Chapter 11

Generic value extensions

11.1 Introduction

Extensible generic values

User-defined generic values are defined by `generic` bindings over more than one types: for example, the addition functions of integers `(+)` and floats `(+.)`, and the concatenation of lists `(@)` and arrays can be overridden in one generic value `plus` like

```
generic plus = case
| int → int → int ⇒ (+)
| float → float → float ⇒ (+.)
| ∀δ.δ list → δ list → δ list ⇒ (@)
| ∀δ.δ array → δ array → δ array ⇒ Array.append
;;
```

Thus generic values provide non-parametric, ad-hoc polymorphism which reduces the programmers' name space complexity. The type methods of a generic value are gathered and form one generic value, but their semantics are independent of each other. They share nothing essential except the name and the vague notion of their functionality, "addition" for example. Otherwise, if they shared something algorithmic, we could unify them as a parametric polymorphic function. Because of this independence of semantics of type methods, it is desirable that these type methods could be defined in different places, even later the initial declaration of the generic value. After the definition of the initial `plus`, if a programmer defines a new type, say `t` which is something "addible", then it is natural that he wants to add the type case of `t` → `t` → `t` to `plus`, by extending the existing generic value `plus`. Therefore the *extension* of generic values must be indispensable feature of extensional polymorphism. However, in the previous chapters, the `generic` binding defined values once and for all: to define a generic value, we had to prepare all the type methods and declare them in a list, and once after the definition, it was too late to have additional type methods. There was no easy way to have the extended generic values except by redefining them from scratch. Therefore, in this chapter, we discuss the user extensibility of generic values.

Closed world assumption with open recursion

We choose the *closed world assumption* as the extension scheme for extensional polymorphism. In the closed world assumption, semantics of an extended overloaded value is statically determined then closed at the extension; its extended semantics are independent of the later extension to the same identifier. In some sense, closed world extension can be regarded as redefinition of values with extended semantics. But our extension in closed world is not mere redefinition, since it supports *open recursion*: the semantics of recursive calls of a generic value are open to its extensions.

Contrary to the closed world, under the *open world assumption*, semantics of overloaded values are always open to extensions until all the program modules are prepared i.e. link time. It is quite powerful, but does not seem to fit with the eager evaluation strategy of ML.

Type case inclusion

Extension of generic values is achieved by *type case inclusion*, insertion of the type cases of the original value inside the type case declaration of a newly extended generic value. For example, the following definition of `plus'` includes the type cases of the `plus` defined at the beginning of this chapter:

```
generic plus' = case
| include plus
| string → string → string ⇒ (^)
;;
```

This extended `plus'` will have the equivalent semantics to the following generic definition whose type case definitions are concatenation of the original and new type cases:

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

This type case inclusion phrase can be placed anywhere inside a type case list. Therefore we can carefully choose its position in the type case list to control the case definition order dependent semantics of an extended user-defined generic value.

Type case compilation with self

To support open recursion at extensions, the compilation of a type case is updated so that it takes an additional argument for its recursive call back point, which corresponds with the “self” parameter of class systems. The recursive occurrences of user generic values are replaced by this self parameter, and the fixed point of a generic value is passed through this self.

Thanks to this abstraction of the self parameter, definition of type cases and their invocations can be separated in the compilation of extensional polymorphism. As a result, the compilation of user generics are simplified to arrays of type case compilations and one "meta-generic" type case invocation function.

Outline of this chapter

First we survey differences of the open and closed world assumptions of overloading, and decide which scheme is appropriate for extensional polymorphism in Section 11.2. Then in Section 11.3, we introduce new syntax of type case inclusion. Its semantics is also explained. Section 11.4 modifies typing systems for the type case inclusions. Corresponding with the type case concatenation semantics of the type inclusion, extended generic values will have concatenated pattern constraints. In Section 11.5, a new compilation scheme for the generic binding and generic instances are defined. Section 11.6 explains an optimization of extended generic values, removal of type cases which are duplicated in the course of extensions.

11.2 Open or closed

There are mainly two schemes of extension of overloaded values, *open* and *closed world assumptions*. First we compare these two and decide which to choose for our system.

11.2.1 Open world assumption

The open world assumption is known also as late-binding of "self" in object-oriented languages. It changes semantics of already defined values globally and this modification automatically affect semantics of their derived values, too. Therefore we can extend sets of many values by single extension.

Let's see an example of extension in the open world. Suppose that we have the following initial definition of `print`:

```
generic rec print = case
| int → unit ⇒ print_int
| ∀δ.δ list → unit ⇒ fun l → List.iter print l
;;
```

In the open world, adding a new type case for `float → unit` modifies the type and semantics of the original `print`. The declaration of this extension could be something like below:

```
extend print with
| float → unit ⇒ print_float
;;
```

The extended `print` should be equivalent to the following generic value

```
generic rec print = case
| int → unit ⇒ print_int
| ∀δ.δ list → unit ⇒ fun l → List.iter print l
| float → unit ⇒ print_float
;;
```

and be able to print not only floats, but also lists of floats:

```
# print 1.0;;
1.0
- : unit = ()

# print [[1.2]; [3.4;5.6]];;
1.2 3.4 5.6
- : unit = ()
```

What is more interesting in the open world assumption is that the derived generic values which use this `print` can also benefit the extensions to `print` automatically, thanks to the global modification of the semantics. For instance, the following derived generic value `print_twice`:

```
let print_twice x = print x; print x;;
```

can work for integers, floats (whose case was extended just before) and their lists. Then if we extend `print` again, for `string → unit` this time:

```
extend print with
| string → unit ⇒ print_string
;;
```

Then this modification of the semantics of `print` will be automatically reflected to `print_twice`, its derived generic value; it can work for strings and their lists:

```
# print_twice ["hello "; "world "];;
hello world hello world
- : unit = ()
```

Thus the semantics of generic values are always open to the later extensions in the open world framework. This openness is quite powerful and useful as far as we could know all the extensions which exist in the system. Otherwise, all the semantics of generic values become suddenly unsure to programmers, even though type checker could provide their type soundness. In the open world assumption, there is always a risk that generic extensions could override the original type case which users want to actually call, and cause unexpected results. Imagine that our extension to `print` might be silently overridden by another extension in a library we link together. The library was written by someone else and we would not know about its extension of `print`. The linked program would work differently from what we imagined and this phenomenon surely annoys us a lot. In addition, late-binding of the open world does not fit well with the ML's eager evaluation strategy.

