Programming with Polymorphic Variants

Jacques Garrigue

garrigue@kurims.kyoto-u.ac.jp

Research Institute for Mathematical Sciences
Kyoto University, 606-8502 Kyoto, JAPAN

Abstract

Type inference for structural polymorphism —*i.e.* record and variant polymorphism— has been an active area of research since more than 10 years ago, and many results have been obtained. However these results are yet to be applied to real programming languages. Based on our experience with the Objective Label system, we describe how variant polymorphism can be integrated in a programming language, and what are the benefits. We give a detailed account of our type inference and compilation schemes.

1 Introduction

The distinction between parametric polymorphism and *ad hoc* polymorphism is well known. In parametric polymorphism, *e.g.* ML polymorphism, types do not interact with evaluation, and polymorphic parts may be instantiated with anything, while in *ad hoc* polymorphism, with overloading or object-orientation, types do interact with evaluation, and possible instances are restricted. More subtle is the notion of structural polymorphism, *i.e.* the ability for a function to access differently shaped data, appearing in record typing for instance. It is indeed parametric, in that types do not interfere with evaluation (at least at the formal level), but instances are restricted.

Interestingly, one may model object-oriented programming in a formalism based on structural polymorphism, thus parametric. Objective ML [RV97] is such an example: objects are formalized as records, and subsumption as instanciation of structurally polymorphic type schemes. Indeed types do not interfere with evaluation: all objects must bring their methods with them.

While structural polymorphism has nice properties from a functional point of view, Objective ML seems to be its only instance in a widely used programming language. Ohori proposed a polymorphic type system for records and implemented it in SML#, but Standard ML'97 is still using structurally monomorphic records.

Records and objects are not the only application field for structural polymorphism. Their dual, variants, may naturally be typed by the same mechanism. Both Rémy [Rém89] and Ohori [Oho95] emphasize this fact.

Both for record and variant polymorphic typing, technical feasibility is a solved problem. The remaining question is whether it is useful or not, and if it is, to provide an easily understandable typing system, and an efficient compilation method.

Based on our experience with the Objective Label [Gar] system, which is a derivative of Objective Caml [Ler], this paper tries to answer these three questions. We first present informally how Objective Label types variants, and what are their applications. Then we give a compilation scheme, extremely simple but also very efficient. Finally we give a full formalization of the type system as it is implemented in the Objective Label compiler.

The contributions of this paper are more practical than theoretical: the type system we provide is not more expressive than Rémy's for instance, but it is simply more adapted to the use we make of it.

2 A naive approach to variants

Typing polymorphic variants is a complex task. As a first step we will make an inventory of the various expressions we want to type, and of the types our language should contain.

Let us define a simple language, extending core ML with variants.

$$\begin{array}{rcll} e & ::= & x \mid c \mid \lambda x.e \mid e\,e \mid \mathsf{let}\ x = e\ \mathsf{in}\ e \mid \ldots & \text{core ML} \\ & \mid & `tag(e) & \text{variant} \\ & \mid & \mathsf{case}\ e\ \mathsf{of}\ `tag(x) \to e; \ldots ; `tag(x) \to e & \text{matching} \end{array}$$

For the sake of simplicity, all variants have a single argument. Since we suppose that we have also *unit* and tuples, this is expressive enough. However we will omit *unit* in examples, and just write ‘*apple* for ‘*apple*().

Here are basic examples for these new constructs. Types are those given by the Objective Label compiler.

$$\begin{array}{l} \mathsf{let}\ a = `apple \\ a : [> apple] \\ \mathsf{let}\ b = `orange(\texttt{"spain"}) \\ b : [> orange(string)] \end{array}$$

The type $[> apple]$ means that a is a variant, containing the tag *apple*, and that it takes no argument for this tag. Similarly $[> orange(string)]$ means that b is a variant, containing the tag *orange*, and that the argument of *orange* is of type *string*. Why this "$>$" ? You can see no type variable, but in fact, since subsumption is achieved through type instanciation, these types have to be polymorphic, as shown in the next example.

$$\begin{array}{l} \mathsf{let}\ l = [a, b] \\ l : [> apple\ orange(string)]\ list \end{array}$$

l is a list of variants, each of them being tagged either *apple* or *orange*. $[> apple\ orange(string)]$ is an instance of both $[> apple]$ and $[> orange(string)]$. "$>$" means that a type is polymorphic, and can be extended by the addition of new tags.

Symmetrically, matching is typed with a "$<$".

$$\begin{array}{l} \mathsf{let}\ show\,x = \\ \quad \mathsf{case}\ x\ \mathsf{of}\ `apple \to \texttt{"apple"}; \\ \qquad\qquad `orange(s) \to \texttt{"orange "}\texttt{\^{}}s \\ show : [< apple\ orange(string)] \to string \end{array}$$

show accepts either an *apple* without argument, or an *orange* with a *string* argument. Again, this type is polymorphic: it may be restricted later.

$$\begin{array}{l} \texttt{let}\ show'\ x = \\ \quad \texttt{case}\ x\ \texttt{of}\ `apple \rightarrow \texttt{"apple"};`pear \rightarrow \texttt{"pear"} \\ show' : [< apple\ pear] \rightarrow string \\ \texttt{let}\ l = [show, show'] \\ l : ([< apple] \rightarrow string)\ list \end{array}$$

You wonder why we would want to put both functions in a list? The same typing arises when apply both function on the same monomorphic argument.

$$\begin{array}{l} \texttt{let}\ show\_both\, x = (show\, x, show'\ x) \\ show\_both : [< apple] \rightarrow string \times string \end{array}$$

Clearly x must be acceptable by both *show* and $show'$.

In the above examples, we have seen two kinds of types: "$>$" types, or lower bounds, and "$<$" types, or upper bound. We have seen that they can be refined by combining their constraints. The next question is, what should happen when combining constraints of the two kinds. We use a non generalizable type variable to show this.

$$\begin{array}{l} \texttt{let}\ r = ref\ `apple \\ r : \_[> apple]\ ref \\ show\ !r \\ - : string = \texttt{"apple"} \\ r : \_[< apple\ orange(string) > apple]\ ref \\ r := `orange(\texttt{"spain"}) \\ r : [apple\ orange(string)]\ ref \end{array}$$

The _ in front of the variant type shows that this type is monomorphic (cannot be made polymorphic due to the value-only restriction of polymorphism), but not yet fully determined. At the beginning we only know that r is a reference to a variant which contains an apple. After applying the *show* function, the type changes to reflect a new constraint: it may only contain apples or oranges. Finally we put an orange in r, and its type becomes fully determined: it contains (potentially) oranges and apples, and may not contain anything else.

