# Typing References by Effect Inference

Andrew K. Wright*
Department of Computer Science
Rice University
Houston, TX 77251-1892
wright@cs.rice.edu

*European Symposium on Programming*
Rennes France, February 1992
Springer-Verlag Lecture Notes in Computer Science

**Abstract**

Hindley/Milner-style polymorphism is a simple, natural, and flexible type discipline for functional languages, but incorporating imperative extensions is difficult. We present a new technique for typing references in the presence of polymorphism by inferring a concise summary of each expression's allocation behavior—a *type effect*. A simple technique for proving soundness with respect to a reduction semantics demonstrates that the type system prevents type errors. By establishing that the system corresponds to an alternate system better suited to implementation, we obtain an algorithm to perform type and effect inference.

## 1 Polymorphism and References

Hindley/Milner-style polymorphism [8, 12] is a simple, natural, and flexible type discipline for functional languages, but incorporating imperative extensions is difficult. While a number of systems for typing reference cells exist [3, 10, 16, 17, 18], we have devised a more direct approach based on inferring a concise summary of each expression's allocation behavior. Our system has several desirable characteristics: the curried version of a function may be used wherever the uncurried version applies; all expressions typable in the functional sublanguage are typable; the system has a direct inference rule formulation; and it can be implemented efficiently. In this paper, we discuss the typing of STANDARD ML's reference cells by our method, give a formal description of a type system for references, and show how it may be implemented. We begin with an illustration of the difficulties involved in typing references in the presence of polymorphism, and outline our solution.

*This research was supported in part by the United States Department of Defense under a National Defense Science and Engineering Graduate Fellowship, and by NSF grant CCR 89–17022.

## 1.1 Hindley/Milner Polymorphism

Hindley/Milner-style type systems express polymorphism with let-expressions. In functional languages, understanding such expressions as abbreviations offers a simple explanation of polymorphism. Semantically, the expression:

$$\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$$

has the same meaning as $e_2[x/e_1]$, the capture-avoiding substitution of $e_1$ for free $x$ in $e_2$ (assuming that $e_1$ does not diverge). However, in typing the substituted expression $e_2[x/e_1]$, each occurrence of the bound expression $e_1$ may have a different type. For example, if $e_1$ is $\lambda\mathsf{z}.\mathsf{z}$, one occurrence could be assigned $int \rightarrow int$, while another occurrence is assigned $bool \rightarrow bool$. Hence to type a let-expression we associate with $x$ the *set* of types of $e_1$. Each occurrence of $x$ in the body $e_2$ may have any type in this set. In the expression:

$$\begin{array}{l}\mathsf{let\ id} = \lambda\mathsf{z}.\mathsf{z}\\ \mathsf{in\ id\ 1;\ \ id\ true}\end{array}$$

id is associated with the infinite set of types $\{\tau \rightarrow \tau \mid \tau \in \mathit{Type}\}$.

*Type schemes* represent the sets of types that are associated with identifiers bound by let-expressions. A type scheme $\forall\alpha_1 \ldots \alpha_n.\tau$ consists of a body ($\tau$), which is a type that may contain *type variables* ($\alpha_i$), and a set of *bound variables* ($\alpha_1 \ldots \alpha_n$). The set of types described by a type scheme consists of those types that may be obtained by substituting types for the bound variables in the body of the type scheme. The type scheme $\forall\alpha.\alpha \rightarrow \alpha$ describes the set of all function types whose input and output types are the same, *i.e.*, $\{\tau \rightarrow \tau \mid \tau \in \mathit{Type}\}$.

To determine the type scheme for $x$ in the expression $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$, we first find a most general type for $e_1$ by using type variables wherever possible. For example, a most general type for $\mathsf{compose} = \lambda\mathsf{f}.\lambda\mathsf{g}.\lambda\mathsf{z}.\mathsf{f}\ (\mathsf{g}\ \mathsf{z})$ is:

$$(\beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \gamma.$$

The type scheme for $x$ is obtained by binding or *generalizing* type variables in the type of $e_1$ (that are not used in typing expressions outside the let-expression in question [13: p. 40]). In the expression:

$$\mathsf{let\ compose} = \lambda\mathsf{f}.\lambda\mathsf{g}.\lambda\mathsf{z}.\mathsf{f}\ (\mathsf{g}\ \mathsf{z})\ \ \mathsf{in}\ \ \ldots$$

$\alpha$, $\beta$, and $\gamma$ are generalized to yield type scheme $\forall\alpha\beta\gamma.(\beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \gamma$ for compose.

## 1.2 References

The operators ref, !, and := provide reference cells as first-class values. When applied to a value, the ref operator creates a reference cell containing that value; applying ! to a reference cell extracts the contents of the cell; and := changes the contents of a cell.

To type reference cells, we introduce the type $\tau\ \mathit{ref}$ for reference cells containing values of type $\tau$. Since the ref operator takes a value of any type and returns a cell containing that value, one might naïvely expect ref to have type scheme $\forall\alpha.\alpha \rightarrow \alpha\ \mathit{ref}$,

! to have type scheme $\forall\alpha.\, \alpha\ \mathit{ref} \rightarrow \alpha$, and := to have type scheme $\forall\alpha.\, \alpha\ \mathit{ref} \rightarrow \alpha \rightarrow \alpha$ (:= returns the value assigned). However, this naïve typing for references is unsound, as the following expression illustrates:

$$\mathsf{let\ x = ref\ (\lambda z.z)\ \ in\ \ x := (\lambda n.\ n + 1);\ \ (!\ x)\ true}$$

The most general type for ref ($\lambda$z.z) is $(\alpha \rightarrow \alpha)\ \mathit{ref}$, and generalizing $\alpha$ yields type scheme $\forall\alpha.\, ((\alpha \rightarrow \alpha)\ \mathit{ref})$ for x. This type scheme can be instantiated to $(\mathit{int} \rightarrow \mathit{int})\,\mathit{ref}$ to type the assignment, and to $(\mathit{bool} \rightarrow \mathit{bool})\ \mathit{ref}$ to type the dereference, but when evaluated, this expression causes a type error by attempting to add 1 to true.

Reference cells invalidate the explanation of let-expressions as abbreviations, due to the sharing implied by references: the expression let $x = e_1$ in $e_2$ no longer has the same meaning as $e_2[x/e_1]$. Just as references change the semantics of let-expressions, they also necessitate a change in how let-expressions are typed.

## 1.3 The Problem is Generalization

The solution our system and all existing systems use is to require that reference cells have only one type. This is achieved by restricting generalization at let-expressions. In the previous example, if $\alpha$ is not generalized, then it is *free* in the resulting type scheme $\forall.\, ((\alpha \rightarrow \alpha)\ \mathit{ref})$. A free type variable may later be replaced with a specific type; in the above example, typing the assignment replaces $\alpha$ with $\mathit{int}$. The subsequent dereference can no longer be typed, as x now has type scheme $\forall.\, ((\mathit{int} \rightarrow \mathit{int})\ \mathit{ref})$.

The type variables that must not be generalized are those that appear in the types of reference cells allocated by the bound expression [18]. However, this set cannot be precisely determined, since the set of cells allocated by an expression cannot be statically determined. Hence any static type system that attempts to integrate reference cells in this way must use a conservative approximation. Our system uses a more direct method than existing systems to approximate the set of type variables that appear in the types of allocated reference cells.

## 1.4 Controlling Generalization with Effects

The essential idea behind our system is to associate with every expression a conservative approximation to the set of reference cells that the expression allocates, the expression's *allocation effect.*[1] Since our type system needs only the type variables in the types of these reference cells, we infer for each expression a *type effect*—the set of type variables that appear in the expression's allocation effect. In typing a let-expression, the type effect of the bound expression provides the information we need to determine which type variables must not be generalized.

The type effect of an application $(e_1\ e_2)$ is a combination of the effects of evaluating the function and argument subexpressions $e_1$ and $e_2$, and of the effect caused by applying the function to which $e_1$ evaluates. Hence, we record the type effect that the function causes when applied above the arrow in the type of a function. For example, the function:

$$\mathsf{\lambda x.\,(ref\ 1;\ ref\ true;\ ref\ x;\ false)}$$

[1] We borrow the term *effect* from FX [11]. Jouvelot and Gifford [9] describe a system that infers types and effects for expressions. While this system records effect information with function types in a manner similar to ours, the information is not used to control generalization of let-expressions.

allocates cells of type *int*, *bool*, and $\alpha$ when applied (where x has type $\alpha$). The type effect of the set $\{int, bool, \alpha\}$ is $\alpha$, since $\alpha$ is the only type variable in the set, hence the type of the above function is $\alpha \xrightarrow{\alpha} bool$. The $\lambda$-expression itself has the empty type effect, written $\emptyset$, because its evaluation to a closure allocates no cells.

