Chapter 10

Reconstruction of constraint typing

10.1 Introduction

Constraint type reconstruction

The generic constraint type system introduced in Chapter 8 checks the validity of generic full instances and assures that a well-typed program is never evaluated to a generic run time type error. We here introduce a type reconstruction algorithm for the generic constraint typing. Similarly to the type systems, type reconstruction of extensional polymorphism is also separated into two: the type reconstruction for generic constraint typing, *constraint type reconstruction algorithm* is an independent phase of the normal type reconstruction algorithm. It takes an expression which may have too general types for the constraint type system, and tries to instantiate its dynamic type variables so that the full instances of the expression become all valid in the constraint typing.

Coherence problem and ambiguity resolution

A generic full instance may have more than one possible valid instance types which provide incompatible semantics. A program is called *incoherent* when it contains such *ambiguous* instances. For example, suppose we have the following generic value `one` with the constraint [`int` | `float`]:

```
generic one = case int ⇒ 1 | float ⇒ 1.0;;
```

Then an untyped full instance `one` is incoherent, since it can be validly instantiated both to `one : int` and `one : float`, that are incompatible each other.

Incoherence of a program can be automatically solved by selecting one of the possible typings and incompatible semantics as default. But it is usually impossible to select such a default, if there is no no strong reasoning for getting one default and abandoning the others. In such cases, incoherent programs are often just rejected by type systems due to irresolvable ambiguities of semantics and typing. In extensional polymorphism, we can use the definition order of type cases to solve this coherence problem, since the type cases for a user-defined generic value are declared as a list once and for all in one generic binding. As the matched type case defined most previously has the selection priority to the other matches defined later in type case selection semantics, our *automatic ambiguity resolver* choose the typing and semantics of the previously defined type case, when a generic instance is ambiguous. In the above example of `one`, if there is no special type constraint, it is typed always as `int`, since its type case is defined prior to the case for `float`:

```
# one;;
- : int = 1
```

In the case when the second type case should be selected, we have to give appropriate type context:

```
# (one : float);;
- : float = 1.0
```

Reconstruction algorithm

The algorithm has mainly three reconstruction phases: in the first initialization phase, generic instances are gathered from the given expression, and the set of full instances and the constraints of generic values defined in the expression are prepared. Then, in the second ambiguity resolution phase, the algorithm tries to find an appropriate type instantiation for each generic full instance to resolve its ambiguity. In the last phase, the remained free occurrences of dynamic variables are unified with the abstract type `dyn`, to make the whole expression well-typed in the constraint typing.

Outline of this chapter

In Section 10.2, we explain why we need the constraint type reconstruction, additional to the normal one. Then Section 10.3 discuss the coherence problem in extensional polymorphism, and present a solution to it, *automatic ambiguity resolution*, the main part of the algorithm. In Section 10.4 we introduce `dyn`-*unification*, the last reconstruction phase and illustrates how it also resolves the coherence problem of generic primitives. The detailed definition of these phases of the algorithm is in Section 10.5, and its correctness is informally discussed. Finally, in Section 10.6, we discuss some future ideas of possible enhancements of the algorithm.

10.2 Constraint type reconstruction

Type annotations of programs inferred only by the normal type reconstruction (Section 2.6) may be too general and ambiguous for generic full instances. Without proper instantiations, their type annotations can be ill-typed and rejected in the constraint typing system. If we force to execute such programs, they may raise run time type match failures. For example, suppose we have the following `plus` definition:

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

```
| float → float → float ⇒ (+.)
| float → int → float ⇒ fun x y → x +. float_of_int y
| int → float → float ⇒ fun x y → float_of_int x +. y
;;
```

This `plus` takes two arguments of integer or float and returns the sum of them. If both arguments are integers, it returns a result in integer. Otherwise, the sum is returned as a float, converting integers to floats if necessary. This function has the following type scheme and generic constraint:

```
plus : ∀δδ'δ''.δ → δ' → δ''
plus : [ int → int → int
       | float → float → float
       | float → int → float
       | int → float → float ]
```

The arguments and return types of the normal type scheme have independently quantified different dynamic type variables, but the actual instantiations of these quantified dynamic type variables are constrained by the generic constraints. For instance, in the following non type-annotated expression,

```
# plus 1.2 3.4;;
```

`plus` cannot be instantiated to nothing else `float` → `float` → `float`, since this is the only type pattern constraint of `plus` which takes two float arguments.

Unfortunately, the normal type reconstruction cannot reconstruct this type, since it only uses the parametric type scheme $\forall\delta\delta'\delta''.\delta \to \delta' \to \delta''$. In the expression `plus 1.2 3.4`, it gives the instance `plus` a type annotation `float` → `float` → δ'', which is the principal type of this instance in the normal type system, but it is not valid instance in the generic constraint typing. Only the work around within the normal type reconstruction is instantiating such too general types so that they can be also valid instance types in the generic constraint typing. In this example of `plus`, to make the expression well-typed in both type systems, we need to use it in certain type contexts like

```
# sqrt (plus 1.2 3.4)   (* sqrt : float → float *)
```

which unifies the dynamic type variable δ'' to `float`, or write with an explicit type constraint as follows:

```
# (plus 1.2 3.4 : float);;
```

The latter work-around is ridiculous once we have introduced the generic constraint type system. Using the constraint information of `plus`, the type of its instance must be inferred as `float` → `float` → `float` without this explicit type constraint. This type reconstruction in the generic constraint typing for automatic overloading resolution is the goal of this chapter.

10.3 Ambiguity resolution

10.3.1 Incoherent programs

In the previous example of `plus 1.2 3.4`, we could guess the type of `plus` uniquely to and the generic constraint type reconstruction may be able to find the type without any difficulty. In general, however, expressions without type annotations can have more than one valid typing which provide different semantics. It often happens when two ore more generic values interact themselves by sharing dynamic type variables. For instance:

```
generic one = case
| int ⇒ 1
| float ⇒ 1.0
in
plus one one;;
```

The expression `plus one one` can have the following four possible valid constraint typings:

```
(plus : int → int → int      ) (one : int  ) (one : int  )
(plus : float → float → float) (one : float) (one : float)
(plus : float → int → float  ) (one : float) (one : int  )
(plus : int → float → float  ) (one : int  ) (one : float)
```

Each of them has different semantics from the others, since the selected cases of `plus` are all different. In this sense `plus one one` is *ambiguous*. If programs have more than one possible constraint typings and they provide incompatible semantics, the programs are called *incoherent*. In most overloading resolution including Haskell [HJW92], the incoherent programs are usually rejected due to their ambiguity and force programmers to add explicit type informations to make expressions unambiguous. For example:

```
plus (one : float) (one :float)
```

is no longer ambiguous. It is a coherent program.

