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 [1], 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 negligible.

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 [2], as one can only construct them through functions, meaning that they can never hold polymorphic values.
2. Polymorphic variants [3]. 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 _ in `'_a` means that the type variable is not generalized: it can be instantiated only once, and is shared between all uses of `f`. We can replace `'_a` by the base type `zero`, obtaining the type `unit -> zero list`. 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 model-based reasoning cannot be translated into a direct proof: we are aware of no set theoretic model of ML extended with references. Neither can we use a direct syntactic proof, as our system, while being sound, does not enjoy the subject reduction. Nonetheless this intuition will lead us to an indirect proof, by translation into a stronger type system with subtyping.

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 answer some concerns. In section 6 we formalize our language and type system, and prove its soundness using semantic types in section 7, before concluding. Proofs of lemmas can be found in appendix.

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. Busy readers may skip directly to section 3, as the material in this section is only indirectly related to our problem.

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
```

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	$(\alpha \to \beta) \to \alpha\ \texttt{list} \to \beta\ \texttt{list}$

Fig. 1. Comparing types

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 [5] 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 [6], used in Standard ML 90 [7], 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 [8]. 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}". Some further improvements have also been proposed [9].

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 [10] or closure typing by Leroy and Weis [11, 12], 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.

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 [13]. 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 [14]: 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, and 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 [15]: by allowing quantification in non-prenex positions, for instance `int` $\rightarrow \forall\alpha.\alpha \rightarrow \alpha$ `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 (independent) 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 [16]), 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 does—, in subsequent examples. Here is our first example.

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

The type variable `'_a` is not generalized: it will be instantiated when used, and fixed afterwards. This basically means that `l` is now of a fixed type, 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

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

Fig. 2. Dangerous variables

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.

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 6, 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 definition is less general than subtyping, as a covariant type variable with multiple occurences will be kept shared. For instance, subtyping would allow `('_a * '_a) list` to be coerced to `('a * 'b) list`, but type inference will only give the less general `('a * 'a) list`.

Of course, our approach cannot 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. Like with the strict value restriction, we would have to eta-expand to obtain a polymorphic type.

However, 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 [15]. 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) = (..)
```

The classical workaround to obtain a polymorphic type involves eta-expansion, which means code changes, extra computation, and is incompatible with side-effects, for instance if we were to count the number of cons-cells created.

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 + in `type +'a t` 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 `t`. 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.

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.

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 = ...
class ['a] cons a b : ['a] list = ...
```

This looks all nice, until you realize that you cannot create a polymorphic empty list: an object constructor is seen by the type system as a function. Again, as *'a* is covariant in *'a list*, it is generalizable, and the relaxed value restriction allows a polymorphic type.

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

4.4 Polymorphic variants

Polymorphic variants [3, 17] 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:

[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.

```
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*, types containing quantifiers 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.

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, problems appear with the value restriction.

We demonstrate here Objective Caml's behavior. The marker variable ϵ on the type $\texttt{poly}^\epsilon$ is hidden in the surface language.

```
class poly : object method id : 'a. 'a -> 'a end
let f (x : poly) = (x#id 1, x#id true)
val f : poly -> int * bool = <fun>
let h () = let x = new poly 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, *i.e.* the type of `x` is $\forall\epsilon.\texttt{poly}^\epsilon$. But `h` would not be accepted under the strict value restriction, because `new poly` is not a value, so that the type $\texttt{poly}^\epsilon$ of `x` is not generalizable. Since refusing cases like `h` would greatly reduce the interest of type inference, it was actually accepted, arguing that markers have no impact on soundness. A system allowing this is formalized in [4], yet it replaces full blown principality by a notion of principality among maximal derivations, which is a weaker property.

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.

5 Concerns

This section addresses some natural concerns about the relaxed value restriction.

5.1 Typing power and usefulness

A first question is how powerful the relaxed value restriction is, compared to the value restriction and other known systems, and whether its improvements are genuinely useful or not. If we considered only benchmarks proposed in the literature [10, 11], we would come to the conclusion that the relaxed value restriction adds no power: its results exactly matches those of the strict value restriction. This is because all examples in the literature are only concerned with polymorphic procedures, not polymorphic data.

In the previous section we have given a fair number of examples handling polymorphic data. They demonstrate the additional power of our system. Compared with system predating the value restriction, we are in general less powerful, with some exceptions as shown in section 3. However, as we have seen in section 2, implementation abstraction matters more than pure typing power, and on this side we keep the good properties of the value restriction.

Our examples with constructor functions and abstract datatypes were expressible in systems predating the value restriction, and are refused by the strict value restriction. This makes one wonder why this didn't cause more problems during the transition. These idioms were apparently rarely used then. However, the author believes he is not alone in having experienced exactly those problems on newly written code. And there have been explicit reports of polymorphism problems with objects and variants, justifying the need for such an improvement.

5.2 Abstraction

While we claim that our scheme is not breaking implementation abstraction, one may remark that we require variance annotations for abstract datatype definitions. Aren't these annotations breaking abstraction?

Clearly, specifying a variance reduces the generality of an interface, and as such it is reducing its abstraction degree. However we claim that this does not mean that we are breaking implementation abstraction. We give here a concrete example, defining covariant vectors on top of nonvariant 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
```

