function many times, even on nonlinear values. What is required is that the closure representing a nonlinear function must not contain any pointers to values of linear type. This is enforced by the use of $!A$ rather than A in the $!\multimap$-I rule. For instance,

$$x : X \vdash (\lambda y.\, x) : !(!Y \multimap X)$$

is *not* a valid typing. If it were, applying this function multiple times would allow multiple access to the supposedly linear variable x. But this typing would be valid if the nonlinear type $!X$ replaced the linear type X.

Examination of the rules should convince the reader that this new system indeed has the desired property: values of nonlinear type will always have exactly one pointer to them. The pairing of the introduction and elimination rules is now particularly exact: each value is allocated, and the sole pointer to it created, by an introduction rule; and that sole pointer will eventually be passed into a corresponding elimination rule, which deallocates the value.

Relation of steadfast and linear types. The new, steadfast system of Figure 8 relates to the old, linear system of Figure 3 as follows. The following are all well-typings in the old system:

$$\begin{aligned}&(\lambda x.\, x) : (!X) \multimap X,\\ &(\lambda z.\, \text{case } z \text{ of } \{(x, y) \to (x, y)\})\\ &\qquad : (!X \otimes !Y) \multimap !(!X \otimes !Y),\\ &(\lambda z.\, \text{case } z \text{ of } \{\mathit{inl}\, x \to \mathit{inl}\, x;\ \mathit{inr}\, y \to \mathit{inr}\, y\})\\ &\qquad : (!X \oplus !Y) \multimap !(!X \oplus !Y).\end{aligned}$$

The first follows from Dereliction and $\multimap$-I; the second from $\otimes$-E, Promotion, $\otimes$-I, and $\multimap$-I; the third from $\oplus$-E, Promotion, $\oplus$-I, and $\multimap$-I. Semantically, each of these terms behaves as the identity: they are in effect coercion functions that remove or add "!" symbols in a permitted way.

The new $!\multimap$-E, $!\otimes$-E and $!\oplus$-E rules follow from the corresponding old elimination rules combined with application of the first coercion function. The new $!\multimap$-I rule follows from the old $\multimap$-I rule and Promotion. The new $!\otimes$-I and $!\oplus$-I rules follow from the corresponding old introduction rules combined with applications of the second and third coercion functions.

For each term that is well-typed in the new system, there is a corresponding term (possibly with some applications of the above coercion functions) that is well-typed in the old system. It follows that in the new system, the type of every well-typed term corresponds to a theorem of Girard's linear logic. However, the converse does not hold. There are theorems of Girard's logic that have no terms of corresponding type in the new system. The most obvious example is the theorem $(!X) \multimap X$, corresponding to Dereliction, which cannot be derived in the new system. Similarly, the theorems $(!X \otimes !Y) \multimap !(!X \otimes !Y)$ and $(!X \oplus !Y) \multimap !(!X \oplus !Y)$, arising from Promotion, also cannot be derived. In this sense, the new system is *sound* but *not complete* with respect to the old system.

Arrays revisited. The old linear system required a single operation

$$\mathit{block} : !(!\,\mathit{Val} \multimap (\mathit{Arr} \multimap (X \otimes \mathit{Arr})) \multimap X)$$

to allocate, initialise, process, and deallocate a linear array. In the new steadfast system, this unwieldy operation may be neatly decomposed into smaller parts, introducing separate operations to allocate and deallocate the array:

$$\begin{aligned}\mathit{alloc} &: !(!\,\mathit{Val} \multimap \mathit{Arr}),\\ \mathit{dealloc} &: !(\mathit{Arr} \multimap \mathit{Unit}).\end{aligned}$$

Here *Unit* is a type containing only a single value, which is introduced by the term () and eliminated by the term (case t of $\{() \to w\}$). Now *block* can be defined as follows:

$$\begin{aligned}&\mathit{block} =\\ &\quad \lambda v.\,\lambda f.\,\text{case } f\,(\mathit{alloc}\,v) \text{ of } \{\\ &\qquad\qquad (x, a) \to \text{case } \mathit{dealloc}\,a \text{ of } \{\\ &\qquad\qquad\qquad\qquad () \to x\}\}.\end{aligned}$$

Since a has a linear type, the type system prevents it from simply being discarded; instead, it must be explicitly deallocated.