10.3.2 Automatic ambiguity resolution

If a program is incoherent, should the type checker select one of the possible typings and fix its semantics automatically? This rather aggressive *automatic ambiguity resolution* is not normally performed in languages with overloading due to the following reasons:

- There is always the risk that the programmer's intension may differ from automatically selected types and semantics.
- There is often no reasonable and easy-to-understand strategy to select such default typing.

Suppose the ambiguity resolution automatically selects `float` → `float` → `float` as an instance type of `plus` in `plus one one`, but the programmer writes this expression with an intension that `plus` is evaluated as `float` → `int` → `float`. Fortunately, in this example, the both typings return the same result `2.0`, but in general different typings provide different semantics, which should be unexpected to programmers.

In addition to the first reason, in languages with overloading, there is often no (or just weak) notion of priority between the definitions overloaded to one value. Therefore such languages cannot have enough strong reasoning to select default cases for the ambiguity resolution. In the type class based formalization for example, overloaded instances of one class (roughly equal to type cases in our system), are defined in different places of programs rather independently. Introducing an instance precedence to such system may lose clarity of codes[1]. For example, (like we will do in our framework,) we could use the definition order of class instances to decide the priority of automatic ambiguity resolution for type classes, but such relationship between class instances should be quite difficult to be traced and understood, since class instances can be defined everywhere, even in different modules.

Then, should ambiguities be reported as errors, also in extensional polymorphism? Our claim is *no*. Even a program which is incoherent in logic often has only one unique practical typing and corresponding semantics. The other possible typings and semantics are less meaningful and popular than this default. In addition, in extensional polymorphism, overloaded definitions are declared once and for all[2], as the list of type cases. We can make use of the definition order of type cases as a clear priority of automatic ambiguity resolution. Let's see such examples:

Multi-argument printing

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

generic rec multi_print =
| unit ⇒ ()
| ∀δδ'.δ → δ' ⇒ fun x → print x; multi_print
;;
```

[1]An extension of the overlapping instances for Haskell which is proposed in [JJM97] introduces a form of instance priority. There, overlapping of instances are permitted only when one of overlapping pair is strictly more specific than the other. The definition of the more specific instance is automatically selected if both the overlapping instances match with the type context, and the other more general instance is used as the default instance for the other cases. Of course, this instance priority is independent of the definition order of the instances.

This feature is implemented in Gofer[Jon94], and a special "`-98 +o`" mode of Hugs98[JP] for example.

[2]This current lack of extensibility of generic values clearly goes against the modular programming style of overloaded values. We will get rid of this restriction in Chapter 11.

Here we define `multi_print` generic value using our familiar `print`. `multi_print` is a generic function which can take variable numbers of arguments (even zero argument is possible). When `multi_print` is evaluated as a function, it sends its first argument to `print` generic function, then calls itself recursively with the other arguments. After printing the all arguments, it returns unit. This mechanism provides simple but powerful printing, for example,

```
# multi_print 2001 " " ["Space"; " "; "Odyssey"] '\n';;
2001 Space Odyssey
- : unit = ()
```

But wait! Who has determined the type of the expression as `unit` ? Actually this is an incoherent example. The normal type reconstruction gives the type `int` → `string` → `string list` → `char` → δ' to the instance of `multi_print`, and it has infinitely many possible valid instance types in the constraint typing. For example:

```
int → string → string list → char → unit
int → string → string list → char → int → unit
int → string → string list → char → int → char → unit
int → string → string list → char → int → char → string → unit
int → string → string list → char → int → char → string → ... → ...
...
```

Therefore, if we want to see the arguments printed without automatic ambiguity resolution, we need to add following type constraint:

```
# (multi_print 2001 " " ["Space"; " "; "Odyssey"] '\n' : unit);;
2001 Space Odyssey
- : unit = ()
```

Generally, if we want to print values using `multi_print` without any ambiguity resolution, we have to take care of explicitly declaring the termination of applications by giving the return type of `unit` in almost cases. In the following simple example, we need such an explicit type constraint:

```
# (multi_print 1 : unit);;
1
- : unit = ()
```

It is too cumbersome! If one writes this expression even without the constraint, practically what he wants is almost clear. What he wants should be the side effect of printing `1` on the screen and never a printing function. This requires that generic type reconstruction should automatically resolve the ambiguity of `multi_print 1`, giving `unit` as its default type:

```
# multi_print 1;;
1
- : unit = ()
```

The other possible typings are still valid if we need. In order to choose such non default types, we have to add some clear typing contexts by an explicit type constraint for example:

```
# (multi_print 1 : string → unit);;
- : string → unit = <fun>
```

Generic list flattening

The second example is generic list flattening function `flat`:

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

The typeless expression `flat [[1]]` is incoherent, because there are two possible typings which provide different semantics:

```
# (flat [[1]] : int list);;
- : int list = [1]

# (flat [[1]] : int list list);;
- : int list list = [[1]]
```

It seems that the first typing `flat : int list list → int list` and its semantics should be more natural than the other, since it actually flatten the list `[[1]]`, while the second typing returns the list itself. Therefore it should be practical if the first typing is selected automatically for the typeless expression `float [[1]]` :

```
# flat [[1]];;
- : int list = [1]
```

Not only for the lists with depth two, the function `flat` should also completely flatten the lists with any depth to lists with depth one as its default behavior when there is no special type constraint:

```
# flat [[[[[[[1]]]]]]];;
- : int list = [1]
```

Of course, we should be able to control this flattening and stop it at certain point by giving some type information:

```
# (flat [[[[[[[1]]]]]]] : int list list list);;
- : int list list list = [[[1]]]
```

Ambiguity resolution priority

In these two examples, we have seen that even if expressions are incoherent and have more than one possible semantics, often we can find the "most default behavior" for them: `multi_print` prints out all the arguments already given and returns a unit, and `flat` tries complete flattening of its argument. The default behavior is usually determined from a quite heuristic criterion: it is useful or not. Automatic selection of such heuristic default from the others is impossible, therefore we programmer need to specify the priority between possible typings manually.

Actually, such priority can be easily introduced, since all the type cases of a user-defined generic value are declared once and for all, as a list. If a program is incoherent i.e. a generic instances has more than one type cases which have possibility of match with some instances of its type inferred by the normal type reconstruction, we can choose the most previously defined one of them as default, and instantiate the type of the instance so that it can actually match with the default case and select it.

Let's return to the previous examples.