Therefore, we do not want to introduce the open world assumption to extensional polymorphism framework. You could add new type cases easily using the open world assumption and it should be helpful in some cases, however, it can also introduce heavy semantic confusion to our programs, since in the open world the semantics of generic values could not be fixed until all modules are linked together. This semantic instability does not go well with the ML stable semantics.

11.2.2 Closed world assumption

Rather than the too global and too dynamic open world assumption, we prefer the static *closed world assumption*. Under the closed world assumption, extensions do not modify the semantics and types of the original values. Instead, they create new values which have extended semantics but are independent of the originals.

A new extended generic value in the closed world assumption will contain the original type cases and newly added ones and nothing more: the semantics of the extended value is statically defined and free from the contamination from the other parallel extensions from the original value. The closed extension is not the global modification of the semantics, therefore we cannot have such magical effect as we have seen in the open world, but in the programming style of ML, we prefer the closed world's static semantics and typing to those of the unstable open world.

11.3 Extension syntax and semantics

11.3.1 Extension is not mere redefinition

Since extensions under the closed world assumption define new generic values, we do not need complicated semantic update which we should require in the open world, but still it is impossible, or very hard to achieve in the current extensional polymorphism framework. Let's see the difficulty to try adding some type cases to an already defined generic value inside the current system. Let the following generic binding be the initial definition of `print`:

```
generic print = case
| int → unit ⇒ print_int
| string → unit ⇒ print_string
;;
```

Then, we define the following `print'`, adding the case for $\forall\delta.\delta$ `list` $\rightarrow$ `unit` :

```
generic rec print' = case
| ∀δ.δ list → unit ⇒ fun l → List.iter print' l
| ∀δ.δ → unit ⇒ print
;;
```

print' first tries the case of $\forall\delta.\delta$ list → unit and if it does not match, the second case calls the original print as the default. This works well. The result print' can print both base types and their lists:

```
# print' 1; print' "hello"; print' [2;3]; print' [["hello "]; ["world"]];;
1 hello 2 3 hello world
- : unit = ()
```

But if we continue extending print' we will have a problem:

```
generic print'' = case
| float → unit ⇒ print_float
| ∀δ.δ → unit ⇒ print'
;;
```

print'' is obtained by adding the new case for float → unit. It can print floats, but it cannot print float lists:

```
# print'' 3.14;;
3.14
- : unit = ()

# print'' [1.0; 2.0];;
Error: This generic full instance is used with type float list → unit
which is not its valid instance of print''
```

It is just since print' and print'' are different generic values and print' never calls print'' recursively. print'' is just another generic value using print' and has no qualification as an "extension" of print'. We expect that the real extension of print' must also be able to handle the lists of floats, something equivalent to the following generic value definition:

```
generic rec print''' = case
| float → unit ⇒ print_float
| int → unit ⇒ print_int
| string → unit ⇒ print_string
| ∀δ.δ list → unit ⇒ fun l → List.iter print''' l
;;
```

Since the last type case recursively calls print'' itself, it can print the lists of floats, too. We cannot obtain a generic value which works equivalently to the above simply by redefining generic values as we have tried.

We could still continue our struggle to have generic extensions inside the current framework, but there would not be a good solution. For example, even if we tried to define print functions separately, as a function which receive a fixed point fp for an extended print:

```
generic p1 = case
| ∀δ.(δ → unit) → int → unit ⇒ fun fp → print_int
| ∀δ.(δ → unit) → float → unit ⇒ fun fp → print_float
;;

generic p2 = case
| ∀δ.(δ → unit) → δ list → unit ⇒
        fun fp → fun l → List.iter fp l
| ∀δ.(δ → unit) → δ array → unit ⇒
        fun fp → fun l → Array.iter fp l
;;
```

And merged them by the following `print` recursive definition, which applies itself to `p1` and `p2` as the fixed point:

```
generic rec print = case
| ∀δ.δ → unit ⇒ fun x → p1 print x
| ∀δ.δ → unit ⇒ fun x → p2 print x
;;
```

It could not work. It is since in the normal typing the recursive occurrences of `print` have the type $\delta' \to$ `unit` which contain a free dynamic type variable δ', and the constraint type inference must unify it to `dyn` to avoid the coherence problem (See Section 10.4). Even if we could solve this problem and make this `print` work for lists and arrays of integers and floats without any typing problem, we would still have a problem of efficiency: we had to introduce a new generic type case selection for each merge, and it should be bad for efficiency of generic values which will be often extended such as `print`.

11.3.2 Open recursion in the closed world

The closed extension should not be just a redefinition of values, since as we have seen, *open recursion* is necessary even in the closed world. Semantics of a recursive overloaded value is open to its extensions, in the open recursion. Thanks to this openness of semantics, newly added definitions can also be called recursively from the original part. The open recursion in the closed world cannot be achieved inside the current framework. Extensional polymorphism requires a new construct for it.