To handle higher-order functions, we introduce *effect variables*, denoted $\varsigma$. Effect variables are analogous to type variables; for example, the function:

$$\mathsf{apply} = \lambda \mathsf{f}.\lambda \mathsf{x}.\mathsf{f}\ \mathsf{x}$$

has type $(\alpha \xrightarrow{\varsigma} \beta) \xrightarrow{\emptyset} \alpha \xrightarrow{\varsigma} \beta$ for any types $\alpha$ and $\beta$ and type effect $\varsigma$. Application of apply to a single argument yields a result of type $\alpha \xrightarrow{\varsigma} \beta$, causing no effect. The type effect $\varsigma$ occurs only when apply is given a second argument, as indicated by the empty type effect on its outermost function constructor. As another example, the function:

$$\mathsf{compose} = \lambda \mathsf{f}.\lambda \mathsf{g}.\lambda \mathsf{x}.\ \mathsf{f}\ (\mathsf{g}\ \mathsf{x})$$

has type $(\alpha \xrightarrow{\varsigma_1} \beta) \xrightarrow{\emptyset} (\gamma \xrightarrow{\varsigma_2} \alpha) \xrightarrow{\emptyset} \gamma \xrightarrow{\varsigma_1\varsigma_2} \beta$. When applied to three arguments, it causes the combined effects of both f and g.

As in the functional case, the type scheme for $x$ in the expression let $x = e_1$ in $e_2$ is determined by generalizing variables in the type of the bound expression $e_1$. To prevent generalizing variables that appear in the types of cells allocated by $e_1$, we simply restrict generalization to those variables not in the effect of $e_1$. For example, in the expression:

$$\mathsf{let}\ \mathsf{x} = \mathsf{ref}\ (\lambda \mathsf{y}.\mathsf{y})\ \ \mathsf{in}\ \ \mathsf{x} := (\lambda \mathsf{n}.\ \mathsf{n} + 1);\ \ (!\ \mathsf{x})\ \mathsf{true}$$

the bound expression ref ($\lambda$y.y) has type $(\alpha \xrightarrow{\emptyset} \alpha)\ ref$ and effect $\{\alpha\}$. Since $\alpha$ appears in the effect, it cannot be generalized, and the expression is not typable.

As a further example, consider the following imperative version of map [10]:

```
let imap = λf.λx. let a = ref x  and  b = ref nil
                  in  while not (null !a) do
                              b := (f (hd !a)) :: !b;
                              a := tl !a
                      reverse !b
in ...
```

If f has type $\alpha \xrightarrow{\varsigma} \beta$ and x has type $\alpha\ list$ ($\tau\ list$ is the type of lists containing elements of type $\tau$), then the body of the function allocates reference cells of type $\alpha\ list\ ref$ and $\beta\ list\ ref$. While $\alpha$ and $\beta$ must not be generalized by the inner let in the type schemes for a and b, they can be generalized by the outer let in the type scheme for imap: $\forall \alpha\beta\varsigma_1\varsigma_2\varsigma_3.\ (\alpha \xrightarrow{\varsigma_1} \beta) \xrightarrow{\varsigma_2} \alpha\ list \xrightarrow{\varsigma_1\varsigma_3\alpha\beta} \beta\ list$.

## 1.5 Outline

In the next section, we present the syntax and semantics of a simple language with references. Section 3 defines our type system for this language in detail, and sketches a proof of soundness. In Section 4 we obtain a corresponding type inference algorithm by reformulating the system in a manner better suited to implementation. We conclude with a comparison to other systems for typing references in the presence of polymorphism.

## 2 A Polymorphic Language with References

We study our type system in the context of a simple language with references. It is easy to extend this language and its type system to a realistic language including pattern matching, exceptions, and modules, such as Standard ML [13, 14].

### 2.1 Abstract Syntax

Our language has expressions, *Exp*, and values, *Val*, of the form:

$$
\begin{array}{ll}
(\mathit{Exp}) & e ::= v \mid e_1\ e_2 \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \\
(\mathit{Val}) & v ::= x \mid c \mid \lambda x.e \mid \mathsf{setref}\ v \\
 & x \in \mathit{Id} \qquad c, \mathsf{ref}, !, \mathsf{setref} \in \mathit{Const}
\end{array}
$$

The set *Id* is a countably infinite set of identifiers (whenever we refer to *variables*, we mean entities of the type system, not identifiers). The set of constants, *Const*, consists of data, of primitive operations, and of the distinguished constants ref, !, and setref.

Juxtaposition denotes application and is left associative; $\lambda$ constructs call-by-value procedural abstractions. Semantically a let-expression behaves like $((\lambda x.e_2)\ e_1)$; however, the type system allows $x$ to be polymorphic. The constants ref, !, and setref provide the usual operations on references. We use the curried binary assignment operator setref rather than the customary infix := to simplify the language. The application of setref to one value *is* a value—it may be thought of as a *capability* to assign to a cell. Therefore, the application of setref to a value is included in the syntactic class of values.

Free and bound identifiers are defined as usual. Following Barendregt [1], we assume that bound identifiers are always distinct from free identifiers in distinct metavariables ranging over expressions, and we identify expressions that differ by only a consistent renaming of the bound identifiers.

### 2.2 Semantics

Rather than using (a variant of) structural operational semantics [13, 14] to give a formal semantics for our language, we define the semantics with a term rewriting system, using the technique of *reduction semantics* [5, 6, 7]. This formulation allows a compact and elegant presentation of the semantics and a simple proof of type soundness [19].

Evaluation proceeds as a sequence of rewriting steps, or *reductions*, from one intermediate *state* of evaluation to another. Each state has a syntactic representation:

$$
\begin{array}{ll}
(\mathit{State}) & s ::= \rho\theta.e \\
(\mathit{Store}) & \theta ::= \{\, \langle x, v\rangle \,\}^*
\end{array}
$$

The sequence $\theta$ represents the contents of the store. The first component of a pair $\langle x, v\rangle$ is a location name; the second component is the value stored there. The phrase $\rho\langle x_1, v_1\rangle \ldots \langle x_n, v_n\rangle.e$ binds $x_1, \ldots, x_n$ in $v_1, \ldots, v_n, e$; hence, $\rho$-phrases permit recursive bindings. While $\theta$ is syntactically a sequence of pairs, we treat $\theta$ as a finite function, *i.e.*, we disregard the order of pairs and require that the first components be unique.

| | | | |
|---|---|---|---|
| $(\delta)$ | $\rho\theta.E[c\ c'] \longmapsto \rho\theta.E[\delta(c,c')]$ | if $\delta(c,c')$ is defined |
| $(\beta_{\mathrm{v}})$ | $\rho\theta.E[(\lambda x.e)\ v] \longmapsto \rho\theta.E[e[x/v]]$ | |
| $(let_{\mathrm{v}})$ | $\rho\theta.E[\mathsf{let}\ x = v\ \mathsf{in}\ e] \longmapsto \rho\theta.E[e[x/v]]$ | |
| $(ref)$ | $\rho\theta.E[\mathsf{ref}\ v] \longmapsto \rho\theta\langle x,v\rangle.E[x]$ | |
| $(!)$ | $\rho\theta\langle x,v\rangle.E[!\ x] \longmapsto \rho\theta\langle x,v\rangle.E[v]$ | |
| $(setref)$ | $\rho\theta\langle x,v'\rangle.E[\mathsf{setref}\ x\ v] \longmapsto \rho\theta\langle x,v\rangle.E[v]$ | |

$$E ::= [\,] \mid E\ e \mid v\ E \mid \mathsf{let}\ x = E\ \mathsf{in}\ e$$

Figure 1: The reduction relation $\longmapsto$

As with expressions, we identify states that differ by a consistent renaming of bound identifiers.