```
generic rec multi_print =
| unit ⇒ ()
| ∀δδ'.δ → δ' ⇒ fun x → print x; multi_print
;;
```

Since the case of `unit` is declared first, `multi_print` tends to choose `unit` as its instance type when the typing inferred by the normal type reconstruction is ambiguous. For example, the expression `multi_print` is given a type δ by the normal reconstruction, and both type patterns have possibilities of match with some instantiations ($\{\delta \Rightarrow \texttt{unit}\}$ for the first and $\{\delta \Rightarrow \texttt{int} \rightarrow \texttt{unit}\}$ for the second for example). In this case the automatic ambiguity resolution chooses the first possibility:

```
# multi_print;;
- : unit = ()
```

This scheme can be applied to more complicated examples:

```
# multi_print 1;;
1
- : unit = ()
```

The above instance `multi_print` is given the type `int` $\rightarrow \delta$ from the normal reconstruction. It chooses the second type case because its type cannot match with the first. In the second type case, `multi_print` is called recursively with the type δ which is ambiguous. This ambiguity is resolved by choosing the first possible match (the first type case). The variable is instantiated to `unit` so that the first type case can be actually selected. As the result the type of the whole expression becomes `(multi_print 1 : unit)` and we obtain desirable printing.

The default behavior of `flat` is also explained by this priority between type cases:

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

If both type case have possibility of match, then the ambiguity resolver chooses the first type case, list flattening, rather than the second, the identity which means the end of flattening. This selection priority is applied recursively, and we obtain the complete list flattening as the default behavior of `flat`:

```
# flat [[[[[[[1]]]]]]];;
- : int list = [1]
```

Changing the order of type cases changes the priority for the automatic ambiguity resolution, therefore the resolved default typing and corresponding semantics of typeless expressions may be modified:

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

# flat' [[1]];;
- : int list list = [[1]]
```

Exchanging the order of the type cases, the ambiguity resolution of `flat'` takes the identify function rather than flattening.

This priority defined by the definition order of type cases can be easily understood, as the ordering of type cases is always visible clearly since they are always declared in groups. As we have argued in the beginning of this section, this style of priority could be introduced also in the class type based system, but it should be more confusing and difficult to understand, since class instances are often defined separately, in different places.

10.3.3 Call number restriction of recursive ambiguity resolution

The automatic ambiguity resolution which we are going to introduce has a risk of falling into infinite loops. The following `multi_print'`, which is obtained by exchanging the order of the type cases of the original `multi_print` is one of such problematic examples:

```
generic rec multi_print' =
| ∀δδ'.δ → δ' ⇒ fun x → print x; multi_print'
| unit ⇒ ()
;;
```

This definition expresses that the functional type has higher priority than the `unit`, when the instance type of `multi_print'` is ambiguous. But if `multi_print'` has an ambiguous instance type, the internal polymorphic instance of `multi_print'` in the first type case is also ambiguous. This implies that this recursive call chooses the first type case too. Thus, this recursive resolution selects the first type case of `multi_print'` infinitely. Therefore, unless we stop the resolution at some place, the following input

```
# multi_print' 1;;
```

never returns from its ambiguity resolution, instantiating the type of the original instance infinitely to $\texttt{int} \to \delta \to \delta' \to \delta'' \to \delta''' \to \ldots$. This is much worse than typing errors which were raised in type system without ambiguity resolution. In this case, the most default type for `multi_print'` has an infinite length.

To prevent infinite loops search space must be restricted to something finite. We introduce *maximum call number restriction* of recursive ambiguity resolution. It restricts the numbers of recursive calls of the resolution algorithm for each instance. The resolver algorithm will check the possible type case selections in the order of their priorities. If the algorithm find a correct selection which makes a generic instance valid, it returns the selection as default selection successfully. Otherwise, if it reaches the maximum calling limit before successful return, it rejects the instance as a type ambiguity error and requests the programmer to give a more detailed type context so that the resolver can finish its job:

```
# multi_print' 1;;
Ambiguous instance: multi_print' has irresolvable type int → δ
# (multi_print' 1 : int → unit);;        (* give an explicit type context *)
1
- : int → unit = <fun>
```

10.4 Ambiguity resolution by `dyn`-unification

The algorithm *kresolv* instantiates the types of full instances in a given program so that they are all valid. But the program may not be well-typed yet, since it may contain free dynamic type variable. Well-typed programs must not have any free dynamic type variable occurrence[3] from Lemma 29. Therefore, at the last stage of the reconstruction, we must instantiate the free dynamic type variable occurrences to something closed.

10.4.1 Ambiguity resolution of generic primitives

Instantiating the free dynamic variables has another functionality: elimination of incoherence of generic primitive instances. Free dynamic type variables in type annotations often cause meaningless semantics of generic primitives. We can resolve this problem by `dyn`-unification — unifying

[3]As explained in Section 8.13, this restriction of dynamic type variables may not be necessary. In such a case, this `dyn`-unification need not be applied to all the free dynamic type variable occurrences.

all the free dynamic type variable in programs with the abstract type `dyn`, which has been originally used for the polymorphic dynamization in Chapter 4. The semantics of a generic primitive is defined for all its instance types. Therefore in theory, there is no incoherent generic primitive instance. But it is sometimes impossible to find proper semantics of generic primitive instances when their type annotation contains free dynamic type variables. This is the coherence problem of generic primitives. To solve this problem, we need another ambiguity resolution, since generic primitives lack definitions, therefore the ambiguity resolution using the definition order of type case definitions is useless for them.

