Relaxing the Value Restriction

Jacques Garrigue
Research Institute for Mathematical Sciences
Kyoto University, Sakyo-ku, Kyoto 606-8502
garrigue@kurims.kyoto-u.ac.jp

ABSTRACT

Restricting polymorphism to values is now the standard way to obtain soundness in ML-like programming languages with imperative features. While this solution has undeniable advantages over previous approaches, it forbids polymorphism in many cases where it would be sound. We use a subtyping based approach to recover part of this lost polymorphism, without changing the type algebra itself, and this has significant applications.

1. INTRODUCTION

Restricting polymorphism to values, as Wright suggested [19], is now the standard way to obtain soundness in ML-like programming languages with imperative features. Section 2 explains how this conclusion was reached. This solution's main advantages are its utter simplicity (only the generalization rule is changed from the original Hindley-Milner type system), and the fact it avoids distinguishing between applicative and imperative type variables, giving identical signatures to pure and imperative functions.

Of course, this solution is sometimes more restrictive than previous ones: by assuming that all functions may be imperative, lots of polymorphism is lost. However, this extra polymorphism appeared to be of limited practical use, and experiments have shown that the changes needed to adapt ML programs typechecked using stronger type systems to the value only polymorphism type system were neglectible.

Almost ten years after the feat, it might be useful to check whether this is still true. Programs written ten years ago were not handicapped by the value restriction, but what about programs we write now, or programs we will write in the future?

In his paper, Wright considers 3 cases of let-bindings where the value restriction causes a loss of polymorphism.

1. Expressions that never return. They do not appear to be really a problem, but he remarks that in the specific case of $\forall\alpha.\alpha$, it would be sound to keep the stronger type.

2. Expressions that compute polymorphic procedures. This amounts to a partial application. Analysis of existing code showed that their evaluation was almost always purely applicative, and as a result one could recover the polymorphism through eta-expansion of the whole expression, except when the returned procedure is itself embedded in a data structure.

3. Expressions that return polymorphic data structures. A typical example is an expression returning always the empty list. It should be given the polymorphic type α `list`, but this is not possible under the value restriction if the expression has to be evaluated.

Of these 3 cases, the last one, together with the data-structure case of the second one, are most problematic: there is no workaround to recover the lost polymorphism, short of recomputing the data structure at each use. This seemed to be a minor problem, because existing code made little use of this kind of polymorphism inside a data structure. However we can think of a number of cases where this polymorphism is expected, sometimes as a consequence of extensions to the type system.

1. Constructor and accessor functions. While algebraic datatype constructors and pattern matching are handled specially by the type system, and can be given a polymorphic type, as soon as we define functions for construction or access, the polymorphism is lost. The consequence is particularly bad for abstract datatypes and objects [15], as one can only construct them through functions, meaning that they can never hold polymorphic values.

2. Polymorphic variants [2]. By nature, a polymorphic variant is a polymorphic data structure, which can be seen as a member of many different variant types. If it is returned by a function, or contains a computation in its argument, it looses this polymorphism.

3. Semi-explicit polymorphism [4]. This mechanism allows to keep principality of type-checking in the presence of first-class polymorphism. This is done through adding type variable markers to first-class polymorphic types, and checking their polymorphism. Unfortunately, value restriction looses this polymorphism. A workaround did exist, but the resulting type system was only "weakly" principal.

We will review these cases, and show how the value restriction can be relaxed a little, just enough for many of these problems to be leveled. As a result, we propose a new type system for ML, with *relaxed value restriction*, that is strictly more expressive (it types more programs) than ML with the usual value restriction.

The starting point is very similar to the original observation about $\forall\alpha.\alpha$: in some cases, polymorphic types are too generic to contain any value. As such they can only

describe empty collections, and it is sound to allow their generalization.

Our basic idea is to use the structural rules of subtyping to recover this polymorphism: by subsumption, if a type appears only in covariant positions in the type of a value, it shall be safe to replace it with any of its supertypes. From a set-theoretic point of view, if this type is not inhabited, then it is a subtype of all other types (they all contain the empty set). If it can be replaced by any type, then we can make it a polymorphic variable. For instance, consider this expansive binding:

```
val f : unit -> '_a list
```

The variable `'_a` is non-generalizable: it can be instantiated only once, and is shared between all uses of `f`. We can replace `'_a` by the base type `zero`. Assuming that `zero` is not inhabited, it is sound to replace all its covariant occurrences by polymorphic variables:

```
val f : unit -> 'a list
```

Since `'_a` had only covariant occurrences, `zero` does not appear in this new type, making it strictly more general than the original one.

Unfortunately, this simple reasoning cannot be translated into a direct proof : we are aware of no set theoretic model of ML extended with references. Nonetheless this intuition will lead us to a semi-syntactic proof using semantic types.

As an interesting aside to this result, we will see that the resulting system, while being sound, does not enjoy the subject reduction. This may explain why it was not considered to date.

This paper is organized as follows. After a short reminder on why the value restriction became so popular, we give some examples of our scheme applied to simple cases, and then show how it helps solving the problems described above. In section 5 we formalize our language and type system, and prove its soundness using semantic types in section 6, before concluding.

2. WHY THE VALUE RESTRICTION

Before discussing in what way we are improving on the value restriction, it is useful to explain why this seemingly weak approach has become the standard solution to the soundness problem created by ML's imperative features. We expose the path chronologically, but do not enter into technical details. Readers familiar with this problem may skip directly to section 3.

2.1 The soundness problem

The original problem is well-known: in the presence of mutable references (and also of other imperative features, like continuations), the usual typing rule for the polymorphic let is unsound. We use Objective Caml syntax and library for our examples. Programs are in typewriter font, and output from the interpreter in italic.

```
let r = ref []
val r : 'a list ref
r := [3]; r
- : 'a list ref
let l =
  List.map (function true -> 1 | false -> 2) !r
val l : int list = Segmentation fault
```

If we apply the usual rule, we can give a polymorphic type to `r`. Since each use of `r` is then assigned a different instance of this polymorphic type, assigning a value of type `int list` does not change the type of other uses of `r`. As a result we are able to use `r` in a context expecting another type, which causes a runtime type error, or undefined behavior if the compiler removed type checks.

The problem at hand is clear enough: `'a` in the above example should not be allowed to be polymorphic, because it is the type of the contents of a reference cell, and this contents can be modified. If `'a` is kept monomorphic, then it must be instantiated to the same type in all uses of `r`, so that we have:

```
r := [3]; r
- : int list ref
```

and unsound uses of `r` are not allowed anymore.

The question is: how can we restrict the type system, keeping principality, so that mutable data will not be given a polymorphic type?

2.2 Conservative solutions

The first natural direction to take is to design a conservative extension to ML, satisfying the above restriction, but also able to type all programs typable in ML without references.

The simplest conservative approach is to just keep monomorphic all variables used somewhere under the `ref` type constructor. The old Caml system [18] made such a choice. However, it became quickly apparent that such a restrictive approach gives imperative features only second-class citizenship. For instance this definition of map using reference cells would not be given a polymorphic type (*c.f.* comparison in figure 1):

```
let imp_map f l =
  let input = ref l and output = ref [] in
  while !input <> [] do
    output := f (List.hd !input) :: !output;
    input := List.tl !input
  done;
  List.rev !output
```

Since `l` is stored in a reference, the only way to have this program accepted would be to explicitly force it to accept only lists of a fixed ground type (`int` or `string` for instance).

This typing seeming too restrictive, more refined type systems were developed to handle the specificity of types affected by side-effects. The Tofte discipline [17], used in Standard ML 90 [11], introduced imperative type variables for references, marked by a "*". They must be instantiated to ground types whenever a side-effect may occur, *i.e.* after any function application or reference cell creation. This was extended in Standard ML of New Jersey to allow for deeper curried functions [6, 5]. You can see in the comparison table that `imp_map` may take two arguments before requiring ground instantiation. This subsumes the Tofte discipline: you just have to replace "*" by "1".

While above typings do allow some degree of polymorphism, one may remark that references in `imp_map` are purely local to the computation, and do not escape from its scope. As such, this would be sound to make them normal polymorphic variables. Yet more refined type systems, based either on effect analysis by Talpin and Jouvelot [16] or closure typing by Leroy and Weis [10, 7], are able to extract this polymorphism, by tracking in more detail creation and access of references. They both give the same type to `imp_map` and an applicative version of `map`, but this is at the price of adding information about the program execution flow. This means complex types, which may be acceptable for a system based on type inference alone, but are awkward when one has to explicitly write them, in ML module signatures for instance.

System	Type of `imp_map`
Old Caml	$(\texttt{int} \to \texttt{string}) \to \texttt{int list} \to \texttt{string list}$
SML 90	$(\alpha^* \to \beta^*) \to \alpha^* \texttt{ list} \to \beta^* \texttt{ list}$
SML/NJ	$(\alpha^2 \to \beta^2) \to \alpha^2 \texttt{ list} \to \beta^2 \texttt{ list}$
Effects	$(\alpha \xrightarrow{S} \beta) \to \alpha \texttt{ list} \xrightarrow{S} \beta \texttt{ list}$
Closure	$(\alpha \xrightarrow{L} \beta) \xrightarrow{M} \alpha \texttt{ list} \xrightarrow{N} \beta \texttt{ list}$ with $\alpha \xrightarrow{L} \beta \triangleright N$
Value	$(\alpha \to \beta) \to \alpha \texttt{ list} \to \beta \texttt{ list}$

Figure 1: Comparing types

2.3 Simplicity and abstraction

By 1993, some people could see that these more and more complex attempts at conservative extensions were doomed. Of this negative conclusion, two requirements emerged: keeping the type algebra simple, and keeping the implementation abstract in types. All the conservative systems have to reveal information about how a function is implemented, breaking this abstraction. In practice, this means that when defining the signature of a module, one has to decide in advance how it will be implemented. This goes against the goal of "programming in the large" promoted by the ML module system, and can be particularly awkward when one changes the implementation and realizes that the types do not fit anymore.