Fully determined variant types are useful for programming, since they are the only ones we may use to define type abbreviations.[1]

$$\texttt{type}\ fruit = [apple\ orange(string)\ pear]$$

Last, to be as powerful as defined datatypes, polymorphic variants shall support recursion. Here is the type inferred for the map function. γ and δ are structural type variables. More details are given in the formal development.

$$\begin{array}{l} map : (\alpha \rightarrow \beta) \rightarrow \\ \qquad \gamma[< nil\ cons(\alpha \times \gamma)] \rightarrow \delta[> nil\ cons(\beta \times \delta)] \end{array}$$

3 Variants at work

In this section we give examples of how polymorphic variants may be used. We distinguish between uses which do only require polymorphism to make inference possible, but are otherwise monomorphic, and uses where polymorphism is really exploited.

[1]Objective Caml also allows the definition of *constrained* type abbreviations, and you can use them in Objective Label to get rid of this limitation.

3.1 Monomorphic uses

Many uses of variants do not really exploit polymorphism. Variants are only used at one type. This is in fact the most frequent case, but we still need polymorphism to allow type inference.

What we essentialy use here is the pervasiveness of variant tags: they are defined nowhere, they simply exist before you use them. Similarly, there is no notion of conflict between two tags, as long as you do not use them in the same data structure. For typing, every tag is a new instance of itself.

3.1.1 Overloaded constructors

This phenomenon happens often. You have defined a datatype, but then you realize that you need many variations on it. Adding one constructor, or changing the type of the argument for a specific constructor.

With the traditional ML approach you have to define as many datatypes as there are variations. Of course, all constructors must have different names. You end up thinking about a nice naming scheme to distinguish the same constructor appearing in various datatypes.

Polymorphic variants were originally introduced in Objective Label [Gar] to solve this problem. In trying to build an interface for Tcl/Tk, it appeared that Tk had as many notions of indices as it has widget classes. Differences are small, but ignoring them would allow for runtime errors. The phenomenon is even stronger for the different options one can pass to a widget: some are allowed for all widgets, some have a standard definition but are only available on some widgets, and some actually differ from widget to widget. Such overloading appears a lot when interfacing to external (C or other) libraries.

With polymorphic variants, only the function needs to know about the different cases it expects. The user may reuse the same tag name, without bothering about this name being reserved for one specific type.

$$\begin{array}{l} Entry.index : entry\, widget \rightarrow \\ \qquad [anchor\ at(int)\ end\ insert\ num(int)\ selfirst\ sellast] \rightarrow int \\ Listbox.index : listbox\, widget \rightarrow \\ \qquad [active\ anchor\ atxy(int \times int)\ end\ num(int)] \rightarrow int \\ Menu.index : menu\, widget \rightarrow \\ \qquad [active\ at(int)\ end\ last\ none\ num(int)\ pattern(string)] \rightarrow int \end{array}$$

This overloading use may be compared with Haskell type classes. Indeed, TkGofer [CV] uses type classes to do exactly that.

3.1.2 Implicit datatypes

This case is similar to the previous one, except that this is more a question of comfort and namespace considerations, than meaningful overloading.

Like the index functions above, frequently library function must receive structured information as input. This information is not to be stored anywhere, but just to be processed by the library function and discarded. In such cases having to define a datatype may be heavy, particularly in a system with modules.

If the datatype is defined together with the function, it means that either one will have to open the module, introducing all names in the current namespace, or use an awkward dot notation for constructors.

On the other hand, with polymorphic variants no type needs to be defined, and variant tags need not be qualified.

$$\begin{array}{l} (*\ \text{In module Arg}\ *) \\ \texttt{type}\ spec = [unit(unit \rightarrow unit)\ set(bool\ ref) \end{array}$$

$\quad clear(bool\ ref)\ string(string \to unit)$
$\quad int(int \to unit)]$
val $parse : (string \times spec)\ list \to (string \to unit) \to unit$

let _ =
  $Arg.parse$ [("-o", ‘$string\ output$), ("-n", ‘$int\ number$)] $others$

This also improves the readability of programs: one may choose to use [*on off*] in place of *bool* when this is more expressive, *etc...*

3.1.3 Shared datatypes

Sharing a datatype between two different libraries is a subtle thing. Either you put it in one of them, and create a dependency between the two libraries; or you define a common header, and have your two libraries depend on a third one; or you have to functorize everything in both libraries to take the datatype as parameter. Anything short of that will give you two datatypes with the same structure, but incompatible.

Polymorphic variants provide a solution to this problem. You just define the same type in both libraries, and since these are only type abbreviations, the two definitions are compatible. The type system checks the structural equality when you pass a value from one library to the other.

3.2 Polymorphism and subtyping

Polymorphic variants have many more possibilities than simple constructor overloading. If we use fully their polymorphism, they can simulate structural subtyping, as is done Objective ML for objects.

3.2.1 Variant hierarchy

Not only can the same variant tag be used in many different variant types, but any polymorphic variant may be viewed under different types. Variant types form a subtyping hierarchy, and both variants and their acceptors (functions receiving them) have multiple views according to this hierarchy.

let a = [‘$orange$("morocco"), ‘$apple$]
a : [> $apple\ orange(string)$] $list$
let b = [‘$apple$, ‘$pear$]
b : [> $apple\ pear$] $list$
let $c = a\ @\ b$
c : [> $apple\ orange(string)\ pear$] $list$ =
  [‘$orange$("morocco"), ‘$apple$, ‘$apple$, ‘$pear$]

a and *b* are both subtypes of *c*. They may be used several times, at any of their supertypes. For acceptors, see *show_both* in section 2.

3.2.2 Safe programming

When a program does many operations on the same datatype, this is quite frequent that part of these operations make in fact some hypotheses on their input, accepting only a specific subset of the datatype.

In ML, the usual way to handle such cases is to raise an exception when the input is outside of this subset. However, if this subset can be defined by the restricted set of constructors it uses, polymorphic variants offer a solution. Each operation may return the most specific type for its output, and it may then be used safely at any supertype.

In order to support effectively this practice, Objective Label provides a specific notation for variant types considered as a subtype of another variant type (this is an extension of a similar notation for objects in Objective ML):

$$\#variant[> tag_1 \dots tag_n]$$

is a shorthand for

$$[< tag_1(\tau_1) \dots tag_{n+m}(\tau_{n+m}) > tag_1 \dots tag_n]$$

when *variant* is defined as $[tag_1(\tau_1) \dots tag_{n+m}(\tau_{n+m})]$.