Figure 1 specifies the reduction relation $\longmapsto$: *State* $\times$ *State*. The notation $e[x/v]$ means the capture-avoiding substitution of $v$ for free $x$ in $e$. The partial function $\delta : \mathit{Const} \times \mathit{Const} \rightharpoonup \mathit{Const}$ interprets the application of constants other than ref, !, and setref; the type system places some additional constraints on $\delta$ to achieve soundness. The renaming conventions ensure that the identifiers bound by $\rho$-phrases in the *ref*, *!*, and *setref* reductions are renamed appropriately to avoid capture, as in $\beta_{\mathrm{v}}$.

The definition of the reduction relation relies on *evaluation contexts*, $E$. An evaluation context is an expression with one subexpression replaced by a hole, denoted $[\,]$. An expression may be placed in the hole of an evaluation context, yielding an expression; we write $E[e]$. The definition of evaluation contexts forces evaluation to proceed from left to right. As a result, the relation $\longmapsto$ is a function.

The evaluation function *eval* maps *programs* to *answers*:

$$\mathit{eval}(e) = a \text{ if and only if } \rho.e \longmapsto\!\!\!\!\rightarrow a$$

where $\longmapsto\!\!\!\!\rightarrow$ is the transitive and reflexive closure of the reduction relation $\longmapsto$. Programs are simply closed expressions; answers ($a$) are states of the form $\rho\theta.v$.

The evaluation function is partial and may be undefined for two reasons: evaluation may *diverge*, or it may become *stuck*. An expression diverges, written $e\Uparrow$, if it has an infinite reduction sequence, *i.e.*, if $e \longmapsto e'$ for some $e'$, and for all $e'$ such that $e \longmapsto\!\!\!\!\rightarrow e'$, there exists $e''$ such that $e' \longmapsto e''$. Evaluation is stuck if it reaches a state that is not an answer, but from which no further reduction is possible. Stuck states represent the application of a primitive function to an argument for which it is not defined, or the application of a non-function; examples are $\rho.$(succ true) and $\rho.(1\ 2)$. When a program reaches a stuck state, it is said to have caused a *type error*. The intent of a static type system is to filter out programs that may cause type errors.

# 3 Typing References

The following subsections present a type system for our language. The system requires a syntactic description of types and type inference rules for assigning types to programs. A type soundness theorem establishes that the system filters out all type errors.

## 3.1 Types and Effects

The sets of types, effects, and type schemes are defined inductively as follows:

| | |
|---|---|
| (*Type*) | $\tau ::= \iota \mid \alpha \mid \tau \xrightarrow{\Delta} \tau \mid \tau\ ref$ |
| (*Effect*) | $\Delta ::= \nu^*$ |
| (*TypeScheme*) | $\sigma ::= \forall \nu^*.\, \tau$ |
| (*Var*) | $\nu ::= \alpha \mid \varsigma$ |

$$\alpha \in \mathit{TypeVar} \quad \varsigma \in \mathit{EffectVar} \quad \iota \in \mathit{TypeConst}$$

The set *TypeConst* is a finite set of ground types, like *int* and *bool*. The sets *TypeVar* and *EffectVar* are countably infinite and disjoint. An effect is syntactically a sequence of type and effect variables, but we treat effects as finite sets, identifying effects that have the same elements. We write $\emptyset$ for the empty effect. We also identify $\forall.\tau$ with $\tau$; hence, the set of types is a proper subset of the set of type schemes.

A type scheme $\forall \nu_1 \dots \nu_n.\, \tau$ binds $\nu_1$ through $\nu_n$ in $\tau$, giving rise to free and bound variables for types and type schemes; $FV(\sigma)$ is the set of free variables of $\sigma$. A type scheme $\forall \nu_1 \dots \nu_n.\tau$ denotes the set of types that may be obtained by substituting for its bound variables:

$$\{\tau' \mid \tau' = \mu\tau \text{ for some substitution } \mu \text{ with domain } \{\nu_1, \dots, \nu_n\}\}.$$

Substitutions are finite maps from type variables to types and effect variables to effects; juxtaposition denotes application. Substitutions are applied to types as usual, but the application of a substitution $\mu$ to an effect $\Delta$ yields the set of variables appearing in the pointwise application of $\mu$ to each member of $\Delta$:

$$\mu\Delta = \bigcup_{\nu \in \Delta} FV(\mu\nu).$$

For example, with $\mu = \{\alpha \mapsto (\beta \xrightarrow{\gamma} \beta)\}$ and $\Delta = \alpha\varsigma$,

$$\mu\Delta = FV(\mu\alpha) \cup FV(\mu\varsigma) = FV(\beta \xrightarrow{\gamma} \beta) \cup FV(\varsigma) = \beta\gamma\varsigma.$$

So far, our definitions admit several different type schemes that denote the same set of types. As usual, we identify type schemes that differ by only a consistent renaming of bound variables. However, this does not identify all type schemes that denote the same set of types; witness $\forall \alpha\varsigma.\, (\alpha \xrightarrow{\varsigma} \alpha) \xrightarrow{\varsigma} \alpha$ and $\forall \beta\varsigma_1\varsigma_2.\, (\beta \xrightarrow{\varsigma_1\varsigma_2} \beta) \xrightarrow{\varsigma_1\varsigma_2} \beta$. To identify these type schemes, we extend the renaming process to allow the consistent replacement of an effect variable by one or more effect variables. Under this extended process of renaming, all type schemes that denote the same set of types are equivalent.

## 3.2 Type and Effect Assignment

The type system in Figure 2 is a deductive proof system that assigns a type and an effect to an expression. A type judgement $\Gamma \vdash e : \tau, \Delta$ states that expression $e$ has type $\tau$ and effect $\Delta$ in type environment $\Gamma$. A type environment is a finite map from identifiers to type schemes; type environments give types to the free identifiers of an

$$TypeOf(\mathsf{ref}) = \forall \alpha\varsigma.\, \alpha \xrightarrow{\alpha\varsigma} \alpha\ ref$$
$$TypeOf(!) = \forall \alpha\varsigma.\, \alpha\ ref \xrightarrow{\varsigma} \alpha$$
$$TypeOf(\mathsf{setref}) = \forall \alpha\varsigma_1\varsigma_2.\, \alpha\ ref \xrightarrow{\varsigma_1} \alpha \xrightarrow{\varsigma_2} \alpha$$

$$(id) \qquad \Gamma \vdash x : \tau, \emptyset \ \text{ if } \ \tau \in \Gamma(x)$$

$$(const) \qquad \Gamma \vdash c : \tau, \emptyset \ \text{ if } \ \tau \in TypeOf(c)$$

$$(abs) \qquad \frac{\Gamma[x/\tau_1] \vdash e : \tau_2, \Delta}{\Gamma \vdash \lambda x.e : \tau_1 \xrightarrow{\Delta} \tau_2, \emptyset}$$

$$(app) \qquad \frac{\Gamma \vdash e_1 : \tau_1 \xrightarrow{\Delta_3} \tau_2, \Delta_1 \quad \Gamma \vdash e_2 : \tau_1, \Delta_2}{\Gamma \vdash e_1\ e_2 : \tau_2, \Delta_1 \cup \Delta_2 \cup \Delta_3}$$

$$(let) \qquad \frac{\Gamma \vdash e_1 : \tau_1, \Delta_1 \quad \Gamma[x/Close(\tau_1, \Gamma, \Delta_1)] \vdash e_2 : \tau_2, \Delta_2}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2, \Delta_1 \cup \Delta_2}$$

$$(sub) \qquad \frac{\Gamma \vdash e : \tau, \Delta_1 \quad \Delta_1 \subseteq \Delta_2}{\Gamma \vdash e : \tau, \Delta_2}$$

$$Close(\tau, \Gamma, \Delta) = \forall \nu_1 \ldots \nu_n.\, \tau \ \text{ where } \{\nu_1, \ldots, \nu_n\} = FV(\tau) \setminus (FV(\Gamma) \cup FV(\Delta))$$

Figure 2: Type and effect assignment

expression. The notation $\Gamma[x/\sigma]$ means the functional extension or update of $\Gamma$ at $x$ to $\sigma$. An expression $e$ is *well-typed* if there exists a derivation that assigns a type $\tau$ and an effect $\Delta$ to $e$ in the empty environment; we write $\vdash e : \tau, \Delta$.

