Polymorphic type inference and assignment

Xavier Leroy, Pierre Weis

Proc. 18th Symp. Principles of Programming Languages, 1991, pages 291–302.

Polymorphic type inference and assignment

Xavier Leroy*
Ecole Normale Supérieure

Pierre Weis*
INRIA Rocquencourt

Abstract

We present a new approach to the polymorphic typing of data accepting in-place modification in ML-like languages. This approach is based on restrictions over type generalization, and a refined typing of functions. The type system given here leads to a better integration of imperative programming style with the purely applicative kernel of ML. In particular, generic functions that allocate mutable data can safely be given fully polymorphic types. We show the soundness of this type system, and give a type reconstruction algorithm.

1 Introduction

Polymorphic type disciplines originate in the study of λ-calculus and its connections to constructive logic [7, 14], so it is no surprise it fits very nicely within purely applicative languages, without side effects. However, polymorphism becomes problematic when we move toward conventional imperative languages (Algol, Pascal), and allow physical modification of data structures. The problem appeared at an early stage of the design of ML [8, p. 52], when assignment operators were provided for the primitive data types of references and vectors. Consider the following example, in ML[1]:

```
let r = ref [] in
    r := [1];
    if head(!r) then ... else ...
```

If we naively give type $\forall\alpha.\,\alpha\ list\ ref$ to the reference r, we can first use it with type *int list ref*, and store in it the list with 1 as single element — an *int list*, indeed. Given its type, we can also consider r as having type *bool list ref*, hence *head*(!r) has type *bool*, and the `if` statement is well-typed. However, *head*(!r) evaluates to 1, which is not a valid boolean. This example shows that physical modification of data compromises type safety, since it can invalidate static typing assumptions.

As demonstrated here, the use of polymorphic mutable data (that is, data structures that can be modified in place) must be restricted. An obvious way to tackle this problem, used in early implementations of ML [3], is to require all such data to have monomorphic, statically-known types. This restriction trivially solves the problem, but it also makes it impossible to write polymorphic functions that create mutable values. This fact has unfortunate consequences.

A first drawback is that it is not possible to provide generic, efficient implementations of most data structures (vectors, hash tables, graphs, B-trees, ...), as they require physical modification. Even a trivial function such as taking a vector of an arbitrary type and returning a copy of it is not well-typed with the policy above, since it creates a vector with a statically unknown type.

Another drawback is that polymorphic mutable values are prohibited even if they are not returned, but used for internal computation only. As a consequence, most generic functions cannot be written in an imperative style, with references holding intermediate results. Consider the familiar *map* functional:

```
let rec applicative_map f l =
    if null l then [] else
        f (head l) :: applicative_map f (tail l)
```

Here is an alternate implementation of *map* in imperative style:

```
let imperative_map f l =
    let argument = ref l and result = ref [] in
        while not (null !argument) do
            result := f(head !argument) :: !result;
            argument := tail !argument
        done;
        reverse !result
```

Some ML type systems reject *imperative_map* as ill-typed. Others give it a less general type than its purely applicative version. In any case, the imperative version cannot be substituted for the applicative one,

*Authors' address: B.P.105, 78153 Le Chesnay, France. E-mail: xleroy@margaux.inria.fr, weis@margaux.inria.fr.

[1]Survival kit for the reader unfamiliar with ML: [] is the (polymorphic) empty list, $[a_1; \ldots; a_n]$ the list with elements $a_1 \ldots a_n$. `ref` x allocates a new reference (indirection cell), initialized to x. $r := x$ updates the contents of reference r by x. !r returns the current contents of reference r.

even though they have exactly the same semantics. As demonstrated here, the programming style (imperative vs. applicative) interferes with the type specifications. This clearly goes against modular programming.

Some enhancements to the ML type system have been proposed [4, 16, 17, 1], that weaken the restrictions over polymorphic mutable data. Standard ML [11] incorporates some of these ideas. These enhanced type systems make it possible to define many useful generic functions over mutable data structures, such as the function that copies a vector. However, these systems are still not powerful enough: they fail to infer the most general type for the *imperative_map* example above; and they do not work well in conjunction with higher-order functions. Because of these shortcomings, Standard ML does not provide adequate support for the imperative programming style. This is a major weakness for a general-purpose programming language.

In this paper, we present a new way to typecheck mutable data within a polymorphic type discipline. Our type system is a simple extension of the one of ML. It permits type reconstruction and possesses the principal type property. It requires minimal modifications to the ML type algebra. Yet it definitely improves the support for imperative programming in ML. It allows generic functions over mutable data structures to have fully polymorphic types. It is powerful enough to assign to a function written in imperative style the same type as its purely applicative counterpart. For example, in this system, the imperative implementation of *map* cannot be distinguished from the usual, applicative one, as far as types are concerned.

The remainder of the paper is organized as follows. In section 2, we introduce informally our type system, and show the need for a more precise typechecking of functions. Section 3 formalizes the ideas above. We state the typechecking rules, show their soundness, and give a type reconstruction algorithm. Section 4 briefly compares our approach with previous ones. We give a few concluding remarks in section 5.

2 Informal presentation

In this section, we informally introduce our typing discipline for mutable data, focusing on references to be more specific. Unlike the Standard ML approach, we do not attempt to detect the creation of polymorphic references. What we prohibit is *the use of a reference with two different types*. The only way to use a value with several types is to generalize its type first, that is, to universally quantify over some of its type variables. (In ML, the only construct that generalizes types is the `let` binding.) Hence, what we restrict is type generalization: we never generalize a type variable that may be free in the type of a reference. Such a type variable is said to be *dangerous*. Not generalizing dangerous type variables ensures that a mutable value always possesses a monotype.

It remains to detect dangerous type variables at generalization-time. Though this suggests a complex and expensive static analysis, this turns out not to be the case: dangerous variables can be determined by mere examination of the type being generalized, as we shall now illustrate.

2.1 Datatypes

Consider the example given in introduction:

```
let r = ref [] in
    r := [1];
    if head(!r) then ... else ...
```

The expression `ref []` has type $A = \alpha\ list\ ref$. This type is a *ref* type, and α is free in it, hence α is dangerous in A. The `let` construct does not generalize α, hence α gets instantiated to *int* when typing $r := [1]$, and typing `if` $head(!r)$ `...` leads to a type clash.

References may be embedded into more complex data structures (pairs, lists, concrete datatypes). Fortunately, the type of a data structure contains enough information to retrieve the types of the components of the structure. For instance, a pair with type $A \times B$ contains one value of type A and one value of type B. Therefore, any variable which is dangerous in A or in B is also dangerous in $A \times B$. For instance, in

```
let r = ([], ref []) in e
```

the expression `([], ref [])` has type $\alpha\ list \times \beta\ list\ ref$, where β is dangerous but not α. Hence in e, variable r has type $\forall\alpha.\, \alpha\ list \times \beta\ list\ ref$.