3.2.3 Encoding subtyping in variants

While variants have their own notion of subtyping, they can also be used for building subtyping relations in abstract types. This feature was heavily used for interfacing Objective Label with the OpenGL graphical library.

Here is part of the interface of a library providing access to raw C arrays. The abstract type of arrays is $\alpha\ t$.

type $\alpha\ t$
type $kind$ = [$bitmap\ byte\ double\ float\ int\ long\ short$]
type $fkind$ = [$double\ float$]
type $ikind$ = [$bitmap\ byte\ int\ long\ short$]
val $create$: (#$kind$ as α) $\to int \to \alpha\ t$
external get : #$ikind\ t \to int \to int$ = "ml_raw_get"
external set : #$ikind\ t \to int \to int \to unit$ =
  "ml_raw_set"
external get_float : #$fkind\ t \to int \to float$ =
  "ml_raw_get_float"
external set_float : #$fkind\ t \to int \to float \to unit$ =
  "ml_raw_set_float"

#*kind* as α constrains α, which appears both as input and as parameter to the output type *t*, to be a subtype of *kind*. Due to the value-only restriction, the result is not polymorphic.

let arr = $create$ ‘$float$ 10
arr : _#$kind$[> $float$] t

The interesting point here is that the distinction between integer and floating point arrays is made at the type level, and the distinction between various data sizes is made at the value level. Polymorphic variants allow one to mix the two levels, and produce both kinds of arrays with the same *create* function.

4 Compiling variants

One may think of many clever schemes for compiling polymorphic variants.

The first to come to mind is generating different integer values for all tags in a program, and then compile everything just as would be done for defined datatypes. This ought to be simple and efficient. If the program does not use too many different tags, we may even be able to compile matching with switches.

However there is a major drawback to this scheme: we need to know all the tags appearing in a program. Separate compilation breaks it. Moreover, adding a tag somewhere in the program may change the representation of other tags. Raw values (*e.g.* obtained by `output_value` in Caml) wouldn't be compatible between different versions of the same program. Interfacing with external libraries through variants would be a nightmare.

Luckily, another very simple approach works transparently, giving a uniform representation to tags, depending only on their names. The idea is just to hash tag names to usual integers. Most often this will mean a 31-bit value, if we need one bit for the GC.

The immediate concern is: but what to do if two different tags get the same hash value? Our answer is simple: nothing. More precisely, if one tries to use two tags with identical hash values in the same variant type, the compiler fails on a type error.

The point here is that this kind of conflict is detected by the type checker. There are two advantages to that:

- No runtime error. One just has to change a tag name in the source and try again.
- The type-checker only generates errors on effectively conflicting tag names.

In an untyped framework, one would have to check any two tags in the whole program for possible conflicts, while here we only have to check individually each variant type. This means that, even in a separate compilation scheme, this check only occurs at compile time, and no conflicts may imply hidden tags (which do not appear in a module's interface) after compilation. This also means that the probability of having a conflict is only proportional to the number of tags in a variant type, rather than the total number of tags in a program.

In the Objective Label implementation, the formula used to convert a variant tag *tag* of length n into a 31-bit integer is:

$$\mathsf{hash}(tag) = \left(\sum_{i=1}^{n} tag[i] * 223^{n-i}\right) \ \mathsf{mod}\ 2^{31}.$$

For compatibility reasons, we have to stick with the same formula on 64-bit implementations also. This formula only guarantees that all 4 character identifiers will be given different hash values. For more than 4 characters they may collide; but supposing that this functions gives us a random distribution, the probability $p(n)$ for two tags to collide in a n tag variant is (we use the average number of collisions):

$$p(n) \leq \sum_{i=1}^{n-1} \frac{i}{2^{31}} \leq \frac{n^2}{2^{32}}.$$

That is, with 256 tags, this probability is only $p(256) = 1.5 \cdot 10^{-5}$. Even $p(10000)$ is about 2 %. In practice this is incomparably lower than the chance of getting a conflict with a predeclared keyword of the language.

Another advantage of this representation is that it allows for an efficient compilation of matching. Admittedly, we cannot use table switch, like with defined datatypes, so matching a variant is not constant time. But since we know the hash value for each tag at compile time, we can generate a choice tree, and do the matching in $\log(n)$. Moreover, on many computer architectures conditional jump is much faster than indirect jump, so that for modest size variants types, we may even be faster than a table switch. Compiling to native code on a DEC Alpha, with the Objective Caml 1.07 backend, we noticed a more than 10 % speedup on ten way cases (the benchmark is a single flat matching inside a for loop). Of course, this speedup does not apply to bytecode.

The uniform data representation also makes the writing of foreign function interfaces easier. For instance, when interfacing with OpenGL, the translation from variant tags to C enumeration types can be done on the C side of the interface. ML and C sides can then be built independently, which simplifies the structure and avoids errors. The conversion from variant tags, which simply denote strings, to C enumeration type is one more chance of dynamic type checking, not to neglect when one knows the fragility of such interfaces.

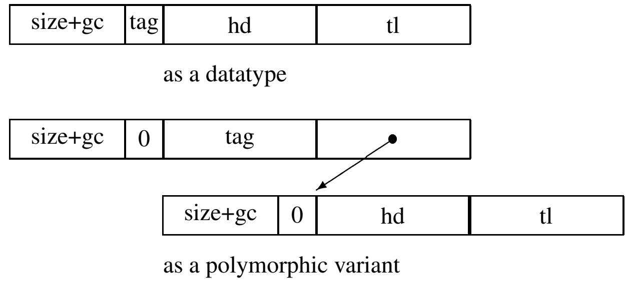

Figure 1: Internal representation of a cons-cell

Finally, the only drawback of compiled polymorphic variants compared to defined datatypes is its space consumption. For variants without argument, a word suffices and nothing needs to be heap allocated, so we are as efficient as Objective Caml datatypes there. But for datatype constructors with arguments, the original Objective Caml takes profit of the presence of an header word on heap blocks for storing its variant tags inside it. Since this header is also used for size and GC information, only 8 bits are available for tagging. That means that we cannot use the same compact representation for polymorphic variants. The tag has to be stored in one more word, so that a polymorphic variant with argument takes 3 words in the heap, instead of 2. Moreover, for datatype constructors with multiple arguments, Objective Caml uses definition information to flatten them, so that they use only one block. With polymorphic variants, we must assume a uniform representation, and represent the arguments as a tuple pointed by the argument field of the variant. This means $n+4$ words of heap instead of $n+1$, and one more level of indirection. A comparison of both representations for a cons-cell is given in figure 1.

