# Observers for Linear Types

Martin Odersky*
Yale University, Department of Computer Science,
Box 2158 Yale Station, New Haven, CT 06520
odersky@cs.yale.edu

*European Symposium on Programming*
*February 1992*

**Abstract**

Linear types provide the framework for a safe embedding of mutable state in functional languages by enforcing the principle that variables of linear type must be used exactly once. A potential disadvantage of this approach is that it places read accesses to such variables under the same restriction as write accesses, and thus prevents reads to proceed in parallel. We present here an extension of linear types which augments the usual distinction between linear and non-linear by a third state, *observers* of linear variables. Since, unlike linear variables, observers can be duplicated, multiple concurrent reads are made possible. On the other hand, observers must be short-lived enough to never overlap with mutations. The resulting type system is in many aspects similar to the one of ML: It is polymorphic, has principal types, and admits a type reconstruction algorithm.

## 1 Introduction

We are investigating a type system that addresses the update problem in functional languages: How can we implement updates efficiently, but still retain a declarative semantics? Methods to solve this problem — of which there are many — usually come under the name of effect analysis. Effect analysis looks for opportunities to replace costly non-destructive operations on aggregates such as arrays or hash tables by cheaper destructive ones. This can take place at run-time, using reference counting [GSH88] or reverse difference lists [Coh84]. It can also be performed at compile-time, using one of the optimization techniques of [Hud87, NPD87, Blo89, Deu90, DP90], for instance. A third alternative is to let the programmer perform effect analysis, and reduce the task of the computer to *effect checking*; the computer simply verifies that the transition from non-destructive to destructive operations is semantics preserving. In this setting it is

*work was done in part while at IBM T.J. Watson Research Center.

natural to regard effect information to be a kind of type information and effect checking to be an extension of type checking.

The main advantage of this programmer-directed approach is that the choice between copying and in-place updates is made visible. Hence, the programmer can avoid the potentially drastic efficiency loss which could otherwise result from missed optimization opportunities. This is most important in the presence of separate compilation and software component libraries. Users of such libraries have to know how they can access the exported components without risking performance degradation. As the standard way of communicating such legal use-patterns is a type system, it seems to be a good idea to augment types with effect information. However, effect checking type systems face the double challenge of avoiding being either too restrictive or too complex. After all, unlike automatic optimizers, programmers are willing to digest only a limited amount of effect information.

We present here an approach towards an effect checking type system which meets these challenges. Observable linear types are loosely based on Wadler's "steadfast, standard" version of linear types and extend it by adding "read-only" (in our terms: observer) accesses to linear variables. In [Wad91] this extension was acknowledged to be an open research problem.

Linear type systems [Laf88, Abr90, Wad91] are related by the Curry-Howard isomorphism to Girard's linear logic [Gir87]. They are based on the principle that a variable of linear type must be used exactly once. If linear types are steadfast, that is, not convertible with non-linear types, this principle allows updates to linear variables to be performed destructively and also obviates the need for garbage collecting them. In the terminology of [Wad90b], linear variables make up the "world", which can be neither duplicated nor discarded.

The "no-duplication" restriction on linear variables makes them a bit awkward to use in programming. Observation of the world is placed under precisely the same restrictions as changes to it, although it is clearly much less intrusive. To address this shortcoming, Wadler suggested in [Wad90b] a construct which exceeds linear logic by allowing the world to be observed in a local context. This is written

$$\textbf{let!}\ (a)\ x = e'\ \textbf{in}\ e. \tag{1}$$

Here, the linear variable $\mathsf{a}$, used once in the outer expression $\mathsf{e}$, may also be read arbitrarily often in the local expression $\mathsf{e}'$. To make this construct safe, Wadler proposed the following measures: First, a hyperstrict evaluation rule which specifies that $\mathsf{e}'$ be reduced to normal form before evaluation of $\mathsf{e}$ is begun. Second, a static restriction that all components of $\mathsf{a}$ and $\mathsf{x}$ have mutually distinct types. Finally, a static restriction that $\mathsf{x}$ may not be of function type. The static restrictions prevent the normal form of $\mathsf{x}$ from sharing the value of the linear variable $\mathsf{a}$. Together with the hyperstrict evaluation rule this ensures safety, but at quite drastic cost: In particular the "mutually distinct types" requirement is an overly conservative approximation to the actual aliasing in a *let!* construct. The approximation becomes even worse if the type system is polymorphic (the one in [Wad90b] isn't). In that case, the notion of equality between types has to be replaced by unifiability. As a consequence, virtually every *let!* construct is unsafe in which the type of either the linear or the bound variable is polymorphic. Hence, we see that linear types have so far been better at changing the world than at observing it.

In this paper, we look at a more thorough solution to the observer problem. We will be concerned only with the "no-duplication" property of linear types, not with the "no-discarding" property which allows static garbage collection. The principal idea is to extend the distinction between linear and nonlinear variables by a third state, which denotes observers of linear variables. In the *let*! construct (1), all occurrences of the linear variable a in e′ would now have type "observer". Unlike linear types themselves, observers can be duplicated freely (this implies that updates to observers are forbidden). However, observers have to be short-lived, they may not be exported out of the scope of a **let**! binding. This enforces observation and updating of linear variables to occur in a strictly alternating fashion, where no observer lives long enough to observe an update.

Linear, non-linear and observer constitute the three basic aliasing states of a variable. These states are attributes of the types in our system. The type system has the following useful properties:

- It is polymorphic in types and alias states. Type polymorphism means that a type variable ranges over all types, linear, non-linear and observers. Aliasing polymorphism means that the aliasing attribute of a type may be a variable.
- It has the principal type property. That is, given a closed initial type assignment $A$, every well-typed expression has a most general type-scheme $\sigma$.
- It admits a type reconstruction algorithm which assigns an expression its principal type-scheme. Type reconstruction can work without type declarations for bound variables.
- With a few straightforward abbreviations, function signatures can be written in a concise form, of comparative complexity to the use of *in* and *out* specifiers in Ada. This observation might seem somewhat surprising, since our type system is definitely more complex than the standard Hindley/Milner system, say. A partial explanation might be that much of our machinery has to do with observer types which occur only in a local context, and by definition do not show up in the type signatures of defined variables.

## Other Related Work

Schmidt [Sch85] suggested a simple type system which gives conditions for safety of in-place updates. Other early work was done in the FX project [LG88, JG91] and the area has been an active research subject in the last few years. Observable linear types build on several previous approaches. Besides the strong connection to linear types, there is also a connection to Baker's "free" region analysis [Bak90] for type reconstruction. Regions do not enter our system explicitly, but the notion of region in [Bak90] or [TJ91] corresponds exactly to a collection of types with the same alias variable as an attribute.

Another popular approach to the update problem uses abstract data types to encapsulate accesses to mutable data structures. The idea is to have an abstract type of "state transformers", but no type for the transformed data structures itself [Wad90a]. There is a single operation, *block*, which creates a mutable data structure serving as a scratch area, applies a state transformer to it, and returns the (immutable) result of the application while discarding the scratch area. This has the advantage that no extension to traditional type systems is needed, but it requires programming in a continuation passing style. Also, it is currently not clear how the method should be extended to deal with several mutated data structures. The latter problem is addressed in the non-standard type system of [SRI91] which again requires continuation passing style. Continuation passing style is problematic since it fully sequentializes lookups as well as updates. By contrast, observable linear types allow lookups to proceed in parallel and generally impose much less restrictions on programming style. The latter point is important in the situation where a purely functional program is transformed into a program with transparent updates by changing the implementation of some data types. Observable linear types allow such efficiency-improving transformations to be performed incrementally, without requiring a complete rewrite.