The treatment of user-defined datatypes is similar. Parameterless datatypes cannot contain polymorphic data, so there is no dangerous variable in them. Parameterized datatypes come in two flavors: those which introduce new *ref* types (such as `type` α *foo* $=$ A | B `of` α *ref*), and those which don't. The former are treated like *ref* types: all variables free in their parameter(s) are considered dangerous. The latter are treated like product types: their dangerous variables are the dangerous variables of their parameter(s).

2.2 Functions

Function types are treated specially. The reason is that a value with type $A \to B$ does not contain a value of

type A, nor a value of type B, in contrast with regular datatypes such as $A \times B$ or $A\ list$. Hence it seems there are no dangerous variables in type $A \to B$, even if A or B contain dangerous variables themselves. For instance, the function

$$\texttt{let}\ make\_ref = \texttt{function}\ x \to\ \texttt{ref}\ x$$

has type $\alpha \to \alpha\ ref$, and α is not dangerous in it, so it is fully polymorphic. It is actually harmless by itself. What's harmful is to apply *make_ref* to a polymorphic argument such as [], bind the result with a `let` construct, and use it with two different types. But this is not possible in the proposed type system, since *make_ref* [] has type $\beta\ list\ ref$, and this type will not be generalized, as β is dangerous in it. In our approach, generic functions that create and return mutable objects are given very general types. Type safety is ensured by controlling what can be done with the result of their application, as described above.

The analysis above is based solely on the type of the function result. Hence, the usage of a polymorphic function is unrestricted if the function does not return any references, as witnessed by the absence of *ref* types in its codomain type. This holds even if the function allocates references with statically unknown types for internal purposes, but does not return them. For instance, this is the case for the *imperative_map* functional given in the introduction: it is given type $\forall\alpha,\beta.\,(\alpha \to \beta) \to \alpha\ list \to \beta\ list$, the very same type as its purely applicative counterpart *applicative_map*. The *imperative_map* functional can be substituted for *applicative_map* in any context. In particular, it can be applied to highly polymorphic arguments: the expression

$$imperative\_map\ (\texttt{function}\ x \to\ x)\ []$$

is well-typed, and returns the fully polymorphic empty list, even though two references were created with type $\alpha\ list\ ref$. The type system guarantees that these references are used consistently inside the function, and are not exported outside.

Another strength of this type-based analysis is its good handling of higher-order functions and partial applications. Consider the partial application of *applicative_map* to *make_ref*: our type system correctly recognize it as harmless, and gives it the fully polymorphic type $\forall\alpha.\,\alpha\ list \to \alpha\ ref\ list$. This is not the case for ML type systems that attempt to control the creation of references (see section 4).

2.3 Functions with free variables

The careful reader may have noticed a flaw in the discussion above: we have neglected the fact that a function may possess free variables. Suppose that a function f has a free variable r which is bound to a reference outside the function. Function f can access and update the reference, yet the type of r does not necessarily appear in the type of f. A classic example is the functional presentation of references as a pair of functions, one for access, the other for update:

$$\begin{array}{l}\texttt{let}\ functional\_ref\ x\ =\\ \quad\texttt{let}\ r = \texttt{ref}\ x\ \texttt{in}\\ \quad\quad(\texttt{function}\ ()\to\ !r),\\ \quad\quad(\texttt{function}\ z\to\ r := z)\end{array}$$

The expression *functional_ref* [] has type $(unit \to \alpha\ list) \times (\alpha\ list \to unit)$, where no type variable is dangerous, so it is fully polymorphic, yet it breaks type safety just as `ref` [] does. The problem is that the types of the free variables of a function do not appear in the type of the function. In this context, it is useful to think of functions as closures. Closures, in contrast with any other data structure, are not adequately described by their (functional) type: we do not know anything about the types of the values contained in the environment part of a closure.

Our solution is to keep track of what is inside a closure. We associate with any function type a set of types, the types of all variables free in the function. We could put this set of types as a third argument to the arrow type constructor. For technical reasons, we find it more convenient to add an extra level of indirection in recording this set of types. Therefore, each function type is adorned with a *label*. Labels are written L, M, and labeled function types are written $A \xrightarrow{L} B$. Separately, we record *constraints* on those labels. Constraints have the format $A \triangleright L$, meaning that objects of type A are allowed to occur in environments of type L. An environment of type L is not required to contain a value of type A. It is not allowed to contain a value of a type B unless $B \triangleright L$ is one of the recorded constraints.

This way, function type labels allow typings of the environment parts of closures. For instance, *functional_ref* now has type:

$$\begin{array}{c}\alpha \xrightarrow{L} (unit \xrightarrow{M} \alpha) \times (\alpha \xrightarrow{N} unit)\\ \texttt{with}\ \alpha\ ref \triangleright M,\ \alpha\ ref \triangleright N\end{array}$$

hence *functional_ref* [] has type:

$$\begin{array}{c}(unit \xrightarrow{M} \beta\ list) \times (\beta\ list \xrightarrow{N} unit)\\ \texttt{with}\ \beta\ list\ ref \triangleright M,\ \beta\ list\ ref \triangleright N\end{array}$$

thus revealing the presence of a polymorphic reference in each function of the pair, and preventing the generalization of β. The rule is that in a functional type

$A \xrightarrow{M} B$, a variable α is dangerous iff there exists a constraint $C \triangleright M$ with α dangerous in C.

This typing of closures gives a precise account of the interleaving of internal computation and parameter passing in (curried) functions, and of the possible data sharing between invocations. For instance, it succeeds in distinguishing the following two functions:

$$\begin{array}{l}\texttt{function}\ (\,) \rightarrow\ \texttt{ref}\ [\,]\ : \\ \quad \mathit{unit} \xrightarrow{L} \alpha\ \mathit{list}\ \mathit{ref} \\ \texttt{let}\ r = \texttt{ref}\ [\,]\ \texttt{in function}\ (\,) \rightarrow\ r\ : \\ \quad \mathit{unit} \xrightarrow{L} \alpha\ \mathit{list}\ \mathit{ref}\ \ \texttt{with}\ \ \alpha\ \mathit{list}\ \mathit{ref}\ \triangleright\ L\end{array}$$

The former is harmless, since it returns a fresh reference each time, so it can be generalized. The latter always return the same reference, which is therefore shared between all calls to the function. It must remain monomorphic, which is indeed the case, since α is a dangerous variable in its type.

Before presenting the type system formally, we now give the main intuitions behind the typing of closures. Constraints are synthesized during the typing of abstractions, as follows: when giving type $A \xrightarrow{M} B$ to the abstraction $e = \texttt{function}\ x \rightarrow \ldots$, for each variable y free in e, we look up the type C_y of y in the typing environment and record the constraint $C_y \triangleright M$.

It is always safe to add new constraints, since the constraints over a label L are intended to give an upper bound for what can go inside closures of type L. (This may lead to less general types, however, since more variables will be declared dangerous.) Unifying two labeled function types $A \xrightarrow{L} B$ and $C \xrightarrow{M} D$ is easy: it suffices to identify both labels L and M, resulting in a single label which bears the previous constraints on M as well as those on L. For instance, assuming $f : \texttt{int} \xrightarrow{L} \texttt{int}$, the expression

$$\begin{array}{l}\texttt{if}\ \ldots\ \texttt{then}\ f\ \texttt{else} \\ \quad \texttt{let}\ z = 1\ \texttt{in function}\ x \rightarrow\ x + z\end{array}$$

has type $\texttt{int} \xrightarrow{L} \texttt{int}$ as well, with the additional constraint that $\texttt{int} \triangleright L$.