What this example demonstrates, is that variance is not limited by the implementation. By changing the superficial definition, while keeping the same internal implementation, we may improve the variance of a datatype. 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.

To be fully honest, there are cases where an overspecified variance results in making some implementations impossible. But this should be seen as a problem of bad design, and the above example gives a natural criterion for proper variance of an abstract datatype: this should at most be the variance of the minimal set of operations which cannot be defined without access to the implementation.

5.3 Ease of use

Does the introduction of variance make the language harder to use? There are actually two problems: understanding the new typing rule, and having to write variance annotations for abstract datatypes.

Seeing that the value restriction itself is rather hard to grasp —notwithstanding the simplicity of its definition—, one might argue that any improvement of polymorphism (when it does not involve changes in the type algebra itself) is good, as it is going to avoid some non-intuitive type errors. Moreover, once you understand the typing algorithm, the relaxed value restriction introduces no leap in complexity.

More disturbing may be the need for variance annotations. For Objective Caml, they were already there, as the language allows explicit subtyping. So we are just exploiting an existing feature. But even if it were to be newly added, keep in mind that explicit annotations are only needed for abstract datatype definitions, and that there is a good semantic criterion as to what they should be. Of course this information is only optional: at worst, we are still as powerful as the value restriction.

5.4 Compilation

A last concern is with compilation, in particular for compilers using type information during compilation or at runtime. These compilers often involve a translation to an explicitly typed second-order lambda-calculus, which does not seem to be a good target for our system since, as we will see in the next sections, our type soundness seems to require subtyping.

A first remark is that the problem lies not so much in our approach as in the inadequation between polymorphic data structures and second-order lambda-calculus.

While there can be no value whose type is a covariant variable inside the data structure, second-order lambda-calculus would have us pass its (useless) type around.

The answer is simple enough: we just have to extend the target type system with the needed subtyping, knowing that this will not impact later stages of compilation as there are no values of type `zero` anyway. To gain full profit of our remark, we may even replace all purely covariant type variables with `zero` —in value bindings too—, so as to minimize the type information passed around.

While `zero` is not a problem, compilation is one of the reasons we have stopped short of exploiting 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 had 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. However, the existence of `top` would require all values to be represented in a uniform way. This is just what type-passing implementations want to avoid. Actually, even Objective Caml, which has only a very conservative use of type information, does not satisfy this property[2].

6 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 [14]. 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.

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

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

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

[2] The function `Obj.repr` can be seen as a coercion to `top` (*aka* `Obj.t`), but it is unsafe.

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

In one sentence: arrays of float values have a special representation, and operations on arrays are not semantically correct when float and int values are mixed —which is of course impossible using the existing type system and safe operations.

Reduction rules are given in figure 3. 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 $e_1 \stackrel{*}{\to} e_n$ whenever $e_1 \to e_2 \ldots \to e_n$. Reduction does not produce badly-formed expressions.

Lemma 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.*

Types are the usual monotypes and polytypes.

$$\tau ::= \alpha \mid \tau\ \texttt{ref} \mid \tau \times \tau \mid \tau\ \texttt{list}$$
$$\sigma ::= \tau \mid \forall \bar{\alpha}.\tau$$

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 4. Those rules are again taken from Reference ML, assuming all type variables to be imperative (which is equivalent to applying the value restriction, *cf* [1] page 6). The only exception is the LET$_e$ rule, which generalizes some variables. In the value case, $Close(\tau_1, \Gamma) = \forall FTV(\tau_1) \backslash FTV(\Gamma).\tau_1$ as usual, but in the non-value case we still generalize safe variables: $CovClose(\tau_1, \Gamma) = \forall FTV(\tau_1) \setminus V^-(\tau_1) \setminus FTV(\Gamma).\tau_1$, 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.

