Chapter 7

Generic definitions

7.1 Introduction

User-defined generic values

We have discussed the typing and compilation of extensional polymorphism in the previous chapters. There, all the generic values were always primitives. In this chapter we argue the formalization of the generic value definitions – *user-defined generic values* (or just *user generics*). A user-defined generic value is declared as a set of pairs of type contexts and corresponding code by the new `generic` binding construct. We introduce the syntax, typing and semantics of the `generic` binding and its instances as extensions of the GML system.

Compilation of user generics

In this chapter, user-defined generics are compiled using pattern matching. Their compilation compare evaluation type contexts with the static type patterns given at definition time. user-defined generics receive total parametric type schemes, but usually they are non-exhaustive; their semantics is not defined for all the instance types of their type schemes. If an instance of user-defined generic value is evaluated in a type context that has no corresponding definition clause, the pattern matching fails and raises a run time error.

Outline of this chapter

We first introduce intuitive notions of user definition of generic values in Section 7.2. Its formal syntax is explained in Section 7.3. In Section 7.4, we extend the typing system and its reconstruction algorithm for `generic` bindings. Section 7.5 presents the type pattern match based semantics of user generics and its type soundness with respect to the extended type system. Section 7.6 defines a compilation algorithm of generic bindings to IML, using pattern matching of run time types.

7.2 user-defined generic values

7.2.1 Generic bindings

An overloaded value can be characterized by a set of pairs of a type scheme and an expression, a type scheme for which the value is defined and the corresponding expression which serves the semantics for the type context. An example of overloaded plus function which adds values in various data types can consist of the following pairs:

type schemes	semantic expression	
`int` $\rightarrow$ `int` $\rightarrow$ `int`	`(+)`	Integer plus
`float` $\rightarrow$ `float` $\rightarrow$ `float`	`(+.)`	Float plus
`string` $\rightarrow$ `string` $\rightarrow$ `string`	`(^)`	String concatenation

We call these pairs *type cases*. The definition of the generic function `plus` is declaring these type cases and binding them to the identifier `plus`. Therefore, its definition can be written like the following expression:

```
generic plus = case
| int → int → int ⇒ (+)
| float → float → float ⇒ (+.)
| string → string → string ⇒ (^)
;;
```

This *generic binding* expression will define a generic value `plus`, whose type scheme is $\forall\delta.\ \delta \rightarrow \delta \rightarrow \delta$. This type scheme is the least general anti-unifier of the type schemes defined for the generic value `plus` and will be inferred automatically by the type system. Each type scheme at the left-hand side of the symbol $\Rightarrow$ is called a *type pattern* (or just a pattern) and the corresponding right-hand side expression is called the *type method* (or just the method) of the type pattern. We can use this `plus` generic function for adding integers and floats and concatenation for strings:

```
# plus 1 2;;
- : int = 3

# plus 1.0 3.14;;
- : float = 4.14

# plus "hello " "world";;
- : string = "hello world"
```

We can add any kind of "adding" function as a type case to this definition, including addition of polymorphic data types. For example, we here overload the list concatenation:

```
generic plus = case
...
| ∀δ.δ list → δ list → δ list ⇒ (@)
;;
```

In the type pattern for the list concatenation `(@)`, we use a polymorphic type scheme $\forall\delta.\delta$ `list` $\rightarrow \delta$ `list` $\rightarrow \delta$ `list` in the pattern so that `plus` can be used as any instance of this type scheme. Note that we do not use regular type variables but dynamic ones in the pattern, since type patterns should be less general than the type scheme of the corresponding generic value (See Section 7.7.1 for more details). Dynamic type variables in type patterns are treated like pattern variables in the normal pattern matching: they match with any type. Therefore, this type case can be used for any instance of this type pattern, for example `int list` $\rightarrow$ `int list` $\rightarrow$ `int list`, and so on:

```
# plus [1] [2; 3];;
- : int list = [1; 2; 3]

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

Now the definition of `plus` is written as follows:

```
generic plus = case
| int → int → int ⇒ (+)
| float → float → float ⇒ (+.)
| string → string → string ⇒ (^)
| ∀δ.δ list → δ list → δ list ⇒ (@)
;;
```

We have no restrictions between type cases: for example, type patterns can have different functional arities and there is no problem for merging non-functional and functional methods. For instance, we define a generic identity which works for several data types. If `ident` is used as a function typed $\tau \rightarrow \tau$, `ident` returns the identity function while for the other type patterns it returns non functional values:

```
generic ident = case
| int ⇒ 1
| float ⇒ 1.0
| unit ⇒ ()
| ∀δ.δ → δ ⇒ fun x → x
| ∀δ.δ list ⇒ []
;;
```

7.2.2 Recursive generics

Now let's think about a print procedure, which prints values of some basic types and the types constructed by those basic types. Of course, how to print values is heavily dependent on their types, therefore we need to define `print` as a generic polymorphic function. The definition of `print` start with the following type cases for basic type printing:

```
generic print = case
| unit → unit ⇒ fun _ → print_string "()"
| int → unit ⇒ print_int
| char → unit ⇒ print_char
| float → unit ⇒ print_float
| string → unit ⇒ print_string
...
;;
```

We can add any printing method easily, just until we consider polymorphic data structures. For example, how do we print lists? One easy but naïve idea is to define a derived generic function to print lists using the above `print` procedure for basic cases:

```
let rec print_list = function
| [] → ()
| x :: xs → print x; print_list xs
;;
val print_list : ∀δ. δ list → unit

# print_list [1;2;3];;
123

# print_list ["hello "; "world"];;
hello world
```

This generic `print_list` works for `int list`, `string list` and so on, however, it cannot print values of τ `list list`, because `print` has no type case for `list` itself.

To define a function which can print any kind of list, including higher order lists, we need to integrate `print_list` into the definition of `print`. For this purpose `print` must be defined recursively, using a *generic recursive binding*:

```
generic rec print = case
| unit → unit ⇒ fun _ → print_string "()"
| int → unit ⇒ print_int
| char → unit ⇒ print_char
| float → unit ⇒ print_float
| string → unit ⇒ print_string
| ∀δ.δ list → unit ⇒
     (function
      | [] → ()
      | x::xs → print x; print xs)
;;
```

Look at the type method for $\forall\delta.\delta$ `list` → `unit`. The method receives a list and if it is non-nil, say `x::xs`, it calls recursively `print` twice. The second occurrence of `print` for the tail of the list `xs` is a normal recursive call on the structure of the list cons-cells, which is familiar to us. On the other hand, the first one is something new in the normal ML world. This is a recursion on *the structure of the type* of the argument of `print`. It prints the content of `x`, using its corresponding type method. This *recursion on types* enables printing not only lists of `int`, `char`, `float` and `string` but also any multi-list like `int list list list`:

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

# print [["Hello "; "world "]; ["I "; "am "; "happy"]];;
Hello world I am happy
```

The combination of extensional polymorphism and recursion allows much more complicated generic definitions than mere simple “overloaded operators”. The following generic function `flat` demonstrates such an example. 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 flattens just one level of a list. The second type case is the base case of this recursion: it just maps 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 depending on which type it is instantiated to, and finally returns a list of level 1:

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

The last example is `vprint`, a printing function with variable length arguments, using the above `print` inside its definition:

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

If it is evaluated as a function, it takes an argument and print it using the generic `print`, then returns `vprint` itself. Thus `vprint` can receive an arbitrary number of arguments until it is evaluated to the type `unit`.

One may notice that recursive generic definitions require *polymorphic recursion*: for example, `print` is bound with a polymorphic type scheme $\forall\delta.\delta \rightarrow$ `unit` and instantiated to different type instance $\delta \rightarrow$ `unit` and δ `list` $\rightarrow$ `unit` at its recursive occurrences. We escape the undecidability of the type reconstruction of the polymorphic recursion by the explicit declaration of type patterns. We will elaborate on this in Section 7.4.5, when describing the type reconstruction of generic bindings.

7.2.3 Generic run time errors

user-defined generic values are often *non-exhaustive*. Unlike the generic primitives that are supposed to be exhaustive and defined for all the instances of their type schemes, a user-defined generic value is defined just for a limited set of instances. If a user-defined generic value is used in a type context where its semantics is not defined, the evaluation fails and a run time exception is raised. For example, the `plus` generic function is not defined for a type `unit` → `unit` → `unit`:

```
Uncaught exception: Type_match_failure(plus,unit → unit → unit)
```

This is similar to the match failure exception of non-exhaustive pattern matching in ML:

```
# let parse_digit c =
    match c with
    | '0'...'9' → int_of_char c - int_of_char '0';;
Warning: this pattern-matching is not exhaustive.
val parse_digit : char → int
# parse_digit '5';;
- : int = 5
# parse_digit 'x';;
Uncaught exception: Match_failure
```

If a pattern matching expression is non-exhaustive and a value does not match any of its patterns, the match failure is reported as a run time exception.

As pattern match failure exceptions, generic type match failures can be caught by an exception handling, therefore it is not so catastrophic as a complete run time crash like a segmentation fault or a bus error. But of course it is preferable to prevent those failures by a static analysis. The detection of illegal use of non-exhaustive generic values is complicated, since the type system must be aware of non-exhaustiveness of user-defined generic values. Therefore in this chapter, we do not try to formalize this detection and leave such misuses well-typed. The detection of illegal uses of generic values will be discussed later, in Chapter 8.

7.3 Formal syntax

We now formally extend the grammar of GML by adding the generic binding:

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

Like `let` expressions, a `generic` expression first takes *definition* part and is followed by a *body* after the `in` keyword. The syntax of the type annotated GML also need to be extended in the same manner:

$$
\begin{array}{rcl}
e_t & ::= & \ldots \\
 & | & \mathtt{generic}\ \mathtt{x}_\sigma = \mathtt{case}\ tc_t \mid \ldots \mid tc_t\ \mathtt{in}\ e_t \\
 & | & \mathtt{generic\ rec}\ \mathtt{x}_\sigma = \mathtt{case}\ tc_t \mid \ldots \mid tc_t\ \mathtt{in}\ e_t \\
tc_t & \equiv & \sigma \Rightarrow e_t
\end{array}
$$

As we have explained, type schemes in type case expressions tc and tc_t are restricted to closed ones without regular type variables.

Various tool functions for the fully type annotated GML have also to be extended for type cases and generic bindings:

$$FV(\sigma \Rightarrow e_t) = FV(e_t) \setminus BV(\sigma)$$

Quantified type variables in type patterns bind the free type variable occurrences of the corresponding type methods, like the generalized type variable set in a `let` binding binds the variables in the type annotations of its definition. Renaming bound variables of type cases are regarded as α-equivalent and we identify α-equivalent type cases:

$$
\begin{array}{l}
(\forall\delta_1 \ldots \delta_n.\tau \Rightarrow e_t) \stackrel{\alpha}{=} (\forall\delta\delta_1 \ldots \delta_n.\tau \Rightarrow e_t) \text{ when } \delta \notin FV(\forall\delta_1 \ldots \delta_n.\tau \Rightarrow e_t) \\
(\forall\delta_1 \ldots \delta_n.\tau \Rightarrow e_t) \stackrel{\alpha}{=} (\forall\delta_1 \ldots \delta'_i \ldots \delta_n.S(\tau) \Rightarrow S(e_t)) \\
\qquad\qquad\qquad \text{where } S = \{\delta_i \Rightarrow \delta'_i\},\ \delta'_i \notin \{\delta_i\} \cup FV(\tau) \cup FV(e_t)
\end{array}
$$

The substitution on type cases is defined as follows, assuming that if necessary we α-convert type cases and rename bound variables so that the bindings are not modified by the substitution:

$$S(\forall V.\tau \Rightarrow e_t) = (\forall V.S(\tau) \Rightarrow S(e_t))$$

Actually, since the type patterns are restricted to closed type schemes, this definition is simply rewritten as $S(\sigma \Rightarrow e_t) = (\sigma \Rightarrow S(e_t))$.

With the help of these functions for type cases, the free variables of generic bindings and their substitutions are defined as follows:

$$FV(\texttt{generic (rec) x}_\sigma \texttt{ = case } tc_{t1} \mid \ldots \mid tc_{tn} \texttt{ in } e_t) = FV(\sigma) \cup FV(tc_{ti}) \cup FV(e_t)$$
$$S(\texttt{generic (rec) x}_\sigma \texttt{ = case } tc_{t1} \mid \ldots \mid tc_{tn} \texttt{ in } e_t) =$$
$$\texttt{generic (rec) x}_{S(\sigma)} \texttt{ = case } S(tc_{t1}) \mid \ldots \mid S(tc_{tn}) \texttt{ in } S(e_t)$$

7.4 Typing of generic bindings

We construct a typing system for the generic binding as an extension of one for GML in Section 2.5. In the type system, user-defined generic values have dynamic polymorphic types. User generics can be instantiated to any instance of their polymorphic type schemes here.

7.4.1 Anti-unification

The type scheme for a user-defined generic value is inferred by our typing rules as the *anti-unifier* of the type patterns so that all the type patterns can be obtained as instances of this type scheme. Anti-unifier of types is the dual notion of unifier of types. The precise definition of anti-unifier is as follows:

Definition 10 (Anti-unifier) *An* anti-unifier τ *of two types* τ_1 *and* τ_2 *is a type expression more general than both, i.e.:*

$$\exists S_1, S_2,\ \tau_1 = S_1(\tau),\ \tau_2 = S_2(\tau)$$

This notion can be easier extensible for non-empty sets of types:

Definition 11 (Anti-unifier for a set of types) *An* anti-unifier τ *of non-empty set of type expressions* T *is a type expression more general than any element of* T*, i.e.:*

$$\forall \tau' \in T,\ \exists S,\ \tau = S(\tau')$$

There can be more than one anti-unifier for types, as types can have more than one unifier. For example, types $\texttt{int} \rightarrow \texttt{unit}$ and $\texttt{float} \rightarrow \texttt{unit}$ have two anti-unifiers, modulo the renaming of type variables: $\alpha \rightarrow \texttt{unit}$, and α. But only one anti-unifier of the type patterns of a generic binding determines its type scheme: the *least general anti-unifier*, the dual notion of the MGU (most general unifier) of types. The least general anti-unifier is the lower bound of anti-unifiers:

Definition 12 (Least General Anti-unifier) *The* least general anti-unifier *of types τ_1 and τ_2 is the least general type expression of the set of anti-unifiers of τ_1 and τ_2. That is, if we suppose τ is the least general anti-unifier of them and τ' is another anti-unifier, then there exists a substitution S such that $\tau = S(\tau')$.*

This definition is also extended to non-empty sets of types:

Definition 13 (Least General Anti-unifier for a set of types) *The* least general anti-unifier *τ of non-empty set of type expressions T is one of the least general type expressions of the set of anti-unifiers A_T of T such that*

$$\forall \tau' \in A_T,\ \exists S,\ \tau = S(\tau')$$

For any non-empty set of types there always exists an anti-unifier, because α is an anti-unifier of any non-empty set of types. Therefore the least general anti-unifier also exists.