The only solution left was to drop conservativity: accept that some existing ML programs will not be typable anymore. A first attempt by Leroy was to restrict polymorphism to call-by-name bindings, as they have clearly no side-effects [8]. This avoids any change in the type algebra, but requires some in the syntax. Yet, this didn't seem to restrict the expressivity of the language.

However, a simpler way to obtain the same result was the value restriction [20]: similarly polymorphism is limited to bindings without side-effects, but the syntax is left unchanged. The choice of the typing rule to apply for `let` is driven by a syntactic definition of *values*, which includes variables, functions, and all constructs except function application and reference cell creation. With the value restriction, imperative and applicative version of functions receive the same type, even if the imperative version hides some references in a closure. There is no magic, rather than tracking the danger carefully as previous systems did, the value restriction just assumes that all function applications are dangerous: their results are not generalizable locally. This is actually equivalent to the Tofte discipline, assuming all variables are imperative. To beginners this may cause some gripes, as some types become monomorphic. This is particularly confusing when using an interpreter, and experimenting with partial applications. However tests on a huge corpus of programs showed that the transition was very easy, with only a few places where eta-expansion was needed. After all the headaches caused by overly specific types, this appeared as the solution.

Since then, the community seems to have settled with the value restriction, which was first adopted by Caml in 1995, and Standard ML in 1997.

To finish this overview, an interesting improvement of the value restriction was suggested by Ohori with the introduction of rank 1 polymorphism [13]: by allowing quantification in non-prenex positions, for instance $\texttt{int} \to \forall\alpha.\alpha \to \alpha \texttt{ list}$, it can recover some lost polymorphism, much in the same way as indexed weak variables improved on imperative type variables. Yet this re-introduces some complexity, and reveals the implementation in some cases.

3. POLYMORPHISM FROM SUBTYPING

With the background of the previous section, we can now better define our intent.

We follow the value restriction, and keep its principles: simplicity and abstraction. That is, we do not distinguish at the syntactic level between *applicative* and *imperative* type variables; neither do we introduce different points of quantification, as in rank-1 polymorphism. All type variables in any function type are to be seen as imperative: by default, they become non-generalizable in the let-binding of a non-value (*i.e.* a term containing a function application), on a purely syntactical criterion.

However we can analyze the semantic properties of types, independently of the implementation. By distinguishing between covariant and contravariant variables in types we are able to partially lift this restriction when generalizing: as before, variables with contravariant occurrences in the type of an expansive expression cannot be generalized, but variables with only covariant occurrences can be generalized.

The argument goes as follows. We introduce a new type constructor, `zero`, which is kept empty. We choose to instantiate all non-contravariant variables in let-bound expressions by `zero`. In a next step we coerce the type of the let-bound variable to a type where all `zero`'s are replaced by fresh type variables. Since the coercion of a variable is a value, in this step we are no longer limited by the value restriction, and these type variables can be generalized.

To make explanations clear, we will present our first two examples following the same pattern: first give the non-generalizable type scheme as by the value restriction (typed by Objective Caml 3.06 [9]), then obtain a generalized version by explicit subtyping. However, as explained in the introduction, our real intent is to provide a replacement for the usual value restriction, so we will only give the generalized version —as Objective Caml 3.07 will—, in subsequent examples. Here is our first example.

```
let l =
  let r = ref [] in !r
val l : '_a list = []
```

The `_` in `'_a` means that the type variable is not generalized: it will be instantiated when used, and fixed afterwards. This basically means that `l` is now of a fixed typed, and cannot be used in polymorphic contexts anymore.

Our idea is to recover polymorphism through subtyping.

```
let l = (l : zero list :> 'a list)
val l : 'a list = []
```

A coercion $(e : \tau_1 :> \tau_2)$ makes sure that e has type τ_1, and that τ_1 is a subtype of τ_2. Then, it can safely be seen as having type τ_2. Since `l` is a value, and the coercion of a value is also a value, this is a value binding, and the new `'a` in the type of the coerced term can be generalized.

Why is it sound? Since we assigned an empty list to `r`, and returned its contents without modification, `l` can only be the empty list; as such it can safely be assigned a polymorphic type.

Note that Leroy's closure-based type system would indeed infer the same polymorphic typing, but Tofte's imperative type variables would not: since the result is not a closure, with Leroy's approach the fact `[]` went through a reference cell doesn't matter; however, Tofte's type system would force its type to be imperative, precluding any further generalization when used inside a non-value binding.

$$
\begin{array}{rcl}
V^-(\alpha) & = & \emptyset \\
V^-(\tau\ \texttt{ref}) & = & FTV(\tau) \\
V^-(\tau_1 \to \tau_2) & = & FTV(\tau_1) \cup V^-(\tau_2) \\
V^-(\tau_1 \times \tau_2) & = & V^-(\tau_1) \cup V^-(\tau_2) \\
V^-(\tau\ \texttt{list}) & = & V^-(\tau)
\end{array}
$$

Figure 2: Dangerous variables

The power of this approach is even more apparent with function types. This is the example from the introduction.

```
let f =
  let r = ref [] in fun () -> !r
val f : unit -> '_a list
```

which we can coerce again

```
let f = (f : unit -> zero list :> unit -> 'a list)
val f : unit -> 'a list
```

This result may look more surprising, as actually `r` is kept in the closure of `f`. But since there is no way to modify its contents, `f` can only return the empty list. This time, even Leroy's closure typing and Talpin&Jouvelot's effect typing cannot meet the mark.

This reasoning holds as long as a variable does not appear in a contravariant position. Yet, for type inference reasons we explain in section 5, we define a set of dangerous variables (figure 2) including all variables appearing on the left of an arrow, which is more restrictive than simple covariance. In a non-value binding, we will generalize all local variables except those in $V^-(\tau)$, assuming the type before generalization is τ.

This restriction to safe variables means that we need a bit of encoding to obtain polymorphism in continuation passing style, where the variable would be of rank 2.

```
type (+'a,'b) cps = CPS of (('a -> 'b) -> 'b)
let f = let r = ref [] in CPS (fun k -> k !r)
val f : ('a list, '_b) cps
let CPS f' = f
val f' : ('a list -> '_b) -> '_b
```

The `+` in `type (+'a,'b) cps` is a variance annotation, and is available in Objective Caml since version 3.01. It means that `'a` appears only in covariant positions in the definition of `cps`. Datatype definitions being generative, they are not expanded during type inference, and all covariant variables can be considered as safe. This additional information was already used for explicit subtyping coercions (between types including objects or variants), but with our approach we can also use it to automatically extract more polymorphism.

Of course, this subtyping approach cannot always recover all the polymorphism lost by the value restriction. Consider for instance the partial application of `map` to the identity function.

```
let map_id = List.map (fun x -> x)
val map_id : '_a list -> '_a list
```

Since `'_a` also appears in a contravariant position, there is no way this partial application can be made polymorphic. But it is clear also that, keeping a simple type algebra, making the above function polymorphic would be unsound. We can design an evil variant of map, with the same type, exhibiting this unsoundness.

```
let evil_map f =
  let r = ref [] in
  List.map (fun x ->
    let y = !r in r := [f x];
    if y = [] then List.hd !r else List.hd y)
val evil_map : ('a -> 'b) -> 'a list -> 'b list
```

This definition is offsetting its output by one. If one was to define `map_id` with `evil_map`, and give it a polymorphic type, this would breach type safety. As usual, we can still recover polymorphism by eta-expansion.

```
let map_id l = List.map (fun x -> x) l
val map_id : 'a list -> 'a list
```

More interestingly, the relaxed value restriction becomes useful if we fully apply `map`, a case where eta-expansion cannot be used.

```
let l = List.map (fun id -> id) []
val l : 'a list
```

Note that all the examples presented in this section cannot be handled by rank-1 polymorphism [13]. This is not necessarily the case for examples in the next section, but this suggests that improvements by both methods are largely orthogonal.

While our improvements are always conceptually related to the notion of empty container, we will see in the following examples that it can show up in many flavors, and that in some cases we are talking about concrete values, rather than empty ones.

4. APPLICATION EXAMPLES

In this section, we give examples of the different problems described in the introduction, and show how we improve their typings.