11.3.3 Type case inclusion

We extend the syntax of the generic binding so that it can have a new *type case inclusion phrase* to import type case definitions of other generic values. For example:

```
generic print' = case
| include print
| float → unit ⇒ print_float
;;
```

This `include` phrase imports the type case definitions of `print` (whose definition is in Section11.3.1) at its place in the definition of `print'`. The new `print'` will have the equivalent semantics to a generic value with the following definition:

```
generic print' = case
| int → unit ⇒ print_int
| string → unit ⇒ print_string
| float → unit ⇒ print_float
;;
```

Now `print'` works for the printing of `int`, `string` and `float`. Unlike the last example of the previous section, all the merged type cases are flattened. Therefore continuous extension will not cause any overhead.

Our extension defines new generic values, but it is not the normal redefinition, but with open recursion. Imagine that now the original `print` is defined recursively with the type case for `∀δ.δ list → unit` :

```
generic rec print = case
| int → unit ⇒ print_int
| string → unit ⇒ print_string
| ∀δ.δ list → unit ⇒ fun l → List.iter print l
;;
```

Then, the following extended generic value `print'`:

```
generic print' = case
| float → unit ⇒ print_float
| include print
;;
```

should be equivalent to the following generic value:

```
generic rec print' = case
| float → unit ⇒ print_float
| int → unit ⇒ print_int
| string → unit ⇒ print_string
| ∀δ.δ list → unit ⇒ fun l → List.iter print' l
;;
```

The type case definitions of the original `print` are included in this definition of `print'`, but the recursive occurrences of `print` is now replaced by `print'`, so that the imported type cases can recursively call the newly extended `print'`, not the original `print`. In this sense, the type case inclusion is not pure inclusion of source, but with open recursion. Thanks to this mechanism, the new `print'` can print not only `float`, but also `float list`, `float list list` and so on, using combinations of the original and new type cases:

```
# print' [1.2; 3.4; 5.6];;
1.2 3.4 5.6
- : unit = ()
```

Thus, the open recursion under the closed world assumption is achieved: generic values are "open" to others who include them, but it is not an open world; `print` and `print'` are different values and independent of each other.

This extension as inclusion of type case definitions with rebinding of recursive calls are quite natural to understand and powerful enough: we will not need any special restriction for the inclusion phrases and additional type cases. Like the normal generic definition, each type case must be admissible, but that is all the required restriction. Without any lousy restriction of additional cases, "extension chains" of generic values are easily formed by including extended generic values and defining new values again and again. This include phrase can be put at anywhere in the type cases and it is possible to have more than one inclusion phrases in one generic definition:

```
generic super_print = case
| include print_basic
| t → unit ⇒ print_t
| include print_tuples
| t' → unit ⇒ print_t'
| include print_lists_and_arrays
;;
```

This enables us to gather extensions of different chains into one, like the multiple inheritance of the class system.

11.3.4 New syntax

The required syntactic extension is simple. We only have to add type case inclusion phrase into the syntax of type cases:

$$
\begin{array}{lcll}
tc & ::= & \sigma \Rightarrow e & \text{Type case definitions} \\
 & | & \texttt{include x} & \text{Type case inclusions} \\
 & & & \\
tc_t & ::= & (\sigma \Rightarrow e_t) : \sigma & \text{Type case definitions} \\
 & | & \texttt{include x} : \sigma & \text{Type case inclusions}
\end{array}
$$

In the type-annotated syntax, type cases now have a new type scheme annotation σ for their typing.

11.4 Typing

11.4.1 Pattern constraints concatenation

Newly created generic values by extension have pattern constraints which are obtained intuitively by concatenating the pattern constraints of the included generic values and newly added cases.

Suppose we have the following initial definition of `print`:

```
generic rec print = case
| int → unit ⇒ print_int
| string → unit ⇒ print_string
| ∀δ.δ list → unit ⇒ fun l → List.iter print l
;;

val print : μκ.[ int → unit
               | string → unit
               | ∀δ.δ list → unit {δ → unit <: κ} ]
```

The following extended generic value `print'`

```
generic print' = case
| float → unit ⇒ print_float
| include print
;;
```

will have the extended constraint as follows:

```
val print' = μκ.[ float → unit
                | int → unit
                | string → unit
                | ∀δ.δ list → unit {δ → unit <: κ} ]
```

Note that in order to join the new type case for `float → unit` in the recursion, its type pattern constraint is stored *inside* of the μ-abstraction of the original constraint. Once the new generic value is defined, its type information has no reference to the original value.

11.4.2 Normal typing layer

The extensions to the normal typing layer is trivial. The types of generic values are given as the least general anti-unifiers of the type patterns and the type schemes of included generic values. Since now we have two kinds of type cases, type case definitions and type case inclusions available, we separate these typing rules from those of generic bindings:

$$\textbf{(t-typecase)}\ \frac{\Gamma \vdash e_t : \tau \qquad \sigma = Gen(\Gamma, e_t) \qquad FV(\sigma) = \emptyset}{\Gamma \vdash \sigma \Rightarrow e_t : \sigma}$$

$$\textbf{(t-include)}\ \frac{\Gamma(\mathtt{x}) = \sigma \qquad FV(\sigma) = \emptyset}{\Gamma \vdash \mathtt{include\ x} : \sigma}$$

$$\textsf{(t-e-generic)}\ \frac{\Gamma \vdash tc_{ti} : \sigma_i \quad \sigma = p\_antiunif(\{\sigma_i\}) \quad \Gamma\{\mathtt{x} : \sigma\} \vdash e_t : \tau}{\Gamma \vdash \mathtt{generic}\ \mathtt{x}_\sigma = \mathtt{case}\ tc_{t1} \mid \ldots \mid tc_{tn}\ \mathtt{in}\ e_t : \tau}$$

$$\textsf{(t-e-genericrec)}\ \frac{\Gamma\{\mathtt{x} : \sigma\} \vdash tc_{ti} : \sigma_i \quad \sigma = p\_antiunif(\{\sigma_i\}) \quad \Gamma\{\mathtt{x} : \sigma\} \vdash e_t : \tau}{\Gamma \vdash \mathtt{generic}\ \mathtt{rec}\ \mathtt{x}_\sigma = \mathtt{case}\ tc_{t1} \mid \ldots \mid tc_{tn}\ \mathtt{in}\ e_t : \tau}$$