7.4.2 Anti-unification algorithm

The following anti-unification algorithm *antiunif* in Figure 7.1 is a dual of the unification algorithm *unify*. It calculates the least general anti-unifiers of a non-empty set of types. The algorithm is defined as the repeating application of *aunif*, the anti-unification algorithm for two types. *aunif* is defined using a memo function Φ which maps pairs of type expressions to variables. Φ keeps track of the anti-unified pairs, and if the algorithm *aunif* tries to anti-unify two sub type expressions τ_1 and τ_2 already stored in Φ, we replace them by $\Phi(\tau_1, \tau_2)$. If they are not registered in Φ and their type constructs are different, the algorithm creates a fresh type variable ν_{fresh} and adds a binding $(\tau_1, \tau_2) \Rightarrow \nu_{fresh}$ to Φ, then return ν_{fresh}. The kind of the fresh type variables are determined if the argument types contain some regular type variables or not.

The algorithm *aunif* calculates the least general anti-unifier of two given type expressions. The general anti-unification algorithm *antiunif* for non-empty, any number of type expressions T can be defined as multiple compositions of the *aunif* algorithm for each type expressions of T. To prove that *aunif* and *antiunif* return the least general anti-unifier of types, we need first to prove the following properties of the memo function Φ:

Lemma 14 (Consistency of the memo-function) *Let τ_1 and τ_2 be type expressions. Suppose $aunif(\emptyset, \tau_1, \tau_2) = (\Phi, \tau_0)$ for some Φ and τ_0. Then, for all the bindings in Φ, we have $((\tau, \tau') \Rightarrow \nu)$, we have $aunif(\emptyset, \tau, \tau') = (\{(\tau, \tau') \Rightarrow \nu_{fresh}\}, \nu_{fresh})$ where ν and ν_{fresh} are the least general anti-unifiers of τ_1 and τ_2.*

PROOF: The bindings of Φ are added by the last cases of the *aunif* definition, when the type constructs of τ and τ' are different. In such case, the possible anti-unifiers must be type variables, and if one of the arguments contains a regular type variable, the anti-unifier type variables must be regular in addition. The additional bindings to the memo-function correspond to this fact. □

$aunif(\Phi,\ \tau,\ \tau') = \Phi, \Phi(\tau,\ \tau')$, when $\Phi(\tau,\ \tau')$ is defined

otherwise,
$aunif(\Phi, base_1, base_2) = (\Phi, base_1)$, when $base_1 = base_2$

$aunif(\Phi, \tau_1 \rightarrow\ \tau_2,\ \tau_1' \rightarrow\ \tau_2') =$
let $(\Phi_1, \tau_1'') = aunif(\Phi, \tau_1,\ \tau_1')$ in
let $(\Phi_2, \tau_2'') = aunif(\Phi_1, \tau_2,\ \tau_2')$ in
$(\Phi_2, \tau_1'' \rightarrow \tau_2'')$

$aunif(\Phi, \tau_1 * \tau_2,\ \tau_1' * \tau_2') =$
let $(\Phi_1, \tau_1'') = aunif(\Phi, \tau_1,\ \tau_1')$ in
let $(\Phi_2, \tau_2'') = aunif(\Phi_1, \tau_n,\ \tau_n')$ in
$(\Phi_2, \tau_1'' * \tau_2'')$

otherwise,
if $RV(\tau) \cup RV(\tau') = \emptyset$, $aunif(\Phi, \tau,\ \tau') = (\Phi\{(\tau,\ \tau') \Rightarrow \delta_{fresh}\}, \delta_{fresh})$
else $aunif(\Phi, \tau,\ \tau') = (\Phi\{(\tau,\ \tau') \Rightarrow \alpha_{fresh}\}, \alpha_{fresh})$

$antiunif(\{\tau\}) = \tau$

$antiunif(\{\tau_1, \ldots, \tau_n\}) =$
let $\tau_1' = \tau_1$ in
let $(\Phi_2, \tau_2') = aunif(\emptyset, \tau_1',\ \tau_2)$ in
$\cdots$
let $(\Phi_i, \tau_i') = aunif(\emptyset, \tau_{i-1}',\ \tau_i)$ in
$\cdots$
let $(\Phi_n, \tau_n') = aunif(\emptyset, \tau_{n-1},\ \tau_n)$ in
τ_n'

Figure 7.1: Anti-unification algorithm

We say a memo-function Φ is *consistent* if it suffices the above property.

Lemma 15 (Globality of *aunif*'s memo-function) *Let Φ be a consistent memo-function and $aunif(\Phi, \tau_1, \tau_2) = (\Phi', \tau_0)$ then $aunif(\Phi', \tau_1, \tau_2) = (\Phi', \tau_0)$*

PROOF: By induction on the recursive calls of *aunif* in $aunif(\Phi, \tau_1, \tau_2)$. All the fresh variable creations in $aunif(\Phi, \tau_1, \tau_2)$ are now replaced as queries to the memo-function Φ'. □

Lemma 16 (*aunif* returns LGA) *For any type τ and τ', and a consistent memo-function Φ, $aunif(\Phi, \tau, \tau')$ returns the least general unifier of them.*

PROOF: By induction over the structure of the argument types.

Types in Φ If the pair of the arguments (τ, τ') is bound in the memo-function Φ, from Lemma 14, $aunif(\emptyset, \tau, \tau') = (\{(\tau, \tau') \Rightarrow \nu_{\mathit{fresh}}\}, \nu_{\mathit{fresh}})$ and ν_{fresh} is the least general anti-unifier of them.

Basic types If the two arguments are the same basic type *base*, then $aunif(\Phi, base, base) = (\Phi, base)$ and this *base* is clearly the LGA of the arguments.

If they are different, $aunif(\Phi, base_1, base_2) = \{(base_1, base_2) \Rightarrow \delta_{\mathit{fresh}}\}, \delta_{\mathit{fresh}}$ The only possible anti-unifiers of the base types are type variables ν, and we can create admissible substitutions $S = \{\nu \Rightarrow \delta_{\mathit{fresh}}\}$.

Type variables Similar to the proof for the basic types.

Arrows Let $\tau = \tau_1 \to \tau_2$ and $\tau' = \tau'_1 \to \tau'_2$. Then

$$
\begin{array}{l}
aunif(\Phi, \tau_1 \to \tau_2,\ \tau'_1 \to \tau'_2) = \\
\quad \text{let } \Phi_1, \tau''_1 = aunif(\Phi, \tau_1,\ \tau'_1) \text{ in} \\
\quad \text{let } \Phi_2, \tau''_2 = aunif(\Phi_1, \tau_2,\ \tau'_2) \text{ in} \\
\quad (\Phi_2, \tau''_1 \to \tau''_2)
\end{array}
$$

From Lemma 15,

$$
\begin{array}{l}
aunif(\Phi_2, \tau_1 \to \tau_2, \tau'_1 \to \tau'_2) = (\Phi_2, \tau''_1 \to \tau''_2) \\
aunif(\Phi_2, \tau_1, \tau'_1) = (\Phi_2, \tau''_1) \\
aunif(\Phi_2, \tau_2, \tau'_2) = (\Phi_2, \tau''_2)
\end{array}
$$

Suppose there is an anti-unifier τ'' of τ and τ'. τ'' must be a type variable or a functional type, otherwise it cannot be an anti-unifier.

If τ'' is a regular type variable α, then there is a substitution $S = \{\alpha \Rightarrow \tau''_1 \to \tau''_2\}$ which maps α to the result of *aunif*.

If τ'' is a dynamic type variable δ, then since δ is an anti-unifier of τ and τ', $RV(\tau) = RV(\tau') = \emptyset$. From the definition of *aunif*, it is easily verified that $\tau_1'' \to \tau_2''$ contains no regular type variable either. Therefore the substitution $S = \{\delta \Rightarrow \tau_1'' \to \tau_2''\}$ is admissible and it maps δ to the $\tau_1'' \to \tau_2''$.

If $\tau'' = \tau_1''' \to \tau_2'''$, because τ'' is an anti-unifier of $\tau_1 \to \tau_2$ and $\tau_1' \to \tau_2'$, there exist substitutions S and S' such that $S(\tau_1''' \to \tau_2''') = \tau_1 \to \tau_2$ and $S'(\tau_1''' \to \tau_2''') = \tau_1' \to \tau_2'$ By the induction hypothesis, τ_1'' is the least general anti-unifier of τ_1 and τ_1', and τ_2'' is of τ_2 and τ_2'. Therefore there exist substitutions S_1 and S_2 such that $S_1(\tau_1''') = \tau_1''$, $S_2(\tau_2''') = \tau_2''$. Suppose that ν is a type variable which is in the both of τ_1''' and τ_2'''. This ν is mapped to $S(\nu)$ in the both of τ_1 and τ_2 and $S'(\nu)$ in τ_1' and τ_2'. In the calculation $aunif(\Phi_2, \tau_1 \to \tau_2, \tau_1' \to \tau_2')$, we have the calls of $aunif(\Phi_2, S(\nu), S'(\nu))$ in the both the anti-unification of sub-expressions, $(\Phi_2, \tau_1'') = aunif(\Phi_2, \tau_1, \tau_1')$ and $(\Phi_2, \tau_2'') = aunif(\Phi_2, \tau_2, \tau_2')$. S_1 and S_2 map ν to its corresponding sub-expression in τ_1'' and τ_2'': $S_1(\nu) = S_2(\nu) = aunif(\Phi_2, S(\nu), S'(\nu))$.

Because thus the shared type variables in τ_1''' and τ_2''' are mapped to the same type expression by S_1 and S_2, we can merge S_1 and S_2 consistently. More precisely, if we let

$$S_0 = \{(\nu \Rightarrow S_1(\nu)) \mid \nu \in FV(\tau_1''')\} \cup \{(\nu \Rightarrow S_2(\nu)) \mid \nu \in FV(\tau_2''') \setminus FV(\tau_1''')\}$$

then $S_0(\tau_1''') = \tau_1''$ and $S_0(\tau_2''') = \tau_2''$. Consequently, $S_0(\tau_1''' \to \tau_2''') = \tau_1'' \to \tau_2''$.

Tuples Similar to the proof for the arrow types.

The others $aunif(\Phi, \tau, \tau') = (\Phi\{(\tau, \tau') \Rightarrow \nu_{fresh}\}, \nu_{fresh})$ and ν_{fresh} is one of the least general anti-unifier of τ and τ' from Lemma 14.

□

Theorem 19 (*antiunif* returns LGA) *For any non empty set of type expressions T, antiunif(T) returns the least general unifier of T.*

Proof: If T is singleton or pair, it is obvious. For the others, it is sufficient to prove the case of triplets. Let $T = \{\tau_1, \tau_2, \tau_3\}$ and $\tau_0 = antiunif(T)$. Then, $\tau_0 = aunif(\emptyset, \tau_1, aunif(\emptyset, (\tau_2, \tau_3)))$.

First we prove τ_0 is an anti-unifier of T: let $\tau' = aunif(\emptyset, (\tau_2, \tau_3))$. Then, since $\tau_0 = aunif(\emptyset, \tau_1, \tau')$, there is a substitution S_1 and S' such that $S_1(\tau_0) = \tau_1$ and $S'(\tau_0) = \tau'$. Also, there is a substitution S_2 and S_3 such that $S_2(\tau') = \tau_2$ and $S_3(\tau') = \tau_3$, since τ' is an anti-unifier of τ_2 and τ_3. From these τ_0 is an anti-unifier of T: $S_1(\tau_0) = \tau_1$, $S_2(S'(\tau_0)) = \tau_2$ and $S_3(S'(\tau_0)) = \tau_3$.

Let τ be an anti-unifier of T. Because τ anti-unify τ_2 and τ_3, there exists a substitution S such that $S(\tau) = \tau' = aunif(\emptyset, (\tau_2, \tau_3))$. Therefore τ is also an anti-unifier of τ_1 and τ'. Since τ_0 is the least anti-unifier of τ_1 and τ', there is a substitution S'' such that $S''(\tau) = \tau_0$. □

7.4.3 Anti-unification of type patterns

The type scheme of a user-defined generic value is an anti-unifier of its type patterns, so that all the type patterns can be contained in (e.g. less general than) the type scheme of the value. Such type scheme may have type instances which have no corresponding definitions. To minimize the possibility of having such unwelcome type instances, the anti-unifier must be the *least general*.

If $\sigma_1, \ldots, \sigma_n$ are the type patterns, then their anti-unification is given by the following *pattern anti-unification* $p\_antiunif(\{\sigma_i\})$:

$$p\_antiunif(\{\sigma_i\}) = \text{ let } \tau = antiunif(\{inst(\sigma_i)\}) \text{ in } \forall FV(\tau).\tau$$

This is the least general type scheme (up to type variable renaming) which are more general than any of the type patterns. Since the type patterns cannot contain any regular type variable, the result of pattern anti-unification does not have regular type variables either.

Example The following generic definition:

```
generic g = case
| ∀δ.δ list → unit ⇒ ...
| ∀δδ'.(δ * δ') → unit ⇒ ...
```

defines g whose type scheme is

$$p\_antiunif(\{\forall\delta.\delta \texttt{ list} \rightarrow \texttt{unit}, \forall\delta\delta'.(\delta * \delta') \rightarrow \texttt{unit}\}) = \forall\delta.\delta \rightarrow \texttt{unit}$$

□

7.4.4 Typing rules

Using the anti-unification algorithm, the typing rules for generic bindings are defined as follows:

$$\textbf{(t-generic)}\ \frac{\Gamma \vdash e_{ti} : \tau_i \quad \sigma_i = Gen(\Gamma, e_{ti} : \tau_i) \quad FV(\sigma_i) = \emptyset \quad BV(\sigma_i) \subset \mathbb{D} \qquad \sigma = p\_antiunif(\{\sigma_i\}) \quad \Gamma\{\texttt{x} : \sigma\} \vdash e_t : \tau}{\Gamma \vdash \texttt{generic x}_\sigma \texttt{ = case } \sigma_1 \Rightarrow e_{t1} \mid \ldots \mid \sigma_n \Rightarrow e_{tn} \texttt{ in } e_t : \tau}$$

$$\textbf{(t-genericrec)}\ \frac{\Gamma\{\texttt{x} : \sigma\} \vdash e_{ti} : \tau_i \quad \sigma_i = Gen(\Gamma, e_{ti} : \tau_i) \quad FV(\sigma_i) = \emptyset \quad BV(\sigma_i) \subset \mathbb{D} \qquad \sigma = p\_antiunif(\{\sigma_i\}) \quad \Gamma\{\texttt{x} : \sigma\} \vdash e_t : \tau}{\Gamma \vdash \texttt{generic rec x}_\sigma \texttt{ = case } \sigma_1 \Rightarrow e_{t1} \mid \ldots \mid \sigma_n \Rightarrow e_{tn} \texttt{ in } e_t : \tau}$$

The typing of generic bindings is almost explicit. The type schemes of the user-defined generic values are the pattern anti-unifications of the type patterns. The methods must be generalizable if it contains type variables and their type scheme $Gen(\Gamma, e_{ti} : \tau_i)$ must be closed and the corresponding type patterns must be more general than them. For the recursion on types, in (t-genericrec), generic recursive bindings bind themselves *polymorphically* inside their environment, unlike recursive let binding.