4.1 Constructor and accessor functions

In ML, we can construct values with data constructors and extract them with pattern matching.

```
let empty2 = ([],[])
val empty2 : 'a list * 'b list = ([], [])
let (_,l2) = empty2
val l2 : 'a list = []
```

As you can see here, since neither operations use functions, the value restriction does not come in the way, and we obtain a polymorphic result. However, if we use a function as accessor, we loose this polymorphism.

```
let l2 = snd empty2
val l2 : '_a list = []
```

Moreover, if we define custom constructors, then polymorphism is lost in the original data itself. Here `pair` assists in building a Lisp-like representation of tuples.

```
let pair x y = (x, (y, ()))
val pair : 'a -> 'b -> 'a * ('b * unit)
let empty2' = pair [] []
val empty2' : '_a list * ('_b list * unit) = (..)
```

If we need to use such values in a polymorphic context, the only workaround allowed by the value restriction is to make them functions by adding a dummy parameter, and recompute them at every usage site.

```
let empty2' () = pair [] []
val empty2' : unit -> 'a list * ('b list * unit)
let l1 () = fst (empty2' ())
val l1 : unit -> 'a list
```

This causes extra computation, and is valid only if the constructor/accessor has no side-effects. A classical side-effect would be to add a counter for the number of cons-cells created.

```
let count = ref 0
val count : int ref
let pair x y = count := !count + 2; (x, (y, ()))
val pair : 'a -> 'b -> 'a * ('b * unit)
```

If the parameters to the constructor have covariant types, then the relaxed value restriction solves all these problems.

```
let l2 = snd empty2
val l2 : 'a list = []
let empty2' = pair [] []
val empty2' : 'a list * ('b list * unit) = (..)
```

This extra polymorphism allows one to share more values throughout a program.

4.2 Abstract datatypes

This problem is made more acute by abstraction. Suppose we want to define an abstract datatype for bounded length lists. This can be done with the following signature:

```
module type BLIST = sig
  type +'a t
  val empty : int -> 'a t
  val cons : 'a -> 'a t -> 'a t
  val list : 'a t -> 'a list
end
module Blist : BLIST = struct
  type 'a t = int * 'a list
  let empty n =
    (n, [])
  let cons a (n, l) =
    if n > 0 then (n-1, a::l)
    else raise (Failure "Blist.cons")
  let list (n, l) =
    l
end
```

The interesting question is what happens when we use `empty`. Using the value restriction, one would obtain:

```
let empty5 = Blist.empty 5
val empty5 : '_a Blist.t = <abstract>
```

Since the type variable is monomorphic, we cannot reuse this `empty5` as *the* empty 5-bounded list; we have to create a new empty list for each different element type. And this time, we cannot get the polymorphism by building the value directly from data constructors, as abstraction has hidden the type's structure.

Just as for the previous example, relaxed valued restriction solves the problem: since `'_a` is not dangerous in `'_a Blist.t`, we shall be able to generalize it.

```
val empty5 : 'a Blist.t = <abstract>
```

With the relaxed value restriction, abstract constructors can be polymorphic as long as their type variables are covariant inside the abstract type.

Note that, at first sight, it may seem that the variance annotations, which we need to recover polymorphism here, are breaking the implementation abstraction of the `Blist` module. Actually, this is not so : variance annotations on abstract datatypes do not restrain the representation itself, but only the way it can be used. To see that, we may consider the implementation of covariant vectors on top of mutable arrays.

```
type +'a vector = {get: int -> 'a; length: int}
let make len f =
  let arr =
    if len = 0 then [||]
    else Array.create len (f 0) in
  for i = 1 to len-1 do arr.(i) <- f i done;
  {get=Array.get arr; length=len}
val make : int -> (int -> 'a) -> 'a vector
let map f vect =
  make vect.length (fun i -> f (vect.get i))
val map : ('a -> 'b) -> 'a vector -> 'b vector
```

This situation is to be compared with imperative type variables, or equality type variables, whose specificity must be propagated through any definition they are used in, making it impossible to abstract from the implementation.

4.3 Object constructors

As one would expect from its name, Objective Caml sports object-oriented features. Programmers are often tempted by using classes in place of algebraic datatypes. A classical example is the definition of lists.

```
class type ['a] list = object
  method empty : bool
  method hd : 'a
  method tl : 'a list
end
class ['a] nil : ['a] list = object
  method empty = true
  method hd = raise (Failure"hd")
  method tl = raise (Failure"tl")
end
class ['a] cons a b : ['a] list = object
  method empty = false
  method hd = a
  method tl = b
end
```

This looks all nice, until you realize that you cannot create a polymorphic empty list:

```
let nil : 'a list = new nil
val nil : '_a list = <obj>
```

Again, as `'a` is covariant in `'a list`, this type variable is generalizable, and we can now infer a polymorphic type.

```
val nil : 'a list = <obj>
```

We are of course restricted to objects with only covariant methods: if you add a method `cons : 'a -> 'a list`, this `'a` will be dangerous in the class type, and we cannot relax the value restriction anymore. This is unfortunate as this method is not expected to change the state of the object itself, but to create a new one. Yet we have no way to know that without looking at the implementation. A workaround is to define such methods outside of the object, as functions, just like abstract datatypes.

```
let cons a l : 'a list = new cons a l
val cons : 'a -> 'a list -> 'a list
let nilist = cons nil nil
val nilist : 'a list list = <obj>
```

4.4 Polymorphic variants

Polymorphic variants [2, 3] are another specific feature of Objective Caml. Their design itself contradicts the assumption that polymorphic data structures are rare in ML programs: by definition a polymorphic variant can belong to any type that includes its tag.

```
let one = `Int 1
val one : [> `Int of int] = `Int 1
let two = `Int (1+1)
val two : _[> `Int of int] = `Int 2
```

Again the value restriction gets in our way: it's enough that the argument is not a value to make the variant constructor monomorphic (as shown by the "_" in front of the type). And of course, any variant returned by a function will be given a monomorphic type. This means that in all previous examples, you can replace the empty list by any polymorphic variant, and the same problem will appear.

Again, we can use our coercion principle[1]:

```
let two = (two : [`Int of int] :> [> `Int of int])
val two : [> `Int of int] = `Int 2
```

This makes using variants in multiple contexts much easier. Polymorphic variants profit considerably from this improvement. One would like to see them simply as the dual of polymorphic records (or objects), but the value restriction has broken the duality. For polymorphic records, it is usually enough to have polymorphism of functions that accept a record, but for polymorphic variants the dual would be polymorphism of variants themselves, including results of computations, which the value restriction did not allow. While Objective Caml allowed polymorphism of functions that accept a variant, there were still many cases where one had to use explicit subtyping, as the same value could not be used in different contexts by polymorphism alone. For instance consider the following program:

```
val all_results :
  [ `Bool of bool | `Float of float | `Int of int]
  list ref
val num_results :
  [ `Float of float | `Int of int] list ref
let div x y =
  if x mod y = 0 then `Int (x/y)
  else `Float (float x /. float y)
val div : int -> int ->
  [> `Float of float | `Int of int]
let comp x y =
    let z = div x y in
    all_results := z :: !all_results;
    num_results := z :: !num_results
val comp : int -> int -> unit
```

Since `all_results` and `num_results` are toplevel references, their types must be ground. With the strict value restriction, `z` would be given a monomorphic type, which would have to be equal to the types of both references. Since the references have different types, this is impossible. With the relaxed value restriction, `z` is given a polymorphic type, and distinct instances can be equal to the two reference types.

4.5 Semi-explicit polymorphism

Since version 3.05, Objective Caml also includes an implementation of semi-explicit polymorphism [4], which allows the definition of polymorphic methods in objects.

The basic idea of semi-explicit polymorphism is to allow universal quantification anywhere in types (not only in the prefix), but to restrict instantiation of these variables to cases where the first-class polymorphism is *known* at the instantiation point. To obtain a principal notion of *knowledge*, quantified types are marked by type variables (which are only used as *markers*), and a quantified type can only be instantiated when its marker variable is generalizable. Explicit type annotations can be used to force markers to be polymorphic.

[1] `zero` amounts here to an empty variant type, and if we show the internal row extension variables the coercion would be `(two : [`Int of int | zero] :> [`Int of int | 'a])`, meaning that in one we case we allow no other constructor, and in the other case we allow any other constructor.

$$\textsc{Intro}\ \frac{\Gamma \vdash e : \sigma_1 \quad (\sigma_1 : \sigma : \sigma_2)}{\Gamma \vdash [e : \sigma] : [\sigma_2]^\epsilon} \qquad \textsc{Elim}\ \frac{\Gamma \vdash e : \forall \epsilon.[\sigma]^\epsilon}{\Gamma \vdash \langle e \rangle : \sigma}$$

$$\textsc{Ann}\ \frac{\Gamma \vdash e : \tau_1 \quad (\tau_1 : \tau : \tau_2)}{\Gamma \vdash (e : \tau) : \tau_2}$$

Figure 3: Rules for semi-explicit polymorphism

The type system only adds three rules to the classical Damas-Milner presentation of ML-polymorphism [1] (see figure 3). You can wrap a polymorphic value with its type $[e : \sigma]$, and unwrap it $\langle e \rangle$ without explicit type information (hence the "semi-explicit" naming). In these rules, ϵ is a special kind of type variables, used only as markers, but with exactly the same generalization and instantiation rules as usual type variables, and $(\sigma_1 : \sigma : \sigma_2)$ gives semantics to type annotations, and stands for the existence of type variable vectors $\bar{\alpha}, \bar{\alpha}_1, \bar{\epsilon}, \bar{\epsilon}_1, \bar{\epsilon}_2$ such that $\sigma_1 = (\sigma[\bar{\epsilon}_1/\bar{\epsilon}])[\bar{\alpha}_1/\bar{\alpha}]$ and $\sigma_2 = (\sigma[\bar{\epsilon}_2/\bar{\epsilon}])[\bar{\alpha}_1/\bar{\alpha}]$. *i.e.* usual type variables must be instantiated identically in both σ_1 and σ_2, but markers can be instantiated differently, allowing the result to be quantified even if the markers in the argument's type appear in the environment.