Compared to analyses based on liabilities and function effects [GH90, Ode91], linear types augmented with observers are less precise in some cases and more precise in others. Liabilities give information about which variables are possible aliases of each other, whereas alias states only record the fact that a variable might be aliased. Hence, using liabilities we can verify some expressions to be safe which cannot be handled by all other approaches. On the other hand, current liability-based approaches are less accurate for non-flat mutable structures. Moreover, when extended to higher-order functions, they do not admit "nice" principal types (i.e. they need disjunctive constraints, see [Ode91] for an example). We believe that approaches based on linear types will turn out to be more practical than liability-based approaches because they tend to be more concise and generalize naturally to the higher-order case.

The rest of this paper is organized as follows: Section 2 defines the syntax of types in a small example language. Section 3 discusses their use in several program examples. Section 4 presents typing rules. Section 5 discusses a type reconstruction algorithm. Section 6 concludes.

## 2 Observable Linear Types

### Language

We use essentially the language of [Wad90b], with the exception of **let!** constructs, where in our case observers of linear variables need not be quoted. Quoting these variables explicitly is undesirable since it restricts polymorphism, and our type reconstruction algorithm can work without it.

| | | | | |
|---|---|---|---|---|
| Expressions | $e$ | $::=$ | $x$ | identifiers |
| | | $\mid$ | $e\ e'$ | application |
| | | $\mid$ | $\lambda x.e$ | abstraction |
| | | $\mid$ | $\lambda_{\mathbf{1}} x.e$ | linear abstraction |
| | | $\mid$ | $\mathbf{let}\ x = e'\ \mathbf{in}\ e$ | definition |
| | | $\mid$ | $\mathbf{let!}\ x = e'\ \mathbf{in}\ e$ | sequential definition |
| | | $\mid$ | $\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3$ | conditional |

## Monomorphic Types

We start with a type system which is monomorphic in its aliasing aspects (but polymorphic in its structural aspects). A type in this system (called a *monotype* in the following) consists of two parts $\alpha \cdot v$ which describe outside aliasing and internal structure, respectively. The components are separated by an infix dot $(\cdot)$.

| | | | | |
|---|---|---|---|---|
| Monomorphic types | $\tau$ | $=$ | $\kappa$ | basic type |
| | | $\mid$ | $list\ \tau$ | list type |
| | | $\mid$ | $\tau_1 \rightarrow \tau_2$ | function type |
| | | $\mid$ | $\alpha \cdot \tau$ | alias state $\cdot$ type |
| | | $\mid$ | $t$ | type variable |
| Alias states | $\alpha$ | $=$ | $\mathbf{0} \mid \mathbf{1} \mid \mathbf{2}$ | observer, linear, non-linear |

In our example language, we will use only a few different forms of types $\tau$, namely (immutable) basic types, mutable lists, and function types. We will see in Section 3 how other mutable data structures such as arrays or matrices can be constructed from mutable lists. Hence, there is no need for modeling these structures in the type system (although an implementation should certainly treat them as special cases).

The aliasing part $\alpha$ of a monotype is one of the three constants $\mathbf{0}$, $\mathbf{1}$, and $\mathbf{2}$. Variables of a $\mathbf{1}$-type may be accessed only once, and we have the invariant that at most one reference can exist to values of these types. $\mathbf{1}$-types correspond to linear types, and, in a slight misuse of language, we will also call them linear. The correspondence is not exact, since we are concerned only with the "no-duplication" property of $\mathbf{1}$-types, and allow discarding a value of $\mathbf{1}$-type, whereas this is forbidden in pure linear type systems. Variables of $\mathbf{2}$-type (or: non-linear type) may be accessed arbitrarily often and may share references with other non-linear variables. The third category of types are the observer-, or $\mathbf{0}$-types. Observer types allow linear variables to be used more than once. They don't "add to" linear uses (that's why they are given denotation $\mathbf{0}$). When used locally in a **let!** construct, all occurrences of a variable which is linear at the outside are given observer type inside. There may be several such occurrences, but no observer variable may form part of the value which is locally defined in that expression. Put in other words, all components of the type of a variable defined by a **let!** must have $\mathbf{1}$- or $\mathbf{2}$-type. Assuming that the evaluation of **let!** definitions is hyperstrict, we can hence ensure that observation and updating of linear variables occur in a strictly alternating fashion.

Composite list types have an aliasing attribute for the whole type and an attribute for the element type at each level. Not every combination of alias attributes is permissible, we require that a list type is well-formed:

**Definition.** The monotype $\alpha{\cdot}list\ (\beta{\cdot}v)$ is *well-formed* iff

$$\alpha \in \{\mathbf{0}, \mathbf{2}\} \quad \Rightarrow \quad \beta \in \{\mathbf{0}, \mathbf{2}\}$$

The well-formedness condition is needed to ensure that a linear element is not shared (or observed) indirectly by sharing (or observing) its parent.

Monomorphic observable linear types give rise to a type system which extends the steadfast types of [Wad91] with observers. As an example of its use, consider a function which copies an array element to another index position. Assume for the time being that arrays are implemented as lists, with operations (!) for indexing and (*update*) for in-place updates.

$$assign \quad = \quad \lambda i.\ \lambda j.\ \lambda a.\ \mathbf{let!}\ x = a!i\ \mathbf{in}\ update\ j\ x\ a$$

Our type system will assign type $\mathbf{0}{\cdot}list\ (\mathbf{2}{\cdot}v)$ to the first, local occurrence of the array $a$. The type of the locally defined variable $x$ is $(\mathbf{2}{\cdot}v)$ and thus satisfies the restriction that local definitions in a *let*! cannot be of observer type. The last occurrence of $a$ has type $\mathbf{1}{\cdot}list\ \mathbf{2}{\cdot}v$, reflecting the fact that variable $a$ is modified. The type of the whole function is:

$$assign \quad : \quad int \to int \to \mathbf{1}{\cdot}list\ (\mathbf{2}{\cdot}v) \to \mathbf{1}{\cdot}list\ (\mathbf{2}{\cdot}v).$$

This expresses that the array argument is modified (and therefore has to be linear), whereas one of its elements is duplicated (and therefore must be non-linear). The observer state was used only locally; it allowed us to use the linear variable $a$ twice.

The monomorphic type system is still quite inflexible. For instance, it is not possible to formulate a function *head* which works equally on linear and non-linear lists, since the alias state of function arguments is fixed. The obvious way to lift this restriction is to introduce variables which range over alias states, and we will do so in the next sub-section.

