Generic polymorphism in ML

J. Furuse

INRIA Rocquencourt

Résumé

Generic values are ad-hoc polymorphic — the behavior of their instances depends on the contexts where they are typed. Using generic values we can implement interesting functions with type manipulations, which are impossible in the normal ML language. We propose a simple extension of ML which can define such generic values. A generic value is compiled as a function which receives its evaluation context as run time types and selects the appropriate type case for the context efficiently using ML pattern matching.

To detect the illegal use of generic values which cannot have any semantics, we developed a new layer of typing, called generic constraint typing system, partially independent of the normal ML typing system. This two-layered design makes the type algebra simpler and the integration to the existing implementation easier.

The generic constraint type checking is the simulation of the possible run time type matching of the generic instances. If we use the result of this "pre-computation", we can compile the flow control of generic values to very simple automata and significantly improve the evaluation speed of generic values.

1. Introduction

ML is a great language. Based on the typed λ-calculus, programs are checked and proved by its type system so that the well typed programs never produce errors. Once the type soundness of the programs is checked, we no longer need any type checking at run time. Therefore by getting rid of all the type information from the execution code, ML programs can be evaluated efficiently.

But this lack of types at run time sacrifices our chances to write programs which use type based computations. Though the parametric polymorphism is powerful abstraction mechanism of ML, what we can write with it are not type based computations, but contrary, only the operations which do not depend on the parameter types. Non parametric, ad-hoc polymorphic values which work differently according to their types are sometimes useful in practice (we call them *generic polymorphic values*). Overloaded arithmetic functions defined over integers and floats are the simplest example of such generic values. For instance, a generic `plus` function can be defined in our language like below, as the list of the *type cases*, pairs of a pattern type and its corresponding expression:

```
generic plus = case
| int → int → int ⇒ add_integer
| float → float → float ⇒ add_float
```

Depending on how it is typed, it calls the different adding primitive function:

```
plus 1 2       ⟹ 3
plus 3.14 1.0 ⟹ 4.14
```

To our knowledge, there are some implementations of ML-like systems which support some form of such generic values like generic equality or comparison primitives, but their power is insufficient.

In SML, some basic math operators including `(+)` are predefined as overloaded, but generic value definitions are not available neither, in spite of using such overloaded operators. Let's see the famous `double` example:

```
let double x = x + x
```

The overloading of `(+)` is not derived to this `double`. In SML'97[9], the types of the overloaded operators must be statically deduced from the context, otherwise their default types are used. In the above example, `(+)`'s type is unified with its default `int` → `int` → `int` and `double` has a monomorphic type `int` → `int`. To avoid the run time type operations, the overloading of SML is thus restricted and somewhat trivial. Some people find the type classes provided by Haskell[7] or Alcool[8] useful for their purposes, but their class based type system is something different from the generic polymorphism. Overloaded functions like `(+)` can be defined without problem, but some of the interesting recursive generic functions (such as `flat` and functions with variable length arguments appear later) cannot be defined due to the restriction of its type system. In addition, the type classes introduce a new programming methodology, which is not essentially required to achieve the generic values. Caml languages lack overloading facility, therefore each numeric type has its own plus function such as `(+)` for integers and `(+.)` for floats and many programmers annoy them. The classes of the Objective Caml[10] can mimic overloaded functions, but by the same reason for the Haskell type classes, they are not appropriate.

Because generic values are defined by sets of type cases, their types should be expressed as the sum of their defined types, which cannot be expressed in the normal ML type system. For example, what we can give to `plus` inside the normal type system is $\forall\alpha.\alpha \rightarrow \alpha \rightarrow \alpha$, but it is too general. The type system must know that `int` → `int` → `int` and `float` → `float` → `float` are the valid typings for `plus`. Direct integration of these information to the ML type system makes it complicated and the clarity of the type system will be lost. It seems quite hard to implement such type theory and difficult for the users to understand error messages it produces.

In this paper we propose a generic polymorphism system which is powerful enough but yet possible to integrate rather easily in an existing ML language based on the extensional polymorphism[4]. To this goal, two layered typing systems are adapted. The first layer is the classical ML typing with minimal modification. It can handle generic values but is not aware of the partiality of their polymorphism. The programs which pass this layer are checked by the second typing layer called *constraint typing*, which ensures that all the uses of the generic values are valid.

Usually the compilation of the generic polymorphism is by run time type dispatching [3, 4, 5, 6]. A generic value is translated into a function which receives run time types which provide its evaluation context then selects one of the appropriate definitions under it. The type case selection is compiled as ML pattern matching efficiently, but its run time overhead is still discouraging heavy use of generic values. We also propose a new radical compilation strategy which translates this run time type matching into very simple array access obtained from the type soundness proof by the constraint typing. The overhead of the run time case selection becomes constant, no matter how the type patterns are complex.

2. Overview

A generic value can be characterized by a set of pairs of a type scheme and a expression called *type cases*, the type scheme for which the value is defined and the corresponding expression which provides the semantics for the context. For example:

```
generic plus = case
| int → int → int ⇒ (+)
```

```
| float → float → float ⇒ (+.)
| ∀δ′.δ′ list → δ′ list → δ′ list ⇒ (@);;
```

Each type scheme in the type case is called *type pattern* (or just pattern) and the corresponding right-hand side expression is its *type method* (or just method). This *generic binding* expression will define a generic value `plus` whose type scheme is $\forall\delta.\ \delta \rightarrow \delta \rightarrow \delta$ in the normal ML typing layer. We use special type variable δ called *dynamic type variable* in the type scheme of `plus` to indicate this value is not normally parametric polymorphic value but a generic one: its instance is evaluated differently depending on how its quantified dynamic type variables are instantiated. This type scheme will be inferred automatically by the type system as the *anti-unifier* of the type patterns.

As already we have seen in the previous section, we can use this `plus` generic function for adding integers and floats, and this time polymorphic list concatenation in addition:

```
# plus ["hello"] ["world"];;
- : string list = ["hello"; "world"]
```

The behaviors of the *generic instances* of `plus` differ based on their *evaluation context*, the type under which they are evaluated. Thus, using generic bindings, we can overload functions or values quite easily.

Derived generics

Using already defined generic values, we can define other generic values. For instance we can define `double` function using `plus`:

```
let double x = plus x x;;
val double : ∀δ.δ → δ
```

Because `plus` is used polymorphically inside this definition, its generic polymorphism is inherited to `double`. Therefore, even though `double` is defined by a let binding, it becomes generic polymorphic and we can use it to double integers, floats and lists. We call this kind of generic values as *derived generic values*.

Recursive generics

Generic definitions can be recursive also. Recursive generic definitions enable us to write recursions over the structure of *types*, which cannot be done in the normal ML:

```
generic rec print = case
| int → unit ⇒ print_int
| string → unit ⇒ print_string
| ∀δ′.δ′ list → unit ⇒ function [] → ()
                        | x::xs → print x; print xs;;
val print : ∀δ.δ → unit
```

The last type method receives a list and if it is non-nil, say `x::xs`, it calls `print` twice recursively. The second occurrence of `print` for the tail `xs` is a normal recursive call on the structure of the list cons cells, which is familiar to us. But the first one is something new to the ML world. This is a recursion on the structure of the type of the argument of `print`. It prints the content of the `x`, using its corresponding type method. This recursion on types enables printing not only lists of `int` and `string` but also their lists of any depth like `int list list list` :

```
# print [[[1];[2;3]]; [[4;5]]];;
12345
```

Another example, the following generic function `flat` demonstrates the power of the generic polymorphism: it can flatten a list of any depth:

```
generic rec flat = case
| ∀δδ'.δ list list → δ' list ⇒ fun l → flat (List.flatten l)
| ∀δ.δ list → δ list ⇒ fun l → l;;
val flat : ∀δδ'.δ list → δ' list
```

Here, `List.flatten` is a normal polymorphic function which flatten just one level of a list. The second type case is the base case of this recursion and used to map lists of depth one to themselves. The first type case receives a multi depth list and flattens its outermost level using `List.flatten` and calls recursively itself on this flattened list. The recursive calls of `flat` selects one of these type cases depends on which type it is evaluated and finally returns a list of level 1:

```
# flat [[[1;2;3]];[[4;5]]];;
- : int list = [1;2;3;4;5]
```

Dubois et al[4] have shown some such interesting functions such as a tautology checker using this recursive generic bindings which involve with type manipulations. Therefore here we present just one more:

```
generic rec vprint = case
| ∀δδ'.δ → δ' ⇒ fun x → print x; vprint
| unit ⇒ ()
val vprint : ∀δ.δ
# (vprint 1 "hello" [2;3] : unit);;
1hello23
```

This `vprint` is a printing function with variable length arguments. If it is evaluated as a function, it receives an argument and print it using the generic `print`, then returns `vprint` itself. Thus `vprint` receives arbitrary number of arguments until it is evaluated under the type `unit`.

Normal typing system

The programs are first checked by the *normal tying system*, which is very similar to the one in the extensional polymorphism[4]. This type system based on the Damas-Milner's[1] has the minimum extensions to handle generic bindings and their instances.

In the language we have the new generic binding. Each type case expression tc must have a type pattern explicitly, the type scheme of the type method:

$$
\begin{array}{rcll}
e & ::= & c_\sigma \mid \mathtt{x} \mid e\ e \mid e, e \mid \mathtt{fun}\ \mathtt{x} \rightarrow e \mid \mathtt{let}\ (\mathtt{rec})\ \mathtt{x} = e\ \mathtt{in}\ e & \\
 & \mid & \mathtt{generic}\ (\mathtt{rec})\ \mathtt{x} = \mathtt{case}\ tc \mid \ \ldots \mid tc\ \mathtt{in}\ e & \text{Generic bindings} \\
tc & \equiv & \sigma \Rightarrow e & \text{Type case expressions}
\end{array}
$$

The generic values are distinguished from the normal polymorphic values by carrying generalized dynamic type variables in their type schemes. Because of the minimum amount of the extension to the type system, this normal typing system cannot provide complex type information which the generic values really require. Usually generic values have type schemes too general compared to their type patterns. For example, `print` function is given a type scheme $\forall\delta.\delta \rightarrow \mathtt{unit}$, but its definition only covers limited set of types (`int` → `unit`, `string` → `unit`, `int list` → `unit`, `string list` → `unit`, etc). Therefore, the type soundness property which the normal typing system can provide is just for the non generic part of the language. The soundness of the generic values is checked later by the second typing layer, the constraint typing system.