We will not explain here in detail how this system works, but the base line is that inferred polymorphism can be used to enforce principality. While this idea works very well with the original Hindley-Milner type system, treating marker variables according to the value restriction would simply force the use of a type annotation when instantiating any explicitly polymorphic function result. This would be clearly self-defeating. Recognizing that in this case markers are completely unrelated to soundness, and are only used to ensure principality of the type system, a possibility (formalized in [4] and used in Objective Caml 3.05) is to generalize these markers even in non-value bindings. However, this also means replacing full blown principality by a notion of principality among maximal derivations, which is a weaker property.

We demonstrate here Objective Caml's behavior, where elimination is implicit at method call sites.

```
class id : object method id : 'a. 'a -> 'a end
let f (x : id) = (x#id 1, x#id true)
val f : id -> int * bool = <fun>
let h () = let x = new id in (x#id 1, x#id true)
val h : unit -> int * bool = <fun>
```

`f` is a valid use of polymorphism: the annotation is on the binding of `x` and can be propagated to all its uses. `h` would not be accepted under the strict value restriction, because marker variables in the type of `x` would not be generalizable. It is only allowed thanks to the above mentioned marker-generalization trick.

By using our scheme of generalizing type variables that do not appear in dangerous positions, we can recover full principality, with all its theoretical advantages, and accept `h` "officially".

Note also that since these markers may appear in types that otherwise have no free type variables, this boosts the

number of data structures containing polymorphic (marker) variables. That is, semi-explicit polymorphism completely invalidates the assumption that polymorphic values that are not functions are rare and not essential to ML programming.

4.6 Impact on real programs

When trying to assess how the relaxed value restriction impacts on real programs, we find ourselves in just the opposite situation to the introduction of the value restriction. The value restriction had evident practical advantages, but was able to type less programs than the systems used before. So it was enough to check that existing programs could be typed with few modifications.

Since the relaxed value restriction can in theory type all programs typable with the value restriction, such a test has little meaning. In practice, there is a potential incompatibility, as there are places in Objective Caml's type system, at the compilation unit level and at the class level, where free type variables are not allowed, and improving polymorphism (a good thing!) may sometime leave free a variable previously instantiated. Fortunately, in the hundreds of thousands code lines that were recompiled since this change was commited, I am only aware of this occurring once, and it was immediately fixed by a type annotation.

We might also try to compare with the systems described in section 2. However, using old benchmarks, like those defined in [16] or [10], yields the same results for both strict and relaxed value restriction, as they do not use empty collections. Differences only appear with examples combining the use of references and empty collections, like we did in section 3. To our knowledge, our second example cannot be typed in any of those previous works, as they do not allow the presence of polymorphic values in the store.

What one really wants to know is how often this new feature allows to write programs that could not be typed with the strict value restriction. This check can only be done on new code, as by definition old code was typable by the value restriction (or we would have to dig code written before the introduction of the value restriction, such code being incompatible with the Objective Caml syntax). Since the relaxed value restriction was introduced in Objective Caml sources about 6 months ago, and there has been no public release since this addition, the amount of new code is rather limited.

Yet, I have regularly reports by colleagues that some new code I wrote (unintentionally) doesn't compile with ocaml 3.06 anymore. This is mostly the result of toplevel data definitions having a polymorphic variant type, as in section 4.4. In simple cases one can return to the value restriction by lifting all non-value subterms from the definition. For instance we can rewrite the second example of section 4.4 as:

```
let i2 = 1 + 1
val i2 : int = 2
let two = `Int i2
val two : [> `Int of int] = `Int 2
```

But if such a lifting cannot be done, finding a workaround can imply changing the uses of the definition too.

Examples with abstract datatypes and object constructors do not appear unintentionally. But they shall become common once programmers get used to this feature. Experience should tell us, but I do not believe that the relaxed value restriction makes errors harder to understand to programmers than the strict value restriction: thanks to it you end up having less errors! The rule itself is simple enough, and you only need to learn it when you want to extract the extra polymorphism.

Last, the case of semi-explicit polymorphism shows a distinct advantage: if we had applied the strict value restriction to it in Objective Caml 3.05, without the not-so-satisfactory trick we explain in section 4.5, then almost no program would be typable: calling a polymorphic method on a freshly created object would not be typable! The relaxed value restriction avoids this, and from this point of view, this is an essential improvement.

5. FORMALIZATION AND TYPE SYSTEM

In this section we fully formalize our language, and propose a type system where the extra polymorphism described in previous examples is recovered automatically (without the need for explicit coercions). Yet this type system, which we call the *relaxed value restriction*, enjoys the principal type property.

We base ourselves on Wright and Felleisen's formalization of Reference ML [20]. For our results to be meaningful, we need to handle more varied data, so we also add pairs and lists, as they do not incur any difficulty in typing.

Expressions distinguish between values and non-values. The store is introduced by the $\rho\theta.e$ binder and is handled explicitly. Two kinds of contexts are defined for reduction rules: R-contexts, used in store operations, and E-contexts, in evaluation.

$$\begin{array}{rcl} e & ::= & v \mid e_1\ e_2 \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \mid \rho\theta.e \\ v & ::= & x \mid \mathsf{Y} \mid \lambda x.e \mid \mathsf{ref} \mid\ ! \mid := \mid :=\ v \\ & \mid & (v,v) \mid \pi_1 \mid \pi_2 \mid \mathsf{nil} \mid \mathsf{cons}\ v \mid \mathsf{uncons}\ v\ v \\ \theta & ::= & \{\langle x, v\rangle\}^* \end{array}$$

$$\begin{array}{rcl} R & ::= & [\,] \mid R\ e \mid v\ R \mid \mathsf{let}\ x = R\ \mathsf{in}\ e \\ E & ::= & [\,] \mid E\ e \mid v\ E \mid \mathsf{let}\ x = E\ \mathsf{in}\ e \mid \rho\theta.E \end{array}$$

As in Reference ML, both $:=$ and $:= v$ are values, reflecting the fact $:=$ can only be reduced when given two arguments.

Reduction rules are given in figure 4. They are those of Reference ML, with a few innocuous additions. We define one-step reduction as $E[e] \to E[e']$ whenever $e \to e'$, and multi-step reduction as $e_1 \xrightarrow{*} e_n$ whenever $e_1 \to e_2 \cdots \to e_n$. Reduction does not produce badly-formed expressions.

PROPERTY 1. *If e is a well-formed expression (i.e. no non-value appears at a value position), and $e \to e'$, then e' is well-formed.*

PROOF. We assume $e = E[e_0]$, $e' = E[e_0']$ and $e \to e'$ by direct application of a rule. We have to check the well-formedness of e_0'. For (β_v) and (let), this comes from the closedness of values under substitution: for any value v' inside e_0, by induction on the structure of values $v'[v/x]$ is also a value. For all other reduction rules, this is immediate. Finally $E[e_1]$ is well-formed for any well-formed expression e_1, so $E[e_0']$ is well-formed. □

Types are the usual monotypes and polytypes.

$$\begin{array}{rcl} \tau & ::= & \alpha \mid \tau\ \mathtt{ref} \mid \tau \times \tau \mid \tau\ \mathtt{list} \\ \sigma & ::= & \tau \mid \forall\bar{\alpha}.\tau \end{array}$$

An instantiation order $\succ$ is defined on polytypes by $\forall\bar{\alpha}.\tau \succ \forall\bar{\beta}.\tau'$ iff $\bar{\beta} \cap FTV(\forall\bar{\alpha}.\tau) = \emptyset$ and there is a vector $\bar{\tau}$ of monotypes such that $[\bar{\tau}/\bar{\alpha}]\tau = \tau'$.

We type this language using typing rules in figure 5. Those rules are again taken from Reference ML, assuming all type variables to be imperative (which is equivalent to applying the value restriction, *cf* [19] page 6). The only exception is the LET$_e$ rule, which generalizes some variables. In the value case, $Close(\tau_1, \Gamma) = \forall FTV(\tau_1) \setminus FTV(\Gamma).\tau_1$ as usual, but in the non-value case we still generalize safe variables:

$$
\begin{array}{lrcll}
(\beta_v) & (\lambda x.e)\ v & \to & e[v/x] & \\
(let) & \mathsf{let}\ x = v\ \mathsf{in}\ e & \to & e[v/x] & \\
(\mathsf{Y}) & \mathsf{Y}\ v & \to & v\ (\lambda x.\mathsf{Y}\ v\ x) & \\
(ref) & \mathsf{ref}\ v & \to & \rho\langle x, v\rangle.x & \\
(deref) & \rho\theta\langle x, v\rangle.R[!\ x] & \to & \rho\theta\langle x, v\rangle.R[v] & \\
(assign) & \rho\theta\langle x, v_1\rangle.R[:=\ x\ v_2] & \to & \rho\theta\langle x, v_2\rangle.R[v_2] & \\
(\rho_{merge}) & \rho\theta_1.\rho\theta_2.e & \to & \rho\theta_1\theta_2.e & \\
(\rho_{lift}) & R[\rho\theta.e] & \to & \rho\theta.R[e] & \text{if } R \neq [\,] \\
(\rho_{drop}) & \rho\theta.e & \to & e & \text{if } dom(\theta) \cap FV(e) = \emptyset \\
(\pi_1) & \pi_1\ (v_1, v_2) & \to & v_1 & \\
(\pi_2) & \pi_2\ (v_1, v_2) & \to & v_2 & \\
(uncons_1) & \mathsf{uncons}\ v_1\ v_2\ \mathsf{nil} & \to & v_1\ \mathsf{nil} & \\
(uncons_2) & \mathsf{uncons}\ v_1\ v_2\ (\mathsf{cons}\ v) & \to & v_2\ v &
\end{array}
$$