## Polymorphic Types

A polymorphic observable linear type (called *polytype* in the following) has a variable in its alias component. The variable usually ranges over the three alias states, but its range can be constrained by predicates. Following [Jon91a], we express this using the syntax of *qualified types*:

$$\begin{array}{llll}
\text{Alias Parts} & \alpha & = & \mathbf{0} \mid \mathbf{1} \mid \mathbf{2} \mid t \mid O\ \alpha \\
\text{Qualified Types} & \rho & = & \pi \Rightarrow \rho \mid \tau \\
\text{Predicates} & \pi & = & \alpha \leq \overline{\mathbf{0}} \mid \alpha \leq \overline{\mathbf{1}} \mid \alpha \leq \overline{\mathbf{2}} \mid \alpha \supseteq \tau \\
\text{Type Schemes} & \sigma & = & \forall t.\sigma \mid \rho
\end{array}$$

Observer tags $O$ are the polymorphic equivalent of the mapping from (monomorphic) linear to observer status in the monomorphic system. If a bound variable $x$ has type $a \cdot v$ outside of a *let*!-construct, it is given type $Oa \cdot v$ inside. This serves as a "reminder" that any value assumed by variable $a$ at the outside has to be translated to observer status inside.

Type variables can be constrained by predicates. There are two forms of such predicates. The first form, $\alpha \leq \overline{n}$, constrains the range of $\alpha$ to a a subset of all three alias-sets. The three two-element alias-sets are characterized as complements of a singleton set. $\overline{\mathbf{0}}$ (non-observer, or original) encompasses $\mathbf{1}$ and $\mathbf{2}$. Variables defined in a **let**! are required to be originals. $\overline{\mathbf{1}}$ (aliased) encompasses $\mathbf{0}$ and $\mathbf{2}$. If a function uses an argument several times outside of a **let**! construct, the argument's type falls in this set. Finally, $\overline{\mathbf{2}}$ encompasses $\mathbf{0}$ and $\mathbf{1}$.

Note that by combining any two of these constraints, we get a monotype¡. For instance, $\forall a.a \leq \overline{\mathbf{0}} \Rightarrow a \leq \overline{\mathbf{1}} \Rightarrow a \cdot \tau$ is equivalent to $\mathbf{2} \cdot \tau$. If a variable is simultaneously bounded by all three constraints, the constraint set is unsatisfiable and the corresponding type is empty.

The second form of constraint makes the well-formedness criterion for list types explicit. The predicate $\alpha \supseteq \tau$ is equivalent to the constraint set

$$\alpha \ \leq \ \overline{\mathbf{1}} \quad \Rightarrow \quad \beta \ \leq \ \overline{\mathbf{1}}$$

where $\beta$ ranges over all the alias parts of $\tau$ and its component types. The typing rules are such that every occurrence of $\alpha \cdot (list\ \tau)$ in a principal type is constrained by a predicate $\alpha \supseteq (list\ \tau)$.

**Example 2.1** The type of function *map* would be expressed as follows:

$$map \quad : \quad \forall s \forall t \forall a \forall b.\ (a \supseteq list\ s) \Rightarrow (b \supseteq list\ t) \Rightarrow \mathbf{2} \cdot (s \rightarrow t) \rightarrow a \cdot list\ s \rightarrow b \cdot list\ t,$$

For conciseness, we will in the following drop $(\alpha \supseteq \tau)$ constraints on a type if they are implied by the structure of the type itself, i.e. if the type contains a subtype of the form $\alpha \cdot \tau$. We will also drop the alias part of a type altogether if it is trivial, i.e. equal to an unconstrained, unshared type variable. Finally, we allow multiple predicates to be grouped together, i.e. $(\pi_1 \Rightarrow \pi_2 \Rightarrow \rho) = (\pi_1, \pi_2 \Rightarrow \rho)$.

**Example 2.2** Using these shorthands, the type of *map* would be written:

$$map \quad : \quad \forall s \forall t.\ \mathbf{2} \cdot (s \rightarrow t) \rightarrow list\ s \rightarrow list\ t.$$

## Predefined Identifiers

As predefined we assume the fixpoint operator $fix$, and a set of operators on lists. Besides the conventional operators $nil, cons, hd$ and $tl$, we also have a destructive update operation on lists. *rplac* takes as arguments two functions $f$ and $g$ which map list heads to list heads and list tails to list tails. Its third argument is a list $xs$ of linear type. The value of

$$rplac\ f\ g\ xs \quad \text{is} \quad cons\ (f\ (hd\ xs))\ (g\ (tl\ xs)),$$

and as a side-effect the first *cons*-node of $xs$ is replaced by this value. The types of the predefined identifiers are:

$$
\begin{array}{lcl}
nil & : & \forall a \forall t.\ a\cdot list\ t \\
cons & : & \forall a \forall t.\ t \rightarrow a\cdot list\ t \rightarrow a\cdot list\ t \\
hd & : & \forall a \forall t.\ a\cdot list\ t \rightarrow t \\
tl & : & \forall a \forall t.\ a\cdot list\ t \rightarrow a\cdot list\ t \\
rplac & : & \forall a \forall b \forall t.\ \mathbf{2}\cdot(t \rightarrow t) \rightarrow \mathbf{2}\cdot(a\cdot list\ t \rightarrow b\cdot list\ t) \rightarrow \mathbf{1}\cdot list\ t \rightarrow b\cdot list\ t \\
fix & : & \forall v.\ \mathbf{2}\cdot(\mathbf{2}\cdot v \rightarrow \mathbf{2}\cdot v) \rightarrow \mathbf{2}\cdot v\ .
\end{array}
$$

The type of *rplac* merits further consideration. One might think that since the tail-replacing function in the second argument is passed a linear list, its type should really be $(\mathbf{1}\cdot list\ t \rightarrow b\cdot list\ t)$. This would lead to some needless loss of polymorphism, however. After all, just because an argument is linear (i.e. unshared), a function applied to it should not be required to exploit the linearity by overwriting the argument. The correct interpretation is that arguments which are known to be linear can safely be used in any way whatsoever. The most general type of the tail-replacing function is therefore $(a\cdot list\ t \rightarrow b\cdot list\ t)$.

The type of the fixpoint operator also needs some explanation. $fix$ is defined only on transformations between non-linear values and its result is again a non-linear value. To see why taking the fixpoint of a transformation between linear values is problematic, consider the expression

$$
\begin{array}{lcl}
mkcirc & : & \mathbf{1}\cdot list\ Int \rightarrow \mathbf{1}\cdot list\ Int \\
mkcirc & = & \lambda xs.\ cons\ 1\ (rplac\ (-1)\ id\ xs)\ ,
\end{array}
$$

where $(-1)$ is the predecessor function on integers. If $fix$ were defined for transformations between linear values, $fix\ mkcirc$ would be legal, of type $\mathbf{1}\cdot\ list\ Int$. But what is the value of this expression? If we disregard side-effects and look at the definition of *rplac*'s result above, it should be the list $[1, 0, 0, ...]$. If we take side-effects into account, however, and assume that the list is evaluated in a head-strict order, we get the list $[0, 0, 0, ...]$. This violates the requirement that all side-effects of well-typed expressions should be transparent.

