année : 2002

THÈSE

présentée à

UNIVERSITÉ DENIS DIDEROT – PARIS 7

pour obtenir le titre de

DOCTEUR EN SCIENCES

Spécialité:

Informatique

soutenue par

Jun FURUSE

le 18 Décembre 2002

Titre:

Extensional Polymorphism: Theory and Application

Jury

Président	Guy Cousineau
Rapporteurs	Richard Kieburtz Atsushi Ohori
Examinateurs	Xavier Leroy Simon Peyton Jones
Directeur de thèse	Pierre Weis
Professeur invité	Roberto di Cosmo

Chapter 1

Introduction

1.1 Motivations

1.1.1 First motivation: overloading

ML is a great language. Based on the typed λ-calculus, programs are statically checked by its type system so that the well-typed programs never produce errors. Once the type soundness of a program is checked, we no longer need any type checking at run time. Its powerful combination of polymorphism and higher order functions allows us to write various algorithms generic to different data types. Additional to these theoretical advantages, thanks to the great effort of compiler optimizations, library maintenance, and documentations, ML is no longer a toy for researchers, but a real programming language: these days, many people from outside of research fields have heard about the language and start writing ML programs.

Just as the other programming languages have their own difficulties to master, there are some hard learning points in ML, too, in spite of its advanced features. For example, its value polymorphism restriction is tricky: who can guess just at a glance that it relates with the typing of references? The ML module system may be another obstacle: the word "functor" from the category theory may sound complicated to learn. But much before colliding with these famous difficulties, many novice people encounter another ML learning barrier: a limitation of ML polymorphism.

ML newcomers are embarrassed when they hear that each data type such as lists, arrays, and strings has its own length function. They complain frequently why these functions are not *overloaded* to one polymorphic `length` function. "if we write `length "hello"` for example, it is obvious that this `length` is for a string. Why do we have to use different identifier names such as `List.length`, `Array.length`, and `String.length`?"

Overloaded arithmetic operators

Another irritating problem concerns the treatment of arithmetic operators and values. This problem has been addressed for a long time by functional programming languages that have given different partial solutions to the problem: SML[Gil] and Haskell[HJW92] feature overloaded arithmetic operators.

In SML, some basic math operators including "+" are predefined overloaded values:

```
1 + 2      ⇝ 3
3.14 + 1.0 ⇝ 4.14
```

But this "+" operator is not treated as a regular function, since the overloading of SML must be statically resolved. Hence "+" cannot be abstracted into another function as in `double`:

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

Here `double` does not inherit the overloading of "+": SML'90 just rejects the definition of `double` as a type error, and SML'97[Gil] assigns `double` a monomorphic type `int` → `int` using the default type for "+", `int` → `int` → `int`. Also, SML does not allow users to define their own overloaded values.

The original version of Caml (V3.1) featured user's defined overloaded values (possibly polymorphic) with static overloading resolution. For instance, the fully polymorphic `length` overloaded function was definable. However, as in SML'97, the definition of `double` was monomorphic using a default resolution type.

Haskell goes a long way beyond SML and Caml V3.1, since basic math operators are overloaded in such a way that `double` is fully polymorphic and gets polymorphic typing `double :: Num a ⇒ a → a`.

We now turn our attention to our favorite language (i.e. the language we use to implement our prototypes and the language we want to extend) Objective Caml[Ler]. The situation is remarkably simple now: there is no overloading at all. Hence there are two different operators `(+)` and `(+.)` for addition of integers and floating pointer numbers. Having these two different adding functions irritates not only novice users, but also experts: is there any Objective Caml programmer who has never been stressed by their distinction?

```
# 1.2 + 3.4;;
Characters 0-3:
  1.2 + 3.4;;
  ^^^
This expression has type float but is here used with type int
# 1.2 +. 3.4;;
- : float = 4.6
```

Worse than this, the standard library of Objective Caml version 3.06 defines additional five arithmetic types (`int32`, `int64`, `nativeint`, `big_int`, `num`). Therefore we have seven independent plus operators!

Our primary goal is to equip Objective Caml with regular overloaded arithmetic operators such as `+` which can be abstracted polymorphically (as in our `double` example).

User definable overloading

In Haskell, users can define overloaded values using type classes. For example, we may write:

```
class Length α where length :: α → Int

instance Length String where length = String.length
instance Length Array α where length = Array.length
...
```

However, type class definitions have several restrictions which prevent free definitions of overloaded instances.

The following type class definition for C like addition function with automatic type casts is rejected in Haskell 98, since its type class parameter is restricted to unary:

```
class Plus α β γ where plus :: α → β → γ

instance Plus Int   Int   Int   where plus = plusInt
instance Plus Float Float Float where plus = plusFloat
instance Plus Int   Float Float where
  plus = \x y → plusFloat (intToFloat x) y
instance Plus Float Int   Float where
  plus = \x y → plusFloat x (intToFloat y)
```

The multi type class parameter extension[1] of Haskell gets rid of this restriction, but we soon face to problems with ambiguity. This `plus` is not practical: for instance, typing of a simple expression `plus 1 2` cannot be inferred automatically to `Int`, even though only the first class instance definition is applicable in the above definition. It can be justified by the *openness* of Haskell overloading: there may be another instance declaration for `Int` $\rightarrow$ `Int` $\rightarrow \tau$ for some type τ somewhere in the program. As proposed by Jones[Jon00], introducing the following functional dependency between these class parameters can solve this situation:

```
class Plus α β γ | (α,β) ⇝ γ
```

which specifies the instantiation of the parameter γ is uniquely determined by instance types of α and β. But functional dependencies must be written explicitly by users.

Our secondary goal is to propose another approach of overloading which matches with the ML semantics. We define an overloaded value by declaring its all instances at one place, once and for all. Our definition expression of overloaded values, `generic` expression, is an extension of the ML `let` polymorphic binding with type indexed multiple cases:

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

[1] Such extension is enabled in Hugs specifying `-98` option.

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

Having a *closed* set of overloaded types, the type of the expression `plus 1 2` is inferred as `int` automatically. The selection of type cases will be done in the same way as pattern matching: an overloaded instance matches its instance type with the pattern types in the order of the declaration and uses the first match when there are more than one possible cases. Using this selection priorities of type patterns, instances of overlapped types can be overloaded freely. In type classes, this is not possible except some special cases, since there are no priority between overloaded instance definitions. The following example is not possible to be defined in Haskell, since the defined types overlap each other and it is ambiguous in the instance type `(int * int)` → `int`:

```
generic find_int =
| ∀δ.(int * δ) → int ⇒ fun (x,y) → x
| ∀δ.(δ * int) → int ⇒ fun (x,y) → y
;;
```

But in the extensional polymorphism, thanks to the type case ordering, the first case is selected when it is evaluated under `(int * int)` → `int`:

```
# find_int (1,2);; (* the first match is selected *)
- : int = 1
```

1.1.2 Second motivation: ad-hoc polymorphism

Certainly, the most often asked feature on the Caml mailing list is a "simple" facility to print *any* ML value. Novice ML users often request this almighty ML value printer function `print` which prints out values of *any* type, just because they see the interactive top-level session printing out *any* evaluation result of *any* type on the screen.

Such a function should be quite useful for debugging, since the so-called "`printf` debugging" is still preferred to the sophisticated source-level replay debugger [TA95]. In addition, this top-level value printer will give users a printer for each data type for free; we would have:

```
# print 1;;
1
- : unit = ()
# print (1,"hello");;
(1,"hello")
- : unit = ()
# print [Some (fun x → x), None];
[Some <fun>; None]
- : unit = ()
```

`print` seems to be implemented easily as a programming interface to the ML value printer of the interactive top-level session. But the top-level value printer implementation takes not only a value to print but also its type in order to select appropriate printer and obtain the corresponding type constructor names. Therefore `print` function must be ad-hoc polymorphic so that it can send its argument and its type to the internal top-level value printer.

To implement the `print` function, and to type it properly is a real challenge that needs much more theoretical foundation and much more implementation efforts than mere overloading of operators. This is due to the fact that `print` provides a lot of different behavior for each data type. This is generalization of overloading values called *ad-hoc polymorphic* values. (They are also called *non-parametric polymorphic*.) Ad-hoc polymorphic values have type-dependent semantics: in evaluation, an ad-hoc polymorphic value `g` is regarded as a function which takes its evaluation type context τ and returns the semantics of the type τ:

$$\texttt{g} : \quad \tau \quad \longmapsto \quad \text{semantics of } \texttt{g} \text{ of } \tau$$

For example, the overloaded `length` is a function as follows:

$$\begin{array}{lll}
\texttt{length :} & \alpha\ \texttt{list} \rightarrow \texttt{int} & \longmapsto\ \texttt{List.length} \\
\texttt{length :} & \alpha\ \texttt{array} \rightarrow \texttt{int} & \longmapsto\ \texttt{Array.length} \\
\texttt{length :} & \texttt{string} \rightarrow \texttt{int} & \longmapsto\ \texttt{String.length}
\end{array}$$

Thus, ad-hoc polymorphism promises two of the most frequently requested features in the ML community: overloaded values and the value printer function. However, type-dependent semantics of ad-hoc polymorphism cannot be covered by ML polymorphism. ML polymorphism is *parametric*: semantics of a polymorphic value is independent from its parameter type instantiations. ML compilation without run time types is an optimization obtained from this type independent semantics, but this optimization prevents the integration of ad-hoc polymorphism into ML.

Novice ML programmers cannot accept this explanation easily these days, since many of them have already experienced the object oriented programming: "in my favorite object oriented language X, methods for length of various data types share the same one name `length`. Why not also in ML? I cannot imagine any difficulties of extending ML with overloading!" Of course, the last statement is not true; there are many non trivial design choices of syntax, semantics, typing and compilation. But, regardless of how it is difficult to obtain, the lack of ad-hoc polymorphism may make ML less appealing to these innocent people nowadays, because of the spread of strongly typed object oriented languages with method overloading such as Java[GM96].

1.1.3 Third motivation: value I/O

ML users enjoy its strong typing discipline which validates type soundness of programs and rejects all potential run time type errors.

Objective Caml provides polymorphic primitives to store ML values on persistent storage:

$$\begin{array}{l}
\texttt{output_value : } \forall\alpha.\texttt{out_channel} \rightarrow \alpha \rightarrow \texttt{unit} \\
\texttt{input_value : } \forall\alpha.\texttt{in_channel} \rightarrow \alpha
\end{array}$$

Expression `output_value oc v` takes an ML value `v` of *any* type, marshals the internal value structure of `v` to a string, and outputs it to the output channel `oc` of a file or a stream. An output value is read back to a program by `input_value ic`, through an input channel `ic`. The string representation of a value is unmarshaled and its internal structure is rebuilt in the memory of the reader program. For example:

```
# let oc = open_out "data" in
  output_value oc 42;                  (* output an integer to a data file *)
  close_out oc;;
- : unit = ()

# let ic = open_in "data" in
  (input_value ic : int);;           (* input the value stored in the file *)
- : int = 42
```

Thus the value I/O frees programmers from writing their own printers and parsers for ML value storage in files, and it is used in many Caml applications.

However, the value I/O has a big problem: the types of the primitives `input_value` and `output_value` are far more too general. The `input_value` primitive reads an encoded string of a value and reconstructs the same structure as the original, but the polymorphic type scheme of `input_value` primitive does not prevent this reconstructed structure from being used as a value of *any* type, even incompatible one from the original type. Such incompatible value inputs can cause fatal run time errors:

```
# let ic = open_in "data" in
  print_string (input_value ic);;   (* wrong use of an integer as a string *)
Segmentation fault
```

This lack of run time type checking of input values makes the value I/O unreliable, since debugging around the value I/O without the help of typing discipline is almost impossible. Actually, we have already enough numbers of victims of this unsafety since the first appearance of value I/O. This insecurity could be fixed by type checking of input values against their expected types in the reader programs. Then why are these checks not performed? The answer is the *same* as for ad-hoc polymorphism: due to the lack of types of ML run time programs, the run time type checking of input values is just impossible.

1.2 Run time types

Even though ad-hoc polymorphism and safe value I/O have been repeatedly requested for a long time by many ML users, from novices to type researchers, the lack of run time types in ML programs has precluded their implementation. But is this “no types at run time” a golden rule of ML? As “well-typed programs cannot go wrong”, said Robin Miner [Mil78], thanks to the type soundness theorem, it is true that compilation of ML needs no type operation at run time. But it is *never* a necessary condition of ML run time programs. We claim that run time types should be available even in well-typed ML programs in order to provide type-dependent semantics: well-typed programs never go wrong, but run time types are still useful and necessary for ad-hoc polymorphism and safe value I/O.

In this thesis, we present *extensional polymorphism*, a framework of ad-hoc, non-parametric polymorphism which extends ML. The operational semantics of ML is reformalized with types – it is defined over full type-annotated expressions. When an instance of a polymorphic value is evaluated, its type instantiation is implicitly applied to the corresponding semantic value, in order to provide appropriate type context for the instance evaluation. *Generic values*, ad-hoc polymorphic values in extensional polymorphism, make use of these type contexts for their type-dependent semantics.

1.3 Realizable design

We are not just simply going to build a theory of extensional polymorphism. Our final goal is integration of extensional polymorphism into an existing ML implementation and the provision of type-dependent semantics of ad-hoc polymorphism and safe value I/O system to the ML users. Hence, we take special care to ensure a "clean" integration of extensional polymorphism:

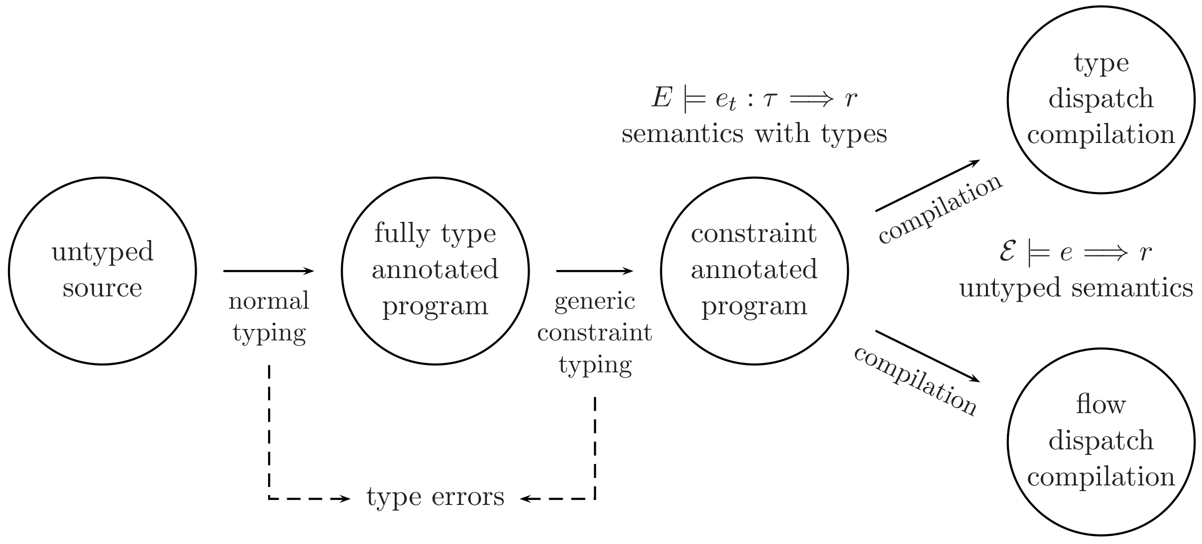


Figure 1.1: Typing and compilation phases

No integration side-effect

All of the semantics, typing and efficiency of the classical features of ML must be preserved and remain compatible even after the integration of extensional polymorphism. In other words, any classical working ML programs must be well-typed and evaluated to the same results with no extra run time overhead.

Type-dependent semantics of extensional polymorphism is compiled to standard untyped ML expressions. The application of evaluation type context instantiation of extensional polymorphism semantics is translated to *run time type dispatching*, an application of *run time* *types*, the ML value representation of types. This application of run time types is suppressed only for the compilations of generic values. Non-generic programs are compiled as usual, and therefore are evaluated with the same efficiency.

Modular design

Integration of extensional polymorphism will require lots of additional code to the ML compiler. The additions must be modular, and be separated from the original source as much as possible, not to complicate the original part of the compiler source code.

Too much intricate mixing of the code for extensional polymorphism into the original compiler source modules would make the entire system difficult to understand. We must avoid such a situation and keep the original modules unchanged, so that they can be maintained even by someone who does not know anything about extensional polymorphism.

To achieve this separate implementation strategy of "non-generic" (i.e. classical) and "generic" part of the compiler, the type system for extensional polymorphism is formalized as two separated sub-systems (Figure 1.1). Programs are first verified by the "normal" type system, which is based on the Hindley-Milner ML type system with a small extension. In this normal typing layer, all the generic values are treated as total, that is, they can take any instance type of their type schemes, even if they have no well-defined semantics in their instance types. Programs must also be checked by the second *generic constraint typing* layer. This additional type checking pass resolves and checks the validity of instances of non-exhaustive generic values.

Through typing by these two layers, a program is annotated type informations and its semantic soundness is proved over the semantics with types. Then, for the efficient evaluation, the program is compiled into an untyped intermediate language. Type instantiation of the semantics with types which is required for generic instances is translated to run time type dispatching. We will provide another compilation method called *flow dispatch compilation*. In this new compilation, what are dispatched to the user defined generic instances are not run time types, but *generic flows*, integer graphs which encodes each type case selection. This flow dispatching will supersede the classical run time type dispatching, since it provides better type case selection semantics with better run time performance.

1.4 Outline

Chapter 2: Generic core system

To start our study of extensional polymorphism, we first build our target language called GML, an extension of ML, which supports non-parametric, ad-hoc polymorphic values called *generic values*. In this chapter, generic values are restricted to primitives and derived values whose genericity are derived from uses of other generic values in their definitions.

Different from the normal non-generic values, semantics of generic values are type-dependent. Therefore the semantic system of GML must be defined over type-annotated expressions. The semantics of a generic primitive is modeled as a primitive function which takes an *evaluation type context*, a type for evaluation of its instance. The definition of a derived generic value is evaluated to a *generic closure*, which takes its evaluation type context and instantiates type contexts of the internal instances for their correct evaluations. Instances of generic primitives and derived generic values are evaluated as implicit applications of their evaluation type contexts to these primitive functions or generic closures.

Similarly to the semantic rules, the GML type system is also defined over fully type-annotated expressions. To indicate the implicit type applications in semantics, generic values have polymorphic type schemes quantified by *dynamic type variables*, a new set of type variables different from the *regular type variables* for the normal parametric polymorphic values. The soundness of the type system assures that execution of a well-typed GML program is free from potential run time type errors. We also define a type reconstruction algorithm, which is complete with respect to the typing rules.

Chapter 3: Compilation of GML

In normal ML, thanks to its type soundness, the evaluation requires no type information at run time, and therefore programs are executed efficiently without run time type checking. In GML evaluation, we do not need to perform run time type check either, but its evaluation system still needs type manipulations, since semantics of generic instances are type-dependent: the semantics is defined using implicit applications of their evaluation type context, which instantiate type annotations of expressions. Such semantics with type instantiations is difficult to implement directly. Instead, we compile GML expressions to an intermediate ML language without types.

The type instantiation mechanism in the GML evaluation system is compiled to *run time type dispatching*, functional applications of *run time types*, normal ML values which represent type expressions. Generic primitives and their derived generic values are compiled to functions, and they are applied to run time type dispatching, which supply appropriate evaluation type contexts. Functional applications and abstractions for dispatched run time types take occur only in the compilations of generic definitions and instances. The other normal non-generic part can be evaluated as efficiently as in normal ML implementations.

Chapter 4: Dynamic values

As an illustration of the power of extensional polymorphism framework, we implement *dynamic values* by means of two polymorphic generic primitives.

A dynamic value is an abstract value of type `dyn` which is represented internally as a pair (v, τ) where v is a value and τ is its type. Two primitives, `dynamic` and `coerce` are associated to this approach: the `dynamic` primitive encapsulates an ML value into a dynamic value. The `coerce` primitive takes the content of the dynamic value back into some evaluation type context, checking that the internal type information is compatible with its dynamic type context. This type hiding capability allows an implementation of heterogeneous data structures and overloaded functions in ML with the cost of explicit encapsulation and decapsulation of dynamic values.

Both the `dynamic` and `coerce` primitives must have access to type information at run time: `dynamic` must retrieve the types of values to be encapsulated, and `coerce` should receive expected type contexts for coercions. Therefore run time types are mandatory for them, but they are not available in pure ML, since compilation usually lacks types at run time. The compilation scheme of extensional polymorphism provides this mechanism of run time types and their dispatch.

By compiling both primitives as functions with additional run time type arguments, extensional polymorphism offers dynamic values for free.

Types inside dynamic values must be closed type schemes, since dynamic values and their type informations may be used out of their context of definition. Similarly to free type variables that cannot be exported outside of the module boundary, ungeneralizable type variables must not be encapsulated into dynamic values. On the other hand, the content of a dynamic value with a polymorphic type scheme must be coerced to a polymorphic value. We therefore introduce additional static typing rules for the dynamic primitives allowing this polymorphic encapsulation of dynamic values. The types of values to be encapsulated by the `dynamic` primitive must be checked for generalization. We propose an approach that makes this generalization check explicit and restricted to the arguments of the `dynamic` primitive. Even under this restriction, users can define *derived dynamizers*, derived generic values which create dynamic values, although their type encapsulation is restricted to closed types. We have two ways of handling type variables related to dynamic values which cannot be proven generalizable by our approach. The first method just report them as type errors. We call the second method `dyn`-unification: the ungeneralizable type variables are erased automatically by unification with the closed type `dyn`. Although they are made closed, the type `dyn` provides pseudo heterogeneousness, unlike unification with the other closed types. For coercions of polymorphic contents, `coerce` cannot be a function, due to the value polymorphism restriction. Hence, `coerce` must be a language construct reminiscent of the usual pattern matching syntax.

Chapter 5: Type safe value I/O

As another important application of generic primitives, we introduce a type safe value input and output system — a type safe way of exchanging values between possibly different ML programs.

In value I/O, an ML value structure is marshaled to a string and output to a file or a stream. At input this string is unmarshaled and decoded back into an ML value tree automatically. Using the universal value printer and marshaler of value I/O, programs can save and load any ML value to a file and exchange values between different programs easily, without writing any custom value printer or unmarshaling parser. Thanks to this easiness, many middle or large class Objective Caml applications use value I/O.

But this value I/O has a big drawback: value I/O in ML is not type safe, since type compatibility is not checked at run time. It has been the last type insecurity known in the Objective Caml system for a long time. ML programs have no type information at run time and therefore run time type checking is clearly impossible. Without the help of typing discipline, it is impossible to fix programming mistakes around value I/O.

Extensional polymorphism avoids this situation and provides a type safe value I/O system, attaching type information, called external types of values, at output. At input, the attached external type is checked against the type context of the input program, and allows unmarshaling only when the external type is proven compatible with the type context. This appears to be quite similar to the dynamic value mechanism. However, dynamic values are not adequate to provide a type safe mechanism for the value I/O.

Indeed, external types are not run time types. In run time types, the data types are represented and identified by their names. Such identification works only inside one program or between different invocations of the same identical program. When exchanging values between different programs, there is no guarantee that data types have the same internal value structure, even if they share the same name. To provide safe value I/O, we need a universal identification of data types throughout different programs. In external types, data types must be represented by their semantics, based on their definitions. In Objective Caml, data type definitions per se contain too much information for this identification. Value layouts of Objective Caml data types are independent of names, either type names, constructors, or record labels. These names in type definitions are not necessary for type safety. Therefore data types in external type representation are identified by their nameless definitions obtained by erasing all the names from the original definitions. To avoid unbound size growth of external types, we encode these nameless type definitions as mere check-sums, the data type *digests*. Due to the existence of abstract data types and a separate module compilation scheme, the calculation of data type digests is sometimes statically impossible at compile time. In such case, we embed a small piece of code that computes the digests at program load time.

Data type identification by nameless definitions may sound strange from the point of view of the semantics of data types in ML. But this identification between different data types in value I/O has practical use. Suppose that in a program, data type names are modified through program version-updates for example. Using nameless definitions, the values of the older type stored in files can be read back as values of the newer type in a safe and well-defined manner.

Chapter 6: Merging dynamic values and safe value I/O

The dynamic values and safe value I/O system were formalized as two different notions. But they use very similar mechanisms — value-type pair encapsulation and decapsulation using the run time type dispatching of extensional polymorphism compilation. We merge the two notions into one. It is not just a mere simplification, it also makes them cover their shortages each other.

In the previous approach of dynamic values, data types were identified by their names only. Type safety of dynamic values was assured by the assumption that data types with the same name always define the same data structure. But this assumption is no longer correct in the combination of dynamic values and value I/O. Run time types encapsulated in dynamic values must also contain data type digests and be universal throughout different programs, since dynamic values can now be exported outside of program by value I/O. When an exported dynamic value is read back, we check that the value-type pair has the type `dyn` at first. At the coercion of this imported dynamic value, we must check not only the names of data types but also their data type digests, since the value may come from a different program.

In the framework of safe value I/O, data types are identified by their nameless definitions: when they define the same data structure in the view of the internal value representation, they are compatible even if their definitions can have different names. This identification of data types provides an interesting and practical safe data exchange between different data types, but in some cases this identification is not welcome. By adding data type name information into the external types and comparing both digests and names at input, we can introduce another more strict data type identification scheme in value I/O.

In the unified system, both primitives for dynamic values and safe value I/O handle “dynamic values with digests”, value-type pairs containing *both* data type names and digests. Encapsulation primitives of values and types are now merged into a single construct of the language: the `dynamic` expression. On the other hand, we keep two decapsulation primitives `coerce` and `import` separated and provide two kinds of dynamic value decapsulations. The `import` primitive only checks the data type digests, while the `coerce` primitive also compares the data type names. Checking digests at every decapsulation is necessary to close the security hole of exporting dynamic values.

Chapter 7: Generic definitions

We extend our language with *generic bindings* so that we can define our own generic values in the language. In Chapters 2 and 3, we have focused mainly on the type aware evaluation system of extensional polymorphism and its compilation using run time type dispatching. For simplification, all generic values were restricted to either primitives whose semantics are specified in a meta level with special evaluation rules, or their derived polymorphic values. Now we propose a way to specify custom semantics of generic values for users.

A `generic` expression binds an identifier and a list of type cases, pairs of defined domain types called type patterns and corresponding expressions, which specify the semantics of the type patterns. In evaluation of a user-defined generic value instance, the evaluation type context is pattern-matched with the type cases and the first matched type case provides its semantics. Listing expressions and their type schemes, we can easily define generic values such as overloaded arithmetic operators. Further more, a recursive generic definition enables polymorphic recursion. The combination of the type cases and polymorphic recursion makes it possible to define very powerful recursive generic values over type structures.

The type system of GML is extended for the generic binding. The type scheme of a user-defined generic value is given as the least general anti-unifier of its type patterns. Since user-defined generic values are declared as finite lists of type cases, their semantics may not be defined for all the instances of their type schemes. Illegal use of user-defined generics in undefined types will raise a run time exception, similarly to the pattern match failure exception in normal ML.

Generic definitions are compiled into functions which receive their evaluation type context as run time type arguments, like the other generic values. The type method selection in the evaluation of user-defined generics is done efficiently by pattern matching: type cases are translated to pattern match expressions and dispatched run time types are matched with the compilation of type patterns. If there is no match, the compilation reports a run time type mismatch error as an exception. Thus the illegal use of user-defined generics can be caught as exceptions. We also prove the correctness of the compilation algorithm of the generic binding with respect to its evaluation rules.

Chapter 8: Constraint typing

Generic bindings are introduced to allow users to define generic values based on run time type pattern matching. We have added typing rules for generic bindings, but they have no power to detect illegal use of non-exhaustive generic values: even if an instance of a user-defined generic value has a type without corresponding semantics, the type system regards it as well-typed. Even though evaluations of illegal instances are reported as exceptions and can be caught easily at run time, it is preferable to detect such illegal instances statically at type checking time.

Our goal is to build a type system which can detect the illegal use of generic instances. We must first extend the notion of type schemes so that they can have non-exhaustive information of generic values called *generic constraints*. Generic constraints express the precise domain types of generic values where their semantics are defined. Intuitively, generic constraints of user-defined generics are lists of their type patterns. Derived generics have generic constraints of their dependency on other generic values. Typing to detect illegal generic instances using generic constraints forms another layer of typing system called *generic constraint typing*.

We have a clear distinction between the generic constraint typing and the "normal" ML typing, and never try to merge them into one type system; such merging would produce a huge and complicated "hybrid" type system, which should be impossible to understand by anyone except by experts of extensional polymorphism. Rather, we separate the generic constraint typing and the normal ML typing into independent typing layers: programs are first type-checked by the normal ML type system layer, which has already been introduced. At this point all type errors are detected except illegal uses of generic instances. Then the type-annotated programs are checked again in the second layer of generic constraint typing, in order to detect illegal uses of generic instances. This separate design of two type systems is much simpler than a unified hybrid system. The normal ML typing layer is kept as a usual ML type system with the dynamic type variable extension. The generic constraint typing can focus on the typing of the definitions of generic values and their instances. Also in the implementation of the type checker, this two-layer design isolates almost every part of the generic constraint typing code from the normal layer. This is very important when extensional polymorphism is implemented as an extension to an existing ML compiler. For the maintenance of the normal typing layer, it is not required to know about extensional polymorphism at all.

At generic value definitions, the generic constraint typing creates their generic constraints and binds them with defined identifiers in the typing environment. Generic instances are grouped into two categories: polymorphic instances whose types contain dynamic type variables generalized by some definition of derived generics, and full instances which do not have such generalized dynamic variables. The type validation of polymorphic instances are not performed in their very occurrences. Their non-exhaustiveness information is registered in the constraints of the corresponding derived generics which use the polymorphic instances. The typing of the polymorphic instances are delegated to the checking of full instances of these derived generics. On the other hand, types of the full instances are checked at their occurrences, since their types are the same as their type context at evaluation. The validation of generic full instances is done by simulating run time type matching of the instances. Unfortunately, the valid generic instance predicate is undecidable, due to the existence of the recursive generic definition. We need to introduce a restriction to the admissible set of recursive generic definitions and their constraints in order to make this predicate decidable.

The type soundness of the generic constraint typing system is proven: programs type checked by the both layers of typing never fall into run time type errors, including run time type match failures.

Chapter 9: Generic flow analysis

Generic constraint typing *statically* checks illegal uses of generic instances by simulating all possible run time type case selections at type checking phase. A question arises here: why do we still need to perform type pattern matching at run time again, which was already checked once at the static typing phase? The run time type matching in the compilation of user-defined generic values has two functions: one is checking the validity of generic instances and selecting type methods that need to be evaluated. Another is invoking the selected type methods. In the presence of static type checking of the generic constraint type system, the first functionality is redundant. By removing the first part and getting rid of run time type pattern matching, we can simplify the compilation of user generics. This dramatically improves the efficiency of the compilation.

Instead of pattern matching of dispatched run time types, the new compilation of generic values receive generic flows, integer graphs obtained from the typing of full generic instances. Each node of generic flows corresponds with type case selection and its index indicates the type method which should be evaluated. By dispatching generic flows to generic instances, the selection and invocation of type cases in the compilation of generic values are reduced to simple access to arrays of type methods.

The separation of type case selection from run time frees the type case selection algorithm from its limitation. Since all the type case selections are *statically* computed at typing and dispatched as generic flows, we can use a more complicated type case selection mechanism, which provides better semantics but could not be performed at run time because of its complexity: we can introduce backtracking in type case selection. With the semantics and typing of user-defined generic values using type case selection with backtracking, we can have more valid instance types and corresponding semantics of user-defined generics, preserving all of the semantics of generic instances which were valid in the original generic constraint type system.

Chapter 10: Reconstruction of constraint typing

We constructed a generic constraint typing system, which detects all invalid generic instances. The type reconstruction algorithm of generic constraint typing must also be studied. Similarly to what we did for typing systems, we use a two-layer design for type reconstruction of extensional polymorphism. In the first "normal" reconstruction phase, programs are type-annotated by the type reconstruction algorithm based on the normal ML typing algorithm, which we already studied in Chapter 2. The result of the first phase may contain generic instances whose types are still too general and invalid. The second type reconstruction phase instantiates such too general type annotations so that the generic instances can have correct instance types. If there is no valid typing of generic instances, generic instances are reported as type errors.

Similarly to other type systems with overloading, type reconstruction of generic instances may be subjected to a coherence problem. If an instance of an overloaded value has more than one possible type and those types provide different semantics each other, the instance is called incoherent and ambiguous. Generally speaking, type reconstructions for overloaded values do not vigorously attempt to resolve such ambiguities by choosing one of the incoherent typings and semantics. This is because, save for a few exceptional cases, It is since we have no notion of priority between possible instance typings and no strong reasoning for selecting one of them automatically.

In extensional polymorphism, on the other hand, we have clear ordering of the overloaded definitions: type cases of user generics are defined as ordered lists and this definition order of type cases has already been utilized in the semantics of generic instances. Also in the type reconstruction, we can use this type case definition order to resolve ambiguous typings. We introduce a priority ordering of generic flows, encoded type case selection information: flows which use type methods which are listed former in definitions have stronger priority than flows of later defined type methods. If a generic instance has more than one possible incoherent typings with different flows, the type reconstruction selects the flow with the strongest priority and instantiate the type of the generic instance to the type corresponding with the selected flow.

Because of the recursive definition of generic values, set of incoherent typings of a generic instance can be infinite, therefore we must introduce a restriction to the search space so that the algorithm can always terminate. The reconstruction algorithm of generic instance has an upper-bound of its recursive calls. If the algorithm is called more times than this upper-bound before it finds a valid instance type, it gives up the reconstruction of the instance by reason that the instance has too many typings to test, and ask the user to add an explicit type constraint to the instance.

Chapter 11: Generic value extensions

In the previous chapters, a user-defined generic value is defined by listing its type cases, once and for all, using a generic binding. This goes against programming modularity: it should be possible to declare type cases in different places, since in many cases, semantics of type methods are independent of each other. For instance, printers for various data types which define a generic printer function together should be placed separately in their corresponding modules for data types. Also, generic values should be extensible, even after the initial definitions. For example, introducing a new data type may induce the addition of new type cases for the data type to already defined generic values. A framework for extensible user-defined generic value must be developed for the above features.

Generally speaking, there are two schemes for extending overloaded values: *open and closed world assumptions*. Under the open world assumption, semantics of overloaded values are late bound: their semantics are open to extensions and never are fixed until all parts of a program are statically prepared. An extension to an overloaded value in the open world affects the semantics of all previous uses of the overloaded value. Under the closed world assumption, extensions create new values: semantics of an extended overloaded value is independent of the original semantics. We propose an extension framework for generic values of the closed world assumption, since the semantics of overloaded values under the open world assumption are dynamic until link time and are against the eagerly evaluated ML semantics. The semantic instability in the open world also enforces link time type checking and compilation of overloaded values, which are not well known in Objective Caml's separate module compilation environment. In the ML semantics, the extension under the closed world assumption is simpler and more natural than the open world extension, since each extension defines a new value and therefore the semantics of generic values are static.

The closed extension should not be just a redefinition of values, because *open recursion* is necessary even in the closed world. In the open recursion, semantics of a recursive overloaded value is open to extensions, so that newly added definitions can also be called recursively in evaluating the original part. We claim that the open recursion is a different notion from the open world assumption, and must also be introduced under the closed world assumption.

We introduce a new *type case inclusion* syntax construct to the generic binding for the closed world extension of generic values. In definition of a new generic value, the type cases of other already defined generic values can be easily included by writing type case inclusions inside the type case declaration list of the new generic value. Recursive calls in the included type cases are redirected to the newly defined generic value, in order to achieve open recursion. Typing for the type case inclusion is added to both layers of type systems so that generic values with type case inclusions can have appropriate type schemes and generic constraints, which contains the constraint information of the included generics. The compilation of a user-defined generic value consists of an array of type method compilations and compilations of included type cases are inserted into this array. For the redirection of recursive calls in the open recursion, type method compilation has a new argument, which takes its recursive call back point and the recursive call identifiers in type methods are bound by this functional abstraction. The run time semantics of a user-defined generic value is given as the fixed point of the set of type method compilation functions.

Chapter 2

Generic core system

2.1 Introduction

To start our quest, we first construct an extension of ML which supports *generic values* — polymorphic values whose semantics are dependent on their evaluation type contexts, types at evaluation. Generic values cannot be achieved without run time types, since evaluation type contexts are dynamic: they may be different from the types statically derived at type checking.

We formalize core GML, our target language which is an extension of the ML language with type-dependent semantics. Additional to normal parametric polymorphic values of ML, GML supports generic polymorphic values whose semantics are type-dependent. We focus in this chapter mainly on the type aware semantics of GML. We do not introduce any user-defined generic value definition here. Generic values are considered to be *generic primitives* or their *derived generic values* only. Semantics of a generic primitive is given explicitly as a function which takes an evaluation context type argument and returns some semantic value. For example the following `plus` generic primitive provides addition functions for various data types:

```
# plus 1 2;;
- : int = 3
# plus 1.2 3.4;;
- : float = 4.6
...
```

Semantics of each data type is not defined in the language level, but given explicitly. *Derived generic values* are polymorphic values whose genericity is derived from internal use of other generic primitives and derived values in definitions. The following `double` is such a derived generic value:

```
let double = fun x → plus x x;;

# double 1;;
- : int = 2
# double 1.2;;
- : float = 2.4
```

We first define our type algebra in Section 2.2. We enlarge our types with a set of *dynamic type variables* in order to indicate type-dependent semantics of values annotated with them. Type schemes of generic values are quantified by one or more dynamic type variables. This distinction between generic and other polymorphic values in types isolates semantic rules and compilation of generics from those for normal values. We therefore easily keep the cleanness and efficiency of the non-generic part of our language as standard ML theory and implementation.

We define the syntax of the language in Section 2.3. It has the same syntactic constructs as standard ML augmented with generic primitives. To define the type-dependent semantics system, a fully type-annotated syntax of GML is introduced. Type substitution is extended to mappings over the fully type-annotated expressions which may modify these annotations.

The evaluation of GML is defined in Section 2.4 as an operational semantic system. Unlike other usual ML semantic formalizations without types, the semantics must be defined over fully type-annotated GML expressions, since semantics of generic values are type-dependent. During evaluation of an instance of a polymorphic identifier, the type instantiation of this identifier is applied to its semantic value. Thanks to this dynamic *type context instantiation* in semantics, generic instances are evaluated under their correct type contexts. The semantics of a generic primitive is explicitly given as a function which takes its evaluation type context as an argument and returns the corresponding semantic value in this context. For example, let $\texttt{ident:}\forall\delta.\delta$ be a generic primitive which returns the identity element of its type context (`1` for integer, the identity function for $\alpha \to \alpha$ and so on). Then its evaluation can be expressed by an explicitly given function from types to values:

$$\texttt{ident : int} \mapsto \texttt{1}$$
$$\texttt{ident : string} \mapsto \texttt{""}$$

A definition of a derived generic value is stored in the evaluation environment as a *generic closure*. A generic closure is abstracted by a set of type variables and is executed with some type context for these abstracted type variables, where an instance of the corresponding derived generic value is evaluated. The following derived generic value definition

```
let f = ident;;
```

stores a generic closure $gclos(E, \_, \{\delta\}, \texttt{ident} : \delta)$ for instance. When an instance of `f` is evaluated under type τ, it executes the code $\texttt{ident:}\delta$ applying a type instantiation $(\delta{\Rightarrow}\tau)$:

$$(\delta{\Rightarrow}\tau)(\texttt{ident} : \delta) = \texttt{ident} : \tau \ \mapsto \ \text{the identity of type } \tau$$

We finally build a type system for GML that prevents run time type error during evaluation in Section 2.5. The GML type system is a variant of the ML type system and most rules are straightforward, but its type judgments must be defined against fully type-annotated expressions, since the semantics suppose fully type annotations, too. The soundness of the type system is proven: once a GML program is validated by the type system, its execution is free from potential run time type errors. We also define a type reconstruction algorithm in Section 2.6 which is complete with respect to the typing rules.

2.2 Types and schemes

2.2.1 Types and their tools

Before defining the syntax of our language, we define the types, since the syntax will incorporate them. Distinction between normal and generic polymorphic values in the type system is essential to our formalization. Special semantic rules and compilation algorithms will be applied only to expressions with dynamic types. On the other hand, normal, non generic expressions are treated as usual, so that their semantic simplicity and compilation efficiency can be maintained.

Two kinds of sets of type variables are available in our type system. The set $\mathbb{R}$ contains *regular type variables* written α, β ..., which are familiar in ML languages. Another set $\mathbb{D}$ contains *dynamic type variables* δ, δ' ... to denote that the polymorphism with these variables is *generic*, non parametric polymorphic. To range on both kinds of type variables we use ν. We also have some basic types *base*, typically `unit` or `int` As in other ML families, we have function types and tuples. Types are written $\tau, \tau_1, \ldots$ and type schemes $\sigma, \sigma_1, \ldots$.

$$
\begin{array}{rcll}
\tau & ::= & base & \text{Basic types} \\
& | & \alpha & \text{Regular type variables} \\
& | & \delta & \text{Dynamic type variables} \\
& | & \tau \to \tau & \text{Functional types} \\
& | & \tau * \tau & \text{Tuple types} \\
\\
\sigma & ::= & \forall \nu \ldots \nu . \tau & \text{Type schemes}
\end{array}
$$

The definition of a type scheme is standard, and both regular and dynamic type variables can be quantified. Instead of listing quantified variables like $\forall \nu_1 \ldots \nu_n . \tau$, we sometimes write them as a set as in $\forall V . \tau$ where $V = \{\nu_1, \ldots, \nu_n\}$. Usually a type scheme $\forall \emptyset . \tau$ is written simply τ.

Free and bound variables

$FV(\tau)$ is the set of *free type variables* of τ and is defined inductively on the structure of the type expressions. Any type variable in a type expression being free, the free type variables of a type coincides with the set of type variables appearing in the type:

$$
\begin{array}{rcl@{\qquad}rcl}
FV(base) & = & \emptyset & & & \\
FV(\alpha) & = & \{\alpha\} & FV(\delta) & = & \{\delta\} \\
FV(\tau \to \tau') & = & FV(\tau) \cup FV(\tau') & FV(\tau * \tau') & = & FV(\tau) \cup FV(\tau')
\end{array}
$$

Because we have two kinds of variables, it will be convenient to introduce some more notations. $RV(\tau)$ and $DV(\tau)$ are the set of regular and dynamic free variables occurring in τ respectively:

$$RV(\tau) = FV(\tau) \cap \mathbb{R} \qquad DV(\tau) = FV(\tau) \cap \mathbb{D}$$

Bound variables of a type scheme σ, $BV(\sigma)$, are the type variables quantified in σ:

$$BV(\forall\nu_1 \ldots \nu_n.\tau) = \{\nu_1, \ldots, \nu_n\}$$

Like RV and DV, we define BRV and BDV, *bound regular variables* and *bound dynamic variables*:

$$BRV(\tau) = BV(\sigma) \cap \mathbb{R} \qquad\qquad BDV(\tau) = BV(\sigma) \cap \mathbb{D}$$

Then the set of *free variables* of a type scheme σ, $FV(\sigma)$, is defined as the rest, the set of non-bound variables.

$$FV(\forall\nu_1 \ldots \nu_n.\tau) = FV(\tau) \setminus BV(\tau)$$

We say a type variable ν is *free* in a type scheme σ if ν is in $FV(\sigma)$. If a type scheme has no free type variables, we call it *closed*. We also naturally extend the notion of RV and DV to the type schemes:

$$RV(\sigma) = FV(\sigma) \cap \mathbb{R} \qquad\qquad DV(\sigma) = FV(\sigma) \cap \mathbb{D}$$

Finite mapping and environments

Before we introduce various type manipulations, we define the notion and notations of *finite mapping* and *environment*.

A mapping from a set P to Q is finite if its *definition* is given as a finite set of *bindings* of pair of elements of P and Q. If for a finite number of elements of P, say $p_1, \ldots, p_n$, a finite mapping F maps p_i to q_i, i.e. $\forall 1 \leq i \leq n,\ F(p_i) = q_i \in Q$, then the definition of F is given as:

$$F \equiv \{p_1 {\Rightarrow} q_1, \ldots, p_n {\Rightarrow} q_n\}$$

We regard a finite mapping and its definition as identical. Of course F cannot contain more than one binding for some p_i. Bindings $\{p_1 {\Rightarrow} q_1, \ldots, p_n {\Rightarrow} q_n\}$ is often simply written as $\{p_i {\Rightarrow} q_i\}$, when the range of i, $1 \leq i \leq n$ is clear from the context.

The *domain* of a finite mapping F, $Dom(F)$ is the (finite) subset of P where $F(p)$ is defined:

$$Dom(F) = \{p_i \in P \mid \exists q_i, (p_i {\Rightarrow} q_i) \in F\}$$

For two finite mappings F_1 and F_2 from P to Q whose domains are disjoint, e.g. $Dom(F_1) \cap Dom(F_2) = \emptyset$, we can define their *direct sum*, $F_1 + F_2$:

$$F_1 + F_2 = F_1 \cup F_2$$

The condition on their domains is required for the well-formedness of the sum.

Overriding bindings of a finite map is also useful. We write $F_1\{F_2\}$ for a finite mapping F_1 overridden by another finite mapping F_2:

$$F_1\{F_2\} = \{(p \Rightarrow F(p)) \mid p \in Dom(F_1) \text{ and } p \notin Dom(F_2)\} + F_2$$

In $F_1\{F_2\}$, the F_2's bindings $(p \Rightarrow q)$ override those of F_1 if $p \in Dom(F_1)$, otherwise just add themselves to F_1. If $F_2 = \{p_i \Rightarrow q_i\}$, we also use the following notation

$$F_1\{p_i \Rightarrow q_i\}$$

for $F_1\{F_2\}$. This notation emphasizes that the bindings of $p_i \Rightarrow q_i$ belong to the whole mapping $F_1\{F_2\}$. For example,

$$F\{p \Rightarrow q\}$$

is a finite mapping with a binding from p to q.

An *environment* is a finite mapping from *identifiers*, (variables in the source language), to some values, for example, type schemes, semantic values and so on. Sometimes we will use different symbols for environment bindings instead of the original '$\Rightarrow$'. For instance, a *typing environment*, a mapping from identifiers to type schemes, is written $E\{\mathtt{x} : \sigma\}$.

2.2.2 Admissible type substitutions

Type substitutions are defined as usual: a *type substitution* (or just a substitution) S is a finite mapping from type variables (either regular or dynamic) to type expressions. A substitution S extends naturally to a homomorphism of type expressions:

$$\begin{array}{rcl} S(base) & = & base \\ S(\nu) & = & \tau \quad \text{if } (\nu \Rightarrow \tau) \in S \\ S(\nu) & = & \nu \quad \text{otherwise} \\ S(\tau_1 \to \tau_2) & = & S(\tau_1) \to S(\tau_2) \\ S(\tau_1 * \tau_2) & = & S(\tau_1) * S(\tau_2) \end{array}$$

We often identify a finite mapping from variables to types, and the homomorphism of type expressions of the same type substitution.

Definition 1 (Compositions of substitutions) *Let S_1 and S_2 be substitutions. The composition of the substitutions S_1 and S_2, $S_1 \circ S_2$ is the substitution whose definition is:*

$$S_1 \circ S_2 = \{(\nu \Rightarrow S_1(\tau)) \mid (\nu \Rightarrow \tau) \in S_2\} \cup \{(\nu \Rightarrow \tau) \in S_1 \mid \nu \notin Dom(S_2)\}$$

Theorem 1 *Let S_1 and S_2 be substitutions. For any type expression τ, $(S_1 \circ S_2)(\tau) = S_1(S_2(\tau))$.*

PROOF: We let

$$S_3 = S_1 \circ S_2 = \{(\nu \Rightarrow S_1(\tau)) \mid (\nu \Rightarrow \tau) \in S_2\} \cup \{(\nu \Rightarrow \tau) \in S_1 \mid \nu \notin Dom(S_2)\}$$

S_3 is a well-formed substitution, because $Dom(\{(\nu \Rightarrow S_1(\tau)) \mid (\nu \Rightarrow \tau) \in S_2) = Dom(S_2)$ and $Dom(\{(\nu \Rightarrow \tau) \in S_1 \mid \nu \notin Dom(S_2)\}) = Dom(S_1) \setminus Dom(S_2)$ are disjoint each other. We prove $S_3(\tau) = S_1(S_2(\tau))$ for any type expression by induction on the structure of types.

Base types $S_3(base) = S_1(S_2(base)) = base$ is clear from the definition of the substitution.

Type variables If $\nu \in Dom(S_2)$, $S_3(\nu) = S_1(S_2(\nu)) = (S_1 \circ S_2)(\nu)$. If $\nu \in Dom(S_1) \setminus Dom(S_2)$, $S_3(\nu) = S_1(\nu)$. $S_1(S_2(\nu)) = S_1(\nu)$ also, because $\nu \notin Dom(S_2)$. Otherwise, $\nu \notin Dom(S_1)$ and $\nu \notin Dom(S_2)$, $S_3(\nu) = S_1(S_2(\nu)) = \nu$.

Functional types and tuples Use the induction hypothesis and the definition of S_3.

□

Lemma 1 **(Substitutions and free variables)** *For any type expression τ and substitution S, we have:*

$$FV(S(\tau)) = \bigcup_{\nu \in FV(\tau)} FV(S(\nu))$$

PROOF: Proven by induction on the structure of τ. □

To prevent mixing regular and dynamic type variables in type substitutions, we need to restrict substitutions to *admissible substitutions*. Admissible substitutions never map dynamic type variables to types with regular type variable. For example, $\{\alpha \Rightarrow \delta',\ \delta \Rightarrow \delta'' \to \delta''\}$ is admissible but $\{\delta \Rightarrow \alpha \to \alpha\}$ is not. More precisely:

Definition 2 (Admissible substitutions)
A substitution S is admissible if

$$\forall \delta \in Dom(S),\ FV(S(\delta)) \subset \mathbb{D}$$

In our system all the substitutions must be admissible, since dynamic type variables are signs of genericity and we do not want to loose them by type substitutions. We usually omit the word "admissible", unless special care must be taken.

Theorem 2 *Compositions of substitutions preserve admissibility i.e. if S_1 and S_2 are admissible substitutions, then $S_1 \circ S_2$ is admissible.*

PROOF: From Theorem 1, $S_1 \circ S_2$ maps a dynamic variable δ to $S_1(S_2(\delta))$. The admissibility of S_1 and S_2 assures that both mappings never introduce regular type variables in $S_1(S_2(\delta))$. □

Definition 3 (Restriction of substitution) *Let S be a substitution and V be a set of type variables. Then the* restriction of S to V, *$S|_V$ is a substitution given by the following definition:*

$$S|_V = \{(\nu \Rightarrow \tau) \in S \mid \nu \in V\}$$

The restriction of a substitution contracts the domain of the substitution to a given set of type variables.

Lemma 2 (Restrictions preserve admissibility) *The restriction of an admissible substitution is admissible: let S be an admissible substitution and V be a set of type variables, then $S|_V$ is admissible.*

PROOF: Clear from the definition of $S|_V$. □

We will often use the property that a substitution S does not affect a set of type variables V, e.g. that V is distinct from both domain and codomain of S. In such case we say that V is *out of the reach of* S. (Or simply, V *is out of* S.):

Definition 4 *A set of type variables V is* out of the reach of *a substitution S, when*

$$Dom(S) \cap V = \emptyset \quad \textit{and} \quad \forall \nu \in Dom(S),\ FV(S(\nu)) \cap V = \emptyset$$

Substitution over type schemes

Substitutions can be extended to type schemes also. Before extending it, we need to define α-*equivalence* between type schemes. Two type schemes are α-equivalent if one can be obtained from the other by:

- renaming its bound variables, or
- adding or removing bound variables which do not occur in the body of the scheme.

The renamings must be admissible: they must not change the regular and dynamic attributes of type variables. The formal definition of this equivalence is given by the following two equations:

$$\forall \nu_1 \ldots \nu_n.\tau \stackrel{\alpha}{=} \forall \nu'_1 \ldots \nu'_n.S(\tau) \text{ where } S = \{\nu_i \Rightarrow \nu'_i\} \text{ and } \nu'_i \notin FV(\tau)$$
$$\forall \nu_1 \ldots \nu_n.\tau \stackrel{\alpha}{=} \forall \nu \nu_1 \ldots \nu_n.\tau \text{ where } \nu \notin FV(\tau)$$

Here, in S, all ν_i's must be distinct each other and not members of $FV(\tau)$. For example,

$$\begin{array}{l}
\forall\alpha_1\delta_2.\,\alpha_1 \to \delta_2 \to \alpha_3 \overset{\alpha}{=} \forall\alpha_4\alpha_5\delta_6.\,\alpha_5 \to \delta_6 \to \alpha_3 \\
\forall\alpha_1\delta_2.\,\alpha_1 \to \delta_2 \to \alpha_3 \overset{\alpha}{\neq} \forall\alpha_4\delta_5\delta_6.\,\delta_5 \to \delta_6 \to \alpha_3 \\
\forall\alpha_1\delta_2.\,\alpha_1 \to \delta_2 \to \alpha_3 \overset{\alpha}{\neq} \forall\alpha_4\alpha_3\delta_6.\,\alpha_3 \to \delta_6 \to \alpha_3
\end{array}$$

The type schemes in the second line are not equivalent because the renaming between α_1 and δ_5 involves the two different kinds of type variables and therefore is not admissible. The third line is also incorrect, because the free occurrence of α_3 in the left hand side is no longer free in the right hand side.

If two type schemes are α-equivalent, we usually identify them and just write $\sigma = \sigma'$, unless subtle handling of bound variables is required. Also, if we write a type scheme $\forall\nu_1 \dots \nu_n.\tau$, we suppose that it must be in one of the minimized form: all the bound variables ν_i occur in the body τ. The tool functions BV, BRV, BDV also follow this assumption:

$$\begin{array}{l}
BV(\forall\nu_1 \dots \nu_n.\tau) = \{\nu_1, \dots, \nu_n\} \cap FV(\tau) \\
BRV(\sigma) = BV(\sigma) \cap \mathbb{R} \\
BDV(\sigma) = BV(\sigma) \cap \mathbb{D}
\end{array}$$

Using α-equivalence, substitutions can be extended to type schemes:

$$S(\forall\nu_1 \dots \nu_n.\tau) = \forall\nu_1 \dots \nu_n.S(\tau) \qquad \text{when } \{\nu_i\} \text{ is out of the reach of } S$$

We rename bound variables and do α-conversions if needed, so that $Dom(S)$ and $S(\tau)$ can contain none of the bound variables, $\nu_1 \dots \nu_n$. For example,

$$\begin{array}{rcl}
\{\alpha_1{\Rightarrow}\alpha_2, \delta_3{\Rightarrow}\delta_4\}\,(\forall\alpha_1\delta_4.\alpha_1 \to \delta_3) & \overset{\alpha}{=} & \{\alpha_1{\Rightarrow}\alpha_2, \delta_3{\Rightarrow}\delta_4\}(\forall\alpha_5\delta_6.\alpha_5 \to \delta_3) \\
& = & \forall\alpha_5\delta_6.\alpha_5 \to \delta_4
\end{array}$$

2.2.3 Instance of type schemes

Instances of type schemes are also defined as usual, but substitution are restricted to admissible ones.

Definition 5 (Instances) *A type expression τ is an* instance *of a type scheme $\sigma = \forall\nu_1 \dots \nu_n.\tau'$ and we write $\tau \le \sigma$, if there is an admissible substitution S such that*

$$Dom(S) = \{\nu_i\} \quad \textit{and} \quad \tau = S(\tau')$$

We also write $\tau \le_S \sigma$, if $\tau \le \sigma$ holds with a substitution S which suffices the above conditions. For example, $\alpha \to \delta \le_S \forall\alpha\beta.\alpha \to \beta$ where $S = \{\alpha{\Rightarrow}\alpha,\ \beta{\Rightarrow}\delta\}$. The substitution S is called the *instantiation* of σ to τ. Note that an instantiation of σ is only valid for σ itself and not for the other α-congruents of σ.

Lemma 3 (α-conversions preserve the instance relationship) *If $\tau \leq \sigma$, then for any σ' such that $\sigma' \stackrel{\alpha}{=} \sigma$, we have $\tau \leq \sigma'$ also.*

Proof: Let $\sigma = \forall \nu_1 \ldots \nu_n.\tau'$, $\sigma' = \forall \nu'_1 \ldots \nu'_m.\tau''$ and S be a renaming substitution $S = \{\nu_i \Rightarrow \nu'_j\}$ such that $S(\tau') = \tau''$. Because S is just a renaming of variables and one-to-one mapping, therefore the inverse of S, $S^{-1} = \{\nu'_j \Rightarrow \nu_i\}$ exists.

Suppose that $\tau \leq_{S'} \sigma$. Then $S'(\tau') = \tau$. From $S^{-1}(\tau'') = \tau'$, $(S' \circ S^{-1})(\tau'') = S'(\tau') = \tau$, this leads to $\tau'' \leq_{(S' \circ S^{-1})} \sigma$. □

Lemma 4 (Substitution lemma of instances) *Substitutions preserves the instance relationship: if $\tau \leq_{S_0} \sigma$, for any admissible substitution S, we have $S(\tau) \leq_{S_1} S(\sigma)$, with $S_1 = (S \circ S_0)|_{BV(\sigma)}$.*

Proof: Let $\sigma = \forall V.\tau'$. Then $S_0(\tau') = \tau$. Let $S' = S_{|Dom(S) \setminus V}$, then $S(\sigma) = \forall V.S'(\tau')$.

We prove $S_1 \circ S' = S \circ S_0$. Let $\nu \notin V$, then $(S \circ S_0)(\nu) = S(\nu)$, on the other hand $(S_1 \circ S')(\nu) = S_1 \circ S(\nu)$, because $Dom(S_0) = Dom(S_1) = \{\nu_i\}$.

Otherwise, if $\nu = \nu_i$, one of the bound variables, from the definition of S_1 and $S(\nu_i) = \nu_i$, we have $(S \circ S_0)(\nu_i) = S_1(\nu_i) = (S_1 \circ S)(\nu_i)$. Therefore,

$$S_1 \circ S' = S \circ S_0, \quad (S_1 \circ S')(\tau') = (S \circ S_0)(\tau') = S(\tau)$$

Because $S(\sigma) = \forall \nu_1 \ldots \nu_n.S'(\tau')$,

$$S(\tau) \leq_{S_1} S(\sigma)$$

□

Using these instances of type schemes, we can define an ordering between type schemes:

Definition 6 (More/less general type schemes) *Let σ and σ' be type schemes. We say σ is more general than σ' (or σ' is less general than σ) and write $\sigma' \leq \sigma$, if and only if $\forall \tau \leq \sigma', \tau \leq \sigma$.*

2.3 The core language and type-annotated syntax

The language GML is the core language which we use in the following chapters. It is a variant of the ML core language, a polymorphic language including the usual call-by-value λ-calculus, extended with some data constructors, tuples, basic constants, let polymorphism, and finally, generic primitives.

Here, generic values are treated as primitives or their derived values. Definitions of generic primitives are omitted, being known by their names and type schemes. For example we have $\mathtt{print}_{\forall \delta.\delta \,\to\, \mathtt{unit}}$ and $\mathtt{plus}_{\forall \delta.\delta \,\to\, \delta \,\to\, \delta}$ as generic primitives in this chapter. In the following `g` ranges over anonymous identifiers for generic primitives. The problem of user-defined generic values is treated in in Chapter 7.

$$
\begin{array}{rcll}
e & ::= & c_\sigma \quad \text{where } FV(\sigma) = \emptyset, BDV(\sigma) = \emptyset & \text{Constants} \\
& | & \mathtt{g}_\sigma \quad \text{where } FV(\sigma) = \emptyset, BDV(\sigma) \neq \emptyset, BRV(\sigma) = \emptyset & \text{Generic primitives} \\
& | & \mathtt{x} & \text{Variables} \\
& | & \mathtt{fun}\ \mathtt{x} \rightarrow e & \lambda \text{ abstractions} \\
& | & e\ e & \text{Applications} \\
& | & e, e & \text{Tuples} \\
& | & \mathtt{let}\ \mathtt{x} = e\ \mathtt{in}\ e & \text{Let bindings} \\
& | & \mathtt{let}\ \mathtt{rec}\ \mathtt{x} = e\ \mathtt{in}\ e & \text{Recursive let bindings}
\end{array}
$$

The syntax contains types: constants and generic primitives are explicitly annotated with their *closed* type schemes. Constants are non-generic: their type schemes do not contain dynamic variables. On the contrary, type schemes of generic primitives are fully generic: they are quantified by non-empty sets of dynamic type variables. We do not consider primitives with type schemes having a mix of regular and dynamic type variables. Other expression constructs, such as functions, applications, tuples, let bindings are regular. For simplicity multiple function abstractions are often abbreviated as follows:

$$\mathtt{fun}\ \mathtt{x}_1\ \ldots\ \mathtt{x}_n \rightarrow e \ \equiv\ \mathtt{fun}\ \mathtt{x}_1 \rightarrow\ \ldots \rightarrow \mathtt{fun}\ \mathtt{x}_n \rightarrow e$$

It is impossible to define the semantics of GML solely on the above syntax, because semantics of generic primitives depend on their typing. Therefore, the semantics must be defined over *fully type-annotated* GML, where all the sub-expressions have type annotations. We sometimes refer the original GML without type annotations as *bare* GML, to distinguish it from this type-annotated syntax. The grammar of fully type-annotated GML expressions e_t is defined as follows:

$$
\begin{array}{rcl}
e_t & ::= & c_\sigma : \tau \\
& | & \mathtt{g}_\sigma : \tau \\
& | & \mathtt{x}_\sigma : \tau \\
& | & \mathtt{fun}\ \mathtt{x} \rightarrow e_t : \tau \\
& | & e_t\ e_t : \tau \\
& | & e_t, e_t : \tau \\
& | & \mathtt{let}\ \mathtt{x}_{\{\nu \ldots \nu\}} = e_t\ \mathtt{in}\ e_t : \tau \\
& | & \mathtt{let}\ \mathtt{rec}\ \mathtt{x}_{\{\nu \ldots \nu\}} = e_t\ \mathtt{in}\ e_t : \tau
\end{array}
$$

Sub-expressions have type annotations after the symbol ‘`:`’. Constants, generic primitives and variables now have two type informations: a type scheme σ attached to a value or primitive itself and a type expression τ which denotes how each occurrence is instantiated. Let bindings have additional type information, *generalized type variable sets* to express which type variables they generalize. For instance, in the following type-annotated expression:

$$\mathtt{let}\ \mathtt{x}_{\{\delta\}} = \mathtt{print}_{\forall \delta.\delta \rightarrow \mathtt{unit}} : \delta \rightarrow \mathtt{unit}\ \mathtt{in}\ e_t : \tau$$

`x` is bound with a polymorphic type scheme $\forall \delta.\delta \rightarrow \mathtt{unit}$ inside the evaluation of e_t. We often write $e_t : \tau$, to emphasize that the out-most type annotation of e_t is τ.

Type substitutions against type annotations

We easily extend FV to the type-annotated expressions. $FV(e_t)$, the free variables of a fully type-annotated expression e_t, is defined as follows:

$$
\begin{array}{rcl}
FV(c_\sigma : \tau) & = & FV(\tau) \\
FV(\mathtt{g}_\sigma : \tau) & = & FV(\tau) \\
FV(\mathtt{x}_\sigma : \tau) & = & FV(\tau) \cup FV(\sigma) \\
FV(\mathtt{fun\ x} \rightarrow e_t : \tau) & = & FV(e_t) \cup FV(\tau) \\
FV(e_{t1}\ e_{t2} : \tau) & = & FV(e_{t1}) \cup FV(e_{t2}) \cup FV(\tau) \\
FV(e_{t1},\ e_{t2} : \tau) & = & FV(e_{t1}) \cup FV(e_{t2}) \cup FV(\tau) \\
FV(\mathtt{let\ (rec)\ x}_{\{\nu_i\}} \mathtt{=} e_{t1}\ \mathtt{in}\ e_{t2} : \tau) & = & (FV(e_{t1}) \setminus \{\nu_i\}) \cup FV(e_{t2})
\end{array}
$$

Type variables of a let definition e_{t1} are bound by the generalized type variable set $\{\nu_i\}$ of the definition, like type variables in a type scheme are bound by its $\forall$-quantified variables.

We may consider *α-conversions of fully type-annotated expressions*. The definition of α-equivalence $\stackrel{\alpha}{=}$ can be defined as a set of equations obtained by renaming variables of generalized type variable sets of `let` expressions, similarly to the definition of $\stackrel{\alpha}{=}$ for type schemes:

$$
\begin{array}{l}
\mathtt{let\ x}_{\{\nu_1,\ldots,\nu_n\}} \mathtt{=} e_{t1}\ \mathtt{in}\ e_{t2} : \tau \ \stackrel{\alpha}{=}\ \mathtt{let\ x}_{\{\nu,\nu_1,\ldots,\nu_n\}} \mathtt{=} e_{t1}\ \mathtt{in}\ e_{t2} : \tau \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{when } \nu \notin FV(e_{t1}) \\
\mathtt{let\ x}_{\{\nu_1,\ldots,\nu_n\}} \mathtt{=} e_{t1}\ \mathtt{in}\ e_{t2} : \tau \ \stackrel{\alpha}{=}\ \mathtt{let\ x}_{\{\nu_1,\ldots,\nu_i',\ldots,\nu_n\}} \mathtt{=} e_{t1}'\ \mathtt{in}\ e_{t2} : \tau \\
\quad \text{where } \ \{\nu_i \Rightarrow \nu_i'\} \text{ is admissible } \ \nu_i' \notin FV(e_{t1}) \\
\qquad\quad e_{t1}' \text{ is obtained by replacing the free occurrences of } \nu_i \text{ of } e_{t1} \text{ to } \nu_i'.
\end{array}
$$

As for type schemes, we usually consider α-equivalent fully type-annotated expressions as identical.

Example

Suppose that we have the following fully type-annotated expression e_t:

$$
\begin{array}{l}
\mathtt{(let\ f}_{\{\alpha\}}\ \mathtt{=} \\
\quad \mathtt{fun\ x} \rightarrow \\
\qquad \mathtt{let\ h}_{\{\alpha\}}\ \mathtt{=} \\
\qquad\quad \mathtt{fun\ y} \rightarrow \mathtt{(y}_\alpha\mathtt{:}\alpha\mathtt{)}\ \mathtt{:}\ \alpha \rightarrow \alpha \\
\qquad \mathtt{in} \\
\qquad \mathtt{(h}_{\forall\alpha.\alpha \rightarrow \alpha}\mathtt{:}\alpha \rightarrow \alpha\mathtt{)}\ \mathtt{(x}_\alpha\mathtt{:}\alpha\mathtt{)}\ \mathtt{:}\ \alpha \\
\ \mathtt{in} \\
\ \mathtt{1:int)\ :\ int}
\end{array}
$$

Renaming the generalized variable α occurring in the definition of `f` to β, we obtain an α-equivalent fully type-annotated expression e_t':

$$
\begin{array}{l}
\texttt{(let f}_{\{\beta\}}\ \texttt{=} \\
\quad \texttt{fun x} \rightarrow \\
\qquad \texttt{let h}_{\{\alpha\}}\ \texttt{=} \\
\qquad\quad \texttt{fun y} \rightarrow \texttt{(y}_\alpha\texttt{:}\alpha\texttt{)}\ :\ \alpha \rightarrow \alpha \\
\qquad \texttt{in} \\
\qquad \texttt{(h}_{\forall\alpha.\alpha\ \rightarrow\ \alpha}\texttt{:}\beta \rightarrow \beta\texttt{)}\ \texttt{(x}_\beta\texttt{:}\beta\texttt{)}\ :\ \beta \\
\ \texttt{in} \\
\ \texttt{1:int)}\ :\ \texttt{int}
\end{array}
$$

The α's inside the definition of `h` are not renamed since they are bound by the generalized type variable occurring in `h`.

□

We write $S(e_t)$ for a fully type-annotated expression e_t in which each type annotation τ or σ is replaced by $S(\tau)$ or $S(\sigma)$ respectively. The precise definition of $S(e_t)$ is quite straightforward except the case of a `let` binding:

$$
\begin{array}{rcl}
S(c_\sigma : \tau) & = & c_\sigma : S(\tau) \\
S(\mathtt{g}_\sigma : \tau) & = & \mathtt{g}_\sigma : S(\tau) \\
S(\mathtt{x}_\sigma : \tau) & = & \mathtt{x}_{S(\sigma)} : S(\tau) \\
S(\mathtt{fun\ x} \rightarrow e_t : \tau_2) & = & \mathtt{fun\ x} \rightarrow S(e_t) : S(\tau_2) \\
S(e_{t1}\ e_{t2} : \tau) & = & S(e_{t1})\ S(e_{t2}) : S(\tau) \\
S(e_{t1}, e_{t2} : \tau) & = & S(e_{t1}), S(e_{t2}) : S(\tau)
\end{array}
$$

For the `let` case, as we did for the substitutions of type schemes, we do an implicit α-conversion of the expression, so that the generalized variable set $\{\nu_i\}$ becomes out of the reach of S and is not affected by S:

$$
\begin{array}{l}
S(\mathtt{let\ (rec)\ x}_{\{\nu_i\}} = e_{t1}\ \mathtt{in}\ e_{t2} : \tau) = \\
\qquad \mathtt{let\ (rec)\ x}_{\{\nu_i\}} = S(e_{t1})\ \mathtt{in}\ S(e_{t2}) : S(\tau) \qquad \text{Rename generalized variables if necessary}
\end{array}
$$

As an auxiliary, we define here $\mathit{strip}(e_t)$, a function which removes all the type annotations in e_t to recover a bare GML expression:

$$
\begin{array}{rcl}
\mathit{strip}(c_\sigma : \tau) & = & c_\sigma \\
\mathit{strip}(g_\sigma : \tau) & = & g_\sigma \\
\mathit{strip}(\mathtt{x}_\sigma : \tau) & = & \mathtt{x} \\
\mathit{strip}(\mathtt{fun\ x} \rightarrow e_t : \tau_2) & = & \mathtt{fun\ x} \rightarrow \mathit{strip}(e_t) \\
\mathit{strip}(e_{t1}\ e_{t2} : \tau) & = & \mathit{strip}(e_{t1})\ \mathit{strip}(e_{t2}) \\
\mathit{strip}(e_{t1}, e_{t2}) & = & \mathit{strip}(e_{t1}), \mathit{strip}(e_{t2})
\end{array}
$$

$$
\begin{array}{l}
\mathit{strip}(\mathtt{let\ (rec)\ x}_V = e_{t1}\ \mathtt{in}\ e_{t2} : \tau) = \\
\qquad\qquad \mathtt{let\ (rec)\ x} = \mathit{strip}(e_{t1})\ \mathtt{in}\ \mathit{strip}(e_{t2})
\end{array}
$$

2.4 Semantics

We present the semantics of fully type-annotated GML as a structured operational semantics system. ML semantics is usually formalized without types. However, GML's semantics system does need types, since generic values are type-dependent. Derived generic values are evaluated to *generic closures*: closures for generic values. A generic closure is a closure abstracted by type variables, and its code is evaluated when an *evaluation type contexts* (or just type contexts) for the corresponding generic value is given as a type instantiation of the abstracted variables.