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

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 neither of the two is an instance of the other. *i.e.* we would have lost principality.

As we explained in section 4.5, rule LET$_e$ does not unshare covariant type variables, as it would be sound to do, but only allows for more type variables to be generalized. Unsharing variables would break even the partial subject reduction we define lower.

We include the RHO typing rule for completeness, but we cannot use it to obtain full subject reduction. We can see this on the following example[3].

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

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

[3] 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.

$$
\begin{array}{lrlllr}
(\beta_v) & (\lambda x.e)\ v & \to e[v/x] & (\pi_1) & & \pi_1\ (v_1, v_2) \to v_1 \\
(let) & \mathsf{let}\ x = v\ \mathsf{in}\ e & \to e[v/x] & (\pi_2) & & \pi_2\ (v_1, v_2) \to v_2 \\
(\mathsf{Y}) & \mathsf{Y}\ v & \to v\ (\lambda x.\mathsf{Y}\ v\ x) & (un_1) & & \mathsf{uncons}\ v_1\ v_2\ \mathsf{nil} \to v_1\ \mathsf{nil} \\
(ref) & \mathsf{ref}\ v & \to \rho\langle x, v\rangle.x & (un_2) & & \mathsf{uncons}\ v_1\ v_2\ (\mathsf{cons}\ v) \to v_2\ v \\
(deref) & \rho\theta\langle x, v\rangle.R[!\,x] & \to \rho\theta\langle x, v\rangle.R[v] & & & \\
(assign) & \rho\theta\langle x, v_1\rangle.R[:=x\ v_2] & \to \rho\theta\langle x, v_2\rangle.R[v_2] & & & \\
(\rho_{merge}) & \rho\theta_1.\rho\theta_2.e & \to \rho\theta_1\theta_2.e & & & \\
(\rho_{lift}) & R[\rho\theta.e] & \to \rho\theta.R[e] & & \text{if } R \neq [\,] & \\
(\rho_{drop}) & \rho\theta.e & \to e & \text{if } dom(\theta) \cap FV(e) = \emptyset & &
\end{array}
$$

Fig. 3. Reduction rules

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

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

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

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

$\textsc{Axioms}$

$$
\begin{array}{lll}
\multicolumn{3}{l}{\Gamma \vdash \mathsf{Y} : ((\tau_1 \to \tau_2) \to \tau_1 \to \tau_2) \to \tau_1 \to \tau_2} \\
\Gamma \vdash \mathsf{ref} : \tau \to \tau\ \mathtt{ref} & \Gamma \vdash \,! : \tau\ \mathtt{ref} \to \tau & \Gamma \vdash := : \tau\ \mathtt{ref} \to \tau \to \tau \\
\Gamma \vdash \pi_1 : \tau_1 \times \tau_2 \to \tau_1 & \Gamma \vdash \pi_2 : \tau_1 \times \tau_2 \to \tau_2 & \Gamma \vdash \mathsf{nil} : \tau\ \mathtt{list} \\
\multicolumn{3}{l}{\Gamma \vdash \mathsf{uncons} : (\tau_1\ \mathtt{list} \to \tau_2) \to (\tau_1 \times \tau_1\ \mathtt{list} \to \tau_2) \to \tau_1\ \mathtt{list} \to \tau_2}
\end{array}
$$

Fig. 4. Typing rules

$$
\textsc{b-Sub}\ \frac{\Gamma \models e : t \quad t \leq t'}{\Gamma \models e : t'}
\qquad
\textsc{b-Let}_v\ \frac{(\forall t \in s)\, \Gamma \models v : t \quad \Gamma[x \mapsto s] \models e : t'}{\Gamma \models \mathsf{let}\ x = v\ \mathsf{in}\ e : t'}
\qquad
\textsc{b-Abs}\ \frac{\Gamma[x \mapsto \uparrow t_1] \models e : t_2}{\Gamma \models \lambda x.e : t_1 \to t_2}
$$

$$
\textsc{b-Var}\ \frac{t \in \Gamma(x)}{\Gamma \models x : t}
\qquad
\textsc{b-App}\ \frac{\Gamma \models e_1 : \tau_2 \to t_1 \quad \Gamma \models e_2 : t_2}{\Gamma \models e_1\ e_2 : t_1}
\qquad
\textsc{b-Let}_e\ \frac{\Gamma \models e_1 : t_1 \quad \Gamma[x \mapsto \uparrow t_1] \models e_2 : t_2}{\Gamma \models \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : t_2}
$$