Suppose that `gread` is a generic primitive whose type is $\forall\delta.\mathtt{unit} \to \delta$. It reads a user input string, then tries to parse it as a value which is requested from its type context:

```
# (gread () : int)                          (* parses user input as int *)
> 1                                                       (* user input *)
- : int = 1                                            (* parsed result *)

# (gread () : string)                    (* parses user input as string *)
> 1                                                       (* user input *)
- : string = "1"
```

Even the same input string `"1"` is parsed differently depending on return types. Note that this generic primitive essentially requires types for return values. Then, what will happen for the following expression ?

```
# gread ();;
```

The normal type reconstruction gives the principal typing δ to this expression. Since the value has a totally polymorphic type scheme $\forall\delta.\mathtt{unit} \to \delta$ and constraint $[\forall\delta.\mathtt{unit} \to \delta]$, this principal typing is also well-typed in the constraint typing. But `gread` cannot return any meaningful result; there is no actual ML value whose type is δ! With this type `gread : unit` $\to \delta$, what it can do is just raising some exception:

```
# gread ();;
> hello                                                   (* user input *)
Uncaught exception: Failure "parse error"
```

This "principal typing" with a free dynamic type variable never crashes the program (it just raises an exception), but in practice it should be automatically rejected because of its completely useless semantics.

This problem can be avoided by preventing `gread` from having *free* dynamic type variables in its instance type: such type annotations should be ill-typed. (On the other hand, if dynamic type variables in annotations of `gread` are generalized by polymorphism, they are considered well-typed.) We also here use the `dyn`-unification of Chapter 4 to eliminate the free dynamic type variables:

```
# gread ();;       (* gread has a type unit → dyn *)
> hello
- : dyn = ("hello" : string)
```

Usually, `gread` is forced to parse its input strings as values of return types which are provided from the typing context. But when `gread` has type `unit` → `dyn`, it can guess the most appropriate type for its input by itself, then return the result encapsulating as a dynamic value, using its heterogeneousness:

```
# gread ();;
> 1
- : dyn = (1 : int)
# gread ();;
> 3.14
- : dyn = (3.14 : float)
# gread ();;
> 1.23.456
- : dyn = ("1.23.456" : string)
```

Generally speaking, free dynamic type variables which do not occur in functional arguments (not in the left hand side of the arrow type) have the same kind of semantic troubles: generic primitives often cannot create results of such types with free type variables because they lack actual ML values. By `dyn`-unification, we can avoid this problem.

10.4.2 `dyn`-unification in the final type reconstruction phase

Since `dyn`-unifications of the problematic free dynamic type variables can solve the coherence problem of generic primitives as we have seen, we introduce this `dyn`-unification at the end of the type reconstruction and ambiguity resolution. After the automatic ambiguity resolution of Section 10.3.2 finds the best type annotations for user generics as possible, we will `dyn`-unify all the remained free dynamic type variables in the type annotations. Because of simplicity of this algorithm, it even unifies the free dynamic type variables which do not relate with generic primitives, however, this makes the given program follow the typing restriction of Lemma 29: it has no free dynamic type variables.

10.5 The constraint type reconstruction algorithm

The constraint type reconstruction algorithm consists of the following three phases. Before their precise definitions, we first overview how each phase works:

Initialization

The initialization takes a compilation unit (i.e. a module) which is already type-checked by the normal type reconstruction algorithm. The algorithm gathers all the generic related definitions and instances which the following phases are interested in: it assigns generic constraints to the generic value definitions and lists up all the full generic instances.

First ambiguity resolution

In the first ambiguity resolution phase, the algorithm checks whether the type of each full instance is a valid instance of its constraint assigned by the initialization. If the type is too general for a valid instance, automatic ambiguity resolution tries to find appropriate instantiation which makes the instance valid. If there are more than one possible instantiations, the automatic resolution chooses one with the highest priority. If there is no possible type instantiation which makes the generic instance valid, or the algorithm call reaches to the maximum call limit, it reports a type error.

Second resolution with `dyn`-unification

After the above first resolution, we `dyn`-unify all the free dynamic type variable annotations, to make the given program well-typed and resolve the coherence problem of generic primitives at the same time.

10.5.1 Initialization

In the initialization, we prepare and assign constraints of generic values and gathers the full generic instances in compilation units. The following ambiguity resolution is applied to these full instances and returns a substitution which should be applied to the compilation unit. The polymorphic instances are not the direct target of the ambiguity resolution; dynamic type variables of polymorphic instances are generalized and therefore registered in derived constraints of the generic values which use them.

Generic constraint assignment

Using the constraint construction function *Derive* defined in Section 8.9.2, function *kassign* (Figure 10.1) adds generic constraint annotations to type-annotated, but not constraint annotated expressions. Function *kassign* takes an environment K which maps identifiers to their constraints. For a recursively defined identifier, *kassign* first temporarily assigns a constraint variable κ for its constraint. After the constraint assignment of the definition body and its constraint $\mu\kappa.\mathcal{K}$ is calculated, this temporal constraint κ is replaced by the recursive constraint $\mu\kappa.\mathcal{K}$.

Full instance list-up

Another thing to be done in the initialization phase is gathering the generic full instances. The constraint annotated expression produced by *kassign* may not be well-typed because of possibly invalid or too general type annotations of generic full instances. In the next ambiguity resolution phase, their type annotations must be instantiated so that they can also be valid instances in the generic constraint type system.

Following *fullinsts* scans a constraint annotated expression and retrieves the generic full in-