2.4.1 Semantic values

We define the semantic domains for the GML semantics system:

$$
\begin{array}{lcll}
r & ::= & v & \text{Regular values} \\
 & | & err & \text{Errors} \\
v & ::= & b_\sigma & \text{Basic constants} \\
 & | & v, v & \text{Tuples} \\
 & | & clos(E, \_, \mathtt{x}, e_t) & \text{Functional closures} \\
 & | & clos(E, \mathtt{f}, \mathtt{x}, e_t) & \text{Functional closures (recursive)} \\
sv & ::= & v & \text{Regular values} \\
 & | & gclos(E, \_, V, e_t) & \text{Generic closures} \\
 & | & gclos(E, \mathtt{x}, V, e_t) & \text{Generic closures (recursive)} \\
V & \equiv & \{\nu_1, \ldots, \nu_n\} & \text{Generalized type variables} \\
E & ::= & \{\mathtt{x} = sv, \ \ldots \ \mathtt{x} = sv\} & \text{Environment}
\end{array}
$$

An evaluation of an expression returns a *result*. A result r may be a *semantic value* or an *error*. A semantic value is a successful result of an evaluation. Actually, only a subset of semantic values called *regular values* v are returned as results. The other semantic values, i.e. generic closures are never returned as results.

Regular values v consist in basic constants, tuples and functional closures. Constants are type scheme annotated and they must be closed. A *closure* is a pair of an environment and a piece of code which is abstracted by variables. It is evaluated when semantic values for the abstracted variables are supplied: the evaluation of its code is performed under its environment and the bindings of abstracted variables to the supplied values. A function `fun x` $\rightarrow e_t$ is evaluated to a *functional closure* $clos(E, \_, \mathtt{x}, e_t)$, and its body e_t is executed when it is applied to the argument `x`. In the case that the closure is recursively defined by a `let rec` expression, the defined identifier `f` is attached to the closure like $clos(E, \mathtt{f}, \mathtt{x}, e_t)$, so that the occurrences of `f` inside the body e_t can be evaluated to this closure itself. In the non recursive case, we write '_' instead of an identifier. We often use a meta variable symbol $*$ for a recursive identifier `f` or _.

An *evaluation environment* E is an environment that maps identifiers to *semantic values sv*. *Semantic values sv* are regular values v or *generic closures* $gclos(E, *, V, e_t)$, another kind of closures. Generic closures only appear inside evaluation environments. A generic closure $gclos(E, *, V, e_t)$ represents a derived generic value whose definition is e_t. The generic closure is abstracted by a set of *generalized type variables* V of the derived generic value, and its definition e_t is evaluated when an *evaluation type context* is available as an type instantiation of V. A variable `x` is written in a generic closure as $gclos(E, \mathtt{x}, V, e_t)$, if the value is defined recursively and bound to `x`, as in recursive functional closures. Same as type schemes and `let` type-annotated expressions, the type variables in V bind their occurrences in the type annotations of e_t. Therefore α-equivalence can be defined between generic closures:

$$
\begin{array}{rcl}
gclos(E, *, \{\nu_1, \ldots, \nu_n\}, e_t) & \stackrel{\alpha}{=} & gclos(E, *, \{\nu, \nu_1, \ldots, \nu_n\}, e_t) \\
& & \text{when } \nu \notin FV(e_t) \\
gclos(E, *, \{\nu_1, \ldots, \nu_n\}, e_t) & \stackrel{\alpha}{=} & gclos(E, *, \{\nu'_1, \ldots, \nu'_n\}, \{\nu_i \Rightarrow \nu'_i\}\, e_t) \\
& & \text{when } \{\nu_i \Rightarrow \nu'_i\} \text{ is admissible and } \nu'_i \notin FV(e_t)
\end{array}
$$

Since closures contain types, type substitutions are extended for semantic values and evaluation environments:

$$
\begin{array}{rcl}
S(b_\sigma) & = & b_\sigma \\
S(v_1, v_2) & = & S(v_1), S(v_2) \\
S(clos(E, *, \mathtt{x}, e_t)) & = & clos(S(E), *, \mathtt{x}, S(e_t)) \\
S(gclos(E, *, V, e_t)) & = & gclos(S(E), *, V, S(e_t)) \\
S(E) & = & \{(\mathtt{x} = S(sv)) \mid (\mathtt{x} = sv) \in E\}
\end{array}
$$

When we apply a substitution S to a generic closure $gclos(E, *, V, e_t)$, we may have to rename generalized type variables so that S does not affect bindings of V in e_t.

There is no special semantic value for generic primitives. Their evaluations are supposed to be always successful and their results are given explicitly depending on their evaluation type contexts.

Semantic values for primitives

For simplicity of the later arguments of compilation of GML, we assume that semantic values for constants and primitives must be restricted within the following set of *semantic values for primitives* P:

$$b_\sigma \in P$$

$$\frac{v_1 \in P \quad v_2 \in P}{v_1, v_2 \in P}$$

$$\frac{e_t \text{ has no generic primitives inside.}}{clos(\emptyset, *, \mathtt{x}, e_t) \in P}$$

The closures in P are restricted to those with the empty evaluation environment whose codes e_t contain no generic primitives.

2.4.2 evaluation rules

Our evaluation system is defined for fully type-annotated GML expressions. We use *evaluation judgments* in the form of $E \models e_t : \tau \Longrightarrow r$. The out-most type annotation τ of e_t is called the *evaluation type context* for e_t. The judgment reads that "in the evaluation environment E and under the evaluation type context τ, a fully type-annotated expression e_t is evaluated to r." When an error occurs, the semantics system gives up all the evaluation procedure and returns *err* immediately. In the following, we introduce and comment the evaluation rules one by one.

Constants and primitives

$$\textsf{(e-cnst)}\quad \frac{\tau \leq_S \sigma}{E \models c_\sigma : \tau \Longrightarrow S(\mathit{cnst}(c_\sigma))}$$

Semantics of constants are explicitly given by the function *cnst*. The returned semantic values are restricted within the set of semantic values for primitives: $\mathit{cnst}(c_\sigma) \in P$. If c_σ is a basic constant b_σ, for example an integer, then $\mathit{cnst}(c_\sigma)$ returns itself. If c_σ is a primitive non-generic function, then $\mathit{cnst}(c_\sigma)$ will return a closure $\mathit{clos}(\emptyset, *, \mathtt{x}, e_t)$, where e_t has no generic primitives occurrence. We apply the instantiation S where $\tau \leq_S \sigma$ to insure that the returned value is semantically well-typed.

$$\textsf{(e-gen)}\quad E \models \mathsf{g}_\sigma : \tau \Longrightarrow \mathit{gen}(\mathtt{g}, \tau)$$

Semantics of generic primitives are given by *gen*, the *generic semantics query function.* $\mathit{gen}(\mathtt{g}, \tau)$ returns the semantic value of generic primitive $\mathtt{g}$ in the type context τ. Like *cnst*, $\mathit{gen}(\mathtt{g}, \tau)$ must return a primitive semantic value: $\mathit{gen}(\mathtt{g}, \tau) \in P$. For example, for the generic printing function $\mathtt{print}_{\forall\delta.\delta \to \mathtt{unit}}$ evaluated under the evaluation type context $\mathtt{int} \to \mathtt{unit}$, $\mathit{gen}(\mathtt{print}, \mathtt{int} \to \mathtt{unit})$ returns a functional closure which receives an integer and prints it out. We suppose that the generic primitives are *total*: *gen* must return some semantic value different from *err* for any type context τ.

Tuples

$$\textsf{(e-tpl)}\quad \frac{E \models e_{t1} \Longrightarrow v_1 \quad E \models e_{t2} \Longrightarrow v_2}{E \models (e_{t1}, e_{t2}) \Longrightarrow v_1, v_2}$$

$$\textsf{(e-tpl-err)}\quad \frac{E \models e_{ti} \Longrightarrow \mathit{err}}{E \models (e_{t1}, e_{t2}) \Longrightarrow \mathit{err}}$$

The semantic value of a tuple is the tuple of semantic values of its contents. However, if one of the sub-expression is evaluated to *err*, then the whole evaluation also fails immediately.

Variables

The rules for variables are mainly separated into two cases, depending on the kind of environmental semantic values that bound to them. The first is for regular values, and the second is for generic closures.

$$\textbf{(e-nvar)}\quad \frac{\tau \leq_S \sigma}{E\{\mathtt{x} = v\} \models \mathtt{x}_\sigma : \tau \Longrightarrow S(v)}$$

If x is bound by a regular value v in the environment, we return v as its semantic value applied the instantiation S, where $\tau \leq_S \sigma$.

$$\textbf{(e-gvar)}\quad \frac{\tau \leq_S \forall V.\tau' \quad E' \models S(e_t : \tau') \Longrightarrow r}{E\{\mathtt{x} = gclos(E', \_, V, e_t : \tau')\} \models \mathtt{x}_{\forall V.\tau'} : \tau \Longrightarrow r}$$

$$\textbf{(e-gvar-r)}\quad \frac{\begin{array}{c}\exists S,\ \tau \leq_S \forall V.\tau' \\ E'\{\mathtt{f} = gclos(E', \mathtt{f}, V, e_t : \tau')\} \models S(e_t : \tau') \Longrightarrow r\end{array}}{E\{\mathtt{x} = gclos(E', \mathtt{f}, V, e_t : \tau')\} \models \mathtt{x}_{\forall V.\tau'} : \tau \Longrightarrow r}$$

$$\textbf{(e-gvar-err)}\quad \frac{\begin{array}{l}S \text{ such that } \tau \leq_S \forall V.\tau' \text{ does not exist,} \\ \text{or } \forall V.\tau' \overset{\alpha}{\not\equiv} \sigma\end{array}}{E\{\mathtt{x} = gclos(E', *, V, e_t : \tau')\} \models \mathtt{x}_\sigma : \tau \Longrightarrow err}$$

If x is bound by a generic closure, we execute the expression in the generic closure, since we have now the evaluation type context τ for it. We require τ to be an instance of the type scheme of the closure $\forall V.\tau'$. It is also necessary that the type scheme annotation σ of the variable x is α-equivalent to $\forall V.\tau'$. The instantiation S indicates how generalized variables V are actually instantiated at evaluation of the stored code e_t. We evaluate $S(e_t)$ under E' so that the generic values inside e_t can also be evaluated in their correct evaluation type context. We return the evaluation result of $S(e_t)$ as one for x. We do *not* apply the instantiation S to the environment E' in the generic closure, because we can assume that S has no relation to E' by renaming the bound variables V.

$$\textbf{(e-var-err)}\quad \frac{\mathtt{x} \notin Dom(E)}{E \models \mathtt{x}_\sigma : \tau \Longrightarrow err}$$

Absence of a binding for variable x in the evaluation environment results in an error.

Functions

$$\textbf{(e-fun)}\quad E \models \mathtt{fun\ x} \rightarrow e_t \Longrightarrow clos(E, \_, \mathtt{x}, e_t)$$

A function is evaluated to its closure, which consists of the current evaluation environment, the abstracted variable x, and the body expression of the function. The body expression is evaluated when a semantic value of its argument is given to it.

Let bindings

Evaluations of let bindings are separated into two groups, depending on whether they define derived generic values. We can distinguish them by checking the generalized type variable set attached to a let identifier contains dynamic type variables or not. In each part we have two rules, for non-recursive and recursive bindings.

First we look at non-generic let bindings. They store a binding of a variable to a semantic value.

$$\textsf{(e-let)}\ \frac{V \cap \mathbb{D} = \emptyset \quad E \models e_{t1} \Longrightarrow v \quad E\{\mathtt{x} = v\} \models e_{t2} \Longrightarrow r}{E \models \mathtt{let}\ \mathtt{x}_V \,\mathtt{=}\, e_{t1}\ \mathtt{in}\ e_{t2} \Longrightarrow r}$$

$$\textsf{(e-let-err)}\ \frac{V \cap \mathbb{D} = \emptyset \quad E \models e_{t1} \Longrightarrow \mathit{err}}{E \models \mathtt{let}\ \mathtt{x}_V \,\mathtt{=}\, e_{t1}\ \mathtt{in}\ e_{t2} \Longrightarrow \mathit{err}}$$

In a non-recursive `let`, first we evaluate the definition e_{t1}, then bound the variable `x` with its result in the environment, then evaluate the body e_{t2} under the updated environment. The result of the whole expression is equal to the result of e_{t2}. If the evaluation of e_{t1} fails, the whole expression also fails.

$$\textsf{(e-letrec)}\ \frac{\begin{array}{c} V \cap \mathbb{D} = \emptyset \\ E\{\mathtt{f} = \mathit{clos}(E, \mathtt{f}, \mathtt{x}, e_{t1})\} \models e_{t2} \Longrightarrow r \end{array}}{E \models \mathtt{let\ rec}\ \mathtt{f}_V \,\mathtt{=}\, \mathtt{fun}\ \mathtt{x} \rightarrow e_{t1}\ \mathtt{in}\ e_{t2} \Longrightarrow r}$$

$$\textsf{(e-letrec-err)}\ \frac{V \cap \mathbb{D} = \emptyset \quad \text{if } e_{t1} \text{ is non-function}}{E \models \mathtt{let\ rec}\ \mathtt{f}_V \,\mathtt{=}\, e_{t1}\ \mathtt{in}\ e_{t2} \Longrightarrow \mathit{err}}$$

Recursive `let` expressions are handled differently. We require that its definition must have the form of a function. The definition is converted immediately to a recursive closure $\mathit{clos}(E, \mathtt{f}, \mathtt{x}, e_{t1})$, and the body of the function e_{t1} will be evaluated when the recursive function `f` is applied later. We add a binding of `x` with the recursive closure to the environment and evaluate the body e_{t2} in this updated environment. The bindings of non-functional recursive values are not permitted, otherwise evaluation of definitions may fall into infinite loops.

Now we move on to the generic let expressions:

$$\textsf{(e-g-let)}\ \frac{V \cap \mathbb{D} \neq \emptyset \quad E\{\mathtt{x} = \mathit{gclos}(E, \_, V, e_{t1})\} \models e_{t2} \Longrightarrow r}{E \models \mathtt{let}\ \mathtt{x}_V \,\mathtt{=}\, e_{t1}\ \mathtt{in}\ e_{t2} \Longrightarrow r}$$

$$\textsf{(e-g-letrec)}\ \frac{V \cap \mathbb{D} \neq \emptyset \quad E\{\mathtt{x} = \mathit{gclos}(E, \mathtt{x}, V, e_{t1})\} \models e_{t2} \Longrightarrow r}{E \models \mathtt{let\ rec}\ \mathtt{x}_V \,\mathtt{=}\, e_{t1}\ \mathtt{in}\ e_{t2} \Longrightarrow r}$$

`let` expressions which bind generic values are differently handled from non-generic let. Their definitions are stored in evaluation environments as generic closures and are evaluated when they receive their evaluation type context by (e-gvar) and (e-gvar-r) rules. Unlike the non-generic recursive `let`, the generic recursive `let` has no such restriction that the definition must be a function. This is because its definition is always evaluated to a generic closure even if the definition is not a function. Therefore, there is no danger of infinite evaluation loop of the definition.

As we saw, the generic closures appear just in evaluation environments. They are never returned as a result of any evaluation.

Applications

$$\textsf{(e-app1)}\quad \frac{E \models e_{t2} \Longrightarrow v_2 \quad E \models e_{t1} \Longrightarrow clos(E', \_, \mathtt{x}, e_t) \qquad E'\{\mathtt{x} = v_2\} \models e_t \Longrightarrow r}{E \models e_{t1}\ e_{t2} \Longrightarrow r}$$

$$\textsf{(e-app2)}\quad \frac{E \models e_{t2} \Longrightarrow v_2 \quad E \models e_{t1} \Longrightarrow clos(E', \mathtt{f}, \mathtt{x}, e_t) \qquad E'\{\mathtt{f} = clos(E', \mathtt{f}, \mathtt{x}, e_t),\ \mathtt{x} = v_2\} \models e_t \Longrightarrow r}{E \models e_{t1}\ e_{t2} \Longrightarrow r}$$

$$\textsf{(e-app-err1)}\quad \frac{E \models e_{t2} \Longrightarrow err}{E \models e_{t1}\ e_{t2} \Longrightarrow err}$$

$$\textsf{(e-app-err2)}\quad \frac{E \models e_{t2} \Longrightarrow v_2 \quad E \models e_{t1} \Longrightarrow r \quad \text{where } r \neq clos(\ldots)}{E \models e_{t1}\ e_{t2} \Longrightarrow err}$$

In applications, we evaluate an argument e_{t2}, then a function e_{t1}[1]. The evaluation of e_{t1} must result in a closure. We add a binding of `x` to the result of e_{t2} into the environment E' in the closure, and execute the closure code e_t in this new environment. For recursive closures, we also bind the closure itself in the new environment so that the closure can be recursively available in the evaluation of the functional body e_t. If an error occurs in the evaluation of the argument e_{t2} or the function e_{t1} is not evaluated to a closure, the application returns an error.

Example

Let $\mathtt{g}_{\forall\delta.\delta\ \rightarrow\ \mathtt{unit}}$ be a generic primitive, then the following type-annotated GML expression

$$\begin{array}{l}
\mathtt{let\ f}_{\{\delta'\}}\ \mathtt{= (fun\ x} \rightarrow \mathtt{(g}_{\forall\delta.\delta\ \rightarrow\ \mathtt{unit}} : \delta' * \delta' \rightarrow \mathtt{unit)\ (x,x)} : \delta' \rightarrow \mathtt{unit)} \\
\ \mathtt{in} \\
\quad \mathtt{((f : int} \rightarrow \mathtt{unit)\ (1_{int} : int) : unit)}
\end{array}$$

[1] This evaluation order is compatible with the Objective Caml implementation

is evaluated as follows (evaluation environments and type annotations are omitted if they are clear):

$$\frac{\{\mathtt{f} = gclos(\emptyset, \_, \{\delta\}, \mathtt{fun\ x} \rightarrow \mathtt{g\ (x, x)} : \delta \rightarrow \mathtt{unit})\} \models \mathtt{(f : int} \rightarrow \mathtt{unit)\ 1} \Longrightarrow r}{\emptyset \models \mathtt{let\ f = fun\ x} \rightarrow \mathtt{g\ (x, x)\ in\ f\ 1} \Longrightarrow r}$$

First we store the definition of `f` inside the environment and evaluate `f 1`. The result r of the original expression is as same as the result of `f 1`.

$$\{\mathtt{f} = \ldots\} \models \mathtt{1} \Longrightarrow \mathtt{1}$$

The argument `1` is evaluated to itself by (e-cnst).

The generic instance `f` has the type annotation $\mathtt{int} \rightarrow \mathtt{unit}$, which is an instance of the type scheme of `f`, $\mathtt{int} \rightarrow \mathtt{unit} \leq_S \forall\delta.\delta \rightarrow \mathtt{unit}$ where the instantiation $S = \{\delta \Rightarrow \mathtt{int}\}$:

$$\frac{\begin{array}{l}\emptyset \models S(\mathtt{fun\ x} \rightarrow \mathtt{g\ (x, x)} : \delta \rightarrow \mathtt{unit}) \\ \qquad\qquad \Longrightarrow clos(\emptyset, \_, \mathtt{x}, (\mathtt{g} : \mathtt{int} * \mathtt{int} \rightarrow \mathtt{unit})\ \mathtt{(x, x)})\end{array}}{\begin{array}{l}\{\mathtt{f} = gclos(\emptyset, \_, \{\delta\}, \mathtt{fun\ x} \rightarrow \mathtt{g\ (x, x)} : \delta \rightarrow \mathtt{unit})\} \models \mathtt{f} \\ \qquad\qquad \Longrightarrow clos(\emptyset, \_, \mathtt{x}, (\mathtt{g} : \mathtt{int} * \mathtt{int} \rightarrow \mathtt{unit})\ \mathtt{(x, x)})\end{array}}$$

The evaluation of `f` by (e-gvar) is equal to the result of the code $S(\mathtt{fun\ x} \rightarrow \mathtt{g\ (x, x)} : \delta \rightarrow \mathtt{unit})$, the definition expression of `f` whose annotations are now instantiated by S. It returns a closure $clos(\emptyset, \_, \mathtt{x}, (\mathtt{g} : \mathtt{int} * \mathtt{int} \rightarrow \mathtt{unit})\ \mathtt{(x, x)})$ where `g`'s evaluation type context is instantiated to $\mathtt{int\ *\ int} \rightarrow \mathtt{unit}$.

To evaluate `f 1`, the last thing we need is to store the semantic value for the argument `1` in the environment and execute the code in the closure $(\mathtt{g} : \mathtt{int} * \mathtt{int} \rightarrow \mathtt{unit})\ \mathtt{(x, x)}$ under the updated environment by (e-app):

$$\{\mathtt{x} = \mathtt{1}\} \models (\mathtt{g} : \mathtt{int} * \mathtt{int} \rightarrow \mathtt{unit})\ \mathtt{(x, x)} \Longrightarrow r$$

Here the argument `(x,x)` is evaluated to:

$$\{\mathtt{x} = \mathtt{1}\} \models \mathtt{(x, x)} \Longrightarrow \mathtt{(1, 1)}$$

And the generic instance `g` is evaluated by (e-gen):

$$\{\mathtt{x} = \mathtt{1}\} \models (\mathtt{g} : \mathtt{int} * \mathtt{int} \rightarrow \mathtt{unit}) \Longrightarrow gen(\mathtt{g}, \mathtt{int} * \mathtt{int} \rightarrow \mathtt{unit}) = clos(\emptyset, \_, \mathtt{x}, e_t)$$

Here we assume that the semantic function gen returns a proper semantic value for `g` under $\mathtt{int} * \mathtt{int} \rightarrow \mathtt{unit}$, some closure $clos(\emptyset, \_, \mathtt{x}, e_t)$ which receives a tuple of integers and returns a unit `()`:

$$\{\mathtt{x} = \mathtt{(1, 1)}\} \models e_t \Longrightarrow \mathtt{()}$$

This means the above application $(\mathtt{g} : \mathtt{int} * \mathtt{int} \rightarrow \mathtt{unit})\ \mathtt{(x, x)}$ is evaluated to `()`, and therefore $r = \mathtt{()}$. □

2.5 Type system

Like usual untyped ML language, if GML programs are not verified by some typing system, they may raise run time type errors at evaluation. Such errors are hard to fix without help of static typing. We must type-check fully type-annotated GML expressions and restrict them to the *well-typed* ones so that well-typed programs can never fall into evaluation errors. Our typing system is basically as same as many systems whose origin is Damas-Milner's[DM82]. Main difference is that in ours type checking is carried out against expressions already fully type-annotated.

2.5.1 Typing environments

A *typing environment* Γ is an environment from identifiers to type schemes. We need to extend our tools for types also for typing environments. $FV(\Gamma)$ is the set of variables freely occur inside Γ's bindings:

$$FV(\Gamma) = \bigcup_{(\mathtt{x}:\sigma)\in\Gamma} FV(\sigma)$$

$S(\Gamma)$ is a typing environment obtained by applying S to the type schemes of the bindings of Γ:

$$S(\Gamma) = \{(\mathtt{x} : S(\sigma)) \mid (\mathtt{x} : \sigma) \in \Gamma\}$$

We say Γ is more general than Γ' and write $\Gamma' \leq \Gamma$, if for the all bindings of Γ', Γ provides more general type schemes than Γ':

$$\Gamma' \leq \Gamma \iff \forall(\mathtt{x} : \sigma) \in \Gamma',\ \sigma \leq \Gamma(\mathtt{x})$$

2.5.2 Generalization

In our type system, we use type generalization with the value polymorphism restriction [Wri95]. This is not just because this restriction is used widely in real implementations of ML to solve the typing problems with reference cells, but also because it plays important role in the evaluation of extensional polymorphism. Consider the following program using a generic polymorphic primitive `input` whose type scheme is $\forall\delta.\mathtt{unit} \rightarrow \delta$.

```
let x = input () in x + x;;
```

This program should be regarded that it reads an integer and doubles the input integer. Without the value polymorphism restriction, however, the following type-annotated expression is regarded as well-typed:

$$\mathtt{let\ x}_{\{\delta\}}\ \mathtt{=\ (input:unit} \rightarrow \delta\mathtt{)\ ()\ in\ (x:int)\ +\ (x:int);;}$$

Because `x` would be defined as a derived polymorphic value whose type scheme is $\forall\delta.\delta$, it would be evaluated and stored in the evaluation environment as a generic closure

$$gclos(\emptyset, \_, \{\delta\}, (\texttt{input} : \texttt{unit} \to \delta)\ \texttt{()})$$

by (e-g-let), and not read a data at the moment of the definition of `x`. The actual evaluation of `(input:unit` $\to \delta$`) ()` would be done in the execution of (e-gvar) rule twice against both uses of `x`:

$$\frac{\emptyset \models \{\delta \Rightarrow \texttt{int}\}(\ (\texttt{input:unit} \to \delta)\ \texttt{()}\ ) \Longrightarrow v}{\{\texttt{x} = gclos(\ldots,\ (\texttt{input:unit} \to \delta)\ \texttt{()}\ )\} \models \texttt{x} : \texttt{int} \Longrightarrow v}$$

This means that for each `(x:int)` would call `input` once, therefore the whole expression would read two integers: this is not what we want.

This undesirable evaluation can be avoided if we introduce the value only polymorphism restriction. Generalization of let bindings are restricted to "*non expansive*" expressions. If the definition is expansive, its type is not generalized. The following predicate $generalizable(e_t)$ defines non expansive expressions:

$$\begin{array}{rcl}
generalizable(c_\sigma) & = & true \\
generalizable(\mathtt{g}_\sigma) & = & true \\
generalizable(\mathtt{x}) & = & true \\
generalizable(\texttt{fun x} \to e) & = & true \\
generalizable(e_1, e_2) & = & generalizable(e_1) \\
 & & \ \&\ generalizable(e_2) \\
generalizable(\texttt{let (rec) x} = e_1\ \texttt{in}\ e_2) & = & generalizable(e_1) \\
 & & \ \&\ generalizable(e_2) \\
\text{otherwise } generalizable(e) & = & false
\end{array}$$

Using *generalizable*, a generalization function *Gen* is defined. If e is generalizable (non-expansive), we permit the generalization of its type: we abstract the type variables which do not appear in the typing environment Γ freely.

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

We can extend this *Gen* for fully type-annotated expressions. The generalized type τ is the out-most type annotation of e_t:

$$Gen(\Gamma, e_t : \tau) = Gen(\Gamma, strip(e_t), \tau)$$

With the value polymorphism restriction, now the previous example has a typing like:

```
let x∅ = (input:unit → int) () in (x:int) + (x:int);;
```

and the evaluation of let definition is immediate using (e-let) rule. The evaluation of the definition expression will be the following:

$$\frac{\emptyset \models \texttt{()} \Longrightarrow \texttt{()} \quad \emptyset \models \texttt{input} : \texttt{unit} \rightarrow \texttt{int} \Longrightarrow clos(\emptyset, \_, \texttt{x}, e_t) \quad \{\texttt{x} = \texttt{()}\} \models e_t \Longrightarrow r}{\emptyset \models \texttt{(input} : \texttt{unit} \rightarrow \texttt{int) ()} \Longrightarrow r}$$

By (e-gen), the instance of `input` is evaluated to some function which takes `()` and returns an integer. In the evaluation of the whole expression, integer reading is performed just once and it returns the double of the input integer.

Before stepping to the typing rules, we prove a small lemma of the type generalization:

Lemma 5 (Type generalization and substitution) *Let Γ be a typing environment, e be a bare GML expression, τ be a type expression. For any substitution S,*

$$Gen(S(\Gamma), e : S(\tau)) \leq S(Gen(\Gamma, e : \tau))$$

PROOF: Let $\sigma = Gen(\Gamma, e : \tau)$ and $\sigma' = Gen(S(\Gamma), e : S(\tau))$.

If e is not generalizable, $S(\sigma) = \sigma' = S(\tau)$. Therefore $\sigma' \leq S(\sigma)$.

Otherwise, let $\{\nu_1, \ldots, \nu_n\} = FV(\tau) \setminus FV(\Gamma)$ then $\sigma = \forall \nu_1 \ldots \nu_n.\tau$. Let R be a substitution $\{\nu_i \Rightarrow \nu_i'\}$, which renames ν_i by fresh type variables ν_i'. Then, $S(\sigma) = \forall \nu_1' \ldots \nu_n'.S \circ R(\tau)$. $\sigma' = \forall V.S(\tau)$ where

$$V = FV(S(\tau)) \setminus FV(S(\Gamma)) = \bigcup_{\nu \in FV(\tau)} FV(S(\nu)) \setminus \bigcup_{\nu \in FV(\Gamma)} FV(S(\nu))$$

Let τ' be a type expression such that $\tau' \leq_{S'} \sigma'$. Then $Dom(S') \subset V$ and $S' \circ S(\tau) = \tau'$.

Let $S'' = \{\nu_i' \Rightarrow S' \circ S(\nu_i)\}$. Then, for any ν_i,

$$S'' \circ S \circ R(\nu_i) = S'' \circ S(\nu_i') = S''(\nu_i') = S' \circ S(\nu_i)$$

since ν_i' are out of S.

For any $\nu \in FV(\tau) \setminus \{\nu_1, \ldots, \nu_n\}$, $\nu \in FV(\Gamma)$ and

$$S'' \circ S \circ R(\nu) = S'' \circ S(\nu) = S(\nu)$$

since $Dom(S'')$ is the set of fresh variables $\{\nu_1', \ldots, \nu_n'\}$. $S(\nu) = S' \circ S(\nu)$ since $Dom(S') \subset V$ and V is distinct from $\bigcup_{\nu \in FV(\Gamma)} FV(S(\nu))$.

From these, $S''(S \circ R(\tau)) = S' \circ S(\tau)$, which means $\tau' \leq_{S''} S(\sigma)$. □

2.5.3 Typing rules

We use *typing judgments* $\Gamma \vdash e_t$, which is read an expression e_t is well-typed under a type environment Γ. Because our expressions have already type annotations, we do not need to write the type of e_t like what we did at the definition of *Gen*. But sometimes we write it like $\Gamma \vdash e_t : \tau$ to make clear that the out-most type annotation of e_t is τ.

The typing rules are shown in Figure 2.1. Even though we have rather complicated semantics system with type annotation instantiations than those of the typical ML languages, our typing rules are as simple as the original and there are just few things to note.

$$\textsf{(t-cnst)}\ \frac{\tau \leq \sigma}{\Gamma \vdash c_\sigma : \tau}$$

$$\textsf{(t-gen)}\ \frac{\tau \leq \sigma}{\Gamma \vdash \mathtt{g}_\sigma : \tau}$$

$$\textsf{(t-inst)}\ \frac{\Gamma(\mathtt{x}) = \sigma \quad \tau \leq \sigma}{\Gamma \vdash \mathtt{x}_\sigma : \tau}$$

$$\textsf{(t-fun)}\ \frac{\Gamma\{\mathtt{x} : \tau_1\} \vdash e_t : \tau_2}{\Gamma \vdash \mathtt{fun\ x} \rightarrow e_t : \tau_1 \rightarrow \tau_2}$$

$$\textsf{(t-app)}\ \frac{\Gamma \vdash e_{t1} : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_{t2} : \tau_1}{\Gamma \vdash e_{t1}\ e_{t2} : \tau_2}$$

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

$$\textsf{(t-let)}\ \frac{\begin{array}{c}\Gamma \vdash e_{t1} : \tau_1 \qquad \forall V.\tau_1 = Gen(\Gamma, e_{t1} : \tau_1) \\ \Gamma\{\mathtt{x} : \forall V.\tau_1\} \vdash e_{t2} : \tau_2\end{array}}{\Gamma \vdash \mathtt{let\ x}_V = e_{t1}\ \mathtt{in}\ e_{t2} : \tau_2}$$

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

Figure 2.1: Typing rules

Generic primitives $\mathtt{g}_\sigma$ is treated same as constants whose type schemes are σ in (t-gen).

The rule for recursive let definitions, (t-letrec), rejects non-functional definitions so that it can be isomorphic to their evaluation rule (e-letrec) which evaluates functions to recursive closures. Like the ordinary ML typing rules, the recursive identifier `f` is treated as monomorphic inside the typing of the definition $\mathtt{fun\ x} \rightarrow e_{t1}$. However, in the conclusion expression, the recursive definition $\mathtt{fun\ x} \rightarrow e_{t1} : \tau_1$ is *not* as it is. We need to replace the type scheme annotations of the variable occurrence of `f` from $\mathtt{f}_{\tau_1}$ to $\mathtt{f}_{\forall V.\tau_1}$. This is for proper evaluation of recursive derived generic instances; recursive occurrences of `f` inside its definition must have dynamic polymorphic type scheme annotations, otherwise they cannot be evaluated by proper semantic rules for generic value instances (e-gvar*). Even though typing of the recursive occurrences of f is monomorphic, they must have polymorphic type annotations. The following *variable type annotation modifier* used in (t-letrec):

Let $Z = \{\mathtt{f}_\sigma \Rightarrow \mathtt{f}_{\sigma'}\}$, then

$$\begin{array}{rcl}
Z(\mathtt{f}_\sigma : \tau) & = & \mathtt{f}_{\sigma'} : \tau \\
Z(\mathtt{fun\ f} \rightarrow e_t : \tau) & = & \mathtt{fun\ f} \rightarrow e_t : \tau \\
Z(\mathtt{fun\ x} \rightarrow e_t : \tau) & = & \mathtt{fun\ x} \rightarrow Z(e_t) : \tau \\
Z(e_{t1}\ e_{t2}) & = & Z(e_{t1})\ Z(e_{t2}) \\
Z(e_{t1}, e_{t2}) & = & Z(e_{t1}), Z(e_{t2}) \\
Z(\mathtt{let\ (rec)\ f}_V = e_{t1}\ \mathtt{in}\ e_{t2} : \tau) & = & \mathtt{let\ (rec)\ f}_V = e_{t1}\ \mathtt{in}\ Z(e_{t2}) : \tau \\
Z(\mathtt{let\ (rec)\ x}_V = e_{t1}\ \mathtt{in}\ e_{t2} : \tau) & = & \mathtt{let\ (rec)\ x}_V = Z(e_{t1})\ \mathtt{in}\ Z(e_{t2}) : \tau \\
\text{Otherwise, } Z(e_t) & = & e_t
\end{array}$$

Variable abstractions (functions and let bindings) guard the internal abstracted variables from the modifications.

Our typing judgments for fully type-annotated GML expressions can be transformed into typing judgments for bare GML expressions, in the familiar style of ML typing theory. If we have a bare GML expression e and if a type-annotated expression e_t such that $strip(e_t) = e$ has a typing $\Gamma \vdash e_t : \tau$, we write $\Gamma \vdash e : \tau$.

Example A bare GML expression

```
let double = fun x → plus x x in double;;
```

has the following typing, if we assume that plus is a generic primitive whose type scheme is $\forall\delta.\delta \rightarrow \delta \rightarrow \delta$ (here also some annotations are omitted):

$$\frac{\begin{array}{c}\emptyset \vdash \mathtt{fun\ x} \rightarrow \mathtt{(plus} : \delta \rightarrow \delta \rightarrow \delta\mathtt{)\ (x\!:\!}\delta\mathtt{)\ (x\!:\!}\delta\mathtt{)} : \delta \rightarrow \delta \\ \forall\delta.\delta \rightarrow \delta = Gen(\emptyset, \mathtt{fun\ x} \ldots : \delta \rightarrow \delta) \\ \{\mathtt{double} : \forall\delta.\delta \rightarrow \delta\} \vdash \mathtt{double} : \mathtt{int} \rightarrow \mathtt{int}\end{array}}{\begin{array}{ll}\emptyset \vdash & \mathtt{let\ double}_{\{\delta\}} = \mathtt{fun\ x} \rightarrow \\ & \quad \mathtt{(plus} : \delta \rightarrow \delta \rightarrow \delta\mathtt{)\ (x\!:\!}\delta\mathtt{)\ (x\!:\!}\delta\mathtt{)} : \delta \rightarrow \delta \\ & \mathtt{in\ double} : \mathtt{int} \rightarrow \mathtt{int}\end{array}}$$

The occurrences of δ inside the let definition expression are properly bound by the generalized type variable set attached at the declaration of double. □

Lemma 6 *Let e be a bare GML expression. Let Γ and Γ' be typing environments such that $\Gamma' \leq \Gamma$. If there is a typing $\Gamma' \vdash e : \tau$, $\Gamma \vdash e : \tau$.*

PROOF: By induction on the structure of the typing delivations.

(t-inst) The expression must be a variable. Let $e = \mathtt{x}$. Let the typing as follows, for some type expression τ and type scheme σ':

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

From $\Gamma' \leq \Gamma$, we have $\sigma' \leq \Gamma(\mathtt{x})$. Let $\sigma = \Gamma(\mathtt{x})$. Since $\sigma' \leq \sigma$, $\tau \leq \sigma$. We can create a typing of $\mathtt{x}$ for Γ with the same type of the expression:

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

(t-const),(t-gen) Trivial since they make no use of typing environments.

The others They are proven easily using the induction.

□

Lemma 7 (Typing stability in recursive let bindings) *If an expression* $\mathtt{fun\ x} \rightarrow e_{t1} : \tau_1$ *has a typing* $\Gamma\{\mathtt{f} : \tau_1\} \vdash (\mathtt{fun\ x} \rightarrow e_{t1}) : \tau_1$*, and* $\forall V.\tau_1 = \mathit{Gen}(\Gamma, (\mathtt{fun\ x} \rightarrow e_{t1}) : \tau_1)$ *then*

$$\Gamma\{\mathtt{f} : \forall V.\tau_1\} \vdash \{\mathtt{f}_{\tau_1} \Rightarrow \mathtt{f}_{\forall V.\tau_1}\}(\mathtt{fun\ x} \rightarrow e_{t1} : \tau_1)$$

and $\forall V.\tau_1 = \mathit{Gen}(\Gamma, \{\mathtt{f}_{\tau_1} \Rightarrow \mathtt{f}_{\forall V.\tau_1}\}(\mathtt{fun\ x} \rightarrow e_{t1} : \tau_1))$

Proof: Each target variable occurrence of the annotation modification $\{\mathtt{f}_{\tau_1} \Rightarrow \mathtt{f}_{\forall V.\tau_1}\}$ must be typed under some $\Gamma'\{\mathtt{f} : \tau_1\}$, some typing environment of $\Gamma\{\mathtt{f} : \tau_1\}$ which have additional bindings to $\Gamma\{\mathtt{f} : \tau_1\}$, as

$$\Gamma'\{\mathtt{f} : \tau_1\} \vdash \mathtt{f} : \tau_1$$

by (t-inst). Because the type scheme of $\mathtt{f}$, τ_1 is completely monomorphic, the type of the variable occurrence must be also τ_1. Because $\tau_1 \leq \forall V.\tau_1$, The typing

$$\Gamma'\{\mathtt{f} : \forall V.\tau_1\} \vdash \mathtt{f} : \tau_1$$

is also correct from Lemma 6. This modification of the type scheme of $\mathtt{f}$ only affects the $\mathtt{f}$'s typings, and the other parts of typing are never modified. Therefore $\Gamma\{\mathtt{f} : \forall V.\tau_1\} \vdash \{\mathtt{f}_{\tau_1} \Rightarrow \mathtt{f}_{\forall V.\tau_1}\}(\mathtt{fun\ x} \rightarrow e_{t1} : \tau_1)$.

The generalization function *Gen* just looks the out-most type and the form of the whole expression, ignoring the others. Therefore replacing the type annotations of $\mathtt{f}$ does not affect the result of generalization. □

Our type system is stable under substitution:

Theorem 3 (Substitution lemma of the typing rules) *Typing is stable under substitution: if there is a typing $\Gamma \vdash e_t : \tau$, then for any admissible substitution S, $S(\Gamma) \vdash S(e_t) : S(\tau)$ also holds.*

Proof: By induction over the structure of e_t.

Constants It is the case $e_t = c_\sigma$. The only applicative rule is (t-cnst), therefore we have $\Gamma \vdash c_\sigma : \tau$, where $\tau \leq_{S_0} \sigma$ for some substitution S_0. From Lemma 4,

$$S(\tau) \leq_{(S \circ S_0)|_{BV(\sigma)}} S(\sigma)$$

From the closedness of σ, we have $S(\sigma) = \sigma$. Using (t-cnst) we have $S(\Gamma) \vdash c_\sigma : S(\tau)$.

Generic primitives The proof is like one for constants, but using (t-gen) rule.

Variables Now $e_t = \mathtt{x}$, $\mathtt{x}$ has a typing $\Gamma \vdash \mathtt{x}_\sigma : \tau$ where $\tau \leq \sigma = \Gamma(\mathtt{x})$, and by Lemma 4, $S(\tau) \leq S(\sigma)$. Because $S(\Gamma)(\mathtt{x}) = S(\sigma)$, by using (t-inst) we can create the judgment $S(\Gamma) \vdash \mathtt{x}_{S(\sigma)} : S(\tau)$.

Let bindings The applicative rule is (t-let) and the judgment ends like:

$$\frac{\begin{array}{c}\Gamma \vdash e_{t1} : \tau_1 \quad \forall \nu_1 \ldots \nu_n.\tau_1 = \mathit{Gen}(\Gamma, e_{t1} : \tau_1) \\ \Gamma\{\mathtt{x} : \forall \nu_1 \ldots \nu_n.\tau_1\} \vdash e_{t2} : \tau_2\end{array}}{\Gamma \vdash \mathtt{let}\ \mathtt{x}_{\{\nu_1,\ldots,\nu_n\}} = e_{t1}\ \mathtt{in}\ e_{t2} : \tau_2}$$

Generalizable case

If e_{t1} is generalizable, $\{\nu_1, \ldots, \nu_n\} = FV(\tau_1) \setminus FV(\Gamma)$ from the definition of *Gen*.

Let S' be a substitution which renames ν_i to ν_i', new type variables which do not appear in any binding of S, Γ, nor the type annotations of e_{t1} and e_{t2}:

$$S' = \{\nu_i \Rightarrow \nu_i'\} \quad \nu_i' \text{ is a fresh variable}$$

Applying the induction hypothesis with this $S \circ S'$ to the first premise, we have:

$$(S \circ S')(\Gamma) \vdash (S \circ S')(e_{t1}) : (S \circ S')(\tau_1)$$

Since ν_i never appear in Γ freely, $S'(\Gamma) = \Gamma$. Therefore $(S \circ S')(\Gamma) = S(\Gamma)$ and

$$S(\Gamma) \vdash (S \circ S')(e_{t1}) : (S \circ S')(\tau_1)$$

Generalization of the type $(S \circ S')(\tau_1)$ is

$$\mathit{Gen}(S(\Gamma), (S \circ S')(e_{t1}) : (S \circ S')(\tau_1)) = \forall V.(S \circ S')(\tau_1)$$

where $V = FV((S \circ S')(\tau_1)) \setminus FV(S(\Gamma))$.

We prove $V = \{\nu'_i\}$. From Lemma 1 ,

$$FV((S \circ S')(\tau_1)) = \bigcup_{\nu \in FV(\tau_1)} FV((S \circ S')(\nu))$$

$$FV(S(\Gamma)) = \bigcup_{\nu \in FV(\Gamma)} FV(S(\nu))$$

and $\nu'_i \in FV((S \circ S')(\tau_1))$, because $\nu_i \in FV(\tau_1)$ and $(S \circ S')(\nu_i) = S(\nu'_i) = \nu'_i$. On the other hand $\nu'_i \notin FV(S(\Gamma))$, because ν'_i is a fresh variable. This proves $\{\nu'_i\} \subset V$.

Now we are going to prove the converse, $V \subset \{\nu'_i\}$. Suppose ν' is a variable in V, but not one of ν'_i. There must be a variable $\nu'' \in FV(\tau_1)$ such that $\nu' \in FV((S \circ S')(\nu''))$. ν'' cannot be one of ν_i, otherwise

$$(S \circ S')(\nu'') = (S \circ S')(\nu_i) = S(\nu'_i) = \nu'_i$$

and it would contradict our assumption that ν' is not one of ν'_i.

Since ν'' is not one of ν_i, $(S \circ S')(\nu'') = S(\nu'')$, hence $\nu' \in FV(S(\nu''))$. Since $\{\nu_i\} = FV(\tau_1) \setminus FV(\Gamma)$ and $\nu'' \in FV(\tau_1)$, we have $\nu'' \in FV(\Gamma)$. These lead to $\nu' \in \bigcup_{\nu \in FV(\Gamma)} FV(S(\nu)) \subset FV(S(\Gamma))$. It contradicts our assumption that $\nu' \in V$.

Finally,

$$Gen(S(\Gamma), (S \circ S')(e_{t1}) : (S \circ S')(\tau_1)) = \forall \nu'_1 \ldots \nu'_n.(S \circ S')(\tau_1)$$

Because ν'_i's are fresh, they are out of the reach of S. Therefore $\forall \nu'_1 \ldots \nu'_n.(S \circ S')(\tau_1) = S(\forall \nu'_1 \ldots \nu'_n.S'(\tau_1))$. Since $\forall \nu'_1 \ldots \nu'_n.S'(\tau_1) \overset{\alpha}{=} \forall \nu_1 \ldots \nu_n.\tau_1$, the generalization is rewritten as follows:

$$Gen(S(\Gamma), (S \circ S')(e_{t1}) : (S \circ S')(\tau_1)) \overset{\alpha}{=} S(\forall \nu_1 \ldots \nu_n.\tau_1)$$

Now we use the induction hypothesis with S for the second premise and obtain:

$$S(\Gamma\{\mathtt{x} : \forall \nu_1 \ldots \nu_n.\tau_1\}) \vdash S(e_{t2}) : S(\tau_2)$$

From these, we prove the following:

$$\frac{\begin{array}{c} S(\Gamma) \vdash (S \circ S')(e_{t1}) : (S \circ S')(\tau_1) \\ Gen(S(\Gamma), (S \circ S')(e_{t1}) : (S \circ S')(\tau_1)) \;=\; \forall \nu'_1 \ldots \nu'_n.(S \circ S')(\tau_1) \\ \qquad\qquad\qquad\qquad\qquad\qquad \overset{\alpha}{=} \; S(\forall \nu_1 \ldots \nu_n.\tau_1) \\ S(\Gamma)\{\mathtt{x} : S(\forall \nu_1 \ldots \nu_n.\tau_1)\}) \vdash S(e_{t2}) : S(\tau_2) \end{array}}{S(\Gamma) \vdash \mathtt{let}\ \mathtt{x}_{\{\nu'_i\}} \mathtt{=} (S \circ S')(e_{t1})\ \mathtt{in}\ S(e_{t2}) : S(\tau_2)}$$

The conclusion expression is α-equivalent to $S(\mathtt{let}\ \mathtt{x}_{\{\nu_i\}} = e_{t1}\ \mathtt{in}\ e_{t2} : \tau_2)$.

Non generalizable case

If *generalizable*(e_{t1}) is false, from the definition of *Gen*,

$$\sigma = Gen(\Gamma, e_{t1} : \tau_1) = \tau_1$$

$$Gen(S(\Gamma), S(e_{t1}) : S(\tau_1)) = S(\tau_1)$$

therefore $S(\sigma) = S(\tau_1) = Gen(S(\Gamma), S(e_{t1}) : S(\tau_1))$ holds. From this we can prove the goal.

Recursive let bindings The proof goes same as the case of let bindings. The typing is obtained by (t-letrec):

$$\frac{\Gamma\{\mathtt{f} : \tau_1\} \vdash (\mathtt{fun\ x} \rightarrow e_{t1}) : \tau_1 \quad \forall V.\tau_1 = Gen(\Gamma, (\mathtt{fun\ x} \rightarrow e_{t1}) : \tau_1) \quad \Gamma\{\mathtt{f} : \forall V.\tau_1\} \vdash e_{t2} : \tau_2}{\Gamma \vdash \mathtt{let\ rec\ f}_V = \{\mathtt{f}_{\tau_1} \Rightarrow \mathtt{f}_{\forall V.\tau_1}\}(\mathtt{fun\ x} \rightarrow e_{t1} : \tau_1)\ \mathtt{in}\ e_{t2} : \tau_2}$$

We prepare a substitution S' which renames all the variables in V to fresh variables V', and use the induction hypothesis with $S \circ S'$ for the first premise:

$$(S \circ S')(\Gamma\{\mathtt{f} : \tau_1\}) \vdash (S \circ S')(\mathtt{fun\ x} \rightarrow e_{t1}) : (S \circ S')(\tau_1)$$

Because the variables in V never appear in Γ freely, $S'(\Gamma) = \Gamma$. Therefore we obtain:

$$S(\Gamma\{\mathtt{f} : \tau_1\}) \vdash (S \circ S')(\mathtt{fun\ x} \rightarrow e_{t1}) : (S \circ S')(\tau_1)$$

The generalization of the type $(S \circ S')(\tau_1)$ is

$$Gen(S(\Gamma), (S \circ S')(\mathtt{fun\ x} \rightarrow e_{t1}) : (S \circ S')(\tau_1)) = \forall V''.(S \circ S')(\tau_1)$$

where $V'' = FV((S \circ S')(\tau_1)) \setminus FV(S(\Gamma))$. We can prove $V' = V''$, as we did in the case of the non recursive let bindings. The generalization is rewritten as:

$$Gen(S(\Gamma), (S \circ S')(\mathtt{fun\ x} \rightarrow e_{t1}) : (S \circ S')(\tau_1)) = S(\forall V'.S'(\tau_1)) \stackrel{\alpha}{=} S(\forall V.\tau_1)$$

Now we use the induction hypothesis with S for the second premise:

$$S(\Gamma\{\mathtt{f} : \forall V.\tau_1\}) \vdash S(e_{t2}) : S(\tau_2)$$

From these, using (t-letrec),

$$\frac{\begin{array}{c} S(\Gamma\{\mathtt{f} : \tau_1\}) \vdash (S \circ S')(\mathtt{fun\ x} \rightarrow e_{t1}) : (S \circ S')(\tau_1) \\ Gen(S(\Gamma), (S \circ S')(\mathtt{fun\ x} \rightarrow e_{t1}) : (S \circ S')(\tau_1)) = \forall V'.(S \circ S')(\tau_1) \stackrel{\alpha}{=} S(\forall V.\tau_1) \\ S(\Gamma\{\mathtt{f} : \forall V.\tau_1\}) \vdash S(e_{t2}) : S(\tau_2) \end{array}}{\begin{array}{l} S(\Gamma) \vdash \mathtt{let\ rec\ f}_{V'} = \\ \qquad \{\mathtt{f}_{S(\tau_1)} \Rightarrow \mathtt{f}_{\forall V'.(S \circ S')(\tau_1)}\}((S \circ S')(\mathtt{fun\ x} \rightarrow e_{t1}))\ \mathtt{in}\ S(e_{t2}) : S(\tau_2) \end{array}}$$

The conclusion expression is equal to

$$S(\texttt{let rec f}_{V'} = \{\texttt{f}_{\tau_1} \Rightarrow \texttt{f}_{\forall V'.S'(\tau_1)}\}(S'(\texttt{fun x} \to e_{t1}))\ \texttt{in}\ e_{t2} : \tau_2)$$

which is α-equivalent to

$$S(\texttt{let rec f}_{V} = \{\texttt{f}_{\tau_1} \Rightarrow \texttt{f}_{\forall V.\tau_1}\}(\texttt{fun x} \to e_{t1})\ \texttt{in}\ e_{t2} : \tau_2)$$

λ bindings, Applications, tuples The proofs of these cases are done in the same following manner: we have only one applicative rule for each construct, (t-fun), (t-app), and (t-tuple) respectively. We apply the induction hypothesis for the premises of each rule, then from these by using the same rule we can create the judgment $S(\Gamma) \vdash S(e_t) : S(\tau)$ easily.

□

2.5.4 Type soundness

A fully type-annotated expression e_t is called *well-typed* under a typing environment Γ, if it has a typing under Γ, i.e. $\Gamma \vdash e_t$. A *program* is an annotated expression e_t and it is evaluated under the empty evaluation environment. A program e_t is well-typed when e_t is a well-typed expression under the empty typing environment, $\emptyset \vdash e_t$. We reject the evaluation of the programs which lack typings. Contrary, if programs are well-typed, they are never evaluated to the error err:

Theorem 4 (Type soundness) *If a fully type-annotated expression e_t has a typing in the empty typing environment, $\emptyset \vdash e_t$, and the evaluation of e_t under the empty evaluation environment $\emptyset \models e_t$ terminates in some result r, then r is not err but a value.*

Theorem 4 does not assure termination of well-typed programs. It says that if a well-typed program terminates, the result must not be an error. Because of the existence of the recursive let definitions, evaluations can loop infinitely due to the (e-letrec) evaluation rule.

Since this property is too weak to prove directly, we need to generalize this theorem to something which also holds for the well-typed expressions under non-empty environments. For each evaluation derivation, we will prove that well-typed expression $e_t : \tau$ is not evaluated to err, but to some value v. Moreover, the return value v must be a semantic value belongs to the type τ. Hence for example we can make sure that the functional expressions are evaluated to functional semantic values (i.e. closures) in each step of the evaluation. For this purpose, we need to construct a typing system for the semantic values.

2.5.5 Typing of semantic values

In our *semantic typing system*, we have two kinds of judgments, $sv \in \sigma$ and $E \in \Gamma$. They mean that a semantic value sv has a semantic typing as σ, or an evaluation environment E has a semantic typing Γ. The semantic typing rules are in Figure 2.2.

The semantic typing rules for basic constants (s-bcnst), tuples (s-tpl), and the closures (s-clos*) and (s-gclos*) are coherent to their corresponding typing rules. In (s-clos-r), recursive functional closures may have polymorphic type schemes, since only the recursive let expressions can create recursive closures and its typing rule (t-letrec) reannotates defined identifiers with polymorphic type schemes. (s-cnst) and (s-gen) are our assumptions that the predefined semantic bindings of the constants and generic primitives are all semantically well-typed. This assumption for (s-gen) is actually too strong and may cause *coherence problem*. We will see it further in Section 2.5.6.

$$\textbf{(s-bcnst)}\ \frac{\tau \leq \sigma}{b_\sigma \in \tau}$$

$$\textbf{(s-tpl)}\ \frac{v_1 \in \tau_1 \quad v_2 \in \tau_2}{v_1, v_2 \in \tau_1 * \tau_2}$$

$$\textbf{(s-clos)}\ \frac{\exists\Gamma, \quad E \in \Gamma \quad \Gamma \vdash \texttt{fun x} \rightarrow e_t : \tau_1 \rightarrow \tau_2}{clos(E, \_, \texttt{x}, e_t) \in \tau_1 \rightarrow \tau_2}$$

$$\textbf{(s-clos-r)}\ \frac{\begin{array}{c}\exists\Gamma, \quad E \in \Gamma \quad \Gamma\{\texttt{f} : \forall V.\tau_1 \rightarrow \tau_2\} \vdash \texttt{fun x} \rightarrow e_t : \tau_1 \rightarrow \tau_2 \\ \forall V.\tau_1 \rightarrow \tau_2 = Gen(\Gamma, \texttt{fun x} \rightarrow e_t) \quad V \cap \mathbb{D} = \emptyset\end{array}}{clos(E, \texttt{f}, \texttt{x}, e_t) \in \forall V.\tau_1 \rightarrow \tau_2}$$

$$\textbf{(s-gclos)}\ \frac{\exists\Gamma, \quad E \in \Gamma \quad \Gamma \vdash e_t : \tau \quad \forall V.\tau = Gen(\Gamma, e_t : \tau) \quad V \cap \mathbb{D} \neq \emptyset}{gclos(E, \_, V, e_t : \tau) \in \forall V.\tau}$$

$$\textbf{(s-gclos-r)}\ \frac{\exists\Gamma, \quad E \in \Gamma \quad \Gamma\{\texttt{x} : \forall V.\tau\} \vdash e_t : \tau \quad \forall V.\tau = Gen(\Gamma, e_t : \tau) \quad V \cap \mathbb{D} \neq \emptyset}{gclos(E, \texttt{x}, V, e_t : \tau) \in \forall V.\tau}$$

$$\textbf{(s-env)}\ \frac{\forall(\texttt{x} : \sigma) \in \Gamma, \quad E(\texttt{x}) = sv \quad sv \in \sigma}{E \in \Gamma}$$

$$\textbf{(s-schm)}\ \frac{\forall\tau \leq_S \sigma, \quad S(v) \in \tau}{v \in \sigma}$$

$$\textbf{(s-cnst)}\ \frac{\tau \leq_S \sigma}{S(cnst(c_\sigma)) \in \tau}$$

$$\textbf{(s-gen)}\ \frac{\tau \leq \sigma}{gen(\mathsf{g}_\sigma, \tau) \in \tau}$$

Figure 2.2: The semantic typing rules

Lemma 8 (Substitution lemma of semantic typing) *If a semantic value sv has a semantic typing $sv \in \sigma$, then $S(sv) \in S(\sigma)$ for any substitution S. If we have an evaluation environment E such that $E \in \Gamma$ for some typing environment Γ, for any substitution S, $S(E) \in S(\Gamma)$.*

PROOF: By induction on the typing derivation of sv.

(s-bcnst) We have $b_\sigma \in \tau$ and $\tau \leq \sigma$. Because σ is closed, for any substitution S, $S(\tau) \leq S(\sigma) = \sigma$, therefore using (s-bcnst) and we obtain $S(b_\sigma) = b_\sigma \in S(\tau)$.

(s-tpl) Let $v = (v_1, v_2)$. Using the induction hypothesis we have $S(v_i) \in S(\tau_i)$. Applying (s-tpl) against these, we get $S(v_1, v_2) \in S(\tau_1 * \tau_2)$.

(s-clos) Suppose that

$$v = clos(E, \_, \mathtt{x}, e_t) \in \tau_1 \to \tau_2$$

This requires that we have some typing environment Γ such that $E \in \Gamma$ and $\Gamma \vdash \mathtt{fun\ x} \to (e_t : \tau_2) : \tau_1 \to \tau_2$. By Theorem 3,

$$S(\Gamma) \vdash S(\mathtt{fun\ x} \to e_t) : S(\tau_1 \to \tau_2)$$

Using the induction hypothesis for $E \in \Gamma$, we have $S(E) \in S(\Gamma)$. From these we use (s-clos) and obtain:

$$clos(S(E), \_, \mathtt{x}, S(e_t)) = S(clos(E, \_, \mathtt{x}, e_t)) \in S(\tau_1 \to \tau_2)$$

(s-clos-r) Suppose

$$v = clos(E, \mathtt{f}, \mathtt{x}, e_t) \in \forall V.\tau_1 \to \tau_2$$

then from this semantic typing, we have Γ such that $E \in \Gamma$ and $\Gamma\{\mathtt{f} : \forall V.\tau_1 \to \tau_2\} \vdash \mathtt{fun\ x} \to (e_t : \tau_2) : \tau_1 \to \tau_2$, where $\forall V.\tau_1 \to \tau_2 = Gen(\Gamma, \mathtt{fun\ x} \to (e_t : \tau_2) : \tau_1 \to \tau_2)$. We rename the variables in V appear in the typing to the fresh ones V' by using a substitution $S' = \{\nu_i \Rightarrow \nu_i' \mid \nu_i \in V\}$. We also α-convert the type scheme of $\mathtt{f}$. In the same way as we did in the substitution lemma of let typing, we have:

$$S'(\Gamma)\{\mathtt{f} : \forall V'.S'(\tau_1 \to \tau_2)\} \vdash S'(\mathtt{fun\ x} \to (e_t : \tau_2) : \tau_1 \to \tau_2)$$
$$\forall V'.S'(\tau_1 \to \tau_2) = Gen(S'(\Gamma), S'(\mathtt{fun\ x} \to (e_t : \tau_2) : \tau_1 \to \tau_2))$$

Because V is a set of variables which never appear in Γ freely, $S'(\Gamma) = \Gamma$.

Using Theorem 3 with the fact that V' are fresh and out of the reach of S,

$$S(\Gamma)\{\mathtt{f} : \forall V'.(S \circ S')(\tau_1 \to \tau_2)\} \vdash (S \circ S')(\mathtt{fun\ x} \to (e_t : \tau_2) : \tau_1 \to \tau_2)$$
$$\forall V'.(S \circ S')(\tau_1 \to \tau_2) = Gen(S(\Gamma), (S \circ S')(\mathtt{fun\ x} \to (e_t : \tau_2) : \tau_1 \to \tau_2))$$

From the induction hypothesis for $E \in \Gamma$, we have $S(E) \in S(\Gamma)$. We use (s-clos-r) and obtain:

$$clos(S(E), \mathtt{f}, \mathtt{x}, S(e_t)) = S(clos(E, \mathtt{f}, \mathtt{x}, e_t)) \in \forall V'.(S \circ S')(\tau_1 \to \tau_2)$$

where $\forall V'.(S \circ S')(\tau_1 \to \tau_2) \overset{\alpha}{=} S(\forall V.\tau_1 \to \tau_2)$.

(s-gclos)

$$\frac{\exists\Gamma, \quad E \in \Gamma \quad \Gamma \vdash e_t : \tau \quad \forall V.\tau = Gen(\Gamma, e_t : \tau)}{gclos(E, \_, V, e_t : \tau) \in \forall V.\tau}$$

Like the part of the let bindings of the proof of Theorem 3, we prepare a substitution S' which renames all the type variables in V to fresh variables:

$$S' = \{(\nu_i \Rightarrow \nu_i') \mid \nu_i \in V\} \text{ where } \nu_i'\text{'s are fresh}$$

and let $V' = \{\nu_i'\}$. By Theorem 3, we have $(S \circ S')(\Gamma) \vdash (S \circ S')(e_t) : (S \circ S')(\tau)$ and $(S \circ S')(\Gamma) = S(\Gamma)$. Therefore:

$$S(\Gamma) \vdash (S \circ S')(e_t) : (S \circ S')(\tau)$$

The generalization of this $(S \circ S')(\tau)$ is $Gen(S(\Gamma), (S \circ S')(e_t) : (S \circ S')(\tau))$ and it is equal to $\forall V'.(S \circ S')(\tau) \stackrel{\alpha}{=} S(\forall V.\tau)$ also by the same discussion of Theorem 3. Using the induction hypothesis with S for $E \in \Gamma$ and (s-gclos), we have

$$\frac{\begin{array}{c} S(E) \in S(\Gamma) \quad S(\Gamma) \vdash (S \circ S')(e_t) \\ S(\forall V.\tau) \stackrel{\alpha}{=} \forall V'.(S \circ S')(\tau) = Gen(S(\Gamma), (S \circ S')(e_t)) \end{array}}{gclos(S(E), \_, V', (S \circ S')(e_t)) \in S(\forall V.\tau)}$$

The generic closure in the conclusion is α-equivalent to $S(gclos(E, \_, V, e_t))$.

(s-gclos-r) This is almost same as (s-gclos).

$$\frac{\exists\Gamma, \quad E \in \Gamma \quad \Gamma\{\mathtt{x} : \tau\} \vdash e_t : \tau \quad \forall V.\tau = Gen(\Gamma, e_t : \tau)}{gclos(E, \mathtt{x}, V, e_t : \tau) \in \forall V.\tau}$$

Similar to (s-gclos), let S' be a substitution which renames all the type variables in V to fresh variables: $S' = \{(\nu_i \Rightarrow \nu_i') \mid \nu_i \in V\}$ and let $V' = \{\nu_i'\}$. By Theorem 3, we have $(S \circ S')(\Gamma\{\mathtt{x} : \tau\}) \vdash (S \circ S')(e_t) : (S \circ S')(\tau)$ and $(S \circ S')(\Gamma) = S(\Gamma)$. Therefore:

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

The generalization $(S \circ S')(\tau)$ is $Gen(S(\Gamma), (S \circ S')(e_t) : (S \circ S')(\tau)) = \forall V'.(S \circ S')(\tau) \stackrel{\alpha}{=} S(\forall V.\tau)$. Using the induction hypothesis with S for $E \in \Gamma$ and (s-gclos), we have

$$\frac{\begin{array}{c} S(E) \in S(\Gamma) \quad S(\Gamma)\{\mathtt{x} : (S \circ S')(\tau)\} \vdash (S \circ S')(e_t) \\ S(\forall V.\tau) \stackrel{\alpha}{=} \forall V'.(S \circ S')(\tau) = Gen(S(\Gamma), (S \circ S')(e_t)) \end{array}}{gclos(S(E), \mathtt{x}, V', (S \circ S')(e_t)) \in S(\forall V.\tau)}$$

The generic closure in the conclusion is α-equivalent to $S(gclos(E, \mathtt{x}, V, e_t))$.

(s-env) For any $(\mathtt{x} : \sigma) \in \Gamma$, there is a semantic value sv such that $E(\mathtt{x}) = sv$ and $sv \in \sigma$. Using the induction hypothesis, we have $S(E(\mathtt{x})) = S(sv) \in S(\sigma)$ From it and $(\mathtt{x} : S(\sigma)) \in S(\Gamma)$,

$$S(E) \in S(\Gamma)$$

(s-schm) We have

$$\frac{\forall \tau \leq_{S_0} \sigma \quad S_0(v) \in \tau}{v \in \sigma}$$

and we can let $\sigma = \forall \nu_1 \ldots \nu_n . \tau'$ so that ν_i never appears in the bindings of S, without losing any generality. Any instance τ'' of $S(\sigma)$, $\tau'' \leq_{S''} S(\sigma) = \forall \nu_1 \ldots \nu_n . S(\tau')$, τ'' is written as $(S'' \circ S)(\tau')$, therefore $\tau'' \leq_{(S'' \circ S)} \sigma$. By induction hypothesis,

$$S''(S(v)) \in \tau''$$

Since this holds for any instance τ'' of $S(\sigma)$, $\tau'' \leq_{S''} S(\sigma)$, using (s-schm) and this result, we have

$$S(v) \in S(\sigma)$$

□

Lemma 9 (Generality of semantic typing) *If a value v has a semantic typing $v \in \tau$, $v \in \forall V.\tau$ where $V \subset FV(\tau)$.*

PROOF: It is clear from Lemma 8 and (s-schm) rule. □

With the semantic typing system, we can enforce Theorem 4 and obtain the following theorem which can be proven by induction over the evaluation steps.

Theorem 5 (Stronger type soundness) *If $\Gamma \vdash e_t : \tau$ for some Γ and there exists E such that $E \in \Gamma$ and $E \models e_t : \tau \Longrightarrow r$, then r is not err but a value v and $v \in \tau$ holds.*

PROOF: By induction over the size of evaluation derivations $E \models e_t \Longrightarrow r$.

Constants Let $e_t = c_\sigma : \tau$, then its typing is given by (t-cnst):

$$\frac{\tau \leq_S \sigma}{\Gamma \vdash c_\sigma : \tau}$$

Constants are evaluated by (e-cnst) rather explicitly using the function *cnst*:

$$\frac{\tau \leq_S \sigma}{E \models c_\sigma : \tau \Longrightarrow S(cnst(c_\sigma))}$$

As we have assumed in (s-cnst), the result $S(cnst(c_\sigma))$ must return a semantic value who belongs to the type τ.

Generic primitives Generic primitives are evaluated by (e-gen) rule:

$$E \models \mathsf{g}_\sigma : \tau \Longrightarrow gen(\mathsf{g}, \tau)$$

From the assumption (s-gen), $gen(\mathsf{g}, \tau) \in \tau$.

Tuples We have a typing

$$\frac{\Gamma \vdash e_{t1} : \tau_1 \quad \Gamma \vdash e_{t2} : \tau_2}{\Gamma \vdash (e_{t1}, e_{t2}) : \tau_1 * \tau_2}$$

and the possible evaluation rules are (e-tpl) and (e-tpl-err), but (e-tpl-err) contradicts the induction hypothesis; one of the sub-expression e_{ti} would be evaluated to err whose evaluation size is smaller than the tuple itself.

Therefore (e-tpl) is the only applicable rule:

$$\frac{E \models e_{t1} \Longrightarrow v_1 \quad E \models e_{t2} \Longrightarrow v_2}{E \models (e_{t1}, e_{t2}) : \tau \Longrightarrow (v_1, v_2)}$$

Using the induction hypothesis for each tuple content e_{ti} we have $v_i \in \tau_i$, from these by (s-tuple),

$$(v_1, v_2) \in \tau_1 * \tau_2$$

Variables For variables we have several rules, but we do not need to think about (e-var-err), because from $\Gamma \vdash \mathtt{x}_\sigma : \tau$, $\tau \leq \sigma$ such that $\Gamma(\mathtt{x}) = \sigma$ and $E \in \Gamma$, $E(\mathtt{x}) = sv$ for some semantic value sv and $sv \in \sigma$.

The rule we can use depends on the form of sv. If sv is a value v, it must be (e-nvar) because $\tau \leq_S \sigma$ for some S, and from it we have

$$E \models \mathtt{x} : \tau \Longrightarrow v$$

Since $\tau \leq_S \sigma$ and $v \in \sigma$, from (s-schm) $v \in \tau$ holds.

If sv is a non-recursive generic closure, from (s-gclos),

$$sv = gclos(E', \_, V, e_t : \tau') \in \sigma = \forall V.\tau'$$

and the following things hold:

$$\exists \Gamma' \quad E' \in \Gamma' \quad \forall V.\tau' = Gen(\Gamma', e_t : \tau')$$

Because $\tau \leq \sigma$, (e-gvar-err) is never applied. Only (e-gvar) is available:

$$\frac{\exists S,\ \tau \leq_S \forall V.\tau' \quad E' \models S(e_t : \tau') \Longrightarrow r}{E\{\mathtt{x} = gclos(E', \_, V, e_t : \tau')\} \models \mathtt{x} : \tau \Longrightarrow r}$$

Because $Dom(S) = V$ and none of the variables in V appear freely in Γ' from the generalization, $S(\Gamma') = \Gamma'$ and $S(E') = E'$. Therefore $E' = S(E') \in S(\Gamma') = \Gamma'$ and $S(\Gamma') \vdash S(e_t : \tau')$ from Theorem 3. We can use the induction hypothesis for $S(e_t)$:

$$E' \models S(e_t : \tau') \Longrightarrow v \in S(\tau')$$

Since $S(\tau') = \tau$, we have $v \in \tau$. And finally we can prove

$$E\{\mathtt{x} = gclos(E', \_, V, e_t : \tau')\} \models \mathtt{x} : \tau \Longrightarrow v \in \tau$$

The case if v is a recursive generic closure, $gclos(E', \mathtt{f}, V, e_t)$, is also proven in the same way. Instead of (e-gvar), we use (e-gvar-r).