Finally, to give the most general type to functionals, we must be able to generalize over labels, in the same way as we generalize over regular type variables. Consider the functional:

$$\texttt{function}\ f \rightarrow\ 2 + (f\ 1)$$

It must be possible to apply it to any function mapping integers to integers, whatever its closure may contain. Yet if we give it the type $(\texttt{int} \xrightarrow{L} \texttt{int}) \xrightarrow{M} \texttt{int}$ without generalizing over L, we could only apply it to functions without free variables, assuming there is no constraint over L in the current environment. Instead, the right typing is $\forall L.\, (\texttt{int} \xrightarrow{L} \texttt{int}) \xrightarrow{M} \texttt{int}$. In more complex situations, the current environment contains constraints over the label to be generalized. These constraints are discharged in the type schema, and reintroduced at specialization time. Type schemas therefore have the format $\forall V_1 \ldots V_n.\, A\ \texttt{with}\ \Gamma$, where the V_i are either labels or type variables, and Γ is a sequence of constraints.

3 Formalization

In this section, we formalize a calculus based on the ideas above.

3.1 Syntax

The language we consider is the core ML language, λ-calculus plus a distinguished `let` construct. We shall assume a built-in *int* type, with integer constants. The store is presented through the type A *ref* of references to a term of type A, and the operations $\texttt{ref}(a)$, to allocate a new reference to term a, $!a$ to get the contents of the reference a, and $a := b$ to update the content of a by b.

We assume given a countable set *Var* of term variables, with typical elements x, y. In the following, i ranges over integers. The syntax of terms, with typical elements a, b, is as follows:

$$\begin{array}{rcl} a & ::= & x \mid i \mid \lambda x.\, a \mid (b\ a) \mid \texttt{let}\ x = a\ \texttt{in}\ b \mid \\ & & \texttt{ref}(a) \mid\ !a \mid a := b \end{array}$$

3.2 Typechecking

Type expressions, with typical elements A, B, have the following syntax:

$$A \quad ::= \quad X \mid \mathit{int} \mid A \xrightarrow{L} B \mid A\ \mathit{ref}$$

In the definition above, X stands for a type variable, ranging over a given countable set *TVar*. Function types $A \xrightarrow{L} B$ are annotated by a label L, taken from a countable set *Lbl*. Labels are distinct from term variables and type variables.

Type schemas, with typical element Σ, are composed of type expressions with some type variables and some labels universally quantified. They also contain a sequence of constraints Γ. Constraints have the format $\Sigma \triangleright L$.

$$\begin{array}{rcl} \Sigma & ::= & A \mid \forall V_1 \ldots V_n.\, A\ \texttt{with}\ \Gamma \\ \Gamma & ::= & \epsilon \mid \Sigma \triangleright L, \Gamma \\ V & ::= & X \mid L \end{array}$$

$$(\textsc{int}) \quad E \vdash i : \mathit{int}\ \texttt{with}\ \Gamma$$

$$(\textsc{varspec}) \quad \frac{E(x) = \forall X_1 \ldots X_n, L_1 \ldots L_m.\, A\ \texttt{with}\ \Gamma}{E \vdash x : A\{X_i \leftarrow B_i, L_j \leftarrow M_j\}\ \texttt{with}\ \Gamma\{X_i \leftarrow B_i, L_j \leftarrow M_j\} \cup \Gamma'}$$

$$(\textsc{fun}) \quad \frac{E[x \leftarrow A] \vdash b : B\ \texttt{with}\ \Gamma \qquad \text{for all } y \text{ free in } \lambda x.\, b,\ (E(y) \triangleright L) \in \Gamma}{E \vdash \lambda x.\, b : A \xrightarrow{L} B\ \texttt{with}\ \Gamma}$$

$$(\textsc{app}) \quad \frac{E \vdash b : A \xrightarrow{L} B\ \texttt{with}\ \Gamma \qquad E \vdash a : A\ \texttt{with}\ \Gamma}{E \vdash (b\ a) : B\ \texttt{with}\ \Gamma}$$

$$(\textsc{letgen}) \quad \frac{E \vdash a : A\ \texttt{with}\ \Gamma \qquad (\Sigma, \Gamma') = \mathit{Gen}(A, E, \Gamma) \qquad E[x \leftarrow \Sigma] \vdash b : B\ \texttt{with}\ \Gamma' \cup \Gamma''}{E \vdash \texttt{let}\ x = a\ \texttt{in}\ b : B\ \texttt{with}\ \Gamma' \cup \Gamma''}$$

$$(\textsc{ref}) \quad \frac{E \vdash a : A\ \texttt{with}\ \Gamma}{E \vdash \texttt{ref}(a) : A\ \mathit{ref}\ \texttt{with}\ \Gamma} \qquad (\textsc{deref}) \quad \frac{E \vdash a : A\ \mathit{ref}\ \texttt{with}\ \Gamma}{E \vdash !a : A\ \texttt{with}\ \Gamma}$$

$$(\textsc{assign}) \quad \frac{E \vdash a : A\ \mathit{ref}\ \texttt{with}\ \Gamma \qquad E \vdash b : A\ \texttt{with}\ \Gamma}{E \vdash a := b : A\ \texttt{with}\ \Gamma}$$

Figure 1: The typing rules.

Given a type A in the context of a constraint sequence Γ, we define its free variables (labels as well as type variables) $FV(A\ \texttt{with}\ \Gamma)$ and its dangerous free variables $DV(A\ \texttt{with}\ \Gamma)$ as follows. The intuition behind the definition of FV is that the components of a functional type $A \xrightarrow{L} B$ are not only A, B and L, but also any type expression C such that $C \triangleright L$ is one of the recorded constraints.

$$\begin{aligned}
FV(X\ \texttt{with}\ \Gamma) &= \{X\} \\
FV(\mathit{int}\ \texttt{with}\ \Gamma) &= \emptyset \\
FV(A\ \mathit{ref}\ \texttt{with}\ \Gamma) &= FV(A\ \texttt{with}\ \Gamma) \\
FV(A \xrightarrow{L} B\ \texttt{with}\ \Gamma) &= \{L\} \cup FV(A\ \texttt{with}\ \Gamma) \cup FV(B\ \texttt{with}\ \Gamma) \cup \bigcup_{(\Sigma \triangleright L) \in \Gamma} FV(\Sigma\ \texttt{with}\ \Gamma) \\
DV(X\ \texttt{with}\ \Gamma) &= \emptyset \\
DV(\mathit{int}\ \texttt{with}\ \Gamma) &= \emptyset \\
DV(A\ \mathit{ref}\ \texttt{with}\ \Gamma) &= FV(A\ \texttt{with}\ \Gamma) \\
DV(A \xrightarrow{L} B\ \texttt{with}\ \Gamma) &= \bigcup_{(\Sigma \triangleright L) \in \Gamma} DV(\Sigma\ \texttt{with}\ \Gamma)
\end{aligned}$$