Figure 4: Reduction rules

$$
\textsc{Var}\ \frac{\Gamma(x) \succ \tau}{\Gamma \vdash x : \tau} \qquad
\textsc{App}\ \frac{\Gamma \vdash e_1 : \tau_2 \to \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau_1} \qquad
\textsc{Abs}\ \frac{\Gamma[x \mapsto \tau_1] \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}
$$

$$
\textsc{Let}_v\ \frac{\Gamma \vdash v : \tau_1 \quad \Gamma[x \mapsto \mathit{Close}(\tau_1, \Gamma)] \vdash e : \tau_2}{\Gamma \vdash \mathsf{let}\ x = v\ \mathsf{in}\ e : \tau_2} \qquad
\textsc{Pair}\ \frac{\Gamma \vdash v_1 : \tau_1 \quad \Gamma \vdash v_2 : \tau_2}{\Gamma \vdash (v_1, v_2) : \tau_1 \times \tau_2}
$$

$$
\textsc{Let}_e\ \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma[x \mapsto \mathit{CovClose}(\tau_1, \Gamma)] \vdash e_2 : \tau_2}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2} \qquad
\textsc{Cons}\ \frac{\Gamma \vdash v : \tau \times \tau\ \mathtt{list}}{\Gamma \vdash \mathsf{cons}(v) : \tau\ \mathtt{list}}
$$

$$
\textsc{Rho}\ \frac{\Gamma[x_j \mapsto \tau_j\ \mathtt{ref}]_1^n \vdash e : \tau \quad \Gamma[x_j \mapsto \tau_j\ \mathtt{ref}]_1^n \vdash v_i : \tau_i \quad (1 \le i \le n)}{\Gamma \vdash \rho\langle x_1, v_1\rangle \ldots \langle x_n, v_n\rangle.e : \tau}
$$

$$
\begin{array}{l}
\textsc{Axioms} \\
\Gamma \vdash \mathsf{Y} : ((\tau_1 \to \tau_2) \to \tau_1 \to \tau_2) \to \tau_1 \to \tau_2 \\
\Gamma \vdash \mathsf{ref} : \tau \to \tau\ \mathtt{ref} \qquad \Gamma \vdash\ ! : \tau\ \mathtt{ref} \to \tau \qquad \Gamma \vdash\ := : \tau\ \mathtt{ref} \to \tau \to \tau \\
\Gamma \vdash \pi_1 : \tau_1 \times \tau_2 \to \tau_1 \qquad \Gamma \vdash \pi_2 : \tau_1 \times \tau_2 \to \tau_2 \qquad \Gamma \vdash \mathsf{nil} : \tau\ \mathtt{list} \\
\Gamma \vdash \mathsf{uncons} : (\tau_1\ \mathtt{list} \to \tau_2) \to (\tau_1 \times \tau_1\ \mathtt{list} \to \tau_2) \to \tau_1\ \mathtt{list} \to \tau_2
\end{array}
$$

Figure 5: Typing rules

$$
\textsc{b-Var}\ \frac{t \in \Gamma(x)}{\Gamma \models x : t} \qquad
\textsc{b-Let}\ \frac{(\forall t \in s)\, \Gamma \models v : t \quad \Gamma[x \mapsto s] \models e : t'}{\Gamma \models \mathsf{let}\ x = v\ \mathsf{in}\ e : t'} \qquad
\textsc{b-Sub}\ \frac{\Gamma \models e : t \quad t \le t'}{\Gamma \models e : t'}
$$

$$
\textsc{b-App}\ \frac{\Gamma \models e_1 : \tau_2 \to t_1 \quad \Gamma \models e_2 : t_2}{\Gamma \models e_1\ e_2 : t_1} \qquad
\textsc{b-Abs}\ \frac{\Gamma[x \mapsto \{t_1\}] \models e : t_2}{\Gamma \models \lambda x.e : t_1 \to t_2}
$$

$$
\textsc{b-Ref}\ \Gamma \models \mathsf{ref} : t \to t\ \mathtt{ref} \qquad
\textsc{b-Deref}\ \Gamma \models\ ! : t\ \mathtt{ref} \to t \qquad
\textsc{b-Assign}\ \Gamma \models\ := : t\ \mathtt{ref} \to t \to t
$$

$$
\textsc{b-Rho}\ \frac{\Gamma[x_j \mapsto \{t_j\ \mathtt{ref}\}]_1^n \models e : t \quad \Gamma[x_j \mapsto \{t_j\ \mathtt{ref}\}]_1^n \models v_i : t_i \quad (1 \le i \le n)}{\Gamma \models \rho\langle x_1, v_1\rangle \ldots \langle x_n, v_n\rangle.e : t}
$$

Figure 6: Typing rules for $\mathbf{B}(T)$

$CovClose(\tau_1, \Gamma) = \forall FTV(\tau_1) \setminus V^-(\tau_1) \setminus FTV(\Gamma).\tau_1$, with V^- the set of dangerous variables defined in figure 2. The definition of V^- captures more variables than the usual definition of contravariant occurrences. We deem dangerous all occurrences appearing in a contravariant branch of a type. While this is not necessary to ensure type soundness, we need it to keep principality of type inference. For instance, consider the following function.

$$\mathsf{let}\ f = \mathsf{let}\ r = \mathsf{ref\ nil}\ \mathsf{in}\ \lambda k.\mathsf{Y}\ (\lambda f.f)\ !r$$

As the type of $\mathsf{Y}\ (\lambda f.f)$ is $\forall\alpha\beta.\alpha \to \beta$, we expect the principal type of f to be $\forall\beta.\gamma \to \beta$, with γ a non generalizable variable. However, if we were to generalize covariant variables at ranks higher than 0, then $\forall\beta\delta.(\delta \to \gamma) \to \beta$ would be another acceptable type for `f`, and none of the two is an instance of the other. *i.e.* we would have lost principality.

We include the Rho typing rule for completeness, but we cannot use it to obtain subject reduction. We can see it on the following example[2].

$$\begin{array}{rl} & \mathsf{let}\ f = (\mathsf{let}\ r = \mathsf{ref\ nil}\ \mathsf{in}\ \lambda x.!r)\ \mathsf{in} \\ & \quad (\mathsf{cons}(\mathsf{nil}, f\ \mathsf{nil}), \mathsf{cons}(\mathsf{ref\ nil}, f\ \mathsf{nil})) \\ \to & \rho\langle r, \mathsf{nil}\rangle.(\mathsf{cons}(\mathsf{nil}, (\lambda x.!r)\ \mathsf{nil}), \mathsf{cons}(\mathsf{ref\ nil}, (\lambda x.!r)\ \mathsf{nil})) \end{array}$$

In the first line, f can be given the polymorphic type $\forall\alpha.\ \beta\ \texttt{list} \to \alpha\ \texttt{list}$, with β a non-generalized type variable. When we apply f to nil we may get any list. The type of the whole expression is $(\tau_1\ \texttt{list list} \times \tau_2\ \texttt{list ref list})$. However, after reduction, r can only be given a monomorphic type, and its two occurrences appear in incompatible type contexts.

If you think that the problem is anecdotical, and that it can be solved for instance by adding polymorphic type information to the store, or even by more extensive changes like making ref a two-parameter type (one covariant, one contravariant), then try replacing the definition of f in the above example by the identically typed

$$\begin{array}{l} \mathsf{let}\ r = \mathsf{ref\ nil}\ \mathsf{in} \\ \lambda x.(\lambda y.\lambda z.\mathsf{let}\ u = !y\ \mathsf{in}\ (:=\ y\ (z\ y)\ ;\ :=\ y\ u\ ;\ u))\ r\ (\lambda x.\mathsf{nil}) \end{array}$$

and consider the typing needed for $\rho\langle r, \mathsf{nil}\rangle.\mathsf{let}\ f = \lambda x.(\lambda y.\lambda z.\ \cdots)\ r\ (\lambda x.\mathsf{nil})\ \mathsf{in}\ e$, where e uses f polymorphically. What this example shows is that this is not enough to be able to extract polymorphic values from references, we need a way to propagate this polymorphism to the type of f after reduction.

In the absence of subject reduction, we must prove type soundness in an indirect way. We will do this in the next section by translation into another type system, which has implicit subtyping. We also prove subject reduction in appendix A, but for a different reduction system, which introduces coercions. We believe that an appropriate form of subsumption (direct or indirect) is essential to proofs of subject reduction for type systems validating our Let_e rule.

On the other hand, principality is a static property of terms, and we can prove it easily by trivially modifying the inference algorithm *W*, using *CovClose* in place of *Close* for non-values. This is clearly sound: this is our rule. This is also complete: *CovClose* is monotonous with respect to the instantiation order $\succ$, that is

$$CovClose(\tau, \Gamma) \succ CovClose(S(\tau), S(\Gamma))$$

for any type substitution S.

[2]For sake of conciseness we use pairs of expressions, rather than an expanded form where pairs contain only values; and we write $e_1\ ;\ e_2$ as a shorthand for $\mathsf{let}\ i = e_1\ \mathsf{in}\ e_2$ (i fresh). This has no impact on typing.

Property 2 (principality). *If, for a given pair* (Γ, e) *there is a* τ_0 *such that* $\Gamma \vdash e : \tau_0$ *is derivable, then there exists a* σ *such that for any* τ, $\Gamma \vdash e : \tau$ *iff* $\sigma \succ \tau$.