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

Fig. 5. Typing rules for $\mathrm{B}(T)$

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 superficial, 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

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

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. For instance, if we assume the type of r to be $(\forall\alpha.\alpha \texttt{ list}) \texttt{ ref}$, then we must assume y to have the same second-order type, and z to be of type $(\forall\alpha.\alpha \texttt{ list}) \texttt{ ref} \to (\forall\alpha.\alpha \texttt{ list})$, which is getting further and further away from ML style polymorphism. 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 direct subject reduction, we must prove type soundness in an indirect way. Following our intuition, we could recover subject reduction in a stronger system, by adding a subsumption rule,

$$\frac{\Gamma \vdash e : \tau[\overline{\texttt{zero}}/\bar{\alpha}]}{\Gamma \vdash e : \tau}\ \bar{\alpha} \cap V^-(\tau) = \emptyset$$

Rather than doing this directly, and bearing the burden of proof, we will do this in the next section by translating our derivations into a known type system validating this rule. We believe that an appropriate form of subsumption (direct or indirect) is essential to proofs of subject reduction for type systems validating our $\textsc{Let}_e$ 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 monotonic with respect to the instantiation order $\succ$, that is, for any type substitution S, we have $\mathit{CovClose}(\tau, \Gamma) \succ \mathit{CovClose}(S(\tau), S(\Gamma))$.

Proposition 1 (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.

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

Proposition 2 (partial subject reduction). *Non side-effecting reductions, i.e. rules $(\beta_v), (let), (\mathsf{Y}), (\pi_i), (un_i)$ 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 $\textsc{Let}_v$ and $\textsc{Let}_e$.

Lemma 2 (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 $\Gamma \vdash e[v/x] : \tau$.*

7 Type soundness

Rather than extending our own type system with subsumption, we will reuse one that already has the required combination of polymorphism, imperative operations, and subtyping. A good choice is Pottier's $\mathrm{B}(T)$ [18], as its typing rules closely match ours. $\mathrm{B}(T)$ was originally developed as an intermediate step in the proof of type soundness for $\mathrm{HM}(X)$, a constraint-based polymorphic type system [19]. $\mathrm{B}(T)$ is particular by its extensional approach to polymorphism: polytypes are not expressed syntactically, but as (possibly infinite) sets of ground monotypes. For us, its main advantages are its simplicity (no need to introduce constraints as in $\mathrm{HM}(X)$), and the directness of the translation of typing derivations.

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

The T in $\mathrm{B}(T)$ 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. $\rightarrow$ should be a total function from $T \times T$ into T, such that $t_1 \rightarrow t_2 \leq t'_1 \rightarrow t'_2$ implies $t'_1 \leq t_1$ and $t_2 \leq t'_2$. $\mathtt{ref}$ should be a total function from T to T, such that $t\ \mathtt{ref} \leq t'\ \mathtt{ref}$ implies $t = t'$. Moreover $t_1 \rightarrow t_2 \leq t\ \mathtt{ref}$ and $t\ \mathtt{ref} \leq t_1 \rightarrow t_2$ 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 $t' \in s$). We write $\uparrow t$ for the upward closure of a monotype (the set of all its supertypes).

The terms and reduction rules in $\mathrm{B}(T)$ 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, the 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[4] are given in figure 5. 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 that $\textsc{b-Let}_e$ is now monomorphic (this is the strict value restriction), subsumption $\textsc{b-Sub}$ is added, and polymorphism is handled semantically in $\textsc{b-Var}$ and $\textsc{b-Let}$. $\textsc{Axioms}$ for references are included.

The following theorem is proved in [18], section 3, for any $(T, \leq)$ satisfying the above requirements.

[4] 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. $\textsc{b-Rho}$ merges $\textsc{b-Store}$ and $\textsc{b-Conf}$ from the original presentation.