To complete the definition above, we extend FV and DV to type schemas and to typing environments in the obvious way:

$$\begin{aligned}
&FV\big((\forall V_1 \ldots V_n.\, A\ \texttt{with}\ \Gamma')\ \texttt{with}\ \Gamma\big) = \\
&\quad FV(A\ \texttt{with}\ \Gamma \cup \Gamma') \setminus \{V_1 \ldots V_n\} \\
&DV\big((\forall V_1 \ldots V_n.\, A\ \texttt{with}\ \Gamma')\ \texttt{with}\ \Gamma\big) = \\
&\quad DV(A\ \texttt{with}\ \Gamma \cup \Gamma') \setminus \{V_1 \ldots V_n\} \\
&FV(E\ \texttt{with}\ \Gamma) = \\
&\quad \bigcup_{x \in Dom(E)} FV(E(x)\ \texttt{with}\ \Gamma)
\end{aligned}$$

The typing rules are given in figure 1. They are very similar to the rules for ML, except for the additional handling of constraints, reminiscent of the treatment of subtyping hypotheses in type inference systems with subtypes [12, 6].

The rules define the proposition "term a has type A under assumptions E and constraints Γ", written $E \vdash a : A\ \texttt{with}\ \Gamma$. The typing environment E is a partial mapping from term variables to type schemas. We write $E[x \leftarrow A]$ for the environment identical to E, except that x is mapped to A. We assume the usual set operations are defined over constraints Γ in the obvious way.

As in Standard ML [11, p. 21], the *Gen* operator used in the (LETGEN) rule is responsible for generalizing as many type variables as possible in a type. Here, we also generalize over labels whenever possible. In addition, we prohibit generalization over dangerous type variables. A tentative definition would therefore be $\mathit{Gen}(A, E, \Gamma) = \forall V_1 \ldots V_n.\, A$ where the set $\{V_1 \ldots V_n\}$ is $FV(A\ \texttt{with}\ \Gamma) \setminus DV(A\ \texttt{with}\ \Gamma) \setminus FV(E\ \texttt{with}\ \Gamma)$. However, it would be incorrect to generalize over a variable which remains free in the constraint sequence used

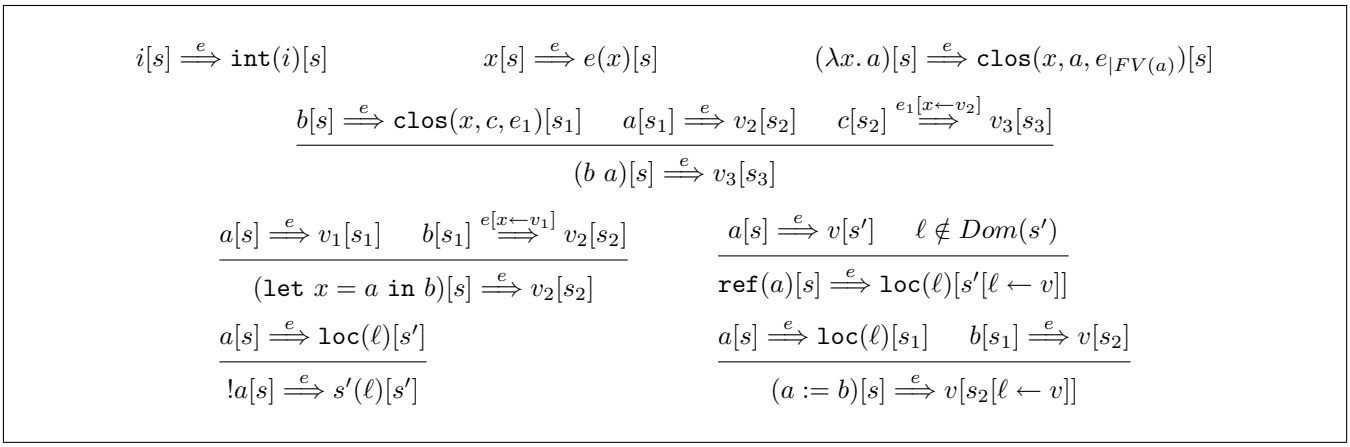

Figure 2: The evaluation rules

later. Therefore, *Gen* also discharges in the schema all "generic" constraints, i.e. the constraints in Γ where one of the V_i is free, and returns the remaining constraints, to be used further in the typing derivation.

Definition 1 (Generalization) *Let A be a type expression, E be a typing environment, Γ be a constraint sequence. Define:*

$$\{V_1 \ldots V_n\} = FV(A\ \mathtt{with}\ \Gamma) \setminus DV(A\ \mathtt{with}\ \Gamma) \setminus FV(E\ \mathtt{with}\ \Gamma).$$

Let Γ' be the sequence of those constraints $\Sigma \triangleright L$ in Γ such that L is one of the V_i. Then:

$$Gen(A, E, \Gamma) = (\forall V_1 \ldots V_n.\, A\ \mathtt{with}\ \Gamma'),\ (\Gamma \setminus \Gamma')$$

3.3 Evaluation

We give here an evaluation mechanism for terms of our calculus, using structural operational semantics. The evaluation relation $a[s] \stackrel{e}{\Longrightarrow} v[s']$ maps a term a, in the context of an evaluation environment e and a store s, to some value v, and a modified store s'. Values have the following syntax:

$$v ::= \mathtt{int}(i) \mid \mathtt{clos}(x, a, e) \mid \mathtt{loc}(\ell)$$

Here, ℓ ranges over a countable set *Loc* of locations. Stores are partial mappings from locations to values. Since we assume call-by-value, evaluation environments are partial mappings from term variables to values. The rules given in figure 2 define precisely the evaluation relation, assuming standard left-to-right evaluation order.

To account for run-time type errors, we introduce the special result `wrong`, and state that if none of the evaluation rules match, then $a[s] \stackrel{e}{\Longrightarrow} \mathtt{wrong}$. This way, we can distinguish between type errors and nontermination.

3.4 Soundness of typing

The type system presented here is sensible with respect to the evaluation mechanism above: no well-typed term can evaluate to `wrong`.

Proposition 1 *Let a be a term, A be a type, Γ be a sequence of constraints such that we can derive $\emptyset \vdash a : A\ \mathtt{with}\ \Gamma$. Then, for all stores s_0, $a[s_0]$ does not evaluate to `wrong`; that is, we cannot derive $a[s_0] \stackrel{\emptyset}{\Longrightarrow} \mathtt{wrong}$.*

The proof can be found in appendix. It closely follows Tofte's [16, chapter 5]. The crucial point is that closure labels and constraints give a better control over the types of store locations than in ML. As a consequence, a variable cannot be free in the type of a location reachable from a value without being dangerous in the type of that value. This guarantees the soundness of type generalization.