Only values defined by generic bindings are permitted for the type case inclusion, however, in the normal typing, we cannot distinguish them from generic primitives and let defined generic values. This check and the construction of newly defined values' constraints are handled in the constraint typing layer.

11.4.3 Constraint typing

Pattern constraint concatenation

We first define the *pattern constraint concatenation* $\oplus$. This is defined between pattern constraints or pattern constraints abstracted by one μ-abstraction:

$$\begin{array}{rcrcr}
[k_1|\ldots|k_n] & \oplus & [k_{n+1}|\ldots|k_m] & = & [k_1|\ldots|k_m] \\
\mu\kappa.[k_1|\ldots|k_n] & \oplus & [k_{n+1}|\ldots|k_m] & = & \mu\kappa.[k_1|\ldots|k_m] \\
[k_1|\ldots|k_n] & \oplus & \mu\kappa.[k_{n+1}|\ldots|k_m] & = & \mu\kappa.[k_1|\ldots|k_m] \\
\mu\kappa.[k_1|\ldots|k_n] & \oplus & \mu\kappa.[k_{n+1}|\ldots|k_m] & = & \mu\kappa.[k_1|\ldots|k_m]
\end{array}$$

When a non μ-abstracted pattern constraint is concatenated with a μ-abstracted constraint, the former is taken into the abstraction of the latter and forms a new recursive constraint which consists of all the patterns. For example,

$$\begin{array}{l}
\text{let } \mathcal{K}_1 = [\mathtt{int} \to \mathtt{unit} \mid \mathtt{float} \to \mathtt{unit}] \\
\text{let } \mathcal{K}_2 = \mu\kappa.[\mathtt{string} \to \mathtt{unit} \mid \forall\delta.\delta\ \mathtt{list} \to \mathtt{unit}\ \{\delta \to \mathtt{unit} <: \kappa\}] \text{ in} \\
\mathcal{K}_1 \oplus \mathcal{K}_2 = \mu\kappa.[\mathtt{int} \to \mathtt{unit} \mid \mathtt{float} \to \mathtt{unit} \mid \mathtt{string} \to \mathtt{unit} \\
\qquad\qquad\qquad \mid \forall\delta.\delta\ \mathtt{list} \to \mathtt{unit}\ \{\delta \to \mathtt{unit} <: \kappa\}]
\end{array}$$

If we concatenate two μ-abstracted constraints, we perform α conversions to them and have one same constraint variable. We also suppose that by putting constraints type pattern constraints inside μ-abstractions, no free occurrences of constraint variables become bound. For instance,

$$\begin{array}{l}
\text{let } \mathcal{K}_1 = \mu\kappa.[\mathtt{int} \to \mathtt{unit} \mid \forall\delta.\delta\ \mathtt{list} \to \mathtt{unit}\ \{\delta \to \mathtt{unit} <: \kappa\}] \text{ in} \\
\text{let } \mathcal{K}_2 = \mu\kappa'.[\mathtt{string} \to \mathtt{unit} \mid \forall\delta.\delta\ \mathtt{array} \to \mathtt{unit}\ \{\delta \to \mathtt{unit} <: \kappa'\}] \text{ in} \\
\mathcal{K}_1 \oplus \mathcal{K}_2 = \mu\kappa.[\mathtt{int} \to \mathtt{unit} \mid \forall\delta.\delta\ \mathtt{list} \to \mathtt{unit}\ \{\delta \to \mathtt{unit} <: \kappa\}] \\
\qquad\qquad\qquad \mid \mathtt{string} \to \mathtt{unit} \mid \forall\delta.\delta\ \mathtt{array} \to \mathtt{unit}\ \{\delta \to \mathtt{unit} <: \kappa\}]
\end{array}$$

We split the typing rules for generic bindings also, as the modifications to the normal typing rules (Figure 11.1). The type case inclusion `include x` is only permitted if `x` has a pattern type constraint or μ-abstracted pattern type constraint with more than one patterns (rules (kt-include)

$$\textbf{(kt-include)}\ \frac{\Gamma(\mathtt{x}) = [k_1|\ldots|k_n] \qquad n \geq 2}{\Gamma, V \Vdash \mathtt{include\ x} : [k_1|\ldots|k_n]}$$

$$\textbf{(kt-includerec)}\ \frac{\Gamma(\mathtt{x}) = \mu\kappa.[k_1|\ldots|k_n] \qquad n \geq 2}{\Gamma, V \Vdash \mathtt{include\ x} : \mu\kappa.[k_1|\ldots|k_n]}$$

$$\textbf{(kt-typecase)}\ \frac{\Gamma, V + V' \Vdash e_t : \tau \qquad D = \mathit{Derive}(V', \emptyset, e_t) \qquad BV(\sigma) = V' \qquad FV(\sigma) = \emptyset}{\Gamma, V \Vdash (\sigma {\Rightarrow} e_t) : [\sigma\ d]}$$