The rules for typing variables and constants depend on a notion of *generalization*. A type scheme $\sigma = \forall \nu_1 \ldots \nu_n.\, \tau'$ generalizes a type $\tau$, written $\tau \in \sigma$, if there exists a substitution $\mu$ with domain $\{\nu_1, \ldots, \nu_n\}$ such that $\mu\tau' = \tau$, *i.e.*, if $\tau$ is in the set of types denoted by $\sigma$. Alternatively, we say that $\tau$ is an instance of $\sigma$. For example,

$$int \xrightarrow{\emptyset} int \in \forall \alpha\varsigma.\, \alpha \xrightarrow{\varsigma} \alpha$$
$$(\beta \xrightarrow{\gamma} \beta) \xrightarrow{\beta\gamma} (\beta \xrightarrow{\gamma} \beta)\ ref \in \forall \alpha\varsigma.\, \alpha \xrightarrow{\alpha\varsigma} \alpha\ ref$$

In the last example, note that when the substitution $\{\alpha \mapsto (\beta \xrightarrow{\gamma} \beta); \varsigma \mapsto \emptyset\}$ is applied to the effect $\alpha\varsigma$ on the function arrow, the result is the "flattened" effect $\beta\gamma$.

As rule $(id)$ indicates, an identifier may have any type that is an instance of its type scheme in the type environment. As an identifier has a binding in the type environment only if it is bound by a surrounding let- or $\lambda$-expression, well-typed programs are closed. An identifier has the empty effect since its evaluation allocates no reference cells.

Rule $(const)$ uses the function $TypeOf : Const \rightarrow TypeScheme$ to assign type schemes to constants. For type soundness to make sense for an unspecified set of constants, we impose a *typability* condition on the interpretation of constants:

$$(\delta\text{-typability}) \qquad \begin{array}{c} \tau' \xrightarrow{\Delta} \tau \in TypeOf(c_1) \text{ and } \tau' \in TypeOf(c_2) \\ \text{implies} \\ \delta(c_1, c_2) \text{ is defined and } \tau \in TypeOf(\delta(c_1, c_2)). \end{array}$$

This condition requires $\delta$ to be defined for all constants of functional type and arguments of matching type, and restricts the range of values that it may produce.[2]

The type of an abstraction $\lambda x.e$ is determined from the type and effect of its body, as $(abs)$ indicates. Assuming that the argument $x$ has type $\tau_1$, if the body $e$ has type $\tau_2$ and effect $\Delta$, then the abstraction is a function of type $\tau_1 \xrightarrow{\Delta} \tau_2$. Since the evaluation of an abstraction itself creates no reference cells, the effect of an abstraction is empty.

To determine the type of an application, the argument's type is required to match the function's type as usual. The effect of an application is the union of the effects of its subexpressions and of the effect the function causes.

The type of a let-expression is determined from the type of the body, in a type environment extended with a generalization of the most general type of the bound expression. The function *Close* generalizes free type and effect variables of the bound expression's type that are not also free in the type environment or in the bound expression's effect (the function $FV$ is extended pointwise to type environments).

The subsumption rule $(sub)$ allows any expression to be treated as having more effects than it actually does. By applying subsumption to the body of an abstraction, the abstraction may be treated as causing more effects when applied than it actually does; for example, $\lambda$x.x may be typed as $\alpha \xrightarrow{\Delta} \alpha$ for any effect $\Delta$. Hence the type scheme for a function includes an effect variable in its type. For example, the type scheme for the identity function $\lambda$x.x is $\forall \alpha\varsigma.\, \alpha \xrightarrow{\varsigma} \alpha$; likewise, the type scheme for ref is $\forall \alpha\varsigma.\, \alpha \xrightarrow{\alpha\varsigma} \alpha\ ref$. Without subsumption, a value of type $int \xrightarrow{\emptyset} int$ and a value of type $int \xrightarrow{int} int$ could not both be passed as arguments to the same (non-polymorphic) function, because the function and argument types would have to match exactly.

### 3.3 Type Soundness

To prove that this type system is sound, we use our extension of subject reduction to imperative languages [19]. By expressing the semantics of the language as a term rewriting system, we can use the type system to check that each intermediate state of evaluation is well-typed, and thus show that evaluation preserves typing. Since no stuck state is typable, it follows that evaluation cannot reach a stuck state and cause a type error. The soundness theorem states that all well-typed programs either diverge or produce an answer of the expected type.

**Theorem 3.1 (Type Soundness)** *If* $\vdash s : \tau, \Delta$ *then* $s \Uparrow$ *or* $s \longmapsto\!\!\!\!\rightarrow a$ *and* $\vdash a : \tau, \Delta$.

**Proof Sketch**. In order to show that evaluation preserves typing, it must be possible to type all intermediate states of evaluation. Therefore, we augment the type system with a typing rule for states:

$$(state) \qquad \frac{\begin{array}{l}\emptyset[x_1/\tau_1\ ref]\ldots[x_n/\tau_n\ ref] \vdash e : \tau, \Delta \\ \emptyset[x_1/\tau_1\ ref]\ldots[x_n/\tau_n\ ref] \vdash v_i : \tau_i, \emptyset \quad 1 \le i \le n\end{array}}{\vdash \rho\langle x_1, v_1\rangle \ldots \langle x_n, v_n\rangle.e : \tau, \Delta}$$

[2] This condition precludes partial constant functions that are not defined on all values of their input type, such as $\div : int \times int \to int$. Such functions may be extended to total functions by raising *exceptions*. Typing exceptions requires similar restrictions on polymorphism as typing references; type soundness for exceptions is treated in [19].

In the augmented system, reductions preserve the type and effect of a state:

$$\textit{if } \vdash s_1 : \tau, \Delta \textit{ and } s_1 \longmapsto s_2 \textit{ then } \vdash s_2 : \tau, \Delta.$$

This lemma (subject reduction) is proved by case analysis according to the reduction.

Stuck states have the form:

$$\begin{array}{l} \rho\theta.E[c\ v] \textit{ where } \delta(c, v) \textit{ is undefined and } c \neq \mathsf{ref}, !, \mathsf{setref}; \\ \rho\theta.E[!\ v] \textit{ where } v \notin Id; \\ \rho\theta.E[\mathsf{setref}\ v_1\ v_2] \textit{ where } v_1 \notin Id; \textit{ or} \\ \rho\theta\langle x, v_1\rangle.E[x\ v_2]. \end{array}$$

By examining each case in the definition of stuck states, we can show that no stuck state is typable.

Finally, we know that either $s \Uparrow$, $s \longmapsto\!\!\!\!\!\!\rightarrow a$, or $s \longmapsto\!\!\!\!\!\!\rightarrow s'$ and $s'$ is stuck. Since $\vdash s : \tau, \Delta$, type and effect preservation implies $\vdash a : \tau, \Delta$ and $\vdash s' : \tau, \Delta$. Suppose $s \longmapsto\!\!\!\!\!\!\rightarrow s'$ and $s'$ is stuck. Since stuck states are untypable, $\vdash s' : \tau, \Delta$ is a contradiction, therefore this case cannot occur. Hence either $s \Uparrow$ or $s \longmapsto\!\!\!\!\!\!\rightarrow a$ and $\vdash a : \tau, \Delta$. ∎

# 4 Implementing the Type System

The type system of the preceding section may be used to verify that a typing is valid, but we would like an *algorithm* that infers a typing if one exists. The algorithms for functional languages rely on ordinary unification to achieve efficient implementation. However, our function types include effect sets, hence unifying two function types apparently requires unifying two sets. Fortunately, we can obtain an *indirect* reformulation of the type system that has a corresponding algorithm using ordinary unification.

Subsumption is the key to avoiding set unification. To type the application $(e_1\ e_2)$ where $e_1$ and $e_2$ have types:

$$e_1 : (\tau \xrightarrow{\Delta_1} \tau') \rightarrow \tau'' \quad \text{and} \quad e_2 : \tau \xrightarrow{\Delta_2} \tau'$$

we must unify $\tau \xrightarrow{\Delta_1} \tau'$ and $\tau \xrightarrow{\Delta_2} \tau'$. But subsumption permits embedded effects to be unbounded, hence we can retype $e_1$ and $e_2$ as:

$$e_1 : (\tau \xrightarrow{\Delta_1 \cup \Delta_2} \tau') \rightarrow \tau'' \quad \text{and} \quad e_2 : \tau \xrightarrow{\Delta_1 \cup \Delta_2} \tau'.$$

The function and argument types now match appropriately.