We can also verify a partial form of subject reduction, limited to non side-effecting reductions, but allowing those reductions to happen anywhere in a term. While insufficient to prove type soundness, this property is useful to reason about program transformations.

$$\begin{array}{rcl} C & ::= & [\,] \mid C\ e \mid e\ C \mid \mathsf{let}\ x = C\ \mathsf{in}\ e \mid \mathsf{let}\ x = e\ \mathsf{in}\ C \\ & \mid & \rho\theta\langle x, C\rangle.e \mid \rho\theta.C \mid \lambda x.C \mid (C, v) \mid (v, C) \end{array}$$

Property 3 (partial subject reduction). *Non side-effecting reductions, i.e. rules* $(\beta_v), (let), (\mathsf{Y}), (\pi_i), (uncons_i)$ *preserve typing: for any context* C, *if* $\Gamma \vdash C[e] : \tau$ *and* $e \to_f e'$, *then* $\Gamma \vdash C[e'] : \tau$.

The proof can be easily transposed from any proof of subject reduction for applicative ML. We only need to verify that the substitution lemma still holds in presence of our distinction between Let_v and Let_e.

Lemma 4 (substitution). *If* $\Gamma[x \mapsto \sigma_1] \vdash e : \tau$ *and* $\Gamma \vdash v : \tau_1$ *and* $Close(\tau_1, \Gamma) \succ \sigma_1$, *then* $\Gamma \vdash e[v/x] : \tau$.

Proof. The proof is by induction on the length of the derivation and case analysis on the last rule used.
Case Let_e. If $\Gamma[x \mapsto \sigma_1] \vdash \mathsf{let}\ x' = v'\ \mathsf{in}\ e : \tau$ using Let_v, then there is a derivation $\Gamma[x \mapsto \sigma_1] \vdash v' : \tau'$. By induction hypothesis, after substitution, $\Gamma \vdash v'[v/x] : \tau'$ holds, and since values are closed under substitution, $v'[v/x]$ is still a value. Since $FTV(\Gamma) \subset FTV(\Gamma) \cup FTV(\sigma_1)$, $Close(\tau', \Gamma) \succ Close(\tau', \Gamma[x \mapsto \sigma_1])$, so that $\Gamma[x \mapsto Close(\tau', \Gamma)] \vdash e[v/x] : \tau$ is derivable, and $\Gamma \vdash \mathsf{let}\ x' = v'[v/x]\ \mathsf{in}\ e[v/x] : \tau$ by Let_v.
Case Let_v. If $\Gamma[x \mapsto \sigma_1] \vdash \mathsf{let}\ x' = e'\ \mathsf{in}\ e : \tau$ using Let_e, then there is a derivation $\Gamma[x \mapsto \sigma_1] \vdash e' : \tau'$. By induction hypothesis, after substitution, $\Gamma \vdash e'[v/x] : \tau'$ holds, and $e'[v/x]$ is not a value. Again $CovClose(\tau', \Gamma) \succ CovClose(\tau', \Gamma[x \mapsto \sigma_1])$, so that $\Gamma[x \mapsto CovClose(\tau', \Gamma)] \vdash e[v/x] : \tau$ is derivable, and $\Gamma \vdash \mathsf{let}\ x' = e'[v/x]\ \mathsf{in}\ e[v/x] : \tau$ by Let_e.
Other cases are all simple and standard. □

6. SEMI-SYNTACTIC TYPE SOUNDNESS

The short path to prove the type soundness of our system is to work in a system providing the desired subtyping. Fortunately there appears to be such a system, including ML polymorphism, imperative operations, and subtyping. This is Pottier's B(T) [14]. It was originally developed as an intermediate step in the proof of type soundness for HM(X), a constraint-based polymorphic type system [12]. B(T) is particular by its extensional approach to polymorphism: polytypes are not expressed syntactically, but as (possibly infinite) sets of ground monotypes.

We give here a condensed account of the definition of B(T), which should be sufficient to understand how a typing derivation in our system can be mapped to a typing derivation in an instance of B(T).

The T in B(T) represents a universe of monotypes, equipped with a subtyping relation $\leq$, serving as parameter to the type system. Monotypes in T are denoted by t. $\to$ should be a total function from $T \times T$ into T, such that $t_1 \to t_2 \leq t_1' \to t_2$ implies $t_1' \leq t_1$ and $t_2 \leq t_1$. ref should be a total function from T to T, such that $t\ \texttt{ref} \leq t'\ \texttt{ref}$ implies $t = t'$. Moreover $t_1 \to t_2 \leq t\ \texttt{ref}$ and $t\ \texttt{ref} \leq t_1 \to t_2$ should both be false for any t, t_1, t_2 in T. Polytypes s are upward-closed subsets of T (*i.e.* if $t \in s$ and $t \leq t'$ then $t' \in s$).

The terms and reduction rules in B(T) are identical to those in our system (excluding pairs and lists). While Pottier's presentation uses a different syntax for representing

and updating the store, both presentations are equivalent, ours requiring only more reduction steps. We will stick to our presentation.

Typing judgments are written $\Gamma \models e : t$ with Γ a polytype environments (mapping identifiers to upward-closed sets of monotypes) and t a monotype. Typing rules[3] are given in figure 6. They are very similar to ours, you just have to transpose all τ's into t's and all $\vdash$ into $\models$. The only changes are the removal of LET$_e$ (this is the strict value restriction), the addition of B-SUB, and a semantical handling of polymorphism in B-VAR and B-LET.

The following theorem is proved in [14], section 3, for any $(T, \leq)$ satisfying the above requirements.

THEOREM 5 (SUBJECT REDUCTION). *If $e \to e'$, where e, e' are closed, then $\Gamma \models e : t$ implies $\Gamma \models e' : t$.*

For our purpose, we choose T as the set of all types generated by the type constructors zero, int, $\to$, ref, $\times$, list and the set of all type variables $\{\alpha, \beta, \dots\}$. The variables are introduced here as type constants, to ease the translation, but they are unrelated to polymorphism: there is no notion of variable quantification in B(T). zero is an extra type constructor, which need not be included in our original language. The subtyping relation is defined as $\mathtt{zero} \leq t$ and $t \leq t$ for any t in T, and extended through constructors, all covariant in their parameters, except ref which is nonvariant, and $\to$ which is contravariant in its first parameter and covariant in its second one. This conforms to the requirements for B(T), meaning that subject reduction holds in the resulting system. We also extend the language, reduction and typing rules with PAIR, CONS and AXIOMS about Y, pairs and lists. Extending subject reduction to these features presents no challenge; the concerned reader is invited to check this (and other details of formalization), on the remarkably short proof in [14].

The progress lemma depends more directly on the syntax of expressions, and we cannot reuse directly Pottier's proof. However, our reduction and typing rules are basically the same as in [20].

LEMMA 6. *For any closed e, if for all e' such that $e \xrightarrow{*} e'$ there is Γ and t such that $\Gamma \models e' : t$, then reducing e either diverges or leads to a value.*

PROOF. We reuse lemmas 5.5 and 5.6 of [20], extending the definition of faulty expressions with cases implying pairs and lists. Lemma 5.5 (uniform evaluation) does not depend on types, and lemma 5.6 (faulty expressions are untypable) uses only the structure of types, which subsumption does not change. □

Combining the above subject reduction and progress, our instance of B(T) is type-sound.

We present now the translation itself. First we must be able to translate each component of a typing judgment. The expression part uses the following translation.

$$\begin{array}{rcll}
[\![\mathsf{let}\ x = v\ \mathsf{in}\ e]\!] & = & \mathsf{let}\ x = [\![v]\!]\ \mathsf{in}\ [\![e]\!] & \\
[\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2]\!] & = & (\lambda x.\mathsf{let}\ x = x\ \mathsf{in}\ [\![e_2]\!])\ [\![e_1]\!] & \\
[\![e_1\ e_2]\!] & = & [\![e_1]\!]\ [\![e_2]\!] & \\
[\![\lambda x.e]\!] & = & \lambda x.[\![e]\!] & \\
[\![(v_1, v_2)]\!] & = & ([\![v_1]\!], [\![v_2]\!]) & \\
[\![v]\!] & = & v & \text{if } v \text{ has no subterms}
\end{array}$$

As you can see, the only change introduced by the translation is to convert let_e into an applied lambda-abstraction (which is standard), adding a $\mathsf{let}\ x = x$ which we will use in our typing.

Types are translated under a substitution $\xi : V \to T$.

$$\begin{array}{rcl}
[\![\alpha]\!]\xi & = & \xi(\alpha) \\
[\![\tau\ \mathtt{ref}]\!]\xi & = & [\![\tau]\!]\xi\ \mathtt{ref} \\
[\![\tau_1 \times \tau_1]\!]\xi & = & [\![\tau_1]\!]\xi \times [\![\tau_2]\!]\xi \\
[\![\tau\ \mathtt{list}]\!]\xi & = & [\![\tau]\!]\xi\ \mathtt{list}
\end{array}$$

This translation is extended to polytypes and typing environments.

$$[\![\forall\alpha_1 \dots \alpha_n.\tau]\!]\xi = \{t \mid (t_1, \dots, t_n) \in T^n, [\![\tau]\!](\xi[\alpha_1 \mapsto t_1, \dots, \alpha_n \mapsto t_n]) \leq t\}$$

Before going on to translate full derivations, we prove a lemma about the single subsumption step we need.