Functions The typing is

$$\frac{\Gamma\{\mathtt{x} : \tau_1\} \vdash e_t : \tau_2}{\Gamma \vdash \mathtt{fun\ x} \rightarrow e_t : \tau_1 \rightarrow \tau_2}$$

and the last evaluation step must use (e-fun):

$$E \models \mathtt{fun\ x} \rightarrow e_t \Longrightarrow clos(E, \_, \mathtt{x}, e_t)$$

where $E \in \Gamma$. From them we can apply (s-clos) and obtain

$$clos(E, \_, \mathtt{x}, e_t) \in \tau_1 \rightarrow \tau_2$$

Let bindings We have the following typing

$$\frac{\Gamma \vdash e_{t1} : \tau_1 \quad \forall V.\tau_1 = Gen(\Gamma, e_{t1} : \tau_1) \quad \Gamma\{\mathtt{x} : \forall V.\tau_1\} \vdash e_{t2} : \tau_2}{\Gamma \vdash \mathtt{let\ x}_V \mathtt{=} e_{t1}\ \mathtt{in}\ e_{t2} : \tau_2}$$

The possible evaluation rules are dependent on whether $V \cap \mathbb{D} = \emptyset$ or not.

<u>Normal let</u>

If $V \cap \mathbb{D} = \emptyset$, it is a normal let definition which defines a non-generic value. The possible rules are (e-let) and (e-let-err). From the same reasoning we have done for tuples, we can say that (e-let-err) cannot be used and (e-let) is the only applicable one.

$$\frac{V \cap \mathbb{D} = \emptyset \quad E \models e_{t1} : \tau_1 \Longrightarrow v_1 \quad E\{\mathtt{x} = v_1\} \models e_{t2} : \tau_2 \Longrightarrow r}{E \models \mathtt{let\ x}_V \mathtt{=} (e_{t1} : \tau_1)\ \mathtt{in}\ (e_{t2} : \tau_2) : \tau \Longrightarrow r}$$

Using the induction hypothesis for the evaluation of this e_{t1}, we have $v_1 \in \tau_1$. From Lemma 9, $v_1 \in \forall V.\tau_1$ and $E\{\mathtt{x} = v_1\} \in \Gamma\{\mathtt{x} : \forall V.\tau_1\}$, therefore we can use the induction hypothesis again for the evaluation of e_{t2}:

$$E\{\mathtt{x} = v_1\} \models e_{t2} : \tau_2 \Longrightarrow v_2 \in \tau_2$$

And from $v_2 \in \tau_2$, we prove

$$E \models \mathtt{let\ x}_V \,\mathtt{=}\, (e_{t1} : \tau_1)\ \mathtt{in}\ (e_{t2} : \tau_2) : \tau_2 \Longrightarrow v \in \tau_2$$

<u>Generic let</u>

If V contains dynamic variables, $V \cap \mathbb{D} \neq \emptyset$, the expression defines a derived generic value. We have (e-g-let) as the only applicable rule:

$$\frac{V \cap \mathbb{D} \neq \emptyset \quad E\{\mathtt{x} = gclos(E, \_, V, e_{t1} : \tau_1)\} \models e_{t2} : \tau_2 \Longrightarrow r}{E \models \mathtt{let\ x}_V \,\mathtt{=}\, (e_{t1} : \tau_1)\ \mathtt{in}\ (e_{t2} : \tau_2) : \tau_2 \Longrightarrow r}$$

From (s-gclos), $gclos(E, \_, V, e_{t1} : \tau_1) \in \forall V.\tau_1$ and therefore

$$E\{\mathtt{x} = gclos(E, \_, V, e_{t1} : \tau_1)\} \in \Gamma\{\mathtt{x} : \forall V.\tau_1\}$$

This permits us to use the induction hypothesis for the evaluation of e_{t2}. It must return a value $v \in \tau_2$. And it is our goal.

Recursive let bindings We can prove in the same way as the non-recursive let bindings, using the recursive version of rules and the rule for the functions. The typing of the expression must be as follows by (t-letrec) with the expression e'_{t1} such that $e_{t1} = \{\mathtt{f}_{\tau_1} \Rightarrow \mathtt{f}_{\forall V.\tau_1}\} e'_{t1}$:

$$\frac{\Gamma\{\mathtt{f} : \tau_1\} \vdash \mathtt{fun\ x} \rightarrow e'_{t1} : \tau_1 \quad \forall V.\tau_1 = Gen(\Gamma\{\mathtt{f} : \tau_1\}, e_{t1} : \tau_1) \qquad \Gamma\{\mathtt{f} : \forall V.\tau_1\} \vdash e_{t2} : \tau_2 \quad e_{t1} = \{\mathtt{f}_{\tau_1} \Rightarrow \mathtt{f}_{\forall V.\tau_1}\} e'_{t1}}{\Gamma \vdash \mathtt{let\ rec\ f}_V \,\mathtt{=}\, \mathtt{fun\ x} \rightarrow e_{t1}\ \mathtt{in}\ e_{t2} : \tau_2}$$

<u>Normal let</u>

If $V \cap \mathbb{D} = \emptyset$, the definition is not a generic one and its evaluation is done by (e-letrec). (e-letrec-err) cannot be applied.

$$\frac{V \cap \mathbb{D} = \emptyset \quad E\{\mathtt{f} = clos(E, \mathtt{f}, \mathtt{x}, e_{t1})\} \models e_{t2} : \tau_2 \Longrightarrow r}{E \models \mathtt{let\ rec\ f}_V \,\mathtt{=}\, \mathtt{fun\ x} \rightarrow e_{t1} : \tau_1\ \mathtt{in}\ e_{t2} : \tau_2 \Longrightarrow r}$$

By (s-clos-r), $clos(E, \mathtt{f}, \mathtt{x}, e_{t1}) \in \forall V.\tau_1$. Therefore

$$E\{\mathtt{f} = clos(E, \mathtt{f}, \mathtt{x}, e_{t1})\} \in \Gamma\{\mathtt{f} : \forall V.\tau_1\}$$

and we can use the induction hypothesis again for the evaluation of e_{t2}:

$$E\{\mathtt{f} = clos(E, \mathtt{f}, \mathtt{x}, e_{t1})\} \models e_{t2} : \tau_2 \Longrightarrow v_2 \in \tau_2$$

this leads to the evaluation of the whole expression:

$$E \models \texttt{let rec f}_V\,\texttt{=fun x} \to e_{t1} \texttt{ in } e_{t2} : \tau_2 \Longrightarrow v \in \tau_2$$

Generic let

If V contains dynamic variables i.e. $V \cap \mathbb{D} \neq \emptyset$, the expression defines a derived generic value. We have (e-g-letrec) as the only applicable evaluation rule:

$$\frac{V \cap \mathbb{D} \neq \emptyset \quad E\{\texttt{x} = gclos(E, \texttt{x}, V, \texttt{fun x} \to e_{t1} : \tau_1)\} \models e_{t2} : \tau_2 \Longrightarrow r}{E \models \texttt{let rec x}_V\,\texttt{=fun x} \to e_{t1} : \tau_1 \texttt{ in } (e_{t2} : \tau_2) : \tau_2 \Longrightarrow r}$$

From (s-gclos-r), $gclos(E, \texttt{x}, V, \texttt{fun x} \to e_{t1} : \tau_1) \in \forall V.\tau_1$ and therefore

$$E\{\texttt{x} = gclos(E, \texttt{x}, V, \texttt{fun x} \to e_{t1} : \tau_1)\} \in \Gamma\{\texttt{x} : \forall V.\tau_1\}$$

This permits us to use the induction hypothesis to prove the safe evaluation of e_{t2} to $v \in \tau_2$.

Application The typing is

$$\frac{\Gamma \vdash e_{t1} : \tau_1 \to \tau_2 \quad \Gamma \vdash e_{t2} : \tau_1}{\Gamma \vdash (e_{t1} : \tau_1 \to \tau_2)\ (e_{t2} : \tau_1) : \tau_2}$$

and the evaluation of the argument e_{t2} is proven successful in the same way we did for tuples. Therefore (e-app-err1) is never used. (e-app-err2) is neither possible, because using the induction hypothesis, $e_{t1} : \tau_1 \to \tau_2$ must return a closure from (s-clos).

If the last used evaluation rule is (e-app1), e_{t1} is evaluated to non-recursive closure:

$$\frac{\begin{array}{c} E \models e_{t2} : \tau_1 \Longrightarrow v_2 \\ E \models e_{t1} : \tau_1 \to \tau_2 \Longrightarrow clos(E', \_, \texttt{x}, e_{t3} : \tau_3) \\ E'\{\texttt{x} = v_2 : \tau_1\} \models e_{t3} : \tau_3 \Longrightarrow r \end{array}}{E \models e_{t1}\ e_{t2} \Longrightarrow r}$$

for some $e_{t3} : \tau_3$. Using the induction hypothesis for the first two premises, we have

$$v_2 \in \tau_1$$

$$clos(E', \_, \texttt{x}, e_{t3} : \tau_3) \in \tau_1 \to \tau_2$$

From (s-clos), $\tau_3 = \tau_2$ and there must be a type environment Γ' such that $E' \in \Gamma'$ and

$$\Gamma' \vdash \texttt{fun x} \to e_{t3} : \tau_1 \to \tau_2$$

This typing must end with (t-fun):

$$\frac{\Gamma'\{\texttt{x} : \tau_1\} \vdash e_{t3} : \tau_2}{\Gamma' \vdash \texttt{fun x} \to e_{t3} : \tau_1 \to \tau_2}$$

From this and $v_2 \in \tau_1$, $E'\{\mathtt{x} = v_2\} \in \Gamma'\{\mathtt{x} : \tau_1\}$. We can now use the induction hypothesis for the last premise and obtain:

$$E'\{\mathtt{x} = v_2\} \models e_{t3} : \tau_2 \Longrightarrow v \in \tau_2$$

for some value v. This ensures the evaluation of the application itself:

$$E \models e_{t1}\ e_{t2} \Longrightarrow v \in \tau_2$$

If the last evaluation rule is (e-app2), e_{t1} is evaluated to a recursive closure, but the outline of the proof is as same as non-recursive version.

□

Theorem 4 is now just a special case of Theorem 5, when the both typing and evaluation environments are empty sets.

2.5.6 Type soundness of generic primitives

In Section 2.5.5, we have assumed that a generic primitive g_σ has a well-defined semantic value for *any* instance type τ of σ. But it is not actually true, since τ may contain free type variables. Let $\sigma = \forall\delta.\delta$ and $\tau = \delta'$ for instance. Then, $gen(\mathsf{g}, \delta')$ must return a semantic value of δ' which has no actual semantic value in practice. Without semantic values of δ', it is impossible to evaluate $\mathsf{g} : \delta'$ and the type soundness is broken.

To overcome this undesirable situation we can take two possible approaches: the first is to ignore the problem at static typing phase and leave such instances without semantics raise run-time exceptions. When the primitive function $gen(\mathsf{g}, \tau)$ finds that the type τ has no actual semantic value, it is responsibility of the primitive to raise a run-time exception which can be caught by an exception handler.

Another solution is to weaken the assumption of gen. We now assume that $gen(\mathsf{g}, \tau)$ returns a semantic value of τ only when τ is closed. The rule (s-gen) is rewritten as follows:

$$\textbf{(s-gen)}\ \frac{\tau \leq \sigma \quad FV(\tau) = \emptyset}{gen(\mathsf{g}_\sigma, \tau) \in \tau}$$

If any basic constant type contains at least one semantic value, we can construct a well-typed semantic value of any closed type τ. Therefore $gen(\mathsf{g}, \tau)$ can always return some semantic value of any closed type τ[2]. However, this weakened assumption requires a new restriction to generic primitives: for the type soundness, a generic primitive must not be evaluated under an evaluation type context with type variables. In type system, this is reflected to a type instantiation restriction of generic values: type annotation of generic instances must not contain free dynamic type variables. For example, the expression

[2] *gen* still needs to raise a run-time exception when it is asked to create values of abstract data types whose definitions are unknown to *gen*.

```
(g∀δ.δ : unit → δ') ();;
```

must be ill-typed, since the type of g contains a free dynamic type variable δ'. On the other hand, to define a derived generic value, the same expression must be well-typed when the dynamic type variable δ' is abstracted by some generalization as below:

```
let derived{δ'} () = (g∀δ.δ : unit → δ') () in ...
```

As generic primitives, derived generic values must be under this type restriction, too. Unfortunately, the core type system as a simple extension of Damas-Milner's is not powerful enough to distinguish free and generalized type variables. Therefore, in this chapter, we keep using the original assumption of *gen* that $gen(\mathtt{g}, \tau)$ always returns some semantic value of τ. We will handle this new typing restriction later when we build *generic constraint typing system* in Chapter 8, that detects illegal uses of generic instances, which is independent of the core typing. In the generic constraint typing system, a type judgment $\Gamma, V \Vdash e_t$ carries a set of *generalized type variables*. Type variables in V are considered to be generalized in following typing derivation, and the free occurrences of dynamic type variables in e_t are restricted within those of V.

The existence of free dynamic type variables in typing also causes *coherence problem* of generic primitives. This will be discussed in Section 3.6 in the chapter of the compilation.

2.6 Type reconstruction

Since the semantics of GML cannot be defined without type annotations, to evaluate GML programs, we require that they must be fully type-annotated. But writing programs with fully type annotations by hand as explicit type constraints is a boring job. We definitely need to have a type reconstruction which attaches type annotations to bare GML expressions which are correct in respect with the typing rules, like other relatives of the ML family.

We here represent a type reconstruction algorithm for GML so that we can write programs without type annotations. The algorithm is based on one of Milner-Hindley's for ML. It will verify the correctness of expressions and return their fully type-annotated versions, if they have some typings. The returned typings are always "principal", which are more general than the other possible typings. If they have no possible typing at all, the algorithm will detect it and report typing errors. By using our type reconstruction algorithm, we do not need to write explicit type constraints in almost cases.

2.6.1 Instantiation

Algorithm *inst* creates an instance of a type scheme, whose quantified type variables are all instantiated by different fresh variables. Distinction of regular and dynamic type variables are important here also. *inst* must return an admissible substitution: a regular quantified type variable must be instantiated to a regular variable and a dynamic variable must be instantiated to a dynamic variable.

$inst(\sigma) =$
  let $\forall\nu_1 \dots \nu_n.\tau = \sigma$
  let ν'_i be a regular or dynamic fresh variable according to ν_i
  let $S = \{\nu_i \Rightarrow \nu'_i\}$
  in $S(\tau)$

Clearly $inst(\sigma)$ is one of the most general instances of σ:

Theorem 6 (Generality of *inst***)** *Let $\tau' = inst(\sigma)$. For any instance τ'' of σ, there exists a substitution S such that $S(\tau') = \tau''$.*

PROOF: Let $\sigma = \forall\nu_1 \dots \nu_n.\tau$. Then $inst(\sigma) = \{\nu_i \Rightarrow \nu'_i\}$ for some fresh variables ν'_i. Suppose $\tau'' \leq_S \sigma$. Then

$$(S \circ \{\nu'_i \Rightarrow \nu_i\})(\tau') = (S \circ \{\nu'_i \Rightarrow \nu_i\} \circ \{\nu_i \Rightarrow \nu'_i\})(\tau) = S(\tau) = \tau''$$

□

Lemma 10 (Substitution lemma of *inst***)** *If $inst(\sigma) = \tau$, then for any substitution S where $BV(\sigma)$ is out of the reach of S, $inst(S(\sigma)) = S(\tau)$.*

PROOF: Let $\sigma = \forall\nu_1 \dots \nu_n.\tau_0$ and $\{\nu_i \Rightarrow \nu'_i\} = inst(\sigma)$. Since $\{\nu_i\}$ is out of S, $S(\sigma) = \forall\nu_1 \dots \nu_n.S(\tau_0)$. We can suppose that ν'_i are selected so that they are also out of the reach of S. $\{\nu_i \Rightarrow \nu'_i\} \circ S = \{\nu_i \Rightarrow \nu'_i\} + S = S \circ \{\nu_i \Rightarrow \nu'_i\}$. From it, $inst(S(\sigma)) \overset{\alpha}{=} \{\nu_i \Rightarrow \nu'_i\}(S(\tau_0)) = S(\{\nu_i \Rightarrow \nu'_i\}(\tau_0)) = S(\tau)$ □

2.6.2 Unification

If for two types τ_1 and τ_2, there exists a substitution U such that $U(\tau_1) = U(\tau_2)$, then we say the types are *unifiable* and call U a *unifier* of τ_1 and τ_2. Our type unification algorithm seeks the *most general unifier* of given two types, if it exists. The returned unifiers must be admissible: they map dynamic type variables to type expressions without regular type variables. If such U exists, we say τ_1 and τ_2 are *unifiable*. When we unify a type expression τ with a dynamic type variable δ, we must *dynamize* the expression in order to have admissible unifiers. The dynamization algorithm $dynamize(\tau)$ returns a substitution which replaces all the regular type variables α_i in τ to dynamic type variables δ_i keeping their distinction:

$$\begin{array}{rcl}
dynamize(base) &=& \emptyset \\
dynamize(\alpha) &=& \{\alpha \Rightarrow \delta\} \text{ where } \delta \text{ is a fresh dynamic type variable} \\
dynamize(\delta) &=& \emptyset \\
dynamize(\tau_1 \to \tau_2) &=& \text{let } D_1 = dynamize(\tau_1),\ D_2 = dynamize(D_1(\tau_1)) \text{ in } D_2 \circ D_1 \\
dynamize(\tau_1 * \tau_2) &=& \text{let } D_1 = dynamize(\tau_1),\ D_2 = dynamize(D_1(\tau_1)) \text{ in } D_2 \circ D_1
\end{array}$$

For example,

$$dynamize(\alpha * \delta_0 \to \beta) = \{\alpha{\Rightarrow}\delta_1, \beta{\Rightarrow}\delta_2\}$$

and applying this to the type $\alpha * \delta_0 \to \beta$, we get its dynamized type expression:

$$\{\alpha{\Rightarrow}\delta_1, \beta{\Rightarrow}\delta_2\}\,(\alpha * \delta_0 \to \beta) = \delta_1 * \delta_0 \to \delta_2$$

With *dynamize* we can define the algorithm *unify* to create the most general unifier of two types. This is a variant of Robinson's algorithm [Rob65] plus the dynamization for unification between a dynamic type variable and a type expression:

$$\begin{array}{rcl} unify(\tau,\tau) & = & \emptyset \\ unify(\alpha,\tau) & = & \{\alpha{\Rightarrow}\tau\} \text{ when } \alpha \notin FV(\tau) \\ unify(\delta,\tau) & = & D \circ \{\delta{\Rightarrow}D(\tau)\} \text{ when } \delta \notin FV(\tau), \text{ where } D = dynamize(\tau) \\ unify(\tau,\nu) & = & unify(\nu,\tau) \\ \multicolumn{3}{l}{unify(\tau_1 \to \tau_2, \tau_1' \to \tau_2') =} \\ & & \text{let } U_1 = unify(\tau_1,\tau_1'),\ U_2 = unify(U_1(\tau_2),U_1(\tau_2')) \text{ in } U_2 \circ U_1 \\ \multicolumn{3}{l}{unify(\tau_1 * \tau_2, \tau_1' * \tau_2') =} \\ & & \text{let } U_1 = unify(\tau_1,\tau_1'),\ U_2 = unify(U_1(\tau_2),U_1(\tau_2')) \text{ in } U_2 \circ U_1 \\ \multicolumn{3}{l}{unify(\tau_1,\tau_2) \text{ fails if } \tau_1 \text{ and } \tau_2 \text{ do not match with one of these above}} \end{array}$$

Theorem 7 *If $unify(\tau_1, \tau_2)$ succeeds, it returns an admissible substitution.*

Proof: It is clear from the definition of *dynamize*, that for any type τ, the domain of $dynamize(\tau)$ contains no dynamic type variable. Therefore $dynamize(\tau)$ always returns an admissible substitution.

In the algorithm *unify*, the third rule $unify(\delta, \tau) = \ldots$ is the only case which introduce a substitution whose domain contains dynamic type variable. $D \circ \{\delta{\Rightarrow}D(\tau)\} = D \cup \{\delta{\Rightarrow}D(\tau)\}$, because $Dom(D)$ has only regular type variables while $D(\tau)$ contains no regular type variable. Because δ is the only dynamic type variable of $Dom(unify(\delta, \tau))$ and it maps δ to $D(\tau)$ which has no regular type variable, it returns an admissible substitution.

From Theorem 2, admissibility is kept in composition of substitutions, the algorithm *unify* always returns an admissible substitution, if it succeeds. □

Theorem 8 (Unification algorithm returns MGU) *The algorithm $unify(\tau_1, \tau_2)$ returns one of their most general unifier, if they are unifiable. If they are unifiable with a unifier S, i.e. $S(\tau_1) = S(\tau_2)$, the algorithm $unify(\tau_1, \tau_2)$ returns a substitution U and $S = S' \circ U$ for some S'.*

Otherwise, in the case they are not unifiable, $unify(\tau_1, \tau_2)$ fails.

PROOF: The proof is by induction over the structure of the argument type expressions τ_1, τ_2. It is almost same as proofs of ordinary unification algorithms. What we should note here is the case for the unification between a dynamic type variable and an expression, $\mathit{unify}(\delta, \tau)$.

If $\delta \in FV(\tau)$, then $U(\delta, \tau)$ always returns $U = D \circ \{\delta \Rightarrow D(\tau)\}$ where $D = \mathit{dynamize}(\tau)$, and actually by the same discussion in the previous proof, we have:

$$U = D + \{\delta \Rightarrow D(\tau)\}$$

and we can easily verify that it actually unifies them, $U(\delta) = U(\tau) = D(\tau)$.

Suppose that there is another unifier S such that $S(\delta) = S(\tau)$. Let

$$\begin{aligned} S_1 &= \{(\nu_i \Rightarrow \tau_i) \in S \mid \nu_i \text{ is neither } \delta,\ \alpha_i \text{ nor } \delta_i, \text{ where } (\alpha_i \Rightarrow \delta_i) \in D\} \\ S_2 &= \{(\delta_i \Rightarrow S(\alpha_i) \mid \alpha_i \in \mathit{Dom}(D)\} \end{aligned}$$

$\mathit{Dom}(S_1)$ and $\mathit{Dom}(S_2)$ are distinct, therefore $S' = S_1 + S_2$ is well defined substitution. By using S', S can be written $S = S' \circ U$, because if we let $(\alpha_i \Rightarrow \delta_i) \in D$, then the related type variables δ and α_i in τ are mapped as follows:

$$\begin{aligned} (S' \circ U)(\alpha_i) &= S'(\delta_i) = S_2(\delta_i) = S(\alpha_i) \\ (S' \circ U)(\delta) &= S'(D(\tau)) = (S_2 \circ D)(\tau) = S(\tau) = S(\delta) \end{aligned}$$

If τ is not δ but contains the type variable δ, δ and τ are not unifiable. To prove it, we define some appropriate size of types and suppose there is some unifier S, then compare the size of $S(\tau)$ and $S(\delta)$. Because $\tau \neq \delta$, they cannot be equal and leads a contradiction. In such case, *unify* fails. □

2.6.3 Type Reconstruction algorithm

The type reconstruction algorithm W is in Figure 2.3. $W(\Gamma, e)$ receives a type environment Γ and a bare GML expression e and usually returns a fully type-annotated expression e_t and a substitution. It fails and reports a type error, if its one of queries of identifiers `x` in Γ, $\Gamma(\mathtt{x})$ is not defined, or one of the internal unifications fails.

Since all the things related with our extensional polymorphism are hidden in the algorithms *inst* and *unify*, the reconstruction algorithm itself looks very similar to the ordinary ML type reconstruction algorithms by Damas and Milner[DM82] and its variants.

Now we prove the correctness and completeness of the algorithm:

Theorem 9 (Type reconstruction correctness) *The algorithm W is correct with respect to the typing rules. Let Γ be a typing environment and e be a bare GML expression and if $W(\Gamma, e) = e_t,\ S$, then $S(\Gamma) \vdash e_t$.*

PROOF: By induction over the structure of bare GML expression.

Constants Because $\mathit{inst}(\sigma) \leq \sigma$, by using (t-cnst), we have $\Gamma \vdash c_\sigma : \mathit{inst}(\sigma)$.

$$
\begin{array}{rcl}
W(\Gamma,\ c_\sigma) & = & c_\sigma : \mathit{inst}(\sigma),\ \emptyset \\
W(\Gamma,\ \mathtt{g}_\sigma) & = & \mathtt{g}_\sigma : \mathit{inst}(\sigma),\ \emptyset \\
W(\Gamma,\ \mathtt{x}) & = & \mathtt{x}_{\Gamma(\mathtt{x})} : \mathit{inst}(\Gamma(\mathtt{x})),\ \emptyset \\
W(\Gamma,\ \mathtt{fun\ x} \to e) & = & \text{let } \alpha \text{ be a fresh variable} \\
 & & \text{let } e_t : \tau,\ S = W(\Gamma\{\mathtt{x} : \alpha\},\ e) \\
 & & \text{in } \mathtt{fun\ x} \to e_t : S(\alpha) \to \tau,\ S \\
W(\Gamma,\ e_1\ e_2) & = & \text{let } e_{t1} : \tau_1,\ S_1 = W(\Gamma,\ e_1) \\
 & & \text{let } e_{t2} : \tau_2,\ S_2 = W(S_1(\Gamma),\ e_2) \\
 & & \text{let } \alpha \text{ be a fresh variable} \\
 & & \text{let } S = \mathit{unify}(S_2(\tau_1), \tau_2 \to \alpha) \\
 & & \text{in } (S \circ S_2)(e_{t1})\ S(e_{t2}) : S(\alpha),\ S \circ S_2 \circ S_1 \\
W(\Gamma, (e_1 \mathtt{,} e_2)) & = & \text{let } e_{t1} : \tau_1,\ S_1 = W(\Gamma,\ e_1) \\
 & & \text{let } e_{t2} : \tau_2,\ S_2 = W(S_1(\Gamma),\ e_2) \\
 & & \text{in } S_2(e_{t1})\mathtt{,}e_{t2} : S_2(\tau_1) * \tau_2,\ S_2 \circ S_1 \\
W(\Gamma, \mathtt{let\ x} \mathtt{=} e_1\ \mathtt{in}\ e_2) & = & \text{let } e_{t1} : \tau_1,\ S_1 = W(\Gamma,\ e_1) \\
 & & \text{let } \sigma = \mathit{Gen}(S_1(\Gamma), (e_{t1} : \tau_1)) \\
 & & \text{let } e_{t2} : \tau_2,\ S_2 = W(S_1(\Gamma)\{\mathtt{x} : \sigma\},\ e_2) \\
 & & \text{in } \mathtt{let\ x}_{BV(\sigma)} \mathtt{=} S_2(e_{t1})\ \mathtt{in}\ e_{t2} : \tau_2,\ S_2 \circ S_1
\end{array}
$$

$$
\begin{array}{l}
W(\Gamma, \mathtt{let\ rec\ f} \mathtt{=} \mathtt{fun\ x} \to e_1\ \mathtt{in}\ e_2) = \\
\qquad \text{let } \alpha \text{ be a fresh variable} \\
\qquad \text{let } (\mathtt{fun\ x} \to e_{t1}) : \tau_1,\ S_1 = W(\Gamma\{\mathtt{f} : \alpha\},\ \mathtt{fun\ x} \to e_1) \\
\qquad \text{let } S'_1 = \mathit{unify}(\tau_1, S_1(\alpha)) \\
\qquad \text{let } \sigma = \mathit{Gen}((S'_1 \circ S_1)(\Gamma), S'_1(\mathtt{fun\ x} \to e_{t1} : \tau_1)) \\
\qquad \text{let } e_{t2} : \tau_2,\ S_2 = W(((S'_1 \circ S_1)(\Gamma))\{\mathtt{f} : \sigma\},\ e_2) \\
\qquad \text{in} \\
\qquad\quad \mathtt{let\ rec\ f}_{BV(\sigma)} \mathtt{=} \\
\qquad\qquad \{\mathtt{f}_{S_2 \circ S'_1(\tau_1)} \Rightarrow \mathtt{f}_{S_2(\sigma)}\}((S_2 \circ S'_1)(\mathtt{fun\ x} \to e_{t1})) \\
\qquad\quad \mathtt{in}\ e_{t2} : \tau_2,\ S_2 \circ S'_1 \circ S_1
\end{array}
$$

Figure 2.3: Type reconstruction algorithm

Generic primitives Similar to constants, using (t-gen) instead.

Variables Let $\sigma = \Gamma(\mathtt{x})$. Because $\mathit{inst}(\sigma) \leq \sigma$, using (t-inst) we have $\Gamma \vdash \mathtt{x}_\sigma : \mathit{inst}(\sigma)$.

Functions Let

$$\begin{array}{l} W(\Gamma, \mathtt{fun}\ \mathtt{x} \rightarrow e) = \mathtt{fun}\ \mathtt{x} \rightarrow (e_t : \tau) : S(\alpha) \rightarrow \tau,\ S \\ \text{where } e_t : \tau,\ S = W(\Gamma\{\mathtt{x} : \alpha\},\ e) \end{array}$$

By using the induction hypothesis for the reconstruction of e, we have

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

and it equals to $S(\Gamma)\{\mathtt{x} : S(\alpha)\} \vdash e_t : \tau$. Using (t-fun) against this judgment, we obtain

$$S(\Gamma) \vdash \mathtt{fun}\ \mathtt{x} \rightarrow (e_t : \tau) : S(\alpha) \rightarrow \tau$$

Applications Suppose that we have the following:

$$\begin{array}{rcl} W(\Gamma,\ e_1\ e_2) & = & \text{let } e_{t1} : \tau_1,\ S_1 = W(\Gamma,\ e_1) \\ & & \text{let } e_{t2} : \tau_2,\ S_2 = W(S_1(\Gamma),\ e_2) \\ & & \text{let } \alpha \text{ be a fresh variable} \\ & & \text{let } S = \mathit{unify}(S_2(\tau_1), \tau_2 \rightarrow \alpha) \\ & & \text{in } (S \circ S_2)(e_{t1})\ S(e_{t2}) : S(\alpha),\ S \circ S_2 \circ S_1 \end{array}$$

By using the induction hypothesis for the reconstruction of e_1 and e_2, we have:

$$\begin{array}{l} S_1(\Gamma) \vdash e_{t1} : \tau_1 \\ S_2(S_1(\Gamma)) \vdash e_{t2} : \tau_2 \end{array}$$

Using Theorem 3 against the first judgment with $S \circ S_2$ and the second with S, we obtain:

$$\begin{array}{l} (S \circ S_2)(S_1(\Gamma)) \vdash (S \circ S_2)(e_{t1} : \tau_1) \\ S(S_2(S_1(\Gamma))) \vdash S(e_{t2} : \tau_2) \end{array}$$

Because $(S \circ S_2)(\tau_1) = S(\tau_2 \rightarrow \alpha)$ we can use (t-app):

$$(S \circ S_2 \circ S_1)(\Gamma) \vdash (S \circ S_2)(e_{t1})\ S(e_{t2}) : S(\alpha)$$

Tuples It is proven like the applications and simpler than it because of the lack of the unification.

Recursive let definitions We have

$$
\begin{array}{l}
W(\Gamma, \texttt{let rec f = fun x} \rightarrow e_1 \texttt{ in } e_2) = \\
\quad \text{let } \alpha \text{ be a fresh variable} \\
\quad \text{let } (\texttt{fun x} \rightarrow e_{t1}) : \tau_1,\ S_1 = W(\Gamma\{\texttt{f} : \alpha\},\ \texttt{fun x} \rightarrow e_1) \\
\quad \text{let } S_1' = \mathit{unify}(\tau_1, S_1(\alpha)) \\
\quad \text{let } \sigma = \mathit{Gen}((S_1' \circ S_1)(\Gamma), S_1'(\texttt{fun x} \rightarrow e_{t1} : \tau_1)) \\
\quad \text{let } e_{t2} : \tau_2,\ S_2 = W(((S_1' \circ S_1)(\Gamma))\{\texttt{f} : \sigma\},\ e_2) \\
\quad \text{in } \ \texttt{let rec f}_{BV(\sigma)} \texttt{=} \{\texttt{f}_{S_2 \circ S_1'(\tau_1)} \Rightarrow \texttt{f}_{S_2(\sigma)}\}((S_2 \circ S_1')(\texttt{fun x} \rightarrow e_{t1})) \\
\quad \phantom{\text{in }} \ \texttt{in } e_{t2} : \tau_2,\ S_2 \circ S_1' \circ S_1
\end{array}
$$

Using the induction hypothesis for the reconstruction of $\texttt{fun x} \rightarrow e_1$

$$S_1(\Gamma\{\texttt{f} : \alpha\}) \vdash \texttt{fun x} \rightarrow e_{t1} : \tau_1$$

By applying Theorem 3 with $S_2 \circ S_1'$,

$$(S_2 \circ S_1' \circ S_1)(\Gamma\{\texttt{f} : \alpha\}) \vdash (S_2 \circ S_1')(\texttt{fun x} \rightarrow e_{t1} : \tau_1)$$

On the other hand, using the induction hypothesis for the reconstruction of e_2,

$$S_2((S_1' \circ S_1)(\Gamma))\{\texttt{f} : S_2(\sigma)\} \vdash e_{t2} : \tau_2$$

From the definition of $\sigma = \mathit{Gen}((S_1' \circ S_1)(\Gamma), S_1'(\texttt{fun x} \rightarrow e_{t1} : \tau_1))$, by the same argument done in the `let` part of the proof of Theorem 3,

$$
\begin{array}{rcl}
S_2(\sigma) & = & S_2(\mathit{Gen}((S_1' \circ S_1)(\Gamma), S_1'(\texttt{fun x} \rightarrow e_{t1} : \tau_1))) \\
& = & \mathit{Gen}((S_2 \circ S_1' \circ S_1)(\Gamma), (S_2 \circ S_1')(\texttt{fun x} \rightarrow e_{t1} : \tau_1))
\end{array}
$$

These two judgment and the generalization, we can construct the goal using (t-letrec):

$$
\begin{array}{ll}
(S_2 \circ S_1' \circ S_1)(\Gamma) \vdash & \texttt{let rec f}_{BV(\sigma)} \texttt{=} \{\texttt{f}_{S_2 \circ S_1'(\tau_1)} \Rightarrow \texttt{f}_{S_2(\sigma)}\}(\texttt{fun x} \rightarrow (S_2 \circ S_1')(e_{t1})) \\
& \texttt{in } e_{t2} : \tau_2
\end{array}
$$

Let definitions Similar to the recursive lets, and easier than them.

□

The algorithm W is not only correct, but also *complete*. If given expression has some typing, it always returns the most general *principal type*:

Theorem 10 (Type reconstruction completeness) *The algorithm W is complete. Let Γ be a typing environment and e_t' be a GML expression. If there exists a typing $S'(\Gamma) \vdash e_t' : \tau'$ where $\mathit{strip}(e_t') = e$ and $\mathit{Dom}(S') \subset FV(\Gamma)$, then $W(\Gamma, e) = (e_t : \tau),\ S$ and there exists a substitution S'' such that*

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

In the latter statement, $\mathit{fresh}(W(\ldots))$ denotes the set of newly introduced type variables in the algorithm W.

PROOF: Like the proof of the previous theorem, by induction over the structure of the bare GML expression.

Variables Let $e'_t = \mathtt{x} : \tau'$. Because $S'(\Gamma) \vdash \mathtt{x} : \tau'$, we have a binding of $\mathtt{x}$ in Γ. We let $\Gamma(\mathtt{x}) = \sigma$. Then from the typing, $\tau' \leq S'(\sigma)$.

Since Γ has a binding for $\mathtt{x}$, the reconstruction does not fail:

$$W(\Gamma, \mathit{strip}(e'_t)) = W(\Gamma, \mathtt{x}) = \mathtt{x} : \mathit{inst}(\sigma),\ \emptyset$$

If we let $\tau = \mathit{inst}(\sigma)$, $e_t = \mathtt{x} : \tau$ and $S = \emptyset$. Let $\sigma = \forall \nu_1 \ldots \nu_n . \tau_0$. By renaming the bound variables, we can suppose that $\{\nu_i\}$ are out of the reach of S' and $\nu_i \notin FV(\Gamma) \cup FV(\tau')$. Then there exists fresh type variables ν'_i and τ is written as:

$$\tau = \{\nu_i \Rightarrow \nu'_i\}(\tau_0)$$

Therefore $\tau_0 = \{\nu'_i \Rightarrow \nu_i\}(\tau)$. On the other hand, since $S'(\sigma) = \forall \nu_1 \ldots \nu_n . S'(\tau_0)$, τ' can be written as

$$\tau' = (\{\nu_i \Rightarrow \tau_i\} \circ S')(\tau_0)$$

for some types $\tau_1, \ldots, \tau_n$. From these,

$$\tau' = (\{\nu_i \Rightarrow \tau_i\} \circ S' \circ \{\nu'_i \Rightarrow \nu_i\})(\tau)$$

Since $\{\nu_i\}$ and $\{\nu'_i\}$ are both out of the reach of S', this becomes

$$\tau' = (\{\nu_i \Rightarrow \tau_i\} \circ \{\nu'_i \Rightarrow \nu_i\} \circ S')(\tau) = (\{\nu'_i \Rightarrow \tau_i\} \circ S')(\tau)$$

Let $S'' = \{\nu'_i \Rightarrow \tau_i\} \circ S'$, then

$$S''(e_t) = \mathtt{x} : S''(\tau) = \mathtt{x} : \tau' = e'_t$$

For any type variable ν except ν'_i,

$$S'' \circ S(\nu) = S''(\nu) = \{\nu'_i \Rightarrow \tau_i\}(S'(\nu)) = S'(\nu)$$

since $S = \emptyset$ and ν'_i are fresh.

Constants, Generic primitives They are proven more simply than the cases for variables.

Functions The typing $S'(\Gamma) \vdash \texttt{fun x} \rightarrow e_1 : \tau'$. Then the typing derivation must end like:

$$\frac{S'(\Gamma)\{\texttt{x} = \tau'_1\} \vdash e'_{t1} : \tau'_2}{S'(\Gamma) \vdash \texttt{fun x} \rightarrow e'_{t1} : \tau'_1 \rightarrow \tau'_2}$$

where $\tau' = \tau'_1 \rightarrow \tau'_2$ and $\mathit{strip}(e'_{t1}) = e_1$.

Let α be a fresh variable, $\Gamma_1 = \Gamma\{\texttt{x} = \alpha\}$ and $S'_1 = S'\{\alpha \Rightarrow \tau'_1\}$, then $S'_1(\Gamma_1) = S'(\Gamma)\{\texttt{x} = \tau'_1\}$. Therefore the premise is rewritten as $S'_1(\Gamma_1) \vdash e'_{t1} : \tau'_2$. Applying the induction hypothesis to this typing, we obtain

$$W(\Gamma_1, e_1) = e_{t1} : \tau_2,\ S_1$$

and there is a substitution S''_1 such that

$$S''_1(e_{t1} : \tau_2) = e'_{t1} : \tau'_2, \quad S'_1(\nu) = S''_1 \circ S_1(\nu) \text{ for any } \nu \notin \mathit{fresh}(W(\Gamma_1, e_1))$$

The latter part is rewritten into the followings:

$$\begin{array}{l} S'_1(\alpha) = \tau'_1 = (S''_1 \circ S_1)(\alpha) \\ \forall \nu \notin \mathit{fresh}(W(\Gamma_1, e_1)) \cup \{\alpha\}, \quad S'_1(\nu) = S'(\nu) = (S''_1 \circ S_1)(\nu) \end{array}$$

Because $W(\Gamma_1, e_1)$ is well-defined, $W(\Gamma, \texttt{fun x} \rightarrow e_1)$ is also well-defined:

$$W(\Gamma, \texttt{fun x} \rightarrow e_1) = \texttt{fun x} \rightarrow e_t :\ S_1(\alpha) \rightarrow \tau_2,\ S_1$$

The fresh variables introduced by $W(\Gamma, \texttt{fun x} \rightarrow e_1)$ are α and those of $W(\Gamma_1, e_1)$, therefore, S''_1 has the following properties, because $\mathit{fresh}(W(\Gamma, \texttt{fun x} \rightarrow e_1)) = \mathit{fresh}(W(\Gamma_1, e_1)) \setminus \{\alpha\}$.

$$\begin{array}{l} S''_1(S_1(\alpha)) \rightarrow \tau'_2 = \tau'_1 \rightarrow \tau'_2 \\ \forall \nu \notin \mathit{fresh}(W(\Gamma, \texttt{fun x} \rightarrow e_1)), \quad S'_1(\nu) = S'(\nu) = (S''_1 \circ S_1)(\nu) \end{array}$$

Applications Let $e'_t = e'_{t1}\ e'_{t2}$. The typing ends like

$$\frac{S'(\Gamma) \vdash e'_{t1} : \tau'_1 \rightarrow \tau'_2 \qquad S'(\Gamma) \vdash e'_{t2} : \tau'_1}{S'(\Gamma) \vdash e'_{t1}\ e'_{t2} : \tau'_2}$$

where $\mathit{strip}(e'_{ti}) = e_i$.

Applying the induction hypothesis to e'_{t1}, we have

$$W(\Gamma, \mathit{strip}(e'_{t1})) = e_{t1} : \tau_1,\ S_1$$

and there exists a substitution S''_1 such that

$$\begin{array}{l} S''_1(e_{t1} : \tau_1) = e'_{t1} : \tau'_1 \rightarrow \tau'_2 \\ \forall \nu \notin \mathit{fresh}(W(\Gamma, \mathit{strip}(e'_{t1}))), \quad S'(\nu) = S''_1 \circ S_1(\nu) \end{array}$$

$S'(\Gamma) = S_1''(S_1(\Gamma))$, since Γ never contains the fresh type variables introduced by $W(\Gamma, \mathit{strip}(e'_{t1}))$.

Using the induction hypothesis for the typing of e'_{t2}, $S_1''(S_1(\Gamma)) \vdash e'_{t2} : \tau_1'$:

$$W(S_1(\Gamma), \mathit{strip}(e'_{t2})) = e_{t2} : \tau_2,\ S_2$$

and there exists S_2'' such that

$$S_2''(e_{t2} : \tau_2) = e'_{t2} : \tau_1',\ \forall \nu \in \mathit{fresh}(W(S_1(\Gamma), \mathit{strip}(e'_{t2}))),\quad S_1''(\nu) = S_2'' \circ S_2(\nu)$$

Adding a binding $(\alpha \Rightarrow \tau_2')$ of a fresh type variable α to this S_2'', the substitution $S_2''\{\alpha \Rightarrow \tau_2'\}$ unifies $S_2(\tau_1)$ and $\tau_2 \rightarrow \alpha$ as follows, since $FV(\tau_1) \cap \mathit{fresh}(W(S_1(\Gamma), \mathit{strip}(e'_{t2})) = \emptyset$ and α is fresh:

$$\begin{array}{l} S_2''\{\alpha \Rightarrow \tau_2'\}(S_2(\tau_1)) = S_2''(S_2(\tau_1)) = S_1''(\tau_1) = \tau_1' \rightarrow \tau_2' \\ S_2''\{\alpha \Rightarrow \tau_2'\}(\tau_2 \rightarrow \alpha) = S_2''(\tau_2) \rightarrow \tau_2' = \tau_1' \rightarrow \tau_2' \end{array}$$

From the existence of this unifier $S_2''\{\alpha \Rightarrow \tau_2'\}$ assures the existence of the most general unifier of these types $S = \mathit{unify}(S_2(\tau_1), \tau_2 \rightarrow \alpha)$. Therefore $W(\Gamma, e_1\ e_2)$ is well defined:

$$W(\Gamma, e_1\ e_2) = (S \circ S_2)(e_{t1})\ S(e_{t2}) : S(\alpha),\ S \circ S_2 \circ S_1$$

Because S is a more general unifier than $S_2''\{\alpha \Rightarrow \tau_2'\}$, there must be a substitution S'' such that

$$S_2''\{\alpha \Rightarrow \tau_2'\} = S'' \circ S$$

This S'' is what we want:

$$\begin{array}{l} S''(S(\alpha)) = S_2''\{\alpha \Rightarrow \tau_2'\}(\alpha) = \tau_2' \\ \forall \nu \notin \mathit{fresh}(W(\Gamma, e_1\ e_2)) = \mathit{fresh}(W(\Gamma, e_1)) \cup \mathit{fresh}(W(S_1(\Gamma), e_2)) \cup \{\alpha\}, \end{array}$$

$$\begin{array}{rcll} (S'' \circ S \circ S_2 \circ S_1)(\nu) & = & (S_2''\{\alpha \Rightarrow \tau_2'\}) \circ S_2 \circ S_1(\nu) & \text{from } S'' \circ S = S_2''\{\alpha \Rightarrow \tau_2'\} \\ & = & S_2'' \circ S_2 \circ S_1(\nu) & \text{because } \alpha \text{ is fresh} \\ & = & S_1'' \circ S_1(\nu) & \text{from } \nu \notin \mathit{fresh}(W(S_1(\Gamma), e_2)) \\ & = & S'(\nu) & \text{from } \nu \notin \mathit{fresh}(W(\Gamma, e_1)) \end{array}$$

Tuples Like applications, but simpler due to the lack of the unification.

Recursive let The typing is written as $S'(\Gamma) \vdash \texttt{let f = fun x} \rightarrow e'_{t1}\ \texttt{in}\ e'_{t2} : \tau'$ where $\mathit{strip}(e'_{ti}) = e_i$ and its type derivation must end with the following derivation:

$$\frac{\begin{array}{c} S'(\Gamma)\{\texttt{f} : \tau_1'\} \vdash \texttt{fun x} \rightarrow e'_{t1} : \tau_1' \quad \forall V'.\tau_1' = \mathit{Gen}(S'(\Gamma), (\texttt{fun x} \rightarrow e'_{t1}) : \tau_1') \\ S'(\Gamma)\{\texttt{f} : \forall V'.\tau_1'\} \vdash e'_{t2} : \tau_2' \end{array}}{S'(\Gamma) \vdash \texttt{let rec f}_{V'} = \{\texttt{f}_{\tau_1'} \Rightarrow \texttt{f}_{\forall V'.\tau_1'}\}(\texttt{fun x} \rightarrow e'_{t1})\ \texttt{in}\ e'_{t2} : \tau_2'}$$

where $strip(e'_{t1}) = e_1$ and $strip(e'_{t2}) = e_2$. Let α be a fresh variable. Then, like what we did in the part of the functions,

$$S'\{\alpha \Rightarrow \tau'_1\}(\Gamma\{\mathtt{f} : \alpha\}) = S'(\Gamma)\{\mathtt{f} : \tau'_1\}$$

Therefore we can use the induction hypothesis for the reconstruction of $\mathtt{fun\ x} \rightarrow e_1$:

$$W(\Gamma\{\mathtt{f} : \alpha\}, \mathtt{fun\ x} \rightarrow e_1) = (\mathtt{fun\ x} \rightarrow e_{t1}) : \tau_1,\ S_1$$

and there is a substitution S''_1 such that

$$S''_1(\tau_1) = \tau'_1, \quad \forall \nu \notin \mathit{fresh}(W(\Gamma\{\mathtt{f} : \alpha\}, \mathtt{fun\ x} \rightarrow e_1)),\ S'\{\alpha \Rightarrow \tau'_1\}(\nu) = (S''_1 \circ S_1)(\nu)$$

The latter part is rewritten into the followings:

$$\begin{array}{l} S'\{\alpha \Rightarrow \tau'_1\}(\alpha) = \tau'_1 = (S''_1 \circ S_1)(\alpha) \\ \forall \nu \notin \mathit{fresh}(W(\Gamma\{\mathtt{f} : \alpha\}, \mathtt{fun\ x} \rightarrow e)) \cup \{\alpha\},\ S'\{\alpha \Rightarrow \tau'_1\}(\nu) = S'(\nu) = (S''_1 \circ S_1)(\nu) \end{array}$$

Because $S''_1(\tau_1) = \tau'_1 = S''_1(S_1(\alpha))$, S''_1 is a unifier of τ_1 and $S_1(\alpha)$. This implies the existence of the most general unifier S'_1 of them:

$$S'_1 = \mathit{unify}(\tau_1, S_1(\alpha))$$

and there exists a substitution S'''_1 such that $S''_1 = S'''_1 \circ S'_1$.

Let the type scheme obtained from the reconstruction σ. This and the scheme from the derivation are:

$$\begin{array}{rcl} \sigma & = & \forall V.S'_1(\tau_1) \\ & = & \mathit{Gen}((S'_1 \circ S_1)(\Gamma), S'_1((\mathtt{fun\ x} \rightarrow e_{t1}) : \tau_1)) \\ \\ \forall V'.\tau'_1 & = & \mathit{Gen}(S'(\Gamma), (\mathtt{fun\ x} \rightarrow e'_{t1}) : \tau'_1) \\ & = & \mathit{Gen}((S''_1 \circ S_1)(\Gamma), S''_1((\mathtt{fun\ x} \rightarrow e_{t1}) : \tau_1)) \\ & = & \mathit{Gen}((S'''_1 \circ S'_1 \circ S_1)(\Gamma), S'''_1 \circ S'_1((\mathtt{fun\ x} \rightarrow e_{t1}) : \tau_1)) \end{array}$$

From Lemma 5, $\forall V'.\tau'_1 \leq \sigma$.

Using Lemma 6, from the last premise, we obtain a typing $S'(\Gamma)\{\mathtt{f} : \sigma\} \vdash e''_{t2} : \tau'_2$, where $\mathit{strip}(e''_{t2}) = \mathit{strip}(e'_{t2})$.

$$\begin{array}{rcll} S'(\Gamma)\{\mathtt{f} : \sigma\} & = & (S'\{\alpha \Rightarrow \tau'_1\}(\Gamma))\{\mathtt{f} : \sigma\} & \alpha \text{ never appears in } \Gamma \\ & = & ((S''_1 \circ S_1)(\Gamma))\{\mathtt{f} : \sigma\} & \Gamma \text{ contains no fresh variables} \\ & = & ((S'''_1 \circ S'_1 \circ S_1)(\Gamma))\{\mathtt{f} : \sigma\} & \\ & = & ((S'''_1 \circ S'_1 \circ S_1)(\Gamma))\{\mathtt{f} : S'''_1(\sigma)\} & \\ & = & S'''_1((S'_1 \circ S_1)(\Gamma)\{\mathtt{f} : \sigma\}) & \end{array}$$

The typing is now rewritten as follows:

$$S'''_1((S'_1 \circ S_1)(\Gamma)\{\mathtt{f} : \sigma\}) \vdash e''_{t2} : \tau'_2$$

From it we can use the induction hypothesis for the reconstruction of e_2:

$$W((S_1' \circ S_1)(\Gamma)\{\mathtt{f} : \sigma\}, e_2) = e_{t2} : \tau_2,\ S_2$$

and there exists a substitution S_2'' such that

$$S_2''(\tau_2) = \tau_2', \quad \forall \nu \notin \mathit{fresh}(W((S_1' \circ S_1)(\Gamma)\{\mathtt{f} : \sigma\}, e_2)),\ S_1'''(\nu) = (S_2'' \circ S_2)(\nu)$$

The well-foundness of $W((S_1' \circ S_1)(\Gamma)\{\mathtt{f} : \sigma\}, e_2)$ assures that the whole reconstruction $W(\Gamma, \mathtt{let\ rec\ f = fun\ x} \rightarrow e_1\ \mathtt{in}\ e_2)$ is also well defined. S_2'' has also another property, which we want:

$$\begin{array}{lcl}
\multicolumn{3}{l}{\forall \nu \notin \mathit{fresh}(W(\Gamma, \mathtt{let\ rec\ f} = e_1\ \mathtt{in}\ e_2)) =} \\
\multicolumn{3}{l}{\qquad \mathit{fresh}(W(\Gamma\{\mathtt{f} : \alpha\}, \mathtt{fun\ x} \rightarrow e_1)) \cup \mathit{fresh}(W((S_1' \circ S_1)(\Gamma)\{\mathtt{f} : \sigma\}, e_2)) \cup \{\alpha\},} \\
(S_2'' \circ (S_2 \circ S_1' \circ S_1))(\nu) & = & (S_1''' \circ S_1' \circ S_1)(\nu) \\
& = & (S_1'' \circ S_1)(\nu) \\
& = & S'(\nu)
\end{array}$$

Non recursive let Provable more simply than the recursive let bindings.

□

2.7 Conclusion

In this chapter, we formalized the first version of our target language GML, an extension of ML which supports basic mechanisms of extensional polymorphism: generic values, ad-hoc polymorphic values whose semantics are dependent on types at evaluation.

First, we introduced dynamic type variables, a new set of type variables to the type system. Having quantified dynamic type variables in type schemes, generic values are distinguished from the other normal values in the type level. The distinction between generic values and the others in type schemes plays an important role to apply special rules of evaluation and typing to generic values. In this chapter, generic values are provided as primitives or their derived polymorphic values. User construction of generic values will be argued later in Chapter 7.

To achieve the type-dependent evaluation, the semantics of GML are defined as an operation semantics system over fully type-annotated GML expressions. When an instance of a polymorphic value is evaluated, its type instantiation is applied to the corresponding value stored in the evaluation environment, so that it can be evaluated under the correct evaluation type context. Generic values are evaluated to functions which take type contexts and return appropriate semantic values of the corresponding type contexts.

A type system was built to restrict GML programs to well-typed expressions which are never evaluated to run time errors. The type system is also in the ML style, but it is defined over the fully type-annotated expressions, as the semantics system. We also defined a type reconstruction algorithm which type annotates bare GML expressions. This algorithm is complete in respect with the typing rules: if a bare expression is typable, it always returns the principal type annotation. The semantics of a bare expression was defined as those under the corresponding principal typing.

Evaluation of expressions with fully type annotations is more complicated than the classical ML semantics without types. Direct implementation of the GML semantics with type instantiation cannot compete with existing implementations of classical semantics. Therefore in the next chapter, we provide a compilation method of GML expressions to an untyped ML language, whose semantics is type free. The type instantiation of GML evaluation rules is translated to *run time type dispatching*, delivering of values which represent ML types to generic values. Since our compilation performs this run time type dispatching only for generic instances, efficiency of the other non generic part of programs are run time type free, hence it can be as efficiently as the classical ML evaluation.

Chapter 3

Compilation of GML

3.1 Introduction

Compilation with run time types

In normal ML, a type soundness theorem can remove all the type information from evaluation systems, and programs are executed efficiently without run time type handling. In GML evaluation, thanks to its type soundness (Theorem 4) we need not to implement error handling either. But the evaluation system itself cannot be completely free from type manipulations, since semantics of generic instances require their evaluation type context. Such semantics with types is difficult to implement directly and is inefficient. Instead, we here compile GML expressions to an intermediate ML language without types. The type instantiations in the GML semantics are transformed to *run time type dispatching* using type abstractions and applications as the second order typed lambda calculus and system F[GTL90]. The type instantiation information is delivered as *run time type* functional arguments, value representations of type expressions. Generic primitives and their derived generic values are compiled to functions which take these run time types to obtain appropriate evaluation type contexts. Since the run time type dispatching is performed only against generic instances, the other normal non-generic part can be evaluated as efficiently as in normal ML implementations.

Outline of this chapter

We first examine generic related semantic evaluation rules, and argue informally our strategy of compilation in Section 3.2. Then in Section 3.3, we define our intermediate language for compilation, IML, and its semantics. In Section 3.4, we define our compilation algorithm which translates fully type-annotated GML expressions to IML. Finally in Section 3.5, we will check both semantics of GML and IML and and prove our compilation is correct. At the end of this chapter, we argue about the coherence problem of implicitly typed programs.

3.2 Analysis of the generic evaluation

GML evaluation rules for generic values contain type related operations. We here look at them closer again and try to figure out what they really do, and grasp some rough sketch of our compilation strategy:

(e-gen) $E \models \mathtt{g}_\sigma : \tau \Longrightarrow gen(\mathtt{g}, \tau)$

An instance of a generic primitive $\mathtt{g}_\sigma : \tau$ returns a semantic value $gen(\mathtt{g}, \tau)$ which depends on its evaluation type context τ. If $\mathtt{g}$ is used in a derived generic definition, evaluation type contexts of $\mathtt{g}$ are not the type annotation statically obtained at typing itself, but a type expression obtained by applying a substitution which is delivered by the following two evaluation rules:

$$\textbf{(e-g-let)} \quad \frac{V \cap \mathbb{D} \neq \emptyset \quad E\{\mathtt{x} = gclos(E, \_, V, e_{t1})\} \models e_{t2} : \tau \Longrightarrow r}{E \models \mathtt{let}\ \mathtt{x}_V \,\mathtt{=}\, (e_{t1} : \tau_1)\ \mathtt{in}\ e_{t2} : \tau \Longrightarrow r}$$

$$\textbf{(e-gvar)} \quad \frac{\exists S,\ \tau \leq_S \forall V.\tau' \quad E' \models S(e_t : \tau') \Longrightarrow r}{E\{\mathtt{x} = gclos(E', \_, V, e_t : \tau')\} \models \mathtt{x}_{\forall V.\tau'} : \tau \Longrightarrow r}$$

If a derived generic value is defined by (e-g-let) or (e-g-let-r), its definition code e_t is stored as a generic closure $gclos(E, *, V, e_t)$, and it is evaluated when the derived generic value is actually used. When an instance of such a derived generic value is evaluated in a type context τ, (e-gvar) or (e-gvar-r) is used for the evaluation. It applies the instantiation S to the code in the generic closure e_t, then evaluate it, where S indicates how the generic value is instantiated at evaluation. This instantiation application provides evaluation type contexts for generic primitives or derived generic values appear inside e_t.

Clearly, (e-g-let$*$) are only the rules which introduce generic closures into evaluation environment, and they are consumed only by (e-gvar$*$) rules. In (e-gvar$*$), an instantiation S is applied to all the type annotations of the code e_t, so that the inside generic primitives can receive instantiations of their evaluation contexts. However, on the other hand, this type substitution takes no semantical effect on the other non-generic stuffs, since the other evaluation rules than the above rules never use type annotations in any essential way. In addition, since the type schemes of generic primitives have no regular variables, bindings of regular variables in an instantiation S play no role in determining the semantics of generic primitives.

It leads to the following ideas of our compilation.

Non-generic values

At evaluation all the type annotations of non- generic expressions can be ignored since their evaluation is independent of their type annotations. This means that we can use the same compilation method as normal ML language for the non-generic part of our language. This also assures that as far as the non-generic part concerns, compilation will be as efficiently as normal ML compilation without any overhead.

Generic closures

A derived generic value stored as a generic closure in an evaluation environment waits information how its quantified dynamic type variables are actually instantiated for each instance. Then it sends the type instantiation information to the internal generic values in its definition code. This mechanism can be replaced by a function – a closure which takes arguments of instantiation information and send them to the internal compilation of generic instances by function application. For example, if there is a generic closure $gclos(E, \_, V, e_t)$ whose generalized type variable set contain n dynamic type variables, i.e.

$$V \cap \mathbb{D} = \{\delta_1, \dots \delta_n\}$$

then creating a new corresponding identifier $\mathtt{d}_i$ for each δ_i, the generic closure is transformed to a normal closure who takes these $\mathtt{d}_i$:

$$\begin{array}{ccc} \text{GML semantics} & & \text{Compilation semantics} \\ gclos(E, \_, V, e_t) & \Longrightarrow & clos(\mathcal{E}, \_, \mathtt{d}_1, \mathtt{fun}\ \mathtt{d}_2 \ \rightarrow \ \dots \rightarrow \mathtt{fun}\ \mathtt{d}_n \rightarrow e) \end{array}$$

Here, the evaluation environments $\mathcal{E}$ and the code e in the right hand side should be some compilation of E and e_t.

Because the generic closure are introduced only by (e-g-let$*$) rules, this semantic translation can be ported into the syntactic compilation as rewriting of let expressions without much difficulty. The let binding which stores the above generic closure $gclos(E, \_, V, e_t)$ can be written as $\mathtt{let}\ \mathtt{x}_V = (e_t : \tau_1)\ \mathtt{in}\ \dots$ and this GML expression is compiled to the following code, whose semantics is equal to the above semantics compilation of the generic closure:

$$\begin{array}{cccl} \text{GML expression} & & \text{Compiled expression} & \\ \mathtt{let}\ \mathtt{x}_V = (e_t : \tau_1)\ \mathtt{in}\ \dots & \Longrightarrow & \mathtt{let}\ \mathtt{x} = & \mathtt{fun}\ \mathtt{d}_1 \rightarrow \\ & & & \dots \\ & & & \mathtt{fun}\ \mathtt{d}_n \rightarrow e\ \ \mathtt{in}\ \dots \end{array}$$

Generic instances

When a generic instance $\mathtt{x}$ is evaluated, there must be a binding of this identifier to a generic closure $gclos(E, \_, V, e_t)$ inside the current evaluation environment. In our compilation, this generic closure will be translated into a sequence of normal closures as we described above. The evaluation of the generic closure code with its type annotation substitution $S(e_t)$ in GML will be simulated by a sequence of applications of type informations to these newly introduced normal closures:

GML evaluation
$$\frac{\exists S,\ \tau' \leq_S \forall V.\tau \quad E \models S(e_t : \tau') \Longrightarrow r}{E'\{\mathtt{x} = gclos(E, \_, V, e_t : \tau)\} \models \mathtt{x}_{\forall V.\tau} : \tau' \Longrightarrow r}$$

$$\Downarrow$$

Evaluation of compilation
$$\frac{\dfrac{\mathcal{E}\{\mathtt{d}_1 = «S(\delta_1)», \ldots \mathtt{d}_n = «S(\delta_n)»\} \models e \Longrightarrow r}{\ldots}}{\mathcal{E}'\{\mathtt{x} = clos(\mathcal{E}, \_, \mathtt{d}_1, \mathtt{fun}\ \mathtt{d}_2 \to\ \ldots\ \to \mathtt{d}_n \to e)\} \models \mathtt{x}\ «S(\delta_1)»\ \ldots\ «S(\delta_n)» \Longrightarrow r}$$

The application $\mathtt{x}\ «S(\delta_1)»\ \ldots\ «S(\delta_n)»$ binds the variables $\mathtt{d}_i$ by *run time type representations* of $S(\delta_i)$, $«S(\delta_i)»$ and evaluate e, the code in the closure under the updated environment. The run time type is a predefined data type which is isomorphic to the type expressions. Its definition will be given later formally. Each run time type $«S(\delta_i)»$ denotes how the generalized dynamic type variable δ_i in the generalized type variable set V is instantiated by the instantiation S.

We have to be careful about the orders of these run time type arguments and the λ-abstractions for compilations of generic closures so that instantiation information of a generalized dynamic type variable is correctly passed to its corresponding abstraction in compilation. We need to have the uniquely ordered list of quantified type variables in a type scheme $\forall V.\tau$. The following function *Vlist* computes such a list ('@' is the list concatenation operator here):

$$\begin{array}{rcl}
\mathit{Vlist}(V, \mathit{base}) & = & [\,] \\
\mathit{Vlist}(V, \nu) & = & [\nu] \text{ if } \nu \in V \\
 & & [\,] \text{ otherwise} \\
\mathit{Vlist}(V, \tau_1 \to \tau_2) & = & \text{let } l = \mathit{Vlist}(V, \tau_1) \\
 & & \text{let } V' = \{\nu \in V \mid \nu \notin l\} \text{ in} \\
 & & l \mathbin{@} \mathit{Vlist}(V', \tau_2) \\
\mathit{Vlist}(V, \tau_1 * \tau_2) & = & \text{let } l = \mathit{Vlist}(V, \tau_1) \\
 & & \text{let } V' = \{\nu \in V \mid \nu \notin l\} \text{ in} \\
 & & l \mathbin{@} \mathit{Vlist}(V', \tau_2)
\end{array}$$

The function $\mathit{Vlist}(V, \tau)$ lists the type variables in V in the order of their first occurrences in τ. For example,

$$\mathit{Vlist}(\{\alpha, \delta\}, (\delta * \alpha) \to (\delta * \beta)) = [\delta;\ \alpha]$$

In compilation of generic closures and instances, we use this ordering function *Vlist* and obtain the unique ordered list of their generalized dynamic type variables:

$$[\delta_1;\ \ldots;\ \delta_n] = \mathit{Vlist}(V \cap \mathbb{D}, \tau)$$

Abstractions added for compilations of derived let definitions and run time types arguments applied to them in compilations for generic instances must be arranged so that their corresponding type variables are in this order.

Generic primitives

Thus instantiation information of derived generic values are passed into their definition codes as run time types. The final destinations of these informations are generic primitive instances $\mathsf{g}_\sigma : \tau$, where τ is the type annotation approved by type system. When this instance g_σ is evaluated by (g-var), it should be instantiated by some instantiation S and becomes $S(\tau)$. In the compilation, instead of applying this substitution, the evaluation type context will be obtained as a form of function application of run time types, where each run time type denotes how each generalized dynamic type variable of σ is instantiated:

GML evaluation
$$E \models \mathsf{g}_{\forall V.\tau'} : S(\tau) \Longrightarrow gen(\mathsf{g}, S(\tau))$$

$$\Downarrow$$

Evaluation of compilation
$$\frac{[\delta_1; \ldots; \delta_n] = \mathit{Vlist}(V, \tau') \quad \mathcal{E} \models e_1 \Longrightarrow «\tau_1» \ \ldots\ \mathcal{E} \models e_n \Longrightarrow «\tau_n»}{\mathcal{E} \models \mathsf{g}\ e_1\ \ldots\ e_n \Longrightarrow gen'(\mathsf{g}, \{\delta_i \Rightarrow \tau_i\}(\tau')) = gen'(\mathsf{g}, S(\tau))}$$

In the evaluation of the compilation, the instantiation information of each generalized dynamic type variable δ_i of $\forall V.\tau'$ is given by the expression e_i, which is evaluated to a run time type $«\tau_i»$. We will have a special evaluation rule for generic primitives which collects these run time type arguments and rebuild the evaluation type context $S(\tau) = \{\delta_i \Rightarrow \tau_i\}(\tau')$. Instead of the original generic primitive semantic query function $gen(\mathsf{g}, S(\tau))$, we need to use $gen'(\mathsf{g}, S(\tau))$ which returns a semantic value of the compiled language semantics, which is *compatible* with the original. This compatibility of semantic values between two systems will be defined later.

3.3 The intermediate language

We formalize the ideas in the previous section. The compilation of our language is done in two steps. Rather than compiling directly from a GML program to an executable, first we translate fully type-annotated GML expressions into expressions without type annotations and generic stuffs. They are almost a subset of the bare GML expressions, but to make the discussion clearer, we should define it as *an intermediate ML like language* called IML. The grammar of IML is obtained by removing generic primitives and explicit type annotations from the bare GML, adding run time type representation $RT(e_1, \ldots, e_n)$. For each run time type constructor RT has its specific arity n and always takes n arguments, like ordinary ML data constructors.

$$\begin{array}{lll} e ::= & c_\sigma & \\ \mid & RT(e, \ldots, e) & \text{Run time types} \\ \mid & \mathtt{x} & \\ \mid & \mathtt{fun}\ \mathtt{x} \rightarrow e & \\ \mid & e\ e & \\ \mid & e, e & \\ \mid & \mathtt{let}\ \mathtt{x} = e\ \mathtt{in}\ e & \\ \mid & \mathtt{let\ rec}\ \mathtt{x} = e\ \mathtt{in}\ e & \end{array}$$

The obtained IML syntax is quite familiar to the ML programmers and can be executed using any existing compilation technique of ML languages.

3.3.1 Run time types

To pass the evaluation type contexts of generic instances at run time, we need to define a new built-in data type, *run time types*. The run time types are isomorphic to the type expressions of GML defined in Section 2.2.1, except that it has no representation for dynamic type variables. The set of the constructors RT and their definitions are as follows:

```
type rtype =
| RTbase   of rpath                        (* basic types *)
| RTvar    of int                      (* type variables *)
| RTarrow  of rtype * rtype            (* function types *)
| RTtuple  of rtype * rtype                     (* tuples *)
| RTconstr of rpath * rtype list            (* data types *)
```

Each constructor represents the corresponding type expression construct: `RTbase` for basic types, `RTvar` for type variables, `RTarrow` for functional types and `RTtuple` for tuples. Type variables are expressed by their unique integer identifications. user-defined data types such as variants and records are not included in our type system for simplicity, but their representation in run time types can be given as `RTconstr` with their paths and type arguments. Basic and user-defined data types are identified by their *run time path* `rpath`, run time representation of data type identification. At this moment they are equivalent to their names[1].

In run time types, we have only one kind of type variables and do not need special distinction of regular and dynamic type variables, because all the run time types created by the compilation algorithm will represent type expressions without regular type variables.

Translation from type expressions to run time types is straightforward. The run time representation of a type expression τ, $«\tau»$ is defined as follows:

$$\begin{aligned}
«base» &= \texttt{RTbase}\ \text{"}base\text{"} \\
«\delta_i» &= \texttt{RTvar}\ i \\
«\tau_1 \to \tau_2» &= \texttt{RTarrow}(«\tau_1», «\tau_2») \\
«\tau_1 * \tau_2» &= \texttt{RTtuple}(«\tau_1», «\tau_2») \\
«(\tau_1, \ldots,\ \tau_n)\ t» &= \texttt{RTconstr}(\text{"}t\text{"},\ [«\tau_1»; \ldots;\ «\tau_n»])
\end{aligned}$$

Each type variable δ is translated to `RTvar` i, where i is the integer index for δ. This index is uniquely defined for each dynamic type variable. We suppose when we write δ_i, its index is always i. Using this integer index, representations of the same type variables share the same index number of `RTvar`. For example,

$$«(\delta_1 * \delta_2) \to \delta_1» = \texttt{RTarrow(RTtuple(RTvar 1, RTvar 2), RTvar 1)}$$

[1]They will be extended to have the fingerprints of their definitions in Chapter 5, to exchange these run time types between different programs.

If τ contains regular type variables, $«\tau»$ is not defined. However, all the run time types created by our compilation algorithm will be well defined, because all the run time type creations are done for type expressions which have no regular type variables. If τ has no regular type variable occurrence, since this translation is somewhat canonical, we usually use $«\tau»$ notation for run time types in IML for simplicity.

3.3.2 Semantics of IML

The semantics of IML is much easier than one of GML. It is same as normal ML like language therefore straightforward. The semantic values for IML are obtained by adding those for run time types to the values for GML without type annotations in closures:

$$
\begin{array}{lcll}
r & ::= & v & \text{Values} \\
 & | & \mathit{err} & \text{Errors} \\
v & ::= & b_\sigma & \text{Basic constants} \\
 & | & \mathit{RT}(v, \ldots, v) & \text{Run time types} \\
 & | & v, v & \text{Tuples} \\
 & | & \mathit{clos}(\mathcal{E}, \_, \mathtt{x}, e) & \text{Functional closures} \\
 & | & \mathit{clos}(\mathcal{E}, \mathtt{f}, \mathtt{x}, e) & \text{Functional closures (recursive)} \\
\mathcal{E} & ::= & \{\mathtt{x} = v, \ \ldots \ \mathtt{x} = v\} & \text{Environment}
\end{array}
$$

Run time type semantic values use the same constructors *RT* and their arities as those for the IML syntax. We also use the notation $«\tau»$ for the semantic value for a type τ.