The least general anti-unifier provides the best type information in the current typing system. For example,

```
generic plus = case
| int → int → int ⇒ (+)
| float → float → float ⇒ (+.)
| string → string → string ⇒ (^)
| ∀δ.δ list → δ list → δ list ⇒ (@)
;;
```

`plus` is given the least general anti-unifier of the type patterns, that is $\forall\delta.\delta \to \delta \to \delta$: a more general anti-unifier such that $\forall\delta.\delta$ is less appropriate, even though $\forall\delta.\delta$ can be instantiated to all these type patterns. In effect, $\forall\delta.\delta$ accept some properly rejected types by the LGA, such as `int` and `int` $\to$ `int`.

7.4.5 Polymorphic recursion in generic binding

As we see in the typing rules of generic bindings (t-generic*), the type patterns are the type schemes of the corresponding type method expressions. It seems that they are automatically deduced from the typing of methods, but actually, they cannot. This is because we have the *polymorphic recursion* inside generic recursive definitions, hence the type reconstruction is undecidable unless type schemes are explicitly given:

```
generic rec print = case
| unit → unit ⇒ fun _ → print_string "()"
| int → unit ⇒ print_int
| char → unit ⇒ print_char
| float → unit ⇒ print_float
| string → unit ⇒ print_string
| ∀δ.δ list → unit ⇒
     (function
      | [] → ()
      | x::xs → print x; print xs)
;;
```

In the last type case of the above code, the first occurrence of the recursive uses of `print` has type $\delta \to$ `unit` which is different from the type of the type pattern δ `list` $\to$ `unit`. This typing is achieved by storing `print` as a polymorphic value with type scheme $\forall\delta.\delta \to$ `unit` inside its definition. This is known as *polymorphic recursion* which is not permitted in the type system of the pure ML, Milner-Hindley calculus [DM82]: in the recursive let definition, the recursive occurrences of the defined values are monomorphic (look at (t-letrec) for the typing rule for recursive lets). The following definition of `f`:

```
let rec f x = f 1; f true; x;;
```

cannot have the type scheme $\forall\alpha.\alpha \rightarrow \alpha$,, but is ill-typed, since `f` is used in two different monomorphic ways inside its definition: `int` $\rightarrow$ `int` and `bool` $\rightarrow$ `bool`. Usually, the polymorphic recursion is forbidden in ML implementations, since it is proven that the type inference is undecidable [Myc84]. However, the polymorphic recursion is the key to the recursion on the structure of types in extensional polymorphism. To make the type inference of generic bindings decidable, we simply force users to declare the type scheme of each type cases as a type pattern:

```
#  generic rec f = case
  | ∀δ.δ → δ ⇒ fun x → f 1; f true; x
  ;;
val f : ∀δ.δ → δ = <fun>
```

Knowing the type scheme of `f` as $\forall\delta.\delta \rightarrow \delta$ explicitly before the type inference, `f` can be polymorphic inside its definition. This "polymorphic recursion by explicit types" solution is also used in the type reconstruction of the polymorphic recursions in Haskell [HJW92]:

```
f :: a → a
f x = let q = f True in
      let r = f 0 in
      x
```

We do not dare introducing the various research results of the restricted kinds of polymorphic recursion type reconstructions such as [Hen93, PL99], since even without the polymorphic recursion, writing the explicit type patterns in the generic bindings still plays a very important role to keep away programmers from writing too general type cases than his intention. Just from the point of view of the type soundness, writing type patterns are not mandatory for the type cases without recursive calls. We could just adapt reconstructed type schemes of methods as their implicit type patterns. However, without type patterns, programmers could make mistakes easily. For instance, look at the following imaginary (bad) definition of `print`, without the explicit type patterns (they are commented out). We still need the explicit type pattern for the last type case, since it contains a polymorphic recursive call:

```
generic rec print = case
| (* unit → unit  ⇒ *) fun _ → print_string "()"
| (* int → unit  ⇒ *) print_int
| (* char → unit  ⇒ *) print_char
| (* float → unit  ⇒ *) print_float
| (* string → unit  ⇒ *) print_string
| ∀δ.δ list → unit ⇒
     (function
      | [] → ()
      | x::xs → print x; print xs)
;;
val print : ∀δ. δ → unit
```

```
# print "hello"
()
- : unit = ()
```

In the above code, the programmer intends that the first type method is for printing unit. Since `()` is the only value in the type unit, he omitted writing the function pattern and used a wild-card '`_`'. Without writing the type patterns explicitly, this function would print the string "`()`" for any value, because the first method is typed as $\delta \rightarrow$ `unit`, thus its implicit type pattern is $\forall\delta.\,\delta \rightarrow$ `unit`, which is too general than his intention, `unit` $\rightarrow$ `unit`. Therefore the first type case is selected for any type of arguments. Such a situation happens when a type scheme of a method reconstructed by the type system is more general than the one the programmer had in mind.

To avoid this kind of errors, we force the programmer to put explicit type constraints for all the methods, even if they do not relate with polymorphic recursion. When we define a generic value, we usually decide the type schemes in which the value is defined at first, then we define the corresponding method for each type scheme. Unlike normal definition of ML values, at the moment of writing the definition of a method, its type scheme is clear for the programmers. Therefore, it is not tiresome to write the type patterns explicitly without the help of the type reconstruction algorithm. Furthermore, writing explicit type patterns makes generic bindings more easily readable.

7.4.6 Reconstruction algorithm

The additional cases of the reconstruction algorithm W for generic bindings are in Figure 7.2. This is rather long, but quite straightforward, since the types of the type cases are given explicitly. It first determines the type scheme of a user-defined generic value as the least general anti-unifier of the type patterns, then checks each type method is well-typed and matches with the corresponding type pattern.

Correctness of the reconstruction

We re-prove the theorems we had in Section 2.5, with the addition of generic bindings. We only prove the cases specific to the generic bindings, because the others are proven exactly as before:

Theorem 20 (Extended substitution lemma of the typing rules) *Typing is stable under substitution: if there is a typing* $\Gamma \vdash e_t : \tau$*, then for any admissible substitution* S*,* $S(\Gamma) \vdash S(e_t) : S(\tau)$ *also holds.*

PROOF: By induction over the structure of e_t.

Generic bindings Suppose we have the same typing in (t-generic) rule. Let $\forall V_i.\tau_i = \sigma_i = Gen(\Gamma, e_{ti} : \tau_i)$. Because $FV(\sigma_i) = \emptyset$, we have $V_i = FV(\tau_i)$. Let $V = \bigcup_{i=1}^{n} V_i$ and $S' = \{(\nu_j \Rightarrow \nu'_j) \mid \nu_j \in V,\ \nu'_j \text{ is fresh}\}$. Then $S'(\Gamma) = \Gamma$ because $Dom(S') \cap FV(\Gamma) = \emptyset$. Let $V'_i = S'(V_i)$.

$$
\begin{array}{l}
W(\Gamma, \texttt{generic (rec) x = case } \sigma_1 \Rightarrow e_1 \mid \ \ldots \ \mid \sigma_n \Rightarrow e_n \texttt{ in } e) = \\
\quad \text{if } \ \exists i,\ FV(\sigma_i) \neq \emptyset \text{ or } BV(\sigma_i) \cap \mathbb{R} \neq \emptyset \text{ then fails} \\
\quad \text{let } \forall V_i.\tau_i = \sigma_i \\
\quad \text{let } \sigma = p\_antiunif(\{\sigma_i\}) \\
\quad \text{let } \Gamma_0 = \text{if recursive then } \Gamma\{\texttt{x} : \sigma\} \text{ else } \Gamma \\
\quad \text{let } \Sigma_0 = \emptyset \\
\quad \text{let } \Gamma_1 = \Sigma_0(\Gamma_0) \\
\quad \text{let } e_{t1} : \tau'_1,\ S_1 = W(\Gamma_1, e_1) \\
\quad \text{let } S'_1 = unify(\tau_1, \tau'_1) \\
\quad \text{let } e'_{t1} = S'_1(e_{t1}) \\
\quad \text{let } \Sigma_1 = S'_1 \circ S_1 \\
\quad \text{let } \sigma'_1 = Gen(\Sigma_1(\Gamma_1), e'_{t1}) \\
\quad \text{if } \sigma_1 \overset{\alpha}{\neq} \sigma'_1 \text{ then } W \text{ fails} \\
\quad \ldots \\
\quad \text{let } \Gamma_i = \Sigma_{i-1}(\Gamma_{i-1}) \\
\quad \text{let } e_{ti} : \tau'_i,\ S_i = W(\Gamma_i, e_i) \\
\quad \text{let } S'_i = unify(\tau_i, \tau'_i) \\
\quad \text{let } e'_{ti} = S'_i(e_{ti}) \\
\quad \text{let } \Sigma_i = S'_i \circ S_i \\
\quad \text{let } \sigma'_i = Gen(\Sigma_i(\Gamma_i), e'_{ti}) \\
\quad \text{if } \sigma_i \overset{\alpha}{\neq} \sigma'_i \text{ then } W \text{ fails} \\
\quad \ldots \\
\quad \text{let } \Gamma_n = \Sigma_{n-1}(\Gamma_{n-1}) \\
\quad \text{let } e_{tn} : \tau'_n,\ S_n = W(\Gamma_n, e_n) \\
\quad \text{let } S'_n = unify(\tau_n, \tau'_n) \\
\quad \text{let } e'_{tn} = S'_n(e_{tn}) \\
\quad \text{let } \Sigma_n = S'_n \circ S_n \\
\quad \text{let } \sigma'_n = Gen(\Sigma_n(\Gamma_n), e'_{tn}) \\
\quad \text{if } \sigma_n \overset{\alpha}{\neq} \sigma'_n \text{ then } W \text{ fails} \\
\quad \text{let } e_t : \tau,\ S = W(\Sigma_n(\Gamma_n)[\texttt{x} : \sigma], e) \\
\quad \text{let } \Sigma = S \circ \Sigma_n \circ \ldots \circ \Sigma_0 \\
\quad \text{in } \ \texttt{generic (rec) x}_\sigma \texttt{ = case} \\
\qquad \mid \sigma_1 \Rightarrow (S \circ \Sigma_n \circ \ldots \circ \Sigma_2)(e_{t1}) \\
\qquad \ldots \\
\qquad \mid \sigma_i \Rightarrow (S \circ \Sigma_n \circ \ldots \circ \Sigma_{i+1})(e_{ti}) \\
\qquad \ldots \\
\qquad \mid \sigma_n \Rightarrow S(e_{tn}) \\
\qquad \texttt{in } (e_t : \tau) : \tau,\ \Sigma
\end{array}
$$

Figure 7.2: Reconstruction algorithm for generic bindings

By induction hypothesis, we have

$$(S \circ S')(\Gamma) \vdash (S \circ S')(e_{ti} : \tau_i)$$

where $(S \circ S')(\Gamma) = S(\Gamma)$. The generalization of $(S \circ S')(e_{ti} : \tau_i)$ is

$$\sigma'_i = Gen(S(\Gamma), (S \circ S')(e_{ti}) : (S \circ S')(\tau_i)) = \forall V''_i.(S \circ S')(\tau_i)$$

where $V''_i = FV((S \circ S')(\tau_i)) \setminus FV(S(\Gamma))$. We can prove $V'_i = V''_i$ like we have done in the proof of Theorem 3, let part. In addition, because $FV(\tau_i) \subset Dom(S')$, S' maps all the free type variables in τ_i to fresh variables out of $Dom(S)$. Therefore $(S \circ S')(\tau_i) = S'(\tau_i)$ and we have

$$\sigma'_i = \forall V'_i.S'(\tau_i) \overset{\alpha}{=} \forall V_i.\tau_i = \sigma_i$$

and $FV(\forall V'_i.S'(\tau_i)) = \emptyset$ is clear from the first assumption.

Without losing any generality, we can assume that

$$p\_antiunif(\{\sigma'_i\}) = \sigma$$

and using the induction hypothesis again to the typing for e_t, we have

$$S(\Gamma\{\mathtt{x} : \sigma\}) \vdash S(e_t : \tau)$$

where $S(\Gamma\{\mathtt{x} : \sigma\}) = S(\Gamma)\{\mathtt{x} : \sigma\}$ since $FV(\sigma) = \emptyset$.

From all these, we can use (t-generic) and obtain

$$S(\Gamma) \vdash \mathtt{generic}\ \mathtt{x}_\sigma = \mathtt{case}\ S(\sigma_1 \Rightarrow e_{t1}) \mid \ldots \mid S(\sigma_n \Rightarrow e_{tn})\ \mathtt{in}\ S(e_t) : S(\tau)$$

Recursive generic bindings Similar to the case of normal generic bindings.

□

Theorem 21 (Extended reconstruction correctness) *The extended algorithm W is correct with respect to the typing rules. Let Γ be a typing environment and e be a bare extended GML expression and if $W(\Gamma, e) = e_t,\ S$, then $S(\Gamma) \vdash e_t$.*

Proof: By induction over the structure of bare extended GML expression. The omitted part of the proof is easily obtained from the proof of Theorem 9, replacing the use of Theorem 3 by Theorem 20.

Generic bindings Suppose that we have the same reconstruction as in Figure 7.2. From the first part of the algorithm, it is easily checked that $FV(\sigma_i) = \emptyset$ and $BV(\sigma_i) \subset \mathbb{D}$ and and τ_i is an instance of σ.