$$
\begin{array}{rcl}
kassign(K,\ c_\sigma : \tau) & = & c_{[\sigma]} : \tau \\
kassign(K,\ \mathtt{g}_\sigma : \tau) & = & \mathtt{g}_{[\sigma]} : \tau \\
kassign(K,\ \mathtt{x}_\sigma : \tau) & = & \mathtt{x}_{K(\mathtt{x})} : \tau \\
kassign(K,\ (\mathtt{fun\ x} \rightarrow e_t) : \tau_1 \rightarrow \tau_2) & = & (\mathtt{fun\ x} \rightarrow kassign(K\{\mathtt{x} : [\tau_1]\}, e_t)) : \tau \\
kassign(K,\ (e_{t1}\ \ e_{t2}) : \tau) & = & (kassign(K, e_{t1})\ \ kassign(K, e_{t2})) : \tau \\
kassign(K, (e_{t1}\,,e_{t2}) : \tau) & = & (kassign(K, e_{t1}),\ kassign(K, e_{t2})) : \tau \\
kassign(K, (\mathtt{let\ x}_V = e_{t1} : \tau_1\ \mathtt{in}\ e_{t2}) : \tau) & = & \text{let } e'_{t1} = kassign(K,\ e_{t1}) \\
& & \text{let } D = Derive(V, \emptyset, e'_{t1}) \\
& & \text{let } \mathcal{K} = [\forall V.\tau_1\ D] \\
& & \text{if } adm^+(\emptyset, \mathcal{K}) = \mathit{false} \text{ then fails} \\
& & \text{let } e'_{t2} = kassign(K\{\mathtt{x} : \mathcal{K}\},\ e_{t2}) \\
& & \text{in } (\mathtt{let\ x}_{V,\mathcal{K}} = e'_{t1}\ \mathtt{in}\ e'_{t2}) : \tau \\
\multicolumn{3}{l}{kassign(K, (\mathtt{let\ rec\ f}_V = e_{t1} : \tau_1\ \mathtt{in}\ e_{t2}) : \tau) =} \\
& & \text{let } e'_{t1} = kassign(K\{\mathtt{f} : \kappa\},\ e_{t1}) \\
& & \text{let } D = Derive(V, \emptyset, e'_{t1}) \\
& & \text{let } \mathcal{K} = [\forall V.\tau_1\ D] \\
& & \text{let } e''_{t1} = \{\kappa \Rightarrow \mu\kappa.\mathcal{K}\}(e'_t) \\
& & \text{if } adm^+(\emptyset, \mu\kappa.\mathcal{K}) = \mathit{false} \text{ then fails} \\
& & \text{let } e'_{t2} = kassign(K\{\mathtt{f} : \mu\kappa.\mathcal{K}\},\ e_{t2}) \\
& & \text{in } (\mathtt{let\ rec\ f}_{V,\mu\kappa.\mathcal{K}} = e''_{t1}\ \mathtt{in}\ e'_{t2}) : \tau \\
\multicolumn{3}{l}{kassign(K,\ (\mathtt{generic\ x}_\sigma = \mathtt{case}\ \sigma_1 \Rightarrow e_{t1}\ \mid\ \ldots\ \mid \sigma_n \Rightarrow e_{tn}\ \mathtt{in}\ e_t) : \tau) =} \\
& & \text{let } e'_{ti} = kassign(K, e_{ti}) \\
& & \text{let } D_i = Derive(BDV(\sigma_i), \emptyset, e'_{ti}) \\
& & \text{let } \mathcal{K} = [\sigma_1\ D_1 \mid \ldots \mid \sigma_n\ D_n] \\
& & \text{if } adm^+(\emptyset, \mathcal{K}) = \mathit{false} \text{ then fails} \\
& & \text{let } e'_t = kassign(K\{\mathtt{x} : \mathcal{K}\}, e_t) \\
& & \text{in } \ (\mathtt{generic\ x}_{\mathcal{K}} = \mathtt{case} \\
& & \quad \mid \sigma_1 \Rightarrow e'_{t1} \mid \ldots \mid \sigma_n \Rightarrow e'_{tn}\ \mathtt{in}\ e'_t) : \tau \\
\multicolumn{3}{l}{kassign(K,\ (\mathtt{generic\ rec\ x}_\sigma = \mathtt{case}\ \sigma_1 \Rightarrow e_{t1}\ \mid\ \ldots\ \mid \sigma_n \Rightarrow e_{tn}\ \mathtt{in}\ e_t) : \tau) =} \\
& & \text{let } e'_{ti} = kassign(K\{\mathtt{x} : \kappa\}, e_{ti}) \\
& & \text{let } D_i = Derive'(BDV(\sigma_i), \emptyset, e'_{ti}) \\
& & \text{let } \mathcal{K} = [\sigma_1\ D_1 \mid \ldots \mid \sigma_n\ D_n] \\
& & \text{let } e''_{ti} = \{\kappa \Rightarrow \mu\kappa.\mathcal{K}\}(e'_{ti}) \\
& & \text{if } adm^+(\emptyset, \mu\kappa.\mathcal{K}) = \mathit{false} \text{ then fails} \\
& & \text{let } e'_t = kassign(K\{\mathtt{x} : \mu\kappa.\mathcal{K}\}, e_t) \\
& & \text{in } \ (\mathtt{generic\ rec\ x}_{\mu\kappa.\mathcal{K}} = \mathtt{case} \\
& & \quad \mid \sigma_1 \Rightarrow e'_{t1} \mid \ldots \mid \sigma_n \Rightarrow e'_{tn}\ \mathtt{in}\ e'_t) : \tau
\end{array}
$$

Figure 10.1: Constraint assignment

stances in it.

$$\begin{array}{rcl}
\mathit{fullinsts}(V,\ c_{[\sigma]} : \tau) &=& \emptyset \\
\mathit{fullinsts}(V,\ \mathtt{g}_{[\sigma]} : \tau) &=& \text{if } \mathit{polyinst}(V, \sigma, \tau) = \mathit{false} \text{ then } \{\mathtt{g} : \tau\} \text{ else } \emptyset \\
\mathit{fullinsts}(V,\ \mathtt{x}_{\mathcal{K}} : \tau) &=& \text{if } \mathit{polyinst}(V, \mathcal{K}, \tau) = \mathit{false} \text{ then } \{\mathtt{x} : \tau\} \text{ else } \emptyset \\
\mathit{fullinsts}(V,\ \mathtt{fun\ x} \rightarrow e_t) &=& \mathit{fullinsts}(V, e_t) \\
\mathit{fullinsts}(V,\ e_{t1}\ e_{t2}) &=& \mathit{fullinsts}(V, e_{t1}) \cup \mathit{fullinsts}(V, e_{t2}) \\
\mathit{fullinsts}(V,\ (e_{t1}, e_{t2})) &=& \mathit{fullinsts}(V, e_{t1}) \cup \mathit{fullinsts}(V, e_{t2})
\end{array}$$

$$\mathit{fullinsts}(V, \mathtt{let\ (rec)\ x}_{V', \mathcal{K}} \mathtt{=} e_{t1} : \tau \ \mathtt{in}\ e_{t2}) = \mathit{fullinsts}(V \cup V', e_{t1}) \cup \mathit{fullinsts}(V, e_{t2})$$

$$\mathit{fullinsts}(V, \mathtt{generic\ (rec)\ x}_{\mathcal{K}} \mathtt{=case}\ \sigma_1 \Rightarrow e_{t1} \mid \ldots \mid \sigma_n \Rightarrow e_{tn}\ \mathtt{in}\ e_t) = \bigcup_{i=1}^{n} \mathit{fullinsts}(V \cup BDV(\sigma_i), e_{ti})$$

We can easily verify that all the generic instances in expressions are either full instances which are scanned by this *fullinsts*, or polymorphic instances whose informations are stored in derived constraints.