To permit the use of ordinary unification, we replace the effect sets on function arrows with labels and record the effect information separately in a *constraint*. The constraint is simply a list of label–effect pairs; the effect information corresponding to a given label is determined by pairs in the constraint with that label. To unite two effects embedded in types, we simply unify their labels in both the types and the constraint. Effect variables provide a convenient source of labels.

In the following subsections we present the indirect system, sketch a proof of its correspondence to the direct system, and give the type inference algorithm.

## 4.1 The Indirect System

In the indirect system, functions are labeled with only a single effect variable. *Constraints* record the effect information separately:

| | |
|---|---|
| (*Type*) | $\tau ::= \iota \mid \alpha \mid \tau \xrightarrow{\varsigma} \tau \mid \tau\ ref$ |
| (*Constraint*) | $\kappa ::= \{\ \langle \varsigma, \nu \rangle\ \}^*$ |
| (*TypeScheme*) | $\sigma ::= \forall \nu^*.\, \tau\ \mathsf{with}\ \kappa$ |

A constraint is treated as a set of pairs of an effect variable and either a type variable or an effect variable. The effect information associated with an effect variable $\varsigma$ under constraint $\kappa$ is the least set $\mathcal{E}$ of type and effect variables such that $\varsigma \in \mathcal{E}$, and if $\varsigma' \in \mathcal{E}$ and $\langle \varsigma', \nu \rangle \in \kappa$ then $\nu \in \mathcal{E}$. For example, the effect information corresponding to $\varsigma_1$ in the constraint $\langle \varsigma_1, \alpha_1 \rangle \langle \varsigma_1, \alpha_2 \rangle \langle \varsigma_1, \varsigma_2 \rangle \langle \varsigma_2, \alpha_3 \rangle \langle \varsigma_3, \alpha_4 \rangle$ is $\{\alpha_1, \alpha_2, \alpha_3, \varsigma_1, \varsigma_2\}$.

Generalization for the indirect system is defined as:

$$
\begin{array}{c}
(\tau, \kappa)\ \in\ \forall \nu_1 \ldots \nu_n.\, \tau'\ \mathsf{with}\ \kappa' \\
\text{iff} \\
\mathrm{Dom}(S) = \{\nu_1, \ldots, \nu_n\} \text{ and } S\tau' = \tau \text{ and } S\kappa' \subseteq \kappa
\end{array}
$$

where $S$ is a substitution from type variables to types and from effect variables *to effect variables*. Application of substitutions to types is defined as usual; substitution on constraints is defined as follows:

$$S\kappa = \bigcup_{\langle \varsigma, \nu \rangle \in \kappa} \{ \langle S\varsigma, \nu' \rangle \mid \nu' \text{ occurs in } S\nu \}.$$

The indirect system derives judgements of the form $\Gamma \triangleright e : \tau, \varsigma, \kappa$, meaning that expression $e$ has type $\tau$ and effect $\varsigma$ in type environment $\Gamma$ under constraint $\kappa$. Figure 3 presents the typing rules for the indirect formulation; Figure 4 defines the function $IFV$ to compute free type and effect variables for the indirect formulation.

## 4.2 Correspondence of the Direct and Indirect Systems

To prove that an expression has a type in the indirect system if and only if it has a type in the direct system, we must account for two differences between the direct and indirect systems. First, the indirect system represents effect information by constraints. Second, the indirect system does not have an explicit subsumption rule, but instead folds subsumption into the other rules. We prove each direction separately by constructing translations from a type derivation in one system into a derivation in the other system.

For the translation from the indirect system to the direct system, Figure 5 defines the translation $\mathcal{D}$ from indirect types to direct types. The two systems must agree on the types of constants, hence we require that all constants be of closed type, and that:

$$TypeOf(c) = \mathcal{D}[\![IndTypeOf(c)]\!]\kappa \text{ for all } \kappa.$$

If $e$ has a derivation in the indirect system, then $e$ has a corresponding derivation in the direct system.

$$\begin{aligned}
\mathit{IndTypeOf}(\mathsf{ref}) &= \forall\alpha\varsigma.\,\alpha \xrightarrow{\varsigma} \alpha\ \mathit{ref}\ \mathsf{with}\ \langle\varsigma,\alpha\rangle\\
\mathit{IndTypeOf}(!) &= \forall\alpha\varsigma.\,\alpha\ \mathit{ref} \xrightarrow{\varsigma} \alpha\ \mathsf{with}\ \emptyset\\
\mathit{IndTypeOf}(\mathsf{setref}) &= \forall\alpha\varsigma_1\varsigma_2.\,\alpha\ \mathit{ref} \xrightarrow{\varsigma_1} \alpha \xrightarrow{\varsigma_2} \alpha\ \mathsf{with}\ \emptyset
\end{aligned}$$

$(id)$ $$\Gamma \triangleright x : \tau, \varsigma, \kappa \quad \text{if} \quad (\tau, \kappa) \in \Gamma(x)$$

$(const)$ $$\Gamma \triangleright c : \tau, \varsigma, \kappa \quad \text{if} \quad (\tau, \kappa) \in \mathit{IndTypeOf}(c)$$

$(abs)$ $$\frac{\Gamma[x/\tau_1] \triangleright e : \tau_2, \varsigma, \kappa}{\Gamma \triangleright \lambda x.e : \tau_1 \xrightarrow{\varsigma} \tau_2, \varsigma', \kappa}$$

$(app)$ $$\frac{\Gamma \triangleright e_1 : \tau_1 \xrightarrow{\varsigma'} \tau_2, \varsigma, \kappa \quad \Gamma \triangleright e_2 : \tau_1, \varsigma, \kappa}{\Gamma \triangleright e_1\ e_2 : \tau_2, \varsigma, \kappa \cup \{\langle\varsigma, \varsigma'\rangle\}}$$

$(let)$ $$\frac{\Gamma \triangleright e_1 : \tau_1, \varsigma, \kappa_1 \quad (\sigma, \kappa_2) = \mathit{Close}(\tau_1, \Gamma, \varsigma, \kappa_1) \quad \Gamma[x/\sigma] \triangleright e_2 : \tau_2, \varsigma, \kappa_2}{\Gamma \triangleright \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2, \varsigma, \kappa_2}$$

$$\begin{aligned}
\mathit{Close}(\tau, \Gamma, \varsigma, \kappa) &= (\forall\alpha_1 \ldots \alpha_m\varsigma_1 \ldots \varsigma_n.\,\tau\ \mathsf{with}\ \kappa', \kappa \setminus \kappa')\\
\text{where } \{\alpha_1, \ldots, \alpha_m, \varsigma_1, \ldots, \varsigma_n\} &= IFV(\tau, \kappa) \setminus (IFV(\Gamma, \kappa) \cup IFV(\varsigma, \kappa))\\
\kappa' &= \{\langle\varsigma, \nu\rangle \mid \langle\varsigma, \nu\rangle \in \kappa,\ \varsigma \in \{\varsigma_1, \ldots, \varsigma_n\}\}
\end{aligned}$$

Figure 3: Indirect type assignment

$$\begin{aligned}
IFV(\iota, \kappa) &= \emptyset\\
IFV(\alpha, \kappa) &= \{\alpha\}\\
IFV(\tau\ \mathit{ref}, \kappa) &= IFV(\tau, \kappa)\\
IFV(\tau_1 \xrightarrow{\varsigma} \tau_2, \kappa) &= IFV(\tau_1, \kappa) \cup IFV(\tau_2, \kappa) \cup IFV(\varsigma, \kappa)\\
IFV(\varsigma, \kappa) &= \{\varsigma\} \cup \textstyle\bigcup_{\langle\varsigma,\nu\rangle\in\kappa} IFV(\nu, \{\langle\varsigma', \nu'\rangle \mid \langle\varsigma', \nu'\rangle \in \kappa,\ \varsigma' \neq \varsigma\})\\
IFV(\forall\nu_1 \ldots \nu_n.\,\tau\ \mathsf{with}\ \kappa', \kappa) &= IFV(\tau, \kappa' \cup \kappa) \setminus \{\nu_1, \ldots, \nu_n\}\\
IFV(\Gamma, \kappa) &= \textstyle\bigcup_{x\in\mathrm{Dom}(\Gamma)} IFV(\Gamma(x), \kappa).
\end{aligned}$$

Figure 4: Free variables for the indirect system

**Theorem 4.1 (Indirect to Direct)** *If* $\Gamma \triangleright e : \tau, \varsigma, \kappa$ *then* $\mathcal{D}[\![\Gamma]\!]\kappa \vdash e : \mathcal{D}[\![\tau]\!]\kappa, \mathcal{D}[\![\varsigma]\!]\kappa$.