By using the induction hypothesis for each reconstruction of the type methods, we have:

$$S_i(\Gamma_i) \vdash e_{ti} : \tau'_i$$

Applying S'_i to this judgment, by Theorem 20,

$(S'_i \circ S_i)(\Gamma_i) \vdash S'_1(e_{t1}) : \tau'_i$, and this is equal to

$$\Sigma_i(\Gamma_i) \vdash e'_{t1} : S'_1(\tau'_i)$$

Using Theorem 20 again with $S \circ \Sigma_n \circ \ldots \circ \Sigma_{i+1}$, we have

$$(S \circ \Sigma_n \circ \ldots \circ \Sigma_i)(\Gamma_i) \vdash (S \circ \Sigma_n \circ \ldots \circ \Sigma_{i+1})(e'_{ti} : S'_i(\tau'_i))$$

and if we let $e''_{ti} = (S \circ \Sigma_n \circ \ldots \circ \Sigma_{i+1})(e'_{ti} : S'_i(\tau'_i))$, then it is equal to

$$\Sigma(\Gamma) \vdash e''_{ti}$$

The type scheme of e'_{ti} is written as $Gen(\Sigma_i(\Gamma_i), e'_{t1} : S'_i(\tau'_i)) = \sigma_i$.

and by applying $S \circ \Sigma_n \circ \ldots \circ \Sigma_{i+1}$ to this, from the same arguments in the `let` part of the proof of Theorem 20, we get

$$\begin{aligned} &Gen((S \circ \Sigma_n \circ \ldots \circ \Sigma_i)(\Gamma_i), (S \circ \Sigma_n \circ \ldots \circ \Sigma_{i+1})(e'_{t1} : S'_1(\tau'_i))) \\ &\qquad\qquad = (S \circ \Sigma_n \circ \ldots \circ \Sigma_{i+1})(\sigma_i) \end{aligned}$$

It is equal to

$$Gen(\Sigma(\Gamma), e''_{ti}) = \sigma_i$$

Applying the induction hypothesis again to the reconstruction of e, we obtain $(S \circ \Sigma_n)(\Gamma_n) \vdash e_t : \tau$, which is equal to

$$\Sigma(\Gamma) \vdash e_t : \tau$$

From these we can use (t-generic) and construct the following typing:

$$\frac{\Sigma(\Gamma) \vdash e''_{ti} \quad \sigma_i = Gen(\Sigma(\Gamma)\{\mathtt{x} : \sigma\}, e''_{ti}) \quad FV(\sigma_i) = \emptyset \quad BV(\sigma_i) \subset \mathbb{D} \quad \sigma = p\_antiunif(\{\sigma_i\}) \quad \Sigma(\Gamma) \vdash e_t : \tau}{\Sigma(\Gamma) \vdash \mathtt{generic}\ \mathtt{x}_\sigma\ = \mathtt{case}\ \sigma_1 \Rightarrow e''_{t1} \mid \ldots \mid \sigma_n \Rightarrow e''_{tn}\ \mathtt{in}\ e_t : \tau}$$

Recursive generic bindings It is similar to the case of the normal generic bindings.

□

The completeness of the type reconstruction is kept as before, because the typing of generic bindings is explicit.

Theorem 22 (Type reconstruction completeness) *The algorithm W is complete. Let Γ be a typing environment and e be a bare GML expression. If there exists a typing $S'(\Gamma) \vdash e : \tau'$ where $Dom(S') \subset FV(\Gamma)$, then $W(\Gamma, e) = (e_t : \tau),\ S$ and there exists a substitution S'' such that*

$$S''(\tau) = \tau', \quad \forall \nu \notin \mathit{fresh}(W(\Gamma, e)), \quad S'(\nu) = S'' \circ S(\nu)$$

PROOF: Here also we omit the proof except the cases for generic bindings, since the proof outline is the same as the original, Theorem 10.

Generic bindings Suppose that e is

$$\texttt{generic x = case } \sigma_1 \Rightarrow e_1 \mid \ldots \mid \sigma_n \Rightarrow e_n \texttt{ in } e_0$$

and e has a typing

$$\frac{S'(\Gamma) \vdash e''_{ti} : \tau''_i \quad \sigma_i = Gen(S'(\Gamma), e''_{ti} : \tau''_i) \quad FV(\sigma_i) = \emptyset \quad BV(\sigma_i) \subset \mathbb{D} \qquad \sigma = p\_antiunif(\{\sigma_i\}) \quad S'(\Gamma)\{\mathtt{x} : \sigma\} \vdash e'_{t0} : \tau'}{S'(\Gamma) \vdash \texttt{generic x}_\sigma\texttt{ = case } \sigma_1 \Rightarrow e'_{t1} \mid \ldots \mid \sigma_n \Rightarrow e''_{tn} \texttt{ in } e'_{t0} : \tau'}$$

Let's follow the algorithm $W(\Gamma, e)$. First, we can easily see that σ_i are all closed without regular type variables, σ in the algorithm W is equal to the above typing.

To prove that the reconstructions of the type cases are all successful, we need to prove the following property inductively over i:

If there is a substitution Σ'_i such that $\Sigma'_i(\Gamma_i) = S'(\Gamma)$, the type case reconstruction of $(\sigma_i \Rightarrow e_i)$

let $\Gamma_i = \Sigma_{i-1}(\Gamma_{i-1})$
let $e_{ti} : \tau'_i,\ S_i = W(\Gamma_i, e_i)$
let $S'_i = \mathit{unify}(\tau_i, \tau'_i)$
let $e'_{ti} = S'_i(e_{ti})$
let $\Sigma_i = S'_i \circ S_i$
let $\sigma'_i = Gen(\Sigma_i(\Gamma_i), e'_{ti})$
if $\sigma_i \not\stackrel{\alpha}{=} \sigma'_i$ then W fails

is successful and there is a substitution Σ'_{i+1} such that $\Sigma'_{i+1}(\Sigma_i(\Gamma_i)) = S'(\Gamma)$.

Using the induction hypothesis for e_i's typing $S'(\Gamma) \vdash e''_{ti} : \tau''_i$ where $S'(\Gamma) = \Sigma'_i(\Gamma_i)$, we have

$$W(\Gamma_i, e_i) = e'_{ti} : \tau'_i,\ S_i$$

for some τ'_i and there is a substitution S''_i such that

$$S''_i(\tau'_i) = \tau''_i, \ \ \forall \nu \notin \mathit{fresh}(W(\Gamma_i, e_i)),\ \Sigma'_i(\nu) = S''_i \circ S_i(\nu)$$

Because $\sigma_i = Gen(S'(\Gamma), e''_{ti} : \tau''_i)$ and $FV(\sigma_i) = \emptyset$ from the typing, $\sigma_i = \forall FV(\tau''_i).\tau''_i$. If we let $\tau_i = inst(\sigma_i)$ and $\tau_i \leq_{I_i} \sigma_i$, $I_i(\tau''_i) = \tau_i$. $I_i^{-1} = \{(\nu' \Rightarrow \nu) \mid (\nu \Rightarrow \nu') \in I_i\}$ is well-defined from the definition of *inst*. Since $(I_i \circ S''_i)(\tau'_i) = I_i(\tau''_i) = \tau_i$, τ'_i and τ_i are unifiable and

$$S'_i = unify(\tau_i, \tau'_i) = I_i \circ S''_i$$

We now check the alpha-equivalence of σ_i and σ'_i. Because $FV(\Gamma_i)$ and $fresh(W(\Gamma_i, e_i))$ are distinct,

$$\Sigma_i(\Gamma_i) = (I_i \circ S''_i \circ S_i)(\Gamma_i) = I_i \circ \Sigma'_i(\Gamma_i) = I_i \circ S'(\Gamma)$$

We generalize $e'_{ti} : \tau_i$. From the generalization of $e''_{ti} : \tau''_i$ in the typing,

$$FV(\tau''_i) \cap FV(S'(\Gamma)) = FV(\tau''_i) \cap FV(\Sigma'_i(\Gamma_i)) = \emptyset$$

Applying the one-to-one mapping I_i to the equation, we obtain $FV(\tau_i) \cap FV((I_i \circ \Sigma'_i)(\Gamma_i)) = \emptyset$ which is equal to

$$FV(\tau_i) \cap FV(\Sigma_i(\Gamma_i)) = \emptyset$$

and

$$Gen(\Sigma_i(\Gamma_i), e'_{ti} : \tau_i) = \forall FV(\tau_i).\tau_i \stackrel{\alpha}{=} \sigma_i$$

If we let $\Sigma'_{i+1} = I_i^{-1}$, $\Sigma'_{i+1}(\Sigma_i(\Gamma_i)) = S'(\Gamma)$.

In the base case of this induction when $i = 1$, $\Gamma_1 = \Gamma$, therefore the existence of $\Sigma'_1 = S'$ is clear. Using the induction, the completeness of all the reconstruction of the type cases is also proven.

We move to reconstruction completeness proof of the body e_0. From the previous property, there exists Σ'_n such that $\Sigma'_n(\Sigma_n(\Gamma_n)) = S'(\Gamma)$. The typing of e_0, $S'(\Gamma)\{\mathtt{x} : \sigma\} \vdash e'_{t0} : \tau'$ is rewritten to $\Sigma'_n(\Sigma_n(\Gamma_n)\{\mathtt{x} : \sigma\}) \vdash e'_{t0} : \tau'$, since σ is closed. We use the induction hypothesis for this rewritten typing and obtain

$$e_{t0} : \tau, S = W(\Sigma_n(\Gamma_n)\{\mathtt{x} : \sigma\}, e_0)$$

and there exists a substitution Σ'' such that

$$\Sigma''(\tau) = \tau' \quad \forall \nu \notin fresh(W(\Sigma_n(\Gamma_n)\{\mathtt{x} : \sigma\}, e_0)),\ \Sigma'_n(\nu) = (\Sigma'' \circ S)(\nu)$$

Let ν is a type variable which is not one of the fresh variables introduced in this entire reconstruction $W(\Gamma, e)$, then $(\Sigma_n \circ \ldots \circ \Sigma_0)(\nu)$ contains none of $fresh(W(\Sigma_n(\Gamma_n)\{\mathtt{x} : \sigma\}, e_0)$, therefore we can rewrite the above equation into the following:

$$\forall \nu \notin fresh(W(\Gamma, e)),\ \Sigma'_n \circ \Sigma_n \circ \ldots \circ \Sigma_0(\nu) = (\Sigma'' \circ S \circ \Sigma_n \circ \ldots \circ \Sigma_0)(\nu)$$

and if we let $\Sigma = S \circ \Sigma_n \circ \ldots \circ \Sigma_0$ then the above equation is rewritten to

$$\forall \nu \notin \mathit{fresh}(W(\Gamma, e)),\ S'(\nu) = (\Sigma'' \circ \Sigma)(\nu)$$

Recursive generic bindings Similar to the case of the normal generic bindings.

□

7.5 Semantics and type soundness

7.5.1 Semantics of `generic` expressions

Semantic values

Similarly to the generic primitives and their derived generic values, the semantics of user-defined generic values also depends on their evaluation type context. When a user-defined generic value definition is evaluated, its definition is stored in the current evaluation environment as a form of closure like the evaluation of derived generics definitions. Type cases are stored inside a closure as they are, and they will be evaluated when a type context is given. We extend the semantic values so that they can contain the representation of the closures for user-defined generic values – *case closures* $gcase(E, *, [tc_{ti}])$:

$$\begin{array}{lcll}
r & ::= & v & \text{Regular values} \\
 & | & err & \text{Error} \\
 & | & gerr & \text{Generic run time error} \\
v & ::= & b_\sigma & \text{Basic constants} \\
 & | & v, v & \text{Tuples} \\
 & | & clos(E, _, \mathtt{x}, e_t) & \text{Functional closures} \\
 & | & clos(E, \mathtt{f}, \mathtt{x}, e_t) & \text{Functional closures (recursive)} \\
sv & ::= & v & \text{Regular values} \\
 & | & gclos(E, _, V, e_t) & \text{Generic closures} \\
 & | & gclos(E, \mathtt{x}, V, e_t) & \text{Generic closures (recursive)} \\
 & | & gcase(E, _, [tc_t, \ldots, tc_t]) & \text{Case closures} \\
 & | & gcase(E, \mathtt{x}, [tc_t, \ldots, tc_t]) & \text{Recursive case closures} \\
E & ::= & \{\mathtt{x} = sv, \ \ldots\ \mathtt{x} = sv\} & \text{Environment}
\end{array}$$

$$S(gcase(E, *, [tc_{t1}, \ldots, tc_{tn}])) = gcase(S(E), *, [S(tc_{t1}), \ldots, S(tc_{tn})])$$

A case closure contains the evaluation environment E where it is created and type case tc_{ti}. A recursive user-defined generic value is stored as a recursive case closure which contains the identifier of the recursive generic value. As generic closures for derived generics, the generic case closures just appear in evaluation environments. We also introduce a new result *gerr* for the *generic run time errors*. If a generic case closure is evaluated in a type context which does not match with any of the type patterns stored in the closure, the evaluation fails and returns *gerr*. This error stops the entire evaluation like the normal error *err*. Since at this moment we cannot statically reject programs which produce *gerr*, in the implementation, *gerr* is reported as a run time exception and can be caught by some appropriate exception handling.

Evaluation rules

Evaluation of generic constructions are separated into two parts: storage of case closures into evaluation environments and evaluation of these closures in evaluation type contexts.

$$\textsf{(e-generic)}\quad \frac{E\{\mathtt{x} = gcase(E, \_, [tc_{t1}, \ldots, tc_{tn}])\} \models e_t \Longrightarrow r}{E \models \mathtt{generic}\ \mathtt{x}_\sigma = \mathtt{case}\ tc_{t1} \mid \ldots \mid tc_{tn}\ \mathtt{in}\ e_t \Longrightarrow r}$$

$$\textsf{(e-genericrec)}\quad \frac{E\{\mathtt{x} = gcase(E, \mathtt{x}, [tc_{t1}, \ldots, tc_{tn}])\} \models e_t \Longrightarrow r}{E \models \mathtt{generic\ rec}\ \mathtt{x}_\sigma = \mathtt{case}\ tc_{t1} \mid \ldots \mid tc_{tn}\ \mathtt{in}\ e_t \Longrightarrow r}$$

(e-generic$*$) rules store user-defined generic value definitions into environments as the form of generic case closures.

$$\textsf{(e-gcase)}\quad \frac{\begin{array}{c}\tau \not\leq \sigma_1 \quad \ldots \quad \tau \not\leq \sigma_{p-1} \quad \tau \leq_S \sigma_p \\ E' \models S(e_{tp}) \Longrightarrow r\end{array}}{E\{\mathtt{x} = gcase(E', \_, [\sigma_1 \Rightarrow e_{t1}, \ldots\ \sigma_p \Rightarrow e_{tp}, \ldots, \sigma_n \Rightarrow e_{tn}])\} \models \mathtt{x} : \tau \Longrightarrow r}$$

$$\textsf{(e-gcase-r)}\quad \frac{\begin{array}{c}\tau \not\leq \sigma_1 \quad \ldots \quad \tau \not\leq \sigma_{p-1} \quad \tau \leq_S \sigma_p \\ E'[\mathtt{y} = gcase(E', \mathtt{y}, [\sigma_1 \Rightarrow e_{t1}, \ldots\ \sigma_p \Rightarrow e_{tp}, \ldots, \sigma_n \Rightarrow e_{tn}])] \models S(e_{tp}) \Longrightarrow r\end{array}}{E\{\mathtt{x} = gcase(E', \mathtt{y}, [\sigma_1 \Rightarrow e_{t1}, \ldots\ \sigma_p \Rightarrow e_{tp}, \ldots, \sigma_n \Rightarrow e_{tn}])\} \models \mathtt{x} : \tau \Longrightarrow r}$$

$$\textsf{(e-gcase-err)}\quad \frac{1 \leq i \leq n, \quad \tau \not\leq \sigma_i}{E\{\mathtt{x} = gcase(E', *, [\sigma_1 \Rightarrow e_{t1}, \ldots, \sigma_n \Rightarrow e_{tn}])\} \models \mathtt{x} : \tau \Longrightarrow gerr}$$

When a user-defined generic value $\mathtt{x}$ which is stored as a generic case closure in an environment is evaluated in some evaluation type context $\mathtt{x} : \tau$, the type patterns are examined to detect whether τ matches one of them, that is, whether τ can be an instance of a type pattern or not. If there are one or more, the first match σ_p is selected and its type method e_{tp} is evaluated with the instantiation S where $\tau \leq_S \sigma_p$ which provides the correct evaluation type contexts of the type method. Like (e-gclos), we need not to apply the instantiation S to the evaluation environment, since the domain of S is the quantified type variables in the type patterns which never appear freely in the environment E' for well-typed programs. If there is no match, (e-gcase-err) is used to return the generic run time error $gerr$.