Initialization

The initialization phase receives an expression e_t from the normal type reconstruction algorithm. It produces constraint annotated expression e'_t and lists up the generic full instances F of e'_t:

$$e'_t = \mathit{kassign}(\emptyset, e_t), \qquad F = \mathit{fullinsts}(\emptyset, e'_t)$$

It passes these e'_t and F to the next phase, the first ambiguity resolution.

10.5.2 First ambiguity resolution

In this phase, we resolve the ambiguity of the generic full instances in F. Each full instance resolution tries to find a type substitution which makes itself valid. If it proves that there is no such substitution, it reports the invalidness as a type error. If the algorithm cannot resolute the ambiguity until the number of its calls reaches the maximum limit, it asks the user to give some more explicit type information. When all the ambiguity resolutions of the generic full instances are done, the substitutions are applied to e'_t, and sent to the last phase.

Ambiguity resolution *kresolv*

The function *kresolv* in Figure 10.3 plays the mail role of the constraint ambiguity resolution. It takes three arguments: M is a memo function which caches the checks already done in the course of the recursive calls of *kresolv*. The last argument is a *problem* which to be solved. The function *kresolv* try to find a substitution which make the problem valid which is one of the following:

Full instances A problem takes the form of a list of derived constraints $\{\tau_1 <: \mathcal{K}_1 \;\&\ldots\&\; \tau_n <: \mathcal{K}_n\}$. These derived constraints intuitively correspond with the full generic instances $\mathtt{x}^1_{\mathcal{K}_1} : \tau_1,\ \ldots,\ \mathtt{x}^n_{\mathcal{K}_n} : \tau_n$. The algorithm tries to find a substitution S which makes these full instances valid i.e. $S(\tau_1) <: S(\mathcal{K}_1), \ldots,\ S(\tau_n) <: S(\mathcal{K}_n)$ are all valid.

We solve the problem using a simple divide-and-conquer method. A problem $\{\tau_1 <: \mathcal{K}_1 \;\&\ldots\&\; \tau_n <: \mathcal{K}_n\}$ is separated into n sub-problems $\tau_i <: \mathcal{K}_i$ independent of each other, which finds a substitution S_i that makes $S_i(\tau_i) <: S_i(\mathcal{K}_i)$ valid. If all the sub-problems are successfully solved, we merge the sub-answers S_i to get an universal answer, substitution S. This is done by the substitution of unifications *unify_substs* in Figure 10.2. Given substitutions $S_1, \ldots, S_n$, *unify_substs* returns one of the most small substitution which unify the types $S_1(\tau), \ldots, S_n(\tau)$ for any type expression τ. If this unification of substitutions fails, the entire algorithm also fails immediately.

To restrict the total number of calls of *kresolv* for a full instance ambiguity resolution, we have a counter of the number of its calls. At the first time when *kresolv* is called from the ambiguity resolution, here we must reset the counter before each subsequent *kresolv* call.

Pattern constraints Here we are given a problem in the form of $\tau <: [\sigma_1\ D_1 \mid \ldots \mid \sigma_n\ D_n]$. First of all, we increment the call time counter of the algorithm *kresolv* here. If it reaches the maximum call number limit, the algorithm immediately terminates and exists to ask the user to add more precise type information.

Otherwise, corresponding with our type case selection policy, the algorithm tries to find the first success, the smallest index number p and a substitution S_p, such that $S_p(\tau) <: S_p(\delta_p\ D_p)$ valid. If there is no such p, the algorithm reports an error.

A pattern constraint with derived constraints A problem takes the form of $\tau <: \forall V.\tau'\ D$. Since τ may not be the instance of $\forall V.\tau'$, we first try to find the substitution S by the unification of τ and τ', which makes $S(\tau)$ be an instance of $S(\forall V.\tau')$. If there is no such S, we return *failure*. Applying this substitution S, we solve the sub-problems D recursively.

Recursive constraints and constraint variables For a recursive constraint problem, $\tau <: \mu\kappa.\mathcal{K}$, we first check whether the same problem has already been asked and stored in the cache memo M. If it is in M, we just return an empty substitution.

Otherwise, we add the problem to the cache, then calls the algorithm recursively for μ-expanded problem, $\tau <: \{\kappa \Rightarrow \mu\kappa.\mathcal{K}\}\mathcal{K}$.

Resolution of the full instances

We apply the resolution function *kresolv* to the set of the full instances $F = \{\mathtt{x}^1_{\mathcal{K}_1} : \tau_1,\ \ldots,\ \mathtt{x}^n_{\mathcal{K}_n} : \tau_n\}$ in the expression e'_t, which are obtained in the initialization phase as follows:

$$
\begin{aligned}
&\text{let } \{\mathtt{x}^1_{\mathcal{K}_1} : \tau_1,\ \ldots,\ \mathtt{x}^n_{\mathcal{K}_n} : \tau_n\} = F \text{ in}\\
&S = \mathit{kresolv}(\emptyset, \{\tau_1 <: \mathcal{K}_1 \;\&\ldots\&\; \tau_n <: \mathcal{K}_n\})\\
&e''_t = S(e'_t)
\end{aligned}
$$

$$
\begin{array}{l}
\mathit{unify\_substs}(S_1, S_2) = \\
\quad \text{let } \{\nu_1, \ldots, \nu_n\} = \mathit{Dom}(S_1) \cup \mathit{Dom}(S_2) \\
\quad \text{let } \Sigma_1 = \mathit{unify}(S_1(\nu_1), S_2(\nu_2)) \\
\quad \text{let } \Sigma_2 = \mathit{unify}((\Sigma_1 \circ S_1)(\nu_2), (\Sigma_1 \circ S_2)(\nu_2)) \\
\quad \ldots \\
\quad \text{let } \Sigma_i = \mathit{unify}((\Sigma_{i-1} \circ \ldots \circ \Sigma_1 \circ S_1)(\nu_i), (\Sigma_{i-1} \circ \ldots \circ \Sigma_1 \circ S_2)(\nu_i)) \\
\quad \ldots \\
\quad \text{let } \Sigma_n = \mathit{unify}((\Sigma_{n-1} \circ \ldots \circ \Sigma_1 \circ S_1)(\nu_n), (\Sigma_{n-1} \circ \ldots \circ \Sigma_1 \circ S_2)(\nu_n)) \\
\quad \text{if one of the unifications fails, return } \mathit{failure} \\
\quad \text{otherwise, return } \Sigma_n \circ \ldots \circ \Sigma_1 \circ S_1 \\
\\
\mathit{unify\_substs}(S_1, S_2, \ldots, S_n) = \mathit{unify\_substs}(\ldots(\mathit{unify\_substs}(S_1, S_2), \ldots), S_n) \\
\quad (\text{return } \mathit{failure} \text{ if any internal } \mathit{unify\_substs} \text{ fails})
\end{array}
$$

