# Chapter 9

# Generic flow analysis

## 9.1 Introduction

### Generic flows

In the last chapter we have developed generic constraint typing system which assures the type soundness of generic values. Using generic constraint, non-exhaustiveness information, generic instances are now checked all *statically*. Here arises an idea: normal ML type checking also statically checks type soundness of values. Thanks to this soundness, programs need no types and type checking at run time, and gains evaluation efficiency. Then, if generic constraint typing assures the type soundness of generic instances, is it possible to remove type checking of generic instances from run time and make generic instance evaluation more efficient? This is the main motivation of this chapter.

Run time type checking of generic instances are done by matching dispatched run time types in compilation of generic bindings. But we cannot simply remove it, since it has another functionality: invocation of type cases to be evaluated. We must extract the essential core for the type case invocation from the run time type matching which is now redundant. This is done by using well-typing proof of each generic full instance at evaluation. Since valid full instance relationship is proven by simulating actual type case selections at run time, therefore if a full instance is proven valid, then we have the complete information of its type case selections: which type case is evaluated for each type matching. We call this flow information of the type case selection *generic flow*, which is basically an integer graph whose node indicates the selected type case of each selection. Dispatching generic flows instead of run time types, type case selection by pattern matching is throughly simplified to an access to an array of type methods using the index number of the top node of given generic flow, and sub graphs of the flow are dispatched to the corresponding polymorphic instance compilations. This dramatically improves the efficiency of type case selection. Selections are performed in constant time, whatever numbers of type patterns and how complex they are.

**New type case selection with backtracking**

Using generic constraint dispatching, type checking and case selection of generic instances are done at static typing. This means that we are now free from restrictions of type case selection semantics at run time, which was tightly combined with the compilation scheme of their compilation as pattern matching.

That is, under the semantics using type pattern matching it sometimes happens that a generic instance can be rejected as invalid even though it has an appropriate case selection combination under its type context. This is due to the semantics of pattern matching: type pattern matching always chose the first match, even though internal generic instances in the corresponding type method may not be valid. Even if there is a later matched type method whose internal generic instances are all proven valid, it cannot be selected. Backtracking of type case selection is too expensive to perform at run time.

But once type case selection is done at static typing, and its results, generic flows are dispatched to generic instances, we are no longer bound to this run time restriction of type case selections. We can do backtracking in static search of valid type case selections. Not only saving the generic instances which have appropriate type case selection combinations but impossible to be evaluated in the classical pattern match based semantics, the new type case selection with backtracking is also upward compatible with the classical semantics.

**Outline of this chapter**

We continue our discussion as follows: rather than defining generic flow compilation, in Section 9.2, we first observe the semantic problem of pattern match based type case selection semantics closely. Then in Section 9.3, we propose new semantics with backtracking and corresponding modifications of constraint type system to overcome this difficulty of type pattern matching. After the modifications of semantics and typing are done, in Section 9.4, we introduce generic flows — encoded proofs of constraint typing — and show how we can make use of them for new compilation. We also note that how it significantly boosts up run time type case selection.

## 9.2 The simple pattern matching is shortsighted

Run time type matching selects type cases to be executed based on normal ML pattern match. If there are more than one matched patterns, the first match is always used. Using this ordering of type matching, we can detail behavior of generic values. For instance, the following `print'` generic function usually calls `print`, but if a given argument is an integer, then its special printing function `special_print_int` is called instead:

```
generic rec print' = case
| int → unit ⇒ special_print_int  (* Special!! *)
| ∀δ.δ → unit ⇒ print             (* The other *)
;;
```

The type case for `int` $\rightarrow$ `unit` must precede the other. Otherwise `special_print_int` would never be called, because the pattern $\forall\delta.\delta \rightarrow$ `unit` is more general than `int` $\rightarrow$ `unit` and instance type `int` $\rightarrow$ `unit` would match with $\forall\delta.\delta \rightarrow$ `unit` first. Thus, the semantics of type case expressions are easy to understand and use for the programmers who are familiar with ML pattern matching. In addition, by compiling run time type matching to normal pattern matching, type case selection, flow control of user generics can be efficient as we saw in Chapter 3.

However, this pattern match based semantics collides with problems when polymorphic instances occur inside type methods. Let's think about a generic function which completely flattens lists of any depth to depth one:

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

Here, `List.flatten` is the normal list flattening function whose definition is

```
val flatten : ∀α.α list list → α list
let rec flatten = function
| [] → []
| x::xs → x @ flatten xs
;;
```

And for example:

```
# (flat [[[1]]; [[2;3]]; [[4;5;6]; [7;8;9;10]]] : int list)
- : int list = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
```

This function `flat` works fine for complete list flattening to depth one. Then, why not trying to stop flattening at some point? This seems to be done by giving a different type to an instance of `flat`... But this does not work:

```
# (flat [[[1]]; [[2;3]]; [[4;5;6]; [7;8;9;10]]] : int list list)
Error: generic value flat cannot be instantiated to
         int list list list → int list list
```

More simply,

```
# (flat [[1;2;3]] : int list list);;
Error: generic value flat cannot be instantiated to
         int list list → int list list
```

In this example, `flat` was instantiated to the type `int list list` $\rightarrow$ `int list list` and we expected it would return `[[1;2;3]]` using the second type case. But in the pattern match based semantics, the function never works as we expected. Note that the type `int list list` $\rightarrow$ `int list list` matches with both of type patterns. The first type case matches with the type first, therefore constraint type system checks validity of the polymorphic instance of the first method under the type `int list` $\rightarrow$ `int list list`. Because there is no type pattern of `flat` which matches with this type, type checker rejects the instance.

This is because our semantics lacks *backtracking*. When type matching selects one of type cases to be evaluated, it only checks type patterns, and never look into polymorphic instance validity inside methods. Once it choose one of type methods just from type patterns, it checks its internal polymorphic instances' validity. If one of polymorphic instances is invalid, it is reported by type checker as an error immediately, without trying other following type cases. As far as we keep using this "shortsighted" semantics, this expression cannot be well-typed nor evaluated, even though the definition of `flat` has the enough set of type cases for the correct evaluation.

For a quick fix, we can reorder the type cases of `flat` and let the case $\forall\delta.\delta$ `list` $\rightarrow \delta$ `list` can be selected when the both pattern match:

```
generic rec flat2 = case
| ∀δ.δ list → δ list ⇒ fun l → l
| ∀δ.δ list list → δ' list ⇒ fun l → flat2 (List.flatten l)
;;
```

Thanks to the reordering of the type cases, `flat2` can flatten lists to any level. But this type case reordering does not work in general:

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

generic print_list = case
| ∀δ.δ list → unit ⇒ fun l → List.iter print l
| ∀δ.δ → unit ⇒ fun x → print_string "sorry, unprintable"
;;
```

In the above code, the second generic function `print_list` is written with an intension that it could print lists of data types (`int` and `string`) which `print` can handle and print an error message for the other types. It has the following generic constraint:

$$\begin{aligned}\mathcal{K}_{\texttt{print\_list}} = \ & [\ \forall\delta.\delta\ \texttt{list} \rightarrow \texttt{unit}\ \{\ \delta \rightarrow \texttt{unit} <: \mathcal{K}_{\texttt{print}}\}\\ & \mid \forall\delta.\delta\ \ \rightarrow \texttt{unit}\ ]\end{aligned}$$

And it works for some types:

```
# print_list [1;2];;
12
```

```
- : unit = ()
# print_list ["hello "; "world"];;
hello world
- : unit = ()
# print_list 3.14;;
sorry, unprintable
- : unit = ()
```

But once we try to print unsupported lists, except printing the error message, the instance of `print_list` is rejected by type checker:

```
# print_list [1.2; 3.4; 5.6];;
Error: generic value print_list cannot be instantiated to float list → unit
```

The application of the function `print_list` is never well-typed, even though the second type case as the default can handle this case perfectly. The instance type `float list` $\rightarrow$ `unit` matches with the first type pattern, then type checker checks the internal instantiated derived constraint `float` $\rightarrow$ `unit` $<: \mathcal{K}_{\texttt{print}}$. When it finds the internal use of `print` is not valid, it is too late to go back and try the other type case for $\forall\delta.\delta \rightarrow$ `unit`. In this case, reordering of type cases cannot help us, since the type pattern for the error printing is more general than the other, therefore reordered `print_list` would print the error message for any instantiation.

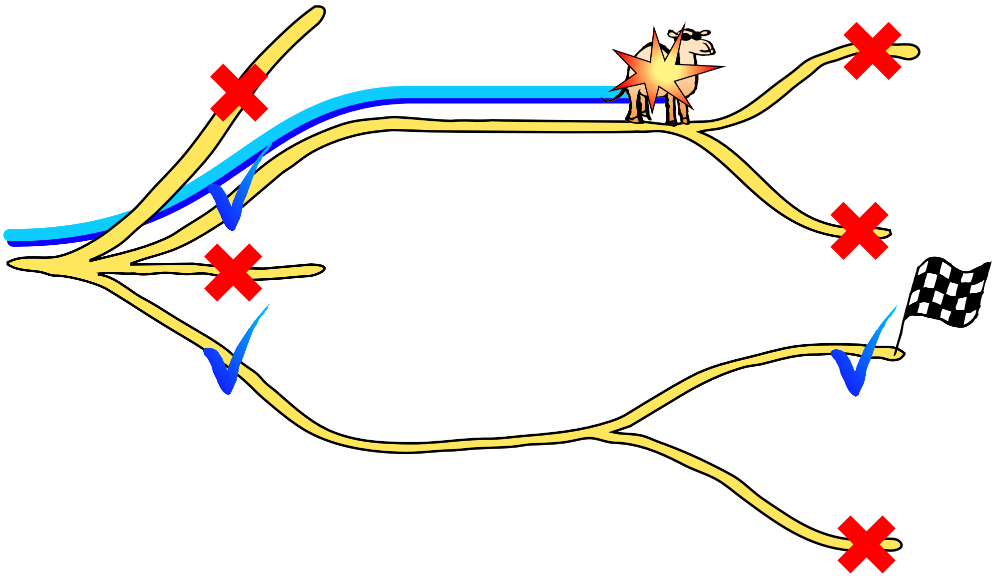
The core of this problem is not in ordering of type cases, but case selection strategy of the semantics of user generics itself. Generally, when a generic definition have two type patterns which share some instance type which match $\tau$, and semantics for $\tau$ is not defined in the first matched type case but in the second, the run time type matching can never make use of the second match. The cartoon of Figure 9.1 explains this in a more fancy way. Not to be trapped in type cases which look good (= matched) at a glance but actually have no way of evaluation inside, we need *backtracking* search of type case selection in order to investigate all the further type case sub-selections.

## 9.3 New generic type case selection with backtracking

We modify the semantic and typing rules of generic type case selections so that now type case selection can avoid type cases only with dead ends using backtracking. To distinguish the new semantic and typing systems from the existing ones, we call the previous systems as *"type match based semantics/typing"*. The system we are constructing will be referred as *"the newer system with backtracking"*.

### 9.3.1 Evaluation rules

Before, the type case selections were done only by checking whether a given evaluation type context $\tau$ is an instance of some type pattern $\sigma_i$, i.e. $\tau \leq \sigma_i$. Now we also look further inside the derived constraints $D_i$ of the selected pattern constraint. This is easily achieved if we ignore its compilation method (we will discuss it later in Section 9.4), by using the predicate $<:$ defined in Section 8.7.2 instead of $\leq$: we test $\tau <: \sigma_i\ D_i$. Because generic closures store no constraint information, we use



Optimistic Joe the camel always takes the first continuing branch in the desert. He often finds himself at dead ends, even when there is a goal in another route. (He is stopped by his type checker before his departure, fortunately.)

Figure 9.1: Ignorance of the future awaits...

constraints annotated to identifiers $\mathtt{x}_{\mathcal{K}}$. The semantic typing which is also modified later will assure that these constraints coincide with those for generic closures. The new modified evaluation rules are in Figure 9.2. Now type cases are selected not only by matching their pattern type schemes, but also by looking their derived constraints. Therefore evaluation rules never select type cases in which internal case selection fails.

### 9.3.2 Constraint typing

Some generic full instances which were evaluated to the generic run time type *gerr* in the type match based semantics are now evaluated successfully in the newer semantics with backtracking. Therefore, we also need to make them granted as well-typed by extending the definition of valid full instances. Here, we omit the proofs of the theorems in this new constraint typing system. All of them can be easily obtained by rewriting the corresponding proofs of the pattern match based system (see Chapter 8).

**Full instance relationship** $<:$

The main modification in type system is redefinition of the full valid instance relationship $<:$. Rule (k-fail-case) is first modified. This rule decided that $\tau$ was not a full instance of a pattern constraint list $[\sigma_1\ D_1|\ldots|\sigma_n\ D_n]$, if it found a type pattern $\sigma_i$ which matched with $\tau$, but $\tau$ was not a full

$$\textsf{(e-gcase)}\ \frac{\mathcal{K} = [\sigma_1\ D_1 \mid \ldots \mid \sigma_n\ D_n] \quad \tau \not<: \sigma_1\ D_1 \quad \ldots \quad \tau \not<: \sigma_{p-1}\ D_{p-1} \quad \tau <:_S \sigma_p\ D_p \quad E' \models S(e_{tp}) \Longrightarrow r}{E\{\mathtt{x} = gcase(E', \_, [\sigma_1 \Rightarrow e_{t1}, \ldots\ \sigma_p \Rightarrow e_{tp}, \ldots, \sigma_n \Rightarrow e_{tn}])\} \models \mathtt{x}_{\mathcal{K}} : \tau \Longrightarrow r}$$

$$\textsf{(e-gcase-r)}\ \frac{\mathcal{K} = \mu\kappa.[\sigma_1\ D_1 \mid \ldots \mid \sigma_n\ D_n] \quad \mu\kappa.[\sigma_1\ D_1 \mid \ldots \mid \sigma_n\ D_n] \xrightarrow{\mu} [\sigma_1\ D'_1 \mid \ldots \mid \sigma_n\ D'_n] \quad \tau \not<: \sigma_1\ D'_1 \quad \ldots \quad \tau \not<: \sigma_{p-1}\ D'_{p-1} \quad \tau <:_S \sigma_p\ D'_p \quad E'[\mathtt{y} = gcase(E', \mathtt{y}, [\sigma_1 \Rightarrow e_{t1}, \ldots\ \sigma_p \Rightarrow e_{tp}, \ldots, \sigma_n \Rightarrow e_{tn}])] \models S(e_{tp}) \Longrightarrow r}{E\{\mathtt{x} = gcase(E', \mathtt{y}, [\sigma_1 \Rightarrow e_{t1}, \ldots\ \sigma_p \Rightarrow e_{tp}, \ldots, \sigma_n \Rightarrow e_{tn}])\} \models \mathtt{x}_{\mathcal{K}} : \tau \Longrightarrow r}$$

$$\textsf{(e-gcase-err)}\ \frac{\mathcal{K} = \sigma\ [\sigma_1\ D_1 \mid \ldots \mid \sigma_n\ D_n] \quad \forall 1 \le i \le n,\ \ \tau \not<: \sigma_i\ D_i}{E\{\mathtt{x} = gcase(E', \_, [\sigma_1 \Rightarrow e_{t1}, \ldots, \sigma_n \Rightarrow e_{tn}])\} \models \mathtt{x}_{\mathcal{K}} : \tau \Longrightarrow gerr}$$

$$\textsf{(e-gcase-r-err)}\ \frac{\mathcal{K} = \sigma\ \mu\kappa.[\sigma_1\ D_1 \mid \ldots \mid \sigma_n\ D_n] \quad \mu\kappa.[\sigma_1\ D_1 \mid \ldots \mid \sigma_n\ D_n] \xrightarrow{\mu} [\sigma_1\ D'_1 \mid \ldots \mid \sigma_n\ D'_n] \quad \forall 1 \le i \le n,\ \ \tau \not<: \sigma_i\ D'_i}{E\{\mathtt{x} = gcase(E', \mathtt{y}, [\sigma_1 \Rightarrow e_{t1}, \ldots, \sigma_n \Rightarrow e_{tn}])\} \models \mathtt{x}_{\mathcal{K}} : \tau \Longrightarrow gerr}$$

Figure 9.2: New evaluation rules

instance of $\sigma_i\ D_i$. The remained pattern constraints $\sigma_{i+1}\ D_{i+1},\ \ldots, \sigma_n\ D_n$ were never tested. Now this rule becomes more patient and checks all the type pattern constraints:

$$\textsf{(k-fail-case)}\ \frac{\forall 1 \le i \le n, \tau \not<: \sigma_i\ D_i}{\tau \not<: [\ \sigma_1\ D_1 \mid \ldots \mid \sigma_n\ D_n\ ]}$$

The other (k-fail$*$) rules do not relate with essential type case selection. Therefore they are kept as they were. The predicate $\tau <: \mathcal{K}$ is now redefined using these new rules.

The valid full instance predicate $kinst^+$ is also modified. Its case for type patterns follows the change of (k-fail-case):

$$\begin{aligned} kinst^+(M, \psi, \tau <: [\sigma_1\ D_1 | \ldots | \sigma_n\ D_n]) = \exists p,\ & kinst^+(M, \psi, \tau <: \sigma_1\ D_1) = false\ \& \\ & \ldots \\ & kinst^+(M, \psi, \tau <: \sigma_{p-1}\ D_{p-1}) = false\ \& \\ & kinst^+(M, \psi, \tau <: \sigma_p\ D_p) = true \end{aligned}$$

We can prove the correctness of $kinst^+$ with respect to new $\tau <: \mathcal{K}$, just like what we have done in the proof of Theorem 29. The same relaxed admissible constraints defined in Section 8.7.2 are used in the new system and the decidability of $<:$ and termination of $kinst^+$ are also proven for these admissible constraints. We omit these proofs, but we should prove something new:

**Lemma 33 (Upward compatibility of $kinst^+$)** *Let $\tau$ be a type expression, $\mathcal{K}$ be a relaxed admissible constraint $M$ be a memo function for $kinst^+$ and $\psi$ be a constraint substitution for $\mu$-expansion. If $kinq^+(M, \psi, \tau <: \mathcal{K}) = true$ in the type match based system, $kinq^+(M, \psi, \tau <: \mathcal{K}) = true$ also in the new system.*

PROOF: By induction over the calculation of $kinst^+$. The only modified case is for type pattern constraints. Therefore we just prove this case. Let $\mathcal{K} = [\sigma_1\ D_1| \ldots |\sigma_n\ D_n]$. Then, in the type matched base system,

$$\begin{aligned}&kinst^+(M, \psi, \tau <: [\sigma_1\ D_1| \ldots |\sigma_n\ D_n]) = \\ &\quad \exists p, \text{where } \tau \not\leq \sigma_1\ \&\ \ldots \tau \not\leq \sigma_{p-1}\ \&\ \tau \leq \sigma_p\ \&\ kinst^+(M, \psi, \tau <: \sigma_p\ D_p)\end{aligned}$$

Since for all $1 \leq i < p$, $\tau \not\leq \sigma_i$, in the new system we have $\tau \not<: \sigma_i\ D_i$, therefore

$$\forall 1 \leq i < p,\ kinst^+(M, \psi, \tau <: \sigma_i\ D_i) = false$$

From the induction hypothesis for the calculation $kinst^+(M, \psi, \tau <: \sigma_p\ D_p) = true$ of the type match based system, we have also $kinst^+(M, \psi, \tau <: \sigma_p\ D_p) = true$ in the newer system.

From these, by the modified definition of $kinst^+$ for the case of the type pattern lists, we have

$$kinst^+(M, \psi, \tau <: [\sigma_1\ D_1| \ldots |\sigma_n\ D_n]) = true$$

in the newer system. □

**Lemma 34 (Upward compatibility of the full instance relationship)** *Let $\tau$ be a type expression, $\mathcal{K}$ be a relaxed admissible constraint. If $\tau <: \mathcal{K}$ in the type matching system, then $\tau <: \mathcal{K}$ also holds in the new system.*

PROOF: This is only the special case of Lemma 33, when $M = \emptyset$ and $\psi = \emptyset$. □

The new full instance relationship $<:$ is not only upward compatible with the original, but also has an additional good property: $<:$ is stable under any substitution. This substitution lemma helps us to define a type reconstruction algorithm of the constraint typing in Chapter 10.

**Theorem 34 (Substitution lemma of new $<:$)** *Let $\tau$ be a type expression and $\mathcal{K}$ be an admissible constraint such that $\tau <: \mathcal{K}$. Then, for any substitution $S$, $S(\tau) <: S(\mathcal{K})$.*

PROOF: From the admissibility of $\mathcal{K}$ and the correctness of $kinst^+$, the theorem is rewritten as follows:

$$\text{if } kinst^+(\emptyset, \emptyset, \tau <: \mathcal{K}) = true \text{ then } kinst^+(\emptyset, \emptyset, S(\tau) <: S(\mathcal{K})) = true$$

We prove its contraposition:

$$\text{if } kinst^+(\emptyset, \emptyset, S(\tau) <: S(\mathcal{K})) = false \text{ then } kinst^+(\emptyset, \emptyset, \tau <: \mathcal{K}) = false$$

which is equal to

if $S(\tau) \not<: S(\mathcal{K})$ then $\tau \not<: \mathcal{K}$

by induction over the proof of $\tau \not<: \mathcal{K}$.

**(k-fail-case)** Let $\mathcal{K} = [\sigma_i\ D_i]_{i=1}^n$. From $S(\tau) \not<: S(\mathcal{K})$, we have

$$\frac{\forall 1 \le i \le n, S(\tau) \not<: S(\sigma_i\ D_i)}{S(\tau) \not<: S([\sigma_i\ D_i])}$$

Using the induction hypothesis against the premises, we get $\tau \not<: \sigma_i\ D_i$. From these, using (k-fail-case), we can prove $\tau \not<: [\sigma_i\ D_i]$.

**(k-fail-rec)** Let $\mathcal{K} = \mu\kappa.\mathcal{K}'$. From $S(\tau) \not<: S(\mathcal{K})$, we have

$$\frac{S(\tau) \not<: S(\{\kappa \Rightarrow \mu\kappa.\mathcal{K}'\}\mathcal{K}')}{S(\tau) \not<: S(\mu\kappa.\mathcal{K}')}$$

Using the induction hypothesis against the premise, we get $\tau \not<: \{\kappa \Rightarrow \mu\kappa.\mathcal{K}'\}\mathcal{K}'$. From this, using (k-fail-rec), we can prove $\tau \not<: \mu\kappa.\mathcal{K}'$.

**(k-fail-derive)** Let the predicate be $\tau \not<: \sigma\ \{\tau_i <: \mathcal{K}_i\}_{i=1}^n$. From $S(\tau) \not<: S(\sigma\ \{\tau_i <: \mathcal{K}_i\})$, we have

$$\frac{S(\tau) \not\le S(\sigma), \quad \text{or} \quad \exists 1 \le p \le n,\ S'(S(\tau_p)) \not<: S'(S(\mathcal{K}_p)) \text{ where } S(\tau) \le_{S'} S(\sigma)}{S(\tau) \not<: S(\sigma\ \{\tau_i <: \mathcal{K}_i\})}$$

By renaming bound variables in $\sigma$, we can suppose that $BV(sigma)$ is out of the reach of $S$: $BV(\sigma) \cap Dom(S) = \emptyset$ and $\forall \nu \in Dom(S),\ FV(S(\nu)) \cap BV(\sigma) = \emptyset$.

From the contraposition of Lemma 4, if $S(\tau) \not\le S(\sigma)$, then $\tau \not\le \sigma$ and using (k-fail-derive), we have $\tau \not<: \sigma\ \{\tau_i <: \mathcal{K}_i\}$.

Otherwise, when $S(\tau) \le_{S'} S(\sigma)$, we have two cases: $\tau \le \sigma$ or not. If $\tau \not\le \sigma$, we have immediately $\tau \not<: \sigma\ \{\tau_i <: \mathcal{K}_i\}$ from (k-fail-derive). If $\tau \le \sigma$, let $S''$ be a substitution such that $\tau \le_{S''} \sigma$. From Lemma 4, $S(\tau) \le_{S'} S(\sigma)$ and $S' = (S \circ S'')|_{BV(\sigma)}$.

If $\nu \notin Dom(S)$ and $\nu \notin BV(\sigma)$, $S'(S(\nu)) = (S \circ S'')|_{BV(\sigma)}(\nu) = \nu$.

If $\nu \notin Dom(S)$ and $\nu \in BV(\sigma)$, $S'(S(\nu)) = (S \circ S'')|_{BV(\sigma)}(\nu) = S \circ S''(\nu)$.

if $\nu \in Dom(S)$, then from $\forall \nu' \in Dom(S),\ FV(S(\nu')) \cap BV(\sigma) = \emptyset$, we have $S'(S(\nu)) = S(\nu)$

$S'(S(\tau_p)) \not<: S'(S(\mathcal{K}_p))$

$S''(\tau_p) \not<: S''(\mathcal{K}_p)$

From this, using (k-fail-derive), we can prove $\tau \not<: \sigma\ \{\tau_i <: \mathcal{K}_i\}$.

**(k-fail-free)** Let $\mathcal{K} = \kappa$, then $\tau \not<: \kappa$ for any type expression $\tau$.

□

### Typing rules and soundness

For the newer semantics with backtracking, we use the same generic constraint typing rules as the type match based system. Of course the rules must use the newly defined full instance relationship $<:$ and predicate $kinst^+$, and so does the semantic typing. We can prove generic soundness of the new system in the same way we have done in the type matching system:

**Theorem 35 (Generic type soundness)** *If a fully type-annotated expression $e_t$ has a typing $\Gamma \vdash e_t : \tau$ and $\Gamma, \emptyset \Vdash e_t$ under some typing environment with constraints $\Gamma$, there exists $E$ such that $E \in \Gamma$ and $E \models e_t : \tau \Longrightarrow r$, then $r$ is neither err nor gerr but a value $v$ such that $v \in \tau$.*

PROOF: We omit the proof, since it goes just like the one of Theorem 32, using the new semantics and typing rules. □

In fact, the new full instance relationship is a pure extension of the old relationship. This leads to the following theorems:

**Theorem 36 (Upward compatibility of typing)** *Let $\Gamma$ be a typing environment, $V$ be a set of type variables and $e_t$ be a type and constraint annotated GML expression. If $\Gamma, V \Vdash e_t$ in the type matching system, then $\Gamma, V \Vdash e_t$ also holds in the new system.*

PROOF: The difference of the two constraint typings are just the definition of predicate $<:$. Therefore the theorem is clear from Lemma 34. □

**Theorem 37 (Upward compatibility of evaluation rules)** *The new semantic rules evaluate and return the same result as the type match based system, when expressions are well-typed in both systems: let $\Gamma$ be a typing environment, $e_t$ be an expression and $E$ be an evaluation environment. If $\Gamma \vdash e_t$, $\Gamma, \emptyset \Vdash e_t$, $E \in \Gamma$ and $E \models e_t \Longrightarrow v$ for some semantic value $v$ in the type match based semantics and typing, then $E \models e_t \Longrightarrow v$ also in the new system.*

PROOF: By induction on the evaluation of $e_t$ under the type match based semantics. We just prove the part of generic type case selections. The other semantic rules are equivalent, therefore they are trivial.

**Generic case closures** Let $e_t = \mathtt{x}_{\mathcal{K}} : \tau$ and $\mathtt{x}$ is bound in $E$ with a generic case closure: $E(\mathtt{x}) = gcase(E', \_, [\sigma_1 \Rightarrow e_{t1}, \ldots\ \sigma_p \Rightarrow e_{tp}, \ldots, \sigma_n \Rightarrow e_{tn}])$. Then from $E \in \Gamma$, we have $E(\mathtt{x}) = \Gamma(\mathtt{x}) \in \mathcal{K}$. From the semantic typing rule (s-kgcase),

$$\frac{\begin{array}{c} \exists \Gamma, E \in \Gamma, \\ \sigma_i = Gen(\Gamma, e_{ti}) \quad D_i = Derive(BV(\sigma_i), \emptyset, e_{ti}) \quad FV(\sigma_i) = \emptyset \\ \Gamma \vdash e_{ti} \quad \Gamma, BV(\sigma_i) \Vdash e_{ti} \\ \mathcal{K} = [\ \sigma_1\ D_1 \mid \ \ldots\ \mid \sigma_n\ D_n\ ] \end{array}}{gcase(E, \_, [\sigma_1 \Rightarrow e_{t1}, \ldots, \sigma_n \Rightarrow e_{tn}]) \in \mathcal{K}}$$

$\mathcal{K}$ must take a type pattern list $[\ \sigma_1\ D_1 \mid \ldots \mid \sigma_n\ D_n\ ]$, where $D_i$'s are derived constraints.

From $\Gamma, \emptyset \Vdash \mathtt{x}_{\mathcal{K}} : \tau$, since $\mathtt{x}$ is typed under the empty set of generalized type variables, $\mathtt{x}$ must be a full instance and its constraint typing must be done by (kt-full-inst):

$$\frac{\Gamma(\mathtt{x}) = \mathcal{K} \quad DV(\tau) \subset V \quad \mathit{polyinst}(V, \mathcal{K}, \tau) = \mathit{false} \quad \tau <: \mathcal{K}}{\Gamma, V \Vdash \mathtt{x}_{\mathcal{K}} : \tau}$$

From $\tau <: \mathcal{K} = [\ \sigma_1\ D_1 \mid \ldots \mid \sigma_n\ D_n\ ]$, there is an integer $p$ such that $1 \leq p \leq n$ and

$$\tau \not\leq \sigma_1, \ldots, \tau \not\leq \sigma_{p-1}, \quad \tau <: \sigma_p\ D_p$$

Since $\tau <: \sigma_p\ D_p$, $\tau \leq \sigma_p$. Therefore in the type matching evaluation of $\mathtt{x}$, $p$-th type case must be selected by (e-gcase):

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

Now we move to the new system. Since $\forall 1 \leq i < p, \tau \not\leq \sigma_i$, $\tau \not<: \sigma_i\ D_i$. Because $\tau <: \sigma_p\ D_p$, from Lemma 34, $\tau <: \sigma_p\ D_p$ in the new system, too. Therefore $\mathtt{x}$ is evaluated by new (e-gcase):

$$\frac{\begin{array}{c}\tau \not<: \sigma_1\ D_1 \quad \ldots \quad \tau \not<: \sigma_{p-1}\ D_{p-1} \quad \tau <: \sigma_p\ D_p \quad \tau \leq_S \sigma_p \\ E' \models S(e_{tp}) \Longrightarrow v'\end{array}}{E\{\mathtt{x} = \mathit{gcase}(E', \_, [\sigma_1 \Rightarrow e_{t1}, \ldots\ \sigma_p \Rightarrow e_{tp}, \ldots, \sigma_n \Rightarrow e_{tn}])\} \models \mathtt{x} : \tau \Longrightarrow v'}$$

From the induction hypothesis for the evaluation of $S(e_{tp})$, the result $v'$ must be equal to $v$, the result in the older evaluation system.

**Recursive generic case closures** This case is proven in the same way as the non recursive case.

□

Theorem 36 and Theorem 37 assure that all the well-typed expressions in the type match based system are also well-typed, and returns the same results in the newer system with backtracking. And what we have now is something more. This is illustrated in the following example:

**Example** Now `print_list` function in Section 9.2 is successfully evaluated even for problematic cases in the type match based system. The function is stored in some evaluation environment $E$ as the following generic case closure $c$:

$$\begin{array}{ll} c = \mathit{gcase}(E, \_, & [\ \forall\delta.\delta\ \mathtt{list} \rightarrow \mathtt{unit} \Rightarrow \mathtt{fun\ l} \rightarrow \mathtt{List.iter\ print\ l} \\ & \mid \forall\delta.\delta\ \ \rightarrow \mathtt{unit} \Rightarrow \mathtt{print\_string}\ \text{"}\mathtt{sorry, unsupported}\text{"}])\end{array}$$

In the example of `print_list [1.2; 3.4; 5.6]`, this function is evaluated with the type `float list` $\rightarrow$ `unit`, using new (e-gcase):

$$\frac{\begin{array}{l}\mathcal{K}_{\texttt{print\_list}} = \begin{array}[t]{l}[\ \forall\delta.\delta\ \texttt{list} \rightarrow \texttt{unit}\ \{\ \delta \rightarrow \texttt{unit} <: \mathcal{K}_{\texttt{print}}\} \\ \mid \forall\delta.\delta\ \ \rightarrow \texttt{unit}\ ]\end{array} \\ \texttt{float list} \rightarrow \texttt{unit} \not<: \forall\delta.\delta\ \texttt{list} \rightarrow \texttt{unit}\ \{\ \delta \rightarrow \texttt{unit} <: \mathcal{K}_{\texttt{print}}\} \\ \texttt{float list} \rightarrow \texttt{unit} <: \forall\delta.\delta\ \ \rightarrow \texttt{unit} \\ \texttt{float list} \rightarrow \texttt{unit} \leq_S \forall\delta.\delta\ \ \rightarrow \texttt{unit}\ \text{where}\ S = \{\delta \Rightarrow \texttt{float}\} \\ E \models S(\texttt{print\_string}\ \texttt{"sorry, unsupported"}) \Longrightarrow r\end{array}}{E\{\texttt{print\_list} = c\} \models \mathtt{x}_{\mathcal{K}} : \tau \Longrightarrow r}$$