An evaluation judgment for IML is written as $\mathcal{E} \models_i e \Longrightarrow v$. The semantic rules are also defined in the same way as GML's, but this time we have no type annotations:

(i-cnst) $\mathcal{E} \models_i c_\sigma \Longrightarrow \mathit{cnst}'(c_\sigma)$

(i-runtype) $$\frac{\mathcal{E} \models_i e_i \Longrightarrow v_i}{\mathcal{E} \models_i \mathit{RT}(e_1, \ldots, e_n) \Longrightarrow \mathit{RT}(v_1, \ldots, v_n)}$$

(i-gen) $$\frac{\sigma = \forall V.\tau \quad [\delta_1; \ldots; \delta_n] = \mathit{Vlist}(V, \tau) \\ \mathcal{E} \models_i e_1 \Longrightarrow «\tau_1» \ \ldots \ \mathcal{E} \models_i e_n \Longrightarrow «\tau_n»}{E \models_i \mathtt{g}_\sigma \ e_1 \ \ldots \ e_n \Longrightarrow \mathit{gen}'(\mathtt{g}, \{\delta_i \Rightarrow \tau_i\}(\tau))}$$

(i-tpl) $$\frac{\mathcal{E} \models_i e_1 \Longrightarrow v_1 \quad \mathcal{E} \models_i e_2 \Longrightarrow v_2}{\mathcal{E} \models_i (e_1, e_2) \Longrightarrow v_1, v_2}$$

(i-var) $\mathcal{E}\{\mathtt{x} = v\} \models_i \mathtt{x} \Longrightarrow v$

(i-fun) $\mathcal{E} \models_i \mathtt{fun}\ \mathtt{x} \rightarrow e \Longrightarrow \mathit{clos}(E, \_, \mathtt{x}, e)$

$$\textbf{(i-let)}\quad \frac{\mathcal{E} \models_i e_1 \Longrightarrow v_1 \quad \mathcal{E}\{\mathtt{x} = v_1\} \models_i e_2 \Longrightarrow v_2}{\mathcal{E} \models_i \mathtt{let}\ \mathtt{x} = e_1\ \mathtt{in}\ e_2 \Longrightarrow v_2}$$

$$\textbf{(i-letrec)}\quad \frac{\mathcal{E}\{\mathtt{x} = clos(\mathcal{E}, \mathtt{f}, \mathtt{x}, e_1)\} \models_i e_2 \Longrightarrow v_2}{\mathcal{E} \models_i \mathtt{let\ rec\ f = fun\ x} \rightarrow e_1\ \mathtt{in}\ e_2 \Longrightarrow v_2}$$

$$\textbf{(i-app1)}\quad \frac{\begin{array}{c}\mathcal{E} \models_i e_2 \Longrightarrow v_2 \quad \mathcal{E} \models_i e_1 \Longrightarrow clos(\mathcal{E}', \_, \mathtt{x}, e) \\ \mathcal{E}'\{\mathtt{x} = v_2\} \models_i e_t \Longrightarrow v\end{array}}{\mathcal{E} \models_i e_1\ e_2 \Longrightarrow v}$$

$$\textbf{(i-app2)}\quad \frac{\begin{array}{c}\mathcal{E} \models_i e_2 \Longrightarrow v_2 \quad \mathcal{E} \models_i e_1 \Longrightarrow clos(\mathcal{E}', \mathtt{f}, \mathtt{x}, e) \\ \mathcal{E}'\{\mathtt{f} = clos(\mathcal{E}', \mathtt{f}, \mathtt{x}, e),\ \mathtt{x} = v_2\} \models_i e \Longrightarrow v\end{array}}{\mathcal{E} \models_i e_1\ e_2 \Longrightarrow v}$$

For constants and generic primitives we use semantic functions $cnst'$ and gen' respectively. Both $cnst'(c_\sigma)$ and $gen'(\mathtt{g}, \tau)$ return the compatible semantic value with the original functions, $cnst(c_\sigma)$ and $gen(\mathtt{g}, \tau)$. They are defined using the following translation function *strip* from GML semantic values for primitives P defined in Section 2.4.1 to semantic values of IML:

$$\begin{array}{l} strip(b_\sigma) = b_\sigma \\ strip(v_1, v_2) = strip(v_1), strip(v_2) \\ strip(clos(\emptyset, y, \mathtt{x}, e_t)) = clos(\emptyset, y, \mathtt{x}, strip(e_t)) \end{array}$$

Using *strip*, definitions of $cnst'$ and gen' are given as follows:

$$\begin{array}{l} cnst'(c_\sigma) = strip(cnst(c_\sigma)) \\ gen'(\mathtt{g}, \tau) = strip(gen(\mathtt{g}, \tau)) \end{array}$$

If there is no applicative rule in the above for an IML expression e under some evaluation environment $\mathcal{E}$, its evaluation returns an error, using the following (i-err):

$$\textbf{(i-err)}\quad \frac{\text{When none of the above rules is applicative.}}{\mathcal{E} \models_i e \Longrightarrow err}$$

This is the only one rule for error. Our compilation algorithm will translate a well-typed fully type-annotated GML expression which never returns an error into an IML expression so that the both can evaluate and return compatible semantic values. This compilation correctness will be proven, therefore we do not need to define an independent error handling rule corresponds with each legal rule intensively, like what we have done for the fully type-annotated GML.

3.4 Compilation algorithm

3.4.1 Compilation of types

In execution of fully type-annotated GML expressions, when a derived generic instance is evaluated and its definition is expanded by (e-gvar) rule, the dynamic type variable annotations which are generalized in the definition are instantiated, so that the internal generic values can be evaluated under appropriate evaluation type contexts. In IML, such generalized dynamic type variables are replaced by identifiers and in evaluation they are substituted by run time types to emulate the instantiations in the GML semantics.

For each generalized dynamic type variable δ_i will be replaced by its special identifier, unique to each dynamic type variable. Because the set of identifiers is infinite, we can prepare such a special identifier for each δ_i which is distinct from the other special identifiers and the normal identifiers used in GML expressions. If we write a dynamic type variable δ_i, we presume that the identifier $\mathtt{d}_i$ is the special identifier for δ_i, and vice versa.

Before we define the compilation algorithm of GML expression to IML, we specify the algorithm for *type compilation* $\lfloor\tau\rfloor_V$, which translates type expressions to their corresponding IML expressions. This is not exactly same as «τ», because we need to replace the dynamic type variables δ_i to their corresponding identifiers $\mathtt{d}_i$ when they are in V, a set of generalized type variables. In the case they are not in V, they are replaced to «δ_i»:

$$\begin{array}{rcl}
\lfloor base\rfloor_V & = & \text{«}base\text{»} \\
\lfloor\delta_i\rfloor_V & = & \mathtt{d}_i \quad \text{if } \delta_i \in V \\
 & & \text{Otherwise, «}\delta_i\text{»} \\
\lfloor\tau_1 \to \tau_2\rfloor_V & = & \mathtt{RTarrow(}\lfloor\tau_1\rfloor_V\mathtt{,}\lfloor\tau_2\rfloor_V\mathtt{)} \\
\lfloor\tau_1 * \tau_2\rfloor_V & = & \mathtt{RTtuple(}\lfloor\tau_1\rfloor_V\mathtt{,}\lfloor\tau_2\rfloor_V\mathtt{)}
\end{array}$$

Type compilation for a regular type variable, $\lfloor\alpha\rfloor_V$ is not defined, but it does not matter since the type compilation algorithm $\lfloor\tau\rfloor_V$ is used against the "dynamized" type expressions which can be written $S(\delta)$ for some admissible substitution S and dynamic type variable δ in the compilation algorithm. Consequently $S(\delta)$ contains no regular type variable.

Run time type representation of type expressions are special cases of this type compilation. The run time type representation of a type expression τ is written as follows:

$$\text{«}\tau\text{»} = \lfloor\tau\rfloor_\emptyset$$

3.4.2 Compilation of expressions

The compilation algorithm $\lfloor e_t \rfloor_V$ takes a full annotated GML expression and a compilation environment, and returns its compilation as an IML expression:

$$
\begin{array}{rcl}
\lfloor c_\sigma : \tau \rfloor_V & = & c_\sigma \\
\lfloor \texttt{fun x} \rightarrow\ e_t \rfloor_V & = & \texttt{fun x}\ \rightarrow \lfloor e_t \rfloor_V \\
\lfloor e_{t1}\ e_{t2} \rfloor_V & = & \lfloor e_{t1} \rfloor_V\ \lfloor e_{t2} \rfloor_V \\
\lfloor e_{t1}, e_{t2} \rfloor_V & = & \lfloor e_{t1} \rfloor_V, \lfloor e_{t2} \rfloor_V \\
\lfloor \texttt{g}_\sigma : \tau \rfloor_V & = & \text{let } \sigma = \forall \delta_1 \ldots \delta_n . \tau' \\
 & & \text{where } \tau \leq_S \sigma \text{ and } [\delta_i] = \mathit{Vlist}(\{\delta_i\}, \tau') \\
 & & \text{in } \texttt{g}\ \lfloor S(\delta_1) \rfloor_V\ \ldots\ \lfloor S(\delta_n) \rfloor_V \\
\lfloor \texttt{x}_\sigma : \tau \rfloor_V & = & \text{let } \sigma = \forall \alpha_1 \ldots \alpha_m \delta_1 \ldots \delta_n . \tau' \\
 & & \text{where } \tau \leq_S \sigma \text{ and } [\delta_i] = \mathit{Vlist}(\{\delta_i\}, \tau') \\
 & & \text{in } \texttt{x}\ \lfloor S(\delta_1) \rfloor_V\ \ldots\ \lfloor S(\delta_n) \rfloor_V \\
\multicolumn{3}{l}{\lfloor \texttt{let (rec) x}_{V'} = (e_{t1} : \tau)\ \texttt{in}\ e_{t2} \rfloor_V =} \\
 & & \text{let } \forall V'.\tau = \forall \alpha_1 \ldots \alpha_m \delta_1 \ldots \delta_n . \tau' \\
 & & \text{where } [\delta_i] = \mathit{Vlist}(\{\delta_i\}, \tau') \\
 & & \texttt{let (rec) x = fun d}_1\ \ldots\ \texttt{d}_n \rightarrow \lfloor e_{t1} \rfloor_{V+V'}\ \texttt{in}\ \lfloor e_{t2} \rfloor_V
\end{array}
$$

As we explained in Section 3.2, generic primitives $\texttt{g}_\sigma : \tau$ and instances of generic values $\texttt{x}_\sigma : \tau$ are compiled to a function application of themselves to run time types for sending instantiation information. Derived generic let definitions are compiled with the abstractions which receive these run time types and their definition expressions must be compiled under the updated generalized type variable set $V + V'$. We can assume that V and V' are distinct by applying appropriate α-conversions of the generalized variable set if necessary. For the other constructs, the compilation just removes all the type annotations from expressions.

Example

Suppose that we have a generic primitive $\texttt{plus}_{\forall \delta.\delta \rightarrow \delta \rightarrow \delta}$. Then the following expression,

```
let double = fun x → plus x x
in double 1.0;;
```

has a typing e_t under the empty typing environment like:

```
let double{δ} = fun x → (plus:δ → δ → δ) x x
in (double∀δ.δ → δ:float → float) 1.0;;
```

and the compilation $\lfloor e_t \rfloor_\emptyset$ is as follows:

```
let double = fun d₁ x → plus d₁ x x
in double «float» 1.0;;
```

We extract some of its evaluation below (the entire evaluation requires too much space). The run time type «`float`» given to `double` will be sent to the generic primitive `plus`, and the expression `plus` $\mathtt{d}_1$ will be evaluated by (i-gen) to $gen'(\mathtt{plus}, \mathtt{float} \to \mathtt{float} \to \mathtt{float})$:

$$\begin{array}{rcl}\{\mathtt{d}_1 = \text{«}\mathtt{float}\text{»}\} \models_i \mathtt{plus}\ \mathtt{d}_1 & \Longrightarrow & gen'(\mathtt{plus}, \mathtt{float} \to \mathtt{float} \to \mathtt{float}) \\ & = & clos(\emptyset, \_, \mathtt{y}_1, \mathtt{fun}\ \mathtt{y}_2 \to \mathtt{y}_1\mathtt{+.}\mathtt{y}_2)\end{array}$$

gen' returns its explicitly defined closure for the plus function for floats, here it is equivalent to one for addition of floats `(+.)`.

Using this closure the entire expression is evaluated to the result of `plus` $\mathtt{d}_1$ `x x`, 2.0.

$$\cfrac{\cfrac{\cfrac{\{\mathtt{x} = 1.0, \mathtt{y}_1 = 1.0\} \models_i (\mathtt{fun}\ \mathtt{y}_2 \to \mathtt{y}_1\mathtt{+.}\mathtt{y}_2)\ \mathtt{x} \Longrightarrow 2.0}{\{\mathtt{d}_1 = \text{«}\mathtt{float}\text{»}, \mathtt{x} = 1.0\} \models_i \mathtt{plus}\ \mathtt{d}_1\ \mathtt{x}\ \mathtt{x} \Longrightarrow 2.0}}{\cdots}}{\{\mathtt{double} = clos(\ldots)\} \models_i \mathtt{double}\ \text{«}\mathtt{float}\text{»}\ 1.0 \Longrightarrow 2.0}$$

□

3.5 Compilation correctness

What finally we argue in this chapter is the correctness of our compilation algorithm. Compiled code in IML must be evaluated to a "compatible" semantic value as the source GML code returns.

3.5.1 Semantic compilation compatibility

To extend the statement in the last paragraph to each step of evaluation, we need to define a relationship called *semantic compatibility* between semantic values for GML and IML. If an IML semantic value v' is semantically compatible to one of GML v, we write $v' \sqsubset v$. The definition of the relationship $\sqsubset$ for those semantic values is as follows in Figure 3.1. This relationship is also naturally extended to a relationship between evaluation environments of both systems. Here, the notation $\mathcal{E}\{S\}$ is for the evaluation environment $\mathcal{E}$ overridden by the evaluation bindings corresponds with S which is obtained by replacing its bindings for the dynamic type variables $(\delta_i \Rightarrow \tau_i)$ to $(\mathtt{d}_i \Rightarrow \text{«}\tau_i\text{»})$. The compatibility rule of normal and generic closures are rather complicated. These rules are intuitively read as follows:

Rule for normal closures There is a subexpression $\mathtt{fun}\ \mathtt{x} \to e_t$ in a program and typed as $\Gamma' \vdash \mathtt{fun}\ \mathtt{x} \to e_t : \tau_1' \to \tau_2'$. In evaluation, e_t is substituted to $S(e_t)$ and the function is evaluated to a closure $clos(E, y, \mathtt{x}, S(e_t))$. Then, the compilation of $\mathtt{fun}\ \mathtt{x} \to e_t$, $\mathtt{fun}\ \mathtt{x} \to \lfloor e_t \rfloor_S$ will be evaluated to some closure $clos(\mathcal{E}\{S\}, y, \mathtt{x}, \lfloor e_t \rfloor_S)$.

$$b_\sigma \sqsubset b_\sigma$$

$$\frac{v_1' \sqsubset v_1 \quad v_2' \sqsubset v_2}{v_1', v_2' \sqsubset v_1, v_2}$$

$$\frac{\begin{array}{c}\mathcal{E} \sqsubset E \quad clos(E, y, \mathtt{x}, S(e_t)) \in \tau_1 \to \tau_2 \\ \exists\Gamma, \quad E \in S(\Gamma) \quad \Gamma \vdash (\mathtt{fun\ x} \to e_t) : \tau_1' \to \tau_2' \ \text{ where } \ S(\tau_1' \to \tau_2') = \tau_1 \to \tau_2\end{array}}{clos(\mathcal{E}\{S\}, y, \mathtt{x}, \lfloor e_t \rfloor_S) \sqsubset clos(E, y, \mathtt{x}, S(e_t))}$$

$$\frac{\begin{array}{c}\mathcal{E} \sqsubset E \quad [\delta_i] = Vlist(V \cup \mathbb{D}, \tau) \\ \exists\Gamma, \quad \Gamma \vdash e_t \quad \forall V.\tau = Gen(\Gamma, e_t) \quad E \in S(\Gamma) \quad V \text{ is out of } S\end{array}}{clos(\mathcal{E}\{S\}, y, \mathtt{d}_1, \mathtt{fun\ d}_2\ \ldots\ \mathtt{d}_n \to \lfloor e_t \rfloor_{Dom(S)+V}) \sqsubset gclos(E, y, V, S(e_t : \tau))}$$

$$\frac{Dom(E) = Dom(\mathcal{E}), \quad \forall \mathtt{x} \in Dom(E), \quad \mathcal{E}(\mathtt{x}) \sqsubset E(\mathtt{x})}{\mathcal{E} \sqsubset E}$$

$$\mathcal{E}\{S\} \equiv \mathcal{E}\{\mathtt{d}_i \Rightarrow «\tau_i» \mid (\delta_i \Rightarrow \tau_i) \in S\}$$

Figure 3.1: Semantic compilation compatibility

Rule for generic closures In a program, there is an expression e_t which define a derived generic value like `let g =` e_t `in ...` with a typing $\Gamma \vdash e_t$. In evaluation of the program, e_t is instantiated to $S(e_t)$ and stored in the generic closure $gclos(E, y, V, S(e_t))$. Then the compilation of the corresponding definition will be evaluated to some compatible closure $clos(\mathcal{E}\{S\}, y, \mathtt{d}_1, \mathtt{fun\ d}_2\ \ldots\ \mathtt{d}_n \to \lfloor e_t \rfloor_{Dom(S)+V})$.

The semantic compatibility does not provide a one-to-one translation from a GML closure to one of IML. For non functional and non generic values, if $v' \sqsubset v$ then $v' = v$, but usually, to one GML closure $clos(E, y, \mathtt{x}, S(e_t))$ for example, we have more than one compatible IML closures $clos(\mathcal{E}\{S\}, y, \mathtt{x}, \lfloor e_t \rfloor_S)$ under the variety of the instantiations S. For example, look at the following two GML programs A and B:

A: `let` $\mathtt{double}_{\{\delta\}}$ `= fun x` $\to$ `plus x x in double 1;;`

B: `let` $\mathtt{double}_{\emptyset}$ `= fun x` $\to$ `(plus : int` $\to$ `int` $\to$ `int) x x in double 1;;`

Both define functions named `double` using `plus` generic primitive. The program A defines a derived generic function `double` whose type is $\forall\delta.\delta \to \delta$, while B instantiates `plus` to `int` $\to$ `int` $\to$ `int` in the definition, and therefore `double` is defined as a non generic function whose type is `int` $\to$ `int`. In the both program, the use of `double` in the expression `double 1` is evaluated to the same closure $c = clos(E, \_, \mathtt{x}, (\mathtt{plus} : \mathtt{int} \to \mathtt{int} \to \mathtt{int})\ \mathtt{x}\ \mathtt{x})$:

$$\text{A:} \quad \frac{E \models \{\delta \Rightarrow \mathtt{int}\}(\mathtt{fun\ x} \to (\mathtt{plus} : \delta \to \delta \to \delta)\ \mathtt{x}\ \mathtt{x}) \Longrightarrow c}{E\{\mathtt{double} = gclos(E, \_, \{\delta\}, \mathtt{fun\ x} \to \mathtt{plus\ x\ x})\} \models \mathtt{double} : \mathtt{int} \to \mathtt{int} \Longrightarrow c}$$

$$\text{B:} \quad \frac{E \models \mathtt{fun\ x} \to (\mathtt{plus} : \mathtt{int} \to \mathtt{int} \to \mathtt{int})\ \mathtt{x}\ \mathtt{x} \Longrightarrow c}{E\{\mathtt{double} = c\} \models \mathtt{double} : \mathtt{int} \to \mathtt{int} \Longrightarrow c}$$

Though their results are the same in the GML semantics, in the evaluations of their compilations:

Compilation of A and its evaluation of `double «int»`:

```
let double = fun ty x → plus ty x x in double «int» 1;;
```

$$\frac{\mathcal{E}\{\texttt{double} = c_{\mathrm{A}}\} \models_i \texttt{double} \Longrightarrow c_{\mathrm{A}} \quad \text{where } c_{\mathrm{A}} = clos(\mathcal{E}, \_, \texttt{ty}, \texttt{fun x} \rightarrow \texttt{plus ty x x}) \qquad \mathcal{E}\{\texttt{double} = c_{\mathrm{A}}\} \models_i \texttt{«int»} \Longrightarrow \texttt{«int»}}{\mathcal{E}\{\texttt{double} = c_{\mathrm{A}}\} \models_i \texttt{double «int»} \Longrightarrow clos(\mathcal{E}\{\texttt{ty} = \texttt{«int»}\}, \_, \texttt{x}, \texttt{plus ty x x})}$$

Compilation of B and its evaluation of `double`:

```
let double = fun x → plus «int» x x in double 1;;
```

$$\frac{\mathcal{E} \models_i \texttt{fun x} \rightarrow \texttt{plus «int» x x} \Longrightarrow c_{\mathrm{B}} \quad \text{where } c_{\mathrm{B}} = clos(\mathcal{E}, \_, \texttt{x}, \texttt{plus «int» x x})}{\mathcal{E}\{\texttt{double} = c_{\mathrm{B}}\} \models_i \texttt{double} \Longrightarrow c_{\mathrm{B}}}$$

The resulting closures of these two compilations which are applied to `1` are slightly different:

$$\begin{array}{lllll} \text{A:} & clos(\mathcal{E}\{\texttt{ty} = \texttt{«int»}\}, & \_, \texttt{x}, & \texttt{plus ty} & \texttt{x x}) \\ \text{B:} & clos(\mathcal{E}, & \_, \texttt{x}, & \texttt{plus «int»} & \texttt{x x}) \end{array}$$

But apparently the closure of B is a partially evaluated form of A and they work same as functions. Both closures are considered semantically compatible with the GML closure

$$clos(E, \_, \texttt{x}, (\texttt{plus} : \texttt{int} \rightarrow \texttt{int} \rightarrow \texttt{int})\ \texttt{x x})$$

Like semantic values we can also have one or more semantically compatible evaluation environments $\mathcal{E}$ to one E.

3.5.2 Compilation correctness

The correctness of the compilation algorithm is now proven using the semantic compatibility:

Theorem 11 (Compilation correctness) *If a fully type-annotated expression e_t has a typing $\emptyset \vdash e_t$ and $\emptyset \models e_t \Longrightarrow v$, then its compilation in IML under the empty compilation environment $\lfloor e_t \rfloor_\emptyset$ is evaluated to v' which is compatible with v without error, i.e.*

$$\emptyset \models_i \lfloor e_t \rfloor_\emptyset \Longrightarrow v' \sqsubset v$$

Like Theorem 4, to prove this theorem we need to enforce the statements for each evaluation step of GML expressions:

Theorem 12 (Stronger compilation correctness) *Let e_t be a fully type-annotated GML expression which has a typing under a typing environment Γ, i.e, $\Gamma \vdash e_t$. Let S be a substitution and E be a GML evaluation environment such that $E \in S(\Gamma)$. If $E \models S(e_t) \Longrightarrow v$ for some v then for any IML evaluation environment $\mathcal{E}$ such that $\mathcal{E} \sqsubset E$,*

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

such that $v' \sqsubset v$.

PROOF: First we prove that $E \models S(e_t)$ always succeeds. From Theorem 3, $S(\Gamma) \vdash S(e_t)$. Since $E \in S(\Gamma)$, from Theorem 5, the evaluation $S(e_t)$ under E always succeeds and returns a semantic value v: $E \models S(e_t) \Longrightarrow v$.

The main proof is done by induction over the evaluation derivation $E \models S(e_t) \Longrightarrow v$.

(e-cnst) e_t must be a constant: let $e_t = c_\sigma : \tau$.

Typing and evaluation in GML:
Its typing is by (t-cnst):

$$\frac{\tau \leq \sigma}{\Gamma \vdash c_\sigma : \tau}$$

Because σ is closed, $S(e_t) = c_\sigma : S(\tau)$. The typing of $S(e_t)$ becomes as follows, for some substitution S_1:

$$\frac{S(\tau) \leq_{S_1} \sigma}{S(\Gamma) \vdash c_\sigma : S(\tau)}$$

The constant is evaluated by (e-cnst):

$$E \models c_\sigma : S(\tau) \Longrightarrow S_1(\mathit{cnst}(c_\sigma))$$

Evaluation of the compiled code:
The compilation of e_t is:

$$\lfloor e_t \rfloor_S = \lfloor c_\sigma : \tau \rfloor_S = c_\sigma$$

and it is evaluated in the IML evaluation as follows, which is compatible to $S_1(\mathit{cnst}(c_\sigma))$ by the definition of cnst': $\mathit{cnst}'(c_\sigma) = \mathit{strip}(\mathit{cnst}(c_\sigma))$.

$$\mathcal{E}\{S\} \models_i c_\sigma \Longrightarrow \mathit{cnst}'(c_\sigma) \sqsubset S_1(\mathit{cnst}(c_\sigma))$$

(e-gen) From the used evaluation rule, e_t must be an instance of a generic primitive: $e_t = \mathsf{g}_\sigma : \tau$.

Typing and evaluation in GML:

The original typing $e_t = \mathsf{g}_\sigma : \tau$ is as follows:

$$\frac{\tau \leq_{S_0} \sigma}{\Gamma \vdash \mathsf{g}_\sigma : \tau}$$

and if we let $\forall V.\tau' = \sigma$, the typing and the evaluation of $S(e_t)$ are:

$$\frac{S_1 = (S \circ S_0)_{|V} \quad S(\tau) \leq_{S_1} \sigma}{S(\Gamma) \vdash \mathsf{g}_\sigma : S(\tau)}$$

$$E \models \mathsf{g}_\sigma : S(\tau) \Longrightarrow \mathit{gen}(\mathsf{g}, S(\tau))$$

Evaluation of the compiled code:

If we let $[\delta_i] = \mathit{Vlist}(V, \tau')$, the expression e_t is compiled to:

$$\lfloor e_t \rfloor_S = \lfloor \mathsf{g}_\sigma : \tau \rfloor_S = \mathsf{g}\ \lfloor S_0(\delta_1) \rfloor_S\ \ldots\ \lfloor S_0(\delta_n) \rfloor_S$$

Each argument $\lfloor S_0(\delta_i) \rfloor_S$ is evaluated by (i-runtype) as follows:

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

It is because $\lfloor \tau \rfloor_S$ replaces the occurrences of $\delta_i \in \mathit{Dom}(S)$ to its special variable d_i, and $\mathcal{E}\{S\}$ has a binding $(\mathsf{d}_i \Rightarrow \text{«}S(\delta_i)\text{»})$ from its definition. Therefore the entire compilation is evaluated by (i-gen) as follows:

$$\mathcal{E}\{S\} \models_i \mathsf{g}\ \lfloor S_0(\delta_1) \rfloor_S\ \ldots\ \lfloor S_0(\delta_n) \rfloor_S \Longrightarrow \mathit{gen}'(\mathsf{g}, \{\delta_i \Rightarrow (S \circ S_0)(\delta_i)\}(\tau'))$$

Since $\tau \leq_{S_0} \sigma = \forall V.\tau'$ and $\{\delta_i\} = V$,

$$\mathit{gen}'(\mathsf{g}, \{\delta_i \Rightarrow (S \circ S_0)(\delta_i)\}(\tau')) = \mathit{gen}'(\mathsf{g}, S(\tau))$$

$\mathit{gen}'(\mathsf{g}, S(\tau)) \sqsubset \mathit{gen}(\mathsf{g}, S(\tau))$ from the definition of gen'.

(e-nvar) Let $e_t = \mathtt{x}_\sigma : \tau$.

Typing and evaluation in GML:

Suppose we have the typing for $e_t = \mathtt{x}_\sigma : \tau$ by (t-inst) for some substitution S_0:

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

From Theorem 3 we have the following typing of $S(e_t)$:

$$\frac{(S(\Gamma))(\mathtt{x}) = S(\sigma) \quad S(\tau) \leq_{S_1} S(\sigma)}{S(\Gamma) \vdash \mathtt{x}_{S(\sigma)} : S(\tau)}$$

where $S_1 = (S \circ S_0)_{|BV(\sigma)}$. Because our rule used here is (e-nvar), $E(\mathtt{x})$ must be a regular value v, not a generic closure. Since $S(\tau) \leq_{S_1} S(\sigma)$,

$$\frac{E(\mathtt{x}) = v}{E \models \mathtt{x}_{S(\sigma)} : S(\tau) \Longrightarrow S_1(v)}$$

Evaluation of the compiled code:

Because $BDV(\sigma) = \emptyset$, the compilation of e_t is simple:

$$\lfloor \mathtt{x}_\sigma : \tau \rfloor_S = \mathtt{x}$$

Since $\mathcal{E} \sqsubset E$, $\mathcal{E}(\mathtt{x}) = v' \sqsubset v$. Hence,

$$\mathcal{E}\{S\} \models_i \mathtt{x} \Longrightarrow v' \sqsubset S(v)$$

(e-gvar) Like in the case of (e-nvar) $e_t = \mathtt{x}_\sigma : \tau$, but its binding in the evaluation environment is a generic closure.

Typing and evaluation in GML:

The typings of $e_t = \mathtt{x}_\sigma : \tau$ and $S(e_t)$ are as follows:

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

$$\frac{(S(\Gamma))(\mathtt{x}) = S(\sigma) \qquad V = BV(\sigma) \qquad S_1 = (S \circ S_0)_{|V} \qquad S(\tau) \leq_{S_1} S(\sigma)}{S(\Gamma) \vdash \mathtt{x}_{S(\sigma)} : S(\tau)}$$

Because we use (e-gvar), $BDV(\sigma) = V \cap \mathbb{D}$ must not empty and $E(\mathtt{x})$ is bound by a generic closure which can be written as $gclos(E', \_, V, e'_t)$:

$$\frac{E(\mathtt{x}) = gclos(E', \_, V, e'_t) \qquad E' \models S_1(e'_t) \Longrightarrow v}{E \models \mathtt{x}_{S(\sigma)} : S(\tau) \Longrightarrow v}$$

Induction step:

Because $\mathcal{E} \sqsubset E$, $\mathcal{E}(\mathtt{x}) \sqsubset E(\mathtt{x}) = gclos(E', \_, V, e'_t)$. From the definition of the semantic compilation compatibility,

$$\mathcal{E}(\mathtt{x}) = clos(\mathcal{E}'\{S'\}, \_, \mathtt{d}_1, \mathtt{fun}\ \mathtt{d}_2\ \ldots\ \mathtt{d}_n \rightarrow \lfloor e''_t \rfloor_{Dom(S')+V})$$

where

$$\begin{array}{l} S'(e''_t) = e'_t \\ E(\mathtt{x}) = gclos(E', \_, V, e'_t) = gclos(E', \_, V, S'(e''_t)) \\ \exists \Gamma',\ \Gamma' \vdash e''_t : \tau''\ \ \forall V.\tau'' = Gen(\Gamma', e''_t) \\ E' \in S'(\Gamma'), \quad V \text{ is out of } S' \end{array}$$

Since $Dom(S_1) = V$ and S' is out of V, $S_1 \circ S' = S_1 + S'$. Now the evaluation of $S_1(e'_t)$ is rewritten as follows:

$$E' \models (S_1 + S')(e''_t) \Longrightarrow v$$

From the generalization of e''_t, $FV(\Gamma') \cap V = \emptyset$. Therefore, from $Dom(S_1) = V$, $(S_1 + S')(\Gamma) = S'(\Gamma)$. Because $E' \in S'(\Gamma')$,

$$E' \in (S_1 + S')(\Gamma') = S'(\Gamma)$$

From this and the typing of $\Gamma' \vdash e''_t$, we can use the induction hypothesis for the evaluation of e''_t:

$$\mathcal{E}'\{S_1 + S'\} \models_i \lfloor e''_t \rfloor_{Dom(S')+V} \Longrightarrow v' \sqsubset v$$

Evaluation of the compiled code:
The expression is compiled into the following IML expression:

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

Each argument $\lfloor S_0(\delta_i) \rfloor_S$ is evaluated as follows, by the same argument we did in (e-gen) part:

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

Since $\mathcal{E}'\{S_1 + S'\} = \mathcal{E}'\{S'\}\{\mathsf{d}_i = «S_1»(\delta_i)\} = \mathcal{E}'\{S'\}\{\mathsf{d}_i = «(S \circ S_0)»(\delta_i)\}$, using (i-app) and (i-fun) n times, we obtain the evaluation of the compilation from the result of the induction:

$$\frac{\dfrac{\mathcal{E}'\{S'\}\{\mathsf{d}_i = «(S \circ S_0)(\delta_i)»\} \models_i \lfloor e''_t \rfloor_{S_1+S'} \Longrightarrow v'}{\ldots}}{\mathcal{E}\{S\} \models_i \mathtt{x}\ \lfloor S_0(\delta_1) \rfloor_S\ \ldots\ \lfloor S_0(\delta_n) \rfloor_S \Longrightarrow v' \sqsubset v}$$

(e-tpl) Let $e_t = e_{t1}\ e_{t2}$.

Typing and evaluation in GML:
We have the following typing of $S(e_t) = S(e_{t1}, e_{t2})$:

$$\frac{S(\Gamma) \vdash S(e_{t1}) \quad S(\Gamma) \vdash S(e_{t2})}{S(\Gamma) \vdash S(e_{t1}, e_{t2})}$$

and its evaluation:

$$\frac{E \models S(e_{t1}) \Longrightarrow v_1 \quad E \models S(e_{t2}) \Longrightarrow v_2}{E \models S(e_{t1}, e_{t2}) \Longrightarrow v_1, v_2}$$

Induction step:
By applying the induction hypothesis to both premises, we get:

$$\mathcal{E}'\{S\} \models_i \lfloor e_{tj} \rfloor_S \Longrightarrow v'_j \sqsubset v_j \text{ for } j = 1, 2$$

Evaluation of the compiled code:
The expression e_{t1}, e_{t2} is compiled as follows:

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

and evaluated in IML system using (i-tpl):

$$\frac{\mathcal{E}\{S\} \models_i \lfloor e_{t1} \rfloor_S \Longrightarrow v'_1 \quad \mathcal{E}\{S\} \models_i \lfloor e_{t2} \rfloor_S \Longrightarrow v'_2}{\mathcal{E}\{S\} \models_i \lfloor e_{t1}, e_{t2} \rfloor_S \Longrightarrow v'_1, v'_2}$$

Clearly, $(v'_1, v'_2) \sqsubset (v_1, v_2)$.

(e-fun) Let $e_t = \texttt{fun x} \rightarrow e'_t$.

Typing and evaluation in GML:
The typing and evaluation of $S(e_t) = \texttt{fun x} \rightarrow S(e'_t)$ are:

$$\frac{S(\Gamma)\{\texttt{x} : S(\tau_1)\} \vdash S(e'_t) : S(\tau_2)}{S(\Gamma) \vdash S(\texttt{fun x} \rightarrow e'_t) : S(\tau_1 \rightarrow \tau_2)}$$

$$E \models S(\texttt{fun x} \rightarrow e'_t) \Longrightarrow \mathit{clos}(E, \_, \texttt{x}, S(e'_t))$$

where $\mathit{clos}(E, \_, \texttt{x}, S(e'_t)) \in S(\tau_1 \rightarrow \tau_2)$ from the Theorem 5.

Evaluation of the compiled code:
The compilation of the expression $\texttt{fun x} \rightarrow e'_t$ is:

$$\lfloor \texttt{fun x} \rightarrow e'_t \rfloor_S = \texttt{fun x} \rightarrow \lfloor e'_t \rfloor_S$$

and it is evaluated as follows:

$$\mathcal{E}\{S\} \models_i \texttt{fun x} \rightarrow \lfloor e'_t \rfloor_S \Longrightarrow \mathit{clos}(\mathcal{E}\{S\}, \_, \texttt{x}, \lfloor e'_t \rfloor_S)$$

Since $\Gamma \vdash \texttt{fun x} \rightarrow e'_t : \tau_1 \rightarrow \tau_2$, from the definition of the semantic compilation compatibility between closures,

$$\mathit{clos}(\mathcal{E}\{S\}, \_, \texttt{x}, \lfloor e'_t \rfloor_S) \sqsubset \mathit{clos}(E, \_, \texttt{x}, S(e'_t))$$

(e-let) This time $e_t = \texttt{let x}_V = e_{t1} \texttt{ in } e_{t2} : \tau_2$. Since it is evaluated by (e-let), the generalized variable set V must not contain dynamic type variables.

Typing and evaluation in GML:
We have the typing of the expression e_t:

$$\frac{\Gamma \vdash e_{t1} : \tau_1 \quad \forall V.\tau_1 = \mathit{Gen}(\Gamma, e_{t1}) \qquad \Gamma\{\texttt{x} : \forall V.\tau_1\} \vdash e_{t2} : \tau_2}{\Gamma \vdash \texttt{let x}_V = e_{t1} \texttt{ in } e_{t2} : \tau_2}$$

where $V \cap \mathbb{D} = \emptyset$. Then one for $S(e_t)$ is as follows (we use the renaming of V so that V is out of S like we did in the proof of Theorem 3):

$$\frac{S(\Gamma) \vdash S(e_{t1}) : S(\tau_1) \quad \forall V.S(\tau_1) = \mathit{Gen}(\Gamma, S(e_{t1})) \qquad S(\Gamma)\{\texttt{x} : \forall V.S(\tau_1)\} \vdash S(e_{t2}) : S(\tau_2)}{S(\Gamma) \vdash S(\texttt{let x}_V = e_{t1} \texttt{ in } e_{t2}) : S(\tau_2)}$$

It is evaluated as below:

$$\frac{E \models S(e_{t1}) \Longrightarrow v_1 \quad E\{\texttt{x} = v_1\} \models S(e_{t2}) \Longrightarrow v_2}{E \models \texttt{let x}_V = S(e_{t1}) \texttt{ in } S(e_{t2}) \Longrightarrow v_2}$$

Induction step:

By using the induction hypothesis for the evaluation of $S(e_{t1})$, the compilation is evaluated to some v_1' which is compatible to v_1:

$$\mathcal{E}\{S\} \models_i \lfloor e_{t1} \rfloor_S \Longrightarrow v_1' \sqsubset v_1$$

From the semantic typing of the evaluation $E \models S(e_{t1} : \tau_1) \Longrightarrow v_1$, $v_1 \in S(\tau_1)$. From Lemma 9, $v_1 \in \forall V.S(\tau_1) = S(\forall V.\tau_1)$. Since $E \in S(\Gamma)$,

$$E\{\mathtt{x} = v_1\} \in S(\Gamma)\{\mathtt{x} : S(\forall V.\tau_1)\} = S(\Gamma\{\mathtt{x} : \forall V.\tau_1\})$$

From this and the typing of

$\Gamma\{\mathtt{x} : \forall V.\tau_1\} \vdash e_{t2}$ and the evaluation $E\{\mathtt{x} = v_1\} \models S(e_{t2}) \Longrightarrow v_2$, we can use the induction hypothesis and obtain:

$$\mathcal{E}\{\mathtt{x} = v_1'\}\{S\} \models_i \lfloor e_{t2} \rfloor_S \Longrightarrow v_2' \sqsubset v_2$$

Evaluation of the compiled code:

The compilation of the entire expression is as follows:

$$\lfloor \mathtt{let}\ \mathtt{x}_V = e_{t1}\ \mathtt{in}\ e_{t2} \rfloor_S = \mathtt{let}\ \mathtt{x}_V = \lfloor e_{t1} \rfloor_S\ \mathtt{in}\ \lfloor e_{t2} \rfloor_S$$

and evaluated by (i-let), using the results of the inductions.

$$\frac{\mathcal{E}\{S\} \models_i \lfloor e_{t1} \rfloor_S \Longrightarrow v_1' \quad \mathcal{E}\{S\}\{\mathtt{x} = v_1'\} \models_i \lfloor e_{t2} \rfloor_S \Longrightarrow v_2'}{\mathcal{E}\{S\} \models_i \mathtt{let}\ \mathtt{x}_V = \lfloor e_{t1} \rfloor_S\ \mathtt{in}\ \lfloor e_{t2} \rfloor_S \Longrightarrow v_2' \sqsubset v_2}$$

(e-g-let) The expression takes the same form as the case of (e-let), $e_t = \mathtt{let}\ \mathtt{x}_V = e_{t1}\ \mathtt{in}\ e_{t2} : \tau_2$. But in this case V contains dynamic type variables.

Typing and evaluation in GML:

We have the same typing as the case of (e-let). The difference is that $V \cap \mathbb{D} \neq \emptyset$. $S(e_t)$ is now evaluated differently, by (e-g-let):

$$\frac{E\{\mathtt{x} = gclos(E, \_, V, S(e_{t1}))\} \models S(e_{t2}) \Longrightarrow v}{E \models \mathtt{let}\ \mathtt{x}_V = S(e_{t1})\ \mathtt{in}\ S(e_{t2}) \Longrightarrow v}$$

Induction step:

From the typing of e_{t1}, we have

$$\Gamma\{\mathtt{x} = \forall V.\tau_1\} \vdash e_{t1}$$

Let

$$\begin{aligned} c &= gclos(E, \_, V, S(e_{t1})) \\ c' &= clos(\mathcal{E}\{S\}, \_, \mathtt{d}_1, \mathtt{fun}\ \mathtt{d}_2\ \ldots\ \mathtt{d}_n \rightarrow \lfloor e_t \rfloor_{Dom(S)+V}) \end{aligned}$$

Then from the typing of $S(e_{t1})$, using (s-gclos), $c \in S(\forall V.\tau_1)$. Therefore,

$$E\{\mathtt{x} = c\} \in S(\Gamma\{\mathtt{x} = \forall V.\tau_1\})$$

From the definition of the semantic compilation compatibility, we have $c' \sqsubset c$, and this leads to:

$$\mathcal{E}\{\mathtt{x} = c'\} \sqsubset E\{\mathtt{x} = c\}$$

From these we can use the induction hypothesis for the evaluation $E\{\mathtt{x} = c\} \models S(e_{t2}) \Longrightarrow v$ and obtain

$$\mathcal{E}\{S\}\{\mathtt{x} = c'\} \models_i \lfloor e_{t2} \rfloor_S \Longrightarrow v' \sqsubset v$$

Evaluation of the compiled code:
The compilation of e_t becomes:

$$\lfloor \mathtt{let}\ \mathtt{x}_V \mathtt{=} e_{t1}\ \mathtt{in}\ e_{t2} \rfloor_S = \mathtt{let}\ \mathtt{x}_V \mathtt{=} \mathtt{fun}\ \mathtt{d}_1 \ldots \mathtt{d}_n \rightarrow \lfloor e_{t1} \rfloor_{Dom(S)+V}\ \mathtt{in}\ \lfloor e_{t2} \rfloor_S$$

and its evaluation is as follows by (i-let) using the result of the induction:

$$\frac{\mathcal{E}\{S\}\{\mathtt{x} = c'\} \models_i \lfloor e_{t2} \rfloor_S \Longrightarrow v'}{\mathcal{E}\{S\} \models_i \mathtt{let}\ \mathtt{x}_V \mathtt{=} \mathtt{fun}\ \mathtt{d}_1 \ldots \mathtt{d}_n \rightarrow \lfloor e_{t1} \rfloor_{Dom(S)+V}\ \mathtt{in}\ \lfloor e_{t2} \rfloor_S \Longrightarrow v' \sqsubset v}$$

(e-app1) Let $e_t = e_{t1}\ e_{t2}$.

Typing and evaluation in GML:
The typing of $S(e_t) = S(e_{t1})\ S(e_{t2})$ is as follows:

$$\frac{S(\Gamma) \vdash S(e_{t1}) : S(\tau_1 \rightarrow \tau_2) \qquad S(\Gamma) \vdash S(e_{t2}) : S(\tau_1)}{S(\Gamma) \vdash S(e_{t1}\ e_{t2}) : S(\tau_2)}$$

Its evaluation is done by (e-app1):

$$\frac{\begin{array}{c} E \models S(e_{t2}) \Longrightarrow v_2 \qquad E \models S(e_{t1}) \Longrightarrow clos(E', \_, \mathtt{x}, e'_t) \\ E'\{\mathtt{x} = v_2\} \models e'_t \Longrightarrow v \end{array}}{E \models S(e_{t1}\ e_{t2}) \Longrightarrow v}$$

Induction step:
Applying the induction hypothesis to the first evaluation premise, we get:

$$\mathcal{E}\{S\} \models_i \lfloor e_{t2} \rfloor_S \Longrightarrow v'_2 \sqsubset v_2$$

We also use the induction hypothesis for the second evaluation premise

$$E \models S(e_{t1}) \Longrightarrow clos(E', \_, \mathtt{x}, e'_t)$$

there is a typing environment Γ', a substitution S' and an expression e''_t such that

$$\begin{array}{l} S'(e''_t) = e'_t \\ \exists\Gamma',\quad \Gamma' \vdash (\texttt{fun x} \rightarrow e''_t) : \tau'_1 \rightarrow \tau'_2 \text{ where } S'(\tau'_1 \rightarrow \tau'_2) = S(\tau_1 \rightarrow \tau_2) \\ E' \in S'(\Gamma') \\ \mathcal{E}' \sqsubset E' \end{array}$$

and

$$\mathcal{E}\{S\} \models_i \lfloor e_{t1} \rfloor_S \Longrightarrow clos(\mathcal{E}'\{S'\}, \_, \texttt{x}, \lfloor e''_t \rfloor_{S'}) \sqsubset clos(E', \_, \texttt{x}, S'(e''_t))$$

The typing of $\texttt{fun x} \rightarrow e''_t$ must be deduced by (t-fun):

$$\frac{\Gamma'\{\texttt{x} : \tau'_1\} \vdash e''_t : \tau'_2}{\Gamma \vdash \texttt{fun x} \rightarrow e''_t : \tau'_1 \rightarrow \tau'_2}$$

Since $S'(\tau'_1) = S(\tau_1)$, $v_2 \in S'(\tau'_1) = S(\tau_1)$. From $E' \in S'(\Gamma')$,

$$E'\{\texttt{x} = v_2\} \in S'(\Gamma'\{\texttt{x} : \tau'_1\})$$

From $\mathcal{E}' \sqsubset E'$ and $v'_2 \sqsubset v_2$, we have

$$\mathcal{E}'\{\texttt{x} = v'_2\} \sqsubset E'\{\texttt{x} = v_2\}$$

Now we can apply the induction hypothesis to the evaluation of $e'_t = S(e''_t)$ and obtain:

$$\mathcal{E}'\{S'\}\{\texttt{x} = v'_2\} \models \lfloor e''_t \rfloor_{S'} \Longrightarrow v'$$

where $v' \sqsubset v$

Evaluation of the compiled code:
The compilation of e_t is as follows:

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

and evaluated using the results and (i-app1):

$$\frac{\mathcal{E}\{S\} \models_i \lfloor e_{t2} \rfloor_S \Longrightarrow v'_2 \quad \mathcal{E}\{S\} \models_i \lfloor e_{t1} \rfloor_S \Longrightarrow clos(\mathcal{E}'\{S'\}, \_, \texttt{x}, \lfloor e'_t \rfloor_{S'}) \quad \mathcal{E}'\{S'\}\{\texttt{x} = v'_2\} \models_i \lfloor e'_t \rfloor_{S'} \Longrightarrow v'}{\mathcal{E}\{S\} \models_i \lfloor e_{t1} \rfloor_S\ \lfloor e_{t2} \rfloor_S \Longrightarrow v' \sqsubset v}$$

(e-gvar-r), (e-let-r), (e-g-let-r) and (e-app2) All are provable in the same way as their non-recursive versions.

□

Now Theorem 11 is just a special case of Theorem 12, when $\Gamma = \emptyset, E = \emptyset, S = \emptyset$.

3.6 Coherence problem of generic primitives

As Ohori[Oho89] pointed out, in the analysis of the denotational semantics of implicitly typed ML language, if we build a semantic system over explicitly typed ML terms, different typing derivations of an implicitly typed program may provide incompatible semantics each other. This problem is know as the *coherence problem*[BTCGS94]. But in the actual execution of ML programs, this coherence problem is not a big trouble, since programs are evaluated under the semantics of the untyped lambda calculus. Ignoring all the type informations, behavior of a program at execution does not depend on its typing. However, as Jones discussed in an ML type system extension for qualified types[?], the coherence has greater importance once we introduce type-dependent semantics of polymorphic values, since different typings of polymorphic instances may provide completely different semantics at execution.

Our language has generic primitives with type-dependent semantics. Therefore, we cannot directly define semantics of a bare expression without type annotations. Its semantics is obtained through the semantics of the type-annotated GML, by mapping the bare expression to a fully type-annotated expression. If a bare expression has more than one possibility of typing, these typings may provide different semantics incompatible each other. For example, consider a generic primitive $\mathtt{g}_{\forall\delta.\delta\ \mathtt{list}}$ with the following semantics:

$$
\begin{array}{lcl}
gen(\mathtt{g},\ \mathtt{int\,list}) & \leadsto & \mathtt{[1]} \\
gen(\mathtt{g},\ \mathtt{string\,list}) & \leadsto & \mathtt{["hello";\ "world"]} \\
\ldots & &
\end{array}
$$

Then a source program `List.length g` is *incoherent*, since depending on its typing derivations, it has different semantics and returns different results:

```
List.length (g : int list)    ⇝ 1
List.length (g : string list) ⇝ 2
```

Incoherence of programs must be detected by type checker, since we cannot fix their semantics.

The coherence of a program can be roughlydefined as follows:

Definition 7 (Coherence) *A bare program e is coherent, when it is typable: $W(\emptyset, e) = e_t$ and for any substitution S, $\lfloor e_t \rfloor_\emptyset \equiv \lfloor S(e_t) \rfloor_\emptyset$.*

Looking at the compilation algorithm $\lfloor\cdot\rfloor_V$, we can easily obtain the following theorem:

Theorem 13 (Free dynamic type variables cause incoherence) *A program e is coherent when its principal typing $e_t = W(\emptyset, e)$ contains no free dynamic type variable occurrence: $DV(e_t) = \emptyset$.*

In other words, when the algorithm W finds a principal typing of a given program e such that $e_t = W(\emptyset, e)$ and e_t contains free occurrences of dynamic type variables, e is incoherent. In the case of incoherence, the programmer must explicitly constrain types of incoherent generic instances to remove the incoherence and provide appropriate semantics to the program. The type inference may be able to choose one of possible typings by instantiating the free dynamic type variable occurrences to some closed types in order to provide a default semantics automatically of an incoherent program, but there is always a risk that such "default" can be incompatible with the programmer's intension. Therefore, the system must print out a warning message if it performs this auto-elimination of incoherence. Chapter 10 will discuss again about the detection of incoherence using *generic constraint typing system* (Chapter 8) which prevents incoherent instantiations of generic values.

Future work around coherence

The definition of coherence Definition 7 is not complete, though the following result Theorem 13 is correct. We considers only the typings obtained by substituting free type variables from the principal typing $e_t = W(\emptyset, e)$ here. For the precise discussion, the other possible typings must be also taken into account, introducing equational theory between compiled code. Also, the coherence can be discussed over the semantics of type-annotated GML defined in Section 2.4, instead of using compilation equality. These formalization is left as a future work.

3.7 Conclusion

We defined a compilation algorithm of GML to IML, an untyped ML like language. The operational semantics of GML with type annotations and instantiations are now translated to classical typeless semantics of IML which can be compiled to efficient executable programs by existing ML compilation techniques.

Type instantiation mechanism of the GML semantics system is translated to run time type dispatching in IML. Evaluation type contexts of generic values are passed as run time types, normal ML value representation of type expressions through functional applications. Generic values are compiled to functions which take these dispatched run time types to reconstruct their evaluation type context. This idea is exactly same as the explicit type abstractions and applications in System F [GTL90], where the definition of `double` whose type scheme is $\forall\delta.\delta \to \delta$ can be written as $\Lambda\delta.\lambda$`x.plus` δ `x x`, for example. In the compilation of extensional polymorphism, however, the type abstractions of regular type variables $\Lambda\alpha$ and type applications to them are removed for efficiency, since polymorphism by regular type variables is parametric and its instantiation does not affect the semantics. Therefore, run time type dispatching is performed only where it is really required by generic values that are quantified by dynamic type variables. Non-generic part of programs is compiled to the identical expressions without type annotations, therefore they are evaluated as efficiently as in the original ML system, without any overhead.

Chapter 4

Dynamic values

4.1 Introduction

Dynamic values — an example of generic primitives

In the previous chapters, the semantics of the generic primitives are defined explicitly. We have defined their typing, but never discussed about the internals — semantics. In this chapter, we try to give an important example of $gen(\mathtt{g}, \tau)$ — the primitives for *dynamic values* [ACPR92, LM94]. Dynamic values provide a way to perform run-time typing on values. This mechanism is provided by two primitives, one for creating dynamic values and another for destructing them. These two primitives are often called as `dynamic` and `coerce` respectively. The expression `dynamic` v creates a dynamic value from v, that is a pair (v, τ) which consists of v and its type τ. This $(\text{value}, \text{type})$ pair is encapsulated into an abstract value of the abstract type ***dyn*** for the dynamic values, hiding the original type of the value. `coerce` expression destructs these dynamic values and can handle their contents differently depending on their types, performing simple run time type check. This type hiding capability is useful to simulate heterogeneous data structure and overloaded functions in ML, even the cost of manual encapsulation and decapsulation of dynamic values.

Dynamic primitives and polymorphic dynamization

We introduce the `dynamic` and `coerce` primitives as generic primitives and see how easily their basic functionality can be implemented in extensional polymorphism framework; the run time type dispatching in the compilation of extensional polymorphism provides these primitives almost for free. Then we try polymorphic dynamic encapsulation: encapsulation of polymorphic values in dynamic values that can be coerced back again to polymorphic values. Since this is not always possible, we must extend the typing for dynamic primitives so that they can perform generalization checks of the types of dynamized values.

Outline of this chapter

Section 4.2 introduces the primitives for dynamic values, `dynamic` and `coerce` as generic primitives, and give their semantics as the forms of compiled expression in the intermediate language IML. From Section 4.3 we add polymorphic encapsulation and decapsulation capability to these primitives. Section 4.4 explains required type system modifications for polymorphic dynamization. Then in Section 4.5, `coerce` primitive is extended to a special expression construct to decapsulate the contents of dynamized values to polymorphic identifiers.

4.2 Dynamic values in extensional polymorphism

4.2.1 Dynamic values

Dynamic values provide simple typing of values at run time. This mechanism is provided by two primitives, known as `dynamic` and `coerce` (or `typecase`). Usually `dynamic` primitive receives a value v and returns a dynamic value which is the pair (v, τ), v itself and its type τ. This pair is abstracted as a value of the abstract type `dyn`, so that we can hide the type information. To destruct the dynamic values and retrieve their content, `coerce` primitive checks whether their internal type information τ is "compatible" to the expected type context τ' (how the coerced content is used). Intuitively, if τ is more general than τ', τ can be considered as compatible to τ'. If incompatible, the `coerce` primitive rejects the coercion of the dynamic values. The following code shows that a dynamic encapsulation of integer. The dynamic values of 1 can be coerced back to integer, but not to float:

```
# let d = dynamic 1;; (* creates a pair (1,int) *)
d : dyn = <abstr>
# coerce d + 1;; (* coercion to int, compatible *)
- : int = 2
# (coerce d : float);; (* coercion to float, incompatible *)
Uncaught_exception: Coercion_failure (* An exception instead of Seg. fault *)
```

The coercion may be able to take multiple cases. In this case, `coerce` is no longer a primitive function, but a special expression syntax: The expression

$$
\begin{array}{l}
\texttt{coerce } d_1\texttt{,...,}d_n \texttt{ with}\\
\texttt{| (x}_{11}\texttt{ : }\tau_{11}\texttt{),..., (x}_{1n}\texttt{ : }\tau_{1n}\texttt{)} \rightarrow e_1\\
\texttt{| ...}\\
\texttt{| (x}_{m1}\texttt{ : }\tau_{m1}\texttt{),..., (x}_{mn}\texttt{ : }\tau_{mn}\texttt{)} \rightarrow e_m
\end{array}
$$

first tries to coerce n dynamic values $d_1, \ldots, d_n$ to $\tau_1, \ldots, \tau_n$ respectively, and if their coercion are all possible, the contents of the dynamic values are bound to the pattern variables $\mathtt{x}_i$ to evaluate e_1. Otherwise, it tries the other cases. If none of the cases matches with the type information of $d_1, \ldots, d_n$, it raises a run time exception.

Even with the cost of the run time type checking at coercion, the dynamic values are sometimes useful to unify various typed values into one `dyn` type. For example, using dynamic values we can simulate heterogeneous lists as follows:

```
# let hlist = [dynamic 1; dynamic 3.14; dynamic "hello"];;
hlist : dyn list = [ <abstr>; <abstr>; <abstr> ]
```

The created list `hlist` has the type `dyn list` which is totally monogeneous, but the types of the elements are stored inside dynamic values. To retrieve elements from this heterogeneous list, we have to coerce manually, of course:

```
# let print_dyn d =
    coerce d with
    | x : int ⇒ print_int x
    | x : float ⇒ print_float x
    | x : string ⇒ print_string x
print_dyn : dyn → unit
# List.iter print_dyn hlist;;
1 3.14 hello
- : unit = ()
```

The above code simulates an overloaded function using dynamic values. This overloaded behavior is quite similar to generic values in extensional polymorphism. However, in the ability of defining overloaded values, dynamics clearly fall behind extensional polymorphism.

First, to use these dynamic generic functions we require explicit use of the `dynamic` primitive to give their arguments, like `print_dyn (dynamic 1)`. In extensional polymorphism framework, the run time type dispatch for generic values is automatic in the compilation, as we have seen in Chapter 3.

Second, hiding the types of encapsulated values makes it impossible to detect illegal applications of dynamic generic functions statically. For example, when we are given an expression `print_dyn d`, whether `d` has a valid type for `print_dyn` (i.e. `int`, `float` or `string`) is not known until at run time evaluation when we have `d` at hand. On the other hand, the user-defined generic values which will be available in Chapter 7 can also define non-total overloaded values, but their instance validity can be statically checked by the generic constraint typing system later introduced in Chapter 8.

In spite of these disadvantages compared to generics, dynamic values are still very interesting and important in extensional polymorphism. First of all, we can obtain its basic primitives almost for free in extensional polymorphism, as already proposed in [DRW95]. In addition, the abstract type `dyn` can provide a powerful type reconstruction tactics called *dyn-unification* to solve ambiguity problems of generic constraint typing. In the course of the type reconstruction of generic values, we may have free dynamic type variables which actually should not exist freely and must be instantiated to some closed type. The abstract type `dyn` is a good candidate of this closed type, since we can postpone the statically unsolved typing problems using the run time type checking mechanism of the dynamic values.

4.2.2 Type variables inside dynamic values — polymorphic contents

Using the dynamic encapsulation, types can be exported outside of their original scope. Therefore, the type information in dynamic values must be independent of any context. This means that dynamic type informations must be closed type schemes: all the type variables inside dynamic values must be quantified. Having type variables inside, dynamic values can encapsulate *polymorphic contents*. For instance, `dynamic []` will create a pair of (`[]`, α `list`) and it will be able to coerced back to a polymorphic list of type $\forall\alpha.\alpha$ `list`. To emphasize this, we write the type information of dynamic values as closed type schemes. For example, the value-type pair of the dynamic value of the polymorphic list is written as follows:

(`[]`, $\forall\alpha.\alpha$ `list`)

When a value v of type τ is encapsulated as a dynamic value $(v, \forall FV(\tau).\tau)$, we must check the generalizability of the type τ at type checking phase, since it is just a mono-type. If τ contains non-generalizable type variables, the encapsulation of $(v, \forall FV(\tau).\tau)$ must not be performed. Type checker must reject such encapsulation or automatically unify such non-generalizable type variables with some closed type. This new typing of the dynamized values is discussed in Section 4.3.

4.2.3 Dynamic primitives

The dynamic value primitives are easily implemented in extensional polymorphism framework. Their types are given as follows, with the abstract type for dynamic values, `dyn`:

```
type dyn (* the abstract type for dynamic values *)
dynamic : ∀δ.δ → dyn
coerce : ∀δ.dyn → δ
```

In the evaluation, the `dynamic` primitive receives the type of its argument as a part of its evaluation type context. On the other hand, the `coerce` receives the expected type of its coercion. They are compiled to functions[1] which receive these (run time) types as first arguments to create and destruct dynamic values:

$$
\begin{aligned}
gen'(\texttt{dynamic}) &= \texttt{fun t v} \rightarrow (\texttt{v},\texttt{t}) \\
gen'(\texttt{coerce}) &= \texttt{fun t } (\texttt{v},\texttt{t}') \rightarrow \texttt{if is\_instance t t}' \texttt{ then v} \\
&\qquad\qquad\qquad\qquad\quad \texttt{else raise Coercion\_failure}
\end{aligned}
$$

Additional to the run time type argument, the compilation of `dynamic` also receives a value `v` and creates a value-type pair `(v,t)`. This pair is returned as a value of the abstract type `dyn`. `coerce` receives a dynamic value whose internal structure is `(v,t')` and returns its contents `v`, checking that the expected type `t` is a proper instance of `t'`. Otherwise, the coercion is rejected as a type violation. The predicate `is_instance` is equivalent to the predicate $\leq$ in our type system, whose definition is given by the pseudo code in Figure 4.1. The data types are identified simply by their names `t` at this moment.

[1] More precisely, these compilations are not the functions themselves, but their closures. For instance, $gen'(\texttt{dynamic}) = clos(\emptyset, \_, \texttt{t}, \texttt{fun v} \rightarrow (\texttt{v},\texttt{t}))$

```
let is_instance t t' =
  let subst = ref [] in
  let rec is_instance_aux t t' =
    match t, t' with
    | «τ», «α» →
        begin try
            if List.assoc «α» !subst = t then true else false
          with
            Not_found → subst := («α», t) :: !subst; true
        end
    | «τ1 → τ2», «τ1' → τ2'» →
        is_instance_aux «τ1» «τ1'» && is_instance_aux «τ2» «τ2'»
    | «τ1 * τ2», «τ1' * τ2'» →
        is_instance_aux «τ1» «τ1'» && is_instance_aux «τ2» «τ2'»
    | «base1», «base2» → base1 = base2
    | «(τ1,...,τn) t» «(τ1',...,τn'') t'» when n = n' && t = t' →
        is_instance_aux «τ1» «τ1'» && ... && is_instance_aux «τn» «τn'»
    | _ → false
  in
  is_instance_aux t t'
```

Figure 4.1: Definition of the predicate is_instance

Corresponding to the implicit generalization of type variables inside dynamic values, the stored run-time types are treated as closed type schemes in is_instance. The following relationship holds in fact:

$$\texttt{is\_instance}\ «\tau»\ «\tau'» \equiv \tau \leq \forall FV(\tau').\tau'$$

Note that the dynamic primitives cannot be defined by generic constructs for the type-case based definition of generic values which will be introduced in Chapter 7, since generic values defined by the generic binding use dispatched run-time types just for branching and have no way to create values from received run time types.

4.3 Polymorphic encapsulation of dynamic values

As shown previously in the definition of the predicate is_instance, dynamic type informations are closed type schemes and all the type variables in them are quantified, so that it can be safe to create dynamic values with polymorphic contents. To ensure this property of value-type pairs, we must introduce new type checks for the dynamic primitives:

- At the encapsulation of v of type τ as $(v, \forall FV(\tau).\tau)$, the generalizability of τ must be checked. Otherwise non polymorphic values with free type variables could be encapsulated into dynamic values as polymorphic and break type safety at coercion.

- Polymorphic contents must be able to be coerced to values with polymorphic type schemes. This is impossible for the previous implementation of the `coerce` primitive as a generic primitive function. The coercion must be a special expression.

In the rest of this chapter, we study these problems of polymorphic encapsulation..

4.4 Polymorphic dynamization

Once type variables are encoded to run time types and captured into dynamic values, they can easily escape from their original scopes at typing. Therefore, the run time type variables inside dynamic values must be treated as closed type schemes which are independent of the typing contexts. This polymorphic encapsulation introduces implicit generalization of the dynamized expressions: the arguments to `dynamic` are silently generalized and stored as polymorphic inside dynamic values. We require special care in the type system so that only the type variables which are safely isolated from the typing context can be generalized and introduced into dynamic values. Otherwise, the type soundness is easily broken: for instance, suppose that `dynamic (ref [] :` α `list ref)` would create a dynamic value with a polymorphic content `(ref [],`$\forall\alpha.\alpha$ `list ref)` (which is clearly incorrect). We could insert anything to the reference cell and violates soundness by coercing this polymorphic content to corresponding types:

```
let d = dynamic (ref []) in
let x = coerce d : int  list ref in x := 1 :: !x;
let x = coerce d : bool list ref in x := true :: !x
(* the list now contains an integer and a boolean! *)
```

This problem can be avoided to restrict the type variable capturing only to the "*safely encapsulable*" type variables which are proven not to damage the type soundness. Most simply, if `dynamic` primitive takes values with non-encapsulable type variables, we should treat it as a typing error:

```
# dynamic (ref []);;
  ^^^^^^^
Error: dynamizer dynamic captures type α list ref,
  which contains a non-generalizable type variable.
```

This asks the programmer to eliminate the non-encapsulable type variables by adding an explicit type constraint without type variables. For instance:

```
# dynamic (ref [] : int list ref);;
- : dyn = <abstr> (* (ref [], int list ref) : this is safe *)
```

The type inference system can automate this elimination of dangerous non-encapsulable dynamic type variables, by unifying with some closed type. This ***dyn**-unification* is explained later in the type inference algorithm for the `dynamic` primitive, Section 4.4.5.

4.4.1 Encapsulable type variables

Now, our question is: what are the encapsulable type variables ?

Argument generalization check

In simple cases, when the run time types received by `dynamic` are statically known, encapsulable type variables are determined quite like the generalizable type variables in let bindings. For the expression `dynamic` (e:τ), we try to generalize the argument type τ implicitly, as if we had the following let binding around it:

```
dynamic (let v = (e:τ) in v)
```

The generalizable type variables in τ in this let binding are independent of the typing context of e, hence they are encapsulable. For example, in `dynamic (fun x → x : δ → δ)`, the type variable δ is encapsulable since it is generalized in `let v = fun x → x`. On the other hand, `dynamic (ref [] : δ list ref)` must not be well-typed and cannot create a polymorphic content since δ is not generalizable in the expression `let v = (ref [] : δ list ref)`.

The generalization checks for dynamization are restricted only against the arguments of the `dynamic` primitive. Making the places where the generalizations occur explicit as the let binding, we can keep the efficiency of the ML type inference. This restriction has, however, a drawback: the expressions which are not arguments of the `dynamic` primitive cannot be encapsulated polymorphically. The following example is rejected to encapsulate the identity function as polymorphic, though it is completely safe:

```
# (fun x → dynamic x) (fun x → x)
           ^^^^^^^
Error: dynamic captures type α → α
  which contains a non-generalizable type variable.
```

Derived dynamizers

The argument generalization of the `dynamic` primitive can be applied only to the type variables which are not generalized by any outer polymorphism. The type variables generalized outside must be treated differently, otherwise any *derived dynamizer*, derived generic values which use `dynamic` inside, would be ill-typed:

```
let double_dynamic = function x → dynamic (x, x) in ...
```

In the above definition of the derived dynamizer `double_dynamic`, the argument of the `dynamic` primitive `(x, x)` has the type $\delta * \delta$, where δ is bound to the variable `x` and not generalizable at the position of `(x, x)`. If we would apply the argument generalization to the type variable δ, it would be regarded as non-encapsulable and therefore had to be eliminated: `double_dynamic` would be forced to be monomorphic. This is too strict and not what we want.

We must not perform the type generalization check of the type variables which are generalized outside, since they are not the actual type which are encapsulated into dynamic values; they are instantiated to other types at run time, depending on how the derived dynamizers are used. For example, if in an application `double_dynamic` e, e has a type τ, the expression `(x, x)` is instantiated to $\tau * \tau$ in its evaluation. Whether the type variables in $\tau * \tau$ are encapsulable or not should not be determined by the generalization of `(x, x)` but by the type generalization of the expression $e : \tau$.

Can we find the encapsulated type variables of the derived dynamizers and secure the polymorphic encapsulation by their argument generalization check, just like the `dynamic` primitive? Unfortunately not. The following example shows that we need more typing analysis:

```
let apply_and_dynamize g =
  let v = g () in
  dynamic v
in
apply_and_dynamize (fun () → ref [])
```

The type of the argument `(fun () → ref [])` itself is generalizable to $\forall\delta.$`unit` $\rightarrow \delta$ `list ref`, but the derived dynamizer `apply_and_dynamize` creates a dynamic value of `ref []` which cannot be polymorphic: though δ is generalizable at the use of the derived dynamizer, it is not encapsulable. In order to assure the security of polymorphic encapsulation of the derived dynamizers, their definitions must be investigates. This additional type checking may affect the efficiency of the type inference (See Section 4.6).

Here we propose a restricted, but simpler solution. We give up the polymorphic encapsulation of derived dynamizers: only the direct arguments to the `dynamic` primitive can be polymorphically encapsulated into dynamic values. Our solution is summarized as follows:

The `dynamic` primitive

- Only the `dynamic` primitive can encapsulate the type variables of its arguments as quantified, after their generalization checks. For instance, `dynamic (fun x → x)` creates a dynamic value `(fun x → x,` $\forall\alpha.\alpha \rightarrow \alpha$`)`.
- The type variables generalized outside, which defines derived dynamizers are kept well-typed. (However, their instantiations at run time are restricted to closed types.): `double_dynamic` can have a polymorphic type scheme $\forall\delta.\delta \rightarrow$ `dynamic`.
- The other type variables annotated to the arguments of the `dynamic` primitive are rejected, since they are not encapsulable.

Derived dynamizers

- The definition of the derived polymorphic dynamizers using `dynamic` or other derived dynamizers is not restricted at all.
- Their arguments cannot have any type variable in their type annotations, unless they are generalized outside. For example, `double_dynamic (fun x → x)` is well-typed only when the expression `(fun x → x)` has a closed type.

4.4.2 Typing rules for the polymorphic dynamization

We need to distinguish the derived dynamizers from the other generic values by looking at their type schemes so that our additional typing restriction can be applied only to the arguments of the derived dynamizers. The current GML type system expressiveness does not tell whether a generic value is a dynamizer or not.

Let us introduce, temporarily, a new typing system, for discussing the safety of the polymorphic dynamization. Once we introduce the generic constraint type system in Chapter 8, all the result of this section will be easily integrated. In addition, with generic constraints we will not need the distinction of the following poly-dynamic and normal dynamic type variables. The poly-dynamic quantifications in this formalizations will be translated to normal dynamic variable quantifications with derived constraints of the `dynamic` primitive. Look at Section 4.7 further details of the type system integration.