The proof of this theorem proceeds by induction on the derivation of $\Gamma \triangleright e : \tau, \varsigma, \kappa$. For this induction, it is critical that the translation $\mathcal{D}$ preserve free type and effect variables:

$$\begin{aligned}
IFV(\tau, \kappa) = FV(\mathcal{D}[\![\tau]\!]\kappa); &\quad IFV(\sigma, \kappa) = FV(\mathcal{D}[\![\sigma]\!]\kappa);\\
IFV(\varsigma, \kappa) = FV(\mathcal{D}[\![\varsigma]\!]\kappa); &\quad IFV(\Gamma, \kappa) = FV(\mathcal{D}[\![\Gamma]\!]\kappa).
\end{aligned}$$

For the translation from the direct system to the indirect system, Figure 6 defines the translation $\mathcal{I}$ from direct types to indirect types. The function $\mathcal{F}$ is any one-to-one function from effects to effect variables such that its range is fresh (no element appears in any derivation). $\Sigma$ represents sets of effect variables. Let $\sqsubseteq$ be the relation between constraints such that $\kappa_1 \sqsubseteq \kappa_2$ iff:

$$\mathrm{Dom}(\kappa_1) \subseteq \mathrm{Dom}(\kappa_2) \text{ and } IFV(\varsigma, \kappa_1) = IFV(\varsigma, \kappa_2) \text{ for all } \varsigma \in \mathrm{Dom}(\kappa_1).$$

$$
\begin{array}{rcl}
\mathcal{D}[\![\iota]\!]\kappa & = & \iota \\
\mathcal{D}[\![\alpha]\!]\kappa & = & \alpha \\
\mathcal{D}[\![\tau\ ref]\!]\kappa & = & (\mathcal{D}[\![\tau]\!]\kappa)\ ref \\
\mathcal{D}[\![\tau_1 \xrightarrow{\varsigma} \tau_2]\!]\kappa & = & \text{let } \Delta = \mathcal{D}[\![\varsigma]\!]\kappa,\ \tau_1' = \mathcal{D}[\![\tau_1]\!]\kappa,\ \tau_2' = \mathcal{D}[\![\tau_2]\!]\kappa \text{ in } \tau_1' \xrightarrow{\Delta} \tau_2' \\
\mathcal{D}[\![\varsigma]\!]\kappa & = & IFV(\varsigma, \kappa) \\
\mathcal{D}[\![\forall \nu_1 \ldots \nu_n . \tau \text{ with } \kappa']\!]\kappa & = & \forall \nu_1 \ldots \nu_n . \mathcal{D}[\![\tau]\!](\kappa \cup \kappa') \qquad (\nu_1, \ldots, \nu_n \notin \kappa) \\
\mathcal{D}[\![\Gamma]\!]\kappa & = & \{x \mapsto \mathcal{D}[\![\Gamma(x)]\!]\kappa \mid x \in \mathrm{Dom}(\Gamma)\}
\end{array}
$$

Figure 5: Translation from indirect types to direct types

$$
\begin{array}{rcl}
\mathcal{I}[\![\iota]\!] & = & (\iota, \emptyset, \emptyset) \\
\mathcal{I}[\![\alpha]\!] & = & (\alpha, \emptyset, \emptyset) \\
\mathcal{I}[\![\tau\ ref]\!] & = & \text{let } (\tau', \kappa, \Sigma) = \mathcal{I}[\![\tau]\!] \text{ in } (\tau'\ ref, \kappa, \Sigma) \\
\mathcal{I}[\![\tau_1 \xrightarrow{\Delta} \tau_2]\!] & = & \text{let } (\tau_1', \kappa_1, \Sigma_1) = \mathcal{I}[\![\tau_1]\!],\ \ (\tau_2', \kappa_2, \Sigma_2) = \mathcal{I}[\![\tau_2]\!],\ \ (\varsigma, \kappa_3) = \mathcal{I}[\![\Delta]\!] \\
 & & \text{in } (\tau_1' \xrightarrow{\varsigma} \tau_2', \kappa_1 \cup \kappa_2 \cup \kappa_3, \Sigma_1 \cup \Sigma_2 \cup \{\varsigma\}) \\
\mathcal{I}[\![\Delta]\!] & = & \text{let } \varsigma = \mathcal{F}(\Delta) \text{ in } (\varsigma, \{\langle \varsigma, \nu \rangle \mid \nu \in \Delta\} \cup \{\langle \varsigma, \varsigma \rangle\}) \\
\mathcal{I}[\![\forall \nu_1 \ldots \nu_n . \tau]\!] & = & \text{let } (\tau', \kappa, \Sigma) = \mathcal{I}[\![\tau]\!],\ \ \kappa' = \{\langle \varsigma_i, \tau_i \rangle \in \kappa \mid \varsigma_i \in \{\nu_1, \ldots, \nu_n\} \cup \Sigma\} \\
 & & \text{in } (\forall \nu_1 \ldots \nu_n \Sigma . \tau' \text{ with } \kappa', \kappa \setminus \kappa') \\
\mathcal{I}[\![\Gamma]\!] & = & \text{let } (\sigma_i, \kappa_i) = \mathcal{I}[\![\Gamma(x_i)]\!] \text{ for } x_i \in \mathrm{Dom}(\Gamma) \\
 & & \text{in } (\{x_i \mapsto \sigma_i\}, \bigcup_i \kappa_i)
\end{array}
$$

Figure 6: Translation from direct types to indirect types

If $e$ has a derivation in the direct system, then $e$ has a derivation in the indirect system.

**Theorem 4.2 (Direct to Indirect)** *If* $\Gamma \vdash e : \tau, \Delta$ *and* $(\overline{\tau}, \kappa_\tau, \Sigma) = \mathcal{I}[\![\tau]\!]$ *and* $(\overline{\varsigma}, \kappa_\Delta) = \mathcal{I}[\![\Delta]\!]$ *and* $(\overline{\Gamma}, \kappa_\Gamma) = \mathcal{I}[\![\Gamma]\!]$ *then* $\overline{\Gamma} \rhd e : \overline{\tau}, \overline{\varsigma}, \kappa$ *where* $\kappa_\Gamma, \kappa_\tau, \kappa_\Delta \sqsubseteq \kappa$.

Again, the proof relies on the translation $\mathcal{I}$ preserving free type and effect variables:

$$
\begin{array}{ll}
\text{if } \mathcal{I}[\![\tau]\!] = (\overline{\tau}, \kappa, \Sigma) & \text{then } IFV(\overline{\tau}, \kappa) = FV(\tau) \cup \Sigma; \\
\text{if } \mathcal{I}[\![\Delta]\!] = (\overline{\varsigma}, \kappa) & \text{then } IFV(\overline{\varsigma}, \kappa) = FV(\Delta) \cup \{\overline{\varsigma}\}; \\
\text{if } \mathcal{I}[\![\sigma]\!] = (\overline{\sigma}, \kappa) & \text{then } IFV(\overline{\sigma}, \kappa) = FV(\sigma); \\
\text{if } \mathcal{I}[\![\Gamma]\!] = (\overline{\Gamma}, \kappa) & \text{then } IFV(\overline{\Gamma}, \kappa) = FV(\Gamma).
\end{array}
$$

## 4.3 Algorithm

Obtaining an algorithm from the indirect system is straightforward [10]. Given an expression, a type environment, and a constraint, algorithm $W$ in Figure 7 computes a type, a substitution to be applied to the type environment, an effect variable, and a new constraint. The function *unify* performs ordinary unification, returning a substitution that unifies its arguments; $I$ is the identity substitution. Soundness and completeness theorems may be demonstrated to establish the correspondence of the algorithm to the indirect system as usual [2, 17].