```
# print_list [1.2; 3.4; 5.6];;
sorry, unprintable
- : unit = ()
```

Unlike the previous type match based evaluation, the new rule finds out that there is no possibility of successful internal type case selection of the first case, since `print` is not defined for the type `float` $\rightarrow$ `unit`, but it does not giving up the search. It then backtracks to check the second case. □

Thus, the newer system has a quite good property: not only it includes the older one completely, but in addition it makes some ill typed examples in the older system well-typed and possible to be evaluated successfully.

The only problem of the newer system is that the run time type pattern matching compilation does no longer work for it, or it is really difficult. If the newer system would lack any efficient evaluation strategy, its advantages would be meaningless. In the next section we will argue a new compilation method of generic type case selection.

## 9.4 Generic flow compilation

The type match based semantics and constraint typing of generic values were designed so that they could follow their compilation based on pattern matching. Now this direction becomes opposite. We modified the semantics and typing, and now fix the compilation to follow these modifications. In this section, first we try adding quick fixes to the pattern match compilation. But soon we will find a crucial limitation of type pattern matching: it is too slow to perform backtracking at run time. An entirely new technique must be adapted to provide the efficient compilation, using generic flows, encoded proofs of validity of generic instances.

### 9.4.1 Dynamic type case selection and new semantics

#### Backtracking

To make compilation match with the new semantics, we could introduce a *backtracking* of generic type cases like Prolog: if we would find a dead end during type case selection, then we could back to the previous selection and try another type case. But this method is not appropriate. First of all, this evaluation model is far from ML style and it is impossible to live together with the semantics if they contain side effects. In addition, even though type matchings are compiled to normal pattern matching efficiently, repeating pattern matchings caused by backtracking should strike the speed of type case selections seriously. In the desert of extensional polymorphism, the backtracking at run time takes too much cost and we must die from thirst while seeking a goal.

### Pattern guards

Instead of the real backtracking, our pattern matching compilation could choose a type case branch confirming that it never ends in dead ends for each type matching. This further check at each type matching can be realized as *pattern guards* of normal pattern matching. If we write the definition of `flat` using *type pattern guards* (or just guards in short) the new semantics can be understood easier:

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

Now it is explicitly written that the first type case is chosen if and only if its internal generic value occurrence `flat` is well-typed in the given evaluation type context: if `flat` is used with a type $\tau$ `list list` $\rightarrow \tau'$ `list` for some $\tau$ and $\tau'$, we proceed to the first case only when `flat` : $\tau$ `list` $\rightarrow \tau'$ `list` is a valid instance. These pattern guards are equivalent to derived constraints of type cases which type checker can provide automatically, therefore we need not to write them by ourselves. Type pattern guards are compiled to guards of normal pattern matching. For example, the function `flat` is intuitively compiled to the following function with a pattern match with the compiled guard:

```
let rec flat = fun t t' →
  match t, t' with
  | d list , d' when (d list → d' list <: K_flat) →
    fun l → flat «d list → d' list» (List.flatten l)
  | d, d → (fun l → l)
;;
```

By examining the additional guards for each run time type matching, we can mimic the type case selection semantics with backtracking, without using real backtracking. Checking a guard at each match is like we check whether our way is correct or not by opening our map for each branch in desert. This compilation is as inefficient as the backtracking or even worse. Roughly speaking both backtracking and type pattern guards do the same thing: dynamic typing of internal generic value occurrences. But if we use the type pattern guards, the same dynamic typing can happen repeatedly. For example, suppose `flat` is evaluated under the type `int list`$^n$ $\rightarrow$ `int list`, where `int list`$^n$ means the integer list with depth $n$. At the first call of `flat`, we check that `int list`$^{n-1}$ $\rightarrow$ `int list` is a valid instance of the constraint of `flat`, $\mathcal{K}_{\texttt{flat}}$. From the definition of <: and *kinst*$^+$, this check requires sub-checks whether types $\texttt{int list}^{n-2} \rightarrow \texttt{int list}$, ..., $\texttt{int list}^1 \rightarrow \texttt{int list}$ are valid instances of `flat`. In the recursive call of `flat` with the type $\texttt{int list}^{n-1} \rightarrow \texttt{int list}$ the compilation checks again the validity of $\texttt{int list}^{n-2} \rightarrow \texttt{int list}$ with $\mathcal{K}_{\texttt{flat}}$, which needs $\texttt{int list}^i \rightarrow \texttt{int list} <: \mathcal{K}_{\texttt{flat}}$ for $1 \leq i \leq n-3$, which all have already been checked previously. Thus the same generic full instance checks are performed repeatedly, and the number of the total checks is quadratic to $n$. We cannot afford this inefficiency.

#### Dynamic selection should be avoided

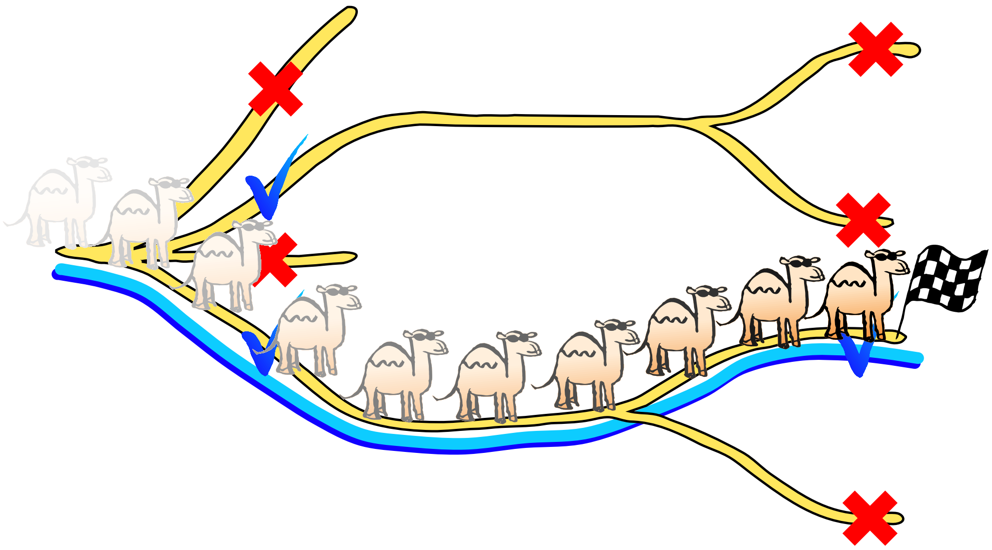
As we have seen, the computation complexity of type case selection with backtracking using the run time type pattern matching is too high to be performed. In the compilation using pattern guards, all the validity checks happen in the recursive calls of generic value have been actually done in its first call, but they are repeatedly executed. This, however, gives us an important hint: if all the validity checks in recursive calls of generic values are already checked before, why not just use the result, instead of rechecking again? This leads to a new compilation method using *generic flows* which are encoded results of instance validity checks.

### 9.4.2 Run time type checking is now redundant

We saw that in the compilation using pattern guards, compilation checks the same validity of full instances repeatedly. It can avoid these redundant checks by carrying results of the first instance check. Here arises a natural and important question: when does "the first check" take occur? It is not performed in the run time, but in the constraint typing phase! By the constraint typing system, we verify the type soundness of the generic instances statically. Therefore the run time type checking by pattern matching of generic value evaluations is just redundant. Under the complete type soundness, the only meaning of the type matching is for the flow control of type cases, selection and invocation of type methods to be evaluated. Since the static checking of full instances by algorithm *kinst*$^+$ is very pre-computation of this flow control, for well-typed expressions we know which type case will be selected in each type matching at type checking. Using this *generic flow* information, a kind of log of instance validity proof, instead of run time type matching, we can improve efficiency of the flow control of type cases. Type case selections are now simplified nothing but to accessing of arrays which contain type method codes by integer indices supplied from case selection pre-computation. If we might use the desert analogy again here, this could be a "future travel log" written by a prophet which instructs you which way you should choose at each branch, like "turn right at the first branch, then choose left at the next..." and so on (Figure 9.3). This type case selection by the pre-computed "paths" improves greatly the efficiency of generic value evaluation.

### 9.4.3 Generic flows

Now what are passed to compilation of user-defined generic values are no longer run time types, but generic flows. To define these generic flows simpler, at this moment we *ignore* the existence of generic primitives, which essentially requires run time types; we still need run time type dispatches for them. We will consider the integration of the generic primitives to the generic flow compilation later in Section 9.5.



A prophet of the desert made an instruction of the correct route to the oasis. This time optimistic Joe can travel safely to the goal. He can go even faster, since Joe can just follow the instruction and need not to choose the route by himself.

Figure 9.3: Generic flow as run time oracle

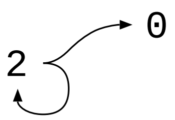
A *generic flow* is a directed integer graph which is basically a tree but which may have some loops. The formal definition of generic flows are given as follows, using ML type declaration:

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

Each node of a generic flow corresponds with each generic type case selection and indicates which type case should be selected. Each sub-tree of a node is the flow for a polymorphic generic instance which occurs in the selected type method. For example, suppose there is a `print` generic function with 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
;;
```