# 3 Examples

This section tries to give a "feel" of our type system by means of small example programs. We hope to convey the impression that the type signatures of most functions occurring in practice are quite reasonable in size and complexity and also closely correspond to the programmer's intuition. First, here is a side-effecting version of the *append* function:

$$
\begin{array}{lcl}
append & = & fix\ \lambda append.\ \lambda xs.\ \lambda_{\mathbf{1}} ys. \\
 & & \mathbf{if}\ xs = nil\ \mathbf{then}\ ys \\
 & & \mathbf{else}\ rplac\ id\ (\lambda tl.\ append\ tl\ ys)\ xs
\end{array}
$$

The typing rules presented in the next section give *append* the type:

$$append \quad : \quad \forall a \forall t.\ \mathbf{1}\cdot list\ t \rightarrow \mathbf{1}\cdot(a\cdot list\ t \rightarrow a\cdot list\ t)$$

Since the first list argument to *append* gets updated, it must be linear, of type $\mathbf{1}\cdot list\ t$. The type of a curried application like *append xs* must also be linear, because *append xs* contains a reference to a linear variable. Otherwise, we could duplicate accesses to $xs$ in an expression such as

$$(\lambda f.\ (f\ ys, f\ zs))\ (append\ xs).$$

The language has a special form of $\lambda$-abstraction, denoted $\lambda_{\mathbf{1}}$, to define linear functions which have "global" side-effects (i.e. which modify variables other than their arguments). Having two forms of $\lambda$-abstraction does cause some loss of polymorphism in that we have to declare statically whether a function is going to have a global side-effect or not. This can be difficult to predict for higher-order functions. It appears that our type system could be extended to deal with just one kind of $\lambda$ abstraction for linear and non-linear functions using a technique similar to the one in [Wad91]. This would add constraints to type signatures, however, something we wanted to avoid because of the syntactic overhead associated with it. A good alternative, which also avoids the use of $\lambda_{\mathbf{1}}$, is to have the modified argument come last:

$$\begin{array}{lcl} append' & : & \forall a \forall t.\ a\cdot list\ t \rightarrow \mathbf{1}\cdot list\ t\ \rightarrow a\cdot list\ t \\ append'\ xs\ ys & = & append\ ys\ xs \end{array}$$

To simplify presentation, we will from now on allow functions to be written in the equational style. The translation to $\lambda$-abstractions and fixpoint operators should be obvious.

The *append* function uses the rather "heavyweight" operation *rplac*. We can simplify this by using specialized versions of *rplac* which replace only heads or only tails:

$$\begin{array}{lcl} rplhd & : & \forall t.\ \mathbf{2}\cdot(t \rightarrow t) \rightarrow \mathbf{1}\cdot list\ t \rightarrow list\ t \\ rpltl & : & \forall a \forall t.\ \mathbf{2}\cdot(list\ t \rightarrow a\cdot list\ t) \rightarrow \mathbf{1}\cdot list\ t \rightarrow a\cdot list\ t \\ rplhd\ f & = & rplac\ f\ id \\ rpltl\ f & = & rplac\ id\ f \end{array}$$

Remember that $list\ t$, the result type of *rplhd*, is an abbreviation for $a\cdot list\ t$, where $a$ is a fresh type variable. That is, the alias part of *rplhd*'s result type is unconstrained.

Here are linear equivalents of the higher order functions *map* and *foldl*:

$$\begin{array}{lcl} maplin & : & \forall t.\ \mathbf{2}\cdot(t \rightarrow t) \rightarrow \mathbf{1}\cdot list\ t \rightarrow list\ t \\ maplin\ f & = & rplac\ f\ (maplin\ f) \\ foldlin & : & \forall s. \forall t.\ \mathbf{2}\cdot(t \rightarrow s \rightarrow s) \rightarrow list\ t \rightarrow s \rightarrow s. \\ foldlin\ f\ xs\ acc & = & \mathbf{if}\ xs = nil\ \mathbf{then}\ acc \\ & & \mathbf{else}\ foldlin\ f\ (tl\ xs)\ (f\ (hd\ xs)\ acc) \end{array}$$

*maplin* maps a function on a linear list, replacing every node of that list by its corresponding node in the result list. *foldlin* does not restrict any argument to be linear, in fact it is just Haskell's $foldl$ with the second and third argument swapped. $foldlin\ f$ is side-effecting if $f$ is, and is pure otherwise.

Here are some other functions on lists:

$$
\begin{array}{lcl}
upd & : & \forall t.\ int \rightarrow \mathbf{2}{\cdot}(t \rightarrow t) \rightarrow \mathbf{1}{\cdot}list\ t \rightarrow list\ t \\
upd\ i\ f & = & \mathbf{if}\ i = 0\ \mathbf{then}\ rplhd\ f\ \mathbf{else}\ upd\,(i-1)\ f \circ tl \\
 & & \\
swap & : & \forall t.\ int \rightarrow int \rightarrow \mathbf{1}{\cdot}list\ t \rightarrow list\ t \\
swap\ i\ j\ \mathrm{xs} & = & \mathbf{let!}\ x\ =\ xs!i\ \ \mathbf{in} \\
 & & \mathbf{let!}\ y\ =\ xs!j \\
 & & \mathbf{in}\ \ (upd\ i\ (K\ y)\ \circ upd\ j\ (K\ x))\ xs
\end{array}
$$

Function $upd$ updates a selected element of a list, and $swap$ exchanges two list elements. Using lookups (!) and updates ($upd$), we can express mutable vectors in terms of lists. Higher-dimensional mutable arrays can be defined, too. For instance, the update operation for a matrix, represented as a list of lists, is:

$$
\mathbf{type}\ a{\cdot}mat\ t\ =\ a{\cdot}list\ (a{\cdot}list\ t)
$$

$$
\begin{array}{lcl}
upd2 & : & int \rightarrow int \rightarrow \mathbf{2}{\cdot}(t \rightarrow t) \rightarrow \mathbf{1}{\cdot}mat\ t \rightarrow mat\ t \\
upd2\ i\ j\ f & = & upd\ i\ (upd\ j\ f)
\end{array}
$$

For a larger example, we now turn to topological sorting. We want to find a total order for the nodes of a graph in which every node precedes its successors. To make our task of designing an efficient algorithm easier, we assume that the graph is in a convenient representation, given by:

- the list $sources : list\ node$ of all sources in the graph,
- a list $succs : list\ (list\ node)$ which contains for every node in the graph the list of all its successors.
- a linear list $npreds : \mathbf{1}\cdot list\ int$ which contains for every node in the graph the number of its predecessors. This list serves as a "scratch area".

We also assume that $node = int$ such that we can index lists with nodes. Given this graph representation, we can formulate the topological sorting function as follows:

$$
tsort\ :\ \mathbf{2}{\cdot}list\ node \rightarrow list\ (list\ node) \rightarrow \mathbf{1}{\cdot}list\ int \rightarrow list\ node
$$

$$
\begin{array}{l}
tsort\ sources\ succs\ npreds\ \ = \\
\quad \mathbf{if}\ sources = nil\ \mathbf{then} \\
\qquad nil \\
\quad \mathbf{else} \\
\qquad \begin{array}{lll}
\mathbf{let} & src & =\ hd\ sources\ \ \mathbf{in} \\
\mathbf{let} & decnth & =\ \lambda n.\ upd\ n\ (-1)\ \ \mathbf{in} \\
\mathbf{let!} & npreds' & =\ foldlin\ decnth\ (succs!src)\ npreds\ \ \mathbf{in} \\
\mathbf{let!} & sources' & =\ filter\ (\lambda x.\ npreds'!x = 0)\ (succs!src) + tl\ sources \\
\mathbf{in} & \multicolumn{2}{l}{cons\ src\ (tsort\ sources'\ succs\ npreds')}
\end{array}
\end{array}
$$

$$
\begin{array}{ll}
var & A, P \vdash x : \sigma \quad (x : \sigma \in A) \\[1em]
\forall I & \dfrac{A, P \vdash e : \sigma}{A, P \vdash e : \forall t.\sigma} \quad (t \notin tv\ A \cup tv\ P) \\[1em]
\forall E & \dfrac{A, P \vdash e : \forall t.\sigma}{A, P \vdash e : [\tau =: t]\ \sigma} \\[1em]
\Rightarrow I & \dfrac{A, P.\pi \vdash e : \rho}{A, P \vdash e : \pi \Rightarrow \rho} \\[1em]
\Rightarrow E & \dfrac{A, P \vdash e : \pi \Rightarrow \rho}{A, P \vdash e : \rho} \quad (P \Vdash \pi)
\end{array}
$$