5 Ordering variant types

As a first step towards a type system for polymorphic variants, we shall analyze the subsumption relation between variant types. This is also a good tool to understand differences with other proposals.

Intuitively a monomorphic variant type (just like a record type) may be represented by a set of tags with their associated types.

$$[tag_1(\tau_1)\dots tag_n(\tau_n)]$$

Naturally we obtain the following subsumption relation between variants:

$$\begin{aligned}&[tag_1(\tau_1)\dots tag_n(\tau_n)] \leq \\ &\qquad [tag_1(\tau_1)\dots tag_n(\tau_n)\ tag_{n+1}(\tau_{n+1})\dots tag_{n+m}(\tau_{n+m})]\end{aligned}$$

That is, any value of some variant type may be used as a value of a variant type with more tags. This is the dual of a similar relation for records.

It looks like this structure has good properties: any pair of variant types has a greatest lower bound, and any pair of variant types with a common upper bound has a lowest upper bound. We just keep all the tags given the same type on both sides for *glb*, and take the union for *lub*.

Intuitively, the *lub* is used when computing the common type of two variant values (covariant subtyping), while the *glb* is needed for acceptors (contravariant subtyping).

However, this setting has a major deficiency: in order to compute the *glb*, one needs to test not only the equality on tags, but also the equality on types. $[tag_1 : \tau_1] \sqcap [tag_2 : \tau_2]$ is not empty only if both $tag_1 = tag_2$ and $\tau_1 = \tau_2$. Clearly, this is not compatible with inference, where one cannot check the equality of type variables, but only enforce it.

On the other hand, there is no such problem for the *lub*. The existence of a common upper bound just amounts to a compatibility condition between variant types: $[tag_1 : \tau_1 \dots tag_n : \tau_n] \simeq [tag'_1 : \tau'_1 \dots tag'_m : \tau'_m]$ if $\forall i, j\ (tag_i = tag'_j) \Rightarrow (\tau_i = \tau'_j)$. This compatibility condition justifies enforcing the equality of types when computing a *lub*.

One solution, and this is the one Ohori [Oho95] chose, is to use only the *lub* for typing. Essentially this means that we cannot use contravariant subtyping. Two acceptor types can only be unified if they are equal. Going back to our informal system of section 2, this amounts to saying that $[< tag_1(\tau_1) \dots tag_n(\tau_n)]$ is interpreted by the fully determined type $[tag_1(\tau_1) \dots tag_n(\tau_n)]$.

The result is much simpler than our system: there are only two kinds of types, lower-bounds $[> tag_1(\tau_1) \dots tag_n(\tau_n)]$ and fixed types $[tag_1(\tau_1) \dots tag_n(\tau_n)]$, and no mixed forms. Typing might be easily done using row-variables.

In such a system, variant values are polymorphic, but acceptors are not. Basically this means that monomorphic use of variants are possible, but polymorphic ones are very restricted. For instance, our C array library would not be possible in such a type system: the *create* function uses the double polymorphism in an essential way. Even using the same variant at several supertypes becomes difficult: with the value-only restriction of polymorphism, all result of functions are monomorphic. If a function returns a polymorphic variant, we can choose for the type of the result any instance of this variant, but only once.

let $a = id$ ‘*apple*
$a : \_[> apple] =$ ‘*apple*
show a
$- : string =$ "apple"
$a : \_[> apple\ orange(string)]$
show′ a
type error!

Since we are particularly interested in such uses of variants, we choose another solution. The notion of variant types is enriched, to allow both covariant and contravariant subtyping.

$$[tag_1 \dots tag_n \mid tag_1 : \tau_1 \dots tag_{n+m} : \tau_{n+m}]$$

$tag_1 \dots tag_n$ is the presence part. It indicates which tags may appear in the variant. The right side is the typing information, and may contain more tags than the presence part. The idea is that the presence part may grow or shrink by unification according to the variance, but typing information can only grow.

For this purpose, we define subsumption independently on the presence part and the typing part. For typings, subsumption works as before:

$$tag_1 : \tau_1 \dots tag_n : \tau_n \le tag_1 : \tau_1 \dots tag_{n+m} : \tau_{n+m}$$

This corresponds to the intuition that specifying more tags in the typing part restricts the variant type as a whole. Since we cannot infer the *glb* of two typing parts, only *lub* is available for subtyping. But we can recover both covariant and contravariant subtyping of variants by having two notions of subsumption for the presence part: $\subset$ or $\supset$.

In programs this corresponds to the two following patterns:

- when $e_1 : [P_1 \mid T_1]$ and $e_2 : [P_2 \mid T_2]$, and $T_1 \simeq T_2$

$$\texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2 : [P_1 \cup P_2 \mid T_1 \sqcup T_2]$$

- when $f_1 : [P_1 \mid T_1] \to \tau_1$ and $f_2 : [P_2 \mid T_2] \to \tau_2$, and $T_1 \simeq T_2$

$$\lambda x.(f_1\,x, f_2\,x) : [P_1 \cap P_2 \mid T_1 \sqcup T_2] \to \tau_1 \times \tau_2$$

The resulting type is obtained by taking the *lub* for two different orderings.

The last step is to combine contravariant and covariant subtyping in one representation, variant constraints, and to use the same subsumption relation for both. The extension is easy: we just use two presence sets instead of one.

$$[L < U \mid T]$$

L is a finite set of tags, U either a finite set of tags or $\top$ (all tags, the maximal element), and T a finite mapping from tags to types. L must be a subset of U. We do not allow indefinite tags, that is all tags in L and U (when not $\top$) must be given a type in T.

The subsumption relation between variant constraints is the following:

$$[L_1 < U_1 \mid T_1] \le [L_2 < U_2 \mid T_2] \quad \text{if } L_1 \subset L_2 \text{ and } U_1 \supset U_2 \text{ and } T_1 \le T_2$$

There are three ways one can tighten (*i.e.* make greater in the constraint order) a variant constraint: by making L larger, by making U smaller, or by adding tags in T. In particular, formally even an already fixed variant (*i.e.* a variant constraint such that $L = U$) may still be refined.

The *lub* of two compatible variant constraints is given as:

$$[L_1 < U_1 \mid T_1] \sqcup [L_2 < U_2 \mid T_2] = [L_1 \cup L_2 < U_1 \cap U_2 \mid T_1 \sqcup T_2]$$

These variant constraints are just a more abstract notation for the naive variant types we introduced in section 2. We give a mapping between the two notations in figure 2. The last two lines are new. They are required to express typing information for tags that do not appear in the presence information.