3.5 Type reconstruction

In this section, we consider an adaptation to our language of the well-known Damas-Milner type reconstruction algorithm for ML (algorithm W of [5]). In the following, we write $mgu(A, B)$ for the principal unifier of types A and B, if A and B are unifiable. (Otherwise, the type inference algorithm fails.) Type expressions are terms of a two-sorted free algebra, hence this ensures the existence of a principal unifier, that can be obtained by the classical unification algorithm between terms of a free algebra.

Let a be a term, E be a typing environment, and Γ be an initial sequence of constraints. We define $infer(E, a, \Gamma)$ as the triple (A, σ, Δ), where A is a type expression, σ a substitution and Δ a constraint sequence, as follows:

$infer(E, i, \Gamma) =$
$\quad (int, Id, \Gamma)$

$infer(E, x, \Gamma) =$
$\quad$ let $(\forall X_1 \dots X_n, L_1 \dots L_m.\, A \texttt{ with } \Delta) = E(x)$
$\quad$ let $Y_1, \dots, Y_n$ be fresh type variables
$\quad\quad$ (not free in E nor in A)
$\quad$ and $M_1, \dots, M_m$ be fresh labels
$\quad$ let $\sigma = \{X_i \leftarrow Y_i, L_j \leftarrow M_j\}$
$\quad$ in $(\sigma A, Id, \Gamma \cup \sigma\Delta)$

$infer(E, \lambda x.\, b, \Gamma) =$
$\quad$ let X, L be fresh variables
$\quad$ let $\Theta = \{E(y) \triangleright L \mid y \text{ free in } \lambda x.\, b\}$
$\quad$ let $B, \sigma, \Delta = infer(E[x \leftarrow X], b, \Gamma \cup \Theta)$
$\quad$ in $(\sigma X \xrightarrow{L} B, \sigma, \Delta)$

$infer(E, (b\ a), \Gamma) =$
$\quad$ let $B, \rho, \Theta = infer(E, b, \Gamma)$
$\quad$ let $A, \sigma, \Delta = infer(\rho E, a, \Theta)$
$\quad$ let X, L be fresh variables
$\quad$ let $\mu = mgu(\sigma B, A \xrightarrow{L} X)$
$\quad$ in $(\mu X, \mu\sigma\rho, \mu\Delta)$

$infer(E, \texttt{let } x = a \texttt{ in } b, \Gamma) =$
$\quad$ let $A, \sigma, \Delta = infer(E, a, \Gamma)$
$\quad$ let $\Sigma, \Delta' = Gen(A, \sigma E, \Delta)$
$\quad$ let $B, \rho, \Theta = infer((\sigma E)[x \leftarrow \Sigma], b, \Delta')$
$\quad$ in $(B, \rho\sigma, \Theta)$

$infer(E, \texttt{ref}(a), \Gamma) =$
$\quad$ let $A, \sigma, \Delta = infer(E, a, \Gamma)$
$\quad$ in $(A\ ref, \sigma, \Delta)$

$infer(E, !a, \Gamma) =$
$\quad$ let $A, \sigma, \Delta = infer(E, a, \Gamma)$
$\quad$ let X be a fresh variable
$\quad$ let $\mu = mgu(A, X\ ref)$
$\quad$ in $(\mu X, \mu\sigma, \mu\Delta)$

$infer(E, a := b, \Gamma) =$
$\quad$ let $A, \sigma, \Delta = infer(E, a, \Gamma)$
$\quad$ let $B, \rho, \Theta = infer(\sigma E, b, \Delta)$
$\quad$ let $\mu = mgu(\rho A, B\ ref)$
$\quad$ in $(\mu B, \mu\rho\sigma, \mu\Theta)$

This algorithm enjoys the good properties of the Damas-Milner algorithm: it is correct and complete with respect to the typing rules, and the inferred type is the most general one. The proof is very similar to Damas' proof [4].

3.6 Relation to ML

We have introduced closure typing as a way to keep track of mutable values embedded in functions. As a consequence, two expressions having the same functional type in ML may now be distinguished by their closure type, and we may fear that this leads to a type system more restrictive than the one of ML. Ideally, we would like the purely applicative fragment of our calculus (that is, without the *ref* type constructor, and the `ref`, := and ! term constructors) to be a conservative extension of ML: any pure, closed term that is well-typed in ML should also be well-typed in our calculus. Unfortunately, this is not the case — but for more subtle reasons than the one outlined above.

Actually, two type expressions in our system cannot be distinguished by their closure labels only. The reason is that unification cannot fail because of the labels: given the syntax of type expressions, a label can only be matched against another label, and labels are treated as variables as far as unification is concerned. Therefore, in our system, unification between types is conservative with respect to unification between the corresponding ML types. To be more precise, we introduce the "strip" operator $\downarrow$, that maps type expressions in our calculus to ML type expressions, by erasing the labels from function types:

$$int\downarrow = int \qquad X\downarrow = X \qquad (A\ ref)\downarrow = A\downarrow\ ref$$
$$(A \xrightarrow{M} B)\downarrow = A\downarrow \to B\downarrow$$

Then, two type expressions A and B are unifiable if and only if $A\downarrow$ and $B\downarrow$ are, and in this case, taking $\sigma = mgu(A, B)$ and $\rho = mgu(A\downarrow, B\downarrow)$, we have $(\sigma C)\downarrow = \rho(C\downarrow)$ for all types C.

This lemma, along with the close resemblance between our algorithm *infer* and Damas-Milner's W algorithm, lead us to believe that, given the same pure term, both algorithms infer the same type, modulo closure labels. However, this does not hold because of the generalization step in the case of a `let` construct. Consider the typing of `let` $x = a$ `in` b. Assume that, starting from typing environment E, algorithm *infer* infers type A `with` Γ for a, while algorithm W, starting from the corresponding ML typing environment $E\downarrow$, infers the corresponding ML type $A\downarrow$. We must check that both algorithms generalize exactly the same type variables, so that they will type b in compatible environments. This could be not the case for two reasons.

The first reason is that algorithm *infer* does not generalize dangerous type variables, while W does. But this is no problem here, since we consider only the pure fragment of our calculus, without the *ref* type constructor, hence the set of dangerous variables of any type is always empty.

The second, more serious reason is that closure typing introduces additional free variables in a given type. In general, $FTV(A \texttt{ with } \Gamma)$, the set of free type variables in $A \texttt{ with } \Gamma$, is a superset of $FV(A\downarrow)$. (Take for instance $A = int \xrightarrow{L} int$ and $\Gamma = X \triangleright L$.) So it is not obvious that $FTV(A \texttt{ with } \Gamma) \setminus FTV(E \texttt{ with } \Gamma)$, the set of type variables generalized by *infer*, is the same as $FV(A\downarrow) \setminus FV(E\downarrow)$, the set of type variables generalized by W. Indeed, there are cases where *infer* does not generalize some type variable X, while W does, because X is free in $E \texttt{ with } \Gamma$, but not in $E\downarrow$. Consider:

$$\begin{array}{l}\lambda z.\ \texttt{let } id\ = \\ \qquad \lambda x.\ \big(\texttt{if } \ldots \texttt{ then } z \texttt{ else } (\lambda y.\ x; y)\big);\ x \\ \texttt{in } id\ id\end{array}$$

Assuming $x : X$ and $y : Y$, the term $(\lambda y.\ x; y)$ is given type $Y \xrightarrow{M} Y \texttt{ with } X \triangleright M$, and the `if` construct forces z to have the same type. Therefore, when we attempt to generalize $X \xrightarrow{L} X$ (the type of $\lambda x.\ \ldots; x$), we have $E(z) = Y \xrightarrow{M} Y$ under constraints $\Gamma = X \triangleright M,\ Y \xrightarrow{M} Y \triangleright L$, and we cannot generalize over X, since it is free in the type of z. Hence, *id* remains monomorphic, and the application *id id* is ill-typed. In ML, we would have $z : Y \to Y$, so we could freely generalize over X, getting $id : \forall X.\, X \to X$, and the whole term would be well-typed.