7.5.2 Type soundness

Here we prove the type soundness of our extended type system for generic bindings. The proof outline is similar to the original proof. We create the corresponding typing system for the semantic values and prove soundness for each evaluation derivation in the conjunction of the type systems and the evaluation rules.

Extended semantic typing

To prove the type soundness, we first need to extend the semantic typing we defined in Section 2.5.5. We add the rules for the generic case closures (s-gcase) and (s-gcase-r), which correspond with (t-generic$*$) typing rules:

$$\textbf{(s-gcase)}\ \frac{\begin{array}{c}\exists\Gamma, E \in \Gamma \\ \Gamma \vdash e_{ti} : \tau_i \quad \sigma_i = Gen(\Gamma, e_{ti} : \tau_i) \quad FV(\sigma_i) = \emptyset \quad BV(\sigma_i) \subset \mathbb{D} \\ \sigma = p\_antiunif(\{\sigma_i\})\end{array}}{gcase(E, \_, [\sigma_1 \Rightarrow e_{t1}, \ldots, \sigma_n \Rightarrow e_{tn}]) \in \sigma}$$

$$\textbf{(s-gcase-r)}\ \frac{\begin{array}{c}\exists\Gamma, E \in \Gamma \\ \Gamma\{\mathtt{x} : \sigma\} \vdash e_{ti} : \tau_i \quad \sigma_i = Gen(\Gamma, e_{ti} : \tau_i) \quad FV(\sigma_i) = \emptyset \quad BV(\sigma_i) \subset \mathbb{D} \\ \sigma = p\_antiunif(\{\sigma_i\})\end{array}}{gcase(E, \mathtt{x}, [\sigma_1 \Rightarrow e_{t1}, \ldots, \sigma_n \Rightarrow e_{tn}]) \in \sigma}$$

Lemma 8 is also extended:

Lemma 17 (Extended substitution lemma of semantic typing) *If a semantic value sv has a semantic typing $sv \in \sigma$, then $S(sv) \in S(\sigma)$ for any substitution S. If we have an evaluation environment E such that $E \in \Gamma$ for some typing environment Γ, for any substitution S, we have $S(E) \in S(\Gamma)$.*

PROOF: By induction on the typing derivation of sv. The cases except (s-gcase$*$) are proven in the same way as Lemma 8. What we prove here is the cases for (s-gcase$*$).

(s-gcase) Suppose that a semantic typing we have is the following:

$$\frac{\begin{array}{c}\exists\Gamma,\ E \in \Gamma \\ \Gamma \vdash e_{ti} : \tau_i \quad \sigma_i = Gen(\Gamma, e_{ti} : \tau_i) \quad FV(\sigma_i) = \emptyset \quad BV(\sigma_i) \in \mathbb{D} \\ \sigma = p\_antiunif(\{\sigma_i\})\end{array}}{gcase(E, *, [\sigma_1 \Rightarrow e_{t1}, \ldots, \sigma_n \Rightarrow e_{tn}]) \in \sigma}$$

Like the proof of Theorem 20, we prepare a substitution S' which renames all the bound type variables of σ_i to fresh variables. Because $\sigma_i = \forall FV(\tau_i).\tau_i$, if we let $V = \bigcup_{i=1}^{n} FV(\tau_i)$ then

$$S' = \{(\nu_j \Rightarrow \nu'_j) \mid \nu_j \in V,\ \nu'_j \text{ is fresh}\}$$

and let $V' = \{\nu'_j\}$. Then $S'(\Gamma) = \Gamma$ because $Dom(S') \cap FV(\Gamma) = \emptyset$.

By Theorem 20, we have $(S \circ S')(\Gamma) \vdash (S \circ S')(e_{ti}) : (S \circ S')(\tau_i)$ but $(S \circ S')(\Gamma) = S(\Gamma)$ (see the proof for the let bindings of Theorem 3.) Therefore:

$$S(\Gamma) \vdash (S \circ S')(e_{ti}) : (S \circ S')(\tau_i)$$

The generalization of this $(S \circ S')(\tau_i)$ is $Gen(\Gamma, (S \circ S')(e_{ti}) : (S \circ S')(\tau_i))$ and it is equal to $\forall V'.(S \circ S')(\tau)$ by the same discussion of Theorem 3. If we let $\sigma_i = \forall V'.(S \circ S')(\tau)$, $\sigma'_i \stackrel{\alpha}{=} \sigma_i$. Because σ'_i is closed, the following type cases are α-equivalent to the original type cases:

$$(\sigma'_i \Rightarrow (S \circ S')(e_{ti} : \tau)) = (\forall V'.(S \circ S')(\tau) \Rightarrow (S \circ S')(e_{ti} : \tau)) \stackrel{\alpha}{=} (\sigma_i \Rightarrow e_{ti} : \tau)$$

Because we can suppose that $\sigma = p\_antiunif(\{\sigma_i\})$ without losing any generality, from the induction hypothesis with S for $E \in \Gamma$ and (s-gcase), we have

$$\frac{\begin{array}{c} S(E) \in S(\Gamma), S(\Gamma) \vdash (S \circ S')(e_{ti}) \\ \sigma'_i = Gen(\Gamma, (S \circ S')(e_{ti})) \quad FV(\sigma'_i) = \emptyset \quad BV(\sigma'_i) \subset \mathbb{D} \\ \sigma = p\_antiunif(\{\sigma_i\}) \end{array}}{gcase(S(E), *, [\sigma'_1 \Rightarrow (S \circ S')(e_{t1}), \ldots, \sigma'_n \Rightarrow (S \circ S')(e_{tn})]) \in \sigma}$$

The generic case closure in the conclusion is α-equivalent to

$$S(gcase(E, *, [\sigma_1 \Rightarrow e_{t1}, \ldots, \sigma_n \Rightarrow e_{tn}]))$$

and $S(\sigma) = \sigma$ because σ is closed.

(s-gcase-r) Provable almost same as (s-gcase).

□

Lemma 18 (Extended generality of semantic typing) *If a value v has a semantic typing $v \in \tau$, $v \in \forall V.\tau$ where $V = FV(\tau)$.*

PROOF: Similar to its original version, clear from Lemma 17 and (s-schm) rule. □

Incomplete type soundness

As we have explained, with non-exhaustive generic bindings, even well-typed programs may return generic run time errors $gerr$. Therefore in the current typing system, the type soundness which we can prove in this stage is somewhat incomplete; the well-typed programs never return the error err, but values or $gerr$, if their evaluation terminate.

Theorem 23 (Incomplete type soundness) *If $\Gamma \vdash e_t : \tau$ for some Γ and there exists E such that $E \in \Gamma$ and $E \models e_t : \tau \Longrightarrow r$, then r is not err, but a value v such that $v \in \tau$ or the generic run time error $gerr$.*

PROOF: Similar to Theorem 5, by induction over the size of the evaluation derivation $E \models e_t \Longrightarrow r$. We just prove the additional cases.

Variables For variables we have new additional applicable rules (e-gcase$*$) and (e-gcase-err). If $e_t : \tau = \mathtt{x}_\sigma : \tau$, these rules are called if the variable $\mathtt{x}$ is bound with a generic case closure $gcase(E', *, [tc_{t1}, \ldots, tc_{tn}])$ whose type is σ, because $E \in \Gamma$.

Suppose the case closure is non recursive, $gcase(E', \_, [tc_{t1}, \ldots, tc_{tn}])$. If there is no matched type pattern with τ, (e-gcase-err) is applied and returns $gerr$. Otherwise, if there are one or more matches and the first match is $tc_{tp} = (\sigma_p \Rightarrow e_{tp})$, there exists a substitution S such that $\tau \leq_S \sigma_p$ and we can apply (e-gcase).

$$\frac{\begin{array}{c}\tau \not\leq \sigma_1 \quad \ldots \quad \tau \not\leq \sigma_{p-1} \quad \tau \leq_S \sigma_p \\ E' \models S(e_{tp}) \Longrightarrow r\end{array}}{E\{\mathtt{x} = gcase(E', \_, [\sigma_1 \Rightarrow e_{t1}, \ldots \sigma_p \Rightarrow e_{tp}, \ldots, \sigma_n \Rightarrow e_{tn}])\} \models (\mathtt{x} : \tau) \Longrightarrow r}$$

Because $gclos(E', *, [tc_{t1}, \ldots, tc_{tn}]) \in \sigma$, by (s-gcase) there is a typing environment Γ' and

$$E' \in \Gamma', \quad \Gamma' \vdash e_{tp} : \tau_p, \quad \sigma_p = Gen(\Gamma', e_{tp} : \tau_p), \quad FV(\sigma_p) = \emptyset, \quad BV(\sigma_p) \in \mathbb{D}$$

Since $Dom(S) = BV(\sigma_p) = FV(\tau_p) \setminus FV(\Gamma')$, $S(\Gamma') = \Gamma'$ and $S(E') = E'$, therefore $E' = S(E') \in S(\Gamma') = \Gamma'$. From Theorem 3, $S(\Gamma') \vdash S(e_{tp} : \tau_p)$. Now we can use the induction hypothesis and obtain:

$$E' \models S(e_{tp} : \tau_p) \Longrightarrow r$$

where r is not err.

The case for the recursive generic case closures is also proven like this way.

Generic bindings We have

$$\frac{\begin{array}{c}\Gamma \vdash e_{ti} : \tau_i \quad \sigma_i = Gen(\Gamma, e_{ti} : \tau_i) \quad FV(\sigma_i) = \emptyset \quad BV(\sigma_i) \subset \mathbb{D} \\ \sigma = p\_antiunif(\{\sigma_i\}) \quad \Gamma\{\mathtt{x} : \sigma\} \vdash e_t : \tau\end{array}}{\Gamma \vdash \mathtt{generic}\ \mathtt{x}_\sigma = \mathtt{case}\ \sigma_1 \Rightarrow e_{t1} \mid \ldots \mid \sigma_n \Rightarrow e_{tn}\ \mathtt{in}\ e_t : \tau}$$

The possible evaluation rule is (e-generic):

$$\frac{E\{\mathtt{x} = gcase(E, \_, [tc_{t1}, \ldots, tc_{tn}])\} \models e_t \Longrightarrow r}{E \models \mathtt{generic}\ \mathtt{x}_\sigma = \mathtt{case}\ tc_{t1} \mid \ldots \mid tc_{tn}\ \mathtt{in}\ e_t \Longrightarrow r}$$

From (s-gcase), $gcase(E, \_, [tc_{t1}, \ldots, tc_{tn}]) \in \sigma$ and therefore

$$E\{\mathtt{x} = gcase(E, \_, [tc_{t1}, \ldots, tc_{tn}])\} \in \Gamma\{\mathtt{x} : \sigma\}$$

This permits us to use the induction hypothesis for the evaluation of e_t. It must return a value $v \in \tau$. And it is our goal.

Recursive generic bindings Similar to as the non recursive generic bindings.

$\square$

7.6 Compilation

7.6.1 Type pattern matching

The semantics of user-defined generic values are based on the matching of types. So does their compilation. In the compilation of generic bindings, generic type cases are translated into *type pattern matching cases*, which try to match a dispatched run time type with type patterns. The checking of $\tau \leq \sigma$ will be translated into this type pattern matching in the intermediate language IML using a pattern match expression. First, we add the match expressions to the IML syntax:

e	$::=$	$\ldots$	Original IML
		`match` e `with` $mc \mid \ldots \mid mc$	Match expression
mc	$::=$	$p \rightarrow e$	Match cases
p	$::=$	b_σ	Basic constants
	$\mid$	`x`	Pattern variables
	$\mid$	p, p	Tuple patterns
	$\mid$	$RT(p, \ldots, p)$	Pattern constructors
RT	$::=$	`RTvar` $\mid$ `RTarrow` $\mid$ `RTtuple` $\mid$ `RTconstr`	Run time type constructors

Our match expression is much simpler than the normal ML pattern match expression, because ours just handles run time types and therefore the patterns consist of only basic constants, pattern variables, tuples and patterns formed by the run time type constructors. Pattern matching between a semantic value and a pattern is defined formally as follows.

Definition 14 *Let v be a semantic value, p be a pattern and S be a substitution from pattern variables to semantic values. When $S(p) = v$ holds, we say v* matches with *p* under *S and write $v \leq_S p$.*

For example, a value «$\delta_1 \rightarrow$ `int`» = `RTarrow(RTvar 1, RTbase "int")` matches with a pattern `RTarrow(RTvar x, y)` under a substitution $S = \{\mathtt{x} \Rightarrow 1,\ \mathtt{y} \Rightarrow (\mathtt{RTbase}\ \text{"}\mathtt{int}\text{"})\}$.

The semantic rules of IML are as follows for this run time type matching:

$$\textbf{(i-match)}\ \frac{\mathcal{E} \models_i e \Longrightarrow v \quad 1 \leq i < q,\ v \not\leq p_i, \quad v \leq_S p_q \quad \mathcal{E}\{S\} \models_i e_q \Longrightarrow r}{\mathcal{E} \models_i \texttt{match}\ e\ \texttt{with}\ p_1\ \rightarrow\ e_1 \mid \ldots \mid p_n\ \rightarrow\ e_n \Longrightarrow r}$$

$$\textbf{(i-match-err)}\ \frac{\mathcal{E} \models_i e \Longrightarrow v \quad 1 \leq i \leq n,\ v \not\leq p_i}{\mathcal{E} \models_i \texttt{match}\ e\ \texttt{with}\ p_1\ \rightarrow\ e_1 \mid \ldots \mid p_n\ \rightarrow\ e_n \Longrightarrow \mathit{gerr}}$$

If the target expression for matching e is evaluated to a semantic value v, there is a pattern p_q which v matches with under a substitution S, and it is the first matched pattern, then its corresponding case expression e_q is selected and evaluated under the updated evaluation environment $\mathcal{E}\{S\}$, an environment $\mathcal{E}$ overridden by S. If there is no matching pattern for v the evaluation of the run time type matching fails and returns the error gerr.

We do not argue the compilation of pattern matching itself to more primitive codes, because it has been well studied and efficient compilation methods such as [Jon97] already exist.

7.6.2 Compilation to type pattern matching

In the compilation of generic bindings to IML value definitions using type pattern matchings, each type pattern is translated into a match pattern and each type case is to a match case. Finally a whole generic bindings will be compiled into a function definition which takes run time type arguments and matches them with its match cases and evaluates the first matched case expression.

We start from a simple compilation algorithm which translates whole part of a type pattern to a match pattern. This is easy to understand, but actually inefficient, since dispatched run time types may be duplicated and matched with the same sub-patterns repeatedly. Later in Section 7.6.3, we introduce an optimized algorithm in which such duplication does not occur.