6 Formal type system

Following Ohori, we might use these constraints to qualify variables. Variant types for e_1 and f_1 would be written, using constrained variables, $e_1 : \alpha[P_1 < \top \mid T_1]$ and $f_1 : \alpha[\emptyset < P_1 \mid T_1] \to \tau_1$.

However, it appears that fully formalizing this aspect using kinds, as did Ohori, results in a quite complex system, particularly when one wants to handle recursive types.

Rémy's type system [Rém89] is another alternative. Constraints it can express are exactly those we just described. However, his approach relies exclusively on sorted variables, which means that we need one type variable by tag in a variant type. Understanding directly these types is difficult (Rémy even suggests hiding part of the types), and translating into our naive type system is not straightforward. The large number of variables implied may also be a problem if we want to quantify them explicitly, using first-class polymorphism [GR97].

For these reason the type system we use lies in between these two systems. Presence information is represented by kinds, but typing information uses row variables, so that we work with a multi-sorted algebra *à la* Rémy. We use only two variables by variant type. This way the kinding environment does not contain types, and polymorphism can be handled easily, even with recursive types.

While the type system we present here is very close to the one in Objective Label, there are a few differences:

$$
\begin{array}{lcl}
[tag_1(\tau_1)\ldots tag_n(\tau_n)] & = & [tag_1 \ldots tag_n < tag_1 \ldots tag_n \mid tag_1 : \tau_1 \ldots tag_n : \tau_n] \\
[< tag_1(\tau_1)\ldots tag_n(\tau_n)] & = & [\emptyset < tag_1 \ldots tag_n \mid tag_1 : \tau_1 \ldots tag_n : \tau_n] \\
[> tag_1(\tau_1)\ldots tag_n(\tau_n)] & = & [tag_1 \ldots tag_n < \top \mid tag_1 : \tau_1 \ldots tag_n : \tau_n] \\
[< tag_1(\tau_1)\ldots tag_{n+m}(\tau_{n+m}) > tag_1 \ldots tag_n] & = & [tag_1 \ldots tag_n < tag_1 \ldots tag_{n+m} \mid tag_1 : \tau_1 \ldots tag_{n+m} : \tau_{n+m}] \\
[tag_1(\tau_1)\ldots tag_{n+m}(\tau_{n+m}) \ldots > tag_1 \ldots tag_n] & = & [tag_1 \ldots tag_n < \top \mid tag_1 : \tau_1 \ldots tag_{n+m} : \tau_{n+m}] \\
[tag_1(\tau_1)\ldots tag_n(\tau_n) \ldots < tag_1 \ldots tag_k > tag_1 \ldots tag_l] & = & [tag_1 \ldots tag_l < tag_1 \ldots tag_k \mid tag_1 : \tau_1 \ldots tag_n : \tau_n]
\end{array}
$$

Figure 2: Mapping from naive types to variant constraints

- we omitted here explicit handling of recursive types and polymorphism restricted to values. These problems appear to be othogonal to the one we are concerned with.
- Objective Label assumes, incorrectly, that one need not keep typing information about absent tags. We show in subsection 7.1 how something close to that can be justified in the formal type system.

Expression are just those of Section 2. Types distinguish between presence and typing information in variants.

$$
\begin{array}{rcl}
\tau & ::= & \alpha \mid u \mid \tau \to \tau \mid [i \mid T] \\
\sigma & ::= & \tau \mid \forall \alpha.\tau \mid \forall \rho.\tau \mid \forall i_{\geq \langle L,U \rangle}.\tau \\
T & ::= & \rho \mid tag : \tau :: T \\
L & ::= & tag \ldots tag \\
U & ::= & tag \ldots tag \mid \top
\end{array}
$$

There are three differences with a usual ML type system. First, we use sorted types: there are three kinds of type variables, α for usual types, i for presence information, and ρ for row types. Then presence variables are kinded. In fact the real information is contained in the kind, of the form $\langle L,U\rangle$. They may only be instantiated to a variable of a smaller kind. Last, the typing information in variants is represented by a row type. Row types are considered modulo the equality $tag : \tau :: tag' : \tau' :: T = tag' : \tau' :: tag : \tau :: T$ when $tag \neq tag'$. Row type variables are not kinded: they may be instantiated to any row type.

Typing judgments are of the form $K;\Gamma \vdash e : \sigma$, and kinding judgments of the form $K \vdash i \geq \langle L,U\rangle$. Inference rules are given in figure 3. The system may look complex as there are many rules, but in fact half of them are simply dedicated to polymorphism, and do exactly the same thing for three different sorts of type variables.

Type reconstruction for this system is given in appendix A.

Due to the many type variables, types inferred in this system would be hard to read. However, we can use a property of the type reconstruction algorithm to show only naive types to the user. Assuming that all i-variables are always associated with the same typing information T in the initial typing environment, and reciprocally that all ρ-variables are always associated with same i, the type reconstruction algorithm preserves this property for newly introduced variables. As a result, we can consider that two variant types are identical in the output of the algorithm if they share the same i, and display them as a shared type. If their i differs, then they are unrelated.

Here is an example. We infer the type for the *map* function.

$$
\begin{array}{l}
\mathsf{let\ rec}\ map f\, l = \\
\quad \mathsf{case}\ l\ \mathsf{of}\ `nil \to `nil; \\
\qquad\qquad `cons(a,l) \to `cons(f\,a, map\,f\,l) \\
map : (\alpha \to \beta) \to \\
\qquad ([i_{\geq\langle \emptyset, nil\,cons\rangle} \mid nil : unit :: cons : \alpha \times \gamma :: \rho]\ \mathsf{as}\ \gamma) \to \\
\qquad ([i'_{\geq\langle nil\,cons, \top\rangle} \mid nil : unit :: cons : \beta \times \delta :: \rho']\ \mathsf{as}\ \delta)
\end{array}
$$

Written using naive types, as we did in section 2, this boils down to:

$$
\begin{array}{l}
map : (\alpha \to \beta) \to \\
\qquad \gamma[< nil\ cons(\alpha \times \gamma)] \to \delta[> nil\ cons(\beta \times \delta)]
\end{array}
$$

Since there is a bijection between naive and formal types, we are not hiding anything to the user, but only using a simpler notation.