The example above is quite convoluted, and it is the simplest one we know that exhibits this "variable capture through labels" phenomenon. We need more experience with the proposed type system to find whether this phenomenon happens in more practical situations. However, in case this turns out to be a serious flaw of our closure typing system, we are investigating two possible improvements that seem to avoid variable captures.

One direction is to record less constraints when typing a λ-abstraction. In the example above, one could argue that the constraint $X \triangleright M$ should not be recorded, on the grounds that x is "not actually used" in the body of the function, as witnessed by the fact that the type variable X is not free in the type of the function result, nor in the type of the parameter.

The other direction is to check that two function types are compatible without actually identifying their closure types. This way, we could avoid the propagation of the constraint $X \triangleright M$ to the type of z.

3.7 Pragmatics of constraint handling

Practically, the main concern with the type inference algorithm above is the additional overhead introduced by the handling of constraints. First, it is possible to simplify sequences of constraints. Here are some possible simplification rules:

$$\begin{array}{rcl}\Sigma \triangleright M, \Sigma \triangleright M & \to & \Sigma \triangleright M \\ \Sigma \triangleright M & \to & \epsilon \quad \text{if } \Sigma \text{ is closed} \\ A \times B \triangleright M & \to & A \triangleright M, B \triangleright M\end{array}$$

These three simplifications are obviously sound with respect to the computation of free and dangerous variables. The third rule actually generate more constraints, but this may open the way to further simplifications, e.g. if $A = B$, or if A is closed.

In addition, constraint handling becomes quite cheap if we distribute constraints inside types, instead of handling a single list of constraints. We suggest grouping together all constraints over the same label L, arranged as a list of type schemes. This list represents the label L itself. To identify two labels, we just have to concatenate the two lists, and this can be done in constant time, using e.g. difference lists. Hence, when unifying two type expressions A and B, the additional work of identifying labels takes time at most proportional to the size of A, B. Therefore, unification can be performed in linear time, as in ML.

4 Comparison with other type systems

In this section, we compare our type system with previous proposals of polymorphic type systems for languages featuring physical modification.

4.1 Standard ML

We consider first the systems proposed for ML. All these systems rely on detecting the creation of mutable values (e.g. by special typechecking rules for the `ref` construct), and ensuring that the resulting mutable values have monomorphic types.

The first system we consider is the one proposed by Tofte [16, 17], and adopted in Standard ML [11]. It makes use of weak type variables (written with a * superscript) to prohibit polymorphic references. Weak type variables cannot be generalized by a `let` binding, unless the corresponding expression is guaranteed to be non-expansive, i.e. that it does not create any references. Damas [4] proposed a related, but slightly different scheme, that gives similar results in most cases.

The other system is an extension of Tofte's, used in the Standard ML of New Jersey implementation [1]. In this scheme, type variables are no longer partitioned into weak and non-weak variables; instead, each type

	Standard ML	SML-NJ	Our system
make_ref *make_ref* []	$\alpha^* \to \alpha^*$ *ref* rejected	$\alpha^1 \to \alpha^1$ *ref* rejected	$\alpha \xrightarrow{L} \alpha$ *ref* rejected
imperative_map *imperative_map id* []	$(\alpha^* \to \beta^*) \to$ α^* *list* $\to \beta^*$ *list* rejected	$(\alpha^2 \to \beta^2) \to$ α^2 *list* $\to \beta^2$ *list* rejected	$(\alpha \xrightarrow{L} \beta) \xrightarrow{M} \alpha$ *list* $\xrightarrow{N}$ β *list* `with` $\alpha \xrightarrow{L} \beta \triangleright N$ α *list*
applicative_map make_ref	rejected	rejected	α *list* $\xrightarrow{L} \alpha$ *ref list*
id make_ref	rejected	rejected	$\alpha \xrightarrow{L} \alpha$ *ref*
(`raise` *Exit* : α *ref*)	α *ref*	α *ref*	rejected

Figure 3: Comparison with other type systems

variable has an associated integer, its "degree of weakness", or "strength". This degree measures the number of abstractions that have to be applied before the corresponding reference is actually created. Regular, unconstrained type variables have strength infinity. Variables with strength zero cannot be generalized. Variables with strength $n > 0$ can be generalized, but each function application decrements the strength of variables.

The comparative results are given in figure 3. For each test program, we give its most general type in each system. We assume these are top-level phrases. As in most ML implementations, we reject top-level phrases whose types cannot be closed (because some free variables cannot be generalized).

The first test is the *make_ref* function, defined as `function` $x \to$ `ref` x. It exercises the possibility of writing generic functions that create and return updatable data structures. Most functions over vectors, matrixes, doubly linked lists, ... are typed similarly. All type systems considered here capture the fact that *make_ref* should be applicable to monomorphic values only.

The second test is the *imperative_map* functional given in the introduction. It illustrates the use of polymorphic references as auxiliaries inside a generic function. Our type system is the only one which gives a fully polymorphic type to *imperative_map*. The others restrict it to be used with monomorphic type.

The third test is the partial application of *applicative_map*, defined in the introduction, to the *make_ref* function. It exercises the compatibility between functions that create mutable data and higher-order functions. SML and SML-NJ refuse to generalize its type, hence reject it. They are unable to detect that *applicative_map* does not apply *make_ref* immediately, hence that *applicative_map make_ref* does not create any polymorphic reference.

A simplified version of this test, shown below, is to apply the identity function *id* to the *make_ref* function. Our type system gives the same type to *id make_ref* and to *make_ref*. The others don't, and we take this as strong evidence that they do not handle full functionality correctly.

The last example illustrates a weakness of our type-based approach. By using exceptions, for instance, one may mimic the creation of a reference as far as types are concerned, without actually creating one. In the expression (`raise` *Exit* : α *ref*), our type system considers that α is dangerous and does not generalize it. Other type systems recognize that no references are created, hence they correctly generalize it. This flaw of our method has little impact on actual programming, however.

4.2 Quest

In our system, we made no attempt at restricting the creation of mutable values, and concentrated on type generalization instead. We were inspired by Cardelli's Quest language [2], which departs significantly from ML, but features mutable data structures and polymorphic typing.