Classical compilation by pattern matching

Even though the type soundness theorem of the normal typing is limited, thanks to it, the non-generic part of GML expressions can be compiled and evaluated as efficient as the normal ML, since they do not need any run time type checking. On the other hand, the generic polymorphic part requires some run time type checking. For the compilation, we translate the GML expressions into a intermediate language, untyped ML with the *run time type* constructors and their pattern matching. The run time type is a data type which is equivalent to the type expression. We write «τ» for the run time type representation for τ.

The generic instances are compiled as function application of the compilations of their values on the run time types of the evaluation context. For instance an expression `print [1;2]` is compiled to `print «int list» [1;2]` because its quantified dynamic type variable is instantiated to `int list`.

To receive these implicitly applied run time types, the generic bindings are translated to functions. The received run time types provide their evaluation contexts. The type case selection is done by pattern matching of these run time types. For instance, `print` generic definition is compiled to the following code:

```
let rec print = function
| «int» → print_int
| «string» → print_string
| «d list» → (function [] → ()
                        | x::xs → print «d» x; print «d list» xs;;
```

For example, in the evaluation of `print «int list» [1;2]`, a run time type `«int list»` is sent to the compiled `print` and matches with the last case. The pattern variable `d` is bound by `«int»`. This binding properly delivers the evaluation contexts of the recursive calls: `print «int» 1` and `print «int list» [2]`.

Constraint typing system

Because generic values are defined by finite set of type cases, generally they are just *partially polymorphic*: their type schemes which are inferred by the normal typing are too general and their semantics are not defined for all their instances. Thereore if a generic value is evaluated under an undefined context, it raises an error, *generic type match failure*. For example, `print 3.14` is compiled to `print «float» 3.14`, but the compilation of `print` has no pattern for `«float»`, therefore it will not be evaluated correctly and will generate a pattern match failure. The static detection of such invalid generic instances can be achieved only by a type system which is aware of the partiality of the generic values.

Rather than extending directly the normal type system to handle the partial type case information, we define another layer of typing called *generic constraint typing* which is independent of the normal typing. It receives programs already type checked and fully annotated by the first layer and focuses just on the partial polymorphism of the generic values to detects the illegal generic instances.

The notion of the type schemes are extended in this level. An *extended type scheme* has extra information for the generic partiality called *generic constraint* in addition to a normal type scheme.

For example the `plus` generic function will have the following extended type scheme:

$$\begin{aligned}
\texttt{plus : } \forall\delta.\delta \to \delta \to \delta\ \texttt{[} & \texttt{int} \to \texttt{int} \to \texttt{int} \\
\texttt{| } & \texttt{float} \to \texttt{float} \to \texttt{float} \\
\texttt{| } & \forall\delta'.\delta'\ \texttt{list} \to \delta'\ \texttt{list} \to \delta'\ \texttt{list]}
\end{aligned}$$

whose constraint consists of the list of its type patterns.

The derived generic values must also derive the constraints of their generic ancestors. If we let $\eta_{\texttt{plus}}$ be the extended type scheme for `plus`, then the constraint of `double` will be the following, a list of one type pattern $\forall\delta.\delta \to \delta$ attached with a *derived constraint*[1] $\{\delta \to \delta \to \delta <: \eta_{\texttt{plus}}\}$:

$$\texttt{double : } \forall\delta.\delta \to \delta\ \texttt{[}\forall\delta.\delta \to \delta\ \{\delta \to \delta \to \delta\ <:\ \eta_{\texttt{plus}}\}\texttt{]}$$

This means the instantiation of the quantified variable δ of `double` is restricted to the types τ where $\tau \to \tau \to \tau$ is a valid instance of `plus`.

We also have *recursive constraints* for the constraints of the recursively defined generic values:

$$\begin{aligned}
\texttt{print : } \forall\delta.\delta \to \texttt{unit}\ \mu\kappa.\texttt{[} & \texttt{int} \to \texttt{unit | string} \to \texttt{unit} \\
\texttt{| } & \forall\delta'.\delta'\ \texttt{list} \to \texttt{unit}\ \{\delta' \to \texttt{unit <: } \forall\delta.\delta \to \texttt{unit}\ \kappa\ \& \\
& \qquad\qquad\qquad \delta'\ \texttt{list} \to \texttt{unit <: } \forall\delta.\delta \to \texttt{unit}\ \kappa\}\texttt{]}
\end{aligned}$$

Constraint variables κ and the fixed point operator μ are introduced to express the constraints for the recursive generics. In the third type pattern, the constraints for the recursive occurrences of `print` are replaced by the constraint variable κ, and the whole constraint is abstracted by a μ operator.

The type checking of a generic instance $\texttt{x} : \tau$ is a seek whether τ is in the valid instance set of `x` characterized by its constraint, but it is undesidable unless we add some restrictions to constraints.

New compilation using the flow pre-computation

Once the complete type soundness of programs is statically verified by the both typing system, the run time type soundness checking by pattern matching is just redundant. The type matching exists in the compilation code only for the flow control of type cases. Since what the constraint typing system actually does is the pre-computation of this flow control, if an expression is verified by the type system, we know exactly how the type case selections will take place in the evaluation. Using the *generic flow information* obtained from this pre-computation instead of run time type matching, we can simplify the code for the flow control and improve its efficiency.

The generic flows are integer trees possibly with loops. Each node of the generic flow corresponds with each generic type case selection and indicates which type case should be selected. Each sub-tree of the node is the flow for a generic instance occur in the selected type method. For example, in the expression of `print [1;2;3]` , `print` has the type $\texttt{int list} \to \texttt{unit}$ and will have the following generic flow:

$$\begin{tikzcd}
\texttt{2} \arrow[r, bend left] \arrow[loop below] & \texttt{0}
\end{tikzcd}$$

This means that at the first type case selection, the third one will be selected (the generic flow index starts from 0). The two branches of the node correspond with the two internal `print` generic instances in the third type method. The upper branch tells that the evaluation of the first occurrence of `print` will choose the first type case to print an integer `1`. The lower branch loops to the top of the flow to call `print [2;3]` recursively.

[1]For simplicity of the type algebra, let bound derived generics are treated as if they were defined by generic bindings with one type case.

The generic flow compilation translates generic instances as implicit applications of the instances on the corresponding generic flows, instead of run time types. A generic binding is compiled to a function which receives a flow, then choose one of the compiled type methods stored in an array according to the index of the flow.

3. The language

Types and schemes

As we have explained, we have two kinds of sets of type variables, one is the set $\mathbb{R}$ of regular type variables written α , β .... Another set is $\mathbb{D}$ of *dynamic type variables* δ , δ' .... To range on any kind of type variable we use ν. We also have some *base* types, like `unit` , `int` , function types and tuples.

$$
\begin{array}{rcll}
\tau & ::= & \mathit{base} \mid \alpha \mid \delta \mid \tau \rightarrow \tau \mid \tau * \tau & \text{Type expressions} \\
\sigma & ::= & \forall \nu^*.\tau & \text{Type schemes}
\end{array}
$$

The notion of free type variables of type expressions $FV(\tau)$ and schemes $FV(\sigma)$, and the bound variables of schemes $BV(\sigma)$ are defined as usual. We also define here tools to filter the regular and dynamic type variables from these sets:

$$
\begin{array}{ll}
RV(\sigma) = FV(\sigma) \cap \mathbb{R} & BRV(\sigma) = BV(\sigma) \cap \mathbb{R} \\
DV(\sigma) = FV(\sigma) \cap \mathbb{D} & BDV(\sigma) = BV(\sigma) \cap \mathbb{D}
\end{array}
$$

Admissible substitutions and instances

Substitutions are defined as usual. But to prevent mixing these two kinds of type variables using substitutions, we need to restrict substitutions to *admissible substitutions*. Admissible substitutions never map dynamic type variables to types with regular type variable. For example, $\{\alpha \Rightarrow \delta';\ \delta \Rightarrow \delta'' \rightarrow \delta''\}$ is admissible but $\{\delta \Rightarrow \alpha \rightarrow \delta'\}$ is not. Precisely:

Definition 1 (Admissible substitutions)
A substitution S is admissible if $\forall \delta \in Dom(S),\ \forall \nu \in FV(S(\delta)),\ \nu \in \mathbb{D}$

In our system all the substitutions must be admissible, therefore we omit "admissible" and just call them substitutions unless special care must be taken.

The instances of type schemes are defined also as usual, but its instantiation must be restricted to admissible one:

Definition 2 (Instances) *A type expression τ is an* instance *of a type scheme $\sigma = \forall \nu_1 \ldots \nu_n.\tau'$, if there is an admissible substitution S such that $Dom(S) = \{\nu_i\}$ and $\tau = S(\tau')$, and we write $\tau \leq \sigma$.*

We also write $\tau \leq_S \sigma$, if $\tau \leq \sigma$ stands with a substitution S which suffice the above conditions. For instance, $\alpha \rightarrow \delta \leq_S \forall \alpha\beta.\alpha \rightarrow \beta$ where $S = \{\alpha \Rightarrow \alpha,\ \beta \Rightarrow \delta\}$.

Language

Our language GML is an extension of ML with the generic binding expressions:

$$\begin{array}{lcll} e_t & ::= & c_\sigma : \tau \mid \mathtt{x}_\sigma : \tau \mid e_t\ e_t : \tau \mid e_t, e_t : \tau \mid \mathtt{fun}\ \mathtt{x} \rightarrow e_t : \tau & \\ & \mid & \mathtt{let}\ (\mathtt{rec})\ \mathtt{x}_{\{\nu\ldots\nu\}} = e_t\ \mathtt{in}\ e_t : \tau & \\ & \mid & \mathtt{generic}\ (\mathtt{rec})\ \mathtt{x}_\sigma = \mathtt{case}\ tc_t \mid \ldots \mid tc_t\ \mathtt{in}\ e_t & \text{Generic bindings} \\ tc_t & \equiv & \sigma \Rightarrow e_t & \text{Type case expressions} \end{array}$$

The expressions are defined as fully annotated for the compilation. But the annotations are not necessary in practice and inferred by our type reconstruction algorithm[2], except the type patterns of the generic bindings explicitly required. We do not need the explicit abstractions and applications of run time types which other formalizations [3, 5, 6] require for ad-hoc polymorphic values. Our compilation algorithm produces such run time type dispatching codes automatically.