The situation becomes more subtle if we use a more complex notion of type identity and sharing. This happens to be necessary when using a combination of first-class polymorphism and value-restricted polymorphism [GR97], recently introduced in Objective Label. Then two types may have the same denotation (same structure and same type variables), but still need to be distinguished at a lower level (in [GR97], their nodes may be labeled differently). Two variant types sharing the same i and ρ may actually not be physically shared. Since denotation is the same, we still choose to display this case as sharing in the naive type system, keeping the labeling information hidden to the user.

7 Refinements and extensions

The system presented above only provides basic features for polymorphic variants. It can be made smoother by adding a few extensions.

7.1 Discarding superfluous typing information

We have seen that for the sake of completeness of unification, one cannot discard typing information in variant types, even if a tag is not included in the upper bound of the variant.

However, this problem only appears during unification. After unification is finished and a substitution is obtained, nothing opposes discarding this superfluous (and maybe harmful) information.

Still we want to do this independently of the type inference algorithm. Doing this discarding when generalizing types seems the right thing to do, since any algorithm has to finish unification there anyway.

$$
\textsc{Forget}\quad
\frac{K;\Gamma \vdash e : \forall i_{\geq\langle L,U\rangle}, \rho.\sigma\{[i \mid \{tag_k : \tau_k\}_1^{n+m} :: \rho]/\gamma\} \qquad i \notin FTV(\sigma), \rho \notin FTV(\sigma)}{K;\Gamma \vdash e : \forall i_{\geq\langle L,\{tag_k\}_1^n\rangle}, \rho.\sigma\{[i \mid \{tag_k : \tau_k\}_1^{n} :: \rho]/\gamma\}}
$$

This rule as such would be difficult to implement, since it implies checking for simpler variant types at every step of type reconstruction. If we restrict its use to inside let expressions, this can be inferred easily by modifying $Clos'$. Such a restricted form would break subject reduction though.

VARIABLE

$$K;\Gamma,x:\sigma \vdash x:\sigma$$

ABSTRACTION

$$\frac{K;\Gamma,x:\tau \vdash e:\tau'}{K;\Gamma \vdash \lambda x.e:\tau \to \tau'}$$

APPLICATION

$$\frac{K;\Gamma \vdash e_1:\tau \to \tau' \quad K,\Gamma \vdash e_2:\tau}{K;\Gamma \vdash e_1\, e_2:\tau'}$$

LET

$$\frac{K;\Gamma \vdash e_1:\sigma \quad K;\Gamma,x:\sigma \vdash e_2:\tau}{K;\Gamma \vdash \mathsf{let}\ x=e_1\ \mathsf{in}\ e_2:\tau}$$

KIND

$$K \oplus i_{\geq\langle L,U\rangle} \vdash i \geq \langle L',U'\rangle \quad \text{if } L \supset L', U \subset U'$$

VARIANT

$$\frac{K;\Gamma \vdash e:\tau \qquad K \vdash i \geq \langle tag,\top\rangle}{K;\Gamma \vdash \text{`}tag(e):[i \mid tag:\tau :: T]}$$

CASE

$$\frac{\begin{array}{l} K;\Gamma \vdash e:[i \mid \{tag_k:\tau_k\}_1^n :: T] \quad K \vdash i \geq \langle \emptyset,\{tag_k\}_1^n\rangle \\ K;\Gamma,x_k:\tau_k \vdash e_k:\tau' \ (1 \leq k \leq n)\end{array}}{K;\Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ \{\text{`}tag_k(x_k) \to e_k\}_1^n:\tau'}$$

POLY α

$$\frac{K;\Gamma \vdash e:\sigma}{K;\Gamma \vdash e:\forall\alpha.\sigma}\ \alpha \notin FTV(\Gamma)$$

INST α

$$\frac{K;\Gamma \vdash e:\forall\alpha.\sigma}{K;\Gamma \vdash \sigma\{\tau/\alpha\}}$$

POLY ρ

$$\frac{K;\Gamma \vdash e:\sigma}{K;\Gamma \vdash e:\forall\rho.\sigma}\ \rho \notin FTV(\Gamma)$$

INST ρ

$$\frac{K;\Gamma \vdash e:\forall\rho.\sigma}{K;\Gamma \vdash \sigma\{T/\rho\}}$$

POLY i

$$\frac{K \oplus i_{\geq\langle L,U\rangle};\Gamma \vdash e:\sigma}{K;\Gamma \vdash e:\forall i_{\geq\langle L,U\rangle}.\sigma}\ i \notin FTV(\Gamma)$$

INST i

$$\frac{K;\Gamma \vdash e:\forall i_{\geq\langle L,U\rangle}.\sigma \quad K \vdash i' \geq \langle L,U\rangle}{K;\Gamma \vdash e:\sigma\{i'/i\}}$$

Figure 3: Inference rules for the formal type system

7.2 Open matching

In the basic system we only provide a closed version of matching: all cases must be provided. Some examples will require an open version of matching, with a default case.

$$e ::= \ldots \mid \mathsf{case}\ e\ \mathsf{of}\ \text{`}tag(x) \to e;\ldots;\text{`}tag(x) \to e\ \mathsf{else}\ e$$

The associated typing rule is:

CASE ELSE

$$\frac{\begin{array}{l} K;\Gamma \vdash e:[i \mid \{tag_k:\tau_k\}_1^n :: T] \quad K \vdash i \geq \langle \emptyset,\top\rangle \\ K;\Gamma,x_k:\tau_k \vdash e_k:\tau' \ (1 \leq k \leq n) \quad K;\Gamma \vdash e_0:\tau'\end{array}}{K;\Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ \{\text{`}tag_k(x_k) \to e_k\}_1^n\ \mathsf{else}\ e_0:\tau'}$$

Notice that this rules adds no presence information, only typing information.

```
let show x =
  case x of `apple -> "apple";
            `orange(s) -> "orange "^s
  else "pear"
show : [apple orange(string) ...] -> string
```

Beware also that the typing introduced by this construct is weak. Many "errors" will not be detected. For instance, if we misspell ‘*apple* into ‘*aple*, we get the following result.

```
show `aple
- : string = "pear"
```

7.3 Variant dispatching

When working with variants and subtyping, a quite natural thing one may want to do is to divide a variant in smaller subtypes (subsets of tags), and to dispatch according to which subtype the variant belongs to. This can also be compared with delegation in an object-oriented framework.

The case statement may do that, but it results in superfluous work: the dispatching must be done individually for each tag. Having this feature as a primitive construct is useful.

$$e ::= \ldots \mid \mathsf{select}\ e\ \mathsf{of}\ \text{`}tag\ldots\text{`}tag\ \mathsf{as}\ x \to e;\ldots;\text{`}tag\ldots\text{`}tag\ \mathsf{as}\ x \to e$$

The typing rule comes as follows.