Quest makes almost no typing restrictions for mutable values. Soundness is ensured by different semantics for type specialization. Namely, an expression with polymorphic type is evaluated each time its type is specialized, in contrast with ML, where it would be evaluated only once. This is consistent with the fact that polymorphism is explicit in Quest programs: polymorphic objects are actually presented as functions that take a type as argument and return a specialized version of the object. But these semantics are incompatible with ML, where generalization and specialization are kept implicit in the source program.

4.3 FX

The FX effect system [9] is a polymorphic type system that performs purity analysis as well: the type of an expression indicates what kind of side-effects its evaluation can perform. This provides a simple way to deal with the problem of mutable values: the type of an expression cannot be generalized unless this expression is referentially transparent. It is easy to see that such "pure" expressions can safely be used with different types.

Though this approach is attractive for other purposes (e.g. automatic program parallelization), it definitely does not address the main issue considered here: how to give the same type to semantically equivalent functions, whether written in applicative style or in imperative style. The reason is that the purity analysis of FX makes it apparent in the types whether a function allocates mutable data for local purposes. For instance, *imperative_map* and *applicative_map* do not have the same type in FX.

5 Conclusion

We have presented herein an extension of the ML type system that considerably enhances the support for data accepting in-place modification. This is a significant step toward the integration of polymorphic type discipline and imperative programming style. We have introduced the notion of closure typing, which is essential to the soundness of this approach. We have given one type system that performs this closure typing, based upon labels and constraints. Our system is simple, but falls short of being a conservative extension of ML. Further work includes investigating alternate, more subtle ways to typecheck closures, that would not reject any well-typed ML program, while correctly keeping track of mutable data.

The scope of this work is not strictly limited to the problem of mutable data structures. For instance, it is well-known that polymorphic exceptions raise the same issues as references, and can be handled in the same way. More generally, similar problems arise in the integration of polymorphic typing within several other programming paradigms. For instance, one approach to the integration of functional and logic programming is to allow partially defined values containing logical variables within a conventional functional language [15]. Polymorphic logical variables break type safety just as polymorphic references do, and are amenable to the same treatment. In object-oriented programming, status variables of objects can be seen as references systematically encapsulated in functions (the methods). Closure typing seems relevant to the polymorphic typechecking of such objects. Finally, some calculi of communicating systems feature channels as first-class values [10]. Polymorphic typing of these channels must guarantee that senders and receivers agree on the types of transmitted values, and this is similar to ensuring that writers and readers of a reference use it consistently [13].

Acknowledgments

Didier Rémy suggested the use of constraints in the typing rules. We also benefited from his expertise in type inference and unification problems. Many thanks to Brad Chen for his editorial help.

A Proof of soundness

In this appendix, we sketch the proof of soundness of the type system with respect to the operational semantics given in section 3.3.

We formalize the fact that a value v semantically belongs to a type expression A under constraints Γ. We write this $S \models v : A$ `with` Γ. The hypothesis S is a *store typing*, that is, a partial mapping from locations to type expressions. The store typing is needed to take into account the sharing of values introduced by the store.

Definition 2 (Semantic typing judgements) *Let S be a store typing, v a value, A a type, Γ a constraint sequence. The predicate $S \models v : A$* `with` Γ *is defined by induction on v:*

- $S \models \texttt{int}(i) : \mathit{int}$ `with` Γ.
- $S \models \texttt{loc}(\ell) : (A\ \mathit{ref})$ `with` Γ *iff ℓ belongs to the domain of S, and $S(\ell) = A$.*
- $S \models \texttt{clos}(x, b, e) : (A \xrightarrow{M} B)$ `with` Γ *iff*
  - *for all $x \in Dom(e)$, there exists a type schema Σ such that $\Sigma \triangleright M$ is in Γ, and $S \models e(x) :$ Σ* `with` Γ*;*
  - *and there exists a typing environment E such that $E[x \leftarrow A] \vdash b : B$* `with` Γ.

The predicate above is extended to type schemas by taking $S \models v : (\forall V_1 \ldots V_n . A$ `with` $\Gamma')$ `with` Γ *iff for all substitutions μ over $V_1 \ldots V_n$, we have $S \models v : \mu A$* `with` $\Gamma \cup \mu\Gamma'$

Let e be an evaluation environment, and E be a typing environment. We say that $S \models e : E$ `with` Γ *iff $Dom(e) \subseteq Dom(E)$ and for all $x \in Dom(e)$, we have $S \models e(x) : E(x)$* `with` Γ.

Similarly, let s be a store. We say that $\models s : S$ $\texttt{with}\ \Gamma$ if $Dom(s) \subseteq Dom(S)$ and for all $\ell \in Dom(s)$, we have $S \models s(\ell) : S(\ell)\ \texttt{with}\ \Gamma$.

In the proof of soundness, we will use the fact that the semantic typing judgement is stable under substitution.

Lemma 1 (Semantic substitution) *If $S \models v : A\ \texttt{with}\ \Gamma$, then $\mu S \models v : \mu A\ \texttt{with}\ \mu\Gamma$ for all substitutions μ.*

Starting from a given value v, it is not possible in general to access any location in the store. The set $R(v)$ of locations reachable from v is defined by induction on v, as follows.

Definition 3 (Reachable locations)

$$\begin{aligned} R(\texttt{int}(i)) &= \emptyset \\ R(\texttt{loc}(\ell)) &= \{\ell\} \\ R(\texttt{clos}(x,a,e)) &= \bigcup_{y \in Dom(e)} R(e(y)) \end{aligned}$$

In the definition of $S \models v : A\ \texttt{with}\ \Gamma$, the types of the locations that cannot be reached from v are irrelevant. We can actually assume any other type for those locations:

Lemma 2 (Garbage collection) *Let S, S' be two store typings such that $S(\ell) = S'(\ell)$ for all $\ell \in R(v)$. If $S \models v : A\ \texttt{with}\ \Gamma$, then $S' \models v : A\ \texttt{with}\ \Gamma$*

Our type expressions are informative enough to allow us to connect the type of a location ℓ reachable from a value v with the type A of v. More precisely, we have the following key lemma which relates the variables free in the type of ℓ with the dangerous variables of A.

Lemma 3 (Store typing control) *Assume $S \models v : A\ \texttt{with}\ \Gamma$. Let ℓ be a location in $R(v)$. Then*

$$FV(S(\ell)\ \texttt{with}\ \Gamma) \subseteq DV(A\ \texttt{with}\ \Gamma).$$

Proof: By induction on v.

Case 1: $A = int$ and $v = \texttt{int}(i)$. Obvious, since there are no locations reachable from v.

Case 2: $A = B\ ref$, $v = \texttt{loc}(\ell')$, and $S(\ell') = B$. By definition of the reachable locations, $\ell = \ell'$. Hence $FV(S(\ell)\ \texttt{with}\ \Gamma) = FV(B\ \texttt{with}\ \Gamma) = DV(A\ \texttt{with}\ \Gamma)$.