$$
\begin{array}{ll}
W(x,\,\Gamma,\,\kappa) & = \text{if } x \notin \mathrm{Dom}(\Gamma) \text{ then } \textbf{fail} \text{ (expression is not closed)} \\
& \quad \text{let } \varsigma, \alpha'_1, \ldots, \alpha'_m, \varsigma'_1, \ldots, \varsigma'_n \text{ be fresh} \\
& \qquad \forall \alpha_1 \ldots \alpha_m \varsigma_1 \ldots \varsigma_n . \tau \ \mathsf{with}\ \kappa' = \Gamma(x) \\
& \qquad S = \{\alpha_i \mapsto \alpha'_i, \varsigma_j \mapsto \varsigma'_j\} \\
& \quad \text{in } (S\tau,\, I,\, \varsigma,\, \kappa \cup S\kappa') \\
W(c,\,\Gamma,\,\kappa) & = \text{let } \varsigma, \alpha'_1, \ldots, \alpha'_m, \varsigma'_1, \ldots, \varsigma'_n \text{ be fresh} \\
& \qquad \forall \alpha_1 \ldots \alpha_m \varsigma_1 \ldots \varsigma_n . \tau \ \mathsf{with}\ \kappa' = \mathit{IndTypeOf}(c) \\
& \qquad S = \{\alpha_i \mapsto \alpha'_i, \varsigma_j \mapsto \varsigma'_j\} \\
& \quad \text{in } (S\tau,\, I,\, \varsigma,\, \kappa \cup S\kappa') \\
W(\lambda x.e,\,\Gamma,\,\kappa) & = \text{let } \alpha, \varsigma \text{ be fresh} \\
& \qquad (\tau, S, \varsigma', \kappa') = W(e,\, \Gamma[x/\alpha],\, \kappa) \\
& \quad \text{in } (S\alpha \xrightarrow{\varsigma} \tau,\, S,\, \varsigma,\, \kappa') \\
W((e_1\ e_2),\,\Gamma,\,\kappa) & = \text{let } \alpha, \varsigma \text{ be fresh} \\
& \qquad (\tau_1, S_1, \varsigma_1, \kappa_1) = W(e_1, \Gamma, \kappa) \\
& \qquad (\tau_2, S_2, \varsigma_2, \kappa_2) = W(e_2, S_1\Gamma, \kappa_1) \\
& \qquad\qquad\qquad S_3 = \mathit{unify}(S_2\tau_1, \tau_2 \xrightarrow{\varsigma} \alpha) \quad \text{(may } \textbf{fail}\text{)} \\
& \qquad\qquad\qquad S_4 = \mathit{unify}(S_3S_2\varsigma_1, S_3\varsigma_2) \quad \text{(will not } \textbf{fail}\text{)} \\
& \quad \text{in } (S_4S_3\alpha,\, S_4 \circ S_3 \circ S_2 \circ S_1,\, S_4S_3\varsigma_2,\, S_4S_3(\kappa_2 \cup \{\langle \varsigma, \varsigma_2 \rangle\})) \\
W(\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2,\,\Gamma,\,\kappa) & = \text{let } (\tau_1, S_1, \varsigma_1, \kappa_1) = W(e_1, \Gamma, \kappa)) \\
& \qquad\qquad\quad (\sigma, \kappa_2) = \mathit{Close}(\tau_1, S_1\Gamma, \varsigma_1, \kappa_1) \\
& \qquad (\tau_2, S_2, \varsigma_2, \kappa_3) = W(e_2, (S_1\Gamma)[x/\sigma], \kappa_2) \\
& \qquad\qquad\qquad S_3 = \mathit{unify}(S_2\varsigma_1, \varsigma_2) \quad \text{(will not } \textbf{fail}\text{)} \\
& \quad \text{in } (S_3\tau_2,\, S_3 \circ S_2 \circ S_1,\, S_3\varsigma_2,\, S_3\kappa_3)
\end{array}
$$

Figure 7: Type Inference Algorithm

Our algorithm is similar to Milner's algorithm $\mathcal{W}$ [12] for a functional language. In our algorithm, the case for applications contains an additional invocation of *unify*, but this second invocation always unifies two effect variables, and is very cheap. The main difference is that our algorithm must maintain a set of constraints that grows linearly with the number of applications and that must be searched in typing a let-expression. The practical performance of our algorithm depends on the efficiency of constraint handling; the techniques for manipulating constraints discussed by Leroy and Weis [10] should be applicable to our algorithm.

# 5 Comparison with Other Systems

There are several other systems for typing references in the presence of polymorphism; O'Toole [15] presents detailed comparisons between four of them. Ideally, we would like a simple, intuitive system that is at least as powerful as any of the others, but unfortunately, there are no systems that meet this goal. While we believe our system is relatively simple and natural, it is incomparable to the other systems: there are expressions typable in one system but not the other, and vice versa. Hence comparisons between systems must be pragmatic.

**Tofte** [18] A system proposed by Tofte and adopted for STANDARD ML [13, 14] has two kinds of type variables: *imperative* variables and *applicative* variables. Types are classified accordingly: imperative types may only contain imperative type variables; applicative types may contain either. The ref operator may only be applied to values of imperative type. In typing the expression let $x = e_1$ in $e_2$, imperative variables in the type of $e_1$ can be generalized only if $e_1$ is *non-expansive*, that is, if $e_1$ has a certain syntactic shape which guarantees that its evaluation does not allocate any references (in STANDARD ML, $e_1$ must be a variable, constant, or $\lambda$-expression). We may think of ref as contaminating all type variables in the type of its argument. Generalizing contaminated (imperative) type variables may result in generalizing the type of a value in a reference cell; hence, generalization of contaminated types must be restricted. For example, the expression $\lambda$x. ! (ref x) has type $\alpha^* \to \alpha^*$, where the superscript $^*$ indicates that $\alpha^*$ is an imperative type variable. However, $\alpha^*$ can be generalized in the expression:

$$\text{let i} = \lambda \text{x.}\,!\,(\text{ref x}) \ \text{ in } \ \ldots$$

as $\lambda$x. ! (ref x) is non-expansive, and hence does not allocate any references.

A drawback to this system is that partial applications of imperative functions cannot be polymorphic. For example, the function imap (from Section 1.4) has type scheme:

$$\forall \alpha^* \beta^*. (\alpha^* \to \beta^*) \to \alpha^* \ list \to \beta^* \ list.$$

When only partially applied, as in:

$$\text{let i} = \text{imap } (\lambda \text{x.x}) \ \text{ in } \ \ldots$$

i cannot be used polymorphically. The expression imap ($\lambda$x.x) has type $\alpha^*$ *list* $\to \alpha^*$ *list* and is expansive, hence $\alpha^*$ cannot be generalized. However, in the expression:

$$\text{let i} = \lambda \text{z. imap } (\lambda \text{x.x})\text{ z} \ \text{ in } \ \ldots$$

$\alpha^*$ can be generalized since the bound expression is non-expansive.

**MacQueen** [16] A system proposed by MacQueen and implemented by STANDARD ML OF NEW JERSEY attempts to address the curried application problem described above by recognizing how many arguments a function must be applied to before it creates any references. In this system, imap has type scheme $(\alpha^2 \to \beta^2) \to \alpha^2$ *list* $\to$ $\beta^2$ *list*. The result of the application imap ($\lambda$x.x) has type $\alpha^1$ *list* $\to \beta^1$ *list* and can be used polymorphically. The superscript indicates the number of times the function must be applied before a cell is allocated whose type involves that type variable; applications decrease the superscript. Variables with superscript 0 may not be generalized.

Although the system addresses common uses of currying, such as the partial application of imap above, there are cases in which it fails. The system assigns a polymorphic type to f in only the second of the following two expressions:

$$\text{let f} = \text{map ref} \ \text{ in } \ \ldots$$
$$\text{let f} = \lambda \text{z. map ref z} \ \text{ in } \ \ldots$$

The system is unable to recognize that map ref does not allocate any references until it is further applied, since the type scheme for the non-imperative function map : $\forall \alpha\beta. (\alpha \to$

$\beta) \rightarrow \alpha\ list \rightarrow \beta\ list$ does not indicate that the functional argument to map is invoked only after a second argument is provided. Furthermore, the only formal description of the system is the NEW JERSEY compiler's source code; no inference rule formulation exists. The system is believed to be strictly stronger than Tofte's system, *i.e.*, any expression typable by Tofte's system is typable by MacQueen's system [18].