The constants and variables have two kinds of type information: a type scheme attached to a value or primitive itself and a type expression which denotes how its each occurrence is instantiated. Let polymorphic bindings have additional type information, *generalized type variable sets*, the variables generalized in the polymorphic bindings. Generic bindings also have their type schemes as their additional annotations. We often write $e_t : \tau$, to emphasize that the outermost type annotation of an expression e_t is τ. Type annotations are often omitted, if they do not play any important role.

4. Normal type system

Generalization

We adopt the value only polymorphism restriction[2] to the type generalization. The generalizations are restricted to "*non expansive*" expressions. The definition of the predicate $generalizable(e_t)$ is as usual, with the addition for the new syntax construction:

$$generalizable(\mathtt{generic}\ (\mathtt{rec})\ \mathtt{x} = \ldots\ \mathtt{in}\ e_2) = generalizable(e_{t2})$$

The generalization function *Gen* is also defined normally. If $e_t : \tau$ is generalizable, we abstract the type variables which do not appear in the typing environment Γ freely.

$$\begin{array}{rl} Gen(\Gamma, e_t : \tau) = & \text{if } generalizable(e_t) \text{ then } \forall\nu_1 \ldots \nu_n.\tau \text{ where } \{\nu_1, \ldots \nu_n\} = FV(\tau) \setminus FV(\Gamma) \\ & \text{else } \forall\emptyset.\tau \end{array}$$

Typing rules

We use *typing judgments* $\Gamma \vdash e_t$. Sometimes we write it like $\Gamma \vdash e_t : \tau$ to make clear that the outermost type annotation of e_t is τ. The typing rules are shown in Figure 1.

In (t-letrec), the recursive identifier `f` is treated as monomorphic inside the typing of the definition `fun x` $\rightarrow e_{t1}$ as usual, however, in the conclusion expression, the type scheme annotations of `f` occur in the definition is replaced from $\mathtt{f}_{\tau_1}$ to $\mathtt{f}_{\forall V.\tau_1}$. This is for the proper evaluation of the recursive derived generic instances; they must be compiled also generic polymorphic inside the definitions.

The typing of generic bindings is almost explicit. The type of each type method must be generalized to the corresponding type pattern. The type scheme of the generic value must be one of the anti-unifier of its type patterns without regular type variables.

[2] We have no enough space to present this algorithm, which consists of two phases: the normal ML type reconstruction and generic type case resolution. It can reconstruct typing without explicit type constraints in almost cases, except when a full generic instance is used ambiguously and cannot be determined which type case should be used.

$$(\text{t-cnst})\ \frac{\tau \le \sigma}{\Gamma \vdash c_\sigma : \tau} \qquad (\text{t-inst})\ \frac{\Gamma(\mathtt{x}) = \sigma, \quad \tau \le \sigma}{\Gamma \vdash \mathtt{x}_\sigma : \tau} \qquad (\text{t-fun})\ \frac{\Gamma\{\mathtt{x} : \tau_1\} \vdash e_t : \tau_2}{\Gamma \vdash \mathtt{fun\ x} \to e_t : \tau_1 \to \tau_2}$$

$$(\text{t-app})\ \frac{\Gamma \vdash e_{t1} : \tau_1 \to \tau_2 \quad \Gamma \vdash e_{t2} : \tau_1}{\Gamma \vdash e_{t1}\ e_{t2} : \tau_2} \qquad (\text{t-tpl})\ \frac{\Gamma \vdash e_{t1} : \tau_1 \quad \Gamma \vdash e_{t2} : \tau_2}{\Gamma \vdash (e_{t1}, e_{t2}) : \tau_1 * \tau_2}$$

$$(\text{t-let})\frac{\begin{array}{c}\Gamma \vdash e_{t1} : \tau_1 \\ \forall V.\tau_1 = \mathit{Gen}(\Gamma, e_{t1} : \tau_1) \\ \Gamma\{\mathtt{x} : \forall V.\tau_1\} \vdash e_{t2} : \tau_2\end{array}}{\Gamma \vdash \mathtt{let\ x}_V \mathtt{=} e_{t1}\ \mathtt{in}\ e_{t2} : \tau_2} \quad (\text{t-letrec})\frac{\begin{array}{c}\Gamma\{\mathtt{f} : \tau_1\} \vdash (\mathtt{fun\ x} \to e_{t1}) : \tau_1 \\ \forall V.\tau_1 = \mathit{Gen}(\Gamma, (\mathtt{fun\ x} \to e_{t1}) : \tau_1) \\ \Gamma\{\mathtt{f} : \forall V.\tau_1\} \vdash e_{t2} : \tau_2\end{array}}{\Gamma \vdash \mathtt{let\ rec\ f}_V \mathtt{=} \{\mathtt{f}_{\tau_1} \Rightarrow \mathtt{f}_{\forall V.\tau_1}\}(\mathtt{fun\ x} \to e_{t1})\ \mathtt{in}\ e_{t2} : \tau_2}$$

$$(\text{t-generic})\ \frac{\begin{array}{c}\Gamma \vdash e_{ti} : \tau_i \quad RV(\tau_i) = \emptyset \quad \sigma_i = \mathit{Gen}(\Gamma, e_{ti} : \tau_i) \quad FV(\sigma_i) = \emptyset \\ \sigma_i \le \sigma \quad BRV(\sigma) = \emptyset \quad \Gamma\{\mathtt{x} : \sigma\} \vdash e_t : \tau\end{array}}{\Gamma \vdash \mathtt{generic\ x}_\sigma \mathtt{=case}\ \sigma_1 \Rightarrow e_{t1} \mid \ldots \mid \sigma_n \Rightarrow e_{tn}\ \mathtt{in}\ e_t : \tau}$$

$$(\text{t-genericrec})\ \frac{\begin{array}{c}\Gamma\{\mathtt{x} : \sigma\} \vdash e_{ti} : \tau_i \quad RV(\tau_i) = \emptyset \quad \sigma_i = \mathit{Gen}(\Gamma, e_{ti} : \tau_i) \quad FV(\sigma_i) = \emptyset \\ \sigma_i \le \sigma \quad BRV(\sigma) = \emptyset \quad \Gamma\{\mathtt{x} : \sigma\} \vdash e_t : \tau\end{array}}{\Gamma \vdash \mathtt{generic\ rec\ x}_\sigma \mathtt{=case}\ \sigma_1 \Rightarrow e_{t1} \mid \ldots \mid \sigma_n \Rightarrow e_{tn}\ \mathtt{in}\ e_t : \tau}$$

Figure 1: Typing rules

With this normal type system we can make GML evaluation partially type safe:

Theorem 1 (Partial type soundness) *Let e_t be a GML expression. If e_t has a typing $\emptyset \vdash e_t$, then its evaluation never raises an error except the run time generic type match failures.*

PROOF: The proof is omitted in this paper, but provable like the usual proof of the ML type soundness. □

5. Compilation of GML by pattern matching

Our compilation algorithm uses is same as one in [4] and uses run time type dispatch. The other studies [3, 5, 6] also use the same strategy. The algorithm is in Figure 2.

For the correct compilation, we must suppose that the quantified type variables of a type scheme $\forall\alpha_1 \ldots \alpha_m\delta_1 \ldots \delta_n.\tau$ are sorted in the order of their occurrences in τ to make the run time types passed to their correct destinations.

The algorithm takes a set of type variables V which are regarded as generalized outside of the compilation target. If V contains a dynamic type variable δ, a run time type which indicates how this δ_i is instantiated will be bound to the variable $\mathtt{d}_i$ specific to δ_i at the execution of the compilation. To refer this run time type information, the occurrence of δ_i in the e_t and τ must be replaced by $\mathtt{d}_i$ in their compilation.

The instances of generic values $\mathtt{x}_\sigma$ are compiled as a function application of themselves on the run time types for sending the evaluation contexts of each quantified dynamic type variables of σ. A derived generic let definitions is compiled as a function which receives evaluation contexts of its bound dynamic type variable. Its definition expressions e_{t1} must be compiled under the updated generalized type variable set $V + V'$, so that the received contexts can be delivered correctly to the internal generic instances. A generic definition is also compiled to a function like generic lets. The received contexts are used to determine which type method should be executed by pattern matching.

$$
\begin{array}{rcl}
\lfloor base \rfloor_V & = & \text{«}base\text{»} \\
\lfloor \tau_1 \rightarrow \tau_2 \rfloor_V & = & \text{«}\lfloor \tau_1 \rfloor_V \rightarrow \lfloor \tau_2 \rfloor_V\text{»} \\
\lfloor \tau_1 * \tau_2 \rfloor_V & = & \text{«}\lfloor \tau_1 \rfloor_V * \lfloor \tau_2 \rfloor_V\text{»} \\
\lfloor \delta_i \rfloor_V & = & \text{if } \delta_i \in V \text{ then } \mathtt{d}_i \\
 & & \text{otherwise, «}\delta_i\text{»}
\end{array}
\qquad
\begin{array}{rcl}
\lfloor c_\sigma : \tau \rfloor_V & = & c_\sigma \\
\lfloor e_{t1}\ e_{t2} \rfloor_V & = & \lfloor e_{t1} \rfloor_V\ \lfloor e_{t2} \rfloor_V \\
\lfloor e_{t1}, e_{t2} \rfloor_V & = & \lfloor e_{t1} \rfloor_V, \lfloor e_{t2} \rfloor_V \\
\lfloor \mathtt{fun\ x} \rightarrow\ e_t \rfloor_V & = & \mathtt{fun\ x}\ \rightarrow \lfloor e_t \rfloor_V
\end{array}
$$