LEMMA 7. *Let $\bar\alpha$ be a set of type variables that appear only covariantly in τ_1. Let ξ be any translation substitution. Then for any t in $[\![\forall\bar\alpha.\tau_1]\!]\xi$, we have $[\![\tau_1]\!](\xi[\bar\alpha \mapsto \overline{\mathtt{zero}}]) \leq t$.*

PROOF. Let $\bar\alpha = \alpha_1 \dots \alpha_n$ and $\xi' = \xi[\bar\alpha \mapsto \overline{\mathtt{zero}}]$.
By definition of $[\![\forall\bar\alpha.\tau_1]\!]\xi$, there exists $\bar t = t_1 \dots t_n$ such that $[\![\tau_1]\!](\xi[\bar\alpha \mapsto \bar t]) \leq t$.
Since the α_i's only have covariant occurrences, and $\mathtt{zero} \leq t_i$ for all t_i's, we also have $[\![\tau_1]\!]\xi' \leq [\![\tau_1]\!](\xi[\bar\alpha \mapsto \bar t]) \leq t$.
By transitivity of $\leq$ we can conclude that $[\![\tau_1]\!]\xi' \leq t$. □

Finally the derivation is translated by induction on its structure, transforming $\Gamma \vdash e : \tau$ into $[\![\Gamma]\!]\xi \models [\![e]\!] : [\![\tau]\!]\xi$ for any ξ.

- if the last rule applied is LET$_e$ then it is transformed as follows.

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma[x \mapsto \forall\bar\alpha.\tau_1] \vdash e_2 : \tau_2}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2}$$

  maps to

$$\frac{\dfrac{\dfrac{[\![\Gamma[x \mapsto \tau_1]]\!]\xi' \models x : [\![\tau_1]\!]\xi'}{[\![\Gamma[x \mapsto \tau_1]]\!]\xi' \models x : t} \qquad [\![\Gamma]\!]\xi[x \mapsto s] \vdash [\![e_2]\!] : [\![\tau_2]\!]\xi}{[\![\Gamma[x \mapsto \tau_1]]\!]\xi' \models \mathsf{let}\ x = x\ \mathsf{in}\ [\![e_2]\!] : [\![\tau_2]\!]\xi} \quad e_1 : [\![\tau_1]\!]\xi'}{[\![\Gamma]\!]\xi \models (\lambda x.\mathsf{let}\ x = x\ \mathsf{in}\ e_2)\ e_1 : [\![\tau_2]\!]\xi}$$

  where $\bar\alpha = \alpha_1 \dots \alpha_n$, $\xi' = \xi[\bar\alpha \mapsto \overline{\mathtt{zero}}]$, $s = [\![\forall\bar\alpha.\tau_1]\!]\xi$ and t ranges over all elements of s. Note that $[\![\Gamma]\!]\xi' = [\![\Gamma]\!]\xi$ as $\alpha_i \notin FTV(\Gamma)$. The left branch of the derivation is valid by Lemma 7.

- if the last rule applied is LET$_v$ and $\mathit{Close}(\tau_1, \Gamma) = \forall\alpha_1 \dots \alpha_n.\tau_1$, then it is translated to B-LET.

$$\frac{\dfrac{[\![\Gamma]\!]\xi' \models [\![v]\!] : [\![\tau_1]\!]\xi'}{[\![\Gamma]\!]\xi' \models [\![v]\!] : t} \qquad [\![\Gamma]\!]\xi[x \mapsto s] \models [\![e]\!] : [\![\tau_2]\!]\xi}{[\![\Gamma]\!]\xi \models \mathsf{let}\ x = v\ \mathsf{in}\ e : [\![\tau_2]\!]\xi}$$

  where $s = [\![\forall\alpha_1 \dots \alpha_n.\tau_1]\!]\xi$, t ranges over all elements of s, and $\xi' = \xi[\alpha_1 \mapsto t_1, \dots, \alpha_n \mapsto t_n]$ is such that $[\![\tau_1]\!]\xi' \leq t$.

- if the rule applied is VAR, then it is translated to B-VAR.

$$\frac{[\![\tau]\!]\xi \in ([\![\Gamma]\!]\xi)(x)}{[\![\Gamma]\!]\xi \models x : [\![\tau]\!]\xi}$$

- other cases are trivial induction.

From this construction we can obtain the following proposition.

PROPOSITION 8. *If $\Gamma \vdash e : \tau$ is derivable in ML with the relaxed value restriction, then $[\![\Gamma]\!]\xi \models [\![e]\!] : [\![\tau]\!]\xi$ is derivable in $B(T)$ for any ξ.*

[3] In Pottier's presentation, a judgment writes $\Gamma, M \models e : t$; we have merged Γ and M (M only mapping to monotypes), as our syntax for references permits. B-RHO merges B-STORE and B-CONF from the original presentation.

The transformation on terms being trivial, it is easy to show that resulting reductions can simulate original ones.

PROPOSITION 9. *(1) If $e \to e'$ then $[\![e]\!] \xrightarrow{*} [\![e']\!]$ in one or two steps. (2) If $[\![e]\!] \to e'$ then there is a reduction $e \to e''$ such that either $e' = [\![e'']\!]$ or $e' \to [\![e'']\!]$.*

PROOF. In order to translate reduction steps, we define two kinds of translations on evaluation contexts: $[\![E]\!]_v$ when the hole is to be filled by a value, and $[\![E]\!]_e$ when it is to be filled by a non-value. The first translation is simply defined as $[\![E]\!]_v = [\![E[x]]\!][[]/x]$ (x fresh). $[\![E]\!]_e$ is defined by cases, according to the minimal subterm strictly containing the hole. If $E = E_0[\mathsf{let}\ x = []\ \mathsf{in}\ e]$, then $[\![E]\!]_e = [\![E_0]\!]_e[(\lambda x.\mathsf{let}\ x = x\ \mathsf{in}\ [\![e]\!])\ []]$. In all other cases, $[\![E]\!]_e = [\![E]\!]_v$.
To prove (1), we look at the reduction step $e \to e'$. If $e = E[\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2] \to E[\mathsf{let}\ x = v\ \mathsf{in}\ e_2] = e'$, then

$$\begin{array}{rcl} [\![e]\!] & = & [\![E]\!]_e[(\lambda x.\mathsf{let}\ x = x\ \mathsf{in}\ [\![e_2]\!])\ [\![e_1]\!]] \\ & \to & [\![E]\!]_e[(\lambda x.\mathsf{let}\ x = x\ \mathsf{in}\ [\![e_2]\!])\ [\![v]\!]] \\ & \to & [\![E]\!]_e[\mathsf{let}\ x = [\![v]\!]\ \mathsf{in}\ [\![e_2]\!]] = [\![e']\!] \end{array}$$

We have reached e' in two steps. Otherwise, *i.e.* if $e \to e'$ is not of the above form, then $[\![e]\!] \to [\![e']\!]$: the translation is not affected by the reduction step.
We prove (2). If e is irreducible, then $[\![e]\!]$ is also irreducible (the translation does not create new redexes where there is none). If e is reducible, then by analysis of the reduction rules and the definition of evaluation contexts, there is only one possible step: $e \to e''$. Similarly, $[\![e]\!] \to e'$ is the only possible reduction step from $[\![e]\!]$. From (1), $[\![e]\!] \xrightarrow{*} [\![e'']\!]$ in one or two steps. If this is one step, $[\![e'']\!] = e'$. If this is two steps, $e' \to [\![e'']\!]$. □

Now, suppose that we restrict ourselves to closed expressions whose types do not contain references nor function types. Normal forms of such expressions can only be data of the form:

$$d ::= \mathsf{nil} \mid (d, d) \mid \mathsf{cons}\ d$$

For any such normal form, $[\![d]\!] = d$.

From this and type soundness for our instance of B(T) we can deduce the type soundness of ML with the relaxed value restriction, as stated below.

THEOREM 10 (TYPE SOUNDNESS). *If $\emptyset \vdash e : \delta$ with δ any type of the form $\delta ::= \alpha \mid \delta \times \delta \mid \delta$ `list`, then reducing e either diverges or leads to a normal form d, and $\emptyset \vdash d : \delta$.*

7. CONCLUSION

Thanks to a small observation on the relation between polymorphism and subtyping —that `zero` in a covariant position is equivalent to a universally quantified type variable—, we have been able to smooth some of the rough edges of the value restriction, while keeping all of its advantages. This is a useful result, which has already been integrated in the Objective Caml 3.07 compiler. Hopefully this should make the use of polymorphic data structures easier.

We could even do a bit more: use the dual observation, that assuming a "type of all values", `top`, the monomorphic type variables that appear only in contravariant positions are generalizable too. This would have an extra advantage: this should alleviate the principality problem, which had us restrict generalizability to type variables of rank 0. Only variables that appear both in covariant and contravariant position would not be generalizable. We have stopped short of that because generalizing contravariant type variables has little use, and contrary to `zero`, there is no guarantee that such a type can be made available in an implementation, or that it would not break the semantics of the language[4].

$$\begin{array}{rcl} C_V(\alpha) & = & \lambda(x : \mathtt{zero}).\mathsf{Y}\ (\lambda f.f)\ x \qquad \text{if } \alpha \in V \\ C_V(\tau) & = & \lambda x.x \qquad \text{if } V \cap TV(\tau) = \emptyset \\ C_V(\tau_1 \to \tau_2) & = & \lambda x.\lambda y.C_V(\tau_2)\ (x\ (C_V(\tau_1)\ y)) \\ C_V(\tau_1 \times \tau_2) & = & \lambda x.\mathsf{let}\ a = C_V(\tau_1)(\pi_1\ x)\ \mathsf{in} \\ & & \quad \mathsf{let}\ b = C_V(\tau_2)(\pi_2\ x)\ \mathsf{in}\ (a, b) \\ C_V(\tau\ \mathtt{list}) & = & \mathsf{Y}(\lambda c.\mathsf{uncons}\ (\lambda x.\mathsf{nil}) \\ & & \quad (\lambda x.\mathsf{let}\ a = C_V(\tau)(\pi_1\ x)\ \mathsf{in} \\ & & \quad\quad \mathsf{let}\ l = c\ (\pi_2\ x)\ \mathsf{in}\ \mathsf{cons}\ (a, l))) \end{array}$$

Figure 7: Type-directed coercions

Notwithstanding our achievements, this paper does nothing to solve the fundamental problem of the value restriction, namely that by assuming all functions to be imperative, it is overly pessimistic. We have been able to rescue some cases that were probably not even considered when it was introduced. But there is no easy solution for more involved cases, with polymorphic function types in the data.