SELECT

$$\frac{\begin{array}{l} K;\Gamma \vdash e:[i \mid \{tag_{kj}:\tau_{kj}\}_{k=1..n,j=1..l_k} :: T] \\ K \vdash i \geq \langle \emptyset,\{tag_{kj}\}_{k=1..n,j=1..l_k}\rangle \\ K;\Gamma,x_k:[i_k \mid \{tag_{kj}:\tau_{kj}\}_{j=1..l_k} :: T_k] \vdash e_k:\tau' \\ K \vdash i_k \geq \langle \{tag_{kj}\}_{j=1..l_k},\top\rangle \qquad (1 \leq k \leq n)\end{array}}{K;\Gamma \vdash \mathsf{select}\ e\ \mathsf{of}\ \{\text{`}tag_{k1}\ldots\text{`}tag_{kl_k}\ \mathsf{as}\ x_k \to e_k\}_1^n:\tau'}$$

select has another nice property: it permits to create a polymorphic variant from a monomorphic one, by breaking the input-output relation.

```
let f x = select x of `left`right as x -> x
f : [< left(α) right(β)] -> [> left(α) right(β)]
```

This function does nothing: since the typing makes sure that only *a* or *b* will come here, there is no need for any runtime check. But it gives different variant types to its input and output. Since the output type is a newly created one, it is polymorphic, even if the input type has to be unified with a monomorphic one.

7.4 Subtyping through coercions

As with Objective ML, not all forms of subtyping may be expressed by structural polymorphism. Full subtyping may be added has a coercion operator.

$$e ::= \ldots \mid (e:\tau :> \tau')$$

Where τ' is a supertype of τ according to an appropriate subtyping relation.

A typing rule for this is simply:

COERCE

$$\frac{K,\Gamma \vdash e:\tau \qquad \tau \prec \tau'}{K,\Gamma \vdash (e:\tau :> \tau'):\tau'}$$

The subtyping relation $\prec$ may be chosen freely, as long as it is compatible with the rest of the type system. We will not develop more on this aspect.

8 Final remarks

We have described in this paper a complete approach to polymorphic variant typing. This includes user-friendly type representation and features, efficient and portable compilation scheme, and an extendible type system with its reconstruction algorithm.

This description is based on the Objective Label system, but is not completely faithful. There are some rough edges in the system, and we preferred describing the "right thing" rather than explaining why we chose another way. There are also differences in the syntax: in Objective Label `case` and `select` are all integrated in the pattern matching mechanism; describing pattern matching as a whole was not the goal of this paper.

We conclude on a technical remark: we explained that the expressive power of our system is equivalent to Rémy's, in terms of expressible variant types. However, Rémy's system is strictly stronger when we consider constraints expressible between two different variant types. Some useful features can be encoded using this mechanism, but this would break our assumption that two variant types are either equal or independent. Since this assumption is required to keep types readable, we must stick to it. At the language level, this weakness is compensated by variant dispatch and coercions.

References

[CV] Koen Claessen and Ton Vullings. The TkGofer home page. URL `http://www.informatik.uni-ulm.de/abt/pm/ftp/tkgofer.html`.

[Gar] Jacques Garrigue. The Objective Label trilogy. URL `http://wwwfun.kurims.kyoto-u.ac.jp/soft/olabl/`.

[GR97] Jacques Garrigue and Didier Rémy. Extending ML with semi-explicit higher-order polymorphism. In Abadi and Ito, editors, *Proc. of the International Conference on Theoretical Aspects of Computer Software*, number 1281 in Springer-Verlag LNCS, pages 20–46, Sendai, Japan, September 1997.

[Ler] Xavier Leroy. Objective Caml. URL `http://pauillac.inria.fr/ocaml/`.

[Oho95] Atsushi Ohori. A polymorphic record calculus and its compilation. *ACM Transactions on Programming Languages and Systems*, 17(6):844–895, November 1995. Long version of a POPL'92 paper.

[Rém89] Didier Rémy. Typechecking records and variants in a natural extension of ML. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 77–87, 1989.

[RV97] Didier Rémy and Jérôme Vouillon. Objective ML: A simple object-oriented extension of ML. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 40–53, January 1997.

A Type reconstruction

A.1 Unification algorithm

We give here a unification algorithm for the monotypes defined above. A unification problem is a conjunction of multi-equations. It is described by the following grammar.

$$
\begin{array}{rcl}
\phi & ::= & \emptyset \mid \phi \wedge e \\
e & ::= & e_t \mid e_r \mid e_i \\
e_t & ::= & \emptyset \mid \tau = e_t \\
e_r & ::= & \emptyset \mid T = e_r \\
e_i & ::= & \emptyset \mid i = e_i \mid \langle L, U \rangle = e_i
\end{array}
$$

$\wedge$ and $=$ are associative and commutative.

A unification problem ϕ is in solved form when:

- the same variable does not appear naked in more than one multi-equation, and
- each multi-equation contains only one non-variable term.

One may directly read a solution substitution from a solved form.

In figure 4, we give the unification algorithm as a set of rewriting rules of the form $\frac{\mathit{before}}{\mathit{after}}$.

Rules are divided in 3 groups, by order of priority: the Merge and Concatenate rules, failure rules, and others. In each of these groups, while no rule of an higher priority group may be applied, any rule of the group can be applied in any order.

In Arrow and Variant, we keep the smallest of the two original terms in the shortened multi-equation. The size $|m|$ of a term m is defined as the number of symbols (excluding all variables).

In Concatenate and Row clash, e and e' may be referring to the same multi-equation.

Occur-check is intentionally not included in the above rules. One probably wants to add it for non-variant types, but without it the algorithm infers regular types.

Proposition 1 *A normal form for the above rewriting system is a unification problem in solved form.*

Lemma 2 *All rules are sound and complete.*

Lemma 3 *Unification terminates.*

The measure is the lexicographical ordering of

- the number of unsolved variables (a variable is solved when it appears in an equation containing at least one non-variable term),
- the sum of monomials $X^{|m|}$ for m any member of a multi-equation.

A.2 Type reconstruction algorithm

The type reconstruction algorithm is given in figure 5. We start from a monomorphic judgment scheme, and convert it to a unification problem. Kinding assumptions are eliminated first. The only case where a unification problem must be solved locally is for `let`: the resulting substitution is needed for deciding which variables are polymorphic.