In the expression of `print [1;2;3]`, `print` has the type `int list` → `unit` and will have the following generic flow:




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

The generic flow compilation translates a generic instance to an implicit application of the compilation of the generic value to the corresponding generic flow instead of run time types. A generic binding is compiled to a function which takes a flow, then choose one of the compiled type methods stored in an array according to the index of the top node of the flow. Sub-flows are dispatched to internal polymorphic instances for internal flow controls.

## 9.4.4 Generic flow compilation

### Generic flow encoding

Generic flows which are encoded proofs of full instance validity are obtained by the following modified $kinst^+$ function (Figure 9.4). The new $kinst^+(\emptyset, \emptyset, \tau <: \mathcal{K})$ returns its encoded proof if $\tau <: \mathcal{K}$ is true, instead of just returning a boolean indicates $\tau <: \mathcal{K}$ or not. The memo function

$$
\begin{array}{l}
\mathit{kinst}^+(M, \psi, \tau <: [\ \sigma_1\ D_1\ |\ldots|\ \sigma_n\ D_n\ ]) = \\
\qquad \text{if } \tau \leq \sigma,\ \forall 1 \leq i < p,\ \mathit{kinst}^+(M, \psi, \tau <: \sigma_i\ D_i) \text{ fails} \\
\qquad \text{and } \mathit{kinst}^+(M, \psi, \tau <: \sigma_p\ D_p) = \texttt{[|}f_1; \ldots; f_n\texttt{|]} \text{ then} \\
\qquad\quad \texttt{Node}(p-1,\ \texttt{[|}f_1, \ldots, f_n\texttt{|]}) \\
\qquad \text{else fails} \\
\mathit{kinst}^+(M, \psi, \tau <: \sigma\ \{\tau_1 <: \mathcal{K}_1\ \&\ \ldots\ \&\ \tau_n <: \mathcal{K}_n\}) = \\
\qquad \text{if } \tau \leq_S \sigma \text{ then} \\
\qquad\quad \texttt{[|}\ \mathit{kinst}^+(M, \psi, S(\tau_1) <: S(\mathcal{K}_1)); \ldots;\ \mathit{kinst}^+(M, \psi, S(\tau_n) <: S(\mathcal{K}_n))\ \texttt{|]} \\
\qquad \text{else fails} \\
\mathit{kinst}^+(M, \psi, \tau <: \mu\kappa.\mathcal{K}) = \\
\qquad \text{if } M(\tau <: \psi(\mu\kappa.\mathcal{K})) \text{ is defined, return it} \\
\qquad \text{else} \\
\qquad\ \ \text{let } \psi' = \psi \setminus \{\kappa \Rightarrow \psi(\kappa)\} \text{ in} \\
\qquad\ \ \texttt{let rec flow}_\kappa = \mathit{kinst}^+(M \cup \{(\tau <: \psi(\mu\kappa.\mathcal{K})) \Rightarrow \texttt{flow}_\kappa\}, \psi'\{\kappa \Rightarrow \mu\kappa.\psi'(\mathcal{K})\}, \tau <: \mathcal{K}) \\
\qquad\ \ \texttt{in flow}_\kappa \\
\mathit{kinst}^+(M, \psi, \tau <: \sigma\ \kappa) = \ \text{if } \kappa \notin \mathit{Dom}(\psi) \text{ then fails} \\
\qquad\qquad\qquad\qquad\qquad\quad \text{else } \mathit{kinst}^+(M, \psi, \tau <: \psi(\kappa))
\end{array}
$$

Figure 9.4: *kinst*$^+$ algorithm with generic flow calculation

$M$ is now extended to have bindings from *kinst*$^+$ predicates to flows. In the check with a recursive constraint, if the questioned predicate is not in the cache $M$, *kinst*$^+$ binds the predicate (applied $\psi$) with the current tree node $\texttt{flow}_\kappa$ and continues the calculation. This binding serves as an entry point for looped flows. If the cache hits, *kinst*$^+$ returns the cached flow identifier and a loop is created. To avoid name collisions, the identifier name must be unique to the corresponding constraint variable $\kappa$. To calculate flows with loops we use recursive definition of non functional values. Because of admissibility of constraints, even flows for type recursions which never terminate can be expressed finitely, as graphs with loops.

### Compilation with generic flows

A new compilation algorithm using generic flows $\lfloor e_t \rfloor_F$ in Figure 9.5 takes an expression and a binding $F$ from variable occurrences to generic flows. If a generic instance `x` is a polymorphic instance, then it must be bound in $F$ and compiled to an application of itself on the flow $F(\texttt{x})$. If `x` is a full instance, $F$ has no binding of `x` (we suppose there is no name collision of identifiers here for simplicity). the flow $f$ of `x` is computed by *kinst*$^+$ and `x` is applied to $f$. A generic value definition is compiled to a function which takes a flow `Node(p,subs)`. For let bound values, the top index `p` is ignored because they have always only one type case, therefore `p` is always 0. Each sub-flow in `subs` is delivered to the corresponding polymorphic instance by compiling the definition expression $e_{t1}$ with the updated binding $F_i$. A set of generic type methods is compiled to an array of method compilations with extra sub-flow arguments. and type case selection is done by accessing and executing `p`-th element of the array (`tm.(p)` is an access to the `p`-th element of the array `tm`). Sub-flows are delivered to polymorphic instances inside the selected type method, like the delivery of run time types in the type matching compilation.

$\lfloor c_\sigma : \tau \rfloor_F = c_\sigma$

$\lfloor \texttt{fun x} \rightarrow e_t \rfloor_F = \texttt{fun x} \rightarrow \lfloor e_t \rfloor_F$

$\lfloor e_{t1}\ e_{t2} \rfloor_F = \lfloor e_{t1} \rfloor_F\ \lfloor e_{t2} \rfloor_F$

$\lfloor e_{t1}, e_{t2} \rfloor_F = \lfloor e_{t1} \rfloor_F,\ \lfloor e_{t2} \rfloor_F$

$\lfloor \texttt{x}_\mathcal{K} : \tau \rfloor_F =$
    if not $generic(\texttt{x})$ then $\texttt{x}$
    else if the occurrence of $\texttt{x}$ is bound in $F$ then $\texttt{x}\ F(\texttt{x})$
        else let $f = kinst^+(\emptyset, \emptyset, \tau <: \mathcal{K})$ in $\texttt{x}\ f$

$\lfloor \texttt{let (rec) x}_{V,\mathcal{K}} = e_{t1}\ \texttt{in}\ e_{t2} \rfloor_F =$
    let $[\texttt{x}^0_{\mathcal{K}_0} : \tau_0;\ \ldots;\ \texttt{x}^n_{\mathcal{K}_n} : \tau_n] = plist(V, \emptyset, e_t)$
    in $\texttt{let (rec) x = fun (Node(\_, subs))} \rightarrow \lfloor e_{t1} \rfloor_{F\{\texttt{x}^i \Rightarrow \texttt{subs}.(i)\}}\ \texttt{in}\ \lfloor e_{t2} \rfloor_F$