Figure 10.2: Unification of substitutions

The substitution S obtained by *kresolv* is applied to e'_t, the constraint annotated expression (= a compilation unit). Since S makes $S(\tau_1) <: S(\mathcal{K}_1), \ldots, S(\tau_n) <: S(\mathcal{K}_n)$ are all valid, validity of the full instances $\mathtt{x}^1, \ldots, \mathtt{x}^n$ in $S(e'_t)$ are also assured.

10.5.3 Second resolution with `dyn`-unification

Finally, we `dyn`-unify all the remained free dynamic type variables in type annotations – dynamic variables never abstracted by any generalization – in the compilation unit e''_t in order to make the given program becomes well-typed and avoid the coherence problems of generic primitives.

$$
\begin{array}{l}
\text{let } S_{\mathtt{dyn}} = \{\delta \Rightarrow \mathtt{dyn} \mid \delta \in DV(e''_t)\} \text{ in} \\
e'''_t = S_{\mathtt{dyn}}(e''_t)
\end{array}
$$

We obtain the above e'''_t as the final result of the ambiguity resolution of the input e_t.

The entire algorithm W_g of the generic type reconstruction is as follows:

$$
\begin{array}{rl}
W_g(e_t) = & \text{let } e'_t = \mathit{kassign}(\emptyset, e_t) \\
& \text{let } \{\mathtt{x}^1_{\mathcal{K}_1} : \tau_1,\ \ldots,\ \mathtt{x}^n_{\mathcal{K}_n} : \tau_n\} = \mathit{fullinsts}(\emptyset, e'_t) \\
& \text{let } S = \mathit{kresolv}(\emptyset, \{\tau_1 <: \mathcal{K}_1 \ \& \ldots \& \ \tau_n <: \mathcal{K}_n\}) \\
& \text{let } e''_t = S(e'_t) \\
& \text{let } S_{\mathtt{dyn}} = \{\delta \Rightarrow \mathtt{dyn} \mid \delta \in DV(e''_t)\} \text{ in} \\
& S_{\mathtt{dyn}}(e''_t)
\end{array}
$$

10.5.4 Correctness of resolution

We roughly discuss the correctness of this constraint type reconstruction for ambiguity resolution.

(∗ resolution of a list of full instances ∗)
$kresolv(M, \{\tau_1 <: \mathcal{K}_1 \ \& \ldots \& \ \tau_n <: \mathcal{K}_n\}) =$
If it is the first call of the algorithm, we reset the call number counter before each following recursive calls of *kresolv*
let $S_1 = kresolv(M, \tau_1 <: \mathcal{K}_1)$ in
...
let $S_n = kresolv(M, \tau_n <: \mathcal{K}_n)$ in
(∗ Merge the results ∗)
let $S = unify\_substs(S_1, \ldots, S_n)$ in
if one of those calculation of S_i and S fails, immediately fails
otherwise, return S

(∗ resolution of type pattern constraints ∗)
$kresolv(M, \tau <: [\sigma_1 \ D_1 \mid \ldots \mid \sigma_n \ D_n]) =$
Increment the call number counter. If it reaches the maximum limit, the whole algorithm fails immediately
let $1 \leq p \leq n$ is the smallest integer such that $S_p = kresolv(M, \tau <: \sigma_p \ D_p)$
if p does not exist, return *failure*
otherwise, return S_p

(∗ resolution of a type pattern with derived constraints ∗)
$kresolv(M, \tau <: \forall V.\tau' \ D) =$
let $S = unify(\tau, \tau')$ if the unification fails return *failure*
let $S' = kresolv(S(M), S(D))$
if the above fails return *failure*
otherwise, return $S' \circ S_{|Dom(S)\setminus V}$

(∗ resolution of recursive constraints ∗)
$kresolv(M, \tau <: \mu\kappa.\mathcal{K}) =$
if $(\tau <: \mu\kappa.\mathcal{K}) \in M$ then return $\emptyset$
otherwise,
let $M' = M\{\tau <: \mu\kappa.\mathcal{K}\}$
let $\mathcal{K}' = \{\kappa \Rightarrow \mu\kappa.\mathcal{K}\}\mathcal{K}$ in
let $S = kresolv(M', \tau <: \mathcal{K}')$

Figure 10.3: Ambiguity resolution function *kresolv*

Normal typing

Constraint type reconstruction W_g produces a type substitution and apply it to the given expression. Theorem 3 assures the typing of the substituted expression is preserved under the normal typing layer.

Constraint assignment

Constraint assignments algorithm *kassign* takes a type-annotated expression and translates it to a constraint annotated one. Assigned constraints are created following the constraint typing rules (kt-$*$). The main work of the assignment is to find the polymorphic instances governed by each generic definition. This is done by using sets of generalized dynamic type variables. Since type substitution by later phases never affects these generalized dynamic type variables, a polymorphic instance which belongs to some generic definition at the assignment is also governed by the same generic definition after the end of all reconstruction phases. Therefore the constraint assignments are still valid in the final result.