$$
\begin{array}{l}
\lfloor \mathtt{x}_\sigma : \tau \rfloor_V = \ \text{let } \sigma = \forall \alpha_1 \ldots \alpha_m \delta_1 \ldots \delta_n.\tau' \text{ and } \tau \leq_S \sigma \\
\qquad\qquad\quad \text{in } \mathtt{x}\ \lfloor S(\delta_1) \rfloor_V\ \ldots\ \lfloor S(\delta_n) \rfloor_V \\
\lfloor \mathtt{let\ (rec)\ x}_{V'} = (e_{t1} : \tau)\ \mathtt{in}\ e_{t2} \rfloor_V = \ \text{let } \forall V'.\tau = \forall \alpha_1 \ldots \alpha_m \delta_1 \ldots \delta_n.\tau' \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{in } \mathtt{let\ (rec)\ x = fun\ d}_1\ \ldots\ \mathtt{d}_n \rightarrow \lfloor e_{t1} \rfloor_{V+V'}\ \mathtt{in}\ \lfloor e_{t2} \rfloor_V \\
\lfloor \mathtt{generic\ (rec)\ x}_\sigma = \mathtt{case}\ \sigma_1 \Rightarrow e_{t1}\ |\ \ldots\ |\ \sigma_n \Rightarrow e_{tn}\ \mathtt{in}\ e_t \rfloor_V = \\
\qquad\qquad \text{let } \forall \delta_1 \ldots \delta_m.\tau = \sigma \text{ and } \forall V_i.\tau_i = \sigma_i \text{ and } \tau_i \leq_{S_i} \sigma \text{ in} \\
\qquad\qquad \mathtt{let\ (rec)\ x = fun\ d}_1\ \ldots\ \mathtt{d}_m\ \rightarrow \\
\qquad\qquad\quad \mathtt{match\ d}_1, \ldots, \mathtt{d}_m\ \mathtt{with} \\
\qquad\qquad\quad |\ \lfloor S_1(\delta_1) \rfloor_{V_1}, \ldots, \lfloor S_1(\delta_m) \rfloor_{V_1} \rightarrow \lfloor e_{t1} \rfloor_{V+V_1} \\
\qquad\qquad\quad |\ \ldots \\
\qquad\qquad\quad |\ \lfloor S_n(\delta_1) \rfloor_{V_n}, \ldots, \lfloor S_n(\delta_m) \rfloor_{V_n} \rightarrow \lfloor e_{tn} \rfloor_{V+V_n} \\
\qquad\qquad \mathtt{in}\ \lfloor e_t \rfloor_V
\end{array}
$$

Figure 2: Pattern matching compilation algorithm $\lfloor \cdot \rfloor_V$

A successful pattern match binds run time types to the pattern variables of the matched case to provide the correct evaluation context of the selected type method. For the other stuffs, the compilation just removes all the type annotations from expressions.

The generic instances are compiled as function application of the compilations of their values on the run time types of the evaluation context. For instance an expression `print [1; 2]` is compiled to `print «int list» [1; 2]` because its quantified dynamic type variable is instantiated to `int list`.

We also need little work around, if type pattern is non linear, such as the second type case of `flat`. Because usually patterns must be linear in ML pattern matching, we must once give different pattern variables for each type variables, then compare their equality using pattern guards:

```
let rec flat d₁ d₂ =
  match d₁, d₂ with
  | d₃ list, d₄ → (fun l → flat d₃ d₄ (List.flatten l))
  | d₅, d₆ when d₅ = d₆ → (fun l → l)
```

6. Constraint typing system

Full and polymorphic instances

In the constraint typing, we categorize the value instances into two: full and polymorphic instances. *Polymorphic instances* are instances $\mathtt{x}_\eta : \tau$ some of whose free type variables of the annotations η and τ are generalized somewhere. Polymorphic instances are found in the derived generic definitions, like the following `plus` generic instance in the definition of `double`. Its type annotation contains the type variable δ which is generalized by the let binding for `double`:

```
let double{δ} = fun x → (plus : δ → δ → δ) x x;;
```

Full instances are the instances without such free type variables: their evaluation contexts are statically known and therefore we can check their validity. For example, the occurrences of `double` in the expressions `double 1` and `double 2.0` whose types are $\texttt{int} \rightarrow \texttt{int}$ and $\texttt{float} \rightarrow \texttt{float}$ are fully instantiated.

On the other hand, the correctness of polymorphic instances cannot be determined at the point of their occurrences, until their generalized type variables are fully instantiated in the evaluation. Therefore the type checking of a polymorphic instance is delegated to the checking of the full instance which use it: the type soundness of `plus` inside `double` will be checked at the `double`'s fully instantiations.

The following predicate *polyinst* distinguishes these instances formally. *polyinst* takes a set of *generalized type variables* V, a type scheme η and a type τ:

$$\mathit{polyinst}(V, \eta, \tau) = \mathit{BDV}(\eta) \neq \emptyset \ \& \ (\mathit{DV}(\tau) \cup \mathit{DV}(\eta)) \cap V \neq \emptyset$$

If $\mathit{polyinst}(V, \eta, \tau) = \mathit{true}$, a generic instance $\texttt{x}_\eta : \tau$ is a polymorphic instance under V. Otherwise it is full instance.

Extended type scheme and constraints

The *extended type schemes* are the type information of generic values in the constraint typing. They do not only have the type pattern informations of generic type cases, but also those of the polymorphic instances occur inside of their type case definitions:

$$\begin{array}{rcll}
\eta & ::= & \sigma\ \mathcal{K} & \text{Extended type schemes} \\
\mathcal{K} & ::= & [\ \sigma\ d \mid \ldots \mid \sigma\ d\ ] & \text{Type case constraints} \\
 & \mid & \kappa & \text{Constraint variables} \\
 & \mid & \mu\kappa.\mathcal{K} & \text{Recursive constraints} \\
d & ::= & \{\tau <: \eta \ \& \ \ldots \ \& \ \tau <: \eta\} & \text{Derived constraints}
\end{array}$$

An extended type scheme is a pair of a normal type scheme σ and a *generic constraint* $\mathcal{K}$. A type case constraint $[\ \sigma_1\ d_1 \mid \ldots \mid \sigma_n\ d_n\ ]$ consists of a list of type patterns σ_i and derived constraints d_i. The instantiation of an extended type scheme $\sigma\ [\ \sigma_1\ d_1 \mid \ldots \mid \sigma_n\ d_n\ ]$ is restricted to instances of σ_i which also satisfy their *derived constraint* d_i. A derived constraint $d = \{\tau_1 <: \eta_1 \ \& \ \ldots \tau_n <: \eta_n\}$ is a set of further instantiation conditions of generic values derived from the polymorphic generic instances used in their definitions. We also have the construction for the recursive constraint, *constraint variables* κ and μ *operator* which abstracts constraint variables. They are used to express the fixed points of the recursively defined constraints. We call extended type schemes just "type schemes" or "schemes" in short, if there is no confusion between them and the normal type schemes. We use $\mathit{KFV}(\mathcal{K})$ to denote the free constraint variables of constraint $\mathcal{K}$.

Since the definition of the constraints are somewhat over-relaxed, they must be restricted to the *well formed extended type schemes*, defined by the predicate *wellformed* below:

$$\begin{array}{rcl}
\mathit{wellformed}(\sigma\ [\sigma_1\ d_1 \mid \ldots \mid \sigma_n\ d_n]) & = & \forall i, \sigma_i \leq \sigma \ \& \ \mathit{wellformed}(d_i) \\
\mathit{wellformed}(\{\tau_1 <: \sigma_1\ \mathcal{K}_1 \ \& \ldots \& \ \tau_n <: \sigma_n\ \mathcal{K}_n\}) & = & \forall i, \tau_i \leq \sigma_i \ \& \ \mathit{wellformed}(\sigma_i\ \mathcal{K}_i) \\
\mathit{wellformed}(\kappa) & = & \mathit{true} \\
\mathit{wellformed}(\mu\kappa.\mathcal{K}) & = & \mathit{wellformed}(\mathcal{K}) \\
\mathit{wellformed}(\Gamma) & = & \forall (\texttt{x} \Rightarrow \eta) \in \Gamma, \mathit{wellformed}(\eta) = \mathit{true}
\end{array}$$

Corresponding with the normal typing rules of generic bindings, type patterns σ_i in constraints must be less general than the normal part of extended type schemes σ. In the derived conditions $\tau <: \sigma\ \mathcal{K}$,

τ must be an instance of σ. A typing environment Γ is *well formed* when each binding has a well formed type scheme. The previous examples of extended type schemes are all well formed.

The notion of the free type variables and the type substitutions are naturally extended for the extended schemes. The free type variables in constraints are also bound by the quantified variables of the normal scheme part:

$$
\begin{array}{rcl}
FV(\forall V.\tau\ \mathcal{K}) & = & (FV(\tau) \cup FV(\mathcal{K})) \setminus V \\
FV([\ \sigma_1\ d_1 \mid \ldots \mid \sigma_n\ d_n\ ]) & = & \bigcup_1^n FV(\sigma_i\ d_i) \\
FV(\forall V.\tau\ d) & = & (FV(\tau) \cup FV(d)) \setminus V \\
FV(\{\ \tau_1 <: \eta_1\ \&\ \ldots\ \&\ \tau_n <: \eta_n\ \}) & = & \bigcup_1^n FV(\tau_i)\ \cup\ \bigcup_1^n FV(\eta_i) \\
FV(\kappa) & = & \emptyset \\
FV(\mu\kappa.\mathcal{K}) & = & FV(\mathcal{K})
\end{array}
$$

$$
\begin{array}{rcll}
S(\sigma\ \mathcal{K}) & = & S(\sigma)\ S(\mathcal{K}) & \text{rename the bound variable if necessary} \\
S([\sigma_1\ d_1 \mid \ldots \mid \sigma_n\ d_n]) & = & [S(\sigma_1\ d_1) \mid \ldots \mid S(\sigma_n\ d_n)] & \\
S(\sigma\ d) & = & S(\sigma)\ S(d) & \\
S(\{\tau_1 <: \eta_1\ \& \ldots \&\ \tau_n <: \eta_n\}) & = & \{S(\tau_1) <: S(\eta_1)\ \& \ldots \&\ S(\tau_n) <: S(\eta_n)\} & \\
S(\kappa) & = & \kappa & \\
S(\mu\kappa.\mathcal{K}) & = & \mu\kappa.S(\mathcal{K}) &
\end{array}
$$

μ-expansion of recursive constraints

Recursive constraints are abstracted by μ-operators and represent the fixed points of the constraints of the recursive generic values. To check a type is a valid instance of an extended type scheme with recursive constraints, we need to expand their μ-abstractions:

Definition 3 *The* μ-expansion *of a recursive constraint $\mu\kappa.\mathcal{K}$ is a constraint $\{\kappa \Rightarrow \mu\kappa.\mathcal{K}\}(\mathcal{K})$. If $\mathcal{K}'$ is a μ-expansion of $\mu\kappa.\mathcal{K}$, we write $\mu\kappa.\mathcal{K} \overset{\mu}{\leadsto} \mathcal{K}'$.*

For instance, if we let $\mathcal{K}_{\texttt{print}}$ be the constraint of `print`,

$$
\begin{array}{ll}
\mathcal{K}_{\texttt{print}} \overset{\mu}{\leadsto} & [\ \texttt{int} \to \texttt{unit} \mid \texttt{string} \to \texttt{unit} \\
& \mid \forall\delta'.\delta'\ \texttt{list} \to \texttt{unit}\ \{\delta' \to \texttt{unit} <: \forall\delta.\delta \to \texttt{unit}\ \mathcal{K}_{\texttt{print}}\ \& \\
& \qquad\qquad\qquad\qquad \delta'\ \texttt{list} \to \texttt{unit} <: \forall\delta.\delta \to \texttt{unit}\ \mathcal{K}_{\texttt{print}}\}]
\end{array}
$$

Once the derived and recursive constraints are introduced, generic constraints are often difficult to understand at a glance. Even the `print` with only three type cases has such a complicated constraints. We can easily imagine that more complicated generic values will have constraints almost unreadable. However, in addition to generic constraints, we have the normal type scheme part provided by the normal typing layer. It is the least general anti-unifiers of the patterns, therefore we can grasp a good approximation of its real type. For example, even we ignore the constraint of `print`, its normal scheme $\forall\delta.\delta \to \texttt{unit}$ tells us that `print` is a generic function returns `unit`.