Compilation of type patterns

The compilation algorithm of type patterns is similar to the type compilation algorithm $\lfloor\cdot\rfloor_V$ defined in Section 3.4, except that now the dynamic type variables are compiled not to normal variables, but pattern variables. Since the type patterns are not type expressions but type schemes, we need to extend $\lfloor\cdot\rfloor_V$ for type schemes:

$$\begin{array}{rcl}
\lfloor \mathit{base} \rfloor_V & = & \text{«}\mathit{base}\text{»} \\
\lfloor \delta_i \rfloor_V & = & \texttt{d}_i \quad \text{if } \delta_i \in V \\
& & \text{Otherwise, «}\delta_i\text{»} \\
\lfloor \tau_1 \rightarrow \tau_2 \rfloor_V & = & \texttt{RTarrow(}\lfloor\tau_1\rfloor_V\texttt{,}\lfloor\tau_2\rfloor_V\texttt{)} \\
\lfloor \tau_1 * \tau_2 \rfloor_V & = & \texttt{RTtuple(}\lfloor\tau_1\rfloor_V\texttt{,}\lfloor\tau_2\rfloor_V\texttt{)} \\
\lfloor \forall V'.\tau \rfloor_V & = & \lfloor \tau \rfloor_{V+V'}
\end{array}$$

In the compilation of type schemes $\lfloor \forall V'.\tau \rfloor_V$, we assume that V and V' are distinct to avoid giving the same pattern variable for different generalized type variables. In the case V and V' are not distinct, we can α-convert the type case and make them distinct. Since the type patterns σ are always closed type schemes, actually $\lfloor \sigma \rfloor_V = \lfloor \sigma \rfloor_\emptyset$ for any V, because $FV(\sigma) \cap V = \emptyset$.

We here prove the correctness of the type pattern compilation:

Lemma 19 *Let τ be a type expression and $\sigma = \forall V.\tau'$ be a type pattern where $FV(\tau') = V$. If and only if τ is an instance of σ under some substitution S, i.e., $\tau \leq_S \sigma$, then the semantic value of τ matches with the compiled pattern for σ, $«\tau» \leq_\Sigma \lfloor \sigma \rfloor_\emptyset$, where $\Sigma = \{\mathtt{d}_i \Rightarrow «S(\delta_i)» \mid \delta_i \in V\}$.*

PROOF: By induction over the type pattern σ.

Base type pattern Trivial.

Pattern variable Let $\sigma = \forall \delta_i.\delta_i$. Then for any type expression τ, we have $\tau \leq_S \forall \delta_i.\delta_i$ under $S = \{\delta_i \Rightarrow \tau\}$. The compilation of the pattern is $\lfloor \sigma \rfloor_\emptyset = \lfloor \delta_i \rfloor_{\{\delta_i\}} = \mathtt{d}_i$. If we let $\Sigma = \{\lfloor \delta_i \rfloor_{\{\delta_i\}} \Rightarrow «S(\delta_i)»\} = \{\mathtt{d}_i \Rightarrow «\tau»\}$, under this Σ, $«\tau» \leq \lfloor \sigma \rfloor_\emptyset$.

Arrow type pattern Let $\sigma = \forall V.\tau_1 \to \tau_2$. Then its type instance τ must be written in $\tau_1' \to \tau_2'$, where $\tau_1' \leq_S \forall V_1.\tau_1$ and $\tau_2' \leq_S \forall V_2.\tau_2$ where $V_i = FV(\tau_i)$.

By applying the induction hypothesis to these sub type patterns, we get:

$$«\tau_k'» \leq_{\Sigma_k} \lfloor \forall V_k.\tau_k \rfloor_\emptyset \text{ where } \Sigma_k = \{\mathtt{d}_i \Rightarrow «S(\delta_i)» \mid \delta_i \in V\}$$

For any $\delta_i \in V_1 \cap V_2$, $\Sigma_1(\mathtt{d}_i) = \Sigma_2(\mathtt{d}_i) = «S(\delta_i)»$. Therefore $\Sigma_1\{\Sigma_2\} = \Sigma_2\{\Sigma_1\}$. If we let $\Sigma = \Sigma_1\{\Sigma_2\}$ then $«\tau_k'» \leq_\Sigma \lfloor \forall V_k.\tau_k \rfloor_\emptyset$ both for $k = 1, 2$. Because $\Sigma(\lfloor \forall V_k.\tau_k' \rfloor_\emptyset) = «\tau_k'»$,

$$\begin{aligned} \Sigma(\lfloor \forall V.\tau_1 \to \tau_2 \rfloor_\emptyset) &= \mathtt{RTarrow}(\Sigma(\lfloor \forall V_1.\tau_1' \rfloor_\emptyset), \Sigma(\lfloor \forall V_2.\tau_2' \rfloor_\emptyset)) \\ &= \mathtt{RTarrow}(«\tau_1'», «\tau_2'»)) \\ &= «\tau_1' \to \tau_2'» \end{aligned}$$

Therefore $«\tau_1' \to \tau_2'» \leq_\Sigma \lfloor \forall V.\tau_1 \to \tau_2 \rfloor_\emptyset$.

If τ is not an arrow type, $«\tau»$ cannot be written in the form of $\mathtt{RTarrow}(\mathtt{v}_1, \mathtt{v}_2)$, therefore it cannot match with $\lfloor \sigma \rfloor_\emptyset = \mathtt{RTarrow}(\lfloor \tau_1 \rfloor_\emptyset, \lfloor \tau_2 \rfloor_\emptyset)$.

Tuple type pattern Similar to the arrow patterns.

□

Compilation of type cases and generic bindings

Now we define the whole compilation of generic bindings using the type pattern compilation:

$$\begin{array}{l}
\lfloor \texttt{generic (rec) } \texttt{x}_\sigma = \texttt{case } \sigma_1 \Rightarrow e_{t1} \mid \ldots \mid \sigma_n \Rightarrow e_{tn} \texttt{ in } e_t \rfloor_V = \\
\qquad \text{let } \forall \delta_1 \ldots \delta_m . \tau = \sigma \text{ where } [\delta_i] = \mathit{Vlist}(\{\delta_i\}, \tau) \text{ in} \\
\qquad \texttt{let (rec) x = fun d}_1 \ \ldots\ \texttt{d}_m \ \rightarrow \\
\qquad\ \ \texttt{match } \lfloor \tau \rfloor_{\{\delta_i\}} \texttt{ with } \lfloor \sigma_1 \Rightarrow e_{t1} \rfloor_V \mid \ldots \mid \lfloor \sigma_n \Rightarrow e_{tn} \rfloor_V \\
\qquad \texttt{in } \lfloor e_t \rfloor_V
\end{array}$$

$$\lfloor (\sigma \Rightarrow e_t) \rfloor_V = \lfloor \sigma \rfloor_\emptyset \ \rightarrow\ \lfloor e_t \rfloor_{V + BV(\sigma)}$$

A generic binding is compiled into a function which takes run time types which indicate how the bound variables δ_i in the type scheme σ are instantiated in the evaluation. From them we rebuild the run time type representation of its evaluation type context $\lfloor \tau \rfloor_{\{\delta_i\}}$ and matches it with each compiled type pattern.

A type case $(\sigma \Rightarrow e_t)$ is compiled into a match case. The bound variables in the type pattern σ are added to the generalized type variables in the compilation of e_t, so that the run time types bound to the compiled pattern variables $\texttt{d}_i$ of δ_i can be delivered correctly.

Example

The following generic binding `add` which adds the elements of two data structures of integers and tuples,

```
generic rec add∀δ1. δ1 → δ1 → δ1 = case
| int → int → int ⇒ (+)
| (δ2 * δ3) → (δ2 * δ3) → δ2 * δ3 ⇒
    fun x y → add (fst x) (snd x), add (snd x) (snd y)
in
add (1,(2,3)) (4,(5,6));;
```

is compiled into the following IML code:

```
let rec add = fun d1 →
  match RTarrow(d1,RTarrow(d1,d1)) with
  | RTarrow(RTbase "int", RTarrow(RTbase "int", RTbase "int")) →
      (+)
  | RTarrow(RTtuple(d2,d3),RTarrow(RTtuple(d2,d3),RTtuple(d2,d3))) →
      (fun x y → add d2 (fst x) (fst y), add d3 (snd x) (snd y))
in
  add «int * (int * int)» (1,(2,3)) (4,(5,6));;
```

and evaluated to `(5,(7,9))` correctly. □

Non linear patterns in compilation

Note that the compilation of generic bindings may produce non linear patterns like the previous example of `add`: in the compilation of the second type pattern, the pattern variables of $\mathtt{d}_2$ and $\mathtt{d}_3$ occur more than once. Usually such non linear patterns are not permitted in ML pattern matching, since pattern matching is primarily intended to match only the form of values, not their internal equality. But the non linear patterns can be easily encoded to patterns with *pattern guards* in our case. The compiled pattern of `add` will be encoded to the following pattern with guards for instance:

```
| RTarrow(RTtuple(d21,d31),RTarrow(RTtuple(d22,d32),RTtuple(d23,d33)))
    when d21 = d22 && d22 = d23 && d31 = d32 && d32 = d33 →
      ...
```

7.6.3 Optimized compilation of type patterns

The previous compilation algorithm of generic bindings is simple and easy to understand, however, it has some inefficiency and can be optimized. The algorithm reconstructs the type context where a generic instance is evaluated from the run time type arguments which inform instantiation of each quantified dynamic type variable, then destructs it again when we try to match with patterns. This inefficiency is clear in the compilation of `add`. What we are interested in is only how `add`'s generalized dynamic type variables are instantiated, but the previous compilation tripled this information and create a value $\mathtt{RTarrow(d_1,RTarrow(d_1,d_1))}$.

We can optimize this compilation algorithm and obtain more efficient code by making the pattern matching not with the whole type patterns but just the part of generalized type variables. Now the compilations of type cases $\lfloor tc \rfloor_{V,\sigma}$ depend not only on the generalized type variable set V but also on the type scheme σ of the generic binding. The type patterns σ' are compiled by the following algorithm $\lfloor \sigma' \rfloor_\sigma$.

$$\begin{array}{rcl} \lfloor \forall V'.\tau' \rfloor_{\forall V.\tau} & = & \text{let } S(\tau) = \tau' \\ & & \text{let } [\delta_1; \ldots; \delta_m] = \mathit{Vlist}(V, \tau) \\ & & \text{let } \sigma_i = \forall FV(S(\delta_i)).S(\delta_i) \text{ in} \\ & & \lfloor \sigma_1 \rfloor_\emptyset, \ldots, \lfloor \sigma_m \rfloor_\emptyset \end{array}$$

In this algorithm we assume that both the type schemes $\forall V'.\tau'$ and $\forall V.\tau$ are closed and $V \subset \mathbb{D}$. It is clearly always true for any well-typed generic binding. We extract S, the information how the generalized dynamic type variables in V are instantiated in τ'. According to S, we compile the sub type patterns $\sigma_i = \forall FV(S(\delta_i)).S(\delta_i)$ and connects them as a tuple. The new compilations of type cases and generic bindings are defined below:

$$\lfloor \sigma' \Rightarrow e_t \rfloor_{V,\sigma} \quad = \quad \lfloor \sigma' \rfloor_\sigma \rightarrow \lfloor e_t \rfloor_{V+BV(\sigma')}$$

$\lfloor$`generic (rec)` $\mathtt{x}_\sigma$ `= case` $\sigma_1 \Rightarrow e_{t1} \mid \ldots \mid \sigma_n \Rightarrow e_{tn}$ `in` $e_t\rfloor_V =$
  let $\forall\delta_1 \ldots \delta_m.\tau = \sigma$ where $[\delta_i] = \mathit{Vlist}(\{\delta_i\}, \tau)$
  `let (rec) x = fun` $\mathtt{d}_1 \ \ldots \ \mathtt{d}_m \ \rightarrow$
    `match` $\mathtt{d}_1, \ldots, \mathtt{d}_m$ `with`
    $\mid \lfloor\sigma_1 \Rightarrow e_{t1}\rfloor_{V,\sigma}$
    $\mid \ldots$
    $\mid \lfloor\sigma_n \Rightarrow e_{tn}\rfloor_{V,\sigma}$
  `in` $\lfloor e_t\rfloor_V$

The pattern matching is done simply with $\mathtt{d}_1, \ldots, \mathtt{d}_n$, a tuple of variables, and there is no more reconstruction of new type scheme instances.

With this new scheme, the `add` example is compiled into the following more simple IML code:

```
let rec add = fun d1 →
  match d1 with
  | RTbase "int" →
      (+)
  | RTtuple(d2,d3) →
      (fun x y → add d2 (fst x) (fst y), add d3 (snd x) (snd y))
in
  add «int*(int*int)» (1,(2,3)) (4,(5,6));;
```

The correctness of this new optimized type matching compilation is proven by Lemma 19, the correctness of the previous scheme, and the following compatibility lemma of both compilation schemes:

Lemma 20 (Compatibility of type pattern compilations) *Let $\sigma_0 = \forall V_0.\tau_0$, a closed type scheme where $[\delta_1; \ldots; \delta_n] = \mathit{Vlist}(V_0, \tau_0)$. Let σ be a type pattern such that $\sigma = \forall FV(\tau').\tau'$ and $S_0(\tau_0) = \tau'$ for some substitution S_0.*

If and only if a type expression τ is $\tau \leq_S \sigma$, then

$$«(S \circ S_0)(\delta_1)», \ldots, «(S \circ S_0)(\delta_n)» \leq_\Sigma \lfloor\sigma\rfloor_{\sigma_0}$$

where $\Sigma = \{\mathtt{d}_i \Rightarrow «(S \circ S_0)(\delta_i)» \mid \delta_i \in BDV(\sigma)\}$.

Proof: From Lemma 19, $«\tau» \leq_\Sigma \lfloor\sigma\rfloor_\emptyset$. By induction over the structure of σ:

Base type patterns Let $\sigma = \mathit{base}$. Because $\mathit{base} \leq_{S_0} \sigma_0$ and $BDV(\sigma_0) \neq \emptyset$, $\sigma_0 = \forall\delta.\delta$ and $S_0 = \{\delta \Rightarrow \mathit{base}\}$. Since $«(S \circ S_0)(\delta)» = «\mathit{base}»$ and $\lfloor \mathit{base}\rfloor_{\forall\delta.\delta} = \lfloor \mathit{base}\rfloor_\emptyset$,

$$«(S \circ S_0)(\delta)» \leq_\Sigma \lfloor \mathit{base}\rfloor_{\forall\delta.\delta}$$

Pattern variables Let $\sigma = \forall\delta.\delta$, then $S(\delta) = \tau$. Since $\delta \leq \sigma_0$ and $BV(\sigma_0) \subset \mathbb{D}$, $\sigma_0 = \forall\delta'.\delta'$ and $S_0 = \{\delta' \Rightarrow \delta\}$. Because $«(S \circ S_0)(\delta')» = «S(\delta)» = «\tau»$ and $\lfloor\forall\delta.\delta\rfloor_{\forall\delta'.\delta'} = \lfloor\forall\delta.\delta\rfloor_\emptyset = \lfloor\sigma\rfloor_\emptyset$, from $«\tau» \leq_\Sigma \lfloor\sigma\rfloor_\emptyset$, we get

$$«(S \circ S_0)(\delta')» \leq_\Sigma \lfloor\forall\delta.\delta\rfloor_{\forall\delta'.\delta'}$$

Arrow type patterns Let $\sigma = \forall V.\tau_1 \to \tau_2$. Then, since $S_0(\tau_0) = \tau_1 \to \tau_2$, τ_0 is either a dynamic type variable δ or an arrow type $\tau_0 = \tau_1' \to \tau_2'$.