Theorem 1 (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 $\{\alpha, \beta, \ldots\}$. 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 $\mathrm{B}(T)$. `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 non-variant, and $\to$ which is contravariant in its first parameter and covariant in its second one. This conforms to the requirements for $\mathrm{B}(T)$, 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 [18].

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 [14].

Lemma 3 (Progress). *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.*

Combining the above subject reduction and progress, our instance of $\mathrm{B}(T)$ is sound.

We present now the translation itself. First we must be able to translate each component of a typing judgment. The expression part is left unchanged. Types are translated under a substitution $\xi : V \to T$.

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

This translation is extended to polytypes appearing in typing environments.

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

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

Lemma 4. *Let $\bar{\alpha}$ be a set of type variables that appear only covariantly in τ_1. Let ξ be any translation substitution. Then $[\![\forall \bar{\alpha}.\tau_1]\!]\xi = \uparrow[\![\tau_1]\!](\xi[\bar{\alpha} \mapsto \overline{\mathtt{zero}}])$.*

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 and $CovClose(\tau_1, \Gamma) = \forall \bar{\alpha}.\tau_1$ then it is translated into
$$\frac{[\![\Gamma]\!]\xi' \models e_1 : [\![\tau_1]\!]\xi' \quad [\![\Gamma[x \mapsto \forall \bar{\alpha}.\tau_1]]\!]\xi \models e_2 : [\![\tau_2]\!]\xi}{[\![\Gamma]\!]\xi \models \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : [\![\tau_2]\!]\xi}\,(\mathrm{B\text{-}LET}_e)$$
where $\xi' = \xi[\bar{\alpha} \mapsto \overline{\mathtt{zero}}]$. By Lemma 4, we have $[\![\forall \bar{\alpha}.\tau_1]\!]\xi = \uparrow[\![\tau_1]\!]\xi'$, and this is an instance of rule $\mathrm{B\text{-}LET}_e$. Note also that $[\![\Gamma]\!]\xi' = [\![\Gamma]\!]\xi$ as $\alpha_i \cap FTV(\Gamma) = \emptyset$.

– if the last rule applied is LET_v and $Close(\tau_1, \Gamma) = \forall\alpha_1 \dots \alpha_n.\tau_1$, then it becomes

$$\dfrac{\dfrac{[\![\Gamma]\!]\xi' \models v : [\![\tau_1]\!]\xi'}{[\![\Gamma]\!]\xi' \models v : t}(\text{B-SUB}) \qquad [\![\Gamma]\!]\xi[x \mapsto s] \models e : [\![\tau_2]\!]\xi}{[\![\Gamma]\!]\xi \models \mathsf{let}\ x = v\ \mathsf{in}\ e : [\![\tau_2]\!]\xi}(\text{B-LET}_v)$$

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 $[\![\tau_1]\!]\xi' \leq t$. Here again $[\![\Gamma]\!]\xi' = [\![\Gamma]\!]\xi$.

– if the last rule applied is VAR, it becomes $\dfrac{[\![\tau]\!]\xi \in ([\![\Gamma]\!]\xi)(x)}{[\![\Gamma]\!]\xi \models x : [\![\tau]\!]\xi}(\text{B-VAR})$.

– other cases are trivial induction.

From this construction we can obtain the following proposition.

Proposition 3. *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.*

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 such normal forms, type derivations in B(T) coincide with our system.

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 2 (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 $\emptyset \vdash d : \delta$.*

8 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.

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) as found in [20]. We preferred B(T) for its robustness, and the lightness of its definition and proof, but this last approach would be purely syntactic.

Acknowledgments

I want to thank here Didier Rémy and the Cristal team at INRIA for discussions and encouragements.

A Proofs of lemmas

Lemma 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.

Lemma 2. *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 $\text{L}\scriptstyle{\text{ET}}_e$. If $\Gamma[x \mapsto \sigma_1] \vdash \mathsf{let}\ x' = v'\ \mathsf{in}\ e : \tau$ using $\text{L}\scriptstyle{\text{ET}}_v$, 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 $\text{L}\scriptstyle{\text{ET}}_v$.
Case $\text{L}\scriptstyle{\text{ET}}_v$. If $\Gamma[x \mapsto \sigma_1] \vdash \mathsf{let}\ x' = e'\ \mathsf{in}\ e : \tau$ using $\text{L}\scriptstyle{\text{ET}}_e$, 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 $\text{L}\scriptstyle{\text{ET}}_e$.
Other cases are all simple and standard.