Constraint annotated terms

The type annotations of GML expressions must also be extended so that they can carry constraints:

$$\begin{array}{lcl} e_t & ::= & \mathtt{x}_\eta : \tau \mid \ldots \\ & \mid & \mathtt{let}\ (\mathtt{rec})\ \mathtt{x}_{V,\mathcal{K}} = e_t\ \mathtt{in}\ e_t : \tau \\ & \mid & \mathtt{generic}\ (\mathtt{rec})\ \mathtt{x}_\eta = \mathtt{case}\ tc_{td} \mid \ldots \mid tc_{td}\ \mathtt{in}\ e_t \\ tc_{td} & \equiv & \sigma\ d \Rightarrow e_t \end{array}$$

The variables now take extended type schemes η instead of normal type schemes. Let bindings take their constraints $\mathcal{K}$ with the generalized type variables V. Also in the generic bindings, the defined identifiers and the type patterns carry their constraints.

Valid full instances

The full instantiations of extended type schemes are restricted by the constraints attached to them. We first define the *invalid full instances* which raise type match failures. If τ is an invalid full instance of a type scheme η, we write $\tau \not<: \eta$. Its definition is given by the following rules:

$$\text{(k-fail-case)}\ \frac{\tau \not\leq \sigma, \quad \text{or} \quad \forall 1 \leq i \leq n, \tau \not<: \sigma_i\ d_i}{\tau \not<: \sigma\ [\ \sigma_1\ d_1 \mid \ldots \mid \sigma_n\ d_n\ ]}$$

$$\text{(k-fail-rec)}\ \frac{\tau \not<: \sigma\ \{\kappa \Rightarrow \mu\kappa.\mathcal{K}\}\mathcal{K}}{\tau \not<: \sigma\ \mu\kappa.\mathcal{K}}$$

$$\text{(k-fail-derive)}\ \frac{\tau \not\leq \sigma, \quad \text{or} \quad \exists 1 \leq i \leq n, \quad S(\tau_i) \not<: S(\eta_i)\ \text{where}\ \tau \leq_S \sigma}{\tau \not<: \sigma\{\tau_1 <: \eta_1\ \&\ \ldots\ \&\ \tau_n <: \eta_n\}}$$

$$\text{(k-fail-free)}\ \tau <: \sigma\ \kappa$$

For example, if we let $\eta_{\mathtt{plus}}$ be the type scheme of `plus`, $\mathtt{string} \to \mathtt{string} \to \mathtt{string}$ is proved as an invalid instance of the type scheme of `double`, $\forall\delta.\delta \to \delta\ \{\delta \to \delta \to \delta <: \eta_{\mathtt{plus}}\}$ as follows:

$$\cfrac{\mathtt{string} \to \mathtt{string} \leq_{\{\delta \Rightarrow \mathtt{string}\}} \forall\delta \to \delta \quad \cfrac{\begin{array}{l}\mathtt{string} \to \mathtt{string} \to \mathtt{string} \not<: \mathtt{int} \to \mathtt{int} \to \mathtt{int} \\ \mathtt{string} \to \mathtt{string} \to \mathtt{string} \not<: \mathtt{float} \to \mathtt{float} \to \mathtt{float} \\ \mathtt{string} \to \mathtt{string} \to \mathtt{string} \not<: \forall\delta.\delta\ \mathtt{list} \to \delta\ \mathtt{list} \to \delta\ \mathtt{list}\end{array}}{\mathtt{string} \to \mathtt{string} \to \mathtt{string} \not<: \eta_{\mathtt{plus}}}}{\mathtt{string} \to \mathtt{string} \not<: \forall\delta.\delta \to \delta\ \{\delta \to \delta \to \delta <: \eta_{\mathtt{plus}}\}}$$

The *valid full instances* are defined as the complement of the invalid full instances:

$$\tau <: \sigma\ \mathcal{K} \iff \tau \not<: \sigma\ \mathcal{K}\ \text{is not provable.}$$

This definition as "the complement of the failures" will make sure the well typed full instances are never evaluated to the type match failure. In some cases, type recursions will not terminate, but this is not a problem, since well typed normal recursions also may loop forever.

The following algorithm $kinst(M, \tau, \eta)$ checks that τ is a valid full instances of η or not. It takes a memo function M which remembers the μ-abstraction tests already done. If $kinst$ is asked the same question in the cache M, it immediately returns *true*:

$$\begin{array}{l} \mathit{kinst}(M, \tau <: \sigma\ [\ \sigma_1\ d_1 \mid \ldots \mid \sigma_n\ d_n\ ]) = \\ \qquad\qquad \tau \leq \sigma\ \&\ (\mathit{kinst}(M, \tau <: \sigma_1\ d_1)\ \|\ \ldots\ \|\ \mathit{kinst}(M, \tau <: \sigma_n\ d_n)) \\ \mathit{kinst}(M, \tau <: \sigma\ \{\tau_1 <: \eta_1\ \&\ \ldots\ \&\ \tau_n <: \eta_n\}) = \\ \qquad\qquad \tau \leq_S \sigma\ \&\ \mathit{kinst}(M, S(\tau_1) <: S(\eta_1))\ \&\ \ldots\ \&\ \mathit{kinst}(M, S(\tau_n) <: S(\eta_n)) \\ \mathit{kinst}(M, \tau <: \sigma\ \mu\kappa.\mathcal{K}) = \ \text{if } \tau <: \sigma\ \mu\kappa.\mathcal{K} \text{ is not cached in } M \text{ then} \\ \qquad\qquad\qquad\qquad \mathit{kinst}(M \cup \{\tau <: \sigma\ \mu\kappa.\mathcal{K}\}, \tau <: \sigma\ \{\kappa \Rightarrow \mu\kappa.\mathcal{K}\}(\mathcal{K})) \\ \qquad\qquad\qquad\quad \text{if cached in } M,\ \mathit{true} \\ \mathit{kinst}(M, \tau <: \sigma\ \kappa) = \mathit{false} \end{array}$$

Theorem 2 (Correctness of *kinst* against $<:$) *Let τ be a type and $\sigma\ \mathcal{K}$ be a type scheme. Suppose $\mathit{kinst}(\emptyset, \tau <: \sigma\ \mathcal{K})$ terminates. Then $\tau \not<: \sigma\ \mathcal{K}$ if and only if $\mathit{kinst}(\emptyset, \tau <: \sigma\ \mathcal{K})$ returns false.*

Proof: Suppose $\tau \not<: \sigma\ \mathcal{K}$. then we can check easily $\mathit{kinst}(\emptyset, \tau \not<: \sigma\ \mathcal{K}) = \mathit{false}$ by induction over the derivation of $\tau \not<: \sigma\ \mathcal{K}$. Contrary, suppose $\mathit{kinst}(\emptyset, \tau <: \sigma\ \mathcal{K}) = \mathit{false}$. We can build the proof of $\tau \not<: \sigma\ \mathcal{K}$ by the induction over the evaluation of the algorithm *kinst*. □

Unfortunately, because of the recursive constraints, the relationship $\not<:$ is undecidable (the outline of this proof is given in the appendix of [4]), therefore so is $<:$. To make them decidable and assure the termination of the algorithm *kinst*, we need to restrict the set of extended type schemes. We define *sizes of types* and restrict the constraints so that the size of types involve with *kinst* can not increase in the recursive call sequence of *kinst*.

Type sizes

The type size of τ, $|\tau|$ is defined as a simple formal arithmetic expressions as follows:

$$|\mathit{base}| = 1 \quad |\alpha| = 1 \quad |\delta_i| = x_i \quad |\tau_1 \to \tau_2| = \textstyle\sum_{i=1}^{2} |\tau_i| + 1 \quad |\tau_1 * \tau_2| = \textstyle\sum_{i=1}^{2} |\tau_i| + 1$$

For each dynamic type variable δ_i, $|\delta_i|$ returns a variable x_i, specific to each dynamic type variable. The *type size ordering* $<$ is defined as their ordering as equations by ranging all the variables x_i appears in the equations more than 0:

$$|\tau| < |\tau'| \iff \text{for all } x_i > 0,\ |\tau| < |\tau'| \text{ as integers}$$

We also define the size of a predicate $\tau <: \eta$. It is defined as follows:

$$|\tau <: \sigma\ \mathcal{K}| = \sum_{\delta \in BDV(\sigma)} |S(\delta)| \quad \text{where } \tau \leq_S \sigma$$

This is the sum of the type sizes $S(\delta)$, the types which indicate how each bound dynamic type variable of σ is instantiated in τ. $|\tau <: \eta|$ is well defined only if τ must be an instance of the normal type scheme part of η. Otherwise it is undefined. For example, $|\delta_1 \to \mathtt{int} <: \forall \delta_2\delta_3.\delta_2 \to \delta_3\ \mathcal{K}| = |\delta_1| + |\mathtt{int}| + 1 = x_1 + 2$.