$\lfloor \texttt{generic (rec) x}_\mathcal{K} = \texttt{case}\ \sigma_1 \Rightarrow e_{t1} \mid \ldots \mid \sigma_n \Rightarrow e_{tn}\ \texttt{in}\ e_t \rfloor_F =$
    let $[\texttt{x}^{i0}_{\mathcal{K}_{i0}} : \tau_{i0};\ \ldots;\ \texttt{x}^{in_i}_{\mathcal{K}_{in_i}} : \tau_{in_i}] = plist(BDV(\sigma_i), \emptyset, e_{ti})$
    let $F_i = F\{\texttt{x}^{ij} \Rightarrow \texttt{subs}.(j)\}$
    in $\texttt{let (rec) x =}$
        $\texttt{let tm = [| fun subs} \rightarrow \lfloor e_{t1} \rfloor_{F_1};\ \ldots;\ \texttt{fun subs} \rightarrow \lfloor e_{tn} \rfloor_{F_n}\ \texttt{|] in}$
        $\texttt{fun (Node(p, subs))} \rightarrow \texttt{tm.(p) subs}$
      $\texttt{in}\ \lfloor e_t \rfloor_F$

Figure 9.5: The compilation algorithm with generic flows

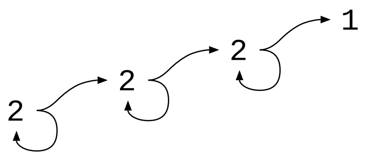
**Example** The previous `print` generic function is compiled as follows:

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

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

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

Then the flow of `print`, $f = \mathit{kinst}^{+}(\emptyset, \emptyset, \texttt{int list list list} \rightarrow \texttt{unit} <: \mathcal{K}_{\texttt{print}})$ is the graph as follows:




The whole expression is compiled to `print` $f$ `[[[1]; [2;3]]; [[4;5]]];;` □

### 9.4.5 Benchmark

Here we do some benchmark. We wrote the following Fibonacci function using a generic binding and compared how this new generic flow compilation method affects evaluation efficiency.

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

Both compilation translate this program as follows:

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

```
let rec fib2 =
  let tm = [| fun subs ->
              function 1 | 2 → 1
                     | n → fib2 (Node (0,[||])) (n - 2)
                           + fib2 (Node (0,[||])) (n - 1)) |]
  in
  fun (Node (p,subs)) →  tm.(p) subs
in fib2 (Node (0,[||]) 36;;
```

Writing Fibonacci function as generic is completely meaningless in practice of course, but here we are just interested in the efficiency of type case selection of the two compilation schemes. Since we can write and time Fibonacci function inside normal ML as below, by subtracting the elapsed time $t_0$ of non-generic version from the time of a generic version $t$, we can hopefully extract $t - t_0$, purely used time in type case selections:

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

In addition, the type match based compilation of `fib` has one of the simplest pattern, «`int`». Therefore we can expect that the comparison condition here is the most advantageous for the type pattern matching compilation; the efficiency of its type case selection is governed by the complexity of compiled patterns, while for the generic flow compilation, it is always constant no matter how complicated patterns are.

We compiled these two compilation and the non generic `fib` by Objective Caml byte code compiler version 3.00 and executed 333MHz Pentium II (not so powerful indeed). By executing `fib 36`, almost 30 million times of type case selections are done.

| | Total time ($t$) | $t - t_0$ |
|---|---:|---:|
| `fib`, Non generic | $t_0 = 7.500$s | – |
| `fib1`, Compiled by pattern matching | 38.610s | 31.110s |
| `fib2`, Compiled with generic flows | 27.920s | 20.420s |

The compilation using generic flows is faster than the original compilation. Even though the type matching in `fib` is the simplest one, type case selections of the new compilation take just 2/3 amount of time of the classical, pattern match based compilation. While the overhead of the pattern matching strategy is proportional to complexity of type patterns, the speed of type case invocation of the generic flow compilation is constant. This means that our new compilation strategy much faster in general.

This performance difference becomes much clearer when generic values have more than one type cases with more complicated patterns, especially latter cases are often selected in evaluation. For example, in the following `flat` application, 90 percent of type case selections choose the second case:

```
generic rec flat = case
| ∀δ.δ list → δ list ⇒ fun x → x
| ∀δδ'.δ list list → δ' list ⇒ fun l → flat (List.flatten l) l
in
for i in 0 to 999999 (* 10^7 type case selections *)
do
  (flat [[[[[[[[[[1]]]]]]]]]] : int list)
done
```

This means that the type pattern matching compilation fails its first pattern and select the second in the almost all cases at run time. In this example, the type matching can no longer compete with the generic flow compilation. To extract the elapsed time in type case selection, we also timed the following non-generic equivalent program:

```
let f = List.flatten

let _ =
  for i = 0 to 999999 do
    f (f (f (f (f (f (f (f [[[[[[[[[[1]]]]]]]]]]))))))))
  done
;;
```

| | Total time ($t$) | $t - t_0$ |
|---|---|---|
| Non generic equivalent program | $t_0 = 13.190$s | – |
| Compiled by pattern matching | 34.820s | 21.630s |
| Compiled with generic flows | 20.100s | 6.910s |

The generic flow type case selection is more than three times as fast as the type matching. Additional to its better and upward compatible semantics, this efficiency of the generic flow compilation expels the older compilation and semantics.

### Size comparison of run time types and generic flows

It is not simple to tell which compilation scheme requires less memory for run time dispatched data. For example, in the following example code `length` is instantiated to $\tau$ `list` → `unit`, where $\tau$ is a long and complicated type expression "*some really long complicated type*":

```
generic length = case
| ∀δ.δ list → unit ⇒ List.length
| ...
in
length (lst : some really long complicated type list)
```

Then the application of the generic value `length` is compiled in both schemes as below:

```
length «some really long complicated type» lst
length (Node (0,[||])) lst
```

While the generic flow for this instance of `length` is quite simple `Node (0,[||])` which instructs to select the first type case, the type matching compilation creates a huge run time type «*some really long complicated type*», even though it is not used in the compilation of the first type case at all.

Contrary, in some cases the type matching compilation has advantage in size:

```
let print_double x =
  print (double x); print " is double of "; print x
in
print_double 1;;
```

Here we suppose that both `print` and `double` have the cases for integers in the first type cases. Both compilation of `print_double 1` are:

```
print_double «int» 1
print_double (Node (0,[| Node(0,[||]); Node(0,[||]); Node(0,[||]) |])) 1
```

In this case, the same run time type «`int`» is sent to each polymorphic instances of `print_double`, but we have to prepare generic flows `Node(0,[||])` for each instance independently.

Thus, in general we cannot tell which scheme is efficient in space, but often the generic flow dispatching requires smaller memory than the dispatching of their corresponding run time types. This is because the generic flows are simple integer graphs and they do not need to carry data type names and tags such as `list`, $\rightarrow$, $\alpha$, etc, like the run time types. This can be highly depending on how we implement the run time types actually, but in our prototype implementation for example, for a full instance of `flat : int list list list list list list list list list list` $\rightarrow$ `int list` in the example of `flat` in this section, while the run time type compilation uses 64 bytes for creating the run time type «`int list list list list list list list list list`» and «`int`», the generic flow compilation just requires 30 bytes to store the corresponding flow `Node(0, [|...[|Node (1, [||])|]...|])`.

## 9.5 Generic primitives and generic flows

Until now, we have just ignored the existence of generic primitives for simplicity of the generic flow data structure as integer graphs. Generic flows need to be extended to support generic primitives, since generic primitives essentially require their instantiation information as run time type arguments. Generic flows now have a new kind of terminal nodes, which contain arrays of run time types. This is implemented as an additional constructor `Type` to the data type `flow` which carries run time types for generic primitive instances:

```
type flow = Node of int * flow array
          | Type of rtype array
```

These run time types in generic flows are delivered to the compilation of the generic primitives at run time.