$$\textbf{(kt-e-generic)}\ \frac{\begin{array}{c}\Gamma, V \Vdash tc_{ti} : \mathcal{K}_i \qquad \mathcal{K} = \oplus_{i=1}^{n} \mathcal{K}_i \\ \mathit{adm}^{+}(\emptyset, \mathcal{K}) = \mathit{true} \qquad \Gamma\{\mathtt{x} : \mathcal{K}\}, V \Vdash e_t\end{array}}{\Gamma, V \Vdash \mathtt{generic\ x}_{\mathcal{K}}\,\mathtt{=case}\ tc_{t1}|\ldots|tc_{tn}\ \mathtt{in}\ e_t}$$

$$\textbf{(kt-e-genericrec)}\ \frac{\begin{array}{c}\text{if } tc_{ti} = \mathtt{include}\ \ldots \text{ then } \Gamma, V \Vdash tc_{ti} : \mathcal{K}_i \\ \text{else } \Gamma\{\mathtt{x} : \mu\kappa.\mathcal{K}\}, V \Vdash tc_i : \mathcal{K}_i' \ \text{ where } \ \mathcal{K}_i \xrightarrow{\mu} \mathcal{K}_i' \\ \mathcal{K} = \oplus_{i=1}^{n} \mathcal{K}_i \\ \mathcal{K} = \mathit{adm}^{+}(\emptyset, \mathcal{K}) = \mathit{true} \qquad \Gamma\{\mathtt{x} : \mathcal{K}\}, V \Vdash e_t\end{array}}{\Gamma, V \Vdash \mathtt{generic\ rec\ x}_{\mathcal{K}}\,\mathtt{=case}\ tc_{t1}|\ldots|tc_{tn}\ \mathtt{in}\ e_t}$$

Figure 11.1: Constraint typing rules with type case inclusion

and (kt-includerec)). Such generic values must be defined by the generic binding. Generic primitives and let bound generic values have always only one element in their pattern constraints (See Figure 8.2).

The typing rule for a type case (kt-typecase) returns a pattern constraint with one element. The constraint of a value which is defined by a generic binding is the constraint concatenation of the constraints of type cases and inclusions. If the generic binding is recursive, it is stored in a type environment for the type case typings with the concatenated constraints $\mathcal{K}$ of type case and inclusions. Each type case definition must have μ-expanded constraint $\mathcal{K}'_i$ of its corresponding part $\mathcal{K}_i$ in the whole constraint. Type case inclusions are typed under a type environment *without* the recursive binding. This is since a type case inclusion of the defined identifier itself has no sense. For example:

```
generic print = case
| int → unit ⇒ print_int
| float → unit ⇒ print_float
;;

generic rec print = case
| include print
| ∀δ.δ list → unit ⇒ fun l → List.iter print l
;;
```

Here, the `include print` phrase in the second `print` definition imports the type cases of the *first* `print` definition and not of the second. Since extensions may often have the same name as the original value, this exception of the recursive scoping rule for type case inclusions is required.

Example Suppose we have the following initial definition of `print`. The typing of `print` is as same as before:

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

val print : μκ.[string → unit
               |∀δ.δ → unit {δ → unit <: κ
                             & δ list → unit <: κ}]
```

We now define `print'` including `print` and adding the case for `int → unit` :

```
generic print' = case
| int → unit ⇒ print_int
| include print
;;
```

The typing of the type case inclusion `include print` is as follows.

$$\frac{\Gamma(\mathtt{print}) = \mu\kappa.[\mathtt{string} \to \mathtt{unit} \mid \forall\delta.\delta\ \mathtt{list} \to \mathtt{unit}\ \{\delta \to \mathtt{unit} <: \kappa\ \&\ \ \delta\ \mathtt{list} \to \mathtt{unit} <: \kappa\}}{\Gamma, V \Vdash \mathtt{include\ print} : \mu\kappa.[\mathtt{string} \to \mathtt{unit} \mid \forall\delta.\delta\ \mathtt{list} \to \mathtt{unit}\ \{\delta \to \mathtt{unit} <: \kappa\ \&\ \ \delta\ \mathtt{list} \to \mathtt{unit} <: \kappa\}}$$

If we let this constraint of `include print` as $\mathcal{K}_1$, the typing of `print'` is as follows:

$$\frac{\begin{array}{l} \Gamma, V \Vdash \mathtt{include\ print} : \mathcal{K}_1 \\ \Gamma, V \Vdash \mathtt{print\_int} : [\mathtt{int} \to \mathtt{unit}] \\ \begin{array}{rcl} \mathcal{K} & = & [\mathtt{int} \to \mathtt{unit}] \oplus \mathcal{K}_1 \\ & = & \mu\kappa.[\ \mathtt{int} \to \mathtt{unit} \\ & & \quad \mid \mathtt{string} \to \mathtt{unit} \\ & & \quad \mid \forall\delta.\delta\ \mathtt{list} \to \mathtt{unit}\ \{\delta \to \mathtt{unit} <: \kappa \\ & & \qquad\qquad\qquad\qquad \&\ \ \delta\ \mathtt{list} \to \mathtt{unit} <: \kappa\} \end{array} \\ adm^{+}(\emptyset, \mathcal{K}) = true \qquad \Gamma\{\mathtt{print}' : \mathcal{K}\}, V \Vdash e_t \end{array}}{\Gamma, V \Vdash \mathtt{generic\ print}'_{\mathcal{K}} = \mathtt{case\ int} \to \mathtt{unit} \Rightarrow \mathtt{print\_int} \mid \mathtt{include\ print\ in}\ e_t}$$

This constraint of `print'`, $\mathcal{K}$ is equivalent to the one for the following definition:

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

□

11.5 Compilation

11.5.1 The outline

We use the generic flow compilation scheme (argued in Chapter 9) to define compilation of the extended generic value definitions. Compilation using type pattern matching is also possible, using the same technique explained here. We can use the same generic flow data structure as before, but the compilation algorithm must be modified. Since we suppose a separate compilation environment, sources of the original type case definitions are not always available. Therefore, instead of including the original source syntactically, we make use of the compilation of the original type cases. Since recursive calls in the original definitions must now point to the new extended generic values, recursive reference of generic values is now dynamic. Therefore, we need to add a new *self* parameter to the compilation of each type case which gives a recursive call-back point for the type case.

Let's start from a compilation example using generic flows *without* any modification, then change it to something usable for the type case inclusions. The following definition of `print` and its full instance `print:`τ,