Figure 1: Structural Rules for OLT

$$
\begin{array}{lll}
taut & P \Vdash \pi \quad (\pi \in P) & \\[0.5em]
lit & P \Vdash \mathbf{1} \leq \overline{\mathbf{0}} & P \Vdash \mathbf{2} \leq \overline{\mathbf{0}} \\
 & P \Vdash \mathbf{0} \leq \overline{\mathbf{1}} & P \Vdash \mathbf{2} \leq \overline{\mathbf{1}} \\
 & P \Vdash \mathbf{0} \leq \overline{\mathbf{2}} & P \Vdash \mathbf{1} \leq \overline{\mathbf{2}} \\[0.5em]
obs & P \Vdash O\alpha \leq \overline{\mathbf{1}} & \dfrac{P \Vdash \alpha \leq \overline{\mathbf{2}}}{P \Vdash O\alpha \leq \overline{\mathbf{2}}} \\[1em]
wf & \dfrac{P \Vdash \alpha \supseteq list\ (\beta \cdot \tau)}{P \Vdash \alpha \supseteq \tau} & \dfrac{P \Vdash \alpha \supseteq list\ (\beta \cdot \tau) \quad P \Vdash \alpha \leq \overline{\mathbf{1}}}{P \Vdash \beta \leq \overline{\mathbf{1}}}
\end{array}
$$

Figure 2: Entailment Rules for $\Vdash$

If we assume that mutable lists are implemented as vectors, such that lookups and updates have both constant cost, then the complexity of *topsort* is $O(|nodes| + |edges|)$, which matches the best known imperative algorithms. This remains true even if we use a more standard graph representation consisting of a node list and an edge list, since these lists can be converted in $O(|nodes| + |edges|)$ time into the representation we have assumed.

# 4 Typing Rules

We formulate the system OLT of observable linear types as a a system of qualified types [Jon91a]. Sequents are of the form $A, P \vdash e : \sigma$, where the type assignment $A$ is a set of assumptions $x : \sigma'$, and the context $P$ is a set of predicates $\pi$. We use $tv\ \sigma$ or $tv\ A$

$$\rightarrow I \quad \frac{A.x:\tau', P \ \vdash \ e : \tau}{A, P \ \vdash \ \lambda x.e : \alpha \cdot (\tau' \rightarrow \tau)} \quad (P \Vdash NL\ A)$$

$$\rightarrow_1 I \quad \frac{A.x:\tau', P \ \vdash \ e : \tau}{A, P \ \vdash \ \lambda_{\mathbf{1}} x.e : \mathbf{1} \cdot (\tau' \rightarrow \tau)}$$

$$\rightarrow E \quad \frac{A, P \ \vdash \ e : \alpha \cdot (\tau' \rightarrow \tau) \qquad A, P \ \vdash \ e' : \tau'}{A, P \ \vdash \ e\, e' : \tau} \quad (P \Vdash \{\alpha \leq \overline{\mathbf{0}}, \} \cup NL\ A|_{fv\ e' \cap fv\ e})$$

$$let \quad \frac{A, P \ \vdash \ e' : \sigma \qquad A.x:sigma, P \ \vdash \ e : \tau}{A, P \ \vdash \ \mathbf{let}\ x = e'\ \mathbf{in}\ e : \tau} \quad (P \Vdash NL\ A|_{fv\ e' \cap fv\ e})$$

$$let! \quad \frac{A', P \ \vdash \ e' : \sigma \qquad A.x:\sigma, P \ \vdash \ e : \tau}{A, P \ \vdash \ \mathbf{let!}\ x = e'\ \mathbf{in}\ e : \tau} \quad (obs(A', A, fv\ e), orig\ (P \Rightarrow \sigma))$$

$$if \quad \frac{A', P \ \vdash \ e_1 : bool \quad A, P \ \vdash \ e_2 : \tau \quad A, P \ \vdash \ e_3 : \tau}{A, P \ \vdash \ \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 : \tau} \quad (obs(A', A, fv e_1 \cup fv\ e_2))$$

Figure 3: Logical Rules for OLT

---

to denote the free type variables in a type scheme or type assignment. We use $fv\ e$ to denote the free program variables in an expression. We use letters $P$, $Q$, $R$ to denote sets of predicates $\pi$. Type schemes $\sigma$ will often be written $\forall \alpha_i.P \Rightarrow \tau$, where $\alpha_i$ denotes the bound variables and $P$ denotes the predicates in $\sigma$. Analogous to qualified types, we will also use *qualified type schemes* of the form $P \Rightarrow \sigma$, where the predicate $P$ constrains the free variables in $\sigma$.

Structural rules for OLT are given in Figure 1. Rule $(\Rightarrow E)$ is based on an entailment relation $\Vdash$ between predicate sets and predicates, which is defined in Figure 2. Here, rules $(lit)$ define the relationship between monomorphic alias sets $\mathbf{0}, \mathbf{1}, \mathbf{2}$ and alias sets $\overline{\mathbf{0}}, \overline{\mathbf{1}}, \overline{\mathbf{2}}$, as explained Section 2. Rules $(obs)$ determine the predicates that hold for tagged alias parts $O\alpha$: they are never linear, and are of $\mathbf{2}$-type iff the untagged alias part is of $\mathbf{2}$-type. Finally, rules $(wf)$ correspond to the well-formedness criterion on list types.

Relation $\Vdash$ is extended to a relation between predicate sets by defining $Q \Vdash P$ iff $Q \Vdash \pi$ for all $\pi \in P$. It has the following useful properties:

**Theorem 4.1** $(a)$ $\Vdash$ is monotonic, $\pi \in P$ implies $P \Vdash \pi$.
$(b)$ $\Vdash$ is transitive, $P \Vdash Q$ and $Q \Vdash R$ imply $P \Vdash R$.
$(c)$ $\Vdash$ is closed under substitution, $P \Vdash Q$ implies $SP \Vdash SQ$ for every substitution $S$.

*Proof:* $(a)$ follows from rule $(taut)$, $(b)$ and $(c)$ follow from the fact that $\Vdash$ is defined by a sequent calculus. □

**Definition.** Let $F$ denote the constraint set $\{\alpha \leq \overline{n} \mid n \in \{\mathbf{0}, \mathbf{1}, \mathbf{2}\}\}$, for an arbitrary alias part $\alpha$. A constraint set $P$ is *satisfiable* iff $P \not\Vdash F$. A qualified type scheme $P \Rightarrow (\forall \alpha_i.Q \Rightarrow \rho)$ is *empty* if $P \cup Q$ is unsatisfiable.