Lemma 3. *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 [14], 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.

Lemma 4. *Let $\bar\alpha$ be a set of type variables that appear only covariantly in τ_1. Let ξ be any translation substitution. Then $[\![\forall\bar\alpha.\tau_1]\!]\xi = \uparrow[\![\tau_1]\!](\xi[\bar\alpha \mapsto \overline{\mathtt{zero}}])$.*

Proof. By definition, the latter is included in the former. So it is enough to show that for any t in $[\![\forall\bar{\alpha}.\tau_1]\!]\xi$, we have $[\![\tau_1]\!](\xi[\bar{\alpha} \mapsto \overline{\mathtt{zero}}]) \leq t$.
Let $\bar{\alpha} = \alpha_1 \ldots \alpha_n$ and $\xi' = \xi[\bar{\alpha} \mapsto \overline{\mathtt{zero}}]$. By definition of $[\![\forall\bar{\alpha}.\tau_1]\!]\xi$, there exists $\bar{t} = t_1 \ldots t_n$ 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 $[\![\tau_1]\!]\xi' \leq t$.

References

1. Wright, A.K.: Simple imperative polymorphism. Lisp and Symbolic Computation **8** (1995)
2. Rémy, D., Vouillon, J.: Objective ML: A simple object-oriented extension of ML. In: Proc. ACM Symposium on Principles of Programming Languages. (1997) 40–53
3. Garrigue, J.: Programming with polymorphic variants. In: ML Workshop, Baltimore (1998)
4. Garrigue, J., Rémy, D.: Extending ML with semi-explicit higher order polymorphism. Information and Computation **155** (1999) 134–171
5. Weis, P., Aponte, M., Laville, A., Mauny, M., Suarez, A.: The CAML reference manual, version 2.6.1. Rapport Technique RT-0121, INRIA (1990)
6. Tofte, M.: Type inference for polymorphic references. Information and Computation **89** (1990) 1–34
7. Milner, R., Tofte, M., Harper, R.: The Definition of Standard ML. MIT Press, Cambridge, Massachusetts (1990)
8. Greiner, J.: SML weak polymorphism can be sound. Technical Report CMU-CS-93-160R, Canegie-Mellon University, School of Computer Science (1993)
9. Hoang, M., Mitchell, J., Viswanathan, R.: Standard ML-NJ weak polymorphism and imperative constructs. In: Proc. IEEE Symposium on Logic in Computer Science. (1993) 15–25
10. Talpin, J.P., Jouvelot, P.: The type and effect discipline. In: Proc. IEEE Symposium on Logic in Computer Science. (1992) 162–173
11. Leroy, X., Weis, P.: Polymorphic type inference and assignment. In: Proc. ACM Symposium on Principles of Programming Languages. (1991) 291–302
12. Leroy, X.: Polymorphic typing of an algorithmic language. Research report 1778, INRIA (1992)
13. Leroy, X.: Polymorphism by name for references and continuations. In: Proc. ACM Symposium on Principles of Programming Languages. (1993) 220–231
14. Wright, A.K., Felleisen, M.: A syntactic approach to type soundness. Information and Computation **115** (1994) 38–94
15. Ohori, A., Yoshida, N.: Type inference with rank 1 polymorphism for type-directed compilation of ML. In: Proc. International Conference on Functional Programming, ACM Press (1999)
16. Leroy, X., Doligez, D., Garrigue, J., Rémy, D., Vouillon, J.: The Objective Caml system release 3.06, Documentation and user's manual. Projet Cristal, INRIA. (2002)
17. Garrigue, J.: Simple type inference for structural polymorphism. In: The Ninth International Workshop on Foundations of Object-Oriented Languages, Portland, Oregon (2002)
18. Pottier, F.: A semi-syntactic soundness proof for HM(X). Research Report 4150, INRIA (2001)
19. Odersky, M., Sulzmann, M., Wehr, M.: Type inference with constrained types. Theory and Practice of Object Systems **5** (1999) 35–55
20. Skalka, C., Pottier, F.: Syntactic type soundness for HM(X). In: Proceedings of the 2002 Workshop on Types in Programming (TIP'02). Volume 75 of Electronic Notes in Theoretical Computer Science., Dagstuhl, Germany (2002)