The triviality of this result brings another question: why wasn't it discovered earlier?

Actually, this specific use of subtyping is not new: the fact has not attracted very much attention, but our LET$_e$ rule is already admissible in HM(X). This could give yet another way to prove type soundness for our system: by defining it as a subsystem of a sufficiently feature-rich instance of HM(X). We preferred B(T) for its robustness, and the lightness of its definition and proof, but this last approach would have the advantage of directness.

APPENDIX

A. TYPABLE REDUCTION

As we have seen in section 5, while our original system is type-sound, it does not enjoy subject reduction. Since the main goal of subject reduction is actually to prove type soundness, one might argue that we don't need it. However, subject reduction also gives insight about the dynamic semantics of the system, and as such is worth considering for its own sake.

Based on the idea presented in section 3, we will state subject reduction using a modified reduction system, where some reduction rules insert type constraints and coercions. Since we do not use the same reduction rules as in section 5, subject reduction for this system is not sufficient to prove the type soundness of ML with the relaxed value restriction.

First, we slightly extend the expression syntax.

$$e ::= \cdots \mid \lambda(x : \tau).e$$

The associated typing rule is

$$\frac{\Gamma[x \mapsto \tau_1] \vdash e : \tau_2 \quad \forall FTV(\tau).\tau \succ \tau_1}{\Gamma \vdash \lambda(x : \tau).e : \tau_1 \to \tau_2}$$

That is, the abstracted variable x is constrained to be typed by a instance of the universally quantified closure of τ.

We also need type-directed coercions, defined in figure 7. The most general scheme for $C_{\bar{\alpha}}(\tau)$ is the expected one:

[4]As a token of the kind of problems which may arise, here is what happens in Objective Caml using the `Obj.repr` function, which can be seen as a coercion to `top` (*aka* `Obj.t`).

```
let l = Array.create 2 (Obj.repr 1.0)
val l :  Obj.t array = [|<abstr>; <abstr>|]
l.(1) <- Obj.repr 1
Segmentation fault
```

$$\emptyset \vdash C_{\bar\alpha}(\tau) : \tau[\overline{\texttt{zero}}/\bar\alpha] \to \tau.$$

Alternatively we could have introduced coercions of the form $(e : \tau[\overline{\texttt{zero}}/\bar\alpha] :> \tau)$ in our calculus. The above type-directed coercions have two advantages: we don't need to extend the calculus, and the soundness of their reduction is immediate from the typing. Furthermore, this shows that we do not need real subtyping, but only an easily simulated form of subsumption.

Reduction rules are now typed: $\Gamma \vdash e \to e'$ means that e can be rewritten in e' if $\Gamma \vdash e : \tau$ for some type τ. There may be extra typing preconditions about Γ and e.
We split (ρ_{lift}) into its different cases.

$$\Gamma \vdash (\rho\theta.e_1)\ e_2 \to \rho\theta.(e_1\ e_2)$$
$$\Gamma \vdash v\ (\rho\theta.e) \to \rho\theta.(v\ e)$$

$$\frac{\Gamma[x_j \mapsto \tau_j\ \texttt{ref}]_1^n \vdash v_i : \tau_i \qquad (0 \le i \le n)}{\Gamma \vdash \mathsf{let}\ x = \rho\theta.v_0\ \mathsf{in}\ e \to \rho\theta.\mathsf{let}\ x = C_{\bar\alpha}(\tau_0)\ v_0\ \mathsf{in}\ e}$$

where $\bar\alpha = FTV(\tau_0) \setminus FTV(\Gamma)$ and $\theta = \langle x_i, v_i\rangle_1^n$. We also need a new rule for typed abstraction, discarding the type:

$$\Gamma \vdash (\lambda(x : \tau).e)\ v \to e[v/x]$$

For all other reduction rules, we just replace $e \to e'$ by $\Gamma \vdash e \to e'$.

Property 11 (Subject Reduction). *If $\Gamma \vdash e : \tau$ is derivable and $\Gamma \vdash e \to e'$ then $\Gamma \vdash e' : \tau$ is derivable.*

Proof. We only have to extend the partial subject reduction proof to the store related rules. We only consider the new (ρ_{lift}) step.
Let's assume that e is $\mathsf{let}\ x = \rho\theta.v_0\ \mathsf{in}\ e_1$ and the conditions for (ρ_{lift}) are satisfied. Let $\tau_i' = \tau_i[\overline{\texttt{zero}}/\bar\alpha]$ and $\Gamma' = \Gamma[x_j \mapsto \tau_j'\ \texttt{ref}]_1^n$. Since $\bar\alpha \cap FTV(\Gamma) = \emptyset$, we have $\Gamma' \vdash v_0 : \tau_0[\overline{\texttt{zero}}/\bar\alpha]$, and for $1 \le i \le n$, $\Gamma' \vdash v_i : \tau_i'$. By the typing of $C_{\bar\alpha}(\tau_0)$, we obtain that $\Gamma' \vdash C_{\bar\alpha}(\tau_0)\ v_0 : \tau_0$.
Using the bound variable convention, $x_1, \ldots, x_n$ do not appear in e_1, so that $\Gamma'[x \mapsto CovClose(\tau_0, \Gamma)] \vdash e_1 : \tau$. Since $\bar\alpha \cap FTV(\Gamma') = \emptyset$ too, $CovClose(\tau_0, \Gamma') = CovClose(\tau_0, \Gamma)$, so that $\Gamma'[x \mapsto CovClose(\tau_0, \Gamma')] \vdash e_1 : \tau$.
From the above conclusions, we can apply $\textsc{Let}_e$ and Rho to deduce $\Gamma \vdash \rho\theta.\mathsf{let}\ x = C_{\bar\alpha}(\tau_0)\ v_0\ \mathsf{in}\ e_1 : \tau$. □

Since this version of the system has subject reduction, should we prefer it to the original one? In our opinion it has at least two drawbacks.

1. Reduction involves types. Since we have to convert a subtyping relation into a parametric polymorphism system, we cannot avoid using type information in the reduction.

2. Reduced terms contain extra information. This makes proving program equalities more complex.

From this viewpoint, the goal of this reduction system is only to give a more direct intuition into how type soundness is preserved, avoiding the *détour* through a stronger type system.

B. REFERENCES

[1] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[2] J. Garrigue. Programming with polymorphic variants. In *ML Workshop*, Baltimore, Sept. 1998.

[3] J. Garrigue. Simple type inference for structural polymorphism. In *The Ninth International Workshop on Foundations of Object-Oriented Languages*, Portland, Oregon, 2002.

[4] J. Garrigue and D. Rémy. Extending ML with semi-explicit higher order polymorphism. *Information and Computation*, 155:134–171, Dec. 1999.

[5] J. Greiner. SML weak polymorphism can be sound. Technical Report CMU-CS-93-160R, Canegie-Mellon University, School of Computer Science, Sept. 1993.

[6] M. Hoang, J. Mitchell, and R. Viswanathan. Standard ML-NJ weak polymorphism and imperative constructs. In *Proc. IEEE Symposium on Logic in Computer Science*, pages 15–25, 1993.

[7] X. Leroy. Polymorphic typing of an algorithmic language. Research report 1778, INRIA, 1992.

[8] X. Leroy. Polymorphism by name for references and continuations. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 220–231, 1993.

[9] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. *The Objective Caml system release 3.06, Documentation and user's manual*. Projet Cristal, INRIA, Aug. 2002.

[10] X. Leroy and P. Weis. Polymorphic type inference and assignment. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 291–302, 1991.

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

[12] M. Odersky, M. Sulzmann, and M. Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999.

[13] A. Ohori and N. Yoshida. Type inference with rank 1 polymorphism for type-directed compilation of ML. In *Proc. International Conference on Functional Programming*. ACM Press, Sept. 1999.

[14] F. Pottier. A semi-syntactic soundness proof for HM(X). Research Report 4150, INRIA, Mar. 2001.

[15] D. Rémy and J. Vouillon. Objective ML: A simple object-oriented extension of ML. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 40–53, Jan. 1997.

[16] J.-P. Talpin and P. Jouvelot. The type and effect discipline. In *Proc. IEEE Symposium on Logic in Computer Science*, pages 162–173, 1992.

[17] M. Tofte. Type inference for polymorphic references. *Information and Computation*, 89:1–34, 1990.

[18] P. Weis, M. Aponte, A. Laville, M. Mauny, and A. Suarez. The CAML reference manual, version 2.6.1. Rapport Technique RT-0121, INRIA, 1990.

[19] A. K. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4), Dec. 1995.

[20] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, Nov. 1994.