**Theorem 4.2** $(a)$ For every constraint set $P$ and substitution $S$, If $P$ is unsatisfiable, then so is $SP$.
$(b)$ For every constraint set $P$, it is decidable whether $P$ is satisfiable or not.

*Proof:* $(a)$ is a direct consequence of Theorem 4.1$(c)$. We now prove $(b)$. Let $P$ be a set of predicates and let $P^*$ be the $\Vdash$ closure of $P$. Then, $P \Vdash F$ iff $P^* \supseteq F$. We show that it suffices to look at the subset $P'$ of $P^*$ which consist of all predicates in $P^*$ whose alias-parts also appear in $P$. A predicate on an alias part that is in $P^*$ but not in $P$ can only be generated by application of rule $(obs)$, with conclusion $O\alpha \leq \overline{n}$, say. But then there is no way to deduce $O\ \alpha \leq \overline{\mathbf{0}}$, since there is no rule with a conclusion of this form. Hence, $P \Vdash F \Leftrightarrow P^* \supseteq F \Leftrightarrow P' \supseteq F$. Since $P'$ is finite, $(b)$ follows. ■□

Logical rules are given in Figure 3. There are two rules for the introduction of functions. Rule $(\rightarrow_1 I)$ introduces linear functions which can have global side-effects. Rule $(\rightarrow I)$ introduces functions without such effects. Absence of global side-effects is enforced in $(\rightarrow I)$ by the condition that no identifier in the type assignment A can have linear type.

Rules $(\lambda I), (\rightarrow E)$ and $(let)$ impose a nonlinearity constraint on (part of a) type assignment. $NL\ A$ yields a set of constraints which together imply that $A$ contains no linear types. It is defined by:

$$NL\ A \quad = \quad \{\alpha \leq \overline{\mathbf{1}} \mid x : \alpha{\cdot}\tau \in A\}$$

The conditions in rule $(let!)$ replace the "distinct types" condition of [Wad90b]. First, the local environment $A'$ and the global environment $A$ are related by a constraint $obs(A', A, fv\ e)$, in words: $A'$ observes $A$ on the free variables of $e$. We define relation $obs$ between type assignments, type schemes and types, and alias parts by a set of Horn clauses as follows:

$$\begin{array}{rcl}
\forall x \in fvs.obs(A'x, Ax) & \Rightarrow & obs(A', A, fvs) \\
obs(\sigma', \sigma) & \Rightarrow & obs(\forall\alpha.\sigma', \forall\alpha.\sigma) \\
obs(\rho', \rho) & \Rightarrow & obs(\pi \Rightarrow \rho', \pi \Rightarrow \rho) \\
obs(\tau', \tau) & \Rightarrow & obs(Ot{\cdot}\tau', t{\cdot}\tau) \\
obs(\tau', \tau) & \Rightarrow & obs(\mathbf{2}{\cdot}\tau', \mathbf{2}{\cdot}\tau) \\
obs(\tau', \tau) & \Rightarrow & obs(\mathbf{0}{\cdot}\tau', \mathbf{1}{\cdot}\tau) \\
obs(\tau', \tau) & \Rightarrow & obs(list\ \tau', list\ \tau) \\
obs(\tau_1', \tau_1),\ obs(\tau_2', \tau_2) & \Rightarrow & obs(\tau_1' \rightarrow \tau_2', \tau_1 \rightarrow \tau_2) \\
 & & obs(\kappa, \kappa)
\end{array}$$

This expresses that the local environment is isomorphic to the global environment, but with every part in $A|_{fvs}$ mapped to observer status. This mapping to observer state, together with the requirement that the type $\tau$ of the locally defined value may not contain observers, make the **let**! construct safe. The latter requirement is expressed by $orig\ (P \Rightarrow \sigma)$, defined as follows:

$$orig\ (P \Rightarrow \forall\alpha_i.Q \Rightarrow \tau) \quad = \quad P \cup Q \Vdash \{\alpha \leq \overline{\mathbf{0}} \mid \alpha \text{ is an alias part in } \tau\}$$

$$
\begin{array}{lll}
var & A,P \vdash x:\tau \quad (x:\sigma \in A,\ \sigma \succeq (P \Rightarrow \tau)) & \\[2ex]
\rightarrow I & \dfrac{A.x:\tau', P \vdash e:\tau}{A,P \vdash \lambda x.e : \alpha\cdot(\tau' \rightarrow \tau)} & (P \Vdash NL\ A) \\[3ex]
\rightarrow_1 I & \dfrac{A.x:\tau', P \vdash e:\tau}{A,P \vdash \lambda_{\mathbf{1}} x.e : \mathbf{1}\cdot(\tau' \rightarrow \tau)} & \\[3ex]
\rightarrow E & \dfrac{A,P \vdash e:\alpha\cdot(\tau' \rightarrow \tau) \qquad A,P \vdash e':\tau'}{A,P \vdash e\, e' : \tau} & (P \Vdash \{\alpha \le \overline{\mathbf{0}}\} \cup NL\ A|_{fv\ e' \cap fv\ e}) \\[3ex]
let & \dfrac{A,P' \vdash e':\tau' \qquad A.x:gen(A,P' \Rightarrow \tau'), P \vdash e:\tau}{A,P \vdash \mathbf{let}\ x = e'\ \mathbf{in}\ e : \tau} & (P \Vdash NL\ A|_{fv\ e' \cap fv\ e}) \\[3ex]
let! & \dfrac{\begin{array}{c} A',P' \vdash e':\tau' \\ A.x:gen(A',P' \Rightarrow \tau'), P \vdash e:\tau \end{array}}{A,P \vdash \mathbf{let!}\ x = e'\ \mathbf{in}\ e : \tau} & (obs(A',A,fv\ e)),\ orig\ (P' \Rightarrow \tau')) \\[3ex]
if & \dfrac{A',P' \vdash e_1 : bool \quad A,P \vdash e_2:\tau \quad A,P \vdash e_3:\tau}{A,P \vdash \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 : \tau} & (obs(A',A,fv\ e_1 \cup fv\ e_2))
\end{array}
$$

Figure 4: Deterministic Typing Rules for DOLT

The interpretation of these rules has to take into account that constraint sets may be unsatisfiable and that types may be empty. Since the primary motivation for type checking is to detect empty types, we adopt the following definition:

**Definition.** An expression $e$ has a type scheme $\sigma$ under type assignment $A$ and constraints $P$, written $A, P \supset e:\sigma$, if there is a proof in OLT of $A, P \vdash e:\sigma$ such that every proof step has a conclusion $A', P' \vdash e':\sigma'$ with $P' \Rightarrow \sigma'$ satisfiable.

# 5 Principal Typings and Type Reconstruction

This section states and proves the principal type property for observable linear types and gives a sketch of a type reconstruction algorithm. To simplify our task, we first define in Figure 4 another type system, DOLT, and prove its equivalence to OLT. Unlike OLT, DOLT is deterministic and syntax-directed; the structure of all proof trees for a given typing are isomorphic, and every proof step is determined uniquely by the form of the expression $e$. The typing rules of DOLT translate directly into a Prolog or Typol [CDD+85] program for type reconstruction.

The following definitions, theorems, and proofs lean heavily on the theory of qualified types developed in [Jon91a, Jon91b]. We will concentrate here on aspects which are specific to observable linear types, while referring to Jones' work for all aspects that apply to systems of qualified types in general. This is possible since the entailment relation $\Vdash$ satisfies the requirements set out in [Jon91a], as stated in Theorem 4.1.