Poly-dynamic type variables and partial quantification

To distinguish normal generic values and dynamizers, we introduce another special subset of dynamic type variables called *poly-dynamic type variables* $\mathbb{P} \subset \mathbb{D}$ whose elements are written as $\partial, \partial_1, \partial_2, \ldots$. Now the type schemes of the dynamizers are quantified by poly-dynamic type variables. For example,

$$\texttt{dynamic} : \forall \partial . \partial \rightarrow \texttt{dyn}$$

Poly-dynamic type variables are dynamic type variables with a special restriction: the existence of non-quantified poly-dynamic type variables ∂ in the well-typed expressions is not allowed, unless they are generalized by some generalization, `let` (and `generic` bindings which will be introduced later), or the implicit generalization of the direct argument of `dynamic`. This "poly-" attribute of the dynamic type variables is propagated by substitution during type checking, by redefining the notion of the admissible substitutions:

Definition 8 (Redefined admissible substitutions)
A substitution S is admissible if

$$\forall \delta \in Dom(S),\ \forall \nu \in FV(S(\delta)),\ \nu \in \mathbb{D}$$
$$\forall \partial \in Dom(S),\ \forall \nu \in FV(S(\partial)),\ \nu \in \mathbb{P}$$

Admissible substitutions map poly-dynamic type variables to expressions without regular and non poly-dynamic type variables.

We also introduce a notation $PV(\tau)$ and $PV(e_t)$, for free poly-dynamic type variables of a type τ and an expression e_t, given by the following definitions:

$$PV(\tau) = DV(\tau) \cap \mathbb{P}$$
$$PV(e_t) = DV(e_t) \cap \mathbb{P}$$
$$PV(\sigma) = FV(\sigma) \cap \mathbb{P}$$

Instantiations of the poly-dynamic quantifications of derived dynamizers are more restricted than the `dynamic` primitive. The type variables of the instances must be generalized by some outer polymorphisms, since the derived dynamizers lack the argument generalization. To emphasize this rule, we put an additional *partial* attribute $\underline{\partial}$ to poly-dynamic quantifications of the derived dynamizers:

$$\texttt{double\_dynamic : } \forall\underline{\partial}.\partial \rightarrow \texttt{dyn}$$

The instance relationship $\leq$ of types and type schemes is as same as before, even with this partial quantification attribute:

$$\tau' \leq_S \forall V.\tau \equiv S(\tau) = \tau',\ Dom(S) \subset V\ \text{('\_' signes in } V \text{ are ignored)}$$

We introduce a notation $\underline{V}$, the partialization of the set of variables V, whose definition is given as follows. This adds the partial quantification attribute to all the poly-dynamic variables in V:

$$\underline{V} = \{\underline{\partial} \mid \partial \in V\} \cup (V \setminus \mathbb{P})$$

At this moment, this partial quantification does not play a big role actually: the poly-dynamic quantifications of the derived dynamizers must be *always partial*. But later, when we relax the instantiation restriction of the derived dynamizers, they will be able to have fully quantified poly-dynamic variables, too.

Typing rules with poly-dynamic type variables

With this distinction between poly-dynamic type variables for dynamizers and non-poly variables for the other generic values, we can build a new type system which can reject non-encapsulable dynamizations. The new typing judgments take the following form: $\Gamma, P \vdash e_t$, where P is the set of poly-dynamic type variables called *generalized poly-dynamic variable set*. This P indicates which poly-dynamic variables can exist in e_t. Its elements must be generalized some generalization rules. The instantiation rule is modified to reflect this:

$$\textbf{(pt-inst)}\ \frac{\Gamma(\texttt{x}) = \sigma, \quad \tau \leq \sigma, \quad PV(\tau) \subset P}{\Gamma, P \vdash \texttt{x}_\sigma : \tau}$$

If variable `x` has a type τ, the poly-dynamic type variables in τ must be members of P.

The generalization rules extend generalized poly-dynamic variable set P with P', and type check their generalization target expressions. The quantified poly-dynamic variables in V must be same as the added poly-dynamic variables P'. The poly-dynamic quantifications must be partialized in the type environment for the definitions of the derived dynamizers.

$$\textbf{(pt-let)}\ \frac{\begin{array}{c}\Gamma, P \cup P' \vdash e_{t1} : \tau_1 \quad \forall V.\tau_1 = Gen(\Gamma, e_{t1} : \tau_1) \quad P' = V \cap \mathbb{P} \\ \Gamma\{\texttt{x} : \forall\underline{V}.\tau_1\}, P \vdash e_{t2} : \tau_2\end{array}}{\Gamma, P \vdash \texttt{let x}_{\underline{V}}\ \texttt{=} e_{t1}\ \texttt{in}\ e_{t2} : \tau_2}$$

$$\textbf{(pt-letrec)}\ \frac{\Gamma\{\mathtt{f}:\tau_1\}, P\cup P' \vdash (\mathtt{fun\ x} \to e_{t1}):\tau_1 \quad \forall V.\tau_1 = Gen(\Gamma, (\mathtt{fun\ x} \to e_{t1}):\tau_1) \\ P' = V \cap \mathbb{P} \quad \Gamma\{\mathtt{f}:\forall \underline{V}.\tau_1\}, P \vdash e_{t2}:\tau_2}{\Gamma, P \vdash \mathtt{let\ rec\ f}_{\underline{V}} = \{\mathtt{f}_{\tau_1} \Rightarrow \mathtt{f}_{\forall \underline{V}.\tau_1}\}((\mathtt{fun\ x} \to e_{t1}):\tau_1)\ \mathtt{in}\ e_{t2}:\tau_2}$$

We have another generalization rule for the application of the `dynamic` generic primitive. In this time, the quantified type variables of the generalized type scheme of the argument e_t must be as same as the extended set P':

$$\textbf{(pt-dynamic)}\ \frac{\Gamma, P\cup P' \vdash e_t:\tau \quad \forall P'.\tau = Gen(\Gamma, e_t:\tau) \quad P' \subset \mathbb{P}}{\Gamma, P \vdash ((\mathtt{dynamic}_{\forall \partial.\partial\ \to\ \mathtt{dyn},\ P'}:\tau \to \mathtt{dyn})\ (e_t:\tau)):\mathtt{dyn}}$$

An instance of the `dynamic` primitive takes an additional type annotation P', the generalized variables of its argument, like the generalized variables annotations of `let` pattern variables. Like the let bindings, this generalized poly-dynamic variables bind their occurrences in e_t:

$$PV((\mathtt{dynamic}_{\forall \partial.\partial\ \to\ \mathtt{dyn},\ P}:\tau \to \mathtt{dyn})\ (e_t:\tau)) \quad = \quad PV(\tau) \setminus P$$

The other typing rules are in Figure 4.2. The rules for constants and generic primitives restrict poly-dynamic variables within their generalized poly-dynamic variable sets like (pt-inst). The other typing rules are same as before, just with the additions of generalized poly-dynamic variable sets.

$$\textbf{(pt-cnst)}\ \frac{\tau \le \sigma \quad PV(\tau) \subset P}{\Gamma, P \vdash c_\sigma:\tau} \qquad \textbf{(pt-gen)}\ \frac{\tau \le \sigma \quad PV(\tau) \subset P}{\Gamma, P \vdash \mathtt{g}_\sigma:\tau}$$

$$\textbf{(pt-fun)}\ \frac{\Gamma\{\mathtt{x}:\tau_1\}, P \vdash e_t:\tau_2}{\Gamma, P \vdash \mathtt{fun\ x} \to e_t:\tau_1 \to \tau_2} \qquad \textbf{(pt-tpl)}\ \frac{\Gamma, P \vdash e_{t1}:\tau_1 \quad \Gamma, P \vdash e_{t2}:\tau_2}{\Gamma, P \vdash (e_{t1}, e_{t2}):\tau_1 * \tau_2}$$

$$\textbf{(pt-app)}\ \frac{\Gamma, P \vdash e_{t1}:\tau_1 \to \tau_2 \quad \Gamma, P \vdash e_{t2}:\tau_1}{\Gamma, P \vdash e_{t1}\ e_{t2}:\tau_2}$$

Figure 4.2: The other typing rules

A program e_t is well-typed if $\emptyset, \emptyset \vdash e_t$ holds: all the poly-dynamic type variables introduced in generalized poly-dynamic variable set must be eliminated by generalization rules.

Now let's see how this type system provides the type safety of the polymorphic dynamizations: the expression `dynamic (fun x` $\to$ `x)` has the following typing, which checks that the arguments can be generalizable to $\forall \partial.\partial \to \partial$. This proves that `fun x` $\to$ `x` is safely dynamizable as `(fun x` $\to$ `x,`$\forall \alpha.\alpha \to \alpha$`)` :

$$\frac{\emptyset, \{\partial\} \vdash \mathtt{fun\ x} \to \mathtt{x}:\partial \to \partial \quad \forall \partial.\partial \to \partial = Gen(\emptyset, \mathtt{fun\ x} \to \mathtt{x}:\partial \to \partial)}{\emptyset, \emptyset \vdash \mathtt{dynamic}\ (\mathtt{fun\ x} \to \mathtt{x}):\mathtt{dyn}}$$

On the other hand, if the argument of `dynamic` cannot be fully generalized, (pt-dynamic) cannot eliminate the ungeneralizable poly-dynamic variables from the set P. For example:

$$\cfrac{\cfrac{\{\mathtt{x}:\partial\},\{\partial\}\vdash(\mathtt{x},\mathtt{x}):\partial*\partial \quad \partial*\partial = Gen(\{\mathtt{x}:\partial\},(\mathtt{x},\mathtt{x}):\partial*\partial)}{\{\mathtt{x}:\partial\},\{\partial\}\vdash \mathtt{dynamic}\ (\mathtt{x},\mathtt{x}):\mathtt{dyn}}}{\emptyset,\{\partial\}\vdash(\mathtt{fun\ x}\rightarrow\mathtt{dynamic}\ (\mathtt{x},\mathtt{x})):\partial\rightarrow\mathtt{dyn}}$$

In the above, the variable ∂ is not generalizable because it appears in the type environment $\{\mathtt{x}:\partial\}$. It remains in the generalized poly-dynamic variable set and must be eliminated by some generalization, in order to make this expression as a part of well-typed program. This is done by using the expression as a definition of a derived dynamizer:

$$\cfrac{\begin{array}{c}\emptyset,\{\partial\}\vdash(\mathtt{fun\ x}\rightarrow\mathtt{dynamic}\ (\mathtt{x},\mathtt{x})):\partial\rightarrow\mathtt{dyn}\\ \forall\partial.\partial\rightarrow\mathtt{dyn}=Gen(\emptyset,(\mathtt{fun\ x}\rightarrow\mathtt{dynamic}\ (\mathtt{x},\mathtt{x})):\partial\rightarrow\mathtt{dyn})\\ \{\mathtt{double\_dynamic}:\forall\underline{\partial}.\partial\rightarrow\mathtt{dyn}\},\emptyset\vdash e_t:\tau\end{array}}{\emptyset,\emptyset\vdash\mathtt{let\ double\_dynamic=fun\ x}\rightarrow\mathtt{dynamic}\ (\mathtt{x},\mathtt{x})\ \mathtt{in}\ e_t:\tau}$$

By the restriction of (pt-inst), the partially quantified poly-dynamic type variable of `double_dynamic` must be instantiated to closed types, unless it is used to define another derived dynamizer. For instance the following application of (pt-app) rule is valid only for closed types τ:

$$\cfrac{\begin{array}{c}\{\mathtt{double\_dynamic}:\forall\underline{\partial}.\partial\rightarrow\mathtt{dyn}\},\emptyset\vdash\mathtt{double\_dynamic}:\tau\rightarrow\mathtt{dyn}\\ \{\mathtt{double\_dynamic}:\forall\underline{\partial}.\partial\rightarrow\mathtt{dyn}\},\emptyset\vdash\mathtt{fun\ x}\rightarrow\mathtt{x}:\tau\rightarrow\tau\end{array}}{\{\mathtt{double\_dynamic}:\forall\underline{\partial}.\partial\rightarrow\mathtt{dyn}\},\emptyset\vdash(\mathtt{double\_dynamic}\ (\mathtt{fun\ x}\rightarrow\mathtt{x})):\mathtt{dyn}}$$

4.4.3 Type safety of the polymorphic dynamization

We here try to prove the type safety of the polymorphic dynamization under the type system defined in the previous section. Suppose we have a well-typed program and inside its typing we have the following use of (pt-dynamic):

$$\cfrac{\Gamma,P\cup P'\vdash e_t:\tau \quad \forall P'.\tau=Gen(\Gamma,e_t:\tau) \quad P'\subset\mathbb{P}}{\Gamma,P\vdash(\mathtt{dynamic}_{\forall\partial.\partial\rightarrow\mathtt{dyn}}:\tau\rightarrow\mathtt{dyn})\ (e_t:\tau):\mathtt{dyn}}$$

the safety of the encapsulation of the poly-dynamic variables of P' are clear, since they are generalized in the rule. The uneliminated poly-dynamic variables of P must be all generalized at the definition of derived dynamizers, by (pt-let) or (pt-letrec), and at the evaluation of the expression they must be instantiated to closed types, since their instantiations are restricted by (pt-inst). Before the formal proof, we need a substitution lemma of the new typing system and make the evaluation system up-to-date.

Lemma 11 (Substitution lemma of typing) *Let Γ be a typing environment, P_0 and P be distinct sets of poly-dynamic type variables, e_t be a fully type-annotated GML expression and S be a substitution where $Dom(S)\cap\mathbb{P}=P$ and $\forall\nu\in Dom(S),PV(S(\nu))=\emptyset$.*

Suppose e_t has a typing $\Gamma, P_0 + P \vdash e_t$. Then,

$$S(\Gamma), P_0 \vdash S(e_t)$$

PROOF: Proof is omitted. See the corresponding lemma in the generic constraint type system, Lemma 32. □

Lemma 12 *If an expression e_t has a typing $\Gamma, P \vdash e_t : \tau$, $PV(\tau) \subset P$.*

PROOF: It is clear by induction on the typing derivation. □

Evaluation system update

We update the evaluation system so that they can also have poly-dynamic variables in their type annotations. Some of the semantic typing rules previously defined in Section 2.5.5 are also to be extended as follows:

$$\textsf{(s-clos)}\ \frac{\exists\Gamma,\quad E \in \Gamma \quad \Gamma, \emptyset \vdash \texttt{fun x} \rightarrow e_t : \tau_1 \rightarrow \tau_2}{clos(E, \_, \texttt{x}, e_t) \in \tau_1 \rightarrow \tau_2}$$

$$\textsf{(s-clos-r)}\ \frac{\begin{array}{c}\exists\Gamma,\quad E \in \Gamma \quad \Gamma\{\texttt{f} : \forall V.\tau_1 \rightarrow \tau_2\}, \emptyset \vdash \texttt{fun x} \rightarrow e_t : \tau_1 \rightarrow \tau_2 \\ \forall V.\tau_1 \rightarrow \tau_2 = Gen(\Gamma, \texttt{fun x} \rightarrow e_t) \quad V \cap \mathbb{D} = \emptyset\end{array}}{clos(E, \texttt{f}, \texttt{x}, e_t) \in \forall V.\tau_1 \rightarrow \tau_2}$$

$$\textsf{(s-gclos)}\ \frac{\exists\Gamma,\quad E \in \Gamma \quad \Gamma, \underline{V} \vdash e_t : \tau \quad \forall V.\tau = Gen(\Gamma, e_t : \tau) \quad V \cap \mathbb{D} \neq \emptyset}{gclos(E, \_, V, e_t : \tau) \in \forall \underline{V}.\tau}$$

$$\textsf{(s-gclos-r)}\ \frac{\exists\Gamma,\quad E \in \Gamma \quad \Gamma\{\texttt{x} : \forall V.\tau\}, \underline{V} \vdash e_t : \tau \quad \forall V.\tau = Gen(\Gamma, e_t : \tau) \quad V \cap \mathbb{D} \neq \emptyset}{gclos(E, \texttt{x}, V, e_t : \tau) \in \forall \underline{V}.\tau}$$

The substitution lemma of the semantic typing, Lemma 8 is also expanded for this new evaluation system:

Lemma 13 (Substitution lemma of semantic typing) *If a semantic value sv has a semantic typing $sv \in \sigma$, then $S(sv) \in S(\sigma)$ for any substitution S. If we have an evaluation environment E such that $E \in \Gamma$ for some typing environment Γ, for any substitution S, we have $S(E) \in S(\Gamma)$.*

PROOF: The proof is almost as same as one for Lemma 8. □

The safety of the polymorphic dynamization is proven by showing that the type variables encapsulated in dynamic values are poly-dynamic type variables only which are generalized in (pt-dynamic). The other poly-dynamic type variables are instantiated to closed types in evaluation:

Theorem 14 (Safety of polymorphic dynamization) *Let e_t be a well-typed program with a typing $\Gamma, \emptyset \vdash e_t$. Let E be an evaluation environment such that $E \in \Gamma$, then in the evaluation of $E \models e_t \Longrightarrow r$, for any sub-evaluation $E' \models e'_t : \tau' \Longrightarrow r'$, we have $PV(\tau') = \emptyset$ except inside the evaluation of dynamic primitive applications* `dynamic` e'_t.

If a dynamic primitive application `dynamic` $(e'_t : \tau')$ *is evaluated, there exists a typing environment Γ' and a set of poly-dynamic type variables P such that $\Gamma', P \vdash e'_t : \tau'$ and $Gen(\Gamma', e'_t : \tau') = \forall P.\tau'$.*

PROOF: Proven by induction on the evaluation steps.

Variables Let $e_t = \mathtt{x}$ and its typing be $\Gamma, \emptyset \vdash \mathtt{x}_\sigma : \tau$ for some σ and τ. From Lemma 12, $PV(\tau) = \emptyset$.

There are two applicable rule groups for variables depending on which kind of semantic values are bound to them — (e-nvar) for regular values and (e-gvar-*) for generic closures (from the type soundness, the rules for errors are never used). If we use (e-nvar):

$$\frac{\tau \leq_S \sigma \qquad E(\mathtt{x}) = v}{E \models \mathtt{x}_\sigma : \tau \Longrightarrow S(v)}$$

We do not require any sub-evaluation therefore we proven its case.

If we use (e-gvar):

$$\frac{\tau \leq_S \forall V.\tau' \qquad E(\mathtt{x}) = gclos(E', \_, V, e'_t : \tau') \qquad E' \models S(e'_t : \tau') \Longrightarrow r}{E \models \mathtt{x}_{\forall V.\tau'} : \tau \Longrightarrow r}$$

The evaluation requires another evaluation $E' \models S(e_t : \tau') \Longrightarrow r$, where $S(\tau') = \tau$ and $PV(\tau) = \emptyset$. From the updated semantic typing rule (s-gclos), we obtain the typing for e'_t, for some type environment Γ':

$$E' \in \Gamma' \qquad \Gamma', \underline{V} \vdash e'_t : \tau' \qquad \forall V.\tau' = Gen(\Gamma', e'_t : \tau') \qquad V \cap \mathbb{D} \neq \emptyset$$

Since $Dom(S) = V$, from Lemma 11, $S(\Gamma'), \emptyset \vdash S(e'_t)$. From Lemma 13, $S(E') \in S(\Gamma')$. The correctness of the theorem for the sub-evaluation $E' \models S(e_t : \tau') \Longrightarrow r$ is clear from the induction hypothesis.

The case (e-gvar-r) is applied is also proven in the same way.

Constants and generic primitives They are proven like the case of variables.

Recursive let bindings Let $e_t = \mathtt{let\ rec\ f}_{\underline{V}} = \{\mathtt{f}_{\tau_1} \to \mathtt{f}_{\forall \underline{V}.\tau_1}\}((\mathtt{fun\ x} \to e_{t1}) : \tau_1)\ \mathtt{in}\ e_{t2} : \tau_2$, then its typing is obtained from (pt-letrec):

$$\frac{\begin{array}{c}\Gamma\{\mathtt{f} : \tau_1\}, P' \vdash (\mathtt{fun\ x} \to e_{t1}) : \tau_1 \qquad \forall V.\tau_1 = Gen(\Gamma, (\mathtt{fun\ x} \to e_{t1}) : \tau_1) \\ P' = V \cap \mathbb{P} \qquad \Gamma\{\mathtt{f} : \forall \underline{V}.\tau_1\}, \emptyset \vdash e_{t2} : \tau_2\end{array}}{\Gamma, \emptyset \vdash \mathtt{let\ rec\ f}_{\underline{V}} = \{\mathtt{f}_{\tau_1} \Rightarrow \mathtt{f}_{\forall \underline{V}.\tau_1}\}((\mathtt{fun\ x} \to e_{t1}) : \tau_1)\ \mathtt{in}\ e_{t2} : \tau_2}$$

Since τ_1 is a type for `fun x` $\rightarrow e_{t1}$, it must be a functional type and we let $\tau_1 = \tau_{11} \rightarrow \tau_{12}$. From Lemma 12, $PV(\tau_2) = \emptyset$.

Like the case of the variables we have two possibility of evaluation. If the quantified variables V contains no dynamic variable, this is a `let`-definition of non-generic value and (e-letrec) is used, otherwise the binding defines a generic value and (e-g-letrec) is applied:

$$\textsf{(e-letrec)}\ \frac{V \cap \mathbb{D} = \emptyset \qquad E\{\mathtt{f} = clos(E, \mathtt{f}, \mathtt{x}, \{\mathtt{f}_{\tau_1} \Rightarrow \mathtt{f}_{\forall \underline{V}.\tau_1}\}(e_{t1}))\} \models e_{t2} : \tau_2 \Longrightarrow r}{E \models \mathtt{let\ rec\ f}_{\underline{V}} = \{\mathtt{f}_{\tau_1} \Rightarrow \mathtt{f}_{\forall \underline{V}.\tau_1}\}(\mathtt{fun\ x} \rightarrow e_{t1})\ \mathtt{in}\ e_{t2} : \tau_2 \Longrightarrow r}$$

$$\textsf{(e-g-letrec)}\ \frac{V \cap \mathbb{D} \neq \emptyset \qquad E\{\mathtt{x} = gclos(E, \mathtt{x}, V, \{\mathtt{f}_{\tau_1} \Rightarrow \mathtt{f}_{\forall \underline{V}.\tau_1}\}(e_{t1}))\} \models e_{t2} : \tau_2 \Longrightarrow r}{E \models \mathtt{let\ rec\ x}_{\underline{V}} = \{\mathtt{f}_{\tau_1} \Rightarrow \mathtt{f}_{\forall \underline{V}.\tau_1}\}(e_{t1})\ \mathtt{in}\ e_{t2} : \tau_2 \Longrightarrow r}$$

No matter which rule is used, its subsequent evaluation is for $e_{t2} : \tau_2$, where $PV(\tau_2) = \emptyset$. The theorem is correct for this sub evaluation of e_{t2} from the induction hypothesis.

Let bindings The possible evaluation rules are (e-let) and (e-g-let). The case of (e-g-let) is proven similarly to the case of recursive let bindings, since it stores its definition as a generic closure without evaluation. In the case of (e-let), the definition expression is also evaluated, but its type does not contain dynamic variables. Therefore the theorem is also correct for the evaluation of the definition expression, using the induction hypothesis.

Applications of `dynamic` primitive This is the exceptional case of the theorem. The correctness of the case is clear from its typing:

$$\frac{\Gamma, P \vdash e'_t : \tau' \qquad \forall P.\tau' = Gen(\Gamma, e'_t : \tau') \qquad P \subset \mathbb{P}}{\Gamma, \emptyset \vdash ((\mathtt{dynamic}_{\forall \partial.\partial \rightarrow \mathtt{dyn}} : \tau' \rightarrow \mathtt{dyn})\ (e'_t : \tau')) : \mathtt{dyn}}$$

The others Their corresponding typing rules do not modify generalized poly-dynamic type variable sets.

□

4.4.4 Type reconstruction in two phases

Our new type system introduced a new restriction: the poly-dynamic type variables must be generalized somehow, by let binding or `dynamic` argument generalizations. Under this restriction, dynamic value related expressions may lose principal typing:

```
let double_dynamic x = dynamic (x,x) in
double_dynamic [];;
```

In this example, the use of `double_dynamic` can have types like `int list` $\rightarrow$ `dyn`, `string list` $\rightarrow$ `dyn`, etc..., but not ∂ `list` $\rightarrow$ `dyn`, which could be the principal typing of `double_dynamic` in the type system without the restriction.

Type reconstruction without the principal typing property always faces to ambiguity problems. Expressions can have more than one possible typings which are incompatible each other. Usually type checkers just reject these ambiguities as typing errors or choose one of the typings as default. We do the same: first we define a reconstruction algorithm which rejects the ambiguity dynamizers as type errors.

Our type reconstruction consists of two phases. The first phase is the usual ML type reconstruction with the poly-dynamic variables. The result of this phase may contain illegal dynamizer instances which have unencapsulable poly-type variables.

The second phase is the handling of these unencapsulable type variables. At first we just reject them as type errors. But later we propose an automatic ambiguity resolution using the abstract type `dyn` (Section 4.4.5).

First phase

The algorithm of the first phase W_1 whose definition is in Figure 4.3 is obtained as a simple extension of the original W (Figure 7.2 of Section 2.6). First, to propagate the attribute of poly-dynamic type variables by unification, we define a function *polydyn*. Its definition is like *dynamize* function:

$$\begin{array}{rcl}
\mathit{polydyn}(\mathit{base}) & = & \emptyset \\
\mathit{polydyn}(\alpha) & = & \{\alpha \Rightarrow \partial\} \text{ where } \partial \text{ is a fresh poly-dynamic type variable} \\
\mathit{polydyn}(\delta) & = & \{\delta \Rightarrow \partial\} \text{ where } \partial \text{ is a fresh poly-dynamic type variable} \\
\mathit{polydyn}(\partial) & = & \emptyset \\
\mathit{polydyn}(\tau_1 \rightarrow \tau_2) & = & \text{let } S_1 = \mathit{polydyn}(\tau_1),\ S_2 = \mathit{polydyn}(S_1(\tau_1)) \text{ in } S_2 \circ S_1 \\
\mathit{polydyn}(\tau_1 * \tau_2) & = & \text{let } S_1 = \mathit{polydyn}(\tau_1),\ S_2 = \mathit{polydyn}(S_1(\tau_1)) \text{ in } S_2 \circ S_1
\end{array}$$

The unification algorithm *unify* is extended for the case of poly-dynamic type variable, using *polydyn* function:

$$\mathit{unify}(\partial, \tau) \quad = \quad S \circ \{\partial \Rightarrow S(\tau)\} \text{ when } \partial \notin FV(\tau), \text{ where } S = \mathit{polydyn}(\tau)$$

The new case for the `dynamic` primitive is added to the algorithm W_1. The type of argument of `dynamic` is poly-dynamized by unifying with ∂, then generalized. The generalized poly-dynamic variables is attached to the type annotation of `dynamic`. The cases for let bindings are slightly modified so that they partially quantify poly-dynamic variables of the derived dynamizers. The other cases are kept as they are.

$$
\begin{aligned}
W_1(\Gamma, \texttt{dynamic}\ e) \quad = \quad & \text{let } e_t : \tau = W_1(\Gamma, e)\\
& \text{let } S = \mathit{unify}(\partial, \tau) \text{ where } \partial \text{ is fresh}\\
& \text{let } \sigma = \mathit{Gen}(S(\Gamma), S(e_t))\\
& \text{in } ((\texttt{dynamic}_{\forall\partial.\partial\ \to\ \texttt{dyn},\ BPV(\sigma)} : S(\tau) \to \texttt{dyn})\ S(e_t)) : \texttt{dyn},\ S
\end{aligned}
$$

$$
\begin{aligned}
W_1(\Gamma, \texttt{let x} = e_1 \texttt{ in } e_2) \quad = \quad & \text{let } e_{t1} : \tau_1,\ S_1 = W_1(\Gamma,\ e_1)\\
& \text{let } \forall V.\tau_1 = \mathit{Gen}(S_1(\Gamma), (e_{t1} : \tau_1))\\
& \text{let } e_{t2} : \tau_2,\ S_2 = W_1(S_1(\Gamma)\{\texttt{x} : \forall \underline{V}.\tau_1\},\ e_2)\\
& \text{in } \texttt{let x}_{\underline{V}} = S_2(e_{t1}) \texttt{ in } e_{t2} : \tau_2,\ S_2 \circ S_1
\end{aligned}
$$

$$
\begin{aligned}
& W_1(\Gamma, \texttt{let rec f} = \texttt{fun x} \to e_1 \texttt{ in } e_2) =\\
& \qquad \text{let } \alpha \text{ be a fresh variable}\\
& \qquad \text{let } (\texttt{fun x} \to e_{t1}) : \tau_1,\ S_1 = W_1(\Gamma\{\texttt{f} : \alpha\},\ \texttt{fun x} \to e_1)\\
& \qquad \text{let } S_1' = \mathit{unify}(\tau_1, S_1(\alpha))\\
& \qquad \text{let } \forall V.\tau_1' = \mathit{Gen}((S_1' \circ S_1)(\Gamma), S_1'(\texttt{fun x} \to e_{t1} : \tau_1))\\
& \qquad \text{let } e_{t2} : \tau_2,\ S_2 = W_1(((S_1' \circ S_1)(\Gamma))\{\texttt{f} : \forall \underline{V}.\tau_1'\},\ e_2)\\
& \qquad \text{in } \ \texttt{let rec f}_{\underline{V}} = (S_2 \circ S_1')(\texttt{fun x} \to e_{t1})\\
& \qquad \qquad \texttt{in } e_{t2} : \tau_2,\ S_2 \circ S_1' \circ S_1
\end{aligned}
$$

For the other cases, $W_1(\Gamma, e)$ is obtained from $W(\Gamma, e)$, replacing all the recursive occurrences of W by W_1.

Figure 4.3: Type reconstruction algorithm for the polymorphic dynamization

Second phase

Once the fully type-annotated program e_t is obtained from the untyped source e as $e_t, S = W_1(\emptyset, e)$, The second reconstruction W_2 checks the existence of the free poly-dynamic type variables. If it is not an empty set, we reject e_t as ill-typed:

$$W_2(e_t) = \ \text{if } PV(e_t) = \emptyset \text{ then } e_t \text{ else reject } e_t \text{ as a type error}$$

For better user interface, the type checker should point out the problematic dynamizer in its error message:

```
# dynamic (ref []);;
  ^^^^^^^
Error: dynamizer dynamic captures type _α list ref,
  which contains a non-generalizable type variable.
```

Algorithm correctness

Now we prove the correctness and partial completeness of the algorithm. They can be proven in the same way as we have done in Section 2.6:

Theorem 15 (Type reconstruction correctness) *The algorithm W_1 is correct with respect to the typing rules. Let Γ be a typing environment and e be a bare GML expression and if $W_1(\Gamma, e) = e_t,\ S$, then $S(\Gamma), PV(e_t) \vdash e_t$.*

PROOF: (Sketch) Proven by induction over the structure of bare GML expression, see the proof of the Theorem 9 for details. □

From Theorem 15, we obtain the following immediately:

Theorem 16 (Type reconstruction correctness for programs) *The algorithm W_1 and W_2 are correct with respect to the typing rules. Let e be a bare GML program and if $W_1(\emptyset, e) = e_t,\ S$ and $W_2(e_t)$ does not reject e_t, then $\emptyset, \emptyset \vdash e_t$.*

PROOF: From $W_2(e_t) = e_t$, we have $PV(e_t) = \emptyset$. Using Theorem 15, $\emptyset, \emptyset \vdash e_t$. □

The type reconstruction cannot be complete, since the new type system lacks principal typing. But we can prove a weak form of completeness:

Theorem 17 (Type reconstruction weak completeness) *Let Γ be a typing environment, $e'_t : \tau'$ be a GML expression, and V be a set of poly-dynamic type variables. If there exists a typing $S'(\Gamma), V \vdash e'_t : \tau'$ where $Dom(S') \subset FV(\Gamma)$, then $W_1(\Gamma, strip(e'_t)) = e_t : \tau,\ S$ and there exists a substitution S'' such that*

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

In the latter statement, $\mathit{fresh}(W_1(\Gamma, \mathit{strip}(e'_t)))$ denotes the set of newly introduced type variables in W_1.

PROOF: (Sketch) By induction over the structure of the bare GML expression. The proof follows the same outline of the one of Theorem 17. The poly-dynamic variable set V does not play any important role in the proof. □

With the second phase W_2, Theorem 17 is rewritten as follows for programs:

Theorem 18 (Type reconstruction weak completeness for programs) *Let $e'_t : \tau'$ be a GML expression and $\emptyset, \emptyset \vdash e'_t : \tau'$ be its typing. If $W_1(\emptyset, strip(e'_t)) = e_t : \tau, S$ and $W_2(e_t)$ does not reject e_t, there is a substitution S' such that $S'(\tau) = \tau'$.*

PROOF: This is a special case of Theorem 17. □

Theorem 18 indicates that when W_1 and W_2 succeed, they always return the principal typings.

4.4.5 `dyn`-unification

The reconstruction algorithm W_1 may return too general and invalid typing which contains unencapsulable poly-dynamic type variables. We can easily obtain valid typing from them, by unifying these unencapsulable type variables with some closed type expressions. However, W_2 does not try it but rejects these type variables as typing errors. In some sense it is a correct behavior, since such expressions lack principal typings. Since possible typings are incompatible each other, the expression typing is ambiguous: even if the type reconstruction selects one of them, it may be completely different from the programmer's intention. For example, to be well-typed in the following expression

```
let x = ref [] in dynamic x;;
```

`x` must have a type expression τ `list ref` for some close type τ. τ can be `unit`, `int` , etc.... None of them have stronger reason to be chosen as the unification candidate than others; once unified, `x` becomes monomorphic and we cannot use it in the other types.

There is an exception. The abstract type `dyn`. Of course even this `dyn` cannot prevent `x` from being monomorphic, but `dyn` works like a polymorphic type variable, thanks to its pseudo heterogeneousness. Any type can be hidden and encapsulated into this type using the `dynamic` primitive:

```
# let x = ref [] in dynamic x;;
- : dyn
# x;; (* x's type variable is unified automatically with dyn *)
- : dyn list ref = {contents = []}
# x := dynamic 1 :: !x;;
# x := dynamic "hello" :: !x;;
...
```

We can regard the type `dyn` is the most "general" type of the closed type expressions. From this generality, we can use it as the type of the auto ambiguous resolution. We call this *`dyn`-unification* of unencapsulable type variables. We will also use this `dyn`-unification later, in the final ambiguity resolution of user-defined generic values (Section 10.4.1).

Once the fully type-annotated expression e_t is reconstructed by the algorithm W_1, the new second phase W'_2 unifies the free occurrences of poly-dynamic type variables with with `dyn`:

$$W'_2(e_t) = S(e_t) \text{ where } S = \{(\partial \Rightarrow \texttt{dyn}) \mid \forall \partial, \partial \in PV(e_t)\}$$

The well-typedness of $S(e_t)$ is preserved thanks to Theorem 3. Since W'_2 never rejects expressions, now the type reconstruction algorithm W_1 and W'_2 always succeed if given expressions are typable.

Printing warnings

In spite of the generality of the type `dyn`, the auto unification of unencapsulable type variables may produce incompatible typing from the user intention and be confusing. To have a better interface, the type reconstruction algorithm should report the `dyn`-unifications as warnings:

```
let x = ref [] in dynamic x;;
Warning: the unencapsulable type variable of x is unified with dyn
- : dyn
```

4.5 Polymorphic coercion

The implementation of coercion as the generic primitive function `coerce` was simple, but it has a critical problem: as far as it is a generic primitive, the coercion of the dynamic values must take a form of application `coerce` e_t, which is expansive and therefore not generalizable under the value polymorphism restriction[Wri95]. This means that the polymorphic contents of dynamic values cannot be coerced polymorphically. For example, the dynamic value of the polymorphic identity function is coerced to just a monomorphic function:

```
# let d = dynamic (fun x → x);;
d : dyn = <abstr> (* = (<fun>, ∀α.α → α) *)
# let f = (coerce d : α → α);;
f : _α → _α = <fun>
```

To coerce back polymorphic contents to polymorphic values, we need to make the application `coerce` e_t exceptionally generalizable, or have a new *coerce expression* which looks like the pattern match expression:

$$\texttt{coerce}\ e_t\ \texttt{with}\ \texttt{x}{:}\sigma\ \Rightarrow\ e'_t$$

A `coerce` expression `coerce` e_t `with x:`$\sigma \Rightarrow e'_t$ takes a dynamic value e_t and checks whether its content can be coerced to the explicitly written type scheme σ, which is closed and without dynamic type variables. If it is, the content of the dynamic value is bound to the pattern variable `x` and e'_t is evaluated. Otherwise it raises a run time exception. The typing rule for this coerce expression is straightforward: the value to be coerced must be a dynamic value, and the result of the whole expression is the type of the e'_t:

$$\frac{\Gamma \vdash e_t : \texttt{dyn} \quad \Gamma\{\texttt{x} : \sigma\} \vdash e'_t : \tau \quad FV(\sigma) = \emptyset \quad BDV(\sigma) = \emptyset}{\Gamma \vdash \texttt{coerce}\ e_t\ \texttt{with}\ \texttt{x} : \sigma \Rightarrow e'_t : \tau}$$

The compilation of the coerce expression is almost same as the original `coerce` generic primitive:

$$\lfloor \texttt{coerce}\ e_t\ \texttt{with}\ \texttt{x} : \forall V'.\tau \Rightarrow e'_t \rfloor_V = \begin{array}{l}\texttt{let (v,ty) = } e_t \texttt{ in} \\ \texttt{if is\_instance } \langle\!\langle\tau\rangle\!\rangle \texttt{ ty then} \\ \quad \texttt{(fun x} \rightarrow \lfloor e'_t \rfloor_V\texttt{) v} \\ \texttt{else raise Coercion\_failure}\end{array}$$

The `coerce` expression provides polymorphic coercion since the content which is bound to the pattern variable `x` has a possibly polymorphic type scheme σ. The following code which uses the content polymorphically is well-typed and correctly executed:

```
coerce (dynamic (fun x → x)) with f:∀α.α → α ⇒ f 1; f true
```

Combining pattern match expression, this `coerce` expression can be extended to have multiple cases with more complex patterns, like `typecase` expression in [ACPP91]:

```
coerce e_t1,...,e_tn with
| (x_11 : τ_11),..., (x_1n : τ_1n) → e'_t1
| ...
| (x_m1 : τ_m1),..., (x_mn : τ_mn) → e'_tm
```

4.6 Future work: relaxing the restriction

We studied the type checking to ensure the safety of polymorphic dynamization, but our typing may be too restricted in practice: the derived dynamizers cannot create polymorphic encapsulation. This restriction should be relaxed. In other words, it is to turn partial quantifications $\forall\underline{\partial}$ of derived dynamizers to fully quantified ones $\forall\partial$ if they are proven safe. For example, `double_dynamic` function should be able to have the following type scheme with a full quantification of the poly-dynamic variable ∂:

```
double_dynamic : ∀∂.∂ → dyn
```

The application of this function yields a generalization check of its argument, just like the `dynamic` primitive. If the argument can be generalized, it is dynamized with a polymorphic type scheme. For example,

```
# double_dynamic (fun x → x);;
```

will create a dynamic value of `∀∂.(∂ → ∂) * (∂ → ∂)`.

However, not all the derived dynamizers can have full poly-dynamic quantifications. For example:

```
let apply_and_dynamize f =
  let v = f () in
  dynamic v
;;
```

Suppose we have an application `apply_and_dynamize (fun () → ref [])` and its argument is generalizable locally as `∀∂.unit → ∂ list ref`. What `apply_and_dynamize` actually dynamize is not the function itself, but the result of the application `(fun () → ref []) () ⇝ ref []`

whose type is ∂ `list ref`. Though the same ∂ type variable was regarded as encapsulable at the generalization of the function, it is now clearly ungeneralizable and non-encapsulable at `ref []`. Therefore we cannot give a full quantification to `apply_and_dynamize` and it must have only a partially quantified type scheme $\forall\underline{\partial}.(\mathtt{unit} \to \partial) \to \mathtt{dyn}$.

These examples indicate us that to allow derived dynamizers polymorphic encapsulation, we need a more complicated type system than before. The type generalization of derived dynamizers must investigate their definitions to discover whether we can safely give full poly-dynamic quantifications. This check will go as follows: once we find a let definition defines a derived dynamizer, we type check the definition again, temporarily quantify its arguments' poly-dynamic type variables. This generalization can be justified from the typing rules of dynamizers, that is, they must take arguments with closed type schemes. In this temporary typing, if an internal dynamizer is applied to an arguments with non-generalizable poly-dynamic variables, the corresponding poly-dynamic variables in the original typing cannot gain full-quantifications. Let's look a simple example:

```
let d = fun f g →
  let v = f () in
  let v' = g () in
  dynamic (v, g)
;;
```

The definition of `d` is typed as $(\mathtt{unit} \to \partial_1) \to (\mathtt{unit} \to \partial_2) \to \mathtt{dyn}$. Before generalize it, we type the functional body again, but with the modified bindings $\{\mathtt{f} : \forall\partial_1.\mathtt{unit} \to \partial_1, \mathtt{g} : \forall\partial_2.\mathtt{unit} \to \partial_2\}$ in the typing environment. Thanks to these quantifications, `v` and `v'` have ∂_1' and ∂_2', different type variables from ∂_1 and ∂_2. Both types cannot be generalized since their definitions are expansive. At the call of the `dynamic` primitive, we find that `v` must not be polymorphically encapsulated but `g` can be. This indicates that in the original typing, ∂_2 can be full-quantified, but ∂_1 cannot: the type scheme of `d` is $\forall\underline{\partial_1}\partial_2.(\mathtt{unit} \to \partial_1) \to (\mathtt{unit} \to \partial_2) \to \mathtt{dyn}$.

This full retyping of definitions of derived dynamizer will surely inflate the complexity of the typing. Further studies are required to find a simple solution, even if it might be less powerful than the full retyping.

4.7 Dynamics and constraint typing system

To distinguish the dynamic type variables related to dynamizations from the others, we introduced the poly-dynamic attribute ∂ to them. When the generic constraint type system will be introduced in Chapter 8, this poly-dynamic attribute can be easily transfered into generic constraints and we can make the type algebra simpler, using the same scheme proposed in Section 8.11.

This main idea of Section 8.11 is, roughly speaking, adding a new derived constraint $\mathtt{generic}(\alpha)$ to the constraints of generic primitives instead of carrying the dynamic attribute of type schemes. For example, a generic primitive `print` of a constraint $[\forall\delta.\delta \to \mathtt{unit}]$ will have the following new constraint:

$$\mathcal{K}_{\mathtt{print}} \quad = \quad [\forall\alpha.\alpha \to \mathtt{unit}\ \{\mathtt{generic}(\alpha)\}]$$

By adding the constraint $\texttt{generic}(\alpha)$, the type variable α is regarded as generic and the constraint is distinguished from a constraint $[\forall\alpha.\alpha \rightarrow \texttt{unit}]$ for a non-generic value.

Like the attribute $\texttt{generic}(\alpha)$, we can transfer the poly-dynamic attribute of the type scheme of the `dynamic` primitive into its constraint as a new $\texttt{poly}(\alpha)$ constraint:

$$\mathcal{K}_{\texttt{dynamic}} \quad = \quad [\forall\alpha.\alpha \rightarrow \texttt{dyn}\ \{\texttt{poly}(\alpha)\}]$$

The poly-dynamic attributes of derived dynamizers can be deduced from their derived constraints on the `dynamic` primitive. For example, `double_dynamic` will have the following generic constraint:

$$\texttt{double\_dynamic : } [\forall\alpha.\alpha \rightarrow \texttt{dyn}\ \{\ \alpha * \alpha \rightarrow \texttt{dyn} <: \mathcal{K}_{\texttt{dynamic}}\}]$$

which corresponds with its current type scheme

$$\forall\underline{\partial}.\partial \rightarrow \texttt{dyn}$$

4.8 Conclusion

As an example of generic primitives, we have implemented the primitives of dynamic values over extensional polymorphism. Using the type dispatching mechanism of extensional polymorphism compilation, the compilations of the primitives are straightforwardly obtained as functions which take type information they need as run time type arguments. More than a mere example of generic primitives, dynamic values provide a simple run time type checking. The user-defined generic values we will introduce in Chapter 7 also perform type operations, but these operations will be limited to something simulatable and verifiable even at type checking phase. At this point, dynamic values are more dynamic than user-defined generics. Dynamic value framework can be used to overcome problems of user-defined generics which come from their staticness.

The latter half of this chapter, we argued polymorphic encapsulation of dynamic values. Since type variables inside dynamic values can be exported outside of their scope, they must be generalizable for type security. To encapsulate values with polymorphic types, we added a new typing of the `dynamic` primitive to check the generalization of its argument. Derived dynamizers, derived generic polymorphic functions which create dynamic values, are also easily defined in extensional polymorphism. We found, however, some of derived dynamizers cannot create polymorphic encapsulation, even when their arguments can be generalizable. Therefore we added another typing to inhibit all the derived dynamizer to create polymorphic encapsulation. This is a limitation of our system for simplicity. It is a future issue to give the polymorphic encapsulation capability also to derived dynamizers, if they are proven safe. We also needed a modification to the `coerce` primitive: to coerce back polymorphic contents of dynamic values to polymorphic, the coercion must be a special form of expression.

Dynamic values are exportation of types by encapsulating them inside of values. Therefore type information of dynamic values must be independent of any typing context. The type generalization check of the `dynamic` primitive assures the universality of dynamic type information inside *one* program. In following chapters (Chapter 5 and 6) we will consider universal data type identification throughout ML programs to build a type safe system of ML value exchange even between different programs.

Chapter 5

Type safe value I/O

5.1 Introduction

Type safe value input and output using extensional polymorphism

All operating systems provide basic input/output facilities that only manipulate elementary pieces of information, namely bytes or characters. Using this simple I/O's, high-level programming languages provide read/write accesses to various devices, including files, screens and keyboards. Usually, the system I/O's are directly reflected into the programmer's interface: the input/output abstractions just handle characters, or at best provide direct input/output of elementary data such as integers or strings. This is far from being satisfactory: in modern programming languages, the programmer can easily define complex data structures; surprisingly, there is no help from the language to input/output those data. The programmer has to write his own specific routines, namely a conversion function from structured data to a stream of characters at output, and the converse conversion from characters to structured data at input. This is a large, complex, and error prone effort. It is also basically not reusable: this code is specific to the data type at hand.

So we claim that the language should also provide a high-level input/output interface that handles directly the complex data that the programmer manipulates. We name "value I/O" such an advanced input/output facility. This new kind of I/O rises new kind of problems, notably safety problems, since a structured value written to a file must be read back as a value of the same type. Or at least of a type somewhat "related" to its original type. Hence, we need to know what is the type of a value when we read it back.

We study here the way we must output a structured value, such as to ensure that the value can be safely read back. In our type safe value I/O system combining an untyped ML value marshaler (Objective Caml unsafe value I/O interface) and the primitives for dynamic values, output values are written with some type information and this type information is used at input to check if the type expected by the type context is safely acceptable.

To exchange values between two or more programs that may not share the same data type definitions, the type information has to be written in some canonical way that can be understood by any program. The simple run time types which attached with the values in dynamic values in the previous chapter are not appropriate for this purpose, since the data types in the run time types are identified just by their names. Such identification works only inside one program, or between different invocation of the some identical program. It is no longer correct between different programs. Therefore, we modify the design of the run time types so that it works correctly through different programs. The new run time types, called *external types* does not contain names of data types, but their digests of data types. The data type digests are encrypted check-sums constructed from type definitions without names in Objective Caml; the names in type definitions do not affect internal value representation of data types. This defines an interesting notion of isomorphism between data type definitions, that captures the notion of being equivalent with respect to external value representation.

Outline of this chapter

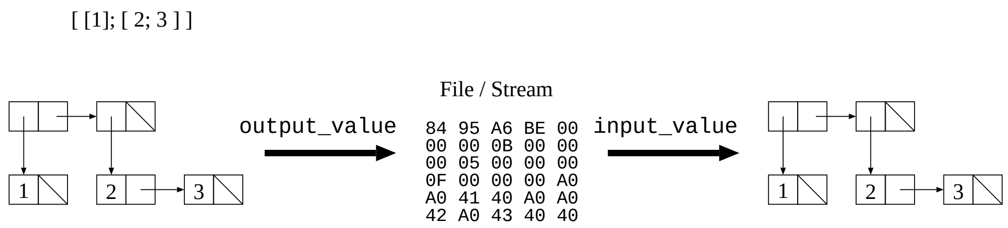
Section 5.2 introduces the notion of value I/O and explains the risk of value I/O without run time type checking, and we present our strategy in Section 5.3. In Section 5.4, we define external types, special type information for value I/O which identify types universally among different programs. Finally in Section 5.5, we define the primitives for value I/O by combining external types and the mechanism of value-type pair encapsulation and decapsulation of dynamic values.

5.2 Basic unsafe value I/O

Objective Caml provides an unsafe value I/O facility, by means of two primitives with the following profile:

$$\begin{array}{ll} \texttt{output\_value} : & \forall\alpha.\texttt{out\_channel} \to \alpha \to \texttt{unit} \\ \texttt{input\_value} : & \forall\alpha.\texttt{in\_channel} \to \alpha \end{array}$$

`output_value` creates a representation of an arbitrary structured value and write it to a file. Conversely, `input_value` decodes the representation and rebuilds in the memory the same structure as the original (preserving sharing).



However, when a value is loaded, there is no way to check whether it has the expected type: input and output types are unrelated, and this way of communicating is essentially untyped. So, this mechanism is unsafe: for instance an integer can be read back as a string, and this leads to a run time type error.

```
# let oc = open_out_bin "foo.dat";;
# output_value oc 1;;
...
# let ic = open_in "foo.dat";;
# print_string (input_value ic);;
```
$\Longrightarrow$ **Segmentation fault**

These primitives do not only serve raw ML data saving in files, but allow a simple kind of communication between two or more Objective Caml programs. In this case the type-checking problem is even worse, since a program may erroneously read in a value of a type that is not even defined in the local type context of the reading program. This makes communicating between two or more programs by using `output_value` and `input_value` difficult and error prone.

Our system isolates the user from those unsafe primitives, building a new typed value exchange layer on top of this untyped layer.

5.3 Our Goal

To build a reliable value communication system between different ML programs, we have to connect value I/O with type checking, in order to get a type-safe value I/O facility.

5.3.1 Tag checking at run time is sub-optimal

One of the strongest solution to this problem is to perform run time type checking of the input values by checking the tags of the structures to prove that they are legal value of the type we expect at input[1]. This value I/O with "run time tag checking" rejects all illegal inputs which may crash programs, and is yet completely compatible with the original unsafe version for all legal inputs, since the output data do not require additional run time type information. Unfortunately, this scheme implies that programs have to keep all the definitions of data types, and in addition, we have to recursively check all the tags contained in values. This is indeed inefficient for complicated data types and values, so we designed a more efficient strategy, that is compatible with fast and simple reading of output values.

5.3.2 Exporting with types

Instead of performing this complex and recursive type check of the input values, we may take advantage of the property that output values were once well-typed in writer programs. It is unnecessary to type-check input values again from scratch. When we *export* a value, we also export a witness of well-typedness: encapsulate the value with its type information called *external types* like a value-type pair of dynamic value, and write this pair outside of programs. Then we read back this external type at input, and check whether the external type matches the type expected by the

[1] IoXML[dR] is such a system implemented recently on Objective Caml. It provides printer and parser functions of data types to and from XML[W3C00] data. Lacking of special typing for auto generation of the printers and parsers, users must specify the types they want to export and import manually.

context at *import* time like coercion of dynamic values. We use the words import/export instead of input/output, because our value I/O system has a strong type checking discipline, reminiscent to an airport custom in the real world.

5.4 External types

5.4.1 External types are not run time types

In Chapter 4, we argued an implementation of the dynamic values over extensional polymorphism. Even though they provide a mechanism for encapsulation and decapsulation of values with their types which we also need here, their type information — the run time type representation — cannot directly be used for the safe value I/O which is essentially an inter-program communication, because of the data types.

The data type identification in the dynamic value coercion was done just by its name in Figure 4.1 and it works perfectly inside one program, or between different invocations of the same one program. But once we think about the communication between different ML programs, this identification is no longer valid; types in different programs are not normally identical even if they share the same name. Therefore it cannot achieve the safety of the value I/O just by combining the primitives of the dynamic values and the unsafe value I/O. Still invalid value inputs can crash programs:

Program A

```
type t = Foo | Bar

let oc = open_out_bin "foo.dat" in
output_value oc (dynamic Foo);
close_out oc;;
```

Program B

```
type t = {x : int; y : int}

let ic = open_in_bin "foo.dat" in
coerce (input_value ic) with
| v : t ⇒ print_int v.x;
          print_int v.y
;;
```

Program A output a representation of the value `Foo` to a data file, encapsulating it as a dynamic value `(Foo,t)`. Then a different program B tries to input this dynamic value as its own `t`. Since both data types in the program A and B have the same name `t`, the output data is *wrongly* granted as a value of type `t` of B and the value representation of `Foo` of Program A is used as one for a record type `t` of Program B. It is clearly unsafe and crashes B, since the internal representations of the both data types are incompatible.

Thus, we cannot use the data type identification by their names of the run time types. We can still use the same value-type encapsulation mechanism of dynamic values, but different type information with new representations of data types is required for our purposes, which provides universal identification over programs.

5.4.2 Data type identification by definitions

Data type names (paths) of run time types are not unique universal identifiers for data types. For example, if two programs P and Q are communicating, there is no guarantee that type `t` of P and `t` of Q are equivalent, since they have been defined independently. For value I/O, data types must not be identified by their names, but by their semantics, or more simply by their definitions.

Internally equivalent type definitions

On the other hand, the same data type structures may have different names and different labels or constructors, while being equivalent. For example, the following two type definitions represent the same data structure for trees, even if their names are different.

Program E:

```
type α tree =
| Leaf of α
| Branch of α tree list
```

Program F:

```
type α arbre =
| Feuille of α
| Branche of α arbre list
```

In Objective Caml, the internal representations of these two types are equal[2]. Therefore, the output value of an `int tree` is safely input as a value of type `int arbre` and vice versa. We call such types *internally equivalent* each other. For type safety of value I/O, these differences of type names can be ignored. Only structural equivalence of type definitions without names specifies compatibility of internal value representations.

Is this kind of “confusion” of data types practical for value I/O? We definitively believe so: in our experience of large scale application programming with some sort of data base, we often use the unsafe version of value I/O to save data structures in files because the value I/O get rid of the boring effort of writing printer functions for saving and parsers for loading. Sometimes, it occurs that names of types, labels, modules which defines data types, etc. are modified through application version-updates, but we can still use the data files created by the previous versions of the application, as far as the internal data representation is not modified.

In the cases where this practical confusion of types would not be preferred, we could integrate type and label names into our system without extra cost to make the value I/O more strict, so that only types with exactly the same type definitions would be equivalent. We formalize these ideas with the minimum requirements for safe value I/O, by ignoring type names.

We now formalize these ideas.

[2]Of course, this is not always true for other functional language implementations.

5.4.3 Nameless data type definitions and data type digests

In Objective Caml, using the following *data type definitions*, we can define variants and records, sums and products of types with labels, as user-defined data types:

typedefs	::=	`type` *typedef* `and` ... `and` *typedef*	Data type declarations
typedef	::=	$(\alpha_1, \ldots, \alpha_m)$ `t` = *defbody*	Type definitions
defbody	::=	C `of` $\tau \mid \ldots \mid C$ `of` τ	Variants
	$\mid$	$\{l : \tau;\ \ldots;\ l : \tau\}$	Records

Variants and records take labels C and l. A type definition takes one of these variant or record definition bodies with type variable parameters $\alpha_1, \ldots, \alpha_n$. The types τ which appears in the right hand of the equation must not contain other variables. A type declaration can consist of more than one type definitions to define mutual recursive data types. For simplicity the variants must take always one type argument like C `of` τ: the constant variant definitions without type arguments like `type t = Foo | Bar` can be encoded into this form of definition adding a dummy type argument `unit` for example: `type t = Foo of unit | Bar of unit`. We also omit the type aliases here.

We assume the following property of the Objective Caml implementation:

Assumption 1 *In Objective Caml, the following information in data type definitions never affects the representation of values.*

- *Data type names*
- *Constructor names of variants*
- *Labels names of records*

Therefore, in our system, data types are identified by their definitions without these irrelevant names, their *nameless type definitions* in short. Intuitively, data types which have the same nameless type definition are internally equivalent. Under Assumption 1, value communication between such equivalent data types never leads to run time crashes.

Nameless type definitions

nameless-typedefs	::=	`type` *typedef* `and` ... `and` *typedef*	Nameless definition
typedef	::=	$(\alpha_1, \ldots, \alpha_m)$ $\#n$ = *defbody*	
defbody	::=	_ `of` $\tau_e \mid \ldots \mid$ _ `of` τ_e	Variants
	$\mid$	$\{$_ $:\tau_e;\ \ldots;\ $_ $:\tau_e\}$	Records

External types

$$
\begin{array}{rcll}
\tau_e & ::= & \alpha & \\
& \mid & \tau_e * \tau_e & \\
& \mid & (\tau_e, \ldots, \tau_e)\ \#n & \text{Recursive index in type definitions} \\
& \mid & (\tau_e, \ldots, \tau_e)\ \mathit{digest} & \text{Data type digest}
\end{array}
$$

Nameless type definitions have the same structure as normal ML type definitions. They are just type definitions without the constructor names and record labels names. Names of types are also omitted, just replaced by an integer index $\#n$. Index $\#n$ indicates the position of the type `t` in its mutual recursively defined types. Each data type `t` is identified by a pair (l, n) where l is the list of the mutually recursive nameless type definitions where `t` appears at n-th position.

The *external types* τ_e appear in the nameless type definitions are also nameless. So the external types can represent any Caml type expression: for variables and tuples, the external types are equivalent to the normal type expressions; there is no functional types in τ_e, since there is no function encoding for value I/O[3]. We have an additional construct for recursively defined data types in type definitions. In the definitions, those types are designated with an index n. As the nameless type definitions, each data type name `t` must be replaced by something nameless in the nameless types. However, we do not use its pair of corresponding mutual recursive nameless definitions and its position index (l, n) directly, since embedding nameless definitions inside types would cause size explosion. Therefore, in the nameless types, instead of pairs of nameless definitions and a position index (l, n), we embed their *digests*. These digests are defined as *MD5 digests*[Riv92] of the string representation of the corresponding nameless definitions and indices (l, n). MD5 algorithm produces a 128-bit "fingerprint" check-sum of strings of any arbitrary length. The use of digests reduces the size of the type information radically; the size of the data type information is now constant, no matter how definitions are complex.

5.4.4 Digest calculation

The algorithm $\mathit{Digest}(\Gamma_P, \mathtt{t})$ to calculate data type digest for type `t` in the type environment Γ_P of program P is described as follows:

- If `t` is one of the primitive type like `int` and `float`, just returns its name `"int"` or `"float"` as $\mathit{Digest}(\Gamma_P, \mathtt{t})$.
- Otherwise, obtain the type definitions of `t` and the data types mutually recursively defined with `t` from the typing environment Γ_P,
- Sort the definitions to have unique data type digests. The ordering of definitions must not depend on the names contained in the definitions.
- Erase all the constructor names and record labels from the sorted type definitions.

[3] Objective Caml provides a facility to output functional values, but these values must be input in the same program. If we had some encoding for functional values which can be input in different programs, it would be easy to add functional types to our system, because our type check system is independent of the value encoding.

- Replace mutually defined data type names by indices #1, #2, and so on; other type names `t'` are simply replaced by their digests, $Digest(\Gamma_P, \mathtt{t}')$.
- Finally, calculate and return the MD5 digest of the string representation of these nameless definitions and an integer of the position of `t` in them.

Using this data type digest, we can define the notion of the equivalent data types more formally:

Definition 9 Equivalent data types *Two data types* `t` *of program* P *and* `t'` *of program* Q *are* internally equivalent *iff* $Digest(\Gamma_P, \mathtt{t}) = Digest(\Gamma_Q, \mathtt{t}')$

For example, the two types `tree` and `arbre` in the previous example are internally equivalent, because they have the same nameless definition:

```
type α #1 =
| * of α
| * of α #1 ⟨digest for list⟩
```

Because the digest for `list` can be considered same throughout all the Caml programs[4], $Digest(\Gamma_E, \mathtt{tree})$ and $Digest(\Gamma_F, \mathtt{arbre})$ have the same value.

5.4.5 Digest compilation

The data type digests are computed by the compiler from their definitions. Then they are stored in Caml object (`.cmo`) files as regular strings. For example, the digest for data type `tree` is compiled as the following definition:

```
let digest_tree =
  let definition_string =
    sprintf "type α #1 = * of α | * of α #1 %s" digest_list
  in
  compute_md5 (definition_string, 1)
```

The digest of `tree`, `digest_tree` is defined as the MD5 digest checksum of the string representation of its nameless type definition and its position index (here it is 1). The data type digest compilations depend on the already computed digests of the dependent data types which occur in their definitions, like in this example: the digest of `tree` is defined using the digest of `list`.

The data type digests can be statically compiled and stored, if all the dependent digests are accessible. However, in the separate module compilation environment, it is not always the case, and we may have to compute them at run time. If some data type used in our target type definition is abstracted and its definition is not available for the module we compile, the digest of the data

[4]The type `list` is a predefined type and has no actual definition. Therefore data type digest of `list` is just a string `"list"`.

type is unknown to us. Consider for instance, two module `M` and `N`, such that the implementation of `M` is not yet written or just does not exist: we just have the interface file `m.mli` when we compile `n.ml`.

m.mli	n.ml

`type t`	`type t' =`
	`| Boo of M.t (* abstract *)`
	`| Far`

In such case we cannot compute the digest of `N.t'` statically, since the implementation of `M.t` is not available and its digest is still undefined.

We can split such abstract data types into the following two groups. For each group we take different approach:

Completely abstract data types (foreign data types)

These data types have no implementation at all, and are used for communication between other languages such as C. Since ML programs do not know the structure of such types, their data type digests are undefined. The digests of the data types derived from these abstract data types are undefined neither. We can create dynamic values of such digest-less data types, treating their digests are null strings. But their exportation must be forbidden for type security reasons.

Instead of just rejecting their exportation, we may be able to give their digest in a somewhat explicit way. For example, if the operations of an abstract data type `t` is defined in a file `impl.c`, we can use the MD5 digest of the content of the file as the digest of `t`, using a new syntax:

```
type t with "impl.c"
```

But of course programmers must give these explicit digests with great care, unless they can easily break the value I/O security by giving the same digest for different non-equivalent data types.

Abstract data types with Caml definitions

In this case, we have the implementations for these data types but they are not accessible when we are compiling different modules. Their digests are available to the other modules when these implementations are compiled and linked. For a data type definition which contains such abstract data types inside, the compiler generates a code to compute its digest at program load time (i.e. the very beginning of program execution). For example, the digest of `N.t'` in the above example is defined as an expression using the value of the digest of `M.t` which will be linked together:

```
let digest_t' = compute_md5 (sprintf "type #1 = * of %s | *" M.digest_t, 1)
```

5.4.6 Run time representation of external types

We define the ML value representation of external types like the run time type representation at Section 3.3.1. Instead of the data type names of run time types external types carry data type digests. data types can carry their definitions as data type digests.

```
type rpath                                  (* type path name (≒ string) *)
type digest = string

type etype =
| ETvar   of int                                    (* type variables *)
| ETtuple of etype * etype                                  (* tuples *)
| ETconstr of digest * etype list     (* data types including basic types *)
```

The following conversion function $Ext(\Gamma_P, \tau)$ is a canonical mapping from type expressions to external types, replacing data type names by the corresponding digests of type environment Γ_P of program P:

$$\begin{aligned} Ext(\Gamma_P, \alpha) &= \alpha \\ Ext(\Gamma_P, \tau_1 * \ldots * \tau_n) &= Ext(\Gamma_P, \tau_1) * \ldots * Ext(\Gamma_P, \tau_n) \\ Ext(\Gamma_P, (\tau_1, \ldots, \tau_n)\ \mathtt{t}) &= (Ext(\Gamma_P, \tau_1), \ldots, Ext(\Gamma_P, \tau_n))\ Digest(\Gamma_P, \mathtt{t}) \end{aligned}$$

This conversion function is implemented in the following pseudo ML code of translation of ML value representations:

```
let rec rtype_to_etype = function
  | RTbase s → ETconst (s, [])
  | RTvar id → ETvar id
  | RTtuple (rt1,rt2) → ETtuple (rtype_to_type rt1, rtype_to_type rt2)
  | RTconstr (path, rts) →
      ETconstr (Digest(Γ_P,path), List.map rtype_to_etype rts)
```

In the above definition of `rtype_to_etype`, we suppose that the data type digests $Digest(\Gamma_P, \mathtt{path})$ are globally available and calculated at run time, when the run time types are actually constructed. For example, «`int tree`» = `RTconstr("tree", [RTbase "int"])` is converted to a value of external type: `ETconst(digest_tree, [ RTconst ("int", []) ])` where the variable `digest_tree` stores $Digest(\Gamma_P, \mathtt{tree})$.

5.4.7 External instance relationship

Like the normal type expressions, we can extend the notion of external types to quantified external type schemes σ_e, and define instance relationship $\tau_e \leq \sigma_e$ without problem. This instance relationship between external types and type schemes is implemented in Figure 5.1 just like as `is_instance` for run time types. `is_external_instance` τ_{e1} τ_{e2} returns true if and only if τ_{e1} is an instance of the external type scheme $\forall FV(\tau_{e2}).\tau_{e2}$. When `is_external_instance` receives two data types, it checks they have the same data type digests. If they are different, they are not instance of each other.

```
let is_external_instance t t' =
  let subst = ref [] in
  let rec is_instance_aux t t' =
    match t, t' with
    | _, ETvar id →
        begin try
            if List.assoc id !subst = t then true else false
          with
            Not_found → subst := (id, t) :: !subst; true
        end
    | ETtuple (t1,t2), ETtuple(t'1,t'2) →
        is_instance_aux t1 t'1 && is_instance_aux t2 t'2
    | ETconstr (d1, [t11;...;t1n]), ETconstr(d2, [t21;...;t2n]) →
        d1 = d2 && is_instance_aux t11 t21 && ... && is_instance_aux t1n t2n
    | _ → false
  in
  is_instance_aux t t'
```

Figure 5.1: Definition of the predicate `is_external_instance`

5.5 Value I/O primitives

With external types, we can define our safe value I/O primitives, `export_value` and `import_value`:

```
val export_value : ∀δ. out_channel → δ → unit
val import_value : ∀δ. in_channel → δ
```

$$
\begin{aligned}
gen'(\texttt{export\_value}) &= \texttt{fun t oc v} \rightarrow \texttt{output\_value oc (v, rtype\_to\_etype t)} \\
gen'(\texttt{import\_value}) &= \texttt{fun t ic} \rightarrow \texttt{let } (\texttt{v}, \texttt{t}') = \texttt{input\_value ic in} \\
&\qquad\qquad \texttt{if is\_external\_instance t}'\texttt{ t then v} \\
&\qquad\qquad \texttt{else raise Coercion\_failure}
\end{aligned}
$$

The compilation of the primitives are defined as mixtures of the compilation of dynamic value primitives and the original unsafe I/O primitive functions. `export_value`, like `dynamic`, creates a pair of the internal value representation of the given argument `v` and its external type representation and write it out to a channel. `import_value` reads such a value-type pair and checks whether it is an instance of the external type representation of the expected type. If the check succeeds the internal value representation of the input value is coerced into the importing program side as a value of the expected type.

5.6 Notes

Polymorphic encapsulation

We can use the same technique studied in Chapter 4 to make the safe I/O primitives possible to encapsulate polymorphic values correctly. The `export_value` primitive will take full poly-dynamic quantification $\forall\partial$ instead of the current normal one $\forall\delta$, and its second argument will be checked its generalization with a special typing rule. To import back exported values with polymorphic external type scheme to polymorphic, `import_value` expression must be a special syntax construct like the `coerce` expression.

Digest computation at link time

In the separate module compilation environment, data type digests of type definitions cannot be computed statically at each module compilation, due to the existence of the abstract types. The definitions of abstract types are inaccessible outside of the module which defines them, or even worse, the definitions might be written yet. Therefore, in Section 5.4.3, for such statically unavailable data type digests at each module compilation, we embed small piece of code to compute them at load time. They are calculated when programs are launched each time. But actually, they can be computable statically at the link time, since all the data type definitions are available at that time. This kind of link time optimization reduces the size of the executables and the load time overhead, but requires additional framework to retrieve type definitions from the compilation result of modules, even they are abstracted by signatures.

5.7 Conclusion

We proposed a type safe ML value I/O over extensional polymorphism. The values are output with small size type information, where data types are identified by an abstraction of their definitions, which represents their universal semantics. This gives to our safe I/O system an interesting property – two or more different programs communicating via this value I/O do not need to share the same type definitions. The data type definitions information is compressed using MD5 digests. This allow extremely compact type information in exported values.

We have already implemented a prototype of the safe value I/O functionality over an extensional polymorphism implementation as an extension of Objective Caml version 2.02 system. It also has polymorphic type-safe value I/O, and its `coerce` and `import` constructs handle multiple type cases. We cannot compute all the data type digests statically due to the abstract data types, but their calculation is lazy-evaluated, so there is no meaningless overhead at run time. The executable (object code) size growth due to data type digest embedding is usually about 0 to 10 percent (0 for programs with no type definitions, 10 for `ocamlc`, which might have a high typedefs/code ratio). This is perfectly acceptable.

Chapter 6

Unification of dynamic values and safe value I/O

6.1 Introduction

Marriage of dynamic values and safe value I/O

To demonstrate the power of extensional polymorphism, in the last two chapters we implemented the basic functionalities of dynamic values and type safe value I/O as generic primitives. They were formalized differently, but their internal mechanisms are identical: they are both systems of encapsulation and decapsulation of values with their types. Only the difference between them is the type informations stored in the encapsulation. In this chapter, we make both systems into one, by unifying their type information for the encapsulation. The unified *external values* are value-type pairs which carry both data type names and their digests. The primitives of dynamic values and safe value I/O are updated to handle this external values. It is not just a mere simplification, it also makes them cover their shortages each other. Having data type digests inside, dynamic values are secure even if they are exported outside of programs. In the value I/O, having another run time type checking which compares the name of data types, we can use more strict value importation.

Outline of this chapter

In Section 6.2, we explain the necessity of unification of dynamic values and safe value I/O, showing possible type insecurity at the combination of these two functionalities. Section 6.3 and 6.4 extend run time types and dynamic values so that they can also carry related data type digests. The two different dynamic decapsulation schemes which come from the decapsulation primitives of dynamic values and safe value I/O, `coerce` and `import`, are explained in Section 6.5. In Section 6.6, we define the primitives for the unified system using the extended run time types with digests and two decapsulation schemes.