Recall that this approach would not work in the old system: combining *alloc* with Promotion and Dereliction spelled disaster.

A syntactic trick. In the steadfast system, there is no way to convert a value of type U to a value of type $!\,U$, or vice versa, so we may as well introduce distinct names for the two families of types. For instance, we can decide to write $!(U \multimap V)$ as $(U \to V)$, to write $!(U \otimes V)$ as $(U \times V)$, and to write $!(U \oplus V)$ as $(U + V)$. We cannot interconvert values of types X and $!X$ either, so we might as well regard these as two distinct families of type variables. Since in practice we might expect nonlinear values to be more common then linear ones, let's reverse the convention and write $!X$ as X, and write X as $¡X$.

These conventions make the intuitionistic type system a subset of the system given here. Every term typable in the intuitionistic system has the same type in the new system, although it may have other types. For instance, the term $(\lambda x.\, x)$ has all the following as valid typings: $(¡X \multimap ¡X)$, $(X \multimap X)$, $(¡X \to ¡X)$, and $(X \to X)$.

The type system yielded by this syntactic trick corresponds to the one described in [Wad90]. Although that paper claimed to be about a type system closely related to linear logic, readers familiar with linear logic found the relationship to not be so clear. The connection with linear logic — and the reason for the confusion — now stands revealed. The type system of [Wad90] is indeed based on linear logic, but it is a variant of linear logic which restricts the use of Dereliction and Promotion.

Steadfast standard types. The syntactic trick just described is something of a dead end. A more productive path is to follow the same development as before, introducing steadfast variants of use types and standard types. In this case, the appropriate version of standard types allows for possible nonlinearity everywhere, though it rules out types such as $!^i(!^j U)$. The grammar of steadfast standard types is given by

$$\begin{aligned}&T, U, V \;::=\\ &\quad X \mid (!^i U \multimap !^j V) \mid (!^i U \otimes !^j V) \mid (!^i U \oplus !^j V),\end{aligned}$$

where T, U, V now range over steadfast standard types. Similarly, a steadfast standard judgement

$$\text{Id}\ \frac{}{x : !^i U \vdash x : !^i U\ []} \qquad \text{Exchange}\ \frac{!^I A,\ x : !^i U,\ y : !^j V,\ !^J B \vdash w : !^l W\ [C]}{!^I A,\ y : !^j V,\ x : !^i U,\ !^J B \vdash w : !^l W\ [C]}$$

$$\text{Weakening}\ \frac{!^I A \vdash v : !^j V\ [C]}{!^I A,\ x : !^1 U \vdash v : !^j V\ [C]} \qquad \text{Contraction}\ \frac{!^I A,\ x : !^1 U,\ y : !^1 U \vdash v : !^j V\ [C]}{!^I A,\ z : !^1 U \vdash v^{z,z}_{x,y} : !^j V\ [C]}$$

$$\multimap\text{-I}\ \frac{!^I A,\ x : !^i U \vdash v : !^j V\ [C]}{!^I A \vdash (\lambda x.\, v) : !^k(!^i U \multimap !^j V)\ [C,\ I \geq k]}$$

$$\multimap\text{-E}\ \frac{!^I A \vdash t : !^k(!^i U \multimap !^j V)\ [C] \qquad !^J B \vdash u : !^i U\ [D]}{!^I A,\ !^J B \vdash (t\, u) : !^j V\ [C,\ D]}$$

$$\otimes\text{-I}\ \frac{!^I A \vdash u : !^i U\ [C] \qquad !^J B \vdash v : !^j V\ [D]}{!^I A,\ !^J B \vdash (u, v) : !^k(!^i U \otimes !^j V)\ [C,\ D,\ i \geq k,\ j \geq k]}$$