**Definition.** A qualified type scheme $P \Rightarrow (\forall\alpha_i.Q \Rightarrow \tau)$ has a *generic instance* $R \Rightarrow \mu$, written $P \Rightarrow (\forall\alpha_i.Q \Rightarrow \tau) \;\succeq\; R \Rightarrow \mu$, iff there are types $\tau_i$ such that

$$\mu = [\alpha_i \mapsto \tau_i]\,\tau \quad and \quad R \Vdash P \cup [\alpha_i \mapsto \tau_i]\,Q.$$

**Definition.** A qualified type scheme $P \Rightarrow \sigma$ is *more general* than a qualified type scheme $P' \Rightarrow \sigma'$, written $P \Rightarrow \sigma \;\succeq\; P' \Rightarrow \sigma'$, iff $(P' \Rightarrow \sigma') \succeq \rho \;\Rightarrow\; (P \Rightarrow \sigma') \succeq \rho$. for all qualified types $\rho$.

Clearly, $\succeq$ is a preorder.

**Definition.** A qualified type scheme $P \Rightarrow \sigma$ is *principal* for an expression $e$ and a type assignment $A$, iff $A, P \supset e : \sigma$, and, if $A, P' \supset e : \sigma'$ then $P \Rightarrow \sigma \succeq P' \Rightarrow \sigma'$.

In the following, we will use $\vdash'$ for deduction in DOLT, and continue to use $\vdash$ for deduction in $OLT$.

**Theorem 5.1** (Soundness of DOLT) If $A, P \vdash' e : \tau$ then $A, P \;\vdash\; e : \tau$.

*Proof:* A straightforward induction on the structure of the proof of $A, P \vdash' e : \tau$. □

The next four lemmata have equivalents in [Jon91b] and are proved in essentially the same way as done there.

**Lemma 5.2** (Substitution lemma) If $A, P \;\vdash'\; e : \tau$ then $SA, SP \;\vdash'\; e : S\tau$, for every substitution $S$.

**Lemma 5.3** If $A, P \vdash' e : \tau$ and $Q \Vdash P$ then $A, Q \vdash' e : \tau$.

**Lemma 5.4** If $P \Vdash P'$ then $gen(A, P' \Rightarrow \tau) \succeq gen(A, P \Rightarrow \tau)$.

**Lemma 5.5** If $A.x : \sigma, P \vdash' e : \tau$ and $\sigma' \succeq P \Rightarrow \sigma$ then $A.x : \sigma', P \vdash' e : \tau$.

**Theorem 5.6** (Completeness of DOLT) If $A, P \;\vdash\; e : \tau$ then there is a set of predicates $P'$ and a type $\tau$ such that $A, P' \;\vdash\; e : \tau$ and $gen(A, P' \Rightarrow \tau) \succeq P \Rightarrow \sigma$.

*Proof:* By induction on the structure of the proof of $A, P \;\vdash\; e : \tau$. The structural rules are treated exactly as in the proof of Theorem 2, [Jon91b]. The cases for the logical rules are as follows:

**Case**$(\rightarrow I)$ : We have a derivation of the form

$$\frac{A.x : \tau', P \;\vdash\; e : \tau}{A, P \;\vdash\; \lambda x.e : \alpha \cdot (\tau' \rightarrow \tau)} \quad (P \Vdash NL\; A).$$

By induction, $A.x : \tau', P' \vdash' e : \upsilon$ for some $P', \upsilon$ with $gen(A.x : \tau', P' \Rightarrow \upsilon) \succeq P \Rightarrow \tau$. By the definition of $gen$, there is a substitution $S$ on the free type variables $\alpha_i$ of $P' \Rightarrow \upsilon$, such that $P \Vdash SP'$ and $\tau = S\upsilon$. By Lemma 5.2, and the fact that none of the $\alpha_i$ appear in $A.x:\tau'$, $A.x:\tau', SP' \vdash' e : \tau$. Define $R = SP' \cup NL\ A$. Then $P \Vdash R \Vdash SP'$. We can thus construct the derivation:

$$\cfrac{\cfrac{A.x:\tau', SP' \vdash' e : \tau}{A.x:\tau', R \vdash' e : \tau}\ (Lemma\ 5.3)}{A, R \vdash' \lambda x.e : \alpha \cdot (\tau' \to \tau)}\ (\to I)$$

Furthermore, by Lemma 5.4, $gen(A, R \Rightarrow \alpha \cdot (\tau' \to \tau)) \vdash' \succeq gen(A, P \Rightarrow \alpha \cdot (\tau' \to \tau)) \succeq P \Rightarrow \alpha \cdot (\tau' \to \tau)$.

**Case** $(\to E)$ : We have a derivation of the form:

$$\frac{A, P \vdash e : \alpha \cdot (\tau' \to \tau) \qquad A, P \vdash e' : \tau'}{A, P \vdash e\ e' : \tau} \quad (P \Vdash \alpha \leq \overline{\mathbf{0}}, P \Vdash NL\ A|_{fv\ e' \cap fv\ e})$$

By induction, $A, P' \vdash' e : \upsilon$ with $gen(A, P' \Rightarrow \upsilon) \succeq P \Rightarrow \alpha \cdot (\tau' \to \tau)$. By the definition of $gen$, there is a substitution $S$ on the free type variables $\alpha_i$ of $P' \Rightarrow \upsilon$, such that $P \Vdash SP'$ and $\alpha \cdot (\tau' \to \tau) = S\upsilon$. By Lemma 5.2, and the fact that none of the $\alpha_i$ appear in $A$, $A, SP' \vdash' e : \alpha \cdot (\tau' \to \tau)$. Using the induction hypothesis on the second premise of $(\to E)$, we can show by a similar argument that $A, S'Q' \vdash' e' : \tau'$ for some predicate set $Q'$ and substitution $S'$ such that $P \Vdash S'Q'$. Define $R = SP \cup S'P' \cup \{\alpha \leq \overline{\mathbf{0}}\} \cup NL\ A|_{fv\ e' \cap fv\ e}$. Then $P \Vdash R$. We can thus construct the following derivation:

$$\cfrac{\cfrac{A, SP' \vdash' e : \alpha \cdot (\tau' \to \tau)}{A, R \vdash' e : \alpha \cdot (\tau' \to \tau)}(Lemma\ 5.3) \qquad \cfrac{A, S'Q' \vdash' e' : \tau'}{A, R \vdash' e' : \tau'}\ (Lemma\ 5.3)}{A, R \vdash' e\ e' : \tau}\ (\to E)$$

Also, by construction, $R \Vdash \{\alpha \leq \overline{\mathbf{0}}\} \cup NL\ A|_{fv\ e' \cap fv\ e})$. Furthermore, using Lemma 5.4, $gen(A, R \Rightarrow \tau) \succeq gen(A, P \Rightarrow \tau) \succeq P \Rightarrow \tau$.