6.2 Dynamic values in value I/O

In the last implementation of dynamic values, we did not suppose that dynamic values can be exported outside of programs. Inside one program, data types can be identified by their names. But this s no longer correct in the combination of dynamic values and value I/O. For example, imagine an exportation of a dynamic value $(v, \mathtt{t})$, a dynamic value of a data type `t`:

```
let oc = open_out "foo.dat" in
let d = dynamic (v : t) in
export_value oc d;;
```

The `export_value` primitive creates a value-type pair $((v, \mathtt{t}),$ `ETconst("dyn", [])` $)$ and export this pair outside of the program. In a reader program, we import back this pair as a dynamic value:

```
let ic = open_in "foo.dat" in
let d = (import_value ic : dyn);;
```

The type digest `"dyn"` is verified against the expected type `dyn` and the importation is granted as $(v, \mathtt{t})$. But this imported dynamic value is not a valid type-value pair for the reader program, since the data type name `t` is came from different program.

Run time types encapsulated in dynamic values must also contain data type digests and be universal throughout different programs, since dynamic values can now be exported outside of the program which created them by value I/O. The coercion of dynamic values must also check the data type digests.

6.3 Run time types with digests

Dynamics require data type digests and value I/O want data type names. Then why not merge them and obtain unique run time type representation? In the new *run time types with digests*, additional to the initial definition at Section 3.3.1, the run time representation of data type `RTconst` has a new member `digest`, the digest of the data type:

```
type rpath                                    (* type path name (≒ string) *)
type digest = string

type rtype =
| RTbase  of rpath                                         (* basic types *)
| RTvar   of int                                       (* type variables *)
| RTarrow of rtype * rtype                             (* function types *)
| RTtuple of rtype * rtype                                      (* tuples *)
| RTconst of rpath * digest * rtype list                    (* data types *)
```

The conversion function $\ll \cdot \gg$ which is for translating the normal ML type expressions to run time types is now extended to embed data type digests:

$$
\begin{array}{lcl}
\ll base \gg & = & \texttt{RTbase}\ \text{"}base\text{"} \\
\ll \delta_i \gg & = & \texttt{RTvar}\ i \\
\ll \tau_1 \rightarrow \tau_2 \gg & = & \texttt{RTarrow}(\ll \tau_1 \gg,\ \ll \tau_2 \gg) \\
\ll \tau_1 * \tau_2 \gg & = & \texttt{RTtuple}(\ll \tau_1 \gg,\ \ll \tau_2 \gg) \\
\ll (\tau_1, \ldots, \tau_n)\ \texttt{t} \gg & = & \texttt{RTconst}(\text{"}\texttt{t}\text{"},\ \mathit{Digest}(\Gamma_P, \texttt{t}),\ [\ll \tau_1 \gg;\ \ldots;\ \ll \tau_n \gg])
\end{array}
$$

For example, «`int tree`» returns a run time type with digests: `RTconst("tree", digest_tree, [ RTbase "int" ])` where the variable `digest_tree` stores $\mathit{Digest}(\Gamma_P, \texttt{tree})$.

Rest of this chapter, run time types with digests are called run time types.

6.4 External dynamic values

The internal representation of dynamic values and exported values in the value I/O is unified to *external dynamic values*, pairs of a value and the new run time type representation of its type. We keep using the abstract type `dyn` for the unified external dynamic values.

6.5 Two modes of value importation

Data type identification of the previous implementation of the safe value I/O is by their semantics based on the internal value representation, i.e. nameless definitions. If they define data with the same internal structure, data types can be compatible even when their definitions can have different names, For example, we can exchange between the following data types via value I/O:

```
type α tree =                  type α arbre =
| Leaf of α                    | Feuille of α
| Branch of α tree list        | Branche of α arbre list
```

This confusion of data types provides an interesting and practical safe data exchange between different data types, but in some cases this confusion is not welcome. In the normal semantics of data types, above two types are completely different. This strict distinction of data types may be necessary in value I/O sometimes, too. By encapsulating values as external dynamic values at export, and comparing also data type names at import, we can introduce an additional mode of value importation with this more strict data type identification. We call these two decapsulation of external dynamic values as *coercion* and *importation*. Both decapsulations check data type digests equality in the data type identification, while at coercion data type path names are additionally taken into account in the identification.

We implement two levels of run time type instance relationships in the unified system of external dynamic values, corresponding to the two decapsulation schemes: `is_coerce_instance` and `is_import_instanace` in Figure 6.1. They are different only in the equivalence check of data types,

```
(* basic function is_instance, shared by both *)
let is_instance p ty ty' =
  let subst = ref [] in
  let rec is_instance_aux ty ty' =
    match ty, ty' with
    | «τ», «α» →
        begin try
            if List.assoc «α» !subst = ty then true else false
          with
            Not_found → subst := («α», ty) :: !subst; true
        end
    | «τ1 → τ2», «τ'1 → τ'2» →
        is_instance_aux «τ1» «τ'1» && is_instance_aux «τ2» «τ'2»
    | «τ1 * τ2», «τ'1 * τ'2» →
        is_instance_aux «τ1» «τ'1» && is_instance_aux «τ2» «τ'2»
    | «base1», «base2» → base1 = base2
    | «(τ1,...,τn) t» «(τ'1,...,τ'n') t'» →
        (* comparison of data types are delegated to the predicate p *)
        p ty ty'
    | _ → false
  in
  is_instance_aux t t'

(* data type comparison for importation *)
let import_equivalence
  (RTconst(_, d, [ty1;...;tyn])) (RTconst(_, d', [ty'1;...;ty'n'])) =
    d = d' && n = n' && ∀1 ≤ i ≤ n, is_instance_aux tyi ty'i

(* data type comparison for coercion = import_equivalence + path check *)
let coerce_equivalence (RTconst(p, d, tys)) (RTconst(p', d', tys')) =
  p = p' && import_equivalence (RTconst(p, d, tys)) (RTconst(p', d', tys'))

(* the two levels of instance relationships for importation and coercion *)
let is_import_instance = is_instance import_equivalence
let is_coerce_instance = is_instance coerce_equivalence
```

Figure 6.1: Two versions of instance predicate `is_*_instance`

and share the rest in `is_instance`. Different from Figure 4.1, this `is_instance` takes an additional argument `p`, a data type equivalence predicate. By applying different equivalence predicates, we obtain the two instance relationships. `import_equivalence` for the importation compares the data type digests, arities and arguments of given two data types. `coerce_equivalence` checks also the path name equality additionally to those of `import_equivalence`.

6.6 New primitives

6.6.1 Dynamic value primitives

The dynamic primitives and their compilations are updated using the new run time type instance relationships.

```
val dynamic : ∀δ.δ → dyn
val coerce : ∀δ.dyn → δ
val import : ∀δ.dyn → δ
```

$$
\begin{aligned}
gen'(\texttt{dynamic}) &= \texttt{fun t v} \;\rightarrow\; (\texttt{v}, \texttt{t}) \\
gen'(\texttt{coerce}) &= \texttt{fun t } (\texttt{v}, \texttt{t}') \;\rightarrow\; \texttt{if is\_coerce\_instance t t}' \texttt{ then v} \\
&\qquad\qquad\qquad\qquad\quad \texttt{else raise Coercion\_failure} \\
gen'(\texttt{import}) &= \texttt{fun t } (\texttt{v}, \texttt{t}') \;\rightarrow\; \texttt{if is\_import\_instance t t}' \texttt{ then v} \\
&\qquad\qquad\qquad\qquad\quad \texttt{else raise Coercion\_failure}
\end{aligned}
$$

The compilation of the `dynamic` primitive is not modified and same as before, but it receives newly defined run time types with digests which are obtained by the new run time type translation function « · ». The `coerce` primitive is diverged to two primitives, `coerce` and `import`, using different instance relationships for each. These primitives themselves cannot treat the polymorphic contents of external dynamic values well, but the same technique we discussed in Section 4.3 can be applied to them, too. Thanks to the data type digests, we can coerce and import external dynamic values safely:

```
# let v = Branch [Leaf 1; Leaf 2];;
v : int tree
# let d = dynamic v;;                         (* creates (v, int tree) *)
d : dyn = <abstr>
# (coerce d : int tree);;
- : int tree = Branch [Leaf 1; Leaf 2]
# (coerce d : int arbre);;             (* coerce checks the type names *)
Uncaught exception: Coercion_failure
# (import d : int tree);;                              (* same as coerce *)
- : int tree = Branch [Leaf 1; Leaf 2]
# (import d : int arbre);;            (* import can rename data types ! *)
- : int arbre = Branche [Feuille 1; Feuille 2]
```

6.6.2 Safe I/O primitives

Once the data type digests provide the universal run time type representations of external dynamic values amongst the different programs, the primitives of the type safe value I/O can be obtained for free. We just restrict the polymorphic types of the unsafe version of the value I/O functions to monomorphic, so that they can just accept external dynamic values:

```
let export_dyn = (output_value : out_channel → dyn → unit)
let import_dyn = (input_value : in_channel → dyn)
```

After these primitives are introduced, the original `input_value` and `output_value` functions must be inaccessible to users since they are unsafe.

We can also shift encapsulation of the output values into the `export_dyn` and make it a derived dynamizer `export_value` as follows:

```
val export_value : ∀∂̲.out_channel → ∂ → unit
let export_value oc v = export_dyn oc (dynamic v)
```

6.7 Conclusion

Systems of dynamic values and type safe value I/O were formalized and implemented independently, though their internal mechanism share many concepts. We unified them to external dynamic values, value-type pairs which can be used for both purposes of dynamics and value I/O. This unification is not a simplification, but extends both systems. The dynamic values are now safe even they are exported by value I/O outside of programs. Having a new decapsulation scheme, we can perform more strict data type identification at value importation.

The run time types are modified to carry data type digests like external types of the type safe value I/O, so that exported dynamic values can be coerced safely inside different reader programs. By checking data type path names additional to digest comparison, imported values can be decapsulated more strictly, using the `coerce` primitive. On the other hand, the `import` primitive weaken the condition of dynamic value decapsulation and enables exchanging between different data types of the same internal structures via external dynamic values.

Chapter 7

Generic definitions

7.1 Introduction

User-defined generic values

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

Compilation of user generics

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

Outline of this chapter

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

7.2 user-defined generic values

7.2.1 Generic bindings

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

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

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

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

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

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

# plus 1.0 3.14;;
- : float = 4.14

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

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

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

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

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

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

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

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

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

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

7.2.2 Recursive generics

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

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

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

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

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

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

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

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

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

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

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

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

The combination of extensional polymorphism and recursion allows much more complicated generic definitions than mere simple "overloaded operators". The following generic function `flat` demonstrates such an example. It can flatten a list of any depth:

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

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

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

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

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

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

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

7.2.3 Generic run time errors

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

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

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

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

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

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

7.3 Formal syntax

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

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

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

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

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

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

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

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

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

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

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

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

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

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

7.4 Typing of generic bindings

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

7.4.1 Anti-unification

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

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

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

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

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

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

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

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

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

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

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

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

7.4.2 Anti-unification algorithm

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

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

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

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

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

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

$aunif(\Phi, \tau_1 \rightarrow\ \tau_2,\ \tau'_1 \rightarrow\ \tau'_2) =$
let $(\Phi_1, \tau''_1) = aunif(\Phi, \tau_1,\ \tau'_1)$ in
let $(\Phi_2, \tau''_2) = aunif(\Phi_1, \tau_2,\ \tau'_2)$ in
$(\Phi_2, \tau''_1 \rightarrow \tau''_2)$

$aunif(\Phi, \tau_1 * \tau_2,\ \tau'_1 * \tau'_2) =$
let $(\Phi_1, \tau''_1) = aunif(\Phi, \tau_1,\ \tau'_1)$ in
let $(\Phi_2, \tau''_2) = aunif(\Phi_1, \tau_n,\ \tau'_n)$ in
$(\Phi_2, \tau''_1 * \tau''_2)$

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

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

$antiunif(\{\tau_1, \ldots, \tau_n\}) =$
let $\tau'_1 = \tau_1$ in
let $(\Phi_2, \tau'_2) = aunif(\emptyset, \tau'_1,\ \tau_2)$ in
$\cdots$
let $(\Phi_i, \tau'_i) = aunif(\emptyset, \tau'_{i-1},\ \tau_i)$ in
$\cdots$
let $(\Phi_n, \tau'_n) = aunif(\emptyset, \tau_{n-1},\ \tau_n)$ in
τ'_n

Figure 7.1: Anti-unification algorithm

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

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

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

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

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

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

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

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

Type variables Similar to the proof for the basic types.

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

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

From Lemma 15,

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

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

If τ'' is a regular type variable α, then there is a substitution $S = \{\alpha \Rightarrow \tau_1'' \to \tau_2''\}$ which maps α to the result of *aunif*.

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

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

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

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

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

Tuples Similar to the proof for the arrow types.

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

□

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

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

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

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

7.4.3 Anti-unification of type patterns

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

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

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

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

Example The following generic definition:

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

defines g whose type scheme is

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

□

7.4.4 Typing rules

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

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

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

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

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

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

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

7.4.5 Polymorphic recursion in generic binding

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

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

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

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

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

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

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

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

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

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

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

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

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

7.4.6 Reconstruction algorithm

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

Correctness of the reconstruction

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

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

Proof: By induction over the structure of e_t.

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

$W(\Gamma,$ `generic (rec) x = case` $\sigma_1 \Rightarrow e_1 \mid \ldots \mid \sigma_n \Rightarrow e_n$ `in` $e) =$

if $\exists i,\ FV(\sigma_i) \neq \emptyset$ or $BV(\sigma_i) \cap \mathbb{R} \neq \emptyset$ then fails
let $\forall V_i.\tau_i = \sigma_i$
let $\sigma = p\_antiunif(\{\sigma_i\})$

let $\Gamma_0 =$ if recursive then $\Gamma\{\mathtt{x} : \sigma\}$ else Γ
let $\Sigma_0 = \emptyset$

let $\Gamma_1 = \Sigma_0(\Gamma_0)$
let $e_{t1} : \tau_1',\ S_1 = W(\Gamma_1, e_1)$
let $S_1' = unify(\tau_1, \tau_1')$
let $e_{t1}' = S_1'(e_{t1})$
let $\Sigma_1 = S_1' \circ S_1$
let $\sigma_1' = Gen(\Sigma_1(\Gamma_1), e_{t1}')$
if $\sigma_1 \stackrel{\alpha}{\neq} \sigma_1'$ then W fails
$\ldots$
let $\Gamma_i = \Sigma_{i-1}(\Gamma_{i-1})$
let $e_{ti} : \tau_i',\ S_i = W(\Gamma_i, e_i)$
let $S_i' = unify(\tau_i, \tau_i')$
let $e_{ti}' = S_i'(e_{ti})$
let $\Sigma_i = S_i' \circ S_i$
let $\sigma_i' = Gen(\Sigma_i(\Gamma_i), e_{ti}')$
if $\sigma_i \stackrel{\alpha}{\neq} \sigma_i'$ then W fails
$\ldots$
let $\Gamma_n = \Sigma_{n-1}(\Gamma_{n-1})$
let $e_{tn} : \tau_n',\ S_n = W(\Gamma_n, e_n)$
let $S_n' = unify(\tau_n, \tau_n')$
let $e_{tn}' = S_n'(e_{tn})$
let $\Sigma_n = S_n' \circ S_n$
let $\sigma_n' = Gen(\Sigma_n(\Gamma_n), e_{tn}')$
if $\sigma_n \stackrel{\alpha}{\neq} \sigma_n'$ then W fails

let $e_t : \tau,\ S = W(\Sigma_n(\Gamma_n)[\mathtt{x} : \sigma], e)$
let $\Sigma = S \circ \Sigma_n \circ \ldots \circ \Sigma_0$
in `generic (rec)` $\mathtt{x}_\sigma$ `= case`
$\mid \sigma_1 \Rightarrow (S \circ \Sigma_n \circ \ldots \circ \Sigma_2)(e_{t1})$
$\ldots$
$\mid \sigma_i \Rightarrow (S \circ \Sigma_n \circ \ldots \circ \Sigma_{i+1})(e_{ti})$
$\ldots$
$\mid \sigma_n \Rightarrow S(e_{tn})$
`in` $(e_t : \tau) : \tau,\ \Sigma$

Figure 7.2: Reconstruction algorithm for generic bindings

By induction hypothesis, we have

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

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

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

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

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

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

Without losing any generality, we can assume that

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

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

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

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

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

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

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

□

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

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

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

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

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

Applying S_i' to this judgment, by Theorem 20,

$(S_i' \circ S_i)(\Gamma_i) \vdash S_1'(e_{t1}) : \tau_i'$, and this is equal to

$$\Sigma_i(\Gamma_i) \vdash e_{t1}' : S_1'(\tau_i')$$

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

$$(S \circ \Sigma_n \circ \ldots \circ \Sigma_i)(\Gamma_i) \vdash (S \circ \Sigma_n \circ \ldots \circ \Sigma_{i+1})(e_{ti}' : S_i'(\tau_i'))$$

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

$$\Sigma(\Gamma) \vdash e_{ti}''$$

The type scheme of e_{ti}' is written as $Gen(\Sigma_i(\Gamma_i), e_{t1}' : S_i'(\tau_i')) = \sigma_i$.

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

$$\begin{aligned} &Gen((S \circ \Sigma_n \circ \ldots \circ \Sigma_i)(\Gamma_i), (S \circ \Sigma_n \circ \ldots \circ \Sigma_{i+1})(e_{t1}' : S_1'(\tau_i'))) \\ &\qquad\qquad\qquad\qquad = (S \circ \Sigma_n \circ \ldots \circ \Sigma_{i+1})(\sigma_i) \end{aligned}$$

It is equal to

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

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

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

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

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

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

□

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

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

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

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

Generic bindings Suppose that e is

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

and e has a typing

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

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

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

> If there is a substitution Σ'_i such that $\Sigma'_i(\Gamma_i) = S'(\Gamma)$, the type case reconstruction of $(\sigma_i \Rightarrow e_i)$
>
> let $\Gamma_i = \Sigma_{i-1}(\Gamma_{i-1})$
> let $e_{ti} : \tau'_i,\ S_i = W(\Gamma_i, e_i)$
> let $S'_i = \mathit{unify}(\tau_i, \tau'_i)$
> let $e'_{ti} = S'_i(e_{ti})$
> let $\Sigma_i = S'_i \circ S_i$
> let $\sigma'_i = Gen(\Sigma_i(\Gamma_i), e'_{ti})$
> if $\sigma_i \overset{\alpha}{\not\equiv} \sigma'_i$ then W fails
>
> is successful and there is a substitution Σ'_{i+1} such that $\Sigma'_{i+1}(\Sigma_i(\Gamma_i)) = S'(\Gamma)$.

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

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

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

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

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

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

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

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

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

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

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

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

and

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

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

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

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

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

and there exists a substitution Σ'' such that

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

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

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

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

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

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

□

7.5 Semantics and type soundness

7.5.1 Semantics of `generic` expressions

Semantic values

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

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

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

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

Evaluation rules

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

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

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

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

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

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

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

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

7.5.2 Type soundness

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

Extended semantic typing

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

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

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

Lemma 8 is also extended:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

□

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

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

Incomplete type soundness

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

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

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

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

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

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

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

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

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

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

where r is not err.

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

Generic bindings We have

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

The possible evaluation rule is (e-generic):

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

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

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

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

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

□

7.6 Compilation

7.6.1 Type pattern matching

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

$$\begin{array}{lcll}
e & ::= & \ldots & \text{Original IML} \\
 & & \mathtt{match}\ e\ \mathtt{with}\ mc \mid \ldots \mid mc & \text{Match expression} \\
mc & ::= & p \ \rightarrow \ e & \text{Match cases} \\
p & ::= & b_\sigma & \text{Basic constants} \\
 & \mid & \mathtt{x} & \text{Pattern variables} \\
 & \mid & p, p & \text{Tuple patterns} \\
 & \mid & RT(p, \ldots, p) & \text{Pattern constructors} \\
RT & ::= & \mathtt{RTvar} \mid \mathtt{RTarrow} \mid \mathtt{RTtuple} \mid \mathtt{RTconstr} & \text{Run time type constructors}
\end{array}$$

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

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

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

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

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

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

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

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

7.6.2 Compilation to type pattern matching

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

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

Compilation of type patterns

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

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

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

We here prove the correctness of the type pattern compilation:

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

Proof: By induction over the type pattern σ.

Base type pattern Trivial.

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

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

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

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

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

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

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

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

Tuple type pattern Similar to the arrow patterns.

$\square$

Compilation of type cases and generic bindings

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

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

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

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

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

Example

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

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

is compiled into the following IML code:

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

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

Non linear patterns in compilation

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

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

7.6.3 Optimized compilation of type patterns

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Therefore we can merge them without contradiction:

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

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

$\square$

7.6.4 Compilation correctness

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

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

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

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

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

such that $v' \sqsubset v$.

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

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

Typing and evaluation in GML:

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

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

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

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

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

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

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

Induction hypothesis:

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

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

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

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

Evaluation of the compiled code:

The compilation is:

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

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

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

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

(e-gcase)

Typing and evaluation in GML:

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

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

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

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

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

Induction hypothesis:

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

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

where

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

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

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

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

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

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

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

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

Evaluation of the compiled code:

$\mathtt{x}$ is compiled as

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

□

7.7 Comparison with related works

7.7.1 No regular type variables in patterns

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

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

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

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

7.7.2 Semantics over the source language

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

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

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

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

$\Longrightarrow$

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

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

7.8 Conclusion

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

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

Non-exhaustiveness and constraint typing

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

Extension of user-defined generic values

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

Chapter 8

Constraint typing

8.1 Introduction

Detection of illegal generic instances

We have introduced generic bindings which permit us to define generic values by declaring list of type cases. The semantics of these "user-defined" generic values — selection of these type cases is by run time type matching, which can be naturally compiled to normal pattern match expressions. We also provided a simple extension of the GML type system for user generics. This simplicity of the typing system extension is achieved by ignoring an important property of user-defined generic values: even if a user-defined generic value has a type scheme σ in the type system, its semantics is not defined for all the instances of σ, in the sense that in some evaluation type context, it may not be evaluated to a semantic values, but *gerr*, the run time type error. This *non-exhaustiveness* of user generics cannot be taken into account, as far as we just give the anti-unifier of type patterns as the type scheme of a user-defined generic value.

Even though generic run time errors are reported as run time exceptions and they can be caught, it should be preferable if they can be detected at compile time statically. Otherwise, invalid use of generic instances must be found by trial and error basis debugging. Therefore in this chapter, we examine and challenge problems of the previous type system, and propose a solution, *generic constraint typing system* which can handle generic non-exhaustiveness.

Two-layer design

We are *not* going to develop this generic constraint typing system as an extension of the type system we have used in the previous chapters. Non-exhaustiveness information of generic values which will be called *generic constraints* cannot be expressed as type schemes. Moreover, they become quite complex: constraints of derived generic values must contain non-exhaustiveness information of other values, and constraints of recursively defined generic values are also recursive. Integrating such complex type information directly into the current type system will surely destroy its theoretical simplicity of ML typing.

Generic constraint typing system is defined as a new typing layer independent of the "normal" generic core type system. This two-layer design makes both type algebra simpler. The generic constraint typing verifies types of expressions which are supposed to be already type-checked in the normal typing layer. Under this assumption, generic constraint typing rules can focus just on generic related expression constructs. The other parts are granted whatever types they have. This separation also benefits implementation of extensional polymorphism. Type reconstruction has two phases corresponding to the two typing layers: the first reconstruction phase is a normal ML type reconstruction with extension of dynamic type variables. This *normal reconstruction phase* treats generic values as parametric polymorphic values. Then type-annotated expressions are passed to the second *generic type reconstruction phase*, which detects illegal uses of generic instances and tries to instantiate over-general types of generic instances to something valid. Thus the generic type reconstruction phase is independent from the normal reconstruction, therefore we can implement them separately. This is important when we try to implement extensional polymorphism as an extension of some existing ML implementation.

Outline of this chapter

We first points out the typing problem of core system with user-defined generic values in Section 8.2. Then in Section 8.3 we informally introduce the notion of full and polymorphic instances which distinguishes instances by their type annotation generalization. In Section 8.4 we define generic constraints, the type information of the generic constraint typing. Section 8.5, 8.6 and 8.7 define type instantiation $\tau <: \mathcal{K}$ of generic constraints to type expressions. Then we back to the formal full and polymorphic instances using generic constraints in Section 8.8. The type system itself is defined in Section 8.9, and 8.10 proves its soundness.

8.2 The limitation of the generic core typing system

As we have proven, the generic core type system including generic bindings for user generics is type-sound in some sense. That is, it makes sure that its well-typed programs are never evaluated into *err*: in other words, their compilation never fall into segmentation faults or bus errors when they are executed by an ML run time system. The other, generic run time error, *gerr* is raised when we have incorrect uses of user-defined generic values. If they are evaluated under types which do not match with any of their type patterns. For example:

```
generic plus = case
| int → int → int ⇒ (+)
| float → float → float ⇒ (+.)
;;
val plus : ∀δ.δ → δ → δ
```

This `plus` generic function is defined only for the types `int` → `int` → `int` and `float` → `float` → `float`. Therefore, using `plus` in another type raises *gerr*:

```
# print_string (plus "hello" "world");;
Uncaught exception: Type_match_failure
```

The type pattern matching of user generics is compiled to IML's normal pattern matching. Therefore, as in the normal pattern matching, a run time type match failure *gerr* is reported as an exception. This exception can be easily handled by catching.

Normal ML typing does not check pattern match failures in a static type system but leave them to be found at run time. Just like pattern match failures, should we take the generic run time error for granted? Our answer here is no, from the following reasons.

First of all, in the GML semantics, the generic run time error is essentially an error of types, while normal pattern match failure is an error on values. The generic run time error is reported as a run time exception, but this is a property derived from our compilation method which translates user generics to normal pattern matching, and has no relationship with the semantics themselves. The above justification of generic run time errors as pattern match failure exception has no sense if we use different compilation scheme of user generics. Actually, in Chapter 9, we will introduce a new user generic value compilation without type pattern matching at run time.

In addition, since we have just finite number of type cases in generic bindings while the set of types is infinite, in almost cases user-defined generic values are non-exhaustive unless they have the default type cases. This means that user-defined generic values can be easily misused under evaluation type contexts undefined for them. Therefore rather than reported as run time exceptions, the incorrect generic instances should be detected in the type checking phase statically, otherwise run time type matching errors would suffer seriously programming productivity of extensional polymorphism.

This task – static detection of illegal generic instances – is impossible for the generic core type system, because of its ignorance of the type case information, i.e. non-exhaustiveness of generic values. To detect illegal generic instances, we must add *generic constraints*, non-exhaustive information of user generics to type systems.

8.3 Full and polymorphic instances

We first need to distinguish generic instances into two groups, instances whose validity can be checked at the point of their occurrences or not.

Full instances

We say an instance is a *full instance*, if its type annotation has no type variables generalized by some let or generic generalization. For example, in the following program,

```
generic plus = case
...
in
plus 1 2;;
```

the instance `plus` in the last line is *fully instantiated*. Evaluation type contexts of full instances are the same as their type annotations, because their annotations are not generalized. Therefore according to their annotations we can check whether they are evaluated correctly without any generic run time type error or not. Full instances which could fail into the generic run time error are regarded as *invalid* and rejected. Only the *valid* full instances which are proven to be safe are permitted in the constraint type system.

Polymorphic instances

On the other hand, instances whose types have dynamic type variables generalized somewhere outside are called *polymorphic instances*. For example, we have such polymorphic instances in definitions of derived polymorphic values:

```
let double x = plus x x in ...;;
```

Here the instance of `plus` is a polymorphic instance. The evaluation type context of `plus` is dynamic and unknown at the point of the definition of `double`, until full instantiations of `double`, like `double 1`. Validity of polymorphic instances is dependent on how the values which use them are fully instantiated. Therefore we delegate our validity investigation against polymorphic instances to checks of full instances of the values which use them.

The distinction of full and polymorphic instances are formally defined in Section 8.8.

8.4 Generic constraints

Now we introduce new type information for generic values, *generic constraints* (or just constraints in short), which indicate non-exhaustiveness information of generics. fully type instantiations of a generic value is restricted to those which satisfy these constraints.

8.4.1 Type patterns

For simplest cases, constraints for generic values are exactly the lists of type patterns. For example, the following simple `print_non_list` generic function

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

has the following constraint, which is identical to the type patterns of the definition:

$$\texttt{print\_non\_list} : [\ \texttt{string} \rightarrow \texttt{unit} \mid \texttt{int} \rightarrow \texttt{unit}\ ]$$

This form of constraint is called *type patterns* too. If `print_non_list` is fully instantiated, its type must be an instance of one of the type patterns, therefore, `string` → `unit` or `int` → `unit`. These two full instance types are called *(valid) instances* of the constraint $[\,$`string`$\rightarrow$`unit`$\mid$`int`$\rightarrow$`unit`$\,]$. We use the symbol <: to denote this relationship. Here, for example,

$$\texttt{int} \rightarrow \texttt{unit} \;<:\; [\,\texttt{string} \rightarrow \texttt{unit} \mid \texttt{int} \rightarrow \texttt{unit}\,]$$

denotes that `int` $\rightarrow$ `unit` is one of the valid instances of the constraint.

8.4.2 Derived constraints

Thus constraints indicate in which types generic values can be instantiated without any generic run time error in evaluation. But things become bit more complicated if a `generic` binding uses other generic values inside:

```
generic print_list_also = case
| ∀δ.δ list → unit ⇒ fun l → List.iter print_non_list l
| ∀δ.δ → unit ⇒ print_non_list
```

In this case, the constraint of `print_list_also` is not just

$$[\,\forall\delta.\delta\ \texttt{list} \rightarrow \texttt{unit} \mid \forall\delta.\delta \rightarrow \texttt{unit}\,]$$

We cannot use this generic value for all the instances of the type patterns, because the both type method expressions contain polymorphic instances of `print_non_list`. For example, suppose that `print_list_also` is instantiated as $\tau = S(\delta\ \texttt{list} \rightarrow \texttt{unit})$ for some substitution S, using the first type case. Then in order to evaluate this instance without any run time type error, the occurrence of `print_non_list` in the type method expression with the type $\delta \rightarrow \texttt{unit}$ must be also a valid full instance under S: $S(\delta \rightarrow \texttt{unit})$ has to be a valid full instance type of the constraint of `print_non_list`, $[\,\texttt{int} \rightarrow \texttt{unit} \mid \texttt{float} \rightarrow \texttt{unit}\,]$. To express this further constraints derived from polymorphic instances of non-exhaustive generic values, we introduce *derived constraints* $\tau <: \mathcal{K}$, where $\mathcal{K}$ is a constraint of such a polymorphic instance. Now a type pattern in constraints accompanies with a (possibly null) list of derived constraints $\{\tau_1 <: \mathcal{K}_1\ \&\ \ldots\ \&\ \tau_n <: \mathcal{K}_n\}$. `print_list_also` must have a generic constraint with these derived constraints:

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

Here $\mathcal{K}_{\texttt{print\_non\_list}} = [\,\texttt{string} \rightarrow \texttt{unit} \mid \texttt{int} \rightarrow \texttt{unit}\,]$. The instantiation of `print_list_also` is now restricted to `string list` $\rightarrow$ `unit`, `int list` $\rightarrow$ `unit`, `string` $\rightarrow$ `unit` and `int` $\rightarrow$ `unit`, with these derived constraints.

Derived constraints are also used for constraints of let bound derived generic polymorphic values:

$$\begin{aligned}&\texttt{let print\_tuple (x,y) = print\_non\_list x; print\_non\_list y;;}\\ &\texttt{print\_tuple} :\ [\,\forall\delta\delta'.\delta * \delta' \rightarrow \texttt{unit}\ \{\ \delta \rightarrow \texttt{unit} <: \mathcal{K}_{\texttt{print\_non\_list}}\ \&\\ &\qquad\qquad\qquad\qquad\qquad\qquad \delta' \rightarrow \texttt{unit} <: \mathcal{K}_{\texttt{print\_non\_list}}\}\,]\end{aligned}$$

Because there is no type matching in `print_tuple`, its constraint consists of only one element of type pattern constraint with derived constraints. The type safety of `print_tuple` depends on those of two polymorphic instances of `print_non_list` inside the definition.

8.4.3 Recursive constraints

We also need some recursive constructions of constraints for those of recursive generic values. We introduce *constraint variables* κ and *the fixed point operator* μ to express the *recursive constraints*. For example, `print_rec`, a revised version of `print_list_also` which can print integer or string lists of any depth:

```
generic rec print_rec = case
| ∀δ.δ list → unit ⇒ fun l → List.iter print_rec l
| ∀δ.δ → unit ⇒ print_non_list
```

has a constraint $\mathcal{K}_{\texttt{print\_rec}}$ where,

$$\begin{aligned}\mathcal{K}_{\texttt{print\_rec}} = \mu\kappa.\, [\ & \forall\delta.\delta\ \texttt{list} \to \texttt{unit}\ \{\ \delta \to \texttt{unit} <: \kappa\ \} \\ |\ & \forall\delta.\delta \to \texttt{unit}\ \{\ \delta \to \texttt{unit} <: \mathcal{K}_{\texttt{print\_non\_list}}\ \}]\end{aligned}$$

If a type τ is a valid full instance of this constraint, then τ must satisfy one of the followings:

- $\tau \leq_S \forall\delta.\delta\ \texttt{list} \to \texttt{unit}$ and $S(\delta \to \texttt{unit}) <: \mathcal{K}_{\texttt{print\_rec}}$
- or, $\tau \leq_S \forall\delta.\delta \to \texttt{unit}$ and $S(\delta \to \texttt{unit}) <: \mathcal{K}_{\texttt{print\_non\_list}}$

The recursive constraint is *μ-expanded* (defined formally later) and the constraint variable κ is substituted by the constraint itself.

8.4.4 Constraints must be isomorphic to the structure of the type cases

Since `print_list_also` can be instantiated to only four types `string list` → `unit`, `string` → `unit` and `int` → `unit`, someone may think that he could give a much simpler constraint for `print_list_also`:

$$\begin{aligned}\texttt{print\_list\_also} :\ [\ & \texttt{string list} \to \texttt{unit} \\ |\ & \texttt{int list} \to \texttt{unit} \\ |\ & \texttt{string} \to \texttt{unit} \\ |\ & \texttt{int} \to \texttt{unit}\ ]\end{aligned}$$

Clearly, if a type τ is a valid full instance of the original constraint, then it is also a valid full instance of the second one and vice versa; both constraints express the same set of possible full instances, therefore they could be equivalent. This task – to have "simpler" equivalent constraints — would seem to be easy if we just look this non-recursive example, but once we introduce recursive constraints, the problem becomes much harder. Defining the clear "normal forms of constraints" seems to be almost impossible, due to the existence of recursive constraints. Therefore we do not try to introduce such equivalence between constraints. Keeping constraints as they are, isomorphic to generic value definitions also has a big advantage. We can optimize the type case selection greatly by pre-computing all the type matchings at the compilation phase, because constraints have the complete information how the type matchings will happen. The run time type matching of the IML compilation can be simplified just into accessing arrays which contain each type method expression. We will discuss this *generic flow* optimization later in Chapter 9.

8.4.5 Formal definition

Now we give the formal definition of *generic constraints*:

$$
\begin{array}{rcll}
\mathcal{K} & ::= & [k|\ldots|k] & \text{Type pattern lists} \\
 & | & \kappa & \text{Constraint variables} \\
 & | & \mu\kappa.[k|\ldots|k] & \text{Recursive constraints} \\
 \\
k & ::= & \sigma\ D & \text{Type patterns} \\
 \\
D & ::= & \{d\ \&\ \ldots\ \&\ d\} & \text{Derived constraint lists} \\
 \\
d & ::= & \tau <: \mathcal{K} & \text{Derived constraints}
\end{array}
$$

A type pattern constraint list $[k_1|\ldots|k_n]$ means that it can be fully instantiated to a type which is one of the instances of the type patterns k_i. A type pattern takes a (possibly null) list of *derived constraints* $D = \{\tau_1 <: \mathcal{K}_1\ \&\ \ldots\ \tau_n <: \mathcal{K}_n\}$ which expresses further derived instantiation conditions. The null derived constraint list $\{\,\}$ is usually omitted to write. For example, a constraint $[\sigma\ \{\,\}]$ is written just $[\sigma]$ in short, because it can be fully instantiated to any instance of σ. For example, a non-generic value whose type scheme is σ in the normal typing layer will have a constraint $[\sigma]$, a singleton type pattern list constraint of the type scheme. The constraint $[\sigma]$ is often written simply as σ.

We also have μ *operator* and *constraint variables* κ to express recursively defined constraints. We use the symbol $\mathbb{K}$ for the set of the constraint variables. When a constraint is μ-abstracted, it must be a type pattern constraint $[k_1|\ldots|k_n]$. Type pattern lists $[k_1|\ldots|k_n]$ and derived constraint lists $\{d_1\ \&\ \ldots\ \&\ d_n\}$ are often written as $[k_i]_{i=1}^n$ and $\{d_i\}_{i=1}^n$ respectively in short. Sometimes index ranges of these lists are also omitted as $[k_i]$ and $\{d_i\}$, if it is clear from the context.

A constraint is called *non-generic* when it is a singleton type pattern list $[\sigma\ \{\}]$ with an empty derived constraint list where σ has no bound dynamic type variable. Otherwise, it is called *generic*. The formal definition of this predicate *generic* is given as follows:

$$
\begin{array}{l}
\mathit{generic}([\sigma\ \{\}]) = \mathit{false}\quad \text{when } BDV(\sigma) = \emptyset \\
\text{Otherwise, } \mathit{generic}(\mathcal{K}) = \mathit{true}
\end{array}
$$

Coercion of constraints to type schemes

In Chapter 7, type schemes of user-defined generic values are defined as the least general anti-unifier of the type patterns. On the other hand, generic constraints of generic values have isomorphic structure to their type patterns. Therefore, generic constraints can be regarded as extended type schemes and a downward coercion from constraints to type schemes can be defined as follows:

$$\begin{array}{rcl} \mathit{coerce}([\sigma\ D]) & = & \sigma \\ \mathit{coerce}([\sigma_i\ D_i]_{i=1}^{n>1}) & = & \mathit{p\_antiunif}(\{\sigma_i\}_{i=1}^{n}) \\ \mathit{coerce}(\mu\kappa.\mathcal{K}) & = & \mathit{coerce}(\mathcal{K}) \end{array}$$

If a constraint is a singleton type pattern list, its coercion to a normal type scheme is the type scheme part of the list. Otherwise, a coercion of a constraint is defined as the type pattern unification of type schemes of its patterns. Actually, if we fix the constraint of a value in the constraint typing to $\mathcal{K}$, its type scheme in the normal typing is uniquely determined as $\mathit{coerce}(\mathcal{K})$.

Coercion of typing environment is also definable. Let Γ be a constraint typing environment from identifiers to constraints, then its coercion to a normal typing environment $\mathit{coerce}(\Gamma)$ is given as follows:

$$\mathit{coerce}(\Gamma) = \{(\mathtt{x} \Rightarrow \mathit{cnst}(\mathcal{K})) \mid (\mathtt{x} \Rightarrow \mathcal{K}) \in \Gamma\}$$

8.4.6 Type substitution

The notion of free type variables and type substitutions are naturally extended for constraints. In a type pattern constraint $\sigma\ D$, the free type variables in a derived constraint list D are bound by the quantified variables of σ:

$$\begin{array}{rcl} FV([k_i]_{i=1}^{n}) & = & \bigcup_{i=1}^{n} FV(k_i) \\ FV(\kappa) & = & \emptyset \\ FV(\mu\kappa.\mathcal{K}) & = & FV(\mathcal{K}) \\ FV(\forall V.\tau\ \{d_i\}_{i=1}^{n}) & = & FV(\forall V.\tau) \cup (\bigcup_{i=1}^{n} FV(d_i) \setminus V) \\ FV(\tau <: \mathcal{K}) & = & FV(\tau) \cup FV(\mathcal{K}) \end{array}$$

When we apply a substitution to a constraint $\mathcal{K}$n, we may need to rename the type variables in $\mathcal{K}$ which are bound by quantifications:

$$\begin{array}{rcll} S([k_i]_{i=1}^{n}) & = & [S(k_i)]_{i=1}^{n} & \\ S(\kappa) & = & \kappa & \\ S(\mu\kappa.\mathcal{K}) & = & \mu\kappa.S(\mathcal{K}) & \\ S(\sigma\ \{d_i\}_{i=1}^{n}) & = & S(\sigma)\ \{S(d_i)\}_{i=1}^{n} & \quad \text{rename bound variables if needed} \\ S(\tau <: \mathcal{K}) & = & S(\tau) <: S(\mathcal{K}) & \end{array}$$

8.4.7 Constraint substitutions and μ-expansion

Free and bound constraint variable notions for μ-abstractions can be defined like the normal lambda terms. A *closed constraint* is a constraint which has no free constraint variable. More precisely, the definition of the function KFV to get the free constraint variables of a constraint is:

$$
\begin{array}{rcl}
KFV([k_i]_{i=1}^n) & = & \bigcup_{i=1}^n KFV(k_i) \\
KFV(\kappa) & = & \{\kappa\} \\
KFV(\mu\kappa.\mathcal{K}) & = & KFV(\mathcal{K}) \setminus \{\kappa\} \\
KFV(\sigma\ \{d_i\}_{i=1}^n) & = & \bigcup_{i=1}^n KFV(d_i) \\
KFV(\tau <: \mathcal{K}) & = & KFV(\mathcal{K})
\end{array}
$$

For expansions of the fixed point operator, the μ-abstraction, we need to define *constraint substitutions*. Constraint substitutions are substitutions from constraint variables to constraints. We use ψ meta-variables for these substitutions. The definition of constraint substitutions as functions from a constraint to a constraint is as follows. Like the normal substitutions for type expressions and schemes, bound constraint variables are renamed if necessary:

$$
\psi \quad ::= \quad \{\kappa \Rightarrow \mathcal{K}, \ldots, \kappa \Rightarrow \mathcal{K}\}
$$

$$
\begin{array}{rcll}
\psi([k_i]_{i=1}^n) & = & [\psi(k_i)]_{i=1}^n & \\
\psi(\mu\kappa.\mathcal{K}) & = & \mu\kappa.\psi(\mathcal{K}) & \text{rename the bound variable if necessary} \\
\psi(\sigma\ \{d_i\}_{i=1}^n) & = & \sigma\ \{\psi(d_i)\}_{i=1}^n & \\
\psi(\tau <: \mathcal{K}) & = & \tau <: \psi(\mathcal{K}) &
\end{array}
$$

The *μ-expansion* of a recursive constraint can be defined by using constraint substitutions:

Definition 15 **(μ-expansion of recursive constraint)** *The* μ-expansion *of a recursive constraint $\mu\kappa.\mathcal{K}$ is a constraint $\{\kappa \Rightarrow \mu\kappa.\mathcal{K}\}(\mathcal{K})$. If $\mathcal{K}'$ is a μ-expansion of $\mu\kappa.\mathcal{K}$, we often write $\mu\kappa.\mathcal{K} \stackrel{\mu}{\to} \mathcal{K}'$.*

If a μ-abstracted constraint variable κ does not occur in the body of a recursive constraint $\mu\kappa.\mathcal{K}$, $\mu\kappa.\mathcal{K} \stackrel{\mu}{\to} \mathcal{K}$. This μ-abstraction is useless, therefore we regard the abstracted constraint equivalent to its body. This is the only structural equivalence we introduce in constraints except the trivial α-conversions of quantified type variables and constraint variables:

Definition 16 **(Constraint equivalence)** *Let $\mathcal{K}$ be a constraint and κ be a constraint variable such that $\kappa \notin KFV(\mathcal{K})$. Then we say the recursive constraint $\mu\kappa.\mathcal{K}$ is equivalent to $\mathcal{K}$, and write $\mu\kappa.\mathcal{K} \equiv \mathcal{K}$.*

For example, $\mu\kappa.[\sigma\ \{\}] \equiv [\sigma\ \{\}]$. This equivalence is trivially generalized for constraints:

$$
\begin{array}{rcll}
\mu\kappa.\mathcal{K} & \equiv & \mathcal{K} & \text{when } \kappa \notin KFV(\mathcal{K}) \\
[k_i]_{i=1}^n & \equiv & [k'_i]_{i=1}^n & \text{when } k_i \equiv k'_i \\
\sigma\ \{d_i\}_{i=1}^n & \equiv & \sigma\ \{d'_i\}_{i=1}^n & \text{when } d_i \equiv d'_i \\
\tau <: \mathcal{K} & \equiv & \tau <: \mathcal{K}' & \text{when } \mathcal{K} \equiv \mathcal{K}'
\end{array}
$$

8.5 Valid full instances

In the generic constraint typing, the full instances of generic values are "constrained" by their constraints. In this section, we define the first version of the set of *valid full instance types* of constraints. This is called *strict* valid full instance relationship, since it is too restricted and not practical actually. But because of the simplicity of its definition, we start from this strict predicate.

Unfortunately, due to the existence of recursive constraints, the strict valid full instance predicate is undecidable. Therefore, in Section 8.6, we introduce a restriction to the *admissible* set of generic constraints in the generic constraint typing, so that the predicate can be decidable within the admissible constraints.

Strict valid full instance relationship is too strong, since it assures not only error free evaluation of generic instances, but also termination of recursion on types. The latter is not required for the type safety of generic instances, and unnecessarily rejects generic instances which never terminates but never raises the generic run time error in evaluation. Later in Section 8.7, we remove this superfluous condition from the valid full instance relationship and obtain a *relaxed* version.

8.5.1 Full instance relationship

The instances are called *strictly valid full instances* or strict full instances. The word "strict" means that this relationship is bit too tight in practice. Later we will define another rather "relaxed" relationship called *relaxed full instances* where more types can be valid full instances of constraints, but at first, we would like to start from something simpler.

The strict full instance relationships $\tau \lll: \mathcal{K}$ between a type τ and a constraint $\mathcal{K}$ is defined by the following rules:

$$
\textsf{(k-case)}\ \frac{\begin{array}{c}\tau \not\leq \sigma_1 \quad \ldots \quad \tau \not\leq \sigma_{p-1} \\ \tau \leq \sigma_p \quad \tau \lll: \sigma_p\ D_p \quad p \leq n\end{array}}{\tau \lll: [\sigma_i\ D_i]_{i=1}^n}
$$

$$
\textsf{(k-rec)}\ \frac{\tau \lll: \{\kappa \Rightarrow \mu\kappa.\mathcal{K}\}\mathcal{K}}{\tau \lll: \mu\kappa.\mathcal{K}}
$$

$$
\textsf{(k-derive)}\ \frac{\tau \leq_S \sigma, \quad \forall 1 \leq i \leq n \ \ S(\tau_i) \lll: S(\mathcal{K}_i)}{\tau \lll: \sigma\ \{\tau_i <: \mathcal{K}_i\}_{i=1}^n}
$$

In (k-case), we always select the first pattern $\sigma_p\ D_p$ where $\tau \leq \sigma_p$. The other latter $\sigma_i\ D_i$'s are ignored, even if $\tau \leq \sigma_i$. This corresponds with the pattern matching compilation of generic type case selection (Chapter 3): if there are more than one matches, the first one is always used.

The following algorithm *kinst* is to check whether a given type is a full instance of a constraint or not. For a technical reason for future proofs, μ-expansions are applied lazily. Lazy applications of μ-expansions are stored inside the constraint substitution ψ of the first argument of *kinst*:

$$
\begin{array}{rcl}
kinst(\psi, \tau \ll: [\sigma_i\ D_i]_{i=1}^n) & = & \exists p, \text{where } \tau \not\leq \sigma_1\ \&\ \ldots \tau \not\leq \sigma_{p-1}\ \&\ \tau \leq \sigma_p \\
 & & \&\ kinst(\psi, \tau \ll: \sigma_p\ D_p) \\
kinst(\psi, \tau \ll: \sigma\ \{\tau_i <: \mathcal{K}_i\}_{i=1}^n) & = & \tau \leq_S \sigma\ \&\ \forall_{i=1}^n kinst(\psi, S(\tau_i) \ll: S(\mathcal{K}_i)) \\
kinst(\psi, \tau \ll: \mu\kappa.\mathcal{K}) & = & \text{let } \psi' = \psi \setminus \{\kappa \Rightarrow \psi(\kappa)\} \\
 & & kinst(\psi'\{\kappa \Rightarrow \mu\kappa.\psi'(\mathcal{K})\}, \tau \ll: \mathcal{K}) \\
kinst(\psi, \tau \ll: \kappa) & = & \text{if } \kappa \notin Dom(\psi) \text{ then } \mathit{false} \text{ else } kinst(\psi, \tau \ll: \psi(\kappa))
\end{array}
$$

Theorem 25 (Correctness of *kinst* against $\ll:$) *Let τ be a type, $\mathcal{K}$ be a constraint, and ψ be a constraint substitution contains μ-expansions such that $\psi \circ \psi = \psi$. If and only if $kinst(\psi, \tau \ll: \mathcal{K})$ terminates and returns true, we have $\tau \ll: \psi(\mathcal{K})$.*

PROOF: First we prove $kinst(\psi, \tau \ll: \mathcal{K}) = true \implies \tau \ll: \psi(\mathcal{K})$ by induction over the *kinq* algorithm:

Type pattern constraints Let $\mathcal{K} = [\sigma_i\ D_i]$. Since $kinst(\psi, \tau \ll: \mathcal{K}) = true$, there must be an integer p such that $\tau \not\leq \sigma_1, \ldots, \tau \not\leq \sigma_{p-1}, \tau \leq \sigma_p$ and $kinst(\psi, \tau \ll: \sigma_p\ D_p) = true$. Let $D_p = \{\tau_{pi} <: \mathcal{K}_{pi}\}$ and S be an instantiation such that $\tau \leq_S \sigma_p$, then for all i, $kinst(\psi, S(\tau_{pi}) \ll: S(\mathcal{K}_{pi})) = true$. By induction hypothesis, $S(\tau_{pi}) \ll: \psi(S(\mathcal{K}_{pi}))$. Since $\psi(S(\mathcal{K}_{pi})) = S(\psi(\mathcal{K}_{pi}))$, using (k-derive) we obtain $\tau \ll: \sigma_p\ \{\tau_{pi} <: \psi(\mathcal{K}_{pi})\}$ and by (k-case), it leads to $\tau \ll: [\sigma_i\ \psi(D_i)]$, which is equal to $\tau \ll: \psi(\mathcal{K})$.

Constraint variable Let $\mathcal{K} = \kappa$. Since $kinst(\psi, \tau \ll: \kappa) = true$, ψ must have a binding for κ, and $kinst(\psi, \tau \ll: \psi(\kappa)) = true$. By induction hypothesis, $\tau \ll: (\psi \circ \psi)(\kappa)$. From the idempotency of ψ, this is equal to $\tau \ll: \psi(\kappa)$.

Recursive constraints Let $\mathcal{K} = \mu\kappa'.\mathcal{K}'$. From $kinst(\psi, \tau \ll: \mu\kappa.\mathcal{K}') = true$, If we let $\psi' = \psi \setminus \{\kappa' \Rightarrow \psi(\kappa')\}$ and $\psi'' = \psi'\{\kappa' \Rightarrow \mu\kappa'.\psi'(\mathcal{K}')\}$,

$$kinst(\psi'', \tau \ll: \mathcal{K}') = true$$

From the idempotency of ψ, ψ' is also idempotent and ψ'', too. Therefore we can use the induction hypothesis and obtain $\tau \ll: \psi''(\mathcal{K}')$.

Since $\psi'' = \psi'\{\kappa' \Rightarrow \mu\kappa'.\psi'(\mathcal{K}')\} = \{\kappa' \Rightarrow \mu\kappa'.\psi'(\mathcal{K}')\} \circ \psi'$ from its idempotency, $\tau \ll: \{\kappa' \Rightarrow \mu\kappa'.\psi'(\mathcal{K}')\}(\psi'(\mathcal{K}'))$. Using (k-rec) we obtain $\tau \ll: \mu\kappa'.\psi'(\mathcal{K}')$, where $\mu\kappa'.\psi'(\mathcal{K}') = \psi(\mu\kappa'.\mathcal{K}')$.

Second, we prove the converse, $\tau \ll: \psi(\mathcal{K}) \Longrightarrow kinst(\psi, \tau \ll: \mathcal{K}) = true$ by induction over the derivation of the predicate:

(k-case) It is the case when $\psi(\mathcal{K}) = [\sigma_i\ D_i]$. The last step of its proof is as follows:

$$\frac{\tau \not\leq \sigma_1 \quad \ldots \quad \tau \not\leq \sigma_{p-1} \tau \leq \sigma_p \quad p \leq n \quad \tau \ll: \sigma_p\ D_p}{\tau \ll: [\sigma_i\ D_i]_{i=1}^n}$$

If we let $D_p = \{\tau_j <: \mathcal{K}_j\}_{j=1}^m$, The predicate $\tau \ll: \sigma_p\ D_p$ must be proven by (k-derive) as follows:

$$\frac{\tau \leq_S \sigma_p, \quad \forall 1 \leq j \leq m \ \ S(\tau_j) \ll: S(\mathcal{K}_j)}{\tau \ll: \sigma\ \{\tau_j <: \mathcal{K}_j\}_{j=1}^m}$$

The constraints $\mathcal{K}_j$ are not be affected by ψ, since they are sub-constraints of $\psi(\mathcal{K})$ and ψ is idempotent. Therefore the predicate $S(\tau_j) \ll: S(\mathcal{K}_j)$ can be rewritten to $S(\tau_j) \ll: \psi(S(\mathcal{K}_j))$. Applying the induction hypothesis to these predicates, we have $kinst(\psi, S(\tau_j) \ll: S(\mathcal{K}_j)) = true$. They leads to $kinst(\psi, \tau \ll: \sigma_p\ D_p) = true$. From it and $\tau \not\leq \sigma_1, \ \ldots, \ \tau \not\leq \sigma_{p-1}$, we obtain $kinst(\psi, \tau \ll: \mathcal{K})$.

(k-rec) It is when $\psi(\mathcal{K}) = \mu\kappa'.\mathcal{K}'$. The proof ends as follows:

$$\frac{\tau \ll: \{\kappa' \Rightarrow \mu\kappa'.\mathcal{K}'\}\mathcal{K}'}{\tau \ll: \mu\kappa'.\mathcal{K}'}$$

When $\mathcal{K}$ is not a constraint variable

If $\mathcal{K}$ is not a constraint variable, it must be a μ-abstraction and can be written as $\mathcal{K} = \mu\kappa'.\mathcal{K}''$, where $\psi(\mathcal{K}'') = \mathcal{K}'$. The premise is rewritten as

$$\tau \ll: \{\kappa' \Rightarrow \mu\kappa'.\psi(\mathcal{K}'')\}(\psi(\mathcal{K}''))$$

Since we can suppose κ' is out of ψ, $\{\kappa' \Rightarrow \mu\kappa'.\psi(\mathcal{K}'')\} = \{\kappa' \Rightarrow \psi(\mu\kappa'.\mathcal{K}'')\}$ and

$$\{\kappa' \Rightarrow \psi(\mu\kappa'.\mathcal{K}'')\} \circ \psi = \psi + \{\kappa' \Rightarrow \mu\kappa'.\psi(\mathcal{K}'')\}$$

Therefore $\tau \ll: (\psi + \{\kappa' \Rightarrow \mu\kappa'.\psi(\mathcal{K}'')\})\mathcal{K}''$. From this and using the induction hypothesis, we have $kinst(\psi + \{\kappa' \Rightarrow \mu\kappa'.\psi(\mathcal{K}'')\}, \tau \ll: \mathcal{K}'') = true$. This is equal to

$$kinst(\psi, \tau \ll: \mu\kappa'.\mathcal{K}'') = true$$

When $\mathcal{K}$ is a constraint variable

If $\mathcal{K}$ is a constraint variable κ, $\psi(\kappa) = \mu\kappa'.\mathcal{K}'$. Since ψ is a constraint substitution made of μ-expansions, $\kappa = \kappa'$. The premise is rewritten to

$$\tau \ll: \psi(\mathcal{K}')$$

Applying the induction hypothesis to the premise, we have: $kinst(\psi, \tau <: \mathcal{K}') = true$.

On the other hand,

$$\begin{array}{rcl} kinst(\psi, \tau <: \kappa) & = & kinst(\psi, \tau <: \psi(\kappa)) \\ & = & kinst(\psi, \tau <: \mu\kappa'.\mathcal{K}') \\ & = & kinst(\psi'', \tau <: \mathcal{K}') \end{array}$$

where $\psi'' = \psi'\{\kappa'{\Rightarrow}\mu\kappa'.\psi'(\mathcal{K}')\}$ and $\psi' = \psi \setminus \{\kappa'{\Rightarrow}\psi(\kappa')\}$. From the idempotency of ψ and $\psi' = \psi \setminus \{\kappa'{\Rightarrow}\mu\kappa'.\mathcal{K}'\}$, $\psi'' = \psi$. Therefore $kinst(\psi, \tau <: \kappa) = kinst(\psi, \tau <: \mathcal{K}') = true$.

□

Unfortunately this $\ll:$ and *kinst* algorithm have a serious problem. Because of (k-rec) recursive rule, reconstruction of derivations for recursive constraints sometimes never terminates. For example, let $\mathcal{K} = \mu\kappa.[\forall\delta.\delta\ \{\delta * \delta <: \kappa\}]$. Then $kinst(\emptyset, \tau \ll: \mathcal{K})$ never terminates, since it recursively calls $kinst(\psi, \tau * \tau \ll: \mathcal{K})$, $kinst(\psi', (\tau * \tau) * (\tau * \tau) \ll: \mathcal{K})$,... infinitely. More generally, this full instance relationship $\ll:$ is *undecidable* [DRW95]. Therefore we need to restrict the generic constraints so that the relationship can be decidable in the restricted set of constraints.

8.6 Admissible constraints

We define *the size of a type*, an arithmetic expression which denotes complexity of a type and restrict the set of constraints to *admissible* ones so that in the recursive checks of *kinst* for an admissible constraint, type sizes of the arguments can decrease and therefore terminate.

8.6.1 Type sizes

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

$$\begin{array}{rcl} |base| & = & 1 \\ |\alpha| & = & 1 \\ |\delta_i| & = & x_i \\ |\tau_1 \to \tau_2| & = & \sum_{i=1}^{2} |\tau_i| + 1 \\ |\tau_1 * \tau_2| & = & \sum_{i=1}^{2} |\tau_i| + 1 \end{array}$$

For each dynamic type variable δ_i, $|\delta_i|$ returns a variable x_i, specific to the type variable. It is clear that for any type τ, $|\tau|$ can be uniquely represented in the form of $\sum_i a_i x_i + c$, where $a_i > 0$ and $c \geq 0$.

The *equality* of type sizes is the arithmetic expression equality. We write $|\tau| = |\tau'|$ if both the expressions are same arithmetically. The *type size ordering* $<$ is defined as their ordering as equations by ranging all the variables x_i appears in the equations more than 0:

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

Checking $|\tau| < |\tau'|$ is done by proving this inequality by the following: Let $|\tau| = \sum_i a_i x_i + c$ and $|\tau'| = \sum_i a'_i x_i + c'$, supposing if x_i does not appear in $|\tau|$ (*resp.* $|\tau'|$) then $a_i = 0$ (*resp.* $a'_i = 0$). Then what we have to prove is:

$$\forall x_i > 0, \quad |\tau'| - |\tau| = \sum_i (a'_i - a_i)x_i + c' - c > 0,$$

and it is checked easily as follows:

- If there exists i such that $a'_i - a_i < 0$ then the inequality is not verified.
- Otherwise, the inequality is equivalent to $\sum_i (a'_i - a_i) + c' - c > 0$, which we can easily verify.

Example

$|\delta_1 * \delta_2| = x_1 + x_2 + 1$ and $|\delta_1 \to \delta_1| = x_1 + x_2 + 1$. Therefore

$$|\delta_1 * \delta_2| = |\delta_1 \to \delta_1|$$

$|\delta_1 \to \mathtt{unit}| = x_1 + 2$ and $|\delta_1 * \delta_1 \to \mathtt{unit}| = 2x_1 + 3$. Therefore

$$|\delta_1 \to \mathtt{unit}| < |\delta_1 * \delta_1 \to \mathtt{unit}|$$

□

Lemma 21 (Dynamic substitution preserves type size ordering) *Type substitutions whose domains are restricted within the dynamic type variables preserve the ordering of the type expressions: for a substitution S such that $Dom(S) \subset \mathbb{D}$, if $|\tau| < |\tau'|$ then $|S(\tau)| < |S(\tau')|$.*

PROOF: Suppose $|\tau| < |\tau'|$ and let $|\tau| = \sum_i a_i x_i + c$ and $|\tau'| = \sum_i a'_i x_i + c'$ where $a_i \geq 0$ and $a'_i \geq 0$ and sharing the same set of x_i. Then $|S(\tau)|$ and $|S(\tau')|$ can be obtained by easy arithmetic manipulation:

$$|S(\tau)| = \sum_i a_i |S(\delta_i)| + c, \quad |S(\tau')| = \sum_i a'_i |S(\delta_i)| + c'$$

Since for all the variables $x'_i > 0$ in $|S(\tau)|$ and $|S(\tau')|$, $|S(\delta_i)| > 0$, the inequation $|S(\tau)| < |S(\tau')|$ is equivalent to $|\tau| < |\tau'|$ when $x_i = |S(\delta_i)|$. which is one of the cases of $\forall x_i > 0, |\tau| < |\tau'|$. □

Type size of substitution $|S|$

The type size of a substitution S, $|S|$ is defined using the type size $|\tau|$:

$$|S| = \sum_{\delta \in Dom(S)} |S(\delta)|$$

Type size of type instantiation $|\tau \leq \sigma|$

To make the discussion smoothly, we also need to define the *size of a predicate* $\tau \leq \sigma$. The size of a predicate is equivalent to the size of the type instantiation:

$$|\tau \leq \sigma| = |S| \quad \text{where } \tau \leq_S \sigma$$

We here suppose $Dom(S) = BV(\sigma)$ to have a unique size. Of course to be well defined, τ must be an instance of σ.

From the view of the compilation of extensional polymorphism, the type size of instantiation $|\tau \leq \sigma|$ indicates the type size of run time type dispatching for a generic full instance $\mathtt{x}_\sigma : \tau$:

$$\lfloor \mathtt{x}_\sigma : \tau \rfloor_\emptyset = \mathtt{x} \ «S(\delta_1)» \ \ldots \ «S(\delta_n)» \quad \text{where } \forall V.\tau' = \sigma \text{ and } [\delta_i] = \mathit{Vlist}(V \cap \mathbb{D}, \tau')$$

$|\tau \leq \sigma|$ is equal to the sum of the type sizes of $S(\delta_i)$.

Lemma 22 (Substitution preserves the order of $|\tau \leq \sigma|$) *Let τ and τ' be types, and σ and σ' be type schemes. If $|\tau \leq \sigma| < |\tau' \leq \sigma'|$, then for any substitution S_0, $|S_0(\tau) <: S_0(\sigma)| < |S_0(\tau') <: S_0(\sigma')|$.*

PROOF: From the well-definedness of $|\tau \leq \sigma|$ and $|\tau' \leq \sigma'|$, there are substitutions S and S' such that $\tau \leq_S \sigma$, $\tau' \leq_{S'} \sigma'$ and:

$$\sum_{\delta \in BDV(\sigma)} |S(\delta)| \quad < \quad \sum_{\delta' \in BDV(\sigma')} |S'(\delta')|$$

From Lemma 4, $S_0(\tau) \leq S_0(\sigma)$, $S_0(\tau') \leq S_0(\sigma')$ and:

$$\begin{array}{rclcl} |S_0(\tau) <: S_0(\sigma)| & = & \sum_{\delta \in BDV(S_0(\sigma))} |(S_0 \circ S)(\delta)| & = & \sum_{\delta \in BDV(\sigma)} |(S_0 \circ S)(\delta)| \\ |S_0(\tau') <: S_0(\sigma')| & = & \sum_{\delta' \in BDV(S_0(\sigma'))} |(S_0 \circ S')(\delta')| & = & \sum_{\delta' \in BDV(\sigma')} |(S_0 \circ S')(\delta')| \end{array}$$

Because S is an admissible substitution, for any dynamic type variable δ, $FV(S(\delta)) \in \mathbb{D}$. Therefore if we let $S'_0 = \{(\delta \Rightarrow \tau) \in S_0\}$, a substitution same as S_0 whose domain is restricted only to the dynamic type variables, $|(S_0 \circ S)(\delta)| = |(S'_0 \circ S)(\delta)|$. By using S'_0, now the sizes are:

$$\begin{array}{rcl} |S_0(\tau) <: S_0(\sigma)| & = & \sum_{\delta \in BDV(\sigma)} |(S'_0 \circ S)(\delta)| \\ |S_0(\tau') <: S_0(\sigma')| & = & \sum_{\delta' \in BDV(\sigma')} |(S'_0 \circ S')(\delta')| \end{array}$$

Because $Dom(S'_0) \subset \mathbb{D}$, from Lemma 21 and the linearity of type sizes,

$$\sum_{\delta \in BDV(\sigma)} |(S'_0 \circ S)(\delta)| \quad < \quad \sum_{\delta' \in BDV(\sigma')} |(S'_0 \circ S)(\delta')|$$

and it is equal to $|S_0(\tau) \leq S_0(\sigma)| < |S_0(\tau') \leq S_0(\sigma')|$ $\square$

Lemma 23 *Let σ_1 and σ_2 be type schemes and τ be a type expression such that $\tau \leq \sigma_1$ and $\tau \leq \sigma_2$. If σ_1 is more general than σ_2, i.e. $inst(\sigma_2) \leq \sigma_1$, $|\tau \leq \sigma_2| \leq |\tau \leq \sigma_1|$.*

PROOF: Let $\tau \leq_{S_1} \sigma_1$ and $\tau \leq_{S_2} \sigma_2$. Let $\sigma_2 = \forall V_2.\tau_2$ and $\tau_2 \leq_S \sigma_1$. Then $S_1 = S_2 \circ S$. $|\tau \leq \sigma_1| = \sum_{\delta \in BDV(\sigma_1)} |S_2(S(\delta))|$. Because for any bound dynamic variable δ_2 of σ_2, there must be a bound dynamic variable δ_1 of σ_1 such that $\delta_2 \in FV(S(\delta_1))$,

$$|\tau \leq \sigma_2| = \sum_{\delta \in BDV(\sigma_2)} |S_2(\delta)| \leq \sum_{\delta \in BDV(\sigma_1)} |S_2(S(\delta))| = |\tau \leq \sigma_1|$$

□

Size of full instance predicate $|\tau <: \mathcal{K}|$

The type size of instantiation $|\tau \leq \sigma|$ is rewritten to a form of $|\tau <: \mathcal{K}|$ by using *coerce*:

$$|\tau <: \mathcal{K}| = |\tau \leq \mathit{coerce}(\mathcal{K})|$$

To be well-defined, τ must be an instance of $\mathit{coerce}(\mathcal{K})$, but it need not to be a valid full instance type of $\mathcal{K}$.

8.6.2 Strictly admissible constraints for $\ll$:

Using the type size $|\cdot|$, we now define the set of the *strictly admissible constraints*. Roughly speaking, if a generic value is typed with a strictly admissible constraint, the sizes of run time type dispatching must *strictly* decrease in its recursive calls. This strict type size decreasing of recursive run time type dispatching ensures termination of recursion on types.

The following predicate *adm* defines the set of the strictly admissible constraints formally. When the constraint is a variable κ, instead of its μ-expansion, it returns *true* immediately when the corresponding recursive constraint $\mu\kappa.\mathcal{K}$ has been checked already. Otherwise *adm* returns *false*. For this purpose, the predicate takes an argument of a constraint substitution ψ, which keeps track of checkings of recursive constraints:

$$\begin{array}{rcl}
\mathit{adm}(\psi, [\forall V_i.\tau_i \ \{\tau_{ij} <: \mathcal{K}_{ij}\}]) & = & \forall i, j, \ \ \mathit{adm}(\psi, \psi(\mathcal{K}_{ij})) = \mathit{true} \\
 & & \quad \text{if } \mathit{Dom}(\psi) \cap \mathit{KFV}(\mathcal{K}_{ij}) \neq \emptyset \\
 & & \quad \text{then } |\tau_i <: [\forall V_i.\tau_i \ \{\tau_{ij} <: \mathcal{K}_{ij}\}]| > |\tau_{ij} <: \psi(\mathcal{K}_{ij})| \\
\mathit{adm}(\psi, \mu\kappa.\mathcal{K}) & = & \text{let } \psi' = \psi \setminus \{\kappa \Rightarrow \psi(\kappa)\} \\
 & & \mathit{adm}(\psi'\{\kappa \Rightarrow \mu\kappa.\psi'(\mathcal{K})\}, \mathcal{K}) \\
\mathit{adm}(\psi, \kappa) & = & \kappa \in \mathit{Dom}(\psi)
\end{array}$$

A constraint $\mathcal{K}$ is *strictly admissible* if and only if $\mathit{adm}(\emptyset, \mathcal{K}) = \mathit{true}$. Since *adm* does not expand μ-abstractions, its termination is trivial.

Lemma 24 (Substitution preserves strict admissibility) *Any type substitution S preserves the admissibility of constraints: if $adm(\psi, \mathcal{K}) = true$, then $adm(\psi, S(\mathcal{K})) = true$.*

PROOF: (Sketch) The part S really affects is in the case of $adm(\psi, \mathcal{K})$ which compares the sizes of predicates $\tau <: \mathcal{K}$. From Lemma 22, this type ordering is kept by any substitution S. □

Lemma 25 (μ-expansion preserves strict admissibility) *Let $\mathcal{K}, \mathcal{K}'$ be constraints, ψ and φ be μ-expansion constraint substitutions and κ be a constraint variable, such that*

$$\kappa \notin Dom(\psi)$$
$$\varphi = \{\kappa \Rightarrow \mathcal{K}\}$$
$$(\psi + \varphi) \circ (\psi + \varphi) = \psi + \varphi$$

Then, if $adm(\psi + \varphi, \mathcal{K}) = true$ and $adm(\psi + \varphi, \mathcal{K}') = true$, $adm(\psi + \varphi, \varphi(\mathcal{K}')) = true$

PROOF: By induction over the constraint $\mathcal{K}$.

Constraint variable If $\mathcal{K}' = \kappa$ then $adm(\psi+\varphi, \varphi(\kappa)) = adm(\psi+\varphi, \mathcal{K}) = true$. Otherwise, when $\mathcal{K} = \kappa' \neq \kappa$, $adm(\psi + \varphi, \varphi(\mathcal{K}')) = adm(\psi + \varphi, \mathcal{K}') = true$.