Admissible type schemes

Using the type sizes we define the set of *the admissible type schemes.* The predicate *adm* which defines the admissible schemes. Roughly speaking, if a generic value has an admissible scheme, the sizes of run time types which its compilation receives must be always greater than those of the run time types delivered to the recursive polymorphic instances in its definition:

$$\begin{array}{rcl}
adm(R, \sigma\ [\sigma_1\ d_1|\ldots|\sigma_n\ d_n]) & = & adm_check(R, \sigma_1\ d_1)\ \&\ \ldots\ \&\ adm_check(R, \sigma_n\ d_n) \\
adm(R, \sigma\ d) & = & adm_check(R, \sigma, \sigma\ d) \\
adm(R, \sigma\ \mu\kappa.\mathcal{K}) & = & adm(R \cup \{\kappa\}, \sigma\ \mathcal{K}) \\
adm(R, \sigma\ \kappa) & = & \text{if } \kappa \in R \text{ then } true \text{ else } false
\end{array}$$

$$\begin{array}{l}
adm_check(R, \sigma, \forall V.\tau\ \{\tau_1 <: \sigma_1\ \mathcal{K}_1\ \&\ \ldots\ \&\ \tau_n <: \sigma_n\ \mathcal{K}_n\})\ = \\
\qquad \forall 1 \le i \le n, \quad adm(R, \sigma_i\ \mathcal{K}_i) = true \\
\qquad\qquad \&\ \text{if } R \cap KFV(\sigma_i\ \mathcal{K}_i) \neq \emptyset \text{ then } |\tau_i <: \sigma_i| < |\tau <: \sigma|
\end{array}$$

A type scheme η is *admissible* if and only if $adm(\emptyset, \eta) = true$. We can easily verify that all the previous examples of the type schemes are admissible.

To prove the termination of the *kinst*, we first require the following lemmas:

Lemma 1 *Let $\eta_1 = \sigma_1\ \mathcal{K}_1$ and $\eta_2 = \sigma_2\ \mathcal{K}_2$ be extended type schemes and τ be a type expression such that $\tau \le \sigma_1$ and $\tau \le \sigma_2$. If σ_1 is more general than σ_2, i.e. $inst(\sigma_2) \le \sigma_1$, $|\tau <: \eta_2| \le |\tau <: \eta_1|$.*

Lemma 2 (Dynamic substitution keeps type size ordering) *Let S be a substitution such that $Dom(S) \subset \mathbb{D}$. If $|\tau| < |\tau'|$ then $|S(\tau)| < |S(\tau')|$.*

Lemma 3 (μ-expansion keeps admissibity) *Let $\sigma\ \mathcal{K}, \sigma\ \mathcal{K}'$ be type schemes, R be a set of constraint variables and ψ be a constraint substitution such that $\psi = \{\kappa \Rightarrow \mathcal{K}'\}$.*

If $adm(R, \sigma\ \mathcal{K}) = true$ and $adm(R, \sigma\ \mathcal{K}') = true$, then $adm(R, \psi(\sigma\ \mathcal{K})) = true$.

Theorem 3 (Admissibity assures the termination of $kinst(\tau <: \eta)$) *Let η be a well formed admissible extended type scheme. Then for any type expression τ, $kinst(\emptyset, \tau <: \eta)$ always terminates and returns true or false.*

PROOF: (Sketch) First we show the sizes of the argument predicate $\tau <: \eta$ of *kinst* do not increase in the sequence of its recursive calls. For each constraint constructor, we use one of the above lemmas.

Type patterns Let $\eta = \forall\nu_0^*.\tau_0\ [\ldots|\eta_1|\ldots]$ where $KFV(\eta_1) \cap R \neq \emptyset$. Then $kinst(\tau <: \eta) = \tau \le \forall\nu_0^*.\tau_0\ \&\ \ldots\ \&\ kinst(\tau <: \eta_1)\ \&\ \ldots$. Because the normal scheme of η is more general than the one of η_1, from Lemma 1, $|\tau <: \eta_1| \le |\tau <: \eta|$.

Derived constraints Let $\eta = \forall\nu_0^*.\tau_0\ \{\ldots\quad \tau_1 <: \eta_1 \quad \ldots\}$ where $KFV(\eta_1) \cap R \neq \emptyset$. Then $kinst(\tau <: \eta) = \tau \le_S \forall\nu_0^*.\tau_0\ \&\ \ldots\ \&\ kinst(S(\tau_1) <: S(\eta_1))\ \&\ \ldots$.

Since $\tau \le_S \forall\nu_0^*.\tau_0$, $S(\tau_0) = \tau$. From $adm(R, \eta_1)$, $|\tau_1 <: \eta_1| \le |\tau_0 <: \eta|$. From Lemma 2, $|S(\tau_1) <: S(\eta_1))| \le |S(\tau_0) <: S(\eta)| = |\tau <: \eta|$

Recursive constraints Let $\eta = \sigma\ \mu\kappa_1.\ldots.\mu\kappa_n.\mathcal{K}$, where $\mathcal{K}$ is a type pattern or derived constraint. By n times of μ-expansions, for some constraint substitution ψ, $kinst(M, \tau <: \eta) = kinst(M, \tau <: \sigma\ \psi(\mathcal{K}))$. Since $adm(R, \eta) = true$, by appling Lemma 3 n times $adm(R\{\kappa_i\}, \sigma\ \psi(\mathcal{K})) = true$ for $1 \le i \le n$ and we obtain $|\tau <: \sigma\ \psi(\mathcal{K})| \le |\tau <: \eta|$

Constraint variables Because *adm* checks the closedness of the constraint variables, therefore we cannot have these cases.

To complete the proof we think the case when the type size never reduces. Suppose $kinst(M, \tau <: \eta)$ recursively calls $kinst(M', \tau' <: \eta')$. Then from the definition of *kinst*, $FV(\tau) \cup FV(\eta) \supset FV(\tau') \cup FV(\eta')$. Therefore the type constructors which involve with the sequence of recursive *kinst* calls are finite, and the argument predicates $\tau <: \eta$ applied to *kinst* which share the same size are also finite. Therefore even when the type sizes do not decrease strictly, sooner or later $kinst(M, \tau <: \eta)$ will find the same predicate stored in the cache M and terminate. □

Constraint typing rules

The constraint typing judgments have the form $\Gamma, V \Vdash e_t$ where Γ and e_t are both extended with constraints. Because the type instantiations are differently handled if it is polymorphic instances or not, a judgment carries also V, *generalized type variable set*. The typing rules are shown in Figure 3.

We have two instantiation rules one for the full and another for the polymorphic instances. In (kt-full-inst), the identifier `x` must be fully instantiated and the type must be a valid full instance of `x`'s type scheme. If a variable is polymorphic instance, (kt-poly-inst) is used and this time the type need not to be a valid full instance of the type scheme.

In the let and generic bindings we create constraints of the defined values. The constraint construction function $Konst(\Gamma, V, e_t)$ gathers the polymorphic instances in e_t whose type annotations relate with V. If such instances are $[\mathtt{x}_i : \tau_i]$ [3], it returns a derived constraint $\{\tau_i <: \Gamma(\mathtt{x}_i)\}$. This derived constraint is bound to the identifier in the typing environment accompanied with the normal part of type schemes. In the case of the generic bindings, the type patterns and their derived constraints are gathered and stored as a type case constraint.

In the recursive bindings, the defined identifier is stored with the recursive constraints in the typing of the sub-expressions. These recursive constraints must be the fixed point: the derived constraint computed by *Konst* must be the μ-expansion of it.

For example, the generic binding of `print` is typed as follows: let the type scheme of `print` be $\eta_{\mathtt{print}} = \forall\delta.\delta \rightarrow \mathtt{unit}\ \mu\kappa.\mathcal{K}$, where $\mathcal{K}$ is still unsure for us. The typing of the first two type cases are trivial. Since the type pattern of the third case has the pattern $\forall\delta'.\delta'\ \mathtt{list} \rightarrow \mathtt{unit}$, it method is checked with the generalized variable set $\{\delta'\}$:

$$\dfrac{\dfrac{\Gamma\{\mathtt{print} : \eta_{\mathtt{print}}\}, \{\delta'\} \Vdash \mathtt{print} : \delta' \rightarrow \mathtt{unit} \quad \Gamma\{\mathtt{print} : \eta_{\mathtt{print}}\}, \{\delta'\} \Vdash \mathtt{print} : \delta'\ \mathtt{list} \rightarrow \mathtt{unit}}{\cdots}}{\begin{array}{l}\Gamma\{\mathtt{print} : \eta_{\mathtt{print}}\}, \{\delta'\} \Vdash \ \mathtt{(function\ [\,]} \rightarrow \mathtt{()} \\ \qquad\qquad\qquad\qquad\qquad \mathtt{|\ x::xs} \rightarrow \mathtt{print\ x;\ print\ xs)}\ : \delta'\ \mathtt{list} \rightarrow \mathtt{unit}\end{array}}$$

Since the two internal `print` generic instances are typed with this variable δ', they are treated as polymorphic instances. In the use of (kt-genericrec), their type informations are exported to the derived constraint of `print` and their validness checks are postponed to full instantiations of `print`:

$$\begin{array}{l}Konst(\Gamma\{\mathtt{print} : \eta_{\mathtt{print}}\}, \{\delta'\}, \ \mathtt{function\ [\,]} \rightarrow \mathtt{\ldots;\ \ print\ \ xs}\ ) = \\ \qquad\qquad\qquad\qquad \{\delta' \ \rightarrow \mathtt{unit} <: \eta_{\mathtt{print}}\ \&\ \delta'\ \mathtt{list} \rightarrow \mathtt{unit} <: \eta_{\mathtt{print}}\}\end{array}$$

Gathering the three derived constraints of the type cases (actually the first two derived constraints are empty), we construct a type case constraint $\mathcal{K}'$:

$$\begin{array}{rl}\mathcal{K}' = & [\mathtt{int} \rightarrow \mathtt{unit} \mid \mathtt{string} \rightarrow \mathtt{unit} \\ & \mid \forall\delta.\delta\ \mathtt{list} \rightarrow \mathtt{unit}\{\ \delta \rightarrow \mathtt{unit} <: \eta_{\mathtt{print}}\ \&\ \delta\ \mathtt{list} \rightarrow \mathtt{unit} <: \eta_{\mathtt{print}}\}]\end{array}$$

[3]These polymorphic instances must be sorted in the order of the occurrences in e_t. This is required for the new compilation using generic flows defined later.