**Case** $(let!)$ : We have a derivation of the form:

$$\frac{A', P \vdash e' : \sigma \qquad A.x:\sigma, P \vdash e : \tau}{A, P \vdash \mathbf{let!}\ x = e'\ \mathbf{in}\ e : \tau} \quad (obs(A', A, fv\ e),\ orig(P \Rightarrow \sigma))$$

By induction, we have

$$\begin{array}{ll} A', P' \vdash' e : \upsilon, & \sigma' = gen(A', P' \Rightarrow \upsilon) \succeq P \Rightarrow \sigma, \\ A.x:\sigma, Q' \vdash' e' : \tau', & gen(A.x:\sigma, Q' \Rightarrow \tau') \succeq P \Rightarrow \tau. \end{array}$$

Without loss of generality, we can assume $(A)$: $tv\ P \cap tv\ (Q' \Rightarrow \tau') \subseteq tv\ (A.x:\sigma)$. This can always be achieved by a suitable renaming of the free variables in $Q' \Rightarrow \tau'$. We can construct the derivation

$$\cfrac{A', P' \vdash' e : \upsilon \qquad \cfrac{\cfrac{A.x:\sigma, Q' \vdash' e' : \tau'}{A.x:\sigma, P \cup Q' \vdash' e' : \tau'}\ (Lemma\ 5.3)}{A.x:\sigma', P \cup Q' \vdash' e' : \tau'}\ (Lemma\ 5.5)}{A, P \cup Q' \vdash' \mathbf{let!}\ x = e'\ \mathbf{in}\ e : \tau'}\ (let!)$$

with conditions $obs(A', A, fv\ e)$, $orig(P \cup Q' \Rightarrow \sigma)$ satisfied. Furthermore, using the induction hypothesis and $(A)$:

$$
\begin{aligned}
gen(A, P \cup Q' \Rightarrow \tau') &\succeq gen(A.x:\sigma, P \cup Q' \Rightarrow \tau') \\
&\succeq P \Rightarrow gen(A.x:\sigma, Q' \Rightarrow \tau') \\
&\succeq P \Rightarrow P \Rightarrow \tau \\
&= P \Rightarrow \tau
\end{aligned}
$$

Cases $(let)$ and $(if)$ are similar to cases $(let!)$ and $(\rightarrow E)$. ■□

The rules in OLT translate directly into a Prolog program where every application of a clause is determined uniquely by the outermost constructor of an expression. This program can be used to find a candidate $\sigma$ for a principal type scheme of an exprssion $e$, together with its proof tree. Given this proof tree, we can check with Theorem 4.2 $(b)$ that the type schemes in the conclusions of all proof steps are nonempty. If they are, $\sigma$ is a principal type scheme for $e$. If one of the types is empty, we can show with Theorem 4.2 $(a)$ that $e$ has no type. It therefore follows:

**Theorem 5.7** $(a)$ If an expression $e$ has a type scheme then it has a principal type scheme. $(b)$ There is a decision procedure $tp$ which returns the principal type scheme of an expression if it has one, and returns failure otherwise.

# 6 Conclusion

We have presented a type system which augments linear types with observers. We claim that the extension makes linear types practical, since it is polymorphic, accommodates a familiar programming style, and allows observer accesses to proceed in parallel. Although the typing rules are more complex than those of the classical Hindley/Milner system, typical type signatures occurring in practice are quite moderate in size and complexity. Furthermore, programmers need not write down types since principal types can be reconstructed. We see the type system as a possible candidate for future programming languages which add state to a functional core.

On the theoretical side, more research is needed to explore connections between observer types and linear logic.

# References


[Abr90] S. Abramsky. Computational interpretations of linear logic. Preprint, Imperial College, London, 1990.

[Bak90] H.G. Baker. Unify and conquer (garbage, updating, aliasing, ...) in functional languages. In *Proc. ACM Conf. on LISP and Functional Programming*, pages 218–226, June 1990.

[Blo89] A. Bloss. Update analysis and the efficient implementation of functional aggregates. In *Proc. ACM Conf. on Functional Programming Languages and Computer Architecture*, August 1989.

[CDD+85] D. Clement, J. Despeyroux, T. Despeyroux, L. Hascoet, and G. Kahn. Natural semantics on the computer. Technical Report RR 416, INRIA, June 1985.

[Coh84] S. Cohen. Multi-version structures in prolog. In *Proc. Conf. on Fifth Generation Computer Systems*, pages 265–274, 1984.

[Deu90] A. Deutsch. On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications. In *Proc. 17th ACM Symposium on Principles of Programming Languages*, Jan. 1990.

[DP90] M. Draghicescu and S. Puroshothaman. A compositional analysis of evaluation order and its application. In *Proc. ACM Conf. on Lisp and Functional Programming*, June 1990.

[GH90] J.C. Guzmán and P. Hudak. Single-threaded polymorphic lambda calculus. In *Proc. 5th IEEE Symp. on Logic in Computer Science*, June 1990.

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

[GSH88] K. Gharachorloo, V. Sarkar, and J.L. Hennessy. A simple and efficient approach for single assignment languages. In *Proc. ACM Conf. on Lisp and Functional Programming*, 1988.

[Hud87] P. Hudak. A semantic model of reference counting and its abstraction. In S. Abramsky and C. Hankin, editors, *Abstract interpretation of declarative languages*. Ellis Horwood Ltd., 1987.

[JG91] P. Jouvelot and D.K. Gifford. Algebraic reconstruction of types and effects. In *Proc. 18th ACM Symp. on Principles of Programming Languages*, pages 303–310, Jan. 1991.

[Jon91a] Mark P. Jones. Towards a theory of qualified types. Technical Report PRG-TR-6-91, Oxford University Computing Laboratory, Oxford, UK, 1991.

[Jon91b] Mark P. Jones. Type inference for qualified types. Technical Report PRG-TR-10-91, Oxford University Computing Laboratory, Oxford, UK, 1991.

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

[LG88] J. Lucassen and D.K. Gifford. Polymorphic effect systems. In *Proc. 15th ACM Symp. on Principles of Programming Languages*, pages 47–57, Jan. 1988.

[NPD87] A. Neirynk, P. Panangaden, and A. Demers. Computation of aliases and support sets. In *Proc. 14th ACM Symp. on Principles of Programming Languages*, pages 274–283, Jan. 1987.

[Ode91] M. Odersky. How to make destructive updates less destructive. In *Proc. 18th ACM Symp. on Principles of Programming Languages*, pages 25–36, Jan. 1991.

[Sch85] D.A. Schmidt. Detecting global variables in denotational specifications. *ACM Transactions on Programming Languages and Systems*, 5(2):299–310, 1985.

[SRI91] V. Swarup, U.S. Reddy, and E. Ireland. Assignments for applictive languages. In *Proc. ACM Conf. on Functional Programming Languages and Computer Architecture*, August 1991.

[TJ91] J.-P. Talpin and P. Jouvelot. Type, effect and region reconstruction in polymorphic functional languages. In *Workshop on Static Analysis of Equational, Functional, and Logic Programs*, Bordeaux, Oct. 1991.

[Wad90a] P. Wadler. Comprehending monads. In *Proc. ACM Conf. on LISP and Functional Programming*, pages 61–78, June 1990.

[Wad90b] Phil Wadler. Linear types can change the world! In *Proc. IFIP TC2 Working Conference on Programming Concepts and Methods*, pages 547–566, April 1990.

[Wad91] P. Wadler. Is there a use for linear logic? In *Proc. ACM Symp. on Partial Evaluation and Semantic-Based Program Manipulation*, pages 255–273, June 1991.