Type patterns Let $\mathcal{K}' = [\forall V_i.\tau_i \ \{\tau_{ij} <: \mathcal{K}_{ij}\}]$. From $adm(\psi + \varphi, \mathcal{K}') = true$, for all i and j, we have

$$adm(\psi + \varphi, \mathcal{K}_{ij}) = true$$
$$\text{if } (Dom(\psi) + \{\kappa\}) \cap KFV(\mathcal{K}_{ij}) \neq \emptyset \text{ then } |\tau_i <: \mathcal{K}'| > |\tau_{ij} <: (\psi + \varphi)(\mathcal{K}_{ij})|$$

From the induction hypothesis, $adm(\psi + \varphi, \varphi(\mathcal{K}_{ij})) = true$.

$KFV(\varphi(\mathcal{K}_{ij})) = KFV(\mathcal{K}_{ij}) \setminus \{\kappa\} \cup KFV(\mathcal{K})$. Since $\psi + \varphi$ is idempotent, $Dom(\psi + \varphi) \cap KFV(\mathcal{K}) = \emptyset$. Therefore,

$$\begin{aligned}
&(Dom(\psi + \{\kappa\}) \cap KFV(\varphi(\mathcal{K}_{ij})) \\
&= (Dom(\psi) + \{\kappa\}) \cap (KFV(\mathcal{K}_{ij}) \setminus \{\kappa\} \cup KFV(\mathcal{K})) \\
&= (Dom(\psi) + \{\kappa\}) \cap (KFV(\mathcal{K}_{ij}) \setminus \{\kappa\}) \\
&\subset (Dom(\psi) + \{\kappa\}) \cap KFV(\mathcal{K}_{ij})
\end{aligned}$$

This means, if $Dom(\psi + \varphi\} \cap KFV(\varphi(\mathcal{K}_{ij})) \neq \emptyset$, $Dom(\psi + \varphi) \cap KFV(\mathcal{K}_{ij}) \neq \emptyset$ and therefore $|\tau_i <: \mathcal{K}'| > |\tau_{ij} <: (\psi + \varphi)(\mathcal{K}_{ij})|$. From the idempotency of $\psi + \varphi$, the right hand side of the inequation is equal to $|\tau_{ij} <: (\psi + \varphi)(\varphi(\mathcal{K}_{ij}))|$.

From these, $adm(\psi + \varphi, \varphi(\mathcal{K}')) = true$.

Recursive constraints Let $\mathcal{K}' = \mu\kappa'.\mathcal{K}''$. Then $adm(\psi+\varphi, \mu\kappa'.\mathcal{K}'') = adm(\psi'', \mathcal{K}'') = true$, where

$$\psi' = (\psi + \varphi) \setminus \{\kappa' \Rightarrow (\psi + \varphi)(\kappa')\}$$
$$\psi'' = \psi'\{\kappa' \Rightarrow \mu\kappa'.\psi'(\mathcal{K}'')\}$$

$\psi'' \circ \psi'' = \psi''$ is easily verified.

Further proof depends on where κ' is κ or not:

When $\kappa' \neq \kappa$
$\psi' = (\psi + \varphi) \setminus \{\kappa' \Rightarrow \psi(\kappa')\}$ and $\psi'' = \psi\{\kappa' \Rightarrow \mu\kappa'.\psi'(\mathcal{K}'')\} + \varphi$.

$adm(\psi'', \mathcal{K}) = true$ is verified since $\mathcal{K} = \varphi(\kappa)$ and the idempotency of the constraint substitutions; the modification to the binding of κ' never affects the admissibility of $\mathcal{K}$, therefore $adm(\psi''.\mathcal{K}) = adm(\psi + \varphi, \mathcal{K}) = true$.

From induction hypothesis, we have $adm(\psi'', \varphi(\mathcal{K}'')) = true$.

From the definition and idempotency of ψ' and φ, $\psi'(\mathcal{K}'') = \psi'(\varphi(\mathcal{K}''))$. Therefore, $\psi'' = \psi\{\kappa' \Rightarrow \mu\kappa'.\psi'(\varphi(\mathcal{K}''))\} + \varphi$. From $adm(\psi'', \varphi(\mathcal{K}'')) = true$ and this, we obtain

$$adm(\psi + \varphi, \mu\kappa'.\varphi(\mathcal{K}'')) = adm(\psi + \varphi, \varphi(\mu\kappa'.\mathcal{K}'')) = true$$

When $\kappa' = \kappa$
$adm(\psi + \varphi, \varphi(\mu\kappa'.\mathcal{K}'')) = adm(\psi + \varphi, \mu\kappa'.\mathcal{K}'') = true$.

□

A generic constraint $\mathcal{K}$ is *strictly admissible* if and only if $adm(\mathcal{K}) = true$. Since *adm* does not expand μ-abstractions in its recursive calls, its termination is trivial.

Theorem 26 $\forall\kappa \in Dom(\psi), adm(\psi, \psi(\kappa)) = true$*, and* $\psi \circ \psi = \psi$*,*

Theorem 27 (Strict admissibility assures the termination of $kinst(\psi, \tau \lll: \mathcal{K})$**)** *Let* τ *be a type expression,* $\mathcal{K}$ *be a generic constraint,* ψ *be a constraint substitution such that* $\psi \circ \psi = \psi$*.*

If $adm(\psi, \mathcal{K}) = true$*,* $kinst(\psi, \tau \lll: \mathcal{K})$ *always terminates.*

Proof: By induction over the algorithm, we prove that type size $|\tau <: \psi(\mathcal{K})|$ strictly decreases at the case of type patterns of the recursive calls of $kinst(\psi, \tau \lll: \mathcal{K})$, when $\mathcal{K}$ has constraint variables which can be μ-expanded by ψ. In other case, we prove sizes of predicates are preserved.

Because the size of predicate $|\tau <: \mathcal{K}| > 0$ for any type,this assures the termination of the algorithm *kinst*.

Type patterns Let $\mathcal{K} = [\forall V_i.\tau_i\ \{\tau_{ij} <: \mathcal{K}_{ij}\}]$. If there is no i such that $\tau \leq \forall V_i.\tau_i$, *kinst* immediately returns *false*.

Otherwise, let p is the first index such that $\tau \leq \forall V_p.\tau_p$. We let S be the instantiation $\tau \leq_S \forall V_p.\tau_p$. Then $kinst(\psi, \tau \lll: \mathcal{K}) = \forall j, kinst(\psi, S(\tau_{pj}) \lll: S(\mathcal{K}_{pj}))$.

From $adm(\psi, \mathcal{K}) = true$, we have the following:

$$\begin{array}{l} adm(\psi, \psi(\mathcal{K}_{pj})) = true \\ \text{if } Dom(\psi) \cap KFV(\mathcal{K}_{pj}) \neq \emptyset \text{ then } |\tau_p <: \mathcal{K}| > |\tau_{pj} <: \psi(\mathcal{K}_j)| \end{array}$$

From Lemma 22, the inequation is rewritten as follows:

$$|\tau <: \mathcal{K}| > |S(\tau_{pj}) <: S(\psi(\mathcal{K}_j))|$$

Since $Dom(S) = V_p$, by renaming V_p, we can suppose $S(\psi(\mathcal{K}_j)) = \psi(S(\mathcal{K}_j))$. $\mathcal{K}$ is a type pattern constraint, ψ does not affect the type size of the predicate $|\tau <: \mathcal{K}|$: $|\tau <: \mathcal{K}| = |\tau <: \psi(\mathcal{K})|$. From these,

$$|\tau <: \psi(\mathcal{K})| > |S(\tau_{pj}) <: \psi(S(\mathcal{K}_j))|$$

Since $adm(\psi, \mathcal{K}_{pj}) = true$, from Lemma 24, $adm(\psi, S(\mathcal{K}_{pj}))$. We can use the induction hypothesis for $kinst(\psi, S(\tau_{pj}) \ll: S(\mathcal{K}_{pj}))$ for all j and assures their termination.

Recursive constraints Let $\mathcal{K} = \mu\kappa.[k_i]$. $kinst(\psi, \tau \ll: \mu\kappa.[k_i]) = kinst(\psi'', \tau \ll: [k_i])$, where $\psi' = \psi \setminus \{\kappa \Rightarrow \psi(\kappa')\}$ and $\psi'' = \psi'\{\kappa \Rightarrow \mu\kappa.\psi'([k_i])\}$. Since

$$coerce(\psi(\mu\kappa.[k_i])) = coerce(\psi(\mu\kappa.\psi''[k_i])) = coerce([k_i])$$

The type size of the arguments of $kinq$ does not change: $|\tau <: \psi(\mu\kappa.[k_i])| = |\tau <: \psi''([k_i])|$.

From the idempotency of ψ, ψ' is also idempotent, therefore $\psi'\{\kappa \Rightarrow \mu\kappa.\psi(\mathcal{K}')\}$, too.

From $adm(\psi, \mu\kappa.[k_i]) = true$, $adm(\psi'', [k_i]) = true$. From Lemma 25, $adm(\psi'', \psi'[k_i]) = true$. This is equal to:

$$\begin{array}{rcl} adm(\psi'', \psi'([k_i])) & = & adm(\psi'', \mu\kappa.\psi'([k_i])) \\ & = & adm(\psi'', \psi''(\kappa)) \end{array}$$

For constraint variables $\kappa' \neq \kappa$, $adm(\psi'', \psi''(\kappa')) = adm(\psi, \psi(\kappa')) = true$ is easily checked. Therefore $\forall \kappa' \in Dom(\psi''), adm(\psi'', \psi''(\kappa')) = true$ and we can use the induction hypothesis for the termination of $kinst(\psi'', \tau \ll: [k_i])$.

Constraint variables Let $\mathcal{K} = \kappa$. From $adm(\psi, \kappa) = true$, we have $\kappa \in Dom(\psi)$ and $\psi(\kappa)$ is well defined. Therefore, $kinst(\psi, \tau \ll: \kappa) = kinst(\psi, \tau \ll: \psi(\kappa))$. Since ψ is idempotent, $|\tau <: \psi(\kappa)| = |\tau <: \psi(\psi(\kappa))|$. Since $adm(\psi, \psi(\kappa)) = true$, we can use the induction hypothesis for $kinst(\psi, \tau \ll: \psi(\kappa))$ and obtain its termination.

$\square$

The following theorem is a special case of Theorem 27:

Theorem 28 (Strict admissibility assures the termination of $kinst(\emptyset, \tau \ll: \mathcal{K})$**)** *Let τ be a type expression and $\mathcal{K}$ be a generic constraint. If $adm(\mathcal{K}) = true$, then $kinst(\emptyset, \tau \ll: \mathcal{K})$ terminates.*

PROOF: This is a case of $\psi = \emptyset$ of Theorem 27 □

Thus, restricting constraints to admissible ones, such that $adm(\mathcal{K}) = true$, we make the predicate $\tau \ll: \mathcal{K}$ decidable and $kinst(\emptyset, \tau \ll: \mathcal{K})$ is the calculation for the predicate.

8.7 Relaxed full instances

The strict full instance relationship defined by construction using the constraint instantiation rules (k-case), (k-rec) and (k-derive) in the previous section is actually too strict in practice. We find difficulties when we define recursive derived generic values. Since their constraints cannot be strictly admissible, under the strict admissible constraint restriction, they cannot be defined in the type level. Therefore we must find a more relaxed restriction of constraints which permits derived recursive definitions, but still ensures the decidability of the full instance relationship.

8.7.1 Recursive let generics are not strictly admissible

Suppose we define a generic multiplication function `mult` for integers and floats:

```
generic mult = case
| int → int → int ⇒ ( * )
| float → float → float ⇒ ( *. )
;;
val mult : [int → int → int | float → float → float]
```

Then, define a recursive derived generic function `power` using this `mult` as follows:

```
let rec power n = function (* calculates pⁿ *)
| 1 → n
| p → mult n (power n (p - 1))
;;
```

Because `power` has a recursive occurrence of itself inside the definition with the type annotation $\delta \to \texttt{int} \to \delta$, `power` will have the following recursive constraint:

$$\mathcal{K}_{\texttt{power}} = \mu\kappa.[\forall\delta.\delta \to \texttt{int} \to \delta \ \{\delta \to \texttt{int} \to \delta <: \kappa\}]$$

But as far as we use $\ll:$ as full instance relationship, this function cannot be instantiated to any type. For example, $\texttt{int} \to \texttt{int} \to \texttt{int} \ll: \mathcal{K}_{\texttt{power}}$ is not provable. In addition, the algorithm *kinst* which checks valid full instantiation does not terminate with this type:

$$
\begin{array}{l}
\mathit{kinst}(\emptyset, \texttt{int} \to \texttt{int} \to \texttt{int} \ll: \mathcal{K}_{\texttt{power}}) \\
= \mathit{kinst}(\{\kappa {\Rightarrow} \mathcal{K}_{\texttt{power}}\}, \texttt{int} \to \texttt{int} \to \texttt{int} \ll: [\forall \delta.\delta \to \texttt{int} \to \delta\ \{\delta \to \texttt{int} \to \delta <: \kappa\}]) \\
= \mathit{kinst}(\{\kappa {\Rightarrow} \mathcal{K}_{\texttt{power}}\}, \texttt{int} \to \texttt{int} \to \texttt{int} \ll: \kappa) \\
= \mathit{kinst}(\{\kappa {\Rightarrow} \mathcal{K}_{\texttt{power}}\}, \texttt{int} \to \texttt{int} \to \texttt{int} \ll: \mathcal{K}_{\texttt{power}}) \\
= \mathit{kinst}(\{\kappa {\Rightarrow} \mathcal{K}_{\texttt{power}}\}, \texttt{int} \to \texttt{int} \to \texttt{int} \ll: [\forall \delta.\delta \to \texttt{int} \to \delta\ \{\delta \to \texttt{int} \to \delta <: \kappa\}]) \\
= \ldots \text{ (loops forever)}
\end{array}
$$

The algorithm loops forever due to μ-expansions, therefore we cannot find it is valid or invalid. This `power` must be rejected by the system using $\ll:$, and actually $\mathcal{K}_{\texttt{power}}$ is not a strictly admissible.

We also have such an example in the recursive generic bindings:

```
generic rec add =
| int → int → int ⇒ (+)
| float → float → float ⇒ (+.)
| δ * δ → δ ⇒ fun (x,y) → add x y
;;
```

Another variant of `add` generic function. In this time, the function is also usable in an uncurried form: `add (1,2)`, `add (1.0, 2.3)` and so on. The constraint of `add`, $\mathcal{K}_{\texttt{add}}$ is:

$$
\mathcal{K}_{\texttt{add}} = \mu\kappa. \quad
\begin{array}{l}
[\ \texttt{int} \to \texttt{int} \to \texttt{int} \\
\mid \texttt{float} \to \texttt{float} \to \texttt{float} \\
\mid \forall \delta.\delta * \delta \to \delta\ \{\delta \to \delta \to \delta <: \kappa\}]
\end{array}
$$

In the last type case, the internal `add` has a type annotation $\delta \to \delta \to \delta$. Therefore,

$$
|\delta{*}\delta \to \delta <: \mathcal{K}_{\texttt{add}}| = |\delta \to \delta \to \delta <: \mathcal{K}_{\texttt{add}}| = 3x + 1
$$

which means that $\mathit{adm}(\emptyset, \mathcal{K}_{\texttt{add}}) = \mathit{false}$, this constraint of `add` is not strictly admissible.

This situation is undesirable, since generally speaking, *all* the recursive derived generic values defined by let expressions and some by generic expressions do not have strictly admissible constraints, even when they have no risk of generic run time errors. This problem is due to over-restricted rules of the full instance relationship $\ll:$. The predicate $\tau \ll: \mathcal{K}$ assures not only the soundness of type recursion concerned with this predicate, but also the termination of the type recursion. What we actually want is only the first: the soundness. The termination is not necessary. Generic values which will never terminate should also be welcome if we can be sure they never fail, just like well-typed recursive functions may never terminates in normal ML. Note also that non-termination of type recursion does not mean non-termination of the evaluation directly. For example, even though type recursion of `power` function loops forever, if we give $\texttt{n} > 0$, it terminates and returns a result.

8.7.2 Instances as the complement of the non-instances

Instead of $\ll:$, we define another full instance relationship $<:$, as the complement of the *invalid full instances* which have possibility of failure. We call $<:$ the *relaxed full instance relationship.*

For the definition of $<:$, the *invalid full instance relationship* $\tau \not<: \mathcal{K}$ must be defined first:

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

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

$$\textsf{(k-fail-derive)}\ \frac{\tau \not\le \sigma, \quad \text{or} \quad \exists 1 \le i \le n, \quad S(\tau_i) \not<: S(\mathcal{K}_i)\ \text{where}\ \tau \le_S \sigma}{\tau \not<: \sigma\ \{\tau_i <: \mathcal{K}_i\}_{i=1}^n}$$

$$\textsf{(k-fail-free)}\ \tau \not<: \kappa$$

The rules are the negative versions of $\ll:$ rules and additional one for free constraint variables. The *relaxed full instance relationship*, $\tau <: \mathcal{K}$ is defined as the complement of $\tau \not<: \mathcal{K}$:

Definition 17 (Relaxed full instance) *Let τ be a type, $\mathcal{K}$ is a generic constraint. Then we say τ is a relaxed full instance (or just an instance) of $\mathcal{K}$ and write $\tau <: \mathcal{K}$, if and only if $\tau \not<: \sigma\ \mathcal{K}$ is not provable.*

The algorithm *kinst* is modified to provide an algorithm to determine $\tau <: \mathcal{K}$ or not, The new algorithm $kinst^+$ takes a new memo function argument M which remembers the μ-abstraction tests already it has tried. If it is asked the same question stored in the cache M, it immediately returns *true*:

$$\begin{array}{rcl}
kinst^+(M, \psi, \tau <: [\sigma_i\ D_i]_{i=1}^n) & = & \exists p, \text{where}\ \tau \not\le \sigma_1\ \&\ \ldots \tau \not\le \sigma_{p-1}\ \&\ \tau \le \sigma_p \\
 & & \&\ kinst^+(M, \psi, \tau <: \sigma_p\ D_p) \\
kinst^+(M, \psi, \tau <: \sigma\ \{\tau_i <: \mathcal{K}_i\}_{i=1}^n) & = & \tau \le_S \sigma\ \&\ \forall_{i=1}^n kinst^+(M, \psi, S(\tau_i) <: S(\mathcal{K}_i)) \\
kinst^+(M, \psi, \tau <: \mu\kappa.\mathcal{K}) & = & \text{if}\ (\tau <: \mu\kappa.\mathcal{K}) \in M\ \text{then}\ true \\
 & & \text{else let}\ M' = M \cup \{\tau <: \mu\kappa.\mathcal{K}\} \\
 & & \quad \text{let}\ \psi' = \psi \setminus \{\kappa \Rightarrow \psi(\kappa)\} \\
 & & \quad kinst^+(M', \psi'\{\kappa \Rightarrow \mu\kappa.\psi'(\mathcal{K})\}, \tau <: \mathcal{K}) \\
kinst^+(M, \psi, \tau <: \kappa) & = & \text{if}\ \kappa \notin Dom(\psi)\ \text{then}\ false \\
 & & \text{else}\ kinst^+(M, \psi, \tau <: \psi(\kappa))
\end{array}$$

Theorem 29 (Correctness of $kinst^+$ against $<:$) *Let τ be a type and $\mathcal{K}$ be a constraint. Suppose $kinst^+(\emptyset, \emptyset, \tau <: \mathcal{K})$ terminates. Then $\tau \not<: \mathcal{K}$ if and only if $kinst^+(\emptyset, \emptyset, \tau <: \mathcal{K})$ returns false.*

Proof: (Sketch) The proof follows the same outline of Theorem 25. Suppose $\tau \not<: \mathcal{K}$. then we can check easily $kinst^+(\emptyset, \emptyset, \tau \not<: \mathcal{K}) = false$ by induction over the derivation of $\tau \not<: \mathcal{K}$. Only thing we should note that while the evaluation of the algorithm, the cached predicate in the memo function M is never used. Otherwise we would have the exactly same judgments in a derivation path:

$$\cfrac{\cfrac{\cdots}{\tau \not<:\sigma\ \mu\kappa.\mathcal{K}}}{\cfrac{\cdots}{\cfrac{\tau \not<:\sigma\ \mu\kappa.\mathcal{K}}{\cdots}}}$$

which cannot happen, because the derivation would become infinite.

Contrary, suppose $kinst^+(\emptyset, \emptyset, \tau <: \mathcal{K}) = false$. We can build the proof of $\tau \not<: \mathcal{K}$ by induction over the evaluation of the algorithm $kinst^+$. □

Same as the strict relationship, we need to restrict the constraints to something admissible in order to make the predicate $\not<:$ decidable and so does $<:$, in order to $kinst^+$ can terminate for any restricted constraint.

8.7.3 Relaxed admissible constraints for $<:$

Like the strictly admissible constraints which were defined for the termination of $kinst(\tau \ll: \mathcal{K})$, we now define the set of *the relaxed admissible constraints* for $kinst^+(\tau <: \mathcal{K})$. The predicate adm^+ which defines the set of relaxed admissible constraints differs from adm in only one point: the type sizes also must be decreasing in the recursive part of the derived constraints in adm^+, but the strict decrease is not required:

$$\begin{array}{rcl}
adm^+(\psi, [\forall V_i.\tau_i\ \{\tau_{ij} <: \mathcal{K}_{ij}\}]) & = & \forall i, j,\ \ adm^+(\psi, \psi(\mathcal{K}_{ij})) = true \\
& & \qquad \text{if } Dom(\psi) \cap KFV(\mathcal{K}_{ij}) \neq \emptyset \\
& & \qquad \text{then } |\tau_i <: [\forall V_i.\tau_i\ \{\tau_{ij} <: \mathcal{K}_{ij}\}]| \geq |\tau_{ij} <: \psi(\mathcal{K}_{ij})| \\
adm^+(\psi, \mu\kappa.\mathcal{K}) & = & \text{let } \psi' = \psi \setminus \{\kappa \Rightarrow \psi(\kappa)\} \\
& & adm^+(\psi'\{\kappa \Rightarrow \mu\kappa.\psi'(\mathcal{K})\}, \mathcal{K}) \\
adm^+(\psi, \kappa) & = & \kappa \in Dom(\psi)
\end{array}$$

A constraint $\mathcal{K}$ is *relaxed admissible* if and only if $adm^+(\emptyset, \mathcal{K})$ holds. Clearly, from this definition, all the strictly admissible constraints are also relaxed admissible.

To prove this restriction assures the termination of $kinst^+$, first we need some lemmas, just like we have done in Section 8.6.2:

Lemma 26 (Substitution preserves relaxed admissibility) *Any type substitution S preserves the relaxed admissibility of constraints: if $adm^+(\psi, \mathcal{K}) = true$, then $adm^+(\psi, S(\mathcal{K})) = true$.*

Proof: Similar to Lemma 24. □

Lemma 27 (μ-expansion preserves relaxed admissibility) *Let $\mathcal{K}, \mathcal{K}'$ be constraints, ψ and φ be μ-expansion constraint substitutions and κ be a constraint variable, such that*

$$\begin{array}{l} \kappa \notin Dom(\psi) \\ \varphi = \{\kappa \Rightarrow \mathcal{K}\} \\ (\psi + \varphi) \circ (\psi + \varphi) = \psi + \varphi \end{array}$$

Then, if $adm^+(\psi + \varphi, \mathcal{K}) = true$ and $adm^+(\psi + \varphi, \mathcal{K}') = true$, $adm^+(\psi + \varphi, \varphi(\mathcal{K}')) = true$

Proof: Similar to Lemma 25. □

Lemma 28 (Dynamic substitution preserves relaxed admissibility) *Type substitutions whose domains are restricted within the dynamic type variables preserve the admissibility of type schemes: for a substitution S such that $Dom(S) \subset \mathbb{D}$, if $adm^+(\emptyset, \eta) = true$, then $adm^+(\emptyset, S(\eta)) = true$.*

Proof: Same as Lemma 24. □

Theorem 30 (Relaxed admissibility assures the termination of $kinst^+(\emptyset, \emptyset, \tau <: \mathcal{K})$) *Let $\mathcal{K}$ be a relaxed admissible extended constraint. Then for any type expression τ, $kinst^+(\emptyset, \emptyset, \tau <: \mathcal{K})$ always terminates.*

Proof: The termination can be proven almost similarly to one for *kinst*, Theorem 27. The difference is the cases when the sizes of $|\tau <: \mathcal{K}|$ do not change and never decrease. Even in such cases, since we have only finite type constructors, the predicates $|\tau <: \mathcal{K}|$ which share the same size are finite. Therefore sooner or later $kinst^+(M, \psi, \tau <: \mathcal{K})$ will find $\tau <: \mathcal{K}$ is stored in the cache M and terminate. □

Using the relaxed admissible constraint restriction, all the let defined recursive derived generic values and many useful recursive values defined by generic expressions such as `power` and `add` in previous examples are now admissible and permitted in type system.

8.8 Polymorphic instances

The full instances are not only the permitted generic instances in constraint type system, otherwise we could not define any derived and recursive generic value. *Polymorphic instances* are also allowed, intuitively whose type annotations are abstracted somewhere outside. Validness of polymorphic instances are postponed until the derived generic values which use these polymorphic instances are instantiated to full instances. Therefore type information of polymorphic instances are stored in the derived generic values' constraints as form of derived constraints. Before we define the constraint typing rules, we have to give the formal definition of the full and polymorphic instances.

When dynamic type variables of type annotation of a generic instance is generalized, the instance is a polymorphic instance. Such polymorphic instances are found in definitions of derived generic values. For example, the following instance of `print` generic value:

```
let print_twin x =
  print (x,x)
in ...
```

The `print` instance has a type $\delta * \delta \to \texttt{unit}$ where the type variable δ is abstracted at the generalization of `print_twin`, whose constraint is $[\forall\delta.\delta \to \texttt{unit}\ \{\delta * \delta \to \texttt{unit} <: \mathcal{K}_{\texttt{print}}\}]$. The type information of the polymorphic instance `print` is stored as the derived constraint $\{\delta * \delta \to \texttt{unit} <: \mathcal{K}_{\texttt{print}}\}$.

A generic instance may be a polymorphic instance even when its type does not contain any generalized dynamic type variable. If the constraint of a generic value contains free dynamic type variable which is generalized at some generalization, the instances of the generic value is polymorphic instances. Suppose `plus` is a generic value whose constraint is

$$\begin{aligned}\mathcal{K}_{\texttt{plus}} = \ & [\ \texttt{int} \to \texttt{int} \to \texttt{int} \\ & \mid \texttt{int} \to \texttt{float} \to \texttt{float} \\ & \mid \texttt{float} \to \texttt{int} \to \texttt{float} \\ & \mid \texttt{float} \to \texttt{float} \to \texttt{float}\ ]\end{aligned}$$

Then in the following example,

```
let f = fun x →
  let g = fun y →
    plus x y
  in
  print_int (g 1)
in ...
```

Let the type of `x` in the definition of `f` is δ. Then the constraint of `g` is

$$\mathcal{K}_{\texttt{g}} = [\forall\delta'\delta''.\delta' \to \delta''\ \{\delta \to \delta' \to \delta'' <: \mathcal{K}_{\texttt{plus}}\}]$$

which contains δ freely in its derived part and this δ is generalized at the definition of `f`. The type soundness of full instances of `f` depends on one for the instance of `g`, whose type is $\texttt{int} \to \texttt{int}$. The soundness of `g` depends on how δ is instantiated for each full instance of `f`. Actually, because of $\mathcal{K}_{\texttt{g}}$'s constraint, the only possible instantiation is $(\delta \Rightarrow \texttt{int})$ so that `plus` can have a type $\texttt{int} \to \texttt{int} \to \texttt{int}$. To express this dependency of type soundness, the generic instance of `g` must be a polymorphic instance, even though its type has no dynamic type variable. The type information of `g` must be exported as a derived constraint of `f`:

$$\mathcal{K}_{\texttt{f}} = [\forall\delta.\delta \to \texttt{unit}\ \{\texttt{int} \to \texttt{int} <: \mathcal{K}_{\texttt{g}}\}]$$

The predicate *polyinst* distinguishes polymorphic instances from the others — full instances. *polyinst* takes a set of type variables V which are generalized, a constraint $\mathcal{K}$ and a type τ:

$$\mathit{polyinst}(V, \mathcal{K}, \tau) = \mathit{generic}(\mathcal{K})\ \&\ (DV(\tau) \cup DV(\mathcal{K})) \cap V \neq \emptyset$$

Then the polymorphic instances are defined as follows:

Definition 18 (Polymorphic instances) *Let* $\texttt{g}$ *be an instance,* V *is the generalized type variable set at the occurrence of* $\texttt{g}$*, e.g. the type variables generalized by* `let` *or* `generic` *polymorphisms whose definition expression contains the instance* $\texttt{g}$*. If* $\texttt{g}$ *has a constraint* $\mathcal{K}$ *and annotated* $\texttt{g} : \tau$*, and* $polyinst(V, \mathcal{K}, \tau) = true$*,* $\texttt{g}$ *is called a* polymorphic instance*.*

8.9 Generic constraint type system

Using the (relaxed) full instance relationship, we build the *constraint type system*. As we have explained in the introduction of this chapter, constraint type system which checks the validity of the generic instances is separated from the *normal typing system* which has been introduced in Section 2.5 to avoid confusion between these two typing layers. Constraint typing system is designed to receive type-annotated terms which have already been checked by the normal type system, and perform further checks whether all the generic full instances in the terms are valid full instances.

8.9.1 Constraint annotated terms

We first add constraints to the type-annotated terms (whose original definition is in Section 2.3 and extended for the generic bindings at Section 7.3). The new expressions are called *constraint annotated terms*.

$$
\begin{array}{rcl}
e_t & ::= & \texttt{x}_{\mathcal{K}} : \tau \\
 & | & c_\sigma : \tau \mid \texttt{g}_\sigma : \tau \mid \texttt{fun x} \rightarrow e_t : \tau \mid e_t\ e_t : \tau \mid e_t, e_t : \tau \\
 & | & \texttt{let x}_{V,\mathcal{K}} \texttt{=} e_t \texttt{ in } e_t : \tau \\
 & | & \texttt{let rec x}_{V,\mathcal{K}} \texttt{=} e_t \texttt{ in } e_t : \tau \\
 & | & \texttt{generic x}_{\mathcal{K}} \texttt{=case } tc_t \mid \ldots \mid tc_t \texttt{ in } e_t : \tau \\
 & | & \texttt{generic rec x}_{\mathcal{K}} \texttt{=case } tc_t \mid \ldots \mid tc_t \texttt{ in } e_t : \tau \\
tc_t & \equiv & \sigma \Rightarrow e_t
\end{array}
$$

Variables now take constraints $\mathcal{K}$ instead of type schemes. Let bindings and generic bindings take their constraints $\mathcal{K}$ additional to the original annotations of identifiers.

The type annotation substitution is canonically extended:

$$
\begin{array}{l}
S(\texttt{x}_{\mathcal{K}} : \tau) = \texttt{x}_{S(\mathcal{K})} : S(\tau) \\
S(\texttt{let (rec) x}_{V,\mathcal{K}} \texttt{=} e_{t1} \texttt{ in } e_{t2}) = \texttt{let (rec) x}_{V,S(\mathcal{K})} \texttt{=} S(e_{t1}) \texttt{ in } S(e_{t2}) \\
S(\texttt{generic (rec) x}_{\mathcal{K}} \texttt{=case } tc_{t1} \mid \ldots \mid tc_{tn} \texttt{ in } e_t) = \\
\qquad\qquad \texttt{generic (rec) x}_{S(\mathcal{K})} \texttt{=case } S(tc_{t1}) \mid \ldots \mid S(tc_{tn}) \texttt{ in } S(e_t) \\
S(\sigma \Rightarrow e_t) = (\sigma \Rightarrow S(e_t))
\end{array}
$$

The definitions for the other constructions are as they were, because all their type annotations are same as the original syntax therefore have no constraints. Note that generalized variable set V in a let binding bounds type variable occurrences in the constraint annotation $\mathcal{K}$ also. If renaming of the bound variables is required for substitution, we also have to rename variables in $\mathcal{K}$.

Extending the coercion from constraints to types *coerce* defined at Section 8.4.5, we obtain the following mapping from constraint annotated terms to the original type-annotated GML expressions:

$$
\begin{array}{l}
\mathit{coerce}(\mathtt{x}_{\mathcal{K}} : \tau) = \mathtt{x}_{\mathit{coerce}(\mathcal{K})} : \tau \\
\mathit{coerce}(c_\sigma : \tau) = c_\sigma : \tau \\
\mathit{coerce}(\mathtt{g}_\sigma : \tau) = \mathtt{g}_\sigma : \tau \\
\mathit{coerce}(\mathtt{fun\ x} \rightarrow e_t : \tau) = (\mathtt{fun\ x} \rightarrow \mathit{coerce}(e_t)) : \tau \\
\mathit{coerce}(e_{t1}\ e_{t2} : \tau) = \mathit{coerce}(e_{t1})\ \mathit{coerce}(e_{t2}) : \tau \\
\mathit{coerce}(e_{t1}, e_{t2} : \tau) = \mathit{coerce}(e_{t1}), \mathit{coerce}(e_{t2}) : \tau \\
\mathit{coerce}(\mathtt{let\ (rec)\ x}_{V,\mathcal{K}} \mathtt{=} e_{t1}\ \mathtt{in}\ e_{t2} : \tau) = \mathtt{let\ (rec)\ x}_{V} \mathtt{=} \mathit{coerce}(e_{t1})\ \mathtt{in}\ \mathit{coerce}(e_{t2}) : \tau \\
\mathit{coerce}(\mathtt{generic\ (rec)\ x}_{\mathcal{K}} \mathtt{=case}\ tc_{t1} \mid \ldots \mid tc_{tn}\ \mathtt{in}\ e_t : \tau) = \\
\qquad \mathtt{generic\ (rec)\ x}_{\mathit{coerce}(\mathcal{K})} \mathtt{=case}\ \mathit{coerce}(tc_{t1}) \mid \ldots \mid \mathit{coerce}(tc_{tn})\ \mathtt{in}\ \mathit{coerce}(e_t) : \tau \\
\mathit{coerce}(\sigma \Rightarrow e_t) = (\sigma \Rightarrow \mathit{coerce}(e_t))
\end{array}
$$

8.9.2 Constraint typing rules

A typing judgment in the constraint typing system takes a form:

$$\Gamma, V \Vdash e_t$$

where e_t is a constraint annotated term defined just before and Γ is a typing environment where variables are bound to constraints. Because type instantiations of generic values are differently handled if instances are polymorphic instances or not, the judgment also carries V, a *generalized type variable set*, which indicates which type variables must be generalized in following typing derivation. In effect, the free occurrences of dynamic type variables in e_t are restricted within those of V.

Instance rules

We have two instantiation rules: one for the full and another for the polymorphic instances. (kt-full-inst) is for full instantiations:

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

To use this rule, the instance of `x` must be fully instantiated. The dynamic type variables in τ are restricted to those of V.

If a variable occurrence is a polymorphic instance, (kt-full-inst) is not applicable, instead, (kt-poly-inst) must be used even if $\tau <: \mathcal{K}$ holds:

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

If $\mathtt{x}$ is a polymorphic instance, *any* type τ can be the type of $\mathtt{x}$, only if its dynamic type variables are members of the generalized variables V. We need neither check whether τ is an instance of the type scheme of $\mathtt{x}$, $\mathit{p\_antiunif}(\mathcal{K})$, because we suppose that this has been proven already in the normal typing layer.

Constraint construction and rules for polymorphic bindings

Corresponding with generalizations of type expressions in the normal typing system, we create constraints in the let and generic bindings in the constraint typing. The constraint construction function $\mathit{Derive}(V, V', e_t)$ in Figure 8.1 takes two sets of type variables V and V' and a type-annotated term e_t, and returns a derived constraint. It based on function $\mathit{plist}(V, V', e_t)$ which lists up the polymorphic instance occurrences $\mathtt{x}_{\mathcal{K}} : \tau$ in e_t whose type annotations relate with V but not with V'. (The symbol @ is used for the list concatenation in the figure). This exclusion is not essentially required for the type soundness itself, but necessary not to store one polymorphic instance information into more than one constraints. Type annotation information of a polymorphic instance is stored into a derived constraint created at the most interior generalization which abstract dynamic type variables of the polymorphic instance, so that we can minimize the complexity of constraints (look at the next example). *Derive* function extracts type annotations from result of $\mathit{plist}\ [\ \mathtt{x}^1_{\mathcal{K}_1} : \tau_1;\ \ldots;\ \mathtt{x}^n_{\mathcal{K}_n} : \tau_n\ ]$ and forms a derived constraint $\{\tau_1 <: \mathcal{K}_1\ \& \ldots \&\ \tau_n <: \mathcal{K}_n\}$. The order of the derived constraints $\tau_i <: \mathcal{K}_i$ must corresponds with the occurrence order of those polymorphic instances[1].

Let bindings

Using *Derive*, constraint typing rules for polymorphic bindings can be defined:

$$\textbf{(kt-let)}\ \frac{\begin{array}{c}\Gamma, V + V' \Vdash e_{t1} : \tau_1 \quad \Gamma\{\mathtt{x} : \mathcal{K}\}, V \Vdash e_{t2} \\ D = \mathit{Derive}(V', \emptyset, e_{t1}) \quad \mathcal{K} = [\forall V'.\tau_1\ D] \quad \mathit{adm}^+(\emptyset, \mathcal{K}) = \mathit{true}\end{array}}{\Gamma, V \Vdash \mathtt{let}\ \mathtt{x}_{V',\mathcal{K}} \mathtt{=} e_{t1}\ \mathtt{in}\ e_{t2}}$$

The constraint typing rule for let bindings checks soundness of e_{t1} under $\Gamma, V + V'$, adding the generalized variable set V' of the conclusion annotation. For e_{t2}, the environment Γ is updated with a binding $\mathtt{x} : \mathcal{K}$ where $\mathcal{K} = [\forall V'.\tau_1\ D]$. The derived constraint D is the result of the constraint construction $\mathit{Derive}(V', \emptyset, e_{t1})$. To assure the termination of typing, the constraint $\mathcal{K}$ must be relaxed admissible. This admissible restriction is required also in the following rules which construct constraints.

[1]Minimization and fixed order of derived constraints are required for the correctness of generic flow compilation. Look Chapter 9 for details.

$$
\begin{array}{ll}
\mathit{plist}(V, V', c_\sigma : \tau) = [\,] & \\
\mathit{plist}(V, V', \mathtt{g}_\sigma : \tau) = [\,] & \\
\mathit{plist}(V, V', \mathtt{x}_{\mathcal{K}} : \tau) = & \text{if } \mathit{polyinst}(V, \mathcal{K}, \tau) = \mathit{true} \ \&\ \mathit{polyinst}(V', \mathcal{K}, \tau) = \mathit{false} \\
& \text{then } [\ \mathtt{x}_{\mathcal{K}} : \tau\ ] \\
& \text{else } [\,] \\
\mathit{plist}(V, V', e_{t1}\ e_{t2}) = \mathit{plist}(V, V', e_{t1}) \,@\, \mathit{plist}(V, V', e_{t2}) & \\
\mathit{plist}(V, V', (e_{t1},\ e_{t2})) = \mathit{plist}(V, V', e_{t1}) \,@\, \mathit{plist}(V, V', e_{t2}) & \\
\mathit{plist}(V, V', \mathtt{fun\ x} \to e_t) = \mathit{plist}(V, V', e_t) & \\
\mathit{plist}(V, V', \mathtt{let\ (rec)\ x}_{V'', \mathcal{K}} \mathtt{=} e_{t1}\ \mathtt{in}\ e_{t2}) = \mathit{plist}(V, V' \cup V'', e_{t1}) \,@\, \mathit{plist}(V, V', e_t) & \\
\mathit{plist}(V, V', \mathtt{generic\ (rec)\ x = case}\ tc_{t1} \mid \ldots \mid tc_{tn}\ \mathtt{in}\ e_{t2}) = & \\
\qquad \mathit{plist}(V, V', tc_{t1}) \,@\, \ldots \,@\, \mathit{plist}(V, V', tc_{tn}) \,@\, \mathit{plist}(V, V', e_t) & \\
\mathit{plist}(V, V', (\forall V''.\tau \Rightarrow e_t)) = \mathit{plist}(V, V' \cup V'', e_t) & \\
\mathit{Derive}(V, V', e_t) = & \text{let } [\ \mathtt{x}^1_{\mathcal{K}_1} : \tau_1;\ \ldots;\ \mathtt{x}^n_{\mathcal{K}_n} : \tau_n\ ] = \mathit{plist}(V, V', e_t) \text{ in} \\
& \{\tau_1 <: \mathcal{K}_1 \ \& \ldots \&\ \tau_n <: \mathcal{K}_n\}
\end{array}
$$

Figure 8.1: Constraint computation function *Derive*

Example In the following type-annotated expression

$$
\begin{array}{l}
\mathtt{val\ f\ :}\ [\forall\delta.\delta \to \mathtt{unit}\ D_{\mathtt{f}}] \\
\mathtt{let\ g}_{\{\delta'\}, \mathcal{K}_{\mathtt{g}}}\ \mathtt{=\ fun\ x} \to \\
\quad \mathtt{let\ h}_{\{\delta''\}, \mathcal{K}_{\mathtt{h}}}\ \mathtt{=\ fun\ y} \to \\
\qquad \mathtt{(f:}\delta'\mathtt{*}\delta'' \to \mathtt{unit)\ (x,y)} \\
\quad \mathtt{in} \\
\quad \mathtt{(h:}\delta' \to \mathtt{unit)\ x} \\
\mathtt{in} \\
\quad \ldots
\end{array}
$$

suppose that the normal typing gives g a type scheme $\forall\delta'.\delta' \to \mathtt{unit}$ and h $\forall\delta''.\delta'' \to \mathtt{unit}$. In constraint typing, let the constraint of the generic function f be $\mathcal{K}_{\mathtt{f}} = [\forall\delta.\delta \to \mathtt{unit}\ D_{\mathtt{f}}]$.

h's constraint $\mathcal{K}_{\mathtt{h}}$ must be as follows, because it contains a polymorphic occurrence of f inside the definition whose type annotation has type variable δ'' which is generalized by the definition of h:

$$
\begin{array}{l}
D_{\mathtt{h}} = \mathit{Derive}(\{\delta''\}, \emptyset, \mathtt{fun\ y} \to \mathtt{(f} : \delta' * \delta'' \to \mathtt{unit)\ (x,y)}) = \{\delta' * \delta'' \to \mathtt{unit} <: \mathcal{K}_{\mathtt{f}}\} \\
\mathcal{K}_{\mathtt{h}} = [\forall\delta''.\delta'' \to \mathtt{unit}\ D_{\mathtt{h}}]
\end{array}
$$

On the other hand, though the type variable δ' of the polymorphic instance of f is generalized by the definition of g, f is not directly taken into $\mathcal{K}_{\mathtt{g}}$, the constraint of g, because it is already listed in $\mathcal{K}_{\mathtt{h}}$ whose definition is interior to $\mathtt{g}$ for the instance $\mathtt{f}$:

$$\begin{aligned}
D_{\mathtt{g}} = & \ \mathit{Derive}(\{\delta'\}, \emptyset, \mathtt{fun\ x} \rightarrow \mathtt{let\ h = \ldots\ in\ (h}_{\mathcal{K}_{\mathtt{h}}} : \delta' \rightarrow \mathtt{unit)\ x}) = \\
& \quad \mathit{Derive}(\{\delta'\}, \{\delta''\}, \mathtt{fun\ y} \rightarrow (\mathtt{f}_{\mathcal{K}_{\mathtt{f}}} : \delta' * \delta'' \rightarrow \mathtt{unit})\ \mathtt{(x, y)})\ @ \\
& \qquad \mathit{Derive}(\{\delta'\}, \emptyset, (\mathtt{h}_{\mathcal{K}_{\mathtt{h}}} : \delta' \rightarrow \mathtt{unit})\ \mathtt{x}) = \\
& \quad \{\delta' \rightarrow \mathtt{unit} <: \mathcal{K}_{\mathtt{h}}\} \\
\mathcal{K}_{\mathtt{g}} = & \ [\forall \delta'.\delta' \rightarrow \mathtt{unit}\ D_{\mathtt{g}}]
\end{aligned}$$

The constraint of $\mathtt{g}$, $\mathcal{K}_{\mathtt{g}}$ does not contain the derived constraint of $\mathtt{f}$ directly, but its dependency on $\mathtt{f}$ is expressed $\mathcal{K}_{\mathtt{h}}$ in $D_{\mathtt{g}}$, since $\mathcal{K}_{\mathtt{h}}$ contains a derived constraint from $\mathtt{f}$.

□

Recursive lets

The recursive counterpart of (kt-let), (kt-letrec) stores its identifier $\mathtt{x}$ to environment Γ with a recursive constraint $\mu\kappa.\mathcal{K}$ in typings of both sub-expressions:

$$\textsf{(kt-letrec)}\ \frac{\begin{array}{c}\Gamma\{\mathtt{x} : \mu\kappa.\mathcal{K}\}, V + V' \Vdash e_{t1} : \tau_1 \quad \Gamma\{\mathtt{x} : \mu\kappa.\mathcal{K}\}, V \Vdash e_{t2} \\ D' = \mathit{Derive}(V', \emptyset, e_{t1}) \quad \mathcal{K}' = [\forall V'.\tau_1\ D'] \quad \mu\kappa.\mathcal{K} \xrightarrow{\mu} \mathcal{K}' \\ \mathit{adm}^{+}(\emptyset, \mu\kappa.\mathcal{K}) = \mathit{true}\end{array}}{\Gamma, V \Vdash \mathtt{let\ rec\ x}_{V', \mu\kappa.\mathcal{K}} = e_{t1}\ \mathtt{in}\ e_{t2}}$$

The constraint $\mathcal{K}'$ which contains derived constraint D' obtained by $\mathit{Derive}(V', \emptyset, e_{t1})$ must be equal to the μ-expansion of $\mu\kappa.\mathcal{K}$: it must be a fixed point. In the typing of the let definition e_{t1}, $\mathtt{x}$ has a polymorphic constraint $\mu\kappa.\mathcal{K}$, unlike normal ML typing where a recursive let identifier must have a monomorphic type inside its definition. It corresponds with the fact that the variable occurrences of $\mathtt{x}$ in e_{t1} have type scheme annotations with $\forall V'.\tau_1$ in (t-letrec) typing rule (Section 2.5.3).

When e_{t1} contains no polymorphic instance occurrence, the derived constraint D' from e_{t1} is an empty set and $\mathcal{K}'$ is simply $[\forall V'.\tau_1]$. In such a non-generic recursive definition, the constraint of the defined value is actually non-recursive, since $\mu\kappa.\mathcal{K} = \mu\kappa.[\forall V'.\tau_1] = \mu\kappa.\mathcal{K}'$ and $\mu\kappa.\mathcal{K}' \equiv \mathcal{K}'$ from Definition 16.

Generic bindings

$$\textsf{(kt-generic)}\ \frac{\begin{array}{c}\Gamma, V + V_i \Vdash e_{ti} \quad D_i = \mathit{Derive}(V_i, \emptyset, e_{ti}) \\ BV(\sigma_i) = V_i \quad FV(\sigma_i) = \emptyset \\ \mathcal{K} = [\sigma_1\ D_1 \mid \ldots \mid \sigma_n\ D_n] \quad \mathit{adm}^{+}(\emptyset, \mathcal{K}) = \mathit{true} \\ \Gamma\{\mathtt{x} : \mathcal{K}\}, V \Vdash e_t\end{array}}{\Gamma, V \Vdash \mathtt{generic\ x}_{\mathcal{K}} = \mathtt{case}\ \sigma_1 \Rightarrow e_{t1} \mid \ldots \mid \sigma_n \Rightarrow e_{tn}\ \mathtt{in}\ e_t}$$

For generic bindings, the type system checks type method expressions e_{ti} and creates their derived constraints D_i like in (kt-let). In the check of the expression's body e_t, $\mathtt{x}$ is stored as $\mathtt{x} : [\sigma_1\ D_1 \mid \ldots \mid \sigma_n\ D_n]$, with a list of combinations of the type pattern schemes and corresponding derived constraints.

Recursive generic bindings

Like (kt-letrec), a recursively defined identifier by `generic` expression has a recursive constraint $\mu\kappa.\mathcal{K}$ and this must be μ-expanded to $\mathcal{K}'$, the collection of the derived constraints $\sigma_i\ D_i$ of the typings of the case expressions.

$$\textbf{(kt-genericrec)}\ \frac{\begin{array}{c}\Gamma\{\mathtt{x} : \mu\kappa.\mathcal{K}\}, V + V_i \Vdash e_{ti} \quad D_i = \mathit{Derive}(V_i, \emptyset, e_{ti}) \\ BV(\sigma_i) = V_i \quad FV(\sigma_i) = \emptyset \\ \mathcal{K}' = [\sigma_1\ D_1 \mid \ldots \mid \sigma_n\ D_n] \quad \mu\kappa.\mathcal{K} \xrightarrow{\mu} \mathcal{K}' \quad \mathit{adm}^+(\emptyset, \mu\kappa.\mathcal{K}) = \mathit{true} \\ \Gamma\{\mathtt{x} : \mu\kappa.\mathcal{K}\}, V \Vdash e_t\end{array}}{\Gamma, V \Vdash \mathtt{generic\ rec\ x}_{\mu\kappa.\mathcal{K}} = \mathtt{case}\ \sigma_1 \Rightarrow e_{t1} \mid \ \ldots \ \mid \sigma_n \Rightarrow e_{tn}\ \mathtt{in}\ e_t}$$

The others

The other rules, for constants, generic primitives, functions, applications and tuples are quite straightforward, except the restriction that they can only introduce dynamic type variables which are stored in their generalized type variable sets.

As a summary, we put all the constraint typing rules in Figure 8.2.

Example Let's think typing of a generic binding of `print` given by the following definition:

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

Let the constraint of `print` be $\mathcal{K}$, which is still unsure for us. The typing of the first two type cases are trivial. Since the type pattern of the third case has an abstract type variable δ, it method is checked with the generalized variable set $\{\delta\}$:

$$\frac{\dfrac{\Gamma\{\mathtt{print} : \mathcal{K}\}, \{\delta\} \Vdash \mathtt{print} : \delta \rightarrow \mathtt{unit} \quad \Gamma\{\mathtt{print} : \mathcal{K}\}, \{\delta\} \Vdash \mathtt{print} : \delta\ \mathtt{list} \rightarrow \mathtt{unit}}{\ldots}}{\Gamma\{\mathtt{print} : \mathcal{K}\}, \{\delta\} \Vdash \begin{array}{l}(\mathtt{function\ [\,]} \rightarrow ()\\ \qquad \mid \mathtt{x::xs} \rightarrow \mathtt{print\ x;\ print\ xs}) : \delta\ \mathtt{list} \rightarrow \mathtt{unit}\end{array}}$$

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

$$\begin{array}{l}\mathit{Derive}(\{\delta\}, \emptyset, \ \mathtt{function\ [\,]} \rightarrow \ldots; \ \ \mathtt{print}_{\mathcal{K}}\ \mathtt{xs}\ ) = \\ \qquad\qquad \{\delta\ \rightarrow \mathtt{unit} <: \mathcal{K}\ \&\ \delta\ \mathtt{list} \rightarrow \mathtt{unit} <: \mathcal{K}\}\end{array}$$

$$\textbf{(kt-const)}\ \frac{DV(\tau) \subset V}{\Gamma, V \Vdash c_\sigma : \tau} \qquad \textbf{(kt-gen)}\ \frac{DV(\tau) \subset V}{\Gamma, V \Vdash \mathtt{g}_\sigma : \tau}$$

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

$$\textbf{(kt-tpl)}\ \frac{\Gamma, V \Vdash e_{t1} \quad \Gamma, V \Vdash e_{t2}}{\Gamma, V \Vdash (e_{t1},\ e_{t2})}$$

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

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

$$\textbf{(kt-let)}\ \frac{\begin{array}{c}\Gamma, V + V' \Vdash e_{t1} : \tau_1 \quad \Gamma\{\mathtt{x} : \mathcal{K}\}, V \Vdash e_{t2} \\ D = \mathit{Derive}(V', \emptyset, e_{t1}) \quad \mathcal{K} = [\forall V'.\tau_1\ D] \quad \mathit{adm}^+(\emptyset, \mathcal{K}) = \mathit{true}\end{array}}{\Gamma, V \Vdash \mathtt{let\ x}_{V',\mathcal{K}} \mathtt{=} e_{t1}\ \mathtt{in}\ e_{t2}}$$

$$\textbf{(kt-letrec)}\ \frac{\begin{array}{c}\Gamma\{\mathtt{x} : \mu\kappa.\mathcal{K}\}, V + V' \Vdash e_{t1} : \tau_1 \quad \Gamma\{\mathtt{x} : \mu\kappa.\mathcal{K}\}, V \Vdash e_{t2} \\ D' = \mathit{Derive}(V', \emptyset, e_{t1}) \quad \mathcal{K}' = [\forall V'.\tau_1\ D'] \quad \mu\kappa.\mathcal{K} \stackrel{\mu}{\rightarrow} \mathcal{K}' \\ \mathit{adm}^+(\emptyset, \mu\kappa.\mathcal{K}) = \mathit{true}\end{array}}{\Gamma, V \Vdash \mathtt{let\ rec\ x}_{V',\mu\kappa.\mathcal{K}} \mathtt{=} e_{t1}\ \mathtt{in}\ e_{t2}}$$

$$\textbf{(kt-generic)}\ \frac{\begin{array}{c}\Gamma, V + V_i \Vdash e_{ti} \quad D_i = \mathit{Derive}(V_i, \emptyset, e_{ti}) \\ BV(\sigma_i) = V_i \quad FV(\sigma_i) = \emptyset \\ \mathcal{K} = [\sigma_1\ D_1 \mid \ldots \mid \sigma_n\ D_n] \quad \mathit{adm}^+(\emptyset, \mathcal{K}) = \mathit{true} \\ \Gamma\{\mathtt{x} : \mathcal{K}\}, V \Vdash e_t\end{array}}{\Gamma, V \Vdash \mathtt{generic\ x}_{\mathcal{K}} \mathtt{=case}\ \sigma_1 \Rightarrow e_{t1} \mid \ \ldots\ \mid \sigma_n \Rightarrow e_{tn}\ \mathtt{in}\ e_t}$$

$$\textbf{(kt-genericrec)}\ \frac{\begin{array}{c}\Gamma\{\mathtt{x} : \mu\kappa.\mathcal{K}\}, V + V_i \Vdash e_{ti} \quad D_i = \mathit{Derive}(V_i, \emptyset, e_{ti}) \\ BV(\sigma_i) = V_i \quad FV(\sigma_i) = \emptyset \\ \mathcal{K}' = [\sigma_1\ D_1 \mid \ldots \mid \sigma_n\ D_n] \quad \mu\kappa.\mathcal{K} \stackrel{\mu}{\rightarrow} \mathcal{K}' \quad \mathit{adm}^+(\emptyset, \mu\kappa.\mathcal{K}) = \mathit{true} \\ \Gamma\{\mathtt{x} : \mu\kappa.\mathcal{K}\}, V \Vdash e_t\end{array}}{\Gamma, V \Vdash \mathtt{generic\ rec\ x}_{\mu\kappa.\mathcal{K}} \mathtt{=case}\ \sigma_1 \Rightarrow e_{t1} \mid \ \ldots\ \mid \sigma_n \Rightarrow e_{tn}\ \mathtt{in}\ e_t}$$

Figure 8.2: Constraint typing rules

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

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

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

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

□

By the constraint typing rules, free dynamic variable occurrences in expressions are restricted:

Lemma 29 **(Dynamic variable closedness)** *Let Γ be a typing environment, V be a set of type variables and e_t be an expression which has a typing $\Gamma, V \Vdash e_t$. Then $DV(e_t) \subset V$ and $DV(Derive(V, \emptyset, e_t)) \subset V$*

PROOF: Easily proven by induction over the constraint typing derivation. When rules introduce new dynamic type variables and create a judgment $\Gamma, V \Vdash e_t$, the dynamic variables are always restricted to those of the generalized type variable set V.

$DV(Derive(V, \emptyset, e_t)) \subset V$ is clear from the definition of *Derive*. □

Lemma 30 *The typing preserves well formedness of typing environments: if a conclusion judgment has a well formed typing environment Γ, then its all premise judgments also have well formed typing environments.*

PROOF: Clear from the definition of the well formedness and typing rules.

□

Like the non-generic type soundness of the normal typing, a key to the *generic type soundness* which assures no generic run time type error is a substitution lemma, too.

Lemma 31 **(Substitution lemma of** *Derive***)** *Let e_t be a constraint annotated expression and $D = Derive(V, V', e_t)$. Let S be a substitution such that V and V' are out of the reach of S. Then $Derive(V, V', S(e_t)) = S(D)$.*

PROOF: By induction over the structure of e_t.

Variables Let e_{t1} be a variable $\mathtt{x}_{\mathcal{K}} : \tau$. Then $D = \mathit{Derive}(V, V', \mathtt{x}_{\mathcal{K}} : \tau)$.

If $\mathit{polyinst}(V, \mathcal{K}, \tau) = \mathit{true}$ & $\mathit{polyinst}(V', \mathcal{K}, \tau) = \mathit{false}$ then $D = \{\tau <: \mathcal{K}\}$. Because V and V' are out of S, S does not affect the free type variable occurrences of V and V' in τ and $\mathcal{K}$. Therefore, $\mathit{polyinst}(V, S(\mathcal{K}), S(\tau)) = \mathit{true}$ & $\mathit{polyinst}(V', S(\mathcal{K}), S(\tau)) = \mathit{false}$ and $\mathit{Derive}(V, V', S(\mathtt{x}_{\mathcal{K}} : \tau)) = \{S(\tau) <: S(\mathcal{K})\} = S(D)$.

Otherwise, if $\mathit{polyinst}(V, \mathcal{K}, \tau) = \mathit{true}$ & $\mathit{polyinst}(V', \mathcal{K}, \tau) = \mathit{false}$ is not true, $\mathit{Derive}(V, V', \mathtt{x}_{\mathcal{K}} : \tau) = \{\}$. By the same reason above, we obtain $\mathit{Derive}(V, V', S(\mathtt{x}_{\mathcal{K}} : \tau)) = \{\}$.

Let bindings $e_t = \mathtt{let}\ \mathtt{y}_{V'',\mathcal{K}} \mathtt{=} e_{t1}\ \mathtt{in}\ e_{t2}$. Supposing that V'' is distinct from V and V' by applying α-conversion if necessary,

$$D = \mathit{Derive}(V, V', e_t) = \mathit{Derive}(V, V' + V'', e_{t1}) \mathbin{@} \mathit{Derive}(V, V', e_{t2})$$

Since we can also assume that V'' is out of the reach of S, using the induction hypothesis we have:

$$\begin{aligned}&\mathit{Derive}(V, V' + V'', S(e_{t1})) = S(\mathit{Derive}(V, V' + V'', e_{t1}))\\ &\mathit{Derive}(V, V', S(e_{t2})) = S(\mathit{Derive}(V, V', e_{t2}))\end{aligned}$$

Therefore $\mathit{Derive}(V, V', S(e_{t1})) = S(\mathit{Derive}(V, V' + V'', e_{t1})) \mathbin{@} \mathit{Derive}(V, V', S(e_{t2}))$.

The others The cases for rules with type generalization are proven same as the case of let bindings. For the other rules without constraint construction, the lemma is trivial.

□

Lemma 32 (Substitution lemma of constraint typing) *Let Γ be a typing environment, V_0 and V be distinct sets of type variables, e_t be a constraint annotated expression and S be a substitution where $Dom(S) = V$ and $\forall \nu \in V, DV(S(\nu)) = \emptyset$.*

Suppose e_t has a constraint typing $\Gamma, V_0 + V \Vdash e_t$ and

$$\{\tau_1 <: \mathcal{K}_1 \,\&\ldots\&\, \tau_n <: \mathcal{K}_n\} = \mathit{Derive}(V, V_0, e_t)$$

and for all i, $S(\tau_i) <: S(\mathcal{K}_i)$. Then,

$$S(\Gamma), V_0 \Vdash S(e_t)$$

PROOF: By induction over the construction of e_t:

Variables Let $e_t = \mathtt{x}_{\mathcal{K}} : \tau$. We have further two cases, depending on full or polymorphic instances:

Full instances If the typing is obtained by (kt-full-inst):

$$\frac{DV(\tau) \subset V + V_0 \quad \tau <: \mathcal{K} \quad \mathit{polyinst}(V + V_0, \mathcal{K}, \tau) = \mathit{false}}{\Gamma, V + V_0 \Vdash \mathtt{x}_{\mathcal{K}} : \tau}$$

Because $S(\tau)$ replaces all the variable occurrences $\nu \in V$ to types without dynamic variables, $DV(S(\tau)) = DV(\tau) \setminus V \subset V_0$.

If $\mathit{generic}(\mathcal{K}) = \mathit{false}$, from the well formedness of Γ, $\mathcal{K}$ must be written simply as $[\sigma]$ for some type scheme σ. From the definition of *polyinst*, $\mathit{polyinst}(V, [\sigma], \tau) = \mathit{false}$. Since $\tau <: [\sigma]$, $\tau \leq \sigma$. From Lemma 4, $S(\tau) \leq S(\sigma)$, therefore $S(\tau) <: S([\sigma])$. Using (kt-full-inst), we have a typing for $S(e_t)$:

$$\frac{DV(S(\tau)) \subset V_0 \quad S(\tau) <: S([\sigma]) \quad \mathit{polyinst}(V_0, S([\sigma]), S(\tau)) = \mathit{false}}{\Gamma, V_0 \Vdash \mathtt{x}_{S([\sigma])} : S(\tau)}$$

Otherwise, in the case $\mathit{generic}(\mathcal{K}) = \mathit{true}$, from

$$\begin{aligned} \mathit{polyinst}(V + V_0, \mathcal{K}, \tau) &= (\mathit{generic}(\mathcal{K}) \ \&\ (DV(\tau) \cup DV(\mathcal{K})) \cap (V + V_0) \neq \emptyset) \\ &= \mathit{false} \end{aligned}$$

$DV(\tau) \cap (V + V_0) = \emptyset$ and $DV(\mathcal{K}) \cap (V + V_0) = \emptyset$. Therefore $DV(\tau) \cap V = \emptyset$ and $DV(\mathcal{K}) \cap V = \emptyset$, and since $Dom(S) = V$, $S(\tau) = \tau$ and $S(\mathcal{K}) = \mathcal{K}$. From these, $S(\mathtt{x}_{\mathcal{K}} : \tau)$ is also a full instance under V_0:

$$\mathit{polyinst}(V_0, S(\mathcal{K}), S(\tau)) = \mathit{polyinst}(V_0, \mathcal{K}, \tau) = \mathit{false}$$

The typing we want is like:

$$\frac{DV(S(\tau)) \subset V_0 \quad S(\tau) = \tau <: \mathcal{K} = S(\mathcal{K}) \quad \mathit{polyinst}(V_0, S(\mathcal{K}), S(\tau)) = \mathit{false}}{\Gamma, V_0 \Vdash \mathtt{x}_{S(\mathcal{K})} : S(\tau)}$$

Polymorphic instances The typing should be as follows:

$$\frac{DV(\tau) \subset V + V_0 \quad \mathit{polyinst}(V + V_0, \mathcal{K}, \tau) = \mathit{true}}{\Gamma, V + V_0 \Vdash \mathtt{x}_{\mathcal{K}} : \tau}$$

Same as the case for the full instances, $DV(S(\tau)) \subset V_0$.

If $\mathit{polyinst}(V_0, S(\mathcal{K}), S(\tau)) = \mathit{true}$, we can easily construct the goal by (kt-poly-inst):

$$\frac{DV(S(\tau)) \subset V_0 \quad \mathit{polyinst}(V_0, S(\mathcal{K}), S(\tau)) = \mathit{true}}{\Gamma, V_0 \Vdash \mathtt{x}_{S(\mathcal{K})} : S(\tau)}$$

Otherwise, if $\mathit{polyinst}(V_0, S(\mathcal{K}), S(\tau)) = \mathit{false}$, Because $\mathit{polyinst}(V+V_0, \mathcal{K}, \tau) = \mathit{true}$,

$$(DV(\mathcal{K}) \cup DV(\tau)) \cap (V + V_0) \neq \emptyset$$

On the other hand, from $\mathit{polyinst}(V_0, S(\mathcal{K}), S(\tau)) = \mathit{false}$, we have $(DV(S(\mathcal{K})) \cup DV(S(\tau))) \cap V_0 = \emptyset$. From the condition of S, this is rewritten as

$$(DV(\mathcal{K}) \cup DV(\tau)) \subset V$$

From these, we have $(DV(\mathcal{K}) \cup DV(\tau)) \cap V \neq \emptyset$, therefore $polyinst(V, \mathcal{K}, \tau) = true$ and $Derive(V, V', \mathtt{x}_{\mathcal{K}} : \tau) = \{\tau <: \mathcal{K}\}$. From the condition of $Derive(V, V', e_t)$, $S(\tau) <: S(\mathcal{K})$.

$S(\mathtt{x})$ becomes a valid full instance under V_0:

$$\frac{DV(S(\tau)) \subset V_0 \quad S(\tau) <: S(\mathcal{K}) \quad polyinst(V_0, S(\mathcal{K}), S(\tau)) = false}{\Gamma, V_0 \Vdash \mathtt{x}_{S(\mathcal{K})} : S(\tau)}$$

Let bindings Let $e_t = \mathtt{let}\ \mathtt{x}_{V',\mathcal{K}} \mathtt{=} e_{t1}\ \mathtt{in}\ e_{t2}$. We suppose V' is distinct with both V_0 and V, and out of the reach of S, so that we can avoid the same discussion about doubly bound type variables in proofs of the previous substitution lemmas. Then,

$$\frac{\begin{array}{c}\Gamma, V_0 + V + V' \Vdash e_{t1} : \tau_1 \quad \Gamma\{\mathtt{x} : \mathcal{K}\}, V_0 + V \Vdash e_{t2} \\ D = Derive(V', \emptyset, e_{t1}) \quad \mathcal{K} = [\forall V'.\tau_1\ D] \quad adm^+(\emptyset, \mathcal{K}) = true\end{array}}{\Gamma, V_0 + V \Vdash \mathtt{let}\ \mathtt{x}_{V',\mathcal{K}} \mathtt{=} e_{t1}\ \mathtt{in}\ e_{t2}}$$

From the definition of $Derive$,

$$\begin{aligned} Derive(V, V_0, \mathtt{let}\ \mathtt{x}_{V',\mathcal{K}} \mathtt{=} e_{t1}\ \mathtt{in}\ e_{t2}) &= \{\tau_1 <: \mathcal{K}_1\ \&\ \ldots\ \&\ \tau_n <: \mathcal{K}_n\} \\ &= Derive(V, V_0 + V', e_{t1})\ @\ Derive(V, V_0', e_{t2}) \end{aligned}$$

Therefore there is an integer $1 \leq m \leq n$ and

$$\begin{aligned} &Derive(V, V_0 + V', e_{t1}) = \{\tau_1 <: \mathcal{K}_1\ \&\ \ldots\ \&\ \tau_m <: \mathcal{K}_m\} \\ &Derive(V, V_0', e_{t2}) = \{\tau_{m+1} <: \mathcal{K}_{m+1}\ \&\ \ldots\ \&\ \tau_n <: \mathcal{K}_n\} \end{aligned}$$

and $S(\tau_i) <: S(\mathcal{K}_i)$ for all $1 \leq i \leq n$. We can apply the induction hypothesis to the premises of e_{t1} and e_{t2}, and obtain the followings:

$$\begin{aligned} &S(\Gamma), V_0 + V' \Vdash S(e_{t1}) \\ &S(\Gamma\{\mathtt{x} : \mathcal{K}\}), V_0 \Vdash S(e_{t2}) \end{aligned}$$

Now let's consider constraint constructions of $S(e_{t1})$. From Lemma 31, since V' is out of the reach of S, $Derive(V', \emptyset, S(e_{t1})) = S(Derive(V', \emptyset, e_{t1})) = S(D)$. From Lemma 28, $adm^+(\emptyset, S(\mathcal{K})) = adm^+(\emptyset, [\forall V'.S(\tau_1)\ S(D)]) = true$.

From all of these, using (kt-let), we prove:

$$\frac{\begin{array}{c}S(\Gamma), V_0 + V' \Vdash S(e_{t1} : \tau_1) \quad S(\Gamma)\{\mathtt{x} : S(\mathcal{K})\}, V_0 \Vdash S(e_{t2}) \\ S(D) = Derive(V', \emptyset, S(e_{t1})) \quad S(\mathcal{K}) = [\forall V'.S(\tau_1)\ S(D)] \quad adm^+(\emptyset, S(\mathcal{K})) = true\end{array}}{S(\Gamma), V_0 \Vdash S(\mathtt{let}\ \mathtt{x}_{V',\mathcal{K}} \mathtt{=} e_{t1}\ \mathtt{in}\ e_{t2})}$$

Recursive let bindings In this time, $e_t = \mathtt{let\ rec}\ \mathtt{x}_{V',\mathcal{K}} \mathtt{=} e_{t1}\ \mathtt{in}\ e_{t2}$. Like what we have done in the part of normal lets, V' is distinct from V_0 and V and out of the reach of S.

The typing is:

$$\frac{\Gamma\{\mathtt{x} : \mu\kappa.\mathcal{K}\}, V_0 + V + V' \Vdash e_{t1} : \tau_1 \quad \Gamma\{\mathtt{x} : \mu\kappa.\mathcal{K}\}, V + V_0 \Vdash e_{t2} \qquad D = \mathit{Derive}(V', \emptyset, e_{t1}) \quad \mathcal{K}' = [\forall V'.\tau_1\ D] \quad \mu\kappa.\mathcal{K} \xrightarrow{\mu} \mathcal{K}' \quad \mathit{adm}^+(\emptyset, \mu\kappa.\mathcal{K}) = \mathit{true}}{\Gamma, V_0 + V \Vdash \mathtt{let\ rec\ x}_{V',\mu\kappa.\mathcal{K}} \mathtt{=} e_{t1}\ \mathtt{in}\ e_{t2}}$$

Using the induction hypothesis we have the following:

$$S(\Gamma)\{\mathtt{x} : S(\mu\kappa.\mathcal{K})\}, V_0 + V' \Vdash S(e_{t1} : \tau_1)$$
$$S(\Gamma)\{\mathtt{x} : S(\mu\kappa.\mathcal{K})\}, V_0 \Vdash S(e_{t2})$$

By Lemma 31, $\mathit{Derive}(V', \emptyset, S(e_{t1})) = S(\mathit{Derive}(V', \emptyset, e_{t1})) = S(D)$ and $S(\mathcal{K}') = [\forall V'.S(\tau_1)\ S(D)]$. Since type substitutions and constraint substitutions are independent of each other,

$$S(\mu\kappa.\mathcal{K}) = \mu\kappa.S(\mathcal{K}) \xrightarrow{\mu} \{\kappa \Rightarrow \mu\kappa.S(\mathcal{K})\}(S(\mathcal{K})) = S(\{\kappa \Rightarrow \mu\kappa.\mathcal{K}\}\mathcal{K}) = S(\mathcal{K}')$$

From Lemma 28, $\mathit{adm}^+(\emptyset, S(\mu\kappa.\mathcal{K})) = \mathit{true}$.