MERGE
$$\frac{\phi \wedge a = e \wedge a = e'}{\phi \wedge a = e = e'} \; a \in \{\alpha, \rho, i\}$$

REDUNDANCY
$$\frac{\phi \wedge a = a = e}{\phi \wedge a = e} \; a \in \{\alpha, \rho, i, u\}$$

CLASH
$$\frac{\phi \wedge \tau = \tau' = e}{\bot} \; sort(\tau) \neq sort(\tau')$$

SORTS
$$sort(u) = u \qquad sort(\tau \to \tau') = \to \qquad sort([i \mid T]) = \diamond$$

CONCATENATE
$$\frac{\phi \wedge \{tag_i : \tau\}_1^m :: \rho = e \wedge \rho = \{tag'_i : \tau'_i\}_1^n :: \rho' = e'}{\phi \wedge \{tag_i : \tau\}_1^m :: \{tag'_i : \tau'_i\}_1^n :: \rho' = e \wedge \rho = \{tag'_i : \tau'_i\}_1^n :: \rho' = e'}$$

COMPLETION
$$\frac{\phi \wedge \{tag_i : \tau_i\}_1^m :: \rho = \{tag'_i : \tau'_i\}_1^n :: \rho' = e \qquad (\forall i,j)\, tag_i \neq tag'_j, \rho'' \text{ fresh}}{\phi \wedge \rho' = \{tag_i : \tau_i\}_1^m :: \rho'' \wedge \rho = \{tag'_i : \tau'_i\}_1^n :: \rho'' \wedge \{tag_i : \tau_i\}_1^m :: \{tag'_i : \tau'_i\}_1^n :: \rho'' = e}$$

ROW CLASH
$$\frac{\phi \wedge \{tag_i : \tau_i\}_1^m :: \rho = \{tag'_i : \tau'_i\}_1^n :: \rho' = e \wedge \rho = \rho' = e'}{\bot} \; (\forall i,j)\, tag_i \neq tag'_j$$

ARROW
$$\frac{\phi \wedge \tau_1 \to \tau_2 = \tau'_1 \to \tau'_2 = e}{\phi \wedge \tau_1 = \tau'_1 \wedge \tau_2 = \tau'_2 \wedge \tau_1 \to \tau_2 = e}$$

VARIANT
$$\frac{\phi \wedge [i \mid T] = [i' \mid T'] = e}{\phi \wedge i = i' \wedge T = T' \wedge [i \mid T] = e}$$

ROW
$$\frac{\phi \wedge tag : \tau :: T = tag : \tau' :: T' = e}{\phi \wedge \tau = \tau' \wedge T = T' \wedge tag : \tau :: T = e}$$

CONSTRAINT MERGE
$$\frac{\phi \wedge \langle L, U\rangle = \langle L', U'\rangle = e}{\phi \wedge \langle L \cup L', U \cap U'\rangle = e} \; (L \cup L') \subset (U \cap U')$$

CONSTRAINT CLASH
$$\frac{\phi \wedge \langle L, U\rangle = \langle L', U'\rangle = e}{\bot} \; (L \cup L') \not\subset (U \cap U')$$

Figure 4: Unification rules

$$\begin{array}{rcl}
\mathsf{Type}(\Gamma \triangleright x : \tau) & = & instance(\Gamma(x), \tau) \\
\mathsf{Type}(\Gamma \triangleright \lambda x.e : \tau) & = & \mathsf{Type}(\Gamma, x : \alpha_1 \triangleright e : \alpha_2) \wedge \tau = \alpha_1 \to \alpha_2 \\
\mathsf{Type}(\Gamma \triangleright e_1\, e_2 : \tau) & = & \mathsf{Type}(\Gamma \triangleright e_1 : \alpha \to \tau) \wedge \mathsf{Type}(\Gamma \triangleright e_2 : \alpha) \\
\mathsf{Type}(\Gamma \triangleright \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau) & = & \phi \wedge \mathsf{Type}(\Gamma, x : Clos(\Gamma, \phi, \alpha) \triangleright e_2 : \tau) \quad \mathsf{where}\ \phi = \mathsf{Type}(\Gamma \triangleright e_1 : \alpha) \\
\mathsf{Type}(\Gamma \triangleright \text{`}tag(e) : \tau) & = & \mathsf{Type}(\Gamma \triangleright e : \alpha) \wedge \tau = [i \mid tag : \alpha :: \rho] \wedge i = \langle tag, \top\rangle \\
\mathsf{Type}(\Gamma \triangleright \mathsf{case}\ e\ \mathsf{of}\ \{\text{`}tag_k(x_k) \to e_k\}_1^n : \tau) & = & \mathsf{Type}(\Gamma \triangleright e : [i \mid \{tag_k : \alpha_k\}_1^n :: \rho]) \wedge i = \langle \emptyset, \{tag_k\}_1^n\rangle \wedge \bigwedge_1^n \mathsf{Type}(\Gamma, x_k : \alpha_k \triangleright e_k : \tau)
\end{array}$$

$$\begin{array}{rcll}
instance(\forall \alpha.\sigma, \tau) & = & instance(\sigma, \tau) & \alpha \text{ fresh} \\
instance(\forall \rho.\sigma, \tau) & = & instance(\sigma, \tau) & \rho \text{ fresh} \\
instance(\forall i_{\geq \langle L, U\rangle}.\sigma, \tau) & = & instance(\sigma, \tau) \wedge i = \langle L, U\rangle & i \text{ fresh} \\
instance(\tau', \tau) & = & (\tau' = \tau) & \\
 & & & \\
Clos(\Gamma, \phi, \tau) & = & Clos'(\phi(\Gamma), \phi, \phi(\tau)) & \\
Clos'(\Gamma, \phi, \sigma) & = & Clos'(\Gamma, \phi, \forall \alpha.\sigma) & \mathsf{when}\ \alpha \in FTV(\sigma) \setminus FTV(\Gamma) \\
Clos'(\Gamma, \phi, \sigma) & = & Clos'(\Gamma, \phi, \forall \rho.\sigma) & \mathsf{when}\ \rho \in FTV(\sigma) \setminus FTV(\Gamma) \\
Clos'(\Gamma, \phi, \sigma) & = & Clos'(\Gamma, \phi, \forall i_{\geq \phi(i)}.\sigma) & \mathsf{when}\ i \in FTV(\sigma) \setminus FTV(\Gamma) \\
Clos'(\Gamma, \phi, \sigma) & = & \sigma & \mathsf{when}\ FTV(\sigma) \subset FTV(\Gamma)
\end{array}$$

Figure 5: Abstract type reconstruction algorithm