```
generic rec print = case
| int → unit ⇒ print_int
| string → unit ⇒ print_string
| ∀δ.δ list → unit ⇒ fun l → List.iter print l
;;

print:τ
```

are compiled as below:

```
let rec print =
  let v = [| fun subs → print_int;
             fun subs → print_string;
             fun subs → fun l → List.iter (print subs.(0)) l |]
  in
  function Node(p, subs) → v.(p) subs
;;

print (kinst+(∅, τ <: K_print))
```

The type cases of the generic value `print` are compiled to functions in an array `v`, which receive sub-flows `subs` and send them to the internal generic value occurrences (`print subs.(0)`, for example).

Now we add them new additional function arguments `self` to retrieve their recursive call-back points. The recursive occurrences of `print` in the type case are now replaced by this `self`:

```
let rec print = function Node(p, subs) →
  let v = [| fun self subs → print_int;
             fun self subs → print_string;
             fun self subs → fun l → List.iter (self subs.(0)) l |]
  in
  v.(p) print subs
;;
```

Now we need to apply `print` to `v.(p)` as the recursive call-back point in addition to the sub-flows.

By replacing the recursive occurrence of `print` by `self` and abstracting the sub-flows by it, now the array `v` can be independently defined of the outer function abstraction which receives a flow. This enables us to define the type case array and its invocation function separately:

```
let print = [| fun self subs → print_int;
               fun self subs → print_string;
               fun self subs → fun l → List.iter (self subs.(0)) l |]
;;

let rec invoke v (Node(p,subs)) = v.(p) (invoke v) subs;;
```

Finally, the `print` generic function is compiled to an array of the functions of its type cases. The `invoke` function receives an array of type case compilations and a generic flow. For example, a full instance of `print`, `print:`τ is now compiled to an application of this `invoke`:

$$\texttt{print:}\tau \implies \texttt{invoke print } (\mathit{kinst}^{+}(\emptyset,\ \tau <: \mathcal{K}_{\texttt{print}}))$$

Note that the definition of the generic invocation function `invoke` is now independent of `print` and any other generic value! In some sense, our new compilation scheme compiles all the generic values to one "meta-generic" function `invoke`. The definition of `invoke` looks quite like the definition of fixed point operator in ML:

```
let rec y      f x               = f     (y      f) x

let rec invoke v (Node(p,subs)) = v.(p) (invoke v) subs
```

`invoke` calls `v.(p)` instead of `v` itself, but it can be regarded as a function which calculates the fixed point which is specified by the vector `v` and the flow `Node(p,subs)`.

Now let's move to the compilation of the extended generic value `print'`:

```
generic print' = case
| float → unit ⇒ print_float
| include print
;;
```

Like `print`, now `print'` is compiled to an array, it is:

```
let print' = [| fun self subs → print_float |] ⊕ print;;
```

where $\oplus$ is the array concatenation operator. It is just the concatenation of the new and original type cases. The full instance of `print'` is compiled to the application of the generic invoker:

$$\texttt{print':}\tau \implies \texttt{invoke print' } (\mathit{kinst}^{+}(\emptyset,\ \tau <: \mathcal{K}_{\texttt{print'}}))$$

The flow creation has all the responsibility so that the proper type case can be called at each situation. This is assured by the isomorphism between the type pattern constraint of `print'` and its compiled array:

```
val print' : μκ.[ float → unit
                | int → unit
                | string → unit
                | ∀δ.δ list → unit {δ → unit <: κ} ]

print' = [| fun self subs → print_float;
            fun self subs → print_int;
            fun self subs → print_string;
            fun self subs → fun l → List.iter (self subs.(0)) l |]
```

For example, the following expression

```
# print' [1.2; 3.4; 5.6]
```

is compiled to the code

```
invoke print' (Flow (3,[| Flow(0, [||]) |])) [1.2; 3.4; 5.6]
```

whose flow correctly instructs `invoke` so that it can call the type case for $\forall\delta.\delta$ `list` → `unit` of `print'` first, then calls one for `float` → `unit` inside.

11.5.2 The algorithm

Before the formal compilation algorithm definition itself, we first have the generic invoker `invoke`. It is a normal ML polymorphic function:

```
let rec invoke v (Node(p,subs)) = v.(p) (invoke v) subs

val invoke : ∀α.((flow → α) → flow array → α) array → flow → α
```

The compilation algorithm is in Figure 11.2. The compilation function $\lfloor\cdot\rfloor_{F,G}$ takes two environments: F is a mapping from generic polymorphic instances to flows which is as same as Figure 9.5. The additional environment G is an environment from an identifier to an expression, which is used for the replacement of recursive occurrences of generic instances to their "self" identifiers.

A generic instance `x` is now compiled in three different ways depending on how it is defined: let defined values, generic defined values and recursive occurrence of generic defined values. If it is a let defined generic instance, the compilation of `x` is itself. If it is defined by a generic binding, it is compiled to `invoke x` since `x` is just an array and we need its invocation. If it is a recursive call of a value which is defined by a generic binding, we must compile it to `self` parameter which will carry the fixed point of the extended value. All these three compilations take their flow arguments, same as Figure 9.5.

Like the typing rules, we separate the compilation of generic binding into three, the compilations of normal type case, the type case inclusion and the binding itself. For the type cases and type case inclusion, the compilation function takes an additional parameter `x` which denotes that these

$$\lfloor c_\sigma : \tau \rfloor_{F,G} = c_\sigma$$
$$\lfloor \texttt{fun x} \to e_t \rfloor_{F,G} = \texttt{fun x} \to \lfloor e_t \rfloor_{F,G}$$
$$\lfloor e_{t1}\ e_{t2} \rfloor_{F,G} = \lfloor e_{t1} \rfloor_{F,G}\ \lfloor e_{t2} \rfloor_{F,G}$$
$$\lfloor e_{t1}, e_{t2} \rfloor_{F,G} = \lfloor e_{t1} \rfloor_{F,G},\ \lfloor e_{t2} \rfloor_{F,G}$$

$$\begin{array}{l}
\lfloor \texttt{x}_{\mathcal{K}} : \tau \rfloor_{F,G} = \\
\quad \text{if } \mathit{generic}(\mathcal{K}) = \mathit{false} \text{ then } \texttt{x} \\
\quad \text{else let } f = \text{if } \texttt{x} \text{ is bound in } F \text{ then } F(\texttt{x}) \text{ else } \mathit{kinst}^+(\emptyset, \tau <: \mathcal{K}) \\
\qquad \text{in } G(\texttt{x})\ f
\end{array}$$

$$\begin{array}{l}
\lfloor \texttt{let (rec) x}_{V,\mathcal{K}} \texttt{=} e_{t1} \texttt{ in } e_{t2} \rfloor_{F,G} = \\
\quad \text{let } [\texttt{x}^0_{\mathcal{K}_0} : \tau_0;\ \ldots;\ \texttt{x}^n_{\mathcal{K}_n} : \tau_n] = \mathit{plist}(V, \emptyset, e_t) \\
\quad \text{let } G' = G\{\texttt{x} \Rightarrow \texttt{x}\} \\
\quad \text{let } G'' = \text{if the binding is recursive then } G' \text{ else } G \\
\quad \text{in } \texttt{let (rec) x = fun (Node(_, subs))} \to \lfloor e_{t1} \rfloor_{F\{\texttt{x}^i \Rightarrow \texttt{subs}.(i)\},G''} \texttt{ in } \lfloor e_{t2} \rfloor_{F,G'}
\end{array}$$

$$\begin{array}{l}
\lfloor \sigma \Rightarrow e_t \rfloor_{F,G,\texttt{x}} = \\
\quad \text{let } [\texttt{x}^0_{\mathcal{K}_0} : \tau_0;\ \ldots;\ \texttt{x}^n_{\mathcal{K}_n} : \tau_n] = \mathit{plist}(\mathit{BDV}(\sigma), \emptyset, e_t) \\
\quad \text{let } F' = F\{\texttt{x}^i \Rightarrow \texttt{subs}.(i)\} \\
\quad \text{let } G' = G\{\texttt{x} \Rightarrow \texttt{self}_{\mathit{fresh}}\} \text{ where } \texttt{self}_{\mathit{fresh}} \text{ is a fresh variable} \\
\quad \text{in } \texttt{[| fun self}_{\mathit{fresh}} \texttt{ subs} \to \lfloor e_t \rfloor_{F',G'} \texttt{ |]}
\end{array}$$

$$\lfloor \texttt{include y} \rfloor_{F,G,\texttt{x}} = \texttt{y}$$

$$\begin{array}{l}
\lfloor \texttt{generic (rec) x}_{\mathcal{K}} \texttt{ = case } \sigma_1 \Rightarrow e_{t1} \mid \ldots \mid \sigma_n \Rightarrow e_{tn} \texttt{ in } e_t \rfloor_{F,G} = \\
\quad \text{let } G' = G\{\texttt{x} \Rightarrow \texttt{invoke x}\} \text{ in} \\
\quad \texttt{let x = } \lfloor tc_1 \rfloor_{F,G,\texttt{x}} \oplus \ldots \oplus \lfloor tc_n \rfloor_{F,G,\texttt{x}} \texttt{ in } \lfloor e_t \rfloor_{F,G'}
\end{array}$$

Figure 11.2: The compilation algorithm with generic flows

phrases belong to the definition of `x`. The type case $\sigma \Rightarrow e_t$ is compiled to an array with one element of a function which takes the fixed point $\texttt{self}_{fresh}$ and a flow array `subs` for the internal occurrences of generic polymorphic instances. The type case inclusion `include y` is just compiled to `y`, the array of the type case compilations of `y`. The generic binding itself is compiled to the concatenation of the compilations of these type cases and inclusions.

11.6 Removal of duplicated type cases

Our extensions of generic values using the type case inclusions are closed, that is, the extended values are independent of the originals. Different extensions for the same initial generic value definition creates extensions branches. Contrary, since more than one type case inclusions are permitted to appear in one generic binding, we can merge these extension branches to one generic value. Therefore the extension relationship can form no longer simple "chains", but complex "extension graphs" as follows in Figure 11.3 for example.

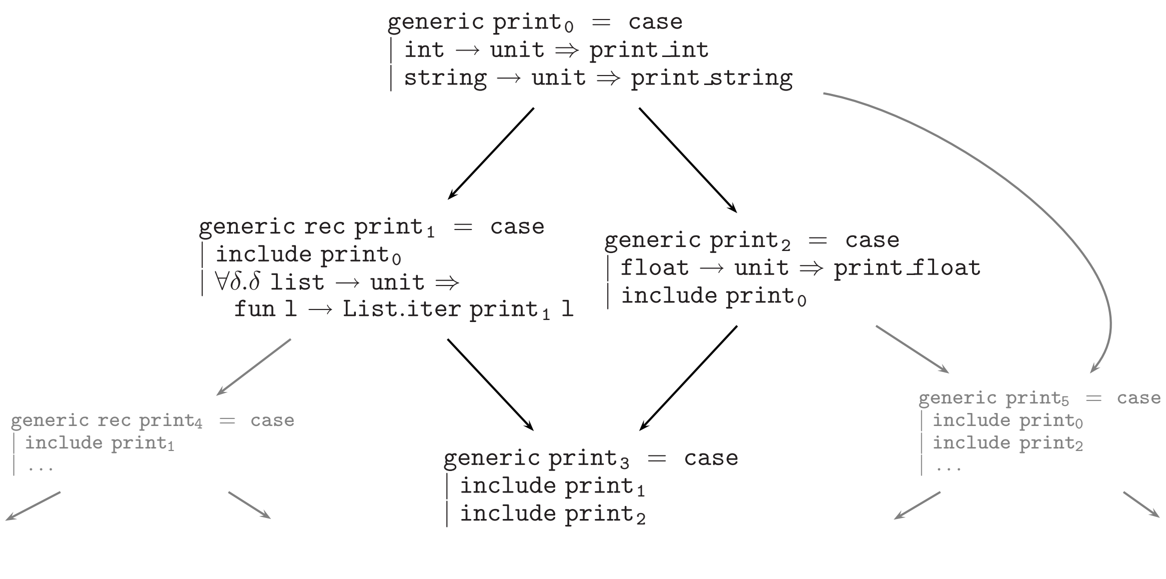

Figure 11.3: An extension graph of `print`

Starting from the initial definition of $\texttt{print}_0$, here we defined from $\texttt{print}_1$ to $\texttt{print}_6$. In the definition of $\texttt{print}_3$, we merge two extension branches. If we follow the extension compilation algorithm defined in this chapter, the semantics of $\texttt{print}_3$ must be equivalent to the following definition:

```
generic rec print'₃ = case
| int → unit ⇒ print_int
| string → unit ⇒ print_string
| ∀δ.δ list → unit ⇒
    fun l → List.iter print'₃ l
| float → unit ⇒ print_float
| int → unit ⇒ print_int
```

```
| string → unit ⇒ print_string
;;
```

Corresponding with this `print'`$_3$, the compilation of `print`$_3$ is:

```
let print3 = [| fun self subs → print_int;
                fun self subs → print_string;
                fun self subs → fun l → List.iter (self subs.(0)) l;
                fun self subs → print_float;
                fun self subs → print_int;
                fun self subs → print_string |]
;;
```

The last two type cases of `print'`$_3$ and their compilations in the array `print`$_3$ are clearly redundant, since the same type case definitions exist already before. These duplicated type cases are created by the merge of two branches which starts from the same generic definition. Since the secondary and latter occurrences of the same type cases are never called, they should be removed from the constraints and compilations for simplicity. The duplications of type cases are easily detected by scanning type pattern constraints, because they have the same type pattern and derived constraints as their first occurrences[1]. By this *removal of duplicated type cases*, we can avoid meaningless type and code explosions. In the above example, by removing the last two type cases, the constraint of `print`$_3$ should become as follows:

```
val print3 : μκ.[ int → unit
                | string → unit
                | ∀δ.δ list → unit { δ → unit <: κ }
                | float → unit ]
```

and its compilation must be optimized as below:

```
let print3 = [| fun self subs → print_int;
                fun self subs → print_string;
                fun self subs → fun l → List.iter (self subs.(0)) l;
                fun self subs → print_float |]
;;
```

11.7 Conclusion

We added user extensibility of generic values to extensional polymorphism, based on the closed world assumption. Semantics of generic values are closed to extensions, in the sense that extended

[1]This optimization also removes type cases which are not duplicated, but have no possibility of call. For example, if we have two different type cases `string → unit ⇒ print_string` and `string → unit ⇒ fun _ → ()` in one generic definition, the latter will be removed, since they have the same type pattern with the same derived constraint (which is null).

values are differently defined from the originals. But our extension is not just redefinition of values, since it supports open recursion: semantics of recursive calls of the original value are opened to newly added type cases. This “open recursion under closed world” allows us to extend recursive generic values which can perform recursive calls of both original and extended type cases.

Type cases are compiled to functions with an additional “self” parameter for fixed point passing. This abstraction of recursive occurrences of generic values separates the compilation of a user generic value into two: the array of its type case compilations and one unique generic invocation function which governs the type case selection of the all user-defined generic values.

Open world extension in extensional polymorphism: future work ?

The open world extension was not adopted due to its non-conformance to the ML semantics, but this does not mean that it is technically impossible in extensional polymorphism. Typing will not be so complicated: generic full instances can be typed under the available constraint information at their occurrences. Their soundness will be preserved even with later additional type cases, if we use the type case selection semantics with backtracking introduced in Chapter 9. The compilation of type cases can be as same as the closed world extension, functions with the self parameters. The generic type case invocation function `invoke` is also usable. The difficult part will be in extension syntax and generic flow compilation. Since the semantics of user generics are dependent on the definition order of type cases, there must be a syntactic specification of the insertion position of extended type cases to the current list of type cases. But it is not well studied yet. In the compilation, the semantics of user generic values cannot be determined until all the program compilation units are available. Therefore the generic flow computations for full instances must be performed at program link time, though their type soundness can be proven at typing of each compilation unit.