$$
\begin{array}{l}
\mathit{Konst}(\Gamma, V, e_t) = \\
\qquad \text{let } [\mathtt{x}_1 : \tau_1;\ \ldots\ ;\ \mathtt{x}_n : \tau_n] \text{ be the variables in } e_t \\
\qquad\qquad \text{such that} \quad \mathit{polyinst}(V, \Gamma(\mathtt{x}_i), \tau_i) = \mathit{true} \\
\qquad \text{in } \{\tau_1 <: \Gamma(\mathtt{x}_1) \ \& \ldots \& \ \tau_n <: \Gamma(\mathtt{x}_n)\}
\end{array}
$$

$$
\text{(kt-const)}\ \frac{DV(\tau) \subset V}{\Gamma, V \Vdash c_\sigma : \tau} \qquad \text{(kt-gen)}\ \frac{DV(\tau) \subset V}{\Gamma, V \Vdash \mathtt{g}_\sigma : \tau} \qquad \text{(kt-tpl)}\ \frac{\Gamma, V \Vdash e_{t1} \quad \Gamma, V \Vdash e_{t2}}{\Gamma, V \Vdash (e_{t1},\ e_{t2})}
$$

$$
\text{(kt-fun)}\ \frac{DV(\tau_1) \subset V \quad \Gamma\{\mathtt{x} : \tau_1\}, V \Vdash e_t}{\Gamma, V \Vdash \mathtt{fun\ x} \rightarrow e_t : \tau_1 \rightarrow \tau_2} \qquad \text{(kt-app)}\ \frac{\Gamma, V \Vdash e_{t1} \quad \Gamma, V \Vdash e_{t2}}{\Gamma, V \Vdash e_{t1}\ e_{t2}}
$$

$$
\text{(kt-full-inst)}\ \frac{\Gamma(\mathtt{x}) = \eta \quad DV(\tau) \subset V \quad \mathit{polyinst}(V, \eta, \tau) = \mathit{false} \quad \tau <: \eta}{\Gamma, V \Vdash \mathtt{x}_\eta : \tau}
$$

$$
\text{(kt-poly-inst)}\ \frac{\Gamma(\mathtt{x}) = \eta \quad DV(\tau) \subset V \quad \mathit{polyinst}(V, \eta, \tau) = \mathit{true}}{\Gamma, V \Vdash \mathtt{x}_\eta : \tau}
$$

$$
\text{(kt-let)}\ \frac{\begin{array}{c}\Gamma, V + V' \Vdash e_{t1} : \tau_1 \quad \Gamma\{\mathtt{x} : \eta\}, V \Vdash e_{t2} \\ \eta = \forall V'.\tau_1\ [\forall V'.\tau_1\ d] \quad d = \mathit{Konst}(\Gamma, V', e_{t1}) \quad \mathit{adm}(\emptyset, \eta) = \mathit{true}\end{array}}{\Gamma, V \Vdash \mathtt{let\ x}_{V',d} \mathtt{=} e_{t1}\ \mathtt{in}\ e_{t2}}
$$

$$
\text{(kt-letrec)}\ \frac{\begin{array}{c}\Gamma\{\mathtt{x} : \eta\}, V + V' \Vdash e_{t1} : \tau_1 \quad \Gamma\{\mathtt{x} : \eta\}, V \Vdash e_{t2} \\ \eta = \forall V'.\tau_1\ \mu\kappa.[\forall V'.\tau_1\ d] \quad d' = \mathit{Konst}(\Gamma\{\mathtt{x} : \eta\}, V', e_{t1}) \quad \mathit{adm}(\emptyset, \eta) = \mathit{true} \\ \mu\kappa.[\forall V'.\tau_1\ d] \overset{\mu}{\leadsto} [\forall V'.\tau_1\ d']\end{array}}{\Gamma, V \Vdash \mathtt{let\ rec\ x}_{V',\mu\kappa.d} \mathtt{=} e_{t1}\ \mathtt{in}\ e_{t2}}
$$

$$
\text{(kt-generic)}\ \frac{\begin{array}{c}\Gamma, V + V_i \Vdash e_{ti} : \tau_i \quad d_i = \mathit{Konst}(\Gamma, V_i, e_{ti}) \\ V_i = FV(\tau_i) \quad \sigma_i = \forall V_i.\tau_i \\ \mathcal{K} = [\sigma_1\ d_1 \mid \ldots \mid \sigma_n\ d_n] \quad \mathit{adm}(\emptyset, \sigma\ \mathcal{K}) = \mathit{true} \\ FV(\sigma) = \emptyset \quad \Gamma\{\mathtt{x} : \sigma\ \mathcal{K}\}, V \Vdash e_t\end{array}}{\Gamma, V \Vdash \mathtt{generic\ x}_{\sigma\mathcal{K}} \mathtt{=case}\ \sigma_1\ d_1 \Rightarrow e_{t1} \mid \ \ldots\ \mid \sigma_n\ d_n \Rightarrow e_{tn}\ \mathtt{in}\ e_t}
$$

$$
\text{(kt-genericrec)}\ \frac{\begin{array}{c}\Gamma\{\mathtt{x} : \sigma\ \mu\kappa.\mathcal{K}\}, V + V_i \Vdash e_{ti} : \tau_i \quad d_i = \mathit{Konst}(\Gamma\{\mathtt{x} : \sigma\ \mu\kappa.\mathcal{K}\}, V_i, e_{ti}) \\ V_i = FV(\tau_i) \quad \sigma_i = \forall V_i.\tau_i \\ \mathcal{K}' = [\sigma_1\ d_1 \mid \ldots \mid \sigma_n\ d_n] \quad \mu\kappa.\mathcal{K} \overset{\mu}{\leadsto} \mathcal{K}' \quad \mathit{adm}(\emptyset, \sigma\ \mu\kappa.\mathcal{K}) = \mathit{true} \\ FV(\sigma) = \emptyset \quad \Gamma\{\mathtt{x} : \sigma\ \mu\kappa.\mathcal{K}\}, V \Vdash e_t\end{array}}{\Gamma, V \Vdash \mathtt{generic\ rec\ x}_{\sigma\mu\kappa.\mathcal{K}} \mathtt{=case}\ \sigma_1\ d_1 \Rightarrow e_{t1} \mid \ \ldots\ \mid \sigma_n\ d_n \Rightarrow e_{tn}\ \mathtt{in}\ e_t}
$$

Figure 3: Constraint typing rules

The constraint of `print` $\mu\kappa.\mathcal{K}$ must be μ-expanded to this $\mathcal{K}'$. Therefore:

$$\begin{aligned}\mu\kappa.\mathcal{K} = \mu\kappa.\ & [\texttt{int} \rightarrow \texttt{unit} \mid \texttt{string} \rightarrow \texttt{unit} \\ & \mid \forall\delta.\delta\ \texttt{list} \rightarrow \texttt{unit}\{\ \delta \rightarrow \texttt{unit} <: \forall\delta.\delta \rightarrow \texttt{unit}\ \kappa \\ & \qquad\qquad\qquad\qquad \&\ \delta\ \texttt{list} \rightarrow \texttt{unit} <: \forall\delta.\delta \rightarrow \texttt{unit}\ \kappa\}]\end{aligned}$$

Theorem 4 (Complete type soundness) *The combination of the normal typing and the constraint typing provides the complete type soundness: If $\emptyset \vdash e_t$ and $\emptyset \Vdash e_t$, then the evaluation e_t never returns an error.*

Proof: (Rough sketch) What to be proved is that the soundness of the generic instances. The case of the full instances are proved easily because their typing is by (kt-full-inst) and their types must be valid full instances. In the evaluation steps, the polymorphic instances will become valid full instances receiving their evaluation contexts, since their type case informations are stored as derived constraints of other generic values and type checked by the constraint typing. Once they become fully instantiated, their evaluation never returns the run time type match failures. □

7. Compilation using the flow pre-computation

By the constraint typing system, we can verify the type soundness of the generic instances statically. Therefore the run time type checking by pattern matching of the generic value evaluations is just redundant. Under the complete type soundness, the only meaning of the type matching is for the flow control of type cases. Since the static checking of the full instances by the algorithm *kinst* is very pre-computation of this flow control, for the well typed expressions we know which type case will be selected in each type matching at the type checking. Using this *generic flow* information instead of run time type matching, we can improve the efficiency of the flow control of type cases.

The generic flow is represented as a tree which may contain loops whose nodes have integer indices. It can be given by the following type definition:

```
type flow = Node of int * flow array
```

Each node of the generic flows corresponds with each generic type case selection and indicates which type case is selected. Each sub-tree of the node is the flow for the internal evaluation of the selected type method. Because of the admissibity of the type schemes, even the flows for the type recursions which never terminate can be expressed finitely, as trees with loops.

The modified $kinst(M, \tau <: \eta)$ algorithm in Figure 4 checks whether the full instance $\texttt{x}_\eta : \tau$ is valid or not, and if it is, returns the generic flow for `x` at the same time. The memo function M is now extended to have bindings from *kinst* predicates to flows. In the check against a type scheme with recursive constraint, if the questioned predicate is not in the cache M, *kinst* binds the predicate with the current tree node and continues the calculation. This binding serves as the entry point for the looped flows. If the cache hits, *kinst* returns the cached flow and a loop is created. To calculate flows with loops we use the recursive definition of non functional values.

Compilation with generic flows

The new compilation algorithm using the generic flows $\lfloor e_t \rfloor_F$ in Figure 5 takes an expression and a binding F from variable occurrences to generic flows. If a generic instance `x` is stored in F, `x` is compiled as an application of itself on the flow $F(\texttt{x})$. Otherwise, since `x` must be a full instance, the flow f of `x` is computed by *kinst* and applied.

$$
\begin{array}{l}
\mathit{kinst}(M, \tau <: \sigma\ [\ \sigma_1\ d_1 \mid \ldots \mid \sigma_n\ d_n\ ]) = \\
\qquad \text{if } \tau \leq \sigma,\ \forall 1 \leq i < p,\ \mathit{kinst}(M, \tau <: \sigma_i\ d_i) = \mathit{false} \\
\qquad \text{and } \mathit{kinst}(M, \tau <: \sigma_p\ d_p) = \mathit{true} \text{ then} \\
\qquad\quad \mathtt{Node}(p,\ \mathit{kinst}(M, \tau <: \sigma_p\ d_p)) \\
\qquad \text{else fails} \\
\mathit{kinst}(M, \tau <: \sigma\ \{\tau_1 <: \eta_1\ \&\ \ldots\ \&\ \tau_n <: \eta_n\}) = \\
\qquad \text{if } \tau \leq_S \sigma \text{ then} \\
\qquad\quad \mathtt{[|}\ \mathit{kinst}(M, S(\tau_1) <: S(\eta_1)); \ldots;\ \mathit{kinst}(M, S(\tau_n) <: S(\eta_n))\ \mathtt{|]} \\
\qquad \text{else fails} \\
\mathit{kinst}(M, \tau <: \sigma\ \mu\kappa.\mathcal{K}) = \\
\qquad \text{if } M(\tau <: \sigma\ \mu\kappa.\mathcal{K}) \text{ is defined, return it} \\
\qquad \text{else} \\
\qquad\quad \mathtt{let\ rec\ flow} = \mathit{kinst}(M \cup \{(\tau <: \sigma\ \mu\kappa.\mathcal{K}) \Rightarrow \mathtt{flow}\}, \tau <: \sigma\ \{\kappa \Rightarrow \mu\kappa.\mathcal{K}\}(\mathcal{K})) \\
\qquad\quad \mathtt{in\ flow} \\
\mathit{kinst}(M, \tau <: \sigma\ \kappa) = \text{fails}
\end{array}
$$