In the case $\tau_0 = \delta$, $S_0 = \{\delta \Rightarrow \tau_1 \to \tau_2\}$. Therefore the tuple of run time types is $«(S \circ S_0)(\delta)» = «S(\tau_1 \to \tau_2)» = «\tau»$ and the optimized pattern becomes $\lfloor\forall V.\tau_1 \to \tau_2\rfloor_{\forall\delta.\delta} = \lfloor\forall FV(S_0)(\delta).S_0(\delta)\rfloor_\emptyset = \lfloor\forall V.\tau_1 \to \tau_2\rfloor_\emptyset$. Since $«\tau» \leq_\Sigma \lfloor\forall V.\tau_1 \to \tau_2\rfloor_\emptyset$,

$$«(S \circ S_0)(\delta)» \leq_\Sigma \lfloor\forall V.\tau_1 \to \tau_2\rfloor_{\forall\delta.\delta}$$

In the other case, when $\tau_0 = \tau_1' \to \tau_2'$, $S_0(\tau_1') = \tau_1$ and $S_0(\tau_2') = \tau_2$. If we let $\sigma_1 = \forall FV(\tau_1').\tau_1'$ and $\sigma_2 = \forall FV(\tau_2').\tau_2'$, $S_0(\tau_1') = \tau_1$ and $S_0(\tau_2') = \tau_2$. In addition, because $\tau \leq_S \sigma$, $\tau = S(\tau_1) \to S(\tau_2)$ and $S(\tau_i) \leq_S \forall FV(\tau_i').\tau_i'$. From these we can use the induction hypothesis and obtain for $k = 1$ and 2:

$$\begin{array}{l}[\delta_{k1}; \ldots; \delta_{km_k}] = \mathit{Vlist}(FV(\tau_k'), \tau_k') \\ «(S \circ S_0)(\delta_{k1})», \ldots «(S \circ S_0)(\delta_{km_k})» \leq_{\Sigma_k} \lfloor\forall FV(\tau_k).\tau_k\rfloor_{\sigma_k} \\ \quad \text{where} \quad \Sigma_k \quad = \quad \{\mathtt{d}_i \Rightarrow «(S \circ S_0)(\delta_i)» \mid \delta_i \in BDV(\sigma_k)\}\end{array}$$

By the same discussion in the arrow type patterns part of Lemma 19, if a variable `x` is in both $Dom(\Sigma_1)$ and $Dom(\Sigma_2)$,

$$\Sigma_1(\mathtt{x}) = \Sigma_2(\mathtt{x})$$

Therefore we can merge them without contradiction:

$$\begin{array}{r}«(S \circ S_0)(\delta_{11})», \ldots, «(S \circ S_0)(\delta_{1m_1})», «(S \circ S_0)(\delta_{21})», \ldots, «(S \circ S_0)(\delta_{2m_2})» \\ \leq_{\Sigma_1\{\Sigma_2\}} \lfloor\forall FV(\tau_1).\tau_1\rfloor_{\sigma_1}, \lfloor\forall FV(\tau_2).\tau_2\rfloor_{\sigma_2}\end{array}$$

Since $\Sigma_1\{\Sigma_2\} = \Sigma$ and by eliminating the double occurrence of the tuple members and replacing them, we get our goal.

$\square$

7.6.4 Compilation correctness

Proving the correctness of our compilation algorithm for the user-defined generic values, we need to extend the semantic compilation compatibility $\sqsubset$ defined Section 3.5 for the newly introduced generic case closures:

$$\frac{\begin{array}{c}\mathcal{E} \sqsubset E \\ \sigma = \forall V.\tau = p\_antiunif(\{\sigma_i\}) \quad [\delta_1; \ldots; \delta_n] = Vlist(V, \tau) \\ \exists \Gamma, \quad E \in S(\Gamma) \;\; \Gamma \vdash e_{ti} \;\; \sigma_i = Gen(\Gamma, e_{ti})\end{array}}{\begin{array}{l} clos(\mathcal{E}\{S\}, y, \mathtt{d}_1, \mathtt{fun}\ \mathtt{d}_2 \ldots \mathtt{d}_n \rightarrow \\ \quad \mathtt{match}\ \mathtt{d}_1, \ldots, \mathtt{d}_n\ \mathtt{with}\ \lfloor \sigma_1 \Rightarrow e_{t1} \rfloor_{S,\sigma} \mid \ldots \mid \lfloor \sigma_n \Rightarrow e_{tn} \rfloor_{S,\sigma}) \\ \qquad\qquad \sqsubset gcase(E, y, [\sigma_1 \Rightarrow S(e_{t1}), \ldots, \sigma_n \Rightarrow S(e_{tn})]) \end{array}}$$

With this extension we can prove the correctness for the part of user-defined generic values:

Theorem 24 (Extended compilation correctness) *Let e_t be a fully type annotated GML expression, which has a typing under a typing environment Γ, $\Gamma \vdash e_t$. Let E be a GML evaluation environment such that $E \in S(\Gamma)$ for some substitution S. If $E \models S(e_t) \Longrightarrow v$ for some v then for any IML evaluation environment $\mathcal{E}$ such that $\mathcal{E} \sqsubset E$,*

$$\mathcal{E}\{S\} \models_i \lfloor e_t \rfloor_S \Longrightarrow v'$$

such that $v' \sqsubset v$.

PROOF: The proof is by induction over the evaluation derivation $E \models S(e_t) \Longrightarrow v$. We just prove the newly added cases:

(e-generic) In this case, $e_t = \mathtt{generic}\ \mathtt{x}_\sigma = \mathtt{case}\ \sigma_1 \Rightarrow e_{t1} \mid \ldots \mid \sigma_n \Rightarrow e_{tn}\ \mathtt{in}\ e'_t$.

Typing and evaluation in GML:

The typing of e_t is by (t-generic):

$$\frac{\begin{array}{c}\Gamma \vdash e_{ti} : \tau_i \quad \sigma_i = Gen(\Gamma, e_{ti} : \tau_i) \quad FV(\sigma_i) = \emptyset \quad BV(\sigma_i) \subset \mathbb{D} \\ \sigma = p\_antiunif(\{\sigma_i\}) \quad \Gamma\{\mathtt{x} : \sigma\} \vdash e'_t : \tau\end{array}}{\Gamma \vdash \mathtt{generic}\ \mathtt{x}_\sigma = \mathtt{case}\ \sigma_1 \Rightarrow e_{t1} \mid \ldots \mid \sigma_n \Rightarrow e_{tn}\ \mathtt{in}\ e'_t : \tau}$$

Because σ_i is closed, $S(\sigma_i) = \sigma_i$. Hence the typing of $S(e_t)$ is as follows:

$$\frac{\begin{array}{c}\Gamma \vdash S(e_{ti}) : S(\tau_i) \quad \sigma_i = Gen(\Gamma, S(e_{ti} : \tau_i)) \quad FV(\sigma_i) = \emptyset \quad BV(\sigma_i) \subset \mathbb{D} \\ \sigma = p\_antiunif(\{\sigma_i\}) \quad \Gamma\{\mathtt{x} : \sigma\} \vdash S(e'_t) : S(\tau)\end{array}}{\Gamma \vdash \mathtt{generic}\ \mathtt{x}_\sigma = \mathtt{case}\ \sigma_1 \Rightarrow S(e_{t1}) \mid \ldots \mid \sigma_n \Rightarrow S(e_{tn})\ \mathtt{in}\ S(e'_t) : S(\tau)}$$

If we let $c = gcase(E, \_, [\sigma_1 \Rightarrow S(e_{t1}), \ldots, \sigma_n \Rightarrow S(e_{tn})])$, its is evaluated like:

$$\frac{E\{\mathtt{x} = c\} \models S(e'_t) \Longrightarrow v}{E \models \mathtt{generic}\ \mathtt{x}_\sigma = \mathtt{case}\ \sigma_1 \Rightarrow S(e_{t1}) \mid \ldots \mid \sigma_n \Rightarrow S(e_{tn})\ \mathtt{in}\ S(e'_t) \Longrightarrow v}$$

From (s-gcase), $c \in S(\sigma) = \sigma$.

Induction hypothesis:

Let $\forall V.\tau_0 = \sigma$ and $[\delta_1;\ldots;\delta_m] = \mathit{Vlist}(V,\tau_0)$. If we let

$$\begin{array}{rcl} c' & = & \mathit{clos}(\mathcal{E}\{S\}, \_, \mathtt{d}_1, \mathtt{fun}\ \mathtt{d}_2 \ldots \mathtt{d}_m \rightarrow \\ & & \quad \mathtt{match}\ \mathtt{d}_1, \ldots, \mathtt{d}_m\ \mathtt{with}\ \lfloor \sigma_1 \Rightarrow e_{t1} \rfloor_{S,\sigma} \mid \ldots \mid \lfloor \sigma_n \Rightarrow e_{tn} \rfloor_{S,\sigma}) \end{array}$$

then $c' \sqsubset c$, therefore $\mathcal{E}\{\mathtt{x} = c'\} \sqsubset E\{\mathtt{x} = c\}$. $\Gamma\{\mathtt{x} = \sigma\} \vdash e'_t$ and from $c \in S(\sigma) = \sigma$, $E\{\mathtt{x} = c\} \in S(\Gamma\{\mathtt{x} = \sigma\})$. From these we can use the induction hypothesis for the evaluation of $S(e'_t)$ and obtain:

$$\mathcal{E}\{S\}\{\mathtt{x} = c'\} \models_i \lfloor e'_t \rfloor_S \Longrightarrow v' \sqsubset v$$

Evaluation of the compiled code:

The compilation is:

$$\begin{array}{rcl} \lfloor e_t \rfloor_S & = & \mathtt{let}\ \mathtt{x} = \mathtt{fun}\ \mathtt{d}_1 \ldots \mathtt{d}_m \rightarrow \\ & & \quad \mathtt{match}\ \mathtt{d}_1, \ldots, \mathtt{d}_m\ \mathtt{with}\ \lfloor \sigma_1 \Longrightarrow e_{t1} \rfloor_{S,\sigma} \mid \ldots \mid \lfloor \sigma_n \Longrightarrow e_{tn} \rfloor_{S,\sigma} \\ & & \mathtt{in}\ \lfloor e'_t \rfloor_S \end{array}$$

This let expression is evaluated by (i-let). The function definition is evaluated to c' by (i-fun). By the result of the induction hypothesis, the body of let expression $\lfloor e'_t \rfloor_S$ is evaluated to v':

$$\frac{\begin{array}{l} \mathcal{E}\{S\} \models_i \mathtt{fun}\ \mathtt{d}_1 \ldots \mathtt{d}_m \rightarrow \\ \qquad \mathtt{match}\ \mathtt{d}_1, \ldots, \mathtt{d}_m\ \mathtt{with} \\ \qquad \mid \lfloor \sigma_1 \Longrightarrow e_{t1} \rfloor_{S,\sigma} \mid \ldots \mid \lfloor \sigma_n \Longrightarrow e_{tn} \rfloor_{S,\sigma} \Longrightarrow c' \\ \qquad \mathcal{E}\{S\}\{\mathtt{x} = c'\} \models_i \lfloor e'_t \rfloor_S \Longrightarrow v' \end{array}}{\mathcal{E}\{S\} \models_i \lfloor e_t \rfloor_S \Longrightarrow v' \sqsubset v}$$

The whole result is therefore v' which is semantically compatible with v.

(e-gcase)

Typing and evaluation in GML:

In the case (e-gcase) we have the following typings of $\mathtt{x}_\sigma : \tau$ and $\mathtt{x}_\sigma : S(\tau)$:

$$\frac{\Gamma(\mathtt{x}) = \sigma \quad \tau \leq_{S_0} \sigma}{\Gamma \vdash \mathtt{x}_\sigma : \tau}$$

$$\frac{\Gamma(\mathtt{x}) = \sigma \quad S_1 = \{(\nu \Rightarrow (S \circ S_0)(\nu) \mid \nu \in BV(\sigma)\} \quad S(\tau) \leq_{S_1} \sigma}{\Gamma \vdash \mathtt{x}_\sigma : S(\tau)}$$

Because the used rule is (e-gcase), $\mathtt{x}$ must be bound by a generic case closure in E. Suppose that p-th type case is selected in this evaluation under an instantiation S_2 like $S(\tau) \leq_{S_2} \sigma_p$:

$$\frac{\begin{array}{c} E(\mathtt{x}) = \mathit{gcase}(E', \_, [\sigma_1 \Rightarrow e_{t1}, \ldots, \sigma_n \Rightarrow e_{tn}]) \\ S(\tau) \not\leq \sigma_1 \ \ldots \ S(\tau) \not\leq \sigma_{p-1} \ \ S(\tau) \leq_{S_2} \sigma_p \\ E' \models S_2(e_{tp}) \Longrightarrow v \end{array}}{E \models \mathtt{x}_\sigma : S(\tau) \Longrightarrow v}$$

Induction hypothesis:

From $\mathcal{E} \sqsubset E$, $E(x) = gcase(E', \_, [\sigma_i \Rightarrow e_{ti}]) = gcase(E', \_, [\sigma_i \Rightarrow S'(e'_{ti})])$ for some S' and type cases e'_{ti} and

$$\mathcal{E}(\mathtt{x}) = clos(\mathcal{E}'\{S'\}, \_, \mathtt{d}_1, \mathtt{fun}\ \mathtt{d}_2 \ldots \mathtt{d}_n \rightarrow \ \mathtt{match}\ \mathtt{d}_1, \ldots, \mathtt{d}_n\ \mathtt{with} \\ \mid \lfloor \sigma_1 \Longrightarrow e'_{t1} \rfloor_{S',\sigma} \mid \ \ldots \ \mid \lfloor \sigma_n \Longrightarrow e'_{tn} \rfloor_{S',\sigma})$$

where

$$\mathcal{E}' \sqsubset E' \\ \exists \Gamma',\ E' \in S'(\Gamma') \\ \Gamma' \vdash e'_{ti} \qquad \sigma_i = Gen(\Gamma', e'_{ti})$$

Now the evaluation of $S_2(e_{tp})$ is rewritten as $E' \models (S_2 \circ S')(e'_{tp}) \Longrightarrow v$. Since $Dom(S_2) = BV(\sigma_p)$, therefore by renaming of $BV(\sigma_p)$, we can make $BV(\sigma_p)$ is out of reach of S' and $S_2 \circ S' = S_2 + S'$. From the generalization of $\sigma_p = Gen(\Gamma', e'_{tp})$, the members of $Dom(S_2) = BV(\sigma_p)$ never occur freely in Γ'. Therefore $(S_2 + S')(\Gamma') = S'(\Gamma')$ and from $E' \in S'(\Gamma')$, we have:

$$E' \in (S_2 + S')(\Gamma') = S'(\Gamma')$$

From these we can use the induction hypothesis for the evaluation of $S_2(e_{tp}) = (S_2 + S')(e'_{tp})$, $E' \models (S_2 + S')(e'_{tp}) \Longrightarrow v$:

$$\mathcal{E}'\{S_2 + S'\} \models_i \lfloor e'_{tp} \rfloor_{S_2+S'} \Longrightarrow v'$$

for some v' such that $v' \sqsubset v$.