$$\otimes\text{-E}\ \frac{!^I A \vdash t : !^k(!^i U \otimes !^j V)\ [C] \qquad !^J B,\ x : !^i U,\ y : !^j V \vdash w : !^l W\ [D]}{!^I A,\ !^J B \vdash (\text{case}\ t\ \text{of}\ \{(x, y) \to w\}) : !^l W\ [C,\ D]}$$

$$\oplus\text{-I}\ \frac{!^I A \vdash u : !^i U\ [C]}{!^I A \vdash (\mathit{inl}\, u) : !^k(!^i U \oplus !^j V)\ [C,\ k \geq i]} \qquad \frac{!^J B \vdash v :\ V\ [D]}{!^J B \vdash (\mathit{inr}\, v) : !^k(!^i U \oplus !^j V)\ [D,\ k \geq j]}$$

$$\oplus\text{-E}\ \frac{!^I A \vdash t : !^k(!^i U \oplus !^j V)\ [C] \quad !^J B,\ x : !^i U \vdash u : !^l W\ [D] \quad !^J B,\ y : !^j U \vdash v : !^l W\ [E]}{!^I A,\ !^J B \vdash (\text{case}\ t\ \text{of}\ \{\mathit{inl}\, x \to u;\ \mathit{inr}\, y \to v\}) : !^l W\ [C,\ D,\ E]}$$

Figure 9: Rules for steadfast standard types

has the form $!^I A \vdash t : !^k T\ [C]$ where T and all types in A are steadfast standard.

The corresponding inference rules are shown in Figure 9. As before, we can show that these rules are sound and complete with regard to the inference rules for steadfast types. And, as before, we can derive from these rules a type reconstruction algorithm that yields principle steadfast standard types. Indeed, the reconstruction algorithm can be read off from the inference rules in just the same style as used previously. This is quite straightforward and left as an exercise for the reader.

In [Wad90] the question was raised as to whether the type system described there possessed a type reconstruction algorithm analogous to Hindley-Milner. Having at last answered this question in the affirmative, here seems an appropriate place to conclude this study of the uses of linear logic.

Acknowledgments. Thanks to Samson Abramsky, Jean-Yves Girard, Juan Guzmán, Sören Holmström, Paul Hudak, and Yves Lafont, for having good ideas and for explaining them to me; and to Yves Lafont, Simon Peyton Jones, John Hughes, and the referees for their perceptive comments. And special thanks to Peter Sestoft for lending a discerning ear.

References

[Abr90] S. Abramsky, Computational interpretations of linear logic. Preprint, Imperial College, London.

[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.

[Cur58] H. B. Curry and R. Feys, *Combinatory Logic.* North Holland, 1958.

[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 9'th ACM Symposium on Principles of Programming Languages*, Albuquerque, N.M., January 1982.

[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.

[GH90] J. Guzmán and P. Hudak, Single-threaded polymorphic lambda calculus. In *Proceedings 5'th IEEE Symposium on Logic in Computer Science*, Philadelphia, Pa., June 1990.

[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.

[How80] W. A. Howard, The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, *To H. B. Curry: Essays on combinatory logic, lambda calculus, and formalism.* Academic Press, 1980.

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

[KM89] T.-M. Kuo and P. Mishra, Strictness analysis: a new perspective based on type inference. In *Proceedings 4'th International Conference on Functional Programming Languages and Computer Architecture*, ACM Press, London, September 1989.

[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.

[Mit84] J. C. Mitchell, Coercion and type inference. In *Proceedings 11'th ACM Symposium on Principles of Programming Languages*, January 1984.

[Mit91] J. C. Mitchell, Extending Curry and ML type inference with subtypes. Preprint. To appear in *Journal of Functional Programming*, **1**(3), June 1991.

[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.)

[Wad90] P. Wadler, Linear types can change the world! In M. Broy and C. Jones, editors, *Programming Concepts and Methods*, Sea of Galilee, Israel, April 1990. North Holland, 1990.

[Wak90] D. Wakeling, Linearity and laziness. Ph.D. dissertation, University of York, November 1990.

[Wri89] D. A. Wright, Strictness analysis via type inference. Technical report, University of Tasmania, September 1989.