In contrast to the above two systems, our system handles partial applications properly. For example, our system assigns a polymorphic type to f in both of the following expressions:

$$\begin{array}{l}\mathsf{let\ f} = \mathsf{imap}\ \lambda \mathsf{x.x}\ \ \mathsf{in}\ \ \ldots \\ \mathsf{let\ f} = \mathsf{map\ ref}\ \ \mathsf{in}\ \ \ldots\end{array}$$

Our system recognizes when allocations occur with greater precision by recording type effects above function type arrows.

**Damas** [3] Damas proposed one of the earliest systems for typing references. This system appears to lie between Tofte's and MacQueen's systems in power [18], and although the system has little advantage over Tofte's system, it did inspire our work. The system infers information similar to type effects, but records information on only the outermost arrow of a function type; hence, the determination of when allocation occurs is fairly imprecise. Damas was apparently aware of this problem, as he states that "the inclusion [of effects among function types], which would be the natural thing to do ... would preclude the extension of the type assignment algorithm ... to this extended type inference system" [3: p. 90]. Our insight was to include effects in function types and use the constraint manipulation ideas of Leroy and Weis's system to construct an inference system with a corresponding algorithm.

**Leroy & Weis** [10] Leroy and Weis propose a closure typing system based on the observation that it is only necessary to prohibit generalization of type variables appearing in the types of cells reachable after the bound expression has been evaluated (*i.e.*, cells that would not be reclaimed by a garbage collection at this point). As cells may be reachable through the free identifiers of closures, the system records the types of the free identifiers of a function in the function's type. This system has the advantage that it can assign a fully polymorphic type to some functions that make purely local use of a reference cell, such as imap: the system is able to recognize when effects can be *masked*. However, it fails to type some purely functional expressions that are typable in the functional sublanguage, such as the following:

$$\begin{array}{l}\lambda \mathsf{z}.\ \mathsf{let\ id} = \lambda \mathsf{x}.\ ((\mathsf{if\ true\ then\ z\ else}\ \lambda \mathsf{y}.(\mathsf{x};\ \mathsf{y}));\ \mathsf{x}) \\ \quad \mathsf{in\ id\ 1;\ \ id\ true}\end{array}$$

The problem is that by introducing the types of free variables into function types, additional type variables may be introduced into the type environment. In the above example, the if-expression forces z and $\lambda$y.(x; y) to have the same type. Suppose that x has type $\alpha$ and y has type $\beta$. Since x is free in $\lambda$y.(x; y), this expression has type $\beta \xrightarrow{\alpha} \beta$, hence z has type $\beta \xrightarrow{\alpha} \beta$. As $\alpha$ appears in the type of z, $\alpha$ is free in the type environment of the let-expression, and cannot be generalized. In the ordinary Hindley/Milner system, $\lambda$y.(x; y) and z have type $\beta \rightarrow \beta$, and $\alpha$ is not free in the type environment. All of the other systems, including ours, will assign types to such functional expressions.

**Talpin & Jouvelot** [17] Talpin and Jouvelot present an ambitious system that infers types, effects, and regions for expressions in a manner similar to our indirect system.[3] A reference cell containing a value of type $\tau$ has type $ref_\rho \tau$; two reference cells share the same region $\rho$ if they may alias. Regions provide finer grained information about allocation behavior than our type effects, and the information about potential aliasing is used to mask local effects. The system includes subsumption on effects, and infers not only allocation effects but also read and write effects that may be useful for compiler optimizations. We conjecture that Talpin and Jouvelot's system is strictly more powerful than our system.

### 5.1 Effect Masking

A shortcoming of our system is its inability to recognize when a reference is only used locally and to mask the allocation. When local effects are masked, the function imap has the same type as its functional counterpart map. However, just as precisely determining when allocations occur is impossible, so too is precise effect masking. The following expression is rejected by both Leroy and Weis's system and Talpin and Jouvelot's system,[4] although these systems incorporate effect masking:

$$\begin{array}{l}\lambda\mathsf{z}.\ \mathsf{let\ id} = \lambda\mathsf{x}.\ ((\mathsf{if\ true\ then\ z\ else}\ \lambda\mathsf{y}.\ (\underline{\underline{\mathsf{ref\ x}}};\ \mathsf{y}));\ \mathsf{x}) \\ \quad \mathsf{in\ id\ 1;\ \ id\ true}\end{array}$$

This example is also rejected by our system, but Tofte's, Damas's, and MacQueen's systems accept it.

Abstractly, adding effect masking to our system is easy:

$$(mask) \qquad \frac{\Gamma \vdash e : \tau, \Delta_1 \quad \Delta_2 \text{ is maskable in } e}{\Gamma \vdash e : \tau, \Delta_1 \setminus \Delta_2}$$

Selecting criteria to determine which effects are maskable is more difficult. Data flow analysis and compile time garbage collection techniques [4] might be used to discover references that are not reachable after the bound expression has been evaluated. Even the simplest local flow analysis algorithms would address examples such as the one above.

Effect masking raises the issue of just how far to push static type systems for references. The more powerful the type system is, the more complex its description becomes. If flow analysis is used to mask effects, then a precise description of the flow analysis algorithm is an essential part of the description of the type system. Without a precise description of the type system, the set of well-typed programs is unspecified, and the programmer is left at the mercy of the implementor.

## Acknowledgements

I am indebted to Jean-Pierre Talpin and Pierre Jouvelot for many helpful discussions on direct and indirect formulations of effect inference, and to Matthias Felleisen for comments on drafts of this paper.

[3] It should be possible to reformulate this system in a direct fashion.

[4] Talpin and Jouvelot now have a refined system that can type this example [personal communication, November '91].

## References


[1] Barendregt, H. P. *The Lambda Calculus: Its Syntax and Semantics*, revised ed., vol. 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, 1984.

[2] Damas, L., and Milner, R. Principal type schemes for functional programs. *Proceedings of the 9th Annual Symposium on Principles of Programming Languages* (January 1982), 207–212.

[3] Damas, L. M. M. *Type Assignment in Programming Languages.* PhD thesis, University of Edinburgh, 1985.

[4] Deutsch, A. On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications. *Proceedings of the 17th Annual Symposium on Principles of Programming Languages* (January 1990), 157–168.

[5] Felleisen, M., and Friedman, D. P. A syntactic theory of sequential state. *Theoretical Computer Science 69*, 3 (1989), 243–287. Preliminary version in: *Proceedings of the 14th Annual Symposium on Principles of Programming Languages*, 1987, 314-325.

[6] Felleisen, M., Friedman, D. P., Kohlbecker, E. E., and Duba, B. A syntactic theory of sequential control. *Theoretical Computer Science 52*, 3 (1987), 205–237. Preliminary version in: *Proceedings of the Symposium on Logic in Computer Science*, 1986, 131–141.

[7] Felleisen, M., and Hieb, R. The revised report on the syntactic theories of sequential control and state. Tech. Rep. TR-100, Rice University, June 1989. To appear in: *Theoretical Computer Science.*

[8] Hindley, R. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society 146* (December 1969), 29–60.

[9] Jouvelot, P., and Gifford, D. K. Algebraic reconstruction of types and effects. *Proceedings of the 18th Annual Symposium on Principles of Programming Languages* (January 1991), 303–310.

[10] Leroy, X., and Weis, P. Polymorphic type inference and assignment. *Proceedings of the 18th Annual Symposium on Principles of Programming Languages* (January 1991), 291–302.

[11] Lucassen, J. M., and Gifford, D. K. Polymorphic effect systems. *Proceedings of the 15th Annual Symposium on Principles of Programming Languages* (January 1988), 47–57.

[12] Milner, R. A theory of type polymorphism in programming. *Journal of Computer and System Sciences 17* (1978), 348–375.

[13] Milner, R., and Tofte, M. *Commentary on Standard ML.* MIT Press, Cambridge, Massachusetts, 1991.

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

[15] O'Toole Jr., J. W. Type abstraction rules for references: A comparison of four which have achieved notoriety. Unpublished, 1990.

[16] Standard ML of New Jersey release notes (version 0.75). AT&T Bell Laboratories, November 1991.

[17] Talpin, J.-P., and Jouvelot, P. The type and effect discipline. Tech. Rep. EMP-CRI A/206, Ecole des Mines de Paris, July 1991.

[18] Tofte, M. Type inference for polymorphic references. *Information and Computation 89*, 1 (November 1990), 1–34.

[19] Wright, A. K., and Felleisen, M. A syntactic approach to type soundness. Tech. Rep. 91-160, Rice University, April 1991. To appear in: *Information and Computation*.