Figure 4: *kinst* algorithm with generic flow calculation

$$
\begin{array}{ll}
\lfloor c_\sigma : \tau \rfloor_F = c_\sigma & \lfloor \mathtt{fun\ x} \rightarrow e_t \rfloor_F = \mathtt{fun\ x} \rightarrow \lfloor e_t \rfloor_F \\
\lfloor e_{t1}\ e_{t2} \rfloor_F = \lfloor e_{t1} \rfloor_F\ \lfloor e_{t2} \rfloor_F & \lfloor e_{t1}, e_{t2} \rfloor_F = \lfloor e_{t1} \rfloor_F,\ \lfloor e_{t2} \rfloor_F
\end{array}
$$

$$
\begin{array}{l}
\lfloor \mathtt{x}_\eta : \tau \rfloor_F = \\
\quad \text{if } \mathtt{x} \text{ is not generic then } \mathtt{x} \\
\quad \text{else if the occurrence of } \mathtt{x} \text{ is bound in } F \text{ then } \mathtt{x}\ F(\mathtt{x}) \\
\qquad \text{else let } f = \mathit{kinst}(\emptyset, \tau <: \eta) \text{ in } \mathtt{x}\ f
\end{array}
$$

$$
\begin{array}{l}
\lfloor \mathtt{let\ (rec)\ x}_{V,\mathcal{K}} = e_{t1}\ \mathtt{in}\ e_{t2} \rfloor_F = \\
\quad \text{let } [\mathtt{x}^i_{\eta_i} : \tau_i] \text{ be the variable occurrences in } e_{t1} \text{ such that } \mathit{polyinst}(V, \eta_i, \tau_i) = \mathit{true} \\
\quad \text{in } \mathtt{let\ (rec)\ x = fun\ (Node(p, flow))} \rightarrow \lfloor e_{t1} \rfloor_{F\{\mathtt{x}^i \Rightarrow \mathtt{flow}.(i)\}}\ \mathtt{in}\ \lfloor e_{t2} \rfloor_F
\end{array}
$$

$$
\begin{array}{l}
\lfloor \mathtt{generic\ (rec)\ x}_\sigma = \mathtt{case}\ tc_{t1} \mid \ldots \mid tc_{tn}\ \mathtt{in}\ e_t \rfloor_F = \\
\quad \text{let } [\mathtt{x}^{ij}_{\eta_{ij}} : \tau_{ij}] \text{ be the variable occurrences in } e_{ti} \text{ such that } \mathit{polyinst}(V, \eta_{ij}, \tau_{ij}) = \mathit{true} \\
\quad \text{let } F_i = F\{\mathtt{x}^{ij} \Rightarrow \mathtt{flow}.(j)\} \\
\quad \text{in}\ \ \mathtt{let\ (rec)\ x = fun\ (Node(p, flow))} \rightarrow \\
\qquad\quad \mathtt{let\ tm = [|}\ \lfloor tc_{t1} \rfloor_{F_1};\ \ldots;\ \lfloor tc_{tn} \rfloor_{F_n}\ \mathtt{|]\ in} \\
\qquad\quad \mathtt{tm}.(\mathtt{p}) \\
\qquad \mathtt{in}\ \lfloor e_t \rfloor_F
\end{array}
$$

Figure 5: The compilation algorithm with generic flows

The generic values are compiled as functions which receives a flow `Node(p,flow)`. The top index `p` is ignored in let bound values because they have no type cases. Each sub-flow in `flow` is delivered to the corresponding polymorphic instances by compiling the definition expression e_{t1} with the updated binding. The set of generic type methods is compiled as an array and the type case selection is done by accessing and executing `p`-th element of the array. Here also the sub-flows are delivered to the polymorphic instances inside the selected type method.

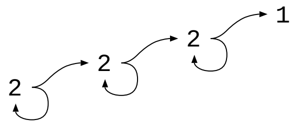
For instance, the generic recursive function `print` is compiled as follows:

```
let rec print = fun (Node (p,flow) →
  let tm = [| print_int; print_string;
              (function [] → ()
                      | x::xs → print flow.(0) x; print flow.(1) xs) |]
  in
  tm.(p);;
```

If `print` is fully instantiated as `int list list list` → `unit` in an expression like:

```
# print [[[1.0]; [1.1;1.2]]; [[1.3;1.4]]];;
```

Then the flow of `print`, $f = \mathit{kinst}(\emptyset,$ `float list list list` $\rightarrow$ `unit` $<: \eta_{\texttt{print}})$ is the looped tree as follows:



The whole expression is compiled to `print` f `[[[1.0]; [1.1;1.2]]; [[1.3;1.4]]];;`

Benchmark

Before we end this paper, we should do a little benchmark. We wrote the following Fibonacci function using the generic binding and compared how this new compilation method affects the evaluation efficiency.

```
generic rec fib = case
| int → int ⇒ (function 1 | 2 → 1 | n → fib (n-2) + fib (n-1))
in
fib 36;;
```

The both compilation translate this program as follows:

```
let rec fib1 = function
    «int» → (function 1 | 2 → 1
                      | n → fib1 t (n - 2) + fib1 t (n - 1))
in fib1 «int» 36;;

let rec fib2 = fun (Node (p,t')) →
  let tm = [| function 1 | 2 → 1
                      | n → fib2 t'.(0) (n - 2) + fib2 t'.(0) (n - 1)) |]
  in
  tm.(p)
```

```
in
let rec trace = Node (0,[|trace|])
in fib2 trace 36;;
```

Of course this generic binding is completely meaningless in practice, but it is good for the benchmark because the classical compilation of this function contains the simplest run time type pattern «int».

To extract the pure time usage by the flow controls in the both compilation, we also timed the normal non-generic version fib. By subtracting the time t_0 used by the non-generic fib from the time usage t of each compilation code, we can hopefully compare the times $t - t_0$, purely consumed in the both type case selections.

```
let rec fib = function
| 1 | 2 → 1
| n → fib (n-2) + fib (n-1)
in fib 36;;
```

We compiled these two compilation and the non generic fib by Objective Caml byte code compiler version 3.00 and executed 300MHz Pentium II (not so powerful indeed). By executing fib 36, almost 30 million times of type case selections are done.

	Total time (t)	$t - t_0$
fib, Non generic	$t_0 = 9.66$s	–
fib1, Compiled by pattern matching	43.35s	33.69s
fib2, Compiled with generic flows	26.21s	16.55s

The compilation using generic flows accelerates the program dramatically! The type case selection of the new compilation is twice as fast as the original, pattern match based compilation, even though the type matching in fib is the simplest one. While the overhead of the pattern matching strategy is proportional to the complexity of the type patterns, the efficiency of the generic flow compilation is constant, no matter how the type patterns are complex. This means that our new compilation strategy much faster in general.

8. Conclusion and related works

We have proposed a framework to type ad-hoc, non parametric polymorphism in ML. Because ad-hoc polymorphic values are not fully polymorphic, the static detection of the illegal instances is important. Intentional polymorphism[3] escapes this problem with the restriction that all the generic values are fully polymorphic, which is not in practice. Kinded types are often used to express such partiality of generic values[5], but in this approach the type system becomes complicated and type schemes of complex generic values tend to be almost impossible to be understood by human. On the other hand, in our framework all the partial polymorphic informations are done in the second typing layer and isolated from the first, normal ML typing. This two layered design minimizes the modification to the normal ML type system, therefore the integration to the existing ML implementation seems to be much easier.

The extensional polymorphism [4], on which our study based mainly, translates the partial polymorphic information of generic values into small pieces of programs called abstractions. The validity of the full instances is statically checked by executing this abstraction. Our constraint typing can be regarded as a pulling-back of these abstractions into the type algebra. By analyzing type cases within the type system, the constraint typing system could provide better type reconstruction which is difficult for the abstraction execution.

As far as we know, all the proposed ML systems with ad-hoc polymorphism [3, 4, 5, 6] execute the type case selections using run time types and pattern matching. By the generic flow compilation, we converted this mechanism to simple automata consist of integer trees and array accesses. This simplification improves the efficiency of the generic values well and encourages us to use them more often.

Bibliographie

[1] L. Damas, R. Milner, *Principal type schemes for functional programs*, Proceedings of the 9th Annual ACM Symposium Principles of Programming Languages (1992), 207-212.

[2] A. K. Wright. *Polymorphism for imperative languages without imperative types.* Technical report 200, RICE University, 1993.

[3] R. Harper and G. Morrisett *Compiling polymorphism using intensional type analysis* In Conference Record of the 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 130–141, San Francisco, California, January 1995. ACM.

[4] Catherine Dubois, François Rouaix and Pierre Weis. *Extensional polymorphism* Proceedings of the 22th ACM Conference on Principles of Programming Languages, January 1995

[5] D. Duggan, G. Cormack and J. Ophel. Kinded type inference for parametric overloading. Acta Informatica 33, 1:21-68, 1996.

[6] R. Hinze *A generic programming extension for Haskell*, Proceedings of the 3rd Haskell Workshop, Paris (1999)

[7] R. Hudak, S. P. Jones, and P. Wadler. *Report on the programming language Haskell, version 1.2.* Technical report, Yale University, 1992.

[8] F. Rouaix. *Safe run-time overloading.* In 17th symposium Principles of Programming Languages, pages 355-366. ACM Press, 1990

[9] Milner, Tofte, Harper, MacQueen *The Definition of Standard ML (Revised)* MIT Press, 1997.

[10] Xavier Leroy, Didier Rémy, Jérôme Vouillon and Damien Doligez. *The Objective CAML reference manual* Available at `http://pauillac.inria.fr/caml/ocaml/htmlman/index.html`