Case 3: $A = A_1 \xrightarrow{L} A_2$, and $v = \texttt{clos}(x,a,e)$. Let y be a variable such that ℓ is reachable from $e(y)$. Let Σ be a type scheme such that $S \models e(y) : \Sigma$ and $(\Sigma \triangleright L) \in \Gamma$. We write $\forall V_1 \ldots V_n.\, B\ \texttt{with}\ \Gamma'$ for Σ. For all substitutions μ over $V_1 \ldots V_n$, since $S \models e(y) : \mu B\ \texttt{with}\ \Gamma \cup \mu\Gamma'$, we get by induction hypothesis:

$$FV(S(\ell)\ \texttt{with}\ \Gamma \cup \mu\Gamma') \subseteq DV(\mu B\ \texttt{with}\ \Gamma \cup \mu\Gamma')$$

Taking well-chosen substitutions μ, it follows that:

$$FV(S(\ell)\ \texttt{with}\ \Gamma) \subseteq DV(B\ \texttt{with}\ \Gamma \cup \Gamma') \setminus \{V_1 \ldots V_n\}$$

Hence the desired result: $FV(S(\ell)\ \texttt{with}\ \Gamma) \subseteq DV(\Sigma\ \texttt{with}\ \Gamma) \subseteq DV(A\ \texttt{with}\ \Gamma)$ □

We are now ready to show the soundness of the typing rules.

Proposition 2 (Soundness) *Assume $E \vdash a : A\ \texttt{with}\ \Gamma$. Let e be an evaluation environment, s a store, S a store typing such that:*

$$S \models e : E\ \texttt{with}\ \Gamma \qquad \models s : S\ \texttt{with}\ \Gamma.$$

Assume $a[s] \stackrel{e}{\Longrightarrow} w$. Then, $w = v[s']$, and there exists a store typing S' extending S, such that:

$$S' \models v : A\ \texttt{with}\ \Gamma \qquad \models s' : S'\ \texttt{with}\ \Gamma$$

Proof: By induction on the length of the evaluation. All cases proceed as in Tofte's proof [16, chapter 5], except when a is a `let` binding. Then, the last step of the typing derivation is:

$$\frac{\begin{array}{c} E \vdash a : A\ \texttt{with}\ \Gamma \\ (\Sigma, \Gamma') = Gen(A, E, \Gamma) \\ E[x \leftarrow \Sigma] \vdash b : B\ \texttt{with}\ \Gamma' \end{array}}{E \vdash \texttt{let}\ x = a\ \texttt{in}\ b : B\ \texttt{with}\ \Gamma'}$$

Applying the induction hypothesis to the first premise, we get v_a, s_a, S_a extending S such that:

$$a[s] \stackrel{e}{\Longrightarrow} v_a[s_a]$$

$$S_a \models v_a : A\ \texttt{with}\ \Gamma \qquad \models s_a : S_a\ \texttt{with}\ \Gamma.$$

To apply the induction hypothesis to the last premise, we need to show that:

$$S_a \models e[x \leftarrow v_a] : E[x \leftarrow \Sigma]\ \texttt{with}\ \Gamma'$$

where we write (Σ, Γ') for $Gen(A, E, \Gamma)$, and $\forall V_1 \ldots V_n.\, A\ \texttt{with}\ \Gamma''$ for Σ. Since S_a extends S, and $V_1 \ldots V_n$ are not free in E, we already have $S_a \models e : E\ \texttt{with}\ \Gamma'$ as a corollary of lemma 2. It remains to show that:

$$S_a \models v_a : (\forall V_1 \ldots V_n.\, A\ \texttt{with}\ \Gamma'')\ \texttt{with}\ \Gamma'.$$

To do so, we must prove that, for any substitution μ over the V_i,

$$S_a \models v_a : \mu A\ \texttt{with}\ \Gamma' \cup \mu\Gamma''. \tag{1}$$

Since $S_a \models v_a : A\ \texttt{with}\ \Gamma$, we have, by lemma 1,

$$\mu S_a \models v_a : \mu A\ \texttt{with}\ \mu\Gamma. \tag{2}$$

By definition of Gen, none of the V_i appears in any constraint of Γ', hence

$$\mu(\Gamma) = \mu(\Gamma' \cup \Gamma'') = \Gamma' \cup \mu\Gamma''. \tag{3}$$

By lemma 3, for any location ℓ reachable from v_a, we have $FV(S_a(\ell)\ \texttt{with}\ \Gamma) \subseteq DV(A\ \texttt{with}\ \Gamma)$, and none of the V_i is dangerous in $A\ \texttt{with}\ \Gamma$, hence none of the V_i is free in $S_a(\ell)$. Therefore,

$$(\mu S_a)(\ell) = S_a(\ell) \text{ for all } \ell \in R(v_a) \tag{4}$$

and the claim (1) above follows from (2), (3), (4), and lemma 2. Therefore, we can apply the induction hypothesis to the right branch of the typing derivation, getting v_b, s_b, S_b extending S_a such that:

$$b[s_a] \overset{e[x \leftarrow v_a]}{\Longrightarrow} v_b[s_b]$$

$$S_b \models v_b : B\ \texttt{with}\ \Gamma' \qquad \models s_b : S_b\ \texttt{with}\ \Gamma'$$

which is the expected result. □

References

[1] A. W. Appel and D. B. MacQueen. *Standard ML reference manual (preliminary)*. AT&T Bell Laboratories, 1989. Included in the Standard ML of New Jersey distribution.

[2] L. Cardelli. Typeful programming. In E. J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts*, pages 431–507. Springer-Verlag, 1989.

[3] G. Cousineau and G. Huet. The CAML primer. Technical report 122, INRIA, 1990.

[4] L. Damas. *Type assignment in programming languages*. PhD thesis, University of Edinburgh, 1985.

[5] L. Damas and R. Milner. Principal type-schemes for functional programs. In *9th symposium Principles of Programming Languages*, pages 207–212. ACM Press, 1982.

[6] Y.-C. Fuh and P. Mishra. Type inference with subtypes. In *ESOP '88*, volume 300 of *Lecture Notes in Computer Science*, pages 94–114. Springer Verlag, 1988.

[7] J.-Y. Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Thèse d'État, Université Paris VII, 1972.

[8] M. J. Gordon, A. J. Milner, and C. P. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[9] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *15th symposium Principles of Programming Languages*, pages 47–57. ACM Press, 1988.

[10] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes: part 1. Research report ECS-LFCS-89-85, University of Edinburgh, 1989.

[11] R. Milner, M. Tofte, and R. Harper. *The definition of Standard ML*. The MIT Press, 1990.

[12] J. C. Mitchell. Coercion and type inference. In *11th symposium Principles of Programming Languages*, pages 175–185. ACM Press, 1984.

[13] J. H. Reppy. First-class synchronous operations in Standard ML. Technical Report TR 89-1068, Cornell University, 1989.

[14] J. C. Reynolds. Toward a theory of type structure. In *Programming Symposium, Paris, 1974*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425. Springer-Verlag, 1974.

[15] G. Smolka. FRESH: a higher-order language with unification and multiple results. In D. DeGroot and G. Lindstrom, editors, *Logic Programming: Functions, Relations, and Equations*, pages 469–524. Prentice-Hall, 1986.

[16] M. Tofte. Operational semantics and polymorphic type inference. PhD thesis CST-52-88, University of Edinburgh, 1988.

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