Ambiguity resolution *kresolv*

In this main part of the generic constraint type reconstruction, the algorithm tries to find a proper substitution S_i for each full instance $\mathtt{x}^i_{\mathcal{K}_i} : \tau_i$, such that $S(\tau_i) <: S(\mathcal{K}_i)$. The correctness of the algorithm *kresolv* is stated as follows:

Conjecture 1 *Let τ be a type expression, $\mathcal{K}$ be a constraint and M be a memo function which contains of problems in the form of $\tau <: \mu\kappa.\mathcal{K}'$. Then $kresolv(M, \tau <: \mathcal{K})$ returns a substitution S such that:*

$$\textit{if for all } (\tau <: \mu\kappa.\mathcal{K}') \in M,\ S(\tau) <: S(\mu\kappa.\mathcal{K}') \textit{ is valid, then } S(\tau) <: S(\mathcal{K})$$

This conjecture is unfortunately not proven yet, but should be done by induction over the algorithm *kresolv*. Stability of the correctness of the problems under substitutions is assured by Theorem 34.

`dyn`-unification

At this stage, all the free dynamic type variable in the annotations are completely free: they were never instantiated by *kresolv*. This means that instantiation of these free dynamic variables to `dyn` never break the validity of the generic full instances reconstructed by *kresolv*.

10.6 Future work

10.6.1 Termination of ambiguity resolution

Termination against non-admissible constraints

Using the call number restriction of the resolution algorithm *kresolv*, we can also make sure the termination of resolution against non-admissible constraints. Therefore we will no longer require admissibility restriction of user-defined generic value definitions.

Relaxing the call number restriction

Currently the resolution is restricted by the total call time of the algorithm *kresolv* to assure its termination. This is too strong in some cases. For example, if a type τ and an admissible constraint $\mathcal{K}$ are closed, the resolution of $\tau <: \mathcal{K}$ is equivalent to checking of the predicate by $kinst^{+}$. In such a case, restricting the number of recursive calls of *kresolv* ($=kinst^{+}$) is awkward, since Theorem 30 assures its termination. Therefore the call time restriction can be removed when the resolution *kresolv* is equivalent to type checking $kinst^{+}$. This has the following benefits: even when automatic resolution cannot guess the type of a generic instance due to the call time restriction, we can make it pass the resolution by constraining its type to instantiate all its dynamic type variables.

Note that this relaxation of the restriction cannot be applied to ambiguity resolutions of non admissible constraints, since $kinq^{+}$ may not terminate for them. Therefore we will have to still use the classical total call number restriction for non admissible constraints.

10.6.2 Coherence problem of mutually related full instances

In the resolution of full instances, $kresolv(M, \{\tau_1 <: \mathcal{K}_1 \,\&\ldots\&\, \tau_n <: \mathcal{K}_n\})$, we solve each sub-problem $\tau_i <: \mathcal{K}_i$ independently and never consider the effects between sub-problems. Therefore this is not the complete resolution. For example:

```
generic one = case
| int   ⇒ 1
| float ⇒ 1.0
in
generic two = case
| string ⇒ "2"
| int    ⇒ 2
in
one = two;;
```

In the above program, we have actually one possible resolution `(one : int) = (two : int)`, however, our resolution algorithm cannot find it. It resolves each full instance independently and obtains `one : int` and `two : string`. These two resolutions contradict each other; they try to unify the same dynamic type variable to different closed types. Therefore they cannot be merged, and the algorithm cannot resolve the ambiguity and asks the user to add some more explicit type information.

Generally speaking, when more than one generic instances share the same dynamic type variables, each instance type reconstructions relate mutually, and for the complete type checking, we have to try every combination of type case selections of the mutually related instances. But we do not do this complete search between related instances, since we do not suppose such complex use of generic values in extensional polymorphism. Actually almost of all the examples we had in this thesis have no such mutually related generic instances. When there are such mutually related instances and the constraint type reconstruction fails for their instantiation, the user can still instantiate their types by hand to eliminate shared dynamic type variables to make the instances independent.

In future we can introduce this full type case search of related instances, but it will be not so successful, compared to the complexity increase due to this extension. The number of combinations we have to search is exponential to the number of mutually related full instances. In spite of this complexity increase, it might not always provide better resolutions than the current. For example:

```
generic plus = case
| int → int → int ⇒ (+)
| float → int → float ⇒ fun x y → x +. float y
| int → float → float ⇒ fun x y → float x +. y
| float → float → float ⇒ (+.)
in
generic one = case
| float ⇒ 1.0
| int → 1
in
plus one one;;
```

The expression `plus one one` has four possible valid resolutions as follows:

```
(plus : int   → int   → int)    (one : int)   (one : int)
(plus : float → int   → float)  (one : float) (one : int)
(plus : int   → float → float)  (one : int)   (one : float)
(plus : float → float → float)  (one : float) (one : float)
```

Though the full resolution between instances could find all of them, it cannot choose one automatically, since there is no clear priority between these four resolutions (`plus : int → int → int` is the first type case of `plus`, but on the other hand `one : int` is not its first case, and so on.) In this case, the coherence problem of the program cannot be resolved using the type case definition ordering. Consequently we would have to select one of them explicitly by giving type annotations anyway. We could introduce some ordering between these possible typings and let the resolution choose one with the biggest priority, but it seems that there would be no such one which can be easily understood by programmers.

10.7 Conclusion

We have constructed constraint type reconstruction algorithm, which infers well-typed annotation of expressions under the generic constraint type system. The two-layer design of the normal and generic typing is also applied to the type reconstruction: the constraint type reconstruction is an independent algorithm phase of the normal reconstruction. A program is first passed to the normal type reconstruction algorithm and its type annotations are inferred so that it is well-typed under the normal typing. Since the result of the normal reconstruction may be still too general (or invalid) for the generic constraint type system, it is then passed to the constraint type reconstruction to instantiate the too general types of generic full instances so that it can be also well-typed under the generic constraint typing.

A full generic instance whose type is reconstructed by the normal type reconstruction can be ambiguous: it may have more than one possible valid instance typing which provide incompatible semantics. The automatic ambiguity resolution selects the default typing and semantics for such an ambiguous generic instance, using the definition ordering of type cases. If there are more than two type case applicable, it tries to choose the firstly defined. Ambiguity is resolved for each generic full instance independently. Unfortunately, when generic instances share dynamic type variables and relate each other mutually, this instance independent resolution does not solve their coherence problem. But it seems to be also true that there is no automatic way to resolve their ambiguities. Manual type constraints may be the only way to eliminate such ambiguities.

After all the full instance types are reconstructed and their ambiguities are resolved, the reconstruction algorithm unifies all the remained free dynamic type variables in annotations with the closed type `dyn`, in order to make the whole program well typed under the generic constraint typing. This also solves the coherence problem of generic primitives.