The new case of generic constraint calculation $kinst^+$ for generic primitives is added to type checking and compilation. Having empty generic constraints, generic primitives are distinguished from the other user-defined generics.

$$\begin{array}{l} kinst^+(M, \tau <: \sigma \ \{\,\}) = \\ \qquad \text{if } \tau \leq_S \forall V.\tau' = \sigma \text{ then} \\ \qquad \text{let } [\delta_1;\ \ldots;\ \delta_n] = \mathit{Vlist}(V \cap \mathbb{D}, \tau') \text{ in} \\ \qquad\quad \texttt{Type [|}\ S(\delta_1);\ \ldots;\ S(\delta_n)\ \texttt{|]} \\ \qquad \text{else fails} \end{array}$$

If the type checker finds a generic primitive instance whose type scheme is $\sigma$ is evaluated with type $\tau$, it creates a "type node" `Type [|` $S(\delta_1);\ \ldots;\ S(\delta_n)$ `|]` where the types $S(\delta_i)$ indicate how each abstracted dynamic type variable $\delta_i$ of $\sigma$ is instantiated in $\tau$. A generic primitive is compiled to a function so that it can make use of these instantiation information stored in a type node. It receives a type node and applies its semantic function `g` to the delivered run time types:

$$\begin{array}{lcl} \lfloor \mathtt{g}_{\forall V.\tau} \rfloor_F & = & \text{let } [\delta_1;\ \ldots;\ \delta_n] = \mathit{Vlist}(V \cap \mathbb{D}, \tau) \text{ in} \\ & & \texttt{fun (Type types)} \rightarrow \texttt{g types.(0) ... types.(n-1)} \end{array}$$

**Example** The following generic function `dynamic_many` creates a dynamic value of the tuple of variable numbers of arguments. (Here we suppose `dynamic` primitive has a simple type $\forall\delta.\delta \rightarrow$ `dyn`. )

```
generic dynamic_many =
| ∀δ.δ → dyn ⇒ dynamic
| ∀δδ′δ″.δ → δ′ → δ″ ⇒ fun x y → dynamic_many (x,y)
;;
```

If the function has only one argument, `dynamic_many` calls `dynamic` and creates the dynamic value of the argument. If there two or more arguments, say `dynamic_many x y z ...`, it creates a tuple of the first two, `x` and `y`, then recursively applies `dynamic_many` itself to this tuple `(x,y)`. With the other third and later arguments `z ...`, this recursive function call is evaluated with the arguments `(x,y) z ...`, one less than the original. Consequently, `dynamic_many x y z ...` creates a dynamic value of a tuple of the arguments `(...((x,y),z),...)`. For example,

```
# dynamic_many 1 3.14 "hello";; (* returns dyn *)
- : dyn = dynamic ((1, 3.14), "hello" : (int * float) * string)
```

Here, `dynamic_many` takes three arguments, therefore its first two calls selects the second type case to create a tuple of them `(1, 3.14), "hello"`. Then it recursively calls its first type case of dynamic value creation. In the first type case, `dynamic` generic primitive is called under the type `(int * float) * string` $\rightarrow$ `dyn`. The compiler prepares a run time type `(int * float) * string` in the generic flow for the `dynamic_many` instance for this call of `dynamic` primitive. The application is compiled to:

dynamize_many $f$ 1 3.14 "hello"

where

$f$ = Node(1,[|Node(1,[|Node(0,[|Type [|«(int * float) * string»|]|])|])|])

This data represents the following generic flow graph, whose terminal node is a type node Type [|«(int * float) * string»|] :

$$1 \longrightarrow 1 \longrightarrow 0 \longrightarrow \texttt{«(int * float) * string»}$$

□

## 9.6 Generic flow dispatching and dictionary passing

We started our research of extensional polymorphism claiming that run time types are indispensable to achieve our goals, but in this chapter, we proven that type case based generic values do not require run time types. This reminds us about Haskell[HJW92]'s dictionary passing program transformation, a compilation of overloaded values without run time types. Actually, our generic flow dispatching has a strong relationship with the dictionary passing. However, rather than a direct comparison with Haskell with type classes, here we look at the dictionary passing transformation of System O, an overloading extension without type classes proposed by Odersky, Wadler and Wehr[OWW95].

Haskell passes a set of the type methods belongs to a class instance as a dictionary. Lacking type classes, a "dictionary" of System O is simpler than Haskell: it is an instance of an overloaded value, which is used in a definition of another overloaded value and its static resolution is impossible at its occurrence, i.e. generic polymorphic instances in our words. The `print` example at the end of Section 9.4.4 is defined as follows in System O (we use Haskell style syntax here):

```
over print

inst print :: Int → Unit
     print = print_int

inst print :: String → Unit
     print = print_string

inst print :: [a] → Unit
     print [] = ()
     print (x:xs) = print x; print xs
```

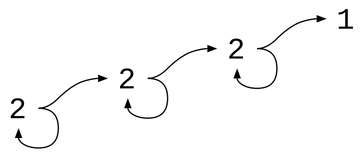
This definition of `print` is transformed to the following instance definitions by dictionary passing. The instance for printing lists, $\texttt{print}_{list}$ is compiled to a function which takes a resolution `f` of the internal use of `print`:

```
let print_{int} = print_int
let print_{string} = print_string
let print_{list} f =
  function [] → ()
         | (x:xs) → f x; (print_{list} f) xs
```

The expression of `print [[[1]; [2;3]]; [[4;5]]]` is transformed to the following:

```
print_{list} (print_{list} (print_{list} print_{int})) [[[1]; [2;3]]; [[4;5]]]
```

Comparing this transformation with our compilation with flows is quite interesting:

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


The syntax tree of the resolution in System O, `print`$_{list}$ (`print`$_{list}$ (`print`$_{list}$ `print`$_{int}$)) has almost the same structure of the flow graph. If we ignore the loop-backs in the flow graph (in the translation of System O, the corresponding overloaded instance with this loop is internally resolved as the recursive call of `print`$_{list}$ `f` in the definition of `print`$_{list}$), flow branches correspond with applications of overload instances, and the integer index of each node represents the corresponding instance of `print`, for example, 0 for `print`$_{int}$ and 2 for `print`$_{list}$.

While generic flow compilation passes integer indices to access overloaded instances stored in an array on the fly, dictionary passing sends instances directly. Therefore dictionary passing style is slightly more efficient than our compilation scheme. However, the dictionary passing style has a drawback: instance definitions of overloaded values of System O, or type classes of Haskell, must be in top-level, so that they are accessible globaly to create dictionaries. Generic flow compilation has no such a restriction: since generic flows are simply graph structured integer indices and have no reference to overloaded instance definitions like dictionaries, user-defined generics can be locally defined.

Local definition of overloaded values does not have great importance in Haskell and System O, partly because of their extension scheme based on the open world assumption: type class instance definition *globally* extendes semantics of overloaded values, therefore their definition must be top-level phrases. In such a circumstance, local definitions of overloaded values are less meaningful than extensional polymorphism, where the `generic` binding to define user generics is formalized as an extension of `let` polymorphism. As we can define parametric polymorphic values locally using `let` expressions, user generics should be also definable in local contexts. The following example demonstrates that local definitions of generic values are indispensable in a conjunction of extensional polymorphism and the ML module system:

```
module F (M:T) = struct
  generic plus = case
  | int → int → int ⇒ (+)
  | M.t → M.t → M.t ⇒ M.plus
```

```
end;;

module N = F(Rational);;
N.plus Rational.zero Rational.one;;
```

The definition of `plus` inside the functor definition is local, due to the dependency on the module argument `M`.

## 9.7 Conclusion

We have invented a completely new compilation method of extensional polymorphism called generic flow compilation. A generic flow is basically an integer graph which encodes a validity proof of the type reconstruction of a generic full instance. Using this new generic flow compilation scheme, we turned the usual run type passing mechanism into execution of simple automata manipulating integer graphs and array accesses. This simplification also improves the efficiency of the generic values. Our idea of passing integer graphs to access overloaded definitions is inspired by the index passing of Ohori's compilation of polymorphic record calculus [Oho95]. Our scheme is more complex due to the arborescent nature of type information dispatch in generic applications.

Dispatching generic flows instead of run time types, we no longer need run time type pattern matching of user-defined generic values. Type case selection and invocation are now simplified to array accesses using indices of generic flows. It improves efficiency of type case selection greatly: the selection is performed in constant time, regardless of the complexity of type patterns.

Thanks to the static analysis and determination of valid type cases for the run time type case selection, the type case selection semantics is now enlarged: the semantics of type case selection is found via a backtracking search, which was out of reach of run time analysis, due to the additional computational complexity. Type case selection semantics with backtracking is completely upward compatible with the classical type match based selection. In addition, by searching type cases using backtracking, the new semantics can provide appropriate semantics to some generic instances which cannot be given meaningful semantics under the type match based selection.