By renaming of bound variables of σ and σ_p, we can make the domains of S_1,S_2 and S' are all distinct each other. Therefore adding $\{S_1\}$ to the environment of the last evaluation never changes the result:

$$\mathcal{E}'\{S_1 + S_2 + S'\} \models_i \lfloor e'_{tp} \rfloor_{S_2+S'} \Longrightarrow v' \sqsubset v$$

Evaluation of the compiled code:

$\mathtt{x}$ is compiled as

$$\lfloor \mathtt{x}_\sigma : \tau \rfloor_S = \mathtt{x}\ \lfloor S_0(\delta_1) \rfloor_S\ \ldots \lfloor S_0(\delta_n) \rfloor_S$$

This compilation is evaluated under $\mathcal{E}\{S\}$ as follows using (i-app) n times, like the proof of (e-gvar) case in the Theorem 12:

$$\frac{\begin{array}{l} \mathcal{E}\{S\} \models_i \mathtt{x} \Longrightarrow\ clos(\mathcal{E}'\{S'\}, \_, \mathtt{d}_1, \mathtt{fun}\ \mathtt{d}_2 \ldots \mathtt{d}_n \rightarrow \ \mathtt{match}\ \mathtt{d}_1, \ldots, \mathtt{d}_n\ \mathtt{with} \\ \qquad\qquad \mid \lfloor \sigma_1 \Longrightarrow e'_{t1} \rfloor_{S',\sigma} \mid \ \ldots \ \mid \lfloor \sigma_n \Longrightarrow e'_{tn} \rfloor_{S',\sigma}) \\ \mathcal{E}\{S\} \models_i \lfloor S_0(\delta_i) \rfloor_S \Longrightarrow «S_1(\delta_i)» \\ \mathcal{E}'\{S'\}\{S_1\} \models_i\ \mathtt{match}\ \mathtt{d}_1, \ldots, \mathtt{d}_n\ \mathtt{with} \\ \qquad \mid \lfloor \sigma_1 \Longrightarrow e'_{t1} \rfloor_{S',\sigma} \mid \ \ldots \ \mid \lfloor \sigma_n \Longrightarrow e'_{tn} \rfloor_{S',\sigma} \Longrightarrow v'' \end{array}}{\mathcal{E}\{S\} \models_i \mathtt{x}\ \lfloor S_0(\delta_1) \rfloor_S\ \ldots \lfloor S_0(\delta_n) \rfloor_S \Longrightarrow v''}$$

Now we are going to prove $v' = v''$. The expression in the match $\mathtt{d}_1, \ldots, \mathtt{d}_n$ is evaluated as follows, because $Dom(S_1) = \{\delta_i\}$:

$$\mathcal{E}'\{S_1\}\{S'\} \models_i \mathtt{d}_1, \ldots, \mathtt{d}_n \Longrightarrow «S_1(\delta_1)», \ldots, «S_1(\delta_n)»$$

Since $S_1(\delta_i) = (S \circ S_0)(\delta_i)$, and $S(\tau) \not\leq \sigma_1 \ \ldots \ S(\tau) \not\leq \sigma_{p-1}\ S(\tau) \leq_{S_2} \sigma_p$ from Lemma 20,

$$\begin{array}{l} 1 \leq i < p, \quad «S_1(\delta_1)», \ldots, «S_1(\delta_n)» \not\leq \lfloor \sigma_i \rfloor_\sigma \\ «S_1(\delta_1)», \ldots, «S_1(\delta_n)» \leq_\Sigma \lfloor \sigma_p \rfloor_\sigma \end{array}$$

where $\Sigma = \{\mathtt{d}_i \Rightarrow «S_2(\delta_i)» \mid \delta_i \in BDV(\sigma)\}$.

Then we evaluate the matched case $\lfloor e'_{tp} \rfloor_{S_2+S'}$ in $\mathcal{E}'\{S_1\}\{S'\}\{\Sigma\}$. This is equal to the environment of the induction hypothesis $\mathcal{E}'\{S_1\}\{S_2\}\{S'\} \models_i \lfloor e'_{tp} \rfloor_{S_2+S'} \Longrightarrow v'$:

$$\mathcal{E}'\{S_1\}\{S'\}\{\Sigma\} \models_i \lfloor e'_{tp} \rfloor_{S_2+S'} \Longrightarrow v' \sqsubset v$$

From these, by applying (i-match), we obtain the evaluation of the match expression:

$$\frac{\begin{array}{c} \mathcal{E}'\{S_1\}\{S'\} \models_i \mathtt{d}_1, \ldots \mathtt{d}_n \Longrightarrow «S_1(\delta_1)», \ldots, «S_1(\delta_n)» \\ 1 \leq i < p,\ «S_1(\delta_1)», \ldots, «S_1(\delta_n)» \not\leq \lfloor \sigma_i \rfloor_\sigma \\ «S_1(\delta_1)», \ldots, «S_1(\delta_n)» \leq_\Sigma \lfloor \sigma_p \rfloor_\sigma \\ \mathcal{E}'\{S_1\}\{S'\}\{\Sigma\} \models_i \lfloor e_{tp} \rfloor_{S_2+S'} \Longrightarrow v' \end{array}}{\begin{array}{ll} \mathcal{E}'\{S_1\}\{S'\} \models_i & \mathtt{match}\ \mathtt{d}_1, \ldots, \mathtt{d}_n\ \mathtt{with} \\ & \mid \lfloor \sigma_1 \Longrightarrow e_{t1} \rfloor_{S',\sigma} \mid \ \ldots \ \mid \lfloor \sigma_n \Longrightarrow e_{tn} \rfloor_{S',\sigma} \Longrightarrow v'' = v' \sqsubset v \end{array}}$$

Finally, from $v'' = v'$, we have the goal:

$$\mathcal{E}\{S\} \models_i \mathtt{x}\ \lfloor S_0(\delta_1) \rfloor_S \ \ldots \lfloor S_0(\delta_n) \rfloor_S \Longrightarrow v' \sqsubset v$$

(e-gcase-err) We have the same typing of x, the environment bindings and the compilation of $\mathtt{x}_\sigma : \tau$ as (e-gcase), but in this case its evaluation fails:

$$\frac{\begin{array}{c} E(\mathtt{x}) = gcase(E', \_, [\sigma_1 \Rightarrow e_{t1}, \ldots, \sigma_n \Rightarrow e_{tn}]) \\ S(\tau) \not\leq \sigma_1 \ \ldots \ S(\tau) \not\leq \sigma_n \end{array}}{E \models \mathtt{x}_\sigma : S(\tau) \Longrightarrow gerr}$$

$$\begin{array}{l} \mathcal{E}(\mathtt{x}) = clos(\mathcal{E}'\{S'\}, \_, \mathtt{d}_1, \mathtt{fun}\ \mathtt{d}_2 \ldots \mathtt{d}_n \rightarrow\ \mathtt{match}\ \mathtt{d}_1, \ldots, \mathtt{d}_n\ \mathtt{with} \\ \qquad\qquad\qquad\qquad \mid \lfloor \sigma_1 \Longrightarrow e'_{t1} \rfloor_\sigma \mid \ \ldots \ \mid \lfloor \sigma_n \Longrightarrow e'_{tn} \rfloor_\sigma) \end{array}$$

$$\lfloor \mathtt{x}_\sigma : \tau \rfloor_S = \mathtt{x}\ \lfloor S_0(\delta_1) \rfloor_S \ \ldots \lfloor S_0(\delta_n) \rfloor_S$$

The run time type arguments in the compiled expression is evaluated in the same way:

$$\mathcal{E}'\{S_1\}\{S'\} \models_i \lfloor S_0(\delta_i) \rfloor_S \Longrightarrow «S_1(\delta_i)»$$

In this time $1 \leq i \leq n,\ S(\tau) \not\leq \sigma_i$. Therefore from Lemma 20,

$$1 \leq i \leq n,\ \ \text{«}S_1(\delta_1)\text{»}, \ldots, \text{«}S_1(\delta_n)\text{»} \not\leq \lfloor \sigma_i \rfloor_\sigma$$

This means the evaluation of the run time type matching fails and raise an error *gerr* by (i-match-err):

$$\frac{\begin{array}{c}\mathcal{E}'\{S_1\}\{S'\} \models_i \mathtt{d}_1, \ldots \mathtt{d}_n \Longrightarrow \text{«}S_1(\delta_1)\text{»}, \ldots, \text{«}S_1(\delta_n)\text{»} \\ 1 \leq i \leq n,\ \text{«}S_1(\delta_1)\text{»}, \ldots, \text{«}S_1(\delta_n)\text{»} \not\leq \lfloor \sigma_i \rfloor_\sigma\end{array}}{\begin{array}{l}\mathcal{E}'\{S_1\}\{S'\} \models_i \ \mathtt{match}\ \mathtt{d}_1, \ldots, \mathtt{d}_n\ \mathtt{with} \\ \qquad\qquad\qquad \mid \lfloor \sigma_1 \Longrightarrow e_{t1} \rfloor_{S',\sigma} \mid \ \ldots \ \mid \lfloor \sigma_n \Longrightarrow e_{tn} \rfloor_{S',\sigma} \Longrightarrow gerr\end{array}}$$

From this evaluation we can construct the evaluation of the compilation of $\mathtt{x}_\sigma : \tau$ like the case of (e-gcase) (the detail is omitted):

$$\mathcal{E}\{S\} \models_i \mathtt{x}\ \lfloor S_0(\delta_1) \rfloor_S\ \ldots \lfloor S_0(\delta_n) \rfloor_S \Longrightarrow gerr$$

(e-genericrec), (e-gcase-r) Provable in the same way as each corresponding non-recursive rule.

□

7.7 Comparison with related works

7.7.1 No regular type variables in patterns

The type variables in type patterns must be closed and dynamic — they must be without regular type variables. However, in the initial study of extensional polymorphism [DRW95], we could have also type patterns with regular type variables. While a dynamic type pattern variable matches with any type, a regular type pattern variables matched only with a type variable. For example, the following `is_polymorphic_null` generic function can distinguish the polymorphic null list from monomorphic lists using a regular type variable in type patterns:

```
generic is_polymorphic_null = case
| ∀α.α list → unit ⇒ true                    (* we have no longer this! *)
| ∀δ.δ list → unit ⇒ false
;;

# is_polymorphic_null [];;
- : bool = true
# is_polymorphic_null [1];;
- : bool = false
# is_polymorphic_null ([]:int list);;
- : bool = false
```

But we removed this regular type pattern variables from our system. The first reason is that we have no useful examples other than `is_polymorphic_null`. Second, much importantly, they will damage the type substitution lemma of the generic constraint typing (Chapter 8). Lack of this good property prohibits us from writing simple type reconstruction algorithm.

7.7.2 Semantics over the source language

The semantics of generic values are type-dependent: the semantics of a generic primitive g under an evaluation type context τ is given by the query funciton *gen* as $gen(\mathtt{g}, \tau)$, and derived generic values and user-defined generic values are evaluated to closures which take evaluation type contexts of their instances. Due to this fundamental design of generic values, the operational semantic system of extensional polymorphism must be defined over type annotated expressions with type operations.

In Haskell[HJW92], semantics of class declarations and overloaded values are *not* directly given as an operational semantic system like ours. They are given as a translation using dictionary passing to its core language without classes, whose semantics is defined without any type operation. Translation results are well-typed program of the core language and their type soundness is proven in this level.

On the other hand, in extensional polymorphism, we do not take the same approach: semantics of generic values are not defined by translation into an intermidiate language (i.e. IML) without generic values. It is since translated IML expressions are often *untypable* and therefore it would be technically impossible to prove evaluation soundness of well-typed GML programs inside the IML semantic system which lacks typing. For example:

```
generic plus = case                         let plus = function ty →
| int → int → int ⇒ (+)                       match ty with
| float → float → float ⇒ (+.)      ⟹        | «int» ⇒ (+)
;;                                             | «float» ⇒ (+.)
                                            ;;
```

The compilation of the above `plus` is not typable, since the types of `(+)` and `(+.)` are not unifiable.

7.8 Conclusion

We introduced a way to define user-defined generic values, `generic` binding. A `generic` binding consists of a list of type cases — each case contains a type scheme where the value is defined and its corresponding expression. `generic` bindings can be recursive and it provides impressively expressive power; we can write overloaded values using recursions over the structure of types.

A user-defined generic value is compiled to a function which takes run time type arguments which inform how it is instantiated and match with type patterns to select the appropriate type methods for the evaluation type context of its instances. The basic compilation framework using the run time type dispatching introduced in Chapter 3 provides efficient compiled code for user-defined generics.

Non-exhaustiveness and constraint typing

Since a user-defined generic value is defined as a finite set of type cases, it can be non-exhaustive: its semantics may not be defined for some instance types. In spite of this partiality, they have total polymorphic type schemes in the current type system, since the typing system for generic values introduced in Chapter 2 is too simple to express such non-exhaustiveness. Under this restricted type system, user-defined generic values must check their well-typedness at run time, verifying dispatched run time types with their type patterns. If there is no applicable type case, they raise a run time exception. This imperfection of the type system will be solved in the generic constraint typing system of Chapter 8, by using type information of non-exhaustiveness of generic values. This generic constraint typing checks the validity of generic instances by precalculating the run time type case selections statically at type checking time. This gives us a hint for further optimizations of the compilation scheme for user-defined generic values. Once the type checker validates all the type case selections, it is not necessary to check them again at run time by pattern matching of run time types. Using the result of the well-typedness proof called *generic flow* of the generic constraint typing, the type case selection can be radically optimized. In Chapter 9, we will formalize this generic flow compilation.

Extension of user-defined generic values

Our compilation scheme of user-defined generic values as pattern matchings unwittingly introduced a restriction to the definition of user-defined generic values: the type cases of a user-defined generic value must be declared once and for all in one `generic` binding, just like the cases of a normal pattern matching expression are defined in one place. This goes against programming modularity. When we overload values to one identifier, usually what they share is only the name and their computational semantics are somewhat independent from each other. Therefore type cases should be definable independently in different places, and the generic value is defined as their union. In addition, generic values should also be extensible. Even after their definitions, there should be a way to add new type cases to them easily. These separate definition and extension of generic values are studied later in Chapter 11.