From these using (kt-letrec), we prove:

$$\frac{S(\Gamma\{\mathtt{x} : \mu\kappa.\mathcal{K}\}), V_0 + V' \Vdash S(e_{t1}) \quad S(\Gamma\{\mathtt{x} : \mu\kappa.\mathcal{K}\}), V_0 \Vdash S(e_{t2}) \qquad S(D) = \mathit{Derive}(V', \emptyset, S(e_{t1})) \quad S(\mathcal{K}') = [\forall V'.S(\tau_1)\ S(D)] \quad S(\mu\kappa.\mathcal{K}) \xrightarrow{\mu} S(\mathcal{K}') \qquad \mathit{adm}^+(\emptyset, S(\mu\kappa.\mathcal{K})) = \mathit{true}}{S(\Gamma), V_0 \Vdash S(\mathtt{let\ rec\ x}_{V',\mu\kappa.\mathcal{K}} \mathtt{=} e_{t1}\ \mathtt{in}\ e_{t2})}$$

Generic bindings Let $e_t = \mathtt{generic\ x}_{\mathcal{K}} \mathtt{=case}\ \sigma_1 \Rightarrow e_{t1} \mid \ldots \mid \sigma_n \Rightarrow e_{tn}\ \mathtt{in}\ e'_t : \tau$. The typing is by (kt-generic), as follows:

$$\frac{\Gamma, V + V_0 + V_i \Vdash e_{ti} : \tau_i \quad D_i = \mathit{Derive}(V_i, \emptyset, e_{ti}) \qquad BV(\sigma_i) = V_i \quad FV(\sigma_i) = \emptyset \qquad \mathcal{K} = [\sigma_1\ D_1 \mid \ldots \mid \sigma_n\ D_n] \quad \mathit{adm}^+(\emptyset, \mathcal{K}) = \mathit{true} \qquad \Gamma\{\mathtt{x} : \mathcal{K}\}, V + V_0 \Vdash e_t}{\Gamma, V + V_0 \Vdash \mathtt{generic\ x}_{\mathcal{K}} \mathtt{=case}\ \sigma_1 \Rightarrow e_{t1} \mid \ldots \mid \sigma_n \Rightarrow e_{tn}\ \mathtt{in}\ e'_t}$$

Here we suppose for each σ_i, its bound type variables $BV(\sigma_i)$ are distinct from V and V_0.

Like the proof for let bindings, since the variable occurrences of e'_t are also occurrences of the whole expression e_t, their type annotation conditions for this lemma in e_t can be used as those for e'_t directly. Therefore from the induction hypothesis:

$$S(\Gamma\{\mathtt{x} : \mathcal{K}\}), V_0 \Vdash S(e'_t : \tau)$$

For each type case expression, we can prove that

$$S(\Gamma), V_0 + V_i \Vdash S(e_{ti} : \tau_i)$$
$$S(D_i) = \mathit{Derive}(V_i, \emptyset, e_{ti})$$

almost in the same way we have done in let bindings. From the closedness of σ_i, $S(\sigma_i) = \sigma_i$. From Lemma 28, $adm^+(\emptyset, S(\mathcal{K})) = true$

From these, using (kt-generic), we can prove the substituted version of the generic expression typing, using the closeness of σ_i:

$$\frac{\begin{array}{c} S(\Gamma), V_0 + V_i \Vdash S(e_{ti}) : \tau_i \quad S(D_i) = \mathit{Derive}(V_i, \emptyset, S(e_{ti})) \\ BV(\sigma_i) = V_i \quad FV(\sigma_i) = \emptyset \\ S(\mathcal{K}) = [\sigma_1\ S(D_1) \mid \ldots \mid \sigma_n\ S(D_n)] \quad adm^+(\emptyset, S(\mathcal{K})) = true \\ S(\Gamma)\{\mathtt{x} : S(\mathcal{K})\}, V_0 \Vdash S(e_t) \end{array}}{\begin{array}{l} S(\Gamma), V_0 \Vdash \mathtt{generic}\ \mathtt{x}_{S(\mathcal{K})} \mathtt{=}\ \mathtt{case} \\ \qquad\qquad \sigma_1 \Rightarrow S(e_{t1}) \mid \ \ldots \ \mid \sigma_n \Rightarrow S(e_{tn})\ \mathtt{in}\ S(e'_t) \end{array}}$$

Recursive generic bindings The case for recursive generic bindings is proven like normal generic bindings, combining the same argument in recursive lets.

The others The other expression constructions are non- generic and their typing are independent of generalized type variable sets. Therefore these are merely special cases of the normal substitution lemma of the typing, Theorem 3. We just prove the case of functions to show they are straightforward.

Let $e_t = \mathtt{fun}\ \mathtt{x} \rightarrow e'_t$. Its typing must be by (kt-fun):

$$\frac{DV(\tau_1) \subset V + V_0 \quad \Gamma\{\mathtt{x} : [\tau_1]\}, V + V_0 \Vdash e'_t}{\Gamma, V + V_0 \Vdash \mathtt{fun}\ \mathtt{x} \rightarrow e'_t}$$

All the variable occurrences $\mathtt{x}^i_{\mathcal{K}_i} : \tau_i$ in e'_t are also occurrences of e_t, therefore we can use the induction hypothesis for the constraint typing of e'_t and obtain:

$$S(\Gamma)\{\mathtt{x} : S([\tau_1])\}, V_0 \Vdash S(e'_t)$$

Like we have done in the part of variables, $DV(S([\tau_1])) \subset V_0$, because S maps all the type variables in V to types without dynamic type variables.

We can use (kt-fun) and put it back to a function:

$$\frac{DV(S(\tau_1)) \subset V_0 \quad S(\Gamma)\{\mathtt{x} : S([\tau_1])\}, V_0 \Vdash S(e'_t)}{S(\Gamma), V_0 \Vdash \mathtt{fun}\ \mathtt{x} \rightarrow S(e'_t)}$$

□

8.9.3 Well typed programs

Now the set of *well-typed programs* are restricted also by the generic constraint typing system: they are not only typable in the normal typing, but also must be checked by the constraint typing. More formally:

Definition 19 (Normal typing of constraint annotated expressions) *Let e_t be a full constraint annotated expression defined in Section 8.9.1 and Γ be a constraint typing environment. If $coerce(\Gamma) \vdash coerce(e_t)$, we simply write $\Gamma \vdash e_t$ and says e_t is well-typed in the normal typing under Γ.*

Definition 20 (well-typed programs) *A full constraint annotated expression e_t is a well-typed program if and only if $\emptyset \vdash e_t$ and $\emptyset, \emptyset \Vdash e_t$.*

8.10 Complete type soundness

Guarded by the normal and constraint typing systems, the well-typed programs are type sound: they are evaluated to neither of errors. First, the soundness from the normal typing assures that the well type programs never fails in *err*. This is easy because we have already it:

Theorem 31 (Non-generic type soundness) *If a full constraint annotated expression e_t has a normal typing in the empty typing environment, $\emptyset \vdash e_t$, and $\emptyset \models e_t$ terminates to some result r in the modified semantics, then r is not err.*

PROOF: This is straightforward from Theorem 23. □

To complete the type soundness theorem, we need to work also with the constraint typing. First we define a new semantic typing with constraints, then prove the type soundness of the semantics using it.

Semantics with constraints

First of all, we modify semantic values and rules in Section 2.4 and Section 7.5.1. Expressions in closures and evaluation judgments are canonically extended with constraints. For example, in (e-gvar-r), the expression e_t and the variable $\mathtt{x}$ have the constraints in their annotations:

$$\textsf{(e-gvar-r)}\ \frac{\begin{array}{c}\exists S,\ \tau \leq_S \forall V.\tau' \\ E'\{\mathtt{y} = gclos(E', \mathtt{y}, V, e_t : \tau')\} \models S(e_t : \tau') \Longrightarrow r\end{array}}{E\{\mathtt{x} = gclos(E', \mathtt{y}, V, e_t : \tau')\} \models \mathtt{x}_{[\forall V.\tau'\ D]} : \tau \Longrightarrow r}$$

Semantic typing with constraints

We also extend the semantic typing so that it could know about the constraints. We replace (s-clos∗), (s-gclos∗), (s-gcase∗) and (s-env) by rules with constraints, while the others are kept as they were (Figure 8.3). In the figure, typing environments with constraints Γ are coerced into one without constraints in the normal typing judgment $\Gamma \vdash e_t$ and the generalization function *Gen*. In the rules, we use natural translation from type schemes to constraints. Type scheme σ is regarded as a generic constraint $[\sigma\ \{\}]$ if required.

$$\textbf{(s-bcnst)}\ \frac{\tau \leq \sigma}{b_\sigma \in \tau} \qquad \textbf{(s-cnst)}\ \frac{\tau \leq_S \sigma}{S(cnst(c_\sigma)) \in \tau}$$

$$\textbf{(s-gen)}\ gen(\mathrm{g}_\sigma, \tau) \in \tau \qquad \textbf{(s-tpl)}\ \frac{v_1 \in \tau_1 \quad v_2 \in \tau_2}{v_1, v_2 \in \tau_1 * \tau_2}$$

$$\textbf{(s-schm)}\ \frac{\forall \tau \leq_S \sigma,\ S(v) \in \tau}{v \in \sigma}$$

$$\textbf{(s-kclos)}\ \frac{\begin{array}{c}\exists \Gamma,\ E \in \Gamma \\ \Gamma \vdash \texttt{fun x} \rightarrow e_t : \tau_1 \rightarrow \tau_2 \\ \Gamma, \emptyset \Vdash \texttt{fun x} \rightarrow e_t : \tau_1 \rightarrow \tau_2\end{array}}{clos(E, \_, \mathtt{x}, e_t) \in \tau_1 \rightarrow \tau_2}$$

$$\textbf{(s-kclos-r)}\ \frac{\begin{array}{c}\exists \Gamma,\ E \in \Gamma \\ \Gamma\{\mathtt{f} : \forall V.\tau_1 \rightarrow \tau_2\} \vdash \texttt{fun x} \rightarrow e_t : \tau_1 \rightarrow \tau_2 \\ \Gamma\{\mathtt{f} : \forall V.\tau_1 \rightarrow \tau_2\}, V \Vdash \texttt{fun x} \rightarrow e_t : \tau_1 \rightarrow \tau_2 \\ \forall V.\tau_1 \rightarrow \tau_2 = Gen(\Gamma, \texttt{fun x} \rightarrow e_t)\end{array}}{clos(E, \mathtt{f}, \mathtt{x}, e_t) \in \forall V.\tau_1 \rightarrow \tau_2}$$

$$\textbf{(s-kenv)}\ \frac{\forall (\mathtt{x} : \mathcal{K}) \in \Gamma,\ E(\mathtt{x}) = sv,\ sv \in \mathcal{K}}{E \in \Gamma}$$

$$\textbf{(s-kgclos)}\ \frac{\begin{array}{c}\exists \Gamma \quad E \in \Gamma \quad \forall V.\tau = Gen(\Gamma, e_t : \tau) \quad D = Derive(V, \emptyset, e_t) \\ \Gamma \vdash e_t \quad \Gamma, V \Vdash e_t\end{array}}{gclos(E, \_, V, e_t : \tau) \in [\forall V.\tau\ D]}$$

$$\textbf{(s-kgclos-r)}\ \frac{\begin{array}{c}\exists \Gamma \quad E \in \Gamma \quad \forall V.\tau = Gen(\Gamma, e_t : \tau) \quad D = Derive(V, \emptyset, e_t) \\ \mu\kappa.\mathcal{K} \xrightarrow{\mu} [\forall V.\tau\ D] \\ \Gamma\{\mathtt{x} : \mu\kappa.\mathcal{K}\} \vdash e_t \quad \Gamma\{\mathtt{x} : \mu\kappa.\mathcal{K}\}, V \Vdash e_t\end{array}}{gclos(E, \mathtt{x}, V, e_t : \tau) \in \mu\kappa.\mathcal{K}}$$

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

$$\textbf{(s-kgcase-r)}\ \frac{\begin{array}{c}\exists \Gamma, E \in \Gamma, \\ \sigma_i = Gen(\Gamma, e_{ti}) \quad D_i = Derive(BV(\sigma_i), \emptyset, e_{ti}) \quad FV(\sigma_i) = \emptyset \\ \Gamma\{\mathtt{x} : \mu\kappa.\mathcal{K}\} \vdash e_{ti} \quad \Gamma\{\mathtt{x} : \mu\kappa.\mathcal{K}\}, BV(\sigma_i) \Vdash e_{ti} \\ \mathcal{K}' = [\ \sigma_1\ D_1 \mid \ \ldots\ \mid \sigma_n\ D_n\ ] \quad \mu\kappa.\mathcal{K} \xrightarrow{\mu} \mathcal{K}'\end{array}}{gcase(E, \mathtt{x}, [\sigma_1 \Rightarrow e_{t1}, \ldots, \sigma_n \Rightarrow e_{tn}]) \in \mu\kappa.\mathcal{K}}$$

Figure 8.3: Semantic typing rules with constraints

The rules for closures ($\mathsf{s\text{-}k}*$) check stored codes e_t in both normal and constraint typing systems. The same generalized type variables V must be used for the normal generalization and its corresponding constraint typing.

Theorem 32 (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 constraint typing environment Γ, 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: Proof by induction over the evaluation steps. The soundness of the non-generic part is already proven in Theorem 31, we just focus on generic stuffs:

Expressions related with non generic closures Although they have new conditions of constraint typings, proofs for their related cases are same as Theorem 31.

Variables of generic values Let $e_t : \tau = \mathtt{x}_{\mathcal{K}} : \tau$ and $generic(\mathcal{K}) = true$. Then from the semantic typing, $E(\mathtt{x})$ must be a generic closure, generic case closure, or their recursive versions:

Generic closures If $E(\mathtt{x}) = gclos(E', \_, V, e'_t : \tau')$, from the semantic typing of $E \in \Gamma$, $E(\mathtt{x}) \in \Gamma(\mathtt{x}) = \mathcal{K}$. By the semantic typing of the generic closure by ($\mathsf{s\text{-}kgclos}$), $\mathcal{K}$ must have the form of $[\forall V.\tau'\ D]$ for some derived constraint list $D = \{\tau_i <: \mathcal{K}_i\}$:

$$\frac{\exists\Gamma', \quad E' \in \Gamma', \quad \forall V.\tau' = Gen(\Gamma', e'_t : \tau') \quad D = \{\tau_i <: \mathcal{K}_i\} = Derive(V, \emptyset, e'_t) \quad \Gamma' \vdash e'_t \quad \Gamma', V \Vdash e'_t}{gclos(E', \_, V, e'_t : \tau') \in [\forall V.\tau'\ D]}$$

Since $polyinst(\emptyset, \mathcal{K}, \tau) = false$, $\mathtt{x}$ must be a full instance. The typings of $\mathtt{x}$ are by ($\mathsf{t\text{-}inst}$) and ($\mathsf{kt\text{-}full\text{-}inst}$) for some substitution S such that $Dom(S) = V$.

$$\frac{\Gamma(\mathtt{x}) = coerce(\mathcal{K}) = \forall V.\tau' \quad \tau \leq_S \forall V.\tau'}{\Gamma \vdash \mathtt{x}_{\forall V.\tau'} : \tau}$$

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

Because $S(\tau') = \tau$ and $DV(\tau) = \emptyset$, $\forall \nu \in Dom(S) = V, DV(S(\nu)) = \emptyset$. From $\tau <: [\forall V.\tau'\ D]$, for all i, $S(\tau_i) <: S(\mathcal{K}_i)$. Therefore, we can use Lemma 32 against the constraint typing of e'_t and obtain $S(\Gamma'), \emptyset \Vdash S(e'_t) : S(\tau') = \tau$. From the generalization, $FV(\Gamma') \cap V = \emptyset$ and $S(\Gamma') = \Gamma'$. Hence,

$$\Gamma', \emptyset \Vdash S(e'_t) : \tau$$

Also from the normal type substitution lemma Theorem 3,

$$\Gamma' \vdash S(e'_t) : \tau$$

The variable is evaluated by ($\mathsf{e\text{-}gvar}$):

$$\frac{\tau \leq_S \forall V.\tau' \quad E' \models S(e'_t : \tau') \Longrightarrow r}{E\{\mathtt{x} = gclos(E', \_, V, e'_t : \tau')\} \models \mathtt{x}_{[\forall V.\tau'\ D]} : \tau \Longrightarrow r}$$

We can use the induction hypothesis for the evaluation $E' \models S(e'_t) \Longrightarrow r$, and obtain that r must be a value v such that $v \in \tau$.

Recursive generic closures In this case $E(\mathtt{x}) = gclos(E', \mathtt{y}, V, e'_t : \tau')$. The semantic typing $E \in \Gamma$ leads to $E(\mathtt{x}) \in \Gamma(\mathtt{x})$. By the semantic typing rule of the generic closure (s-kgclos-r), $\Gamma(\mathtt{x}) = \mu\kappa.\mathcal{K} = \mu\kappa.[\forall V.\tau'\ D]$ where

$$\begin{array}{l} \mu\kappa.[\forall V.\tau'\ D] \xrightarrow{\mu} [\forall V.\tau'\ D'] \\ D' = Derive(V, \emptyset, e'_t) \\ \forall V.\tau' = Gen(\Gamma, e'_t) \\ \Gamma\{\mathtt{y} : \mu\kappa.\mathcal{K}\} \vdash e'_t \quad \Gamma\{\mathtt{y} : \mu\kappa.\mathcal{K}\} \Vdash e'_t \end{array}$$

Since $polyinst(\emptyset, \mu\kappa.\mathcal{K}, \tau) = false$, $\mathtt{x}$ is a full instance. The typings of $\mathtt{x}$ are by (t-inst) and (kt-full-inst) for some substitution S such that $Dom(S) = V$.

$$\frac{\Gamma(\mathtt{x}) = coerce(\mu\kappa.\mathcal{K}) = \forall V.\tau' \quad \tau \leq_S \forall V.\tau'}{\Gamma \vdash \mathtt{x}_{\forall V.\tau'} : \tau}$$

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

Because $S(\tau') = \tau$ and $DV(\tau) = \emptyset$, $\forall \nu \in Dom(S) = V, DV(S(\nu)) = \emptyset$. From $\tau <: \mu\kappa.\mathcal{K}$, if we let $D' = \{\tau_i <: \mathcal{K}_i\}$, for all i, $S(\tau_i) <: S(\mathcal{K}_i)$. We can use Lemma 32 against the constraint typing of e'_t and obtain $S(\Gamma'\{\mathtt{x} : \mu\kappa.\mathcal{K}\}), \emptyset \Vdash S(e'_t) : S(\tau') = \tau$. From the generalization, $FV(\Gamma') \cap V = \emptyset$. Since $Dom(S) = V$, $S([\forall V.\tau'\ D']) = [\forall V.\tau'\ D']$. From this, we can easily prove that $S(\mu\kappa.\mathcal{K}) = \mu\kappa.\mathcal{K}$. Hence, $S(\Gamma'\{\mathtt{y} : \mu\kappa.\mathcal{K}\}) = \Gamma'\{\mathtt{y} : \mu\kappa.\mathcal{K}\}$ and

$$\Gamma'\{\mathtt{y} : \mu\kappa.\mathcal{K}\}, \emptyset \Vdash S(e'_t) : \tau$$

Also from the normal type substitution lemma Theorem 3,

$$\Gamma'\{\mathtt{y} : \mu\kappa.\mathcal{K}\} \vdash S(e'_t) : \tau$$

The variable is evaluated by (e-gvar-r):

$$\frac{\begin{array}{c}\exists S,\ \tau \leq_S \forall V.\tau' \\ E'\{\mathtt{y} = gclos(E', \mathtt{y}, V, e'_t : \tau')\} \models S(e'_t : \tau') \Longrightarrow r\end{array}}{E\{\mathtt{x} = gclos(E', \mathtt{y}, V, e'_t : \tau')\} \models \mathtt{x}_{\mu\kappa.\mathcal{K}} : \tau \Longrightarrow r}$$

Since $glos(E', \mathtt{y}, V, e'_t) \in \mu\kappa.\mathcal{K}$, we can use the induction hypothesis for the evaluation $E'\{\mathtt{y} = gclos(E', \mathtt{y}, V, e'_t : \tau')\} \models S(e'_t) \Longrightarrow r$, and obtain that r must be a value v such that $v \in \tau$.

Generic case closures In the case $E(\mathtt{x}) = gcase(E', \_, [\sigma_1 \Rightarrow e_{t1},\ \ldots,\ \sigma_n \Rightarrow e_{tn}])$, from $E(\mathtt{x}) \in \Gamma(\mathtt{x})$, if we let $\Gamma(\mathtt{x}) = \mathcal{K}$, the semantic typing of $E(\mathtt{x})$ is done by (s-kgcase) as follows:

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

Like in the case of generic closures, since $\mathit{polyinst}(\emptyset, \mathcal{K}, \tau) = \mathit{false}$, $\mathtt{x}$ must be a full instance. Therefore its constraint typing is done by (kt-full-inst):

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

Because the constraint $\mathcal{K}$ is a pattern list $[\ \sigma_1\ D_1\ | \ldots |\ \sigma_n\ D_n\ ]$ and $\tau <: \mathcal{K}$, by (k-fail-case), there must exist at least one type pattern which matches with τ: if we let the first such matched pattern be $\sigma_p\ d_p$, there exists a pattern substitution S whose domain is $BV(\sigma_p)$ and:

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

If we let $D_p = \{\tau_j <: \mathcal{K}_j\}$, for all j, $S(\tau_j) <: S(\mathcal{K}_j)$.

Since $\tau \leq_S \sigma_p$ and $DV(\tau) = \emptyset$, $\forall \nu \in \mathit{Dom}(S), DV(S(\nu)) = \emptyset$ and we can use Lemma 32 and obtain $S(\Gamma'), \emptyset \Vdash S(e_{tp}) : \tau$. From the generalization of e_{tp}, we have $S(\Gamma') = \Gamma'$ and

$$\Gamma', \emptyset \Vdash S(e_{tp}) : \tau$$

We obtain $\Gamma' \vdash S(e_{tp})$ from the normal typing $\Gamma' \vdash e_{tp}$ using Theorem 3.

The evaluation of $\mathtt{x}_{\mathcal{K}} : \tau$ under E is done by (e-gcase) as follows:

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

From all the above we can use the induction hypothesis and get $E' \models S(e_{tp}) \Longrightarrow v$ for some value v.

Recursive generic case closures Provable as same as the non recursive generic case closures, using also the discussions we have done in the case of the recursive generic closures.

Generic let $e_t = \mathtt{let}\ \mathtt{x}_{V,\mathcal{K}} \mathtt{=} e_{t1}\ \mathtt{in}\ e_{t2}$. Because $\Gamma \vdash e_t$ and $\Gamma, \emptyset \Vdash e_t$, the typings must be done by (t-let) and (kt-let):

$$\frac{\begin{array}{c}\Gamma \vdash e_{t1} : \tau_1 \quad \forall V.\tau_1 = \mathit{Gen}(\Gamma, e_{t1} : \tau_1) \\ \Gamma\{\mathtt{x} : \forall V.\tau_1\} \vdash e_{t2} : \tau\end{array}}{\Gamma \vdash \mathtt{let}\ \mathtt{x}_V \mathtt{=} e_{t1}\ \mathtt{in}\ e_{t2}} \qquad \frac{\begin{array}{c}\Gamma, V \Vdash e_{t1} \quad \Gamma\{\mathtt{x} : \mathcal{K}\}, \emptyset \Vdash e_{t2} \\ \mathcal{K} = [\forall V.\tau_1\ D] \\ D = \mathit{Derive}(V, \emptyset, e_{t1}) \quad \mathit{adm}^+(\emptyset, \mathcal{K}) = \mathit{true}\end{array}}{\Gamma, \emptyset \Vdash \mathtt{let}\ \mathtt{x}_{V,\mathcal{K}} \mathtt{=} e_{t1}\ \mathtt{in}\ e_{t2}}$$

The evaluation is done by (e-g-let):

$$\frac{V \cap \mathbb{D} \neq \emptyset \quad E\{\mathtt{x} = gclos(E, \_, V, e_{t1})\} \models e_{t2} \Longrightarrow r}{E \models \mathtt{let\ rec\ x}_{V,\mathcal{K}}\,\mathtt{=}\,e_{t1}\ \mathtt{in}\ e_{t2} \Longrightarrow r}$$

Using (s-kgclos), the typing of the generic closure $gclos(E, \_, V, e_{t1})$ is:

$$\frac{\begin{array}{c} E \in \Gamma \quad \forall V.\tau_1 = Gen(\Gamma, e_{t1}) \quad D = Derive(V, \emptyset, e_{t1}) \\ \Gamma \vdash e_{t1} \quad \Gamma, V \Vdash e_{t1} \end{array}}{gclos(E, \_, V, e_{t1}) \in [\forall V.\tau_1\ D] = \mathcal{K}}$$

It leads to $E\{\mathtt{x} = gclos(E, \_, V, e_{t1})\} \in \Gamma\{\mathtt{x} : \mathcal{K}\}$. We can apply the induction hypothesis to the evaluation of e_{t2}: r must be a value $v \in \tau$.

Generic recursive let This time, $e_t = \mathtt{let\ rec\ x}_{V,\mu\kappa.\mathcal{K}}\,\mathtt{=}\,e_{t1}\ \mathtt{in}\ e_{t2}$. The attached constraint to $\mathtt{x}$ must be a recursive constraint $\mu\kappa.\mathcal{K} = \mu\kappa.[\forall V.\tau_1\ D]$ from the following typings by (t-letrec) and (kt-letrec):

$$\frac{\begin{array}{c} \Gamma\{\mathtt{x} : \tau_1\} \vdash e'_{t1} : \tau_1 \quad \forall V.\tau_1 = Gen(\Gamma\{\mathtt{x} : \tau_1\}, e'_{t1} : \tau_1) \\ \Gamma\{\mathtt{x} : \forall V.\tau_1\} \vdash e_{t2} : \tau \quad e_{t1} = \{\mathtt{x}_{\tau_1} \Rightarrow \mathtt{x}_{\forall V.\tau_1}\} e'_{t1} \end{array}}{\Gamma \vdash \mathtt{let\ rec\ x}_V\,\mathtt{=}\,e_{t1}\ \mathtt{in}\ e_{t2}}$$

$$\frac{\begin{array}{c} \Gamma\{\mathtt{x} : \mu\kappa.\mathcal{K}\}, V \Vdash e_{t1} \quad \Gamma\{\mathtt{x} : \mu\kappa.\mathcal{K}\}, \emptyset \Vdash e_{t2} \\ D' = Derive(V, \emptyset, e_{t1}) \quad \mu\kappa.\mathcal{K} = \mu\kappa.[\forall V.\tau_1\ D] \xrightarrow{\mu} [\forall V.\tau\ D'] \\ adm^{+}(\emptyset, \mu\kappa.\mathcal{K}) = true \end{array}}{\Gamma, \emptyset \Vdash \mathtt{let\ rec\ x}_{V,\mu\kappa.\mathcal{K}}\,\mathtt{=}\,e_{t1}\ \mathtt{in}\ e_{t2}}$$

From the normal typing of e'_{t1} and its generalization, $\Gamma\{\mathtt{x} : \forall V.\tau_1\} \vdash e_{t1} : \tau_1$ by Lemma 7.

The evaluation is done by (e-g-letrec):

$$\frac{V \cap \mathbb{D} \neq \emptyset \quad E\{\mathtt{x} = gclos(E, \mathtt{x}, V, e_{t1})\} \models e_{t2} \Longrightarrow r}{E \models \mathtt{let\ rec\ x}_{V,\mu\kappa.\mathcal{K}}\,\mathtt{=}\,e_{t1}\ \mathtt{in}\ e_{t2} \Longrightarrow r}$$

From the above typings by (s-kgclos-r),

$$\frac{\begin{array}{c} E \in \Gamma \quad \forall V.\tau_1 = Gen(\Gamma, e_{t1}) \quad D' = Derive(V, \emptyset, e_{t1}) \quad \mu\kappa.\mathcal{K} \xrightarrow{\mu} [\forall V.\tau_1\ D'] \\ \Gamma\{\mathtt{x} : \mu\kappa.\mathcal{K}\} \vdash e_{t1} \quad \Gamma\{\mathtt{x} : \mu\kappa.\mathcal{K}\}, V \Vdash e_{t1} \end{array}}{gclos(E, \mathtt{x}, V, e_{t1}) \in \mu\kappa.\mathcal{K}}$$

Therefore $E\{\mathtt{x} = gclos(E, \mathtt{x}, V, e_{t1})\} \in \Gamma\{\mathtt{x} : \mu\kappa.\mathcal{K}\}$. We can use the induction hypothesis for the evaluation of e_{t2}. r is a value v where $v \in \tau$.

Generic bindings This is the case $e_t = \mathtt{generic\ x}_{\mathcal{K}}\,\mathtt{=}\,\mathtt{case}\ \sigma_1 \Rightarrow e_{t1} \mid \ldots \mid \sigma_n \Rightarrow e_{tn}\ \mathtt{in}\ e'_t$.

The normal typing is by (t-generic):

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

and (kt-generic) for the constraint typing:

$$\frac{\begin{array}{c}\Gamma, V_i \Vdash e_{ti} : \tau_i \quad D_i = Derive(V_i, \emptyset, e_{ti}) \\ BV(\sigma_i) = V_i \quad FV(\sigma_i) = \emptyset \\ \mathcal{K} = [\sigma_1\ D_1 \mid \ldots \mid \sigma_n\ D_n] \quad adm^+(\emptyset, \mathcal{K}) = true \\ \Gamma\{\mathtt{x} : \mathcal{K}\}, \emptyset \Vdash e'_t\end{array}}{\Gamma, \emptyset \Vdash \mathtt{generic}\ \mathtt{x}_{\sigma\mathcal{K}}\ \mathtt{=}\ \mathtt{case}\ \sigma_1 \Rightarrow e_{t1} \mid \ldots \mid \sigma_n \Rightarrow e_{tn}\ \mathtt{in}\ e'_t}$$

The expression is evaluated by (e-generic):

$$\frac{E\{\mathtt{x} = gcase(E, \_, [\sigma_1 \Rightarrow e_{t1}, \ldots, \sigma_n \Rightarrow e_{tn}])\} \models e'_t \Longrightarrow r}{E \models \mathtt{generic}\ \mathtt{x}_{\mathcal{K}}\ \mathtt{=}\ \mathtt{case}\ \sigma_1 \Rightarrow e_{t1} \mid \ldots \mid \sigma_n \Rightarrow e_{tn}\ \mathtt{in}\ e'_t \Longrightarrow r}$$

The generic case closure created by this evaluation can be typed by (s-kgcase), using premises obtained from the typings:

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

We can use the induction hypothesis for the evaluation of e'_t and it must be successful, since from the semantic typing, $E\{\mathtt{x} = gcase(E, \_, [\sigma_1 \Rightarrow e_{t1}, \ldots, \sigma_n \Rightarrow e_{tn}])\} \in \Gamma\{\mathtt{x} : \mathcal{K}\}$ and the normal and constraint typings give us $\Gamma\{\mathtt{x} : \mathcal{K}\} \vdash e'_t : \tau$ and $\Gamma\{\mathtt{x} : \mathcal{K}\}, \emptyset \Vdash e'_t$.

Recursive generic bindings This case is proven in the same way as the case of non recursive generic bindings. The rules in the proof must be replaced by their recursive counterparts.

□

This theorem is immediately simplified to the following statement:

Theorem 33 (Generic type soundness of well-typed programs) *Let e_t be a full constraint annotated, well-typed program. Then evaluation $\emptyset \models e_t$ returns neither err nor gerr, but always some value v.*

PROOF: This is a special case of Theorem 32, when $\Gamma = \emptyset$ and $E = \emptyset$. □

8.11 Dynamic variable attribute in constraints

The distinction of regular and dynamic type variables has played essential role in every aspect of extensional polymorphism: semantics, typing and compilation. With the help of generic constraints, however, these two kinds of type variables can be merged into one. This merge simplifies the type algebra, identical to the one of the normal ML. With small assumption and modification to generic constraints, "dynamic" attribute of type variables can be deduced by examining constraints. We here outline this idea.

Pattern constraints

If we suppose that user-defined generic values always have more than one type cases, user-defined generic values and the other values can be distinguished by checking whether their constraints contain only one type pattern constraint or more. For example, a normal polymorphic value `List.map` has the following constraint of singleton type pattern:

$$\texttt{List.map : } [\forall \alpha\beta \texttt{.} (\alpha \rightarrow \beta) \rightarrow \alpha \texttt{ list} \rightarrow \beta \texttt{ list}]$$

identical to its type scheme in the normal typing. On the other hand, a use generic value `plus` which is overridden by addition functions for integers and floats have the following constraint with two type patterns:

$$\mathcal{K}_{\texttt{plus}} = [\texttt{int} \rightarrow \texttt{int} \rightarrow \texttt{int | float} \rightarrow \texttt{float} \rightarrow \texttt{float}]$$

Its type scheme in the normal typing is identical to the least general anti-unifier of these type patterns, and therefore in the new simplified type algebra, it is $\forall \alpha \texttt{.} \alpha \rightarrow \alpha \rightarrow \alpha$. Though it has no longer dynamic attribute, looking at its generic constraint with two patterns, it is clear that `plus` is a generic value and the type scheme must be dynamic, which is $\forall \delta \texttt{.} \delta \rightarrow \delta \rightarrow \delta$ in the current type algebra.

Derived constraints

Derived generic values have only one element of type pattern, same as the normal values. But their dynamic type scheme can be deduced, since they have derived constraints. For example, the following derived generic value `double`

```
let double = fun x → plus x x;;
```

has a constraint $[\forall \delta \texttt{.} \delta \rightarrow \delta \; \{\delta \rightarrow \delta \; <: \mathcal{K}_{\texttt{plus}}\}]$ in the current type system. In the new simplified type algebra, we can remove the explicit dynamic attribute of δ, since it is clear from the existence of the derived constraint:

$$[\forall \alpha \texttt{.} \alpha \rightarrow \alpha \; \{\alpha \rightarrow \alpha \; <: \mathcal{K}_{\texttt{plus}}\}]$$

and the type scheme of `double` can be written as $\forall \alpha \texttt{.} \alpha \rightarrow \alpha$.

Generic primitives

Since generic primitives are defined as totally polymorphic, their type schemes are singleton lists of their type schemes, $[\sigma]$. Here we cannot just remove the dynamic attributes from σ, since such "undynamized" constraints of singleton type pattern without derived constraints could not distinguished from constraints of non-generic values. Therefore we introduce a new form of derived constraint $\mathtt{generic}(\alpha)$ to indicate their genericity explicitly. For example, a generic primitive `print` whose type scheme is $\forall\delta.\delta \rightarrow \mathtt{unit}$ in the current type system will have the following generic constraint in the new simplified type algebra:

$$[\forall\alpha.\alpha \rightarrow \mathtt{unit}\ \{\mathtt{generic}(\alpha)\}]$$

Thanks to this new constraint, `print` is identified as a generic primitive.

Dynamic attribute is still useful internally

Thus, using generic constraints, the distinction between regular and dynamic type variables is no longer necessary. This will lower the barrier to use extensional polymorphism a lot, since users need not learn their difference. At a glance, the normal typing layer of extensional polymorphism is now equivalent to the classical ML typing, except the newly added `generic` bindings.

In spite of this simplification of type algebra, the original dynamic attribute of type variables is still helpful for the constraint typing and compilation, since they handle generic values differently from non-generic values. Therefore in the following chapters, we will continue using this distinction of type variables.

8.12 Related works

8.12.1 Abstract interpretation

In the initial study of extensional polymorphism by Dubois, Rouaix and Weis [DRW95], static detection of type failures of generic instances is not by types, but based on abstract interpretation of programs. The abstraction of a program is a predicate on types that captures type abstractions, type matching, and type applications extracted from its compilation. For example:

```
generic plus = case
| int → int → int ⇒ (+)
| float → float → float ⇒ (+.)
in
let double = fun x → plus x x in
double 1;;
```

```
let abs_plus t = match t with
| «int» → true
| «float» → true
| _ → false (* type match failure *)
in
let abs_double =
  fun t → abs_plus t in
abs_double «int»;;
```

The left hand program has the abstraction in the right. Evaluating this abstraction and simulates all the possible type applications and matchings, the type safety of generic instances is verified.

Abstractions are not types and are too abstract, therefore type system cannot make use of them for overloading resolution. As a result, type schemes of [DRW95] are always full parametric polymorphic, which are same as in our normal typing layer. The lack of non-exhaustiveness of generic values in type system prevents overload resolution of generic instances: users must often manually constrain instance types of generic values, even when overloading resolutions are trivial from definitions.

Our generic constraints can be regarded as a translation of these abstraction interpretations in type algebra. Having non-exhaustive information of generic values, type reconstruction now can resolve their instance types.

8.13 Conclusion and discussion

In this chapter we constructed generic constraint type system which statically detects all the invalid generic instances which could raise generic run time type errors. Combining this and the normal typing, we can make sure that well-typed GML programs cannot be evaluated to any kind of errors.

Two-layer design

We formalized the constraint typing as an independent system of the normal typing which is based on the classical ML typing. This separation benefits both theory and implementation. As we have seen, to detect illegal user-defined generic instance types which cannot match with any type pattern, type information of values in the constraint typing is inevitable to become much more complicated than normal type schemes. Migrating such type information directly into the normal and classical type system would extend it to something too hard to understand. Thanks to the two-layer design of type systems, the first layer of the normal typing is kept as a mild extension of ML typing system. Also in implementation, we can implement type reconstruction into two steps: first we type expressions by the normal type reconstruction algorithm in Section 2.6. This is rather easily achieved by modifying some existing ML implementation to add dynamic type variables and related rules. Then type-annotated expressions by this first reconstruction are sent to the second reconstruction phase which detects invalid generic instances. Since code for the first phase is isolated from the second, maintenance of the first phase can be done even by who do not understand the generic constraint typing at all.

Generic constraints

In the generic constraint typing system, a value is associated with a generic constraint, type information which denotes the set of the types where semantics of the value is defined. A constraint is basically represented as a list of type pattern constraints, type scheme where the corresponding value is well defined. Pattern constraints may also contain derived constraints, the information of internal polymorphic generic instances whose validity is dependent on full instantiation of the whole constraint. Constraints may also be abstracted by constraint variables to express recursiveness. Due to this recursively defined constraints, valid generic instance type relationship $\tau <: \mathcal{K}$ is undecidable, unless we introduce some restriction to the recursive constraints. In order to make the instance relationship decidable, we introduced admissible constraint restriction which limits constraints so that size of run time dispatching of generic values must decrease monotonically (but need not to be strictly) in the course of recursive calls of GML compilation. Unfortunately, we have some useful but inadmissible generic values which are therefore impossible to define, even though their instance check always terminates. In actual implementation, however, this admissibility restriction will not be necessary, since in type reconstruction of generic instances (Chapter 10) we use another restriction which limits maximum number of reconstruction algorithm recursive calls. Under this restriction, type checking is assured to always terminate even with non-admissible constraints. However, since non-admissible generic definitions tend to be meaningless, our type checker should print out warnings when it finds them.

Typing rules

Typing rules were defined using generic constraints. Under an assumption that expressions are already well-typed by the normal typing, these rules only check generic related constructions. To prove the generic type soundness, we extended the semantic typing system so that they can be also aware of constraint typing, additional to the normal typing. In some sense this is the only part where both typing layers meet together; the soundness proof uses typings of both layers intensively.

No free dynamic variables

The free occurrences of dynamic type variables are restricted to the generalized variable set of typing judgment: Lemma 29 says that if $\Gamma, V \Vdash e_t$ holds, $DV(e_t) \subset V$. This means that in a well-typed program e_t whose typing is $\emptyset, \emptyset \Vdash e_t$, dynamic type variables cannot occur freely: they must be generalized by some polymorphism. This restriction might be too strong. For example, in the following program,

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

the instance `plus` cannot have a type δ `list` $\rightarrow \delta$ `list` $\rightarrow \delta$ `list`, even though the expression can be evaluated to a null list without any problem with this type annotation.

It is highly possible to remove the restriction of free dynamic type variable occurrences from the generic constraint type system without big difficulty, but it is a future issue.

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` → `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` → `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` → δ `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:

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

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.

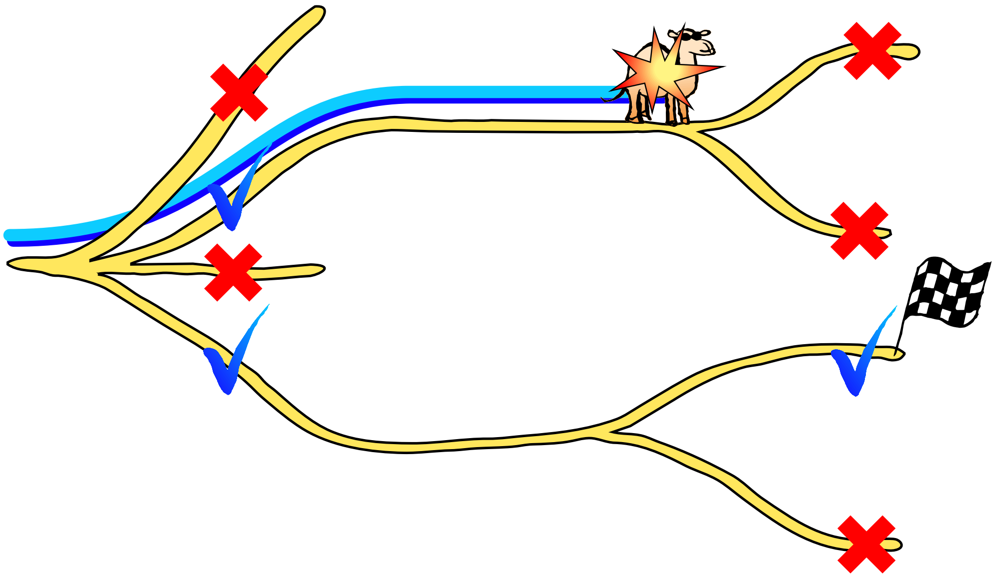
The core of this problem is not in ordering of type cases, but case selection strategy of the semantics of user generics itself. Generally, when a generic definition have two type patterns which share some instance type which match τ, and semantics for τ is not defined in the first matched type case 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 τ is an instance of some type pattern σ_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 τ 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 σ_i which matched with τ, but τ was not a full

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

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

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

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

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 τ be a type expression, $\mathcal{K}$ be a relaxed admissible constraint M be a memo function for $kinst^+$ and ψ be a constraint substitution for μ-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,

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

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 τ 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 τ 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

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

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

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

By renaming bound variables in σ, 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 $S(\tau) \not\leq S(\sigma)$, then $\tau \not\leq \sigma$ and using (k-fail-derive), we have $\tau \not<: \sigma\ \{\tau_i <: \mathcal{K}_i\}$.

Otherwise, when $S(\tau) \leq_{S'} S(\sigma)$, we have two cases: $\tau \leq \sigma$ or not. If $\tau \not\leq \sigma$, we have immediately $\tau \not<: \sigma\ \{\tau_i <: \mathcal{K}_i\}$ from (k-fail-derive). If $\tau \leq \sigma$, let S'' be a substitution such that $\tau \leq_{S''} \sigma$. From Lemma 4, $S(\tau) \leq_{S'} S(\sigma)$ 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 τ.

□

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 Γ, 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 Γ 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 Γ 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 x is typed under the empty set of generalized type variables, 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 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 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):

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

```
# 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 τ `list list` $\rightarrow \tau'$ `list` for some τ and τ', we proceed to the first case only when `flat` : τ `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 $\texttt{int list}^{n-2} \to \texttt{int list}$, ..., $\texttt{int list}^1 \to \texttt{int list}$ are valid instances of `flat`. In the recursive call of `flat` with the type $\texttt{int list}^{n-1} \to \texttt{int list}$ the compilation checks again the validity of $\texttt{int list}^{n-2} \to \texttt{int list}$ with $\mathcal{K}_{\texttt{flat}}$, which needs $\texttt{int list}^i \to \texttt{int list} <: \mathcal{K}_{\texttt{flat}}$ 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

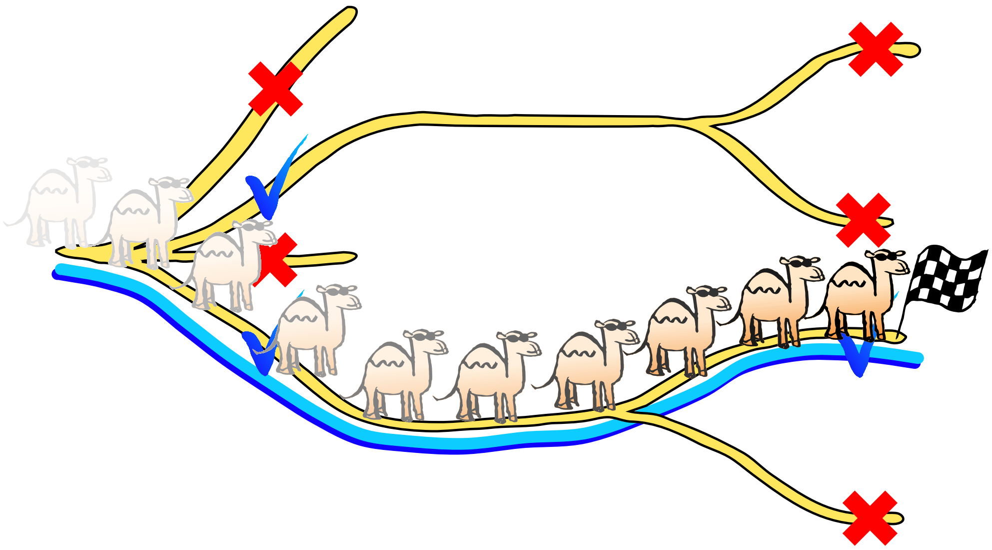
As we have seen, the computation complexity of type case selection with backtracking using the run time type pattern matching is too high to be performed. In the compilation using pattern guards, all the validity checks happen in the recursive calls of generic value have been actually done in its first 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

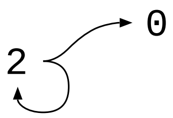
A *generic flow* is a directed integer graph which is basically a tree but which may have some loops. The formal definition of generic flows are given as follows, using ML type declaration:

```
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

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

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 ψ) 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 κ. 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 $F($`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.

$$\lfloor c_\sigma : \tau \rfloor_F = c_\sigma$$
$$\lfloor \texttt{fun x} \rightarrow e_t \rfloor_F = \texttt{fun x} \rightarrow \lfloor e_t \rfloor_F$$
$$\lfloor e_{t1}\ e_{t2} \rfloor_F = \lfloor e_{t1} \rfloor_F\ \lfloor e_{t2} \rfloor_F$$
$$\lfloor e_{t1}, e_{t2} \rfloor_F = \lfloor e_{t1} \rfloor_F,\ \lfloor e_{t2} \rfloor_F$$

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

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

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

Figure 9.5: The compilation algorithm with generic flows

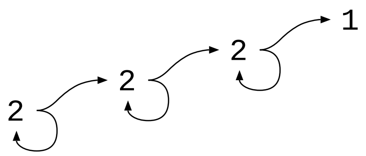
Example The previous `print` generic function is compiled as follows:

```
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 τ `list` → `unit`, where τ 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`, $\rightarrow$, α, 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.

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

If the type checker finds a generic primitive instance whose type scheme is σ is evaluated with type τ, 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 δ_i of σ is instantiated in τ. 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:

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

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»|]` :

$$1 \longrightarrow 1 \longrightarrow 0 \longrightarrow \ll\texttt{(int * float) * string}\gg$$

□

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
```

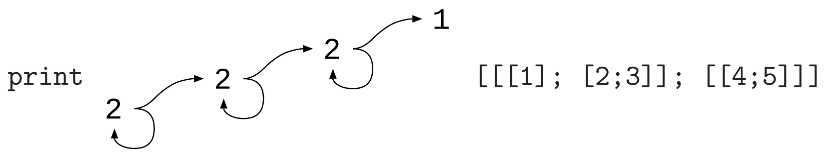
This definition of `print` is transformed to the following instance definitions by dictionary passing. The instance for printing lists, $\texttt{print}_{list}$ is compiled to a function which takes a resolution `f` of the internal use of `print`:

`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:



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.

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 $\forall\delta.\mathtt{unit} \rightarrow \delta$. 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 $\forall\delta.\mathtt{unit} \rightarrow \delta$ and constraint $[\forall\delta.\mathtt{unit} \rightarrow \delta]$, 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 $\mathtt{gread} : \mathtt{unit} \rightarrow \delta$, 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} \to e_t) : \tau_1 \to \tau_2) & = & (\mathtt{fun\ x} \to 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}\, \mathtt{,}\, e_{t2}) : \tau) & = & (kassign(K, e_{t1}),\ kassign(K, e_{t2})) : \tau \\
kassign(K, (\mathtt{let\ x}_V \mathtt{=} 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}} \mathtt{=} e'_{t1}\ \mathtt{in}\ e'_{t2}) : \tau \\
\multicolumn{3}{l}{kassign(K, (\mathtt{let\ rec\ f}_V \mathtt{=} 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} \\
& & \qquad \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} \\
& & \qquad \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.

$$
\begin{array}{rcl}
\mathit{fullinsts}(V,\ c_{[\sigma]} : \tau) & = & \emptyset \\
\mathit{fullinsts}(V,\ \mathtt{g}_{[\sigma]} : \tau) & = & \text{if } \mathit{polyinst}(V,\sigma,\tau) = \mathit{false} \text{ then } \{\mathtt{g} : \tau\} \text{ else } \emptyset \\
\mathit{fullinsts}(V,\ \mathtt{x}_{\mathcal{K}} : \tau) & = & \text{if } \mathit{polyinst}(V,\mathcal{K},\tau) = \mathit{false} \text{ then } \{\mathtt{x} : \tau\} \text{ else } \emptyset \\
\mathit{fullinsts}(V,\ \mathtt{fun\ x} \rightarrow e_t) & = & \mathit{fullinsts}(V, e_t) \\
\mathit{fullinsts}(V,\ e_{t1}\ e_{t2}) & = & \mathit{fullinsts}(V, e_{t1}) \cup \mathit{fullinsts}(V, e_{t2}) \\
\mathit{fullinsts}(V,\ (e_{t1},\ e_{t2})) & = & \mathit{fullinsts}(V, e_{t1}) \cup \mathit{fullinsts}(V, e_{t2}) \\
\multicolumn{3}{l}{\mathit{fullinsts}(V, \mathtt{let\ (rec)\ x}_{V',\mathcal{K}} \mathtt{=} e_{t1} : \tau\ \mathtt{in}\ e_{t2}) =} \\
 & & \mathit{fullinsts}(V \cup V', e_{t1}) \cup \mathit{fullinsts}(V, e_{t2}) \\
\multicolumn{3}{l}{\mathit{fullinsts}(V, \mathtt{generic\ (rec)\ x}_{\mathcal{K}} \mathtt{=case}\ \sigma_1 \Rightarrow e_{t1} \mid \ldots \mid \sigma_n \Rightarrow e_{tn}\ \mathtt{in}\ e_t) =} \\
 & & \bigcup_{i=1}^{n} \mathit{fullinsts}(V \cup BDV(\sigma_i), e_{ti})
\end{array}
$$

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:

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

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

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:

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

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.

Chapter 11

Generic value extensions

11.1 Introduction

Extensible generic values

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

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

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

Closed world assumption with open recursion

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

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

Type case inclusion

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

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

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

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

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

Type case compilation with self

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

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

Outline of this chapter

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

11.2 Open or closed

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

11.2.1 Open world assumption

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

11.2.2 Closed world assumption

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

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

11.3 Extension syntax and semantics

11.3.1 Extension is not mere redefinition

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

11.3.2 Open recursion in the closed world

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

11.3.3 Type case inclusion

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

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

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

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

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

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

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

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

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

should be equivalent to the following generic value:

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

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

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

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

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

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

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

11.3.4 New syntax

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

tc	$::=$	$\sigma \Rightarrow e$	Type case definitions
	$\mid$	`include x`	Type case inclusions
tc_t	$::=$	$(\sigma \Rightarrow e_t) : \sigma$	Type case definitions
	$\mid$	`include x` $: \sigma$	Type case inclusions

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

11.4 Typing

11.4.1 Pattern constraints concatenation

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

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

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

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

The following extended generic value `print'`

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

will have the extended constraint as follows:

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

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

11.4.2 Normal typing layer

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

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

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

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

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

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

11.4.3 Constraint typing

Pattern constraint concatenation

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

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

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

let $\mathcal{K}_1 = [\mathtt{int} \rightarrow \mathtt{unit} \mid \mathtt{float} \rightarrow \mathtt{unit}]$
let $\mathcal{K}_2 = \mu\kappa.[\mathtt{string} \rightarrow \mathtt{unit} \mid \forall\delta.\delta\ \mathtt{list} \rightarrow \mathtt{unit}\ \{\delta \rightarrow \mathtt{unit} <: \kappa\}]$ in

$$\begin{aligned} \mathcal{K}_1 \oplus \mathcal{K}_2 = \mu\kappa.[&\mathtt{int} \rightarrow \mathtt{unit} \mid \mathtt{float} \rightarrow \mathtt{unit} \mid \mathtt{string} \rightarrow \mathtt{unit} \\ &\mid \forall\delta.\delta\ \mathtt{list} \rightarrow \mathtt{unit}\ \{\delta \rightarrow \mathtt{unit} <: \kappa\}] \end{aligned}$$

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

let $\mathcal{K}_1 = \mu\kappa.[\mathtt{int} \rightarrow \mathtt{unit} \mid \forall\delta.\delta\ \mathtt{list} \rightarrow \mathtt{unit}\ \{\delta \rightarrow \mathtt{unit} <: \kappa\}]$ in
let $\mathcal{K}_2 = \mu\kappa'.[\mathtt{string} \rightarrow \mathtt{unit} \mid \forall\delta.\delta\ \mathtt{array} \rightarrow \mathtt{unit}\ \{\delta \rightarrow \mathtt{unit} <: \kappa'\}]$ in

$$\begin{aligned} \mathcal{K}_1 \oplus \mathcal{K}_2 = \mu\kappa.[&\mathtt{int} \rightarrow \mathtt{unit} \mid \forall\delta.\delta\ \mathtt{list} \rightarrow \mathtt{unit}\ \{\delta \rightarrow \mathtt{unit} <: \kappa\}] \\ &\mid \mathtt{string} \rightarrow \mathtt{unit} \mid \forall\delta.\delta\ \mathtt{array} \rightarrow \mathtt{unit}\ \{\delta \rightarrow \mathtt{unit} <: \kappa\}] \end{aligned}$$

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

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

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

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

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

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

Figure 11.1: Constraint typing rules with type case inclusion

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

□

11.5 Compilation

11.5.1 The outline

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

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

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

print:τ
```

are compiled as below:

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

print (kinst⁺(∅, τ <: 𝒦_print))
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

For example, the following expression

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

is compiled to the code

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

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

11.5.2 The algorithm

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

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

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

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

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

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

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

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

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

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

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

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

Figure 11.2: The compilation algorithm with generic flows

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

11.6 Removal of duplicated type cases

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

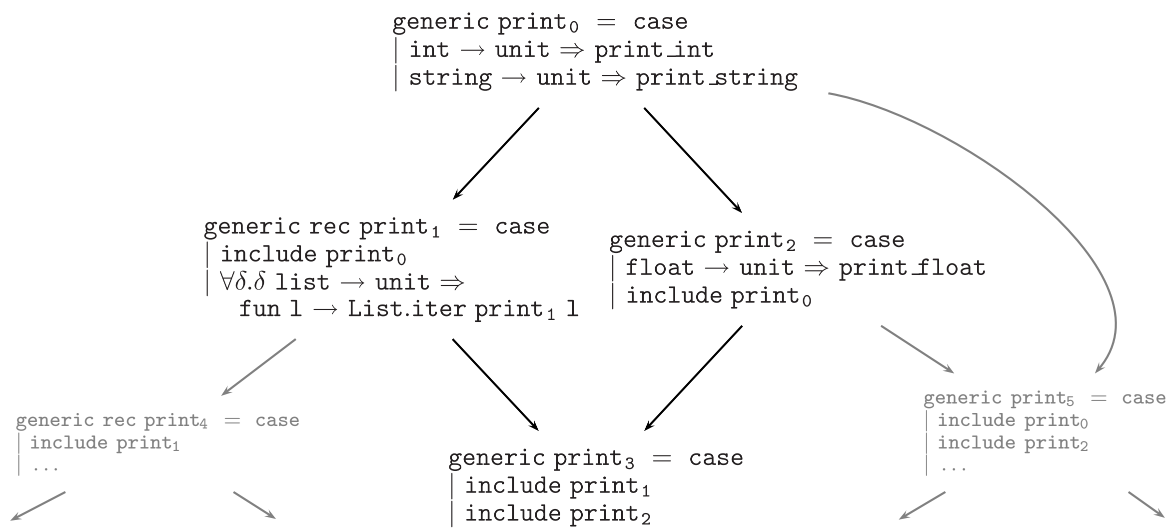


Figure 11.3: An extension graph of `print`

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

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

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

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

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

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

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

and its compilation must be optimized as below:

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

11.7 Conclusion

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

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

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

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

Open world extension in extensional polymorphism: future work ?

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

Chapter 12

Conclusion

We have developed extensional polymorphism, an ad-hoc polymorphism extension over the ML language. Generic values, ad-hoc polymorphic values in our framework, provide us various useful features, which were not possible in the classical ML parametric polymorphism: type dependent semantics of ad-hoc polymorphism, run time type tagged values such as dynamic values and for typed value I/O, and user definability of overloaded values. This extension of polymorphism reduces irritation of programmers a lot, who were captured inside the simple but restricted ML parametric polymorphism.

Core system

To achieve the type dependent semantics in the classical ML run time system without types, we proposed a compilation scheme which translates our full type-annotated source language called GML to an intermediate language called IML without type annotations. Type informations required for generic values are converted to normal ML values called run time types. These run time types are dispatched and delivered to generic values in the manner of type applications and abstractions of system F.

The run time type dispatching is useless for normal parametric polymorphic values, since their semantics are independent of types. Therefore, for efficiency, these unwanted type dispatching should be eliminated from the system F style translation with full type applications and abstractions. For this purpose, generic values must be distinguished from normal parametric polymorphic values in our type system: quantified type variables in type schemes for ad-hoc polymorphism have a special dynamic attribute. Thanks to this distinction, type dispatching is performed only for generalizations and instantiations of these dynamic type variables, therefore classical programs without generic values are compiled to themselves and are evaluated without any overhead.

The mechanism of run time type dispatching allows us to introduce ad-hoc polymorphic primitive values, called generic primitives. Their type-dependent semantics are only defined in the level of the intermediate language, as functions which take run time types indicating type instantiation of primitives. Having type instantiation information, generic primitives can perform type-dependent calculation, such as ML value printing for example. In the source language, derived generic values can be defined by users using let definitions with generic primitives or other derived generic values.

Applications of generic primitives – dynamic values and type safe value I/O

As an illustration of the power of extensional polymorphism framework, we implemented dynamic values. Dynamic values are an abstract data type, which encapsulates a value and its type. The primitives of encapsulation and decapsulation of values and their types can be implemented as generic primitives: for example, the encapsulation primitive `dynamic` is compiled to a simple function which takes a type and a value, then returns a tuple of them. Thus, compilation of dynamic value primitives are obtained almost for free in extensional polymorphism. However, special care must be taken for their typing. Captured inside dynamic values, type variables can be exported outside of their scope. Therefore encapsulation of type variables inside the type-value pair of a dynamic value must be restricted only to those generalizable and its content with type variables are considered to be parametric polymorphic. For this purpose, we added another poly-dynamic attribute to type variables related to dynamization and a special typing rule of `dynamic` primitive for explicit type generalization check of its argument: polymorphic dynamization, encapsulation of type variables into dynamic values is limited to the argument of `dynamic` primitive. We can still define derived dynamizers, derived generic values using `dynamic` primitive polymorphically inside their definitions, but they can only encapsulate values with closed types. We have also studied further typing discipline of derived dynamizers which reduces this limitation inside ML's first order polymorphism.

As another application example based on generic primitives, we proposed type safe ML value I/O system, which assures type safety of value communication between different ML programs. The same approach as dynamic values is taken again: when we output a value, it is automatically tagged with its type information using type dispatching mechanism of extensional polymorphism. When it is read back, encapsulated type information of the type-value pair is compared to the expected type in the context for the type safety. Unlike dynamic values, type-value pairs of value I/O can be exchanged between arbitrary different ML programs via files or streams. Therefore encapsulated type information must be universally meaningful. Even if data types of different programs share the same name, they may represent completely unrelated data structure. Hence, in type safe value I/O, data types are not identified by their names as in dynamic values, but by their semantics, which are given intuitively by their definitions. More specifically in Objective Caml, since names of data type definitions do not affect their internal value representation, we use nameless version of data type definitions as their identifications in type safe value I/O, which allows type safe value exchange between different data types with the same internal data structure. In the actual implementation, to avoid the size explosion of type information of type-value pairs for type safe value I/O, nameless type definitions are stored as encrypted checksums called data type digests.

Under the existence of value I/O, type information encapsulated inside dynamic value's type-value pair can also be exported outside of programs. In such a circumstance, data type identification at coercions of dynamic values cannot be done only by names either. At coercion of a data type, we also have to check its semantics are really compatible to the expected data type for type security. Therefore data type information encapsulated inside dynamic values must be tagged with their data type digests, too. The type-value pairs of dynamic values and type safe value I/O are extended to one unified format, external values: type information stored inside an external value contains both of data type names and their digests. The encapsulation primitives are also unified into the unique `dynamic` primitive, which creates this new type-value pair. On the other hand, we have two decapsulation primitives `import` and `coerce`, corresponding with two decapsulation schemes which come from both systems: `import` primitive only checks data type digests at decapsulation and allows value exchange between different data types but with the same internal representation. `coerce` primitive provides more strict decapsulation of external values: in addition to the digest check, it also compares data type names.

User definition of generic values

Based on the core system of extensional polymorphism, we proposed user-definability of generic values in the source language. The new `generic` binding defines a user-defined generic value, by overloading a fixed list of type cases, expressions indexed by their type schemes called type patterns. Type case selection in evaluation of a user-defined generic value instance is performed in the manner of ML pattern matching: its evaluation type context is matched against the type patterns, in their declaration order. If there are more than one matching type case, the first one is always selected. This type case selection scheme is achieved by pattern matching of dispatched run time types in the compilation of user-defined generic value definitions. Thanks to the selection priority embedded in the `generic` binding, we have almost no restrictions between type cases overloaded in one generic definition. For example, a user-defined generic value can have type cases whose type patterns overlap each other, which are not permitted in Haskell type classes where overloaded instances are defined separately and therefore there is no their clear selection priority. Though user generics are not open to extensions *in the sense of Haskell type classes*, this type case overloading without any restriction of `generic` binding provides us a great expressive power of overloading. Just by listing type cases, we can easily define generic values from simple overloaded arithmetic operators to complex recursive generic values such as functions with variable numbers of arguments, which are not definable, or are definable but difficult to use in the Haskell framework.

In our ML style typing system, the type scheme of a user-defined generic value is given as the least general anti-unifiers of its type patterns. But usually this type scheme is too general: a user-defined generic value can be non-exhaustive compared to its type scheme and its semantics may not be defined for some instance type. Normal type schemes with dynamic type variables do not have enough power to express non-exhaustiveness of user generics. To detect illegal use of user generics without semantics, we have built a new generic constraint typing system. For the theoretical cleanness and implementation modularity, this generic constraint typing forms an independent typing layer of the existing ML style normal typing layer. In the constraint typing system, generic values are assigned new type information called generic constraints. Intuitively a generic constraint of a user-defined generic value consists of the list of its type patterns, and constrains instantiation of the type scheme of the generic value which is assigned in the normal typing layer.

The generic constraint typing layer validates user-defined generic instances by simulating all the possible type case selections statically. A validated program is compiled to a code which perform the same type case selections by pattern matching of dispatched run time types at run time. This run time pattern matching of run time types is clearly redundant once the program is validated by the constraint typing. We can improve the efficiency of type case selections of user generics by dispatching some "witness" of their well-typedness obtained in the generic constraint typing, instead of run time types. This witness is an integer graph called a generic flow and obtained from the generic constraint type checking. Each node of a generic flow corresponds with a generic instance evaluation, and its integer index specifies which type case should be selected in the evaluation. There are two important benefits of using generic flows instead of run time type matching: the one is efficiency of type case selection. In the classical compilation with run time type matching, the time for type case selection was proportional to length and complexity of patterns. With generic flows, type case selection is done in constant time, no matter how many and complex the type patterns are. In addition, even in examples with the simplest type pattern, type case selection by an integer index of generic flow passing is 1.5 time faster than the selection by pattern matching. The other more important benefit of generic flows is separation of type case selection strategy from run time restrictions. By migrating actual work in the static typing phase, we can perform back-tracking of type case selections, which was practically impossible in the type case selection by run time type matching due to its complexity. Back-tracking of type case selection expands semantics and adds more valid instance types of user generics, preserving all of the semantics of generic instances which were valid in the type matching based system. In addition, under the back-tracking selection, we have a type substitution lemma of user generic instances, which is a very important property for the type reconstruction of user generics.

Generic flow dispatching can be considered as a generalization of dictionary passing of Haskell. While dictionaries consist of overloaded instances themselves to be evaluated, in the generic flows, a selected type case is represented as an integer index, which selects the type case on the fly. This indirection is required in extensional polymorphism, since user generics can be defined locally, and their type case definition are not always globally available, while class instances must be global in Haskell.

Like other systems for overloading, extensional polymorphism faces to type ambiguity problem of user generics, since they lack principal typing. We have proposed a type reconstruction algorithm for user generics with automatic ambiguity resolution. As the generic constraint typing forms an independent layer of the normal ML style typing, generic constraint type reconstruction algorithm is separated from the normal type reconstruction. Source programs without types are first typed in the normal type reconstruction, then the results are sent to the generic constraint type reconstruction algorithm, which instantiates too general types of user generic instances so that they can be valid instances in the constraint typing layer. Same as the type case selection mechanism, we make use of the definition order of type cases in the ambiguity resolution: if a generic instance has ambiguity: more than one type patterns which may match with its current type, the first match possibility is selected automatically and its type is instantiated so that it can really match with the selected type pattern. This ambiguity resolution by type case definition ordering allows us to control default typing of user generics easily by reordering type case definitions.

Finally, we have argued extensibility of user generics. A user-defined generic value was defined by `generic` binding, listing all the type case definitions once and for all. This goes against programming modularity, since in many cases, overloaded type cases are semantically independent each other and only they share is the name and the vague notion of their functionality, "plus" for example. Because of this independence of semantics of type cases, it is desirable that type case could be defined in different places. Our approach to the extension of overloaded values are different from Haskell type classes based on the open world assumption, where each class instance declaration extends overloaded values globally. Such open extension system does not match with the eagerly evaluated ML semantics, since under the open world assumption, the semantics of overloaded values are dynamic until link time. Our extension of user generics is closed: each extension defines a new value and therefore the semantics of generic values are static. A new type case inclusion syntax construct is introduced to `generic` binding. By "including" already existing generic values inside a new definition of user generics, the type cases of included generics can be easily imported into the newly defined generic value. Merging type cases of type cases of user generics, we can quickly define generic values with extended semantics. Thus, our closed extension is based on redefinition of values, however, it is not just so simple. It also supports open recursion: semantics of a recursive generics are open to extensions, so that merged type cases by case inclusion can call each other recursively. For the redirection of recursive calls in the open recursion, compilation of a type case has a new functional argument for its recursive call back point and the recursive occurrences of user generics are replaced by this functional abstraction. The semantics of a user-defined generic value is given as the fixed point of the set of type case compilation functions.

Bibliography

[ACPP91] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically-typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, 1991.

[ACPR92] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Didier Rémy. Dynamic typing in polymorphic languages. In *Proceedings of the 1992 workshop on ML and its applications*, 1992.

[BTCGS94] Val Breazu-Tannen, Thierry Coquand, Carl A. Gunter, and Andre Scedrov. Inheritance as implicit coercion. In C. A. Gunter and J. C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, pages 197–245. The MIT Press, Cambridge, MA, 1994.

[DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *9th symposium Principles of Programming Languages*, pages 207–212. ACM Press, 1982.

[dR] Daniel de Rauglaudre. IoXML: an OCaml syntax extension to parse and print one's OCaml values in XML format. Available at `http://cristal.inria.fr/~ddr/IoXML/`.

[DRW95] Catherine Dubois, François Rouaix, and Pierre Weis. Extensional polymorphism. In *Proceedings of the 22th ACM Conference on Principles of Programming Languages*, January 1995.

[Gil] Stephen Gilmore. Programming in Standard ML '97. Available at `http://www.dcs.ed.ac.uk/home/stg/NOTES/`.

[GM96] James Gosling and Henry McGilton. The Java language environment, a white paper. Available at `http://java.sun.com/docs/white/langenv/`, 1996.

[GTL90] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types*. Cambridge University Press, 1990.

[Hen93] Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, 1993.

[HJW92] Paul Hudak, Simon Peyton Jones, and Philip Wadler. Report on the programming language Haskell, version 1.2. Technical report, Yale University, 1992.

[JJM97] S. Jones, M. Jones, and E. Meijer. Type classes: an exploration of the design space, 1997.

[Jon97] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1997.

[Jon00] Mark P. Jones. Type classes with functional dependencies. *Lecture Notes in Computer Science*, 1782:230–, 2000.

[Jon94] Mark P. Jones. The implementation of the Gofer functional programming system. Technical Report YALEU/DCS/RR-1030, New Haven, Connecticut, USA, May 1994, May 94.

[JP] Mark P. Jones and John C. Peterson. Hugs 98: A functional programming system based on haskell 98 - user manual.

[Ler] Xavier Leroy. The Objective Caml system release 3.06. Available at `http://caml.inria.fr/ocaml/htmlman/index.html`.

[LM94] Xavier Leroy and Michel Mauny. Dynamics in ML. *Journal of Functional Programming*, 3(4):431–463, 1994.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[Myc84] Alan Mycroft. Polymorphic type schemes and recursive definitions. In *International Symposium on Programming*, number 167 in Lecture Notes in Computer Science, pages 217–228. Springer-Verlag, 1984.

[Oho89] Atsushi Ohori. A simple semantics for ml polymorphism. In *Proc ACM/IFIP Conference on Functional Programming Languages and Computer Architecture*, pages 281–292, 1989.

[Oho95] Atsushi Ohori. A polymorphic record calculus and its compilation. *ACM Transactions on Programming Languages and Systems*, 17(6):844–895, November 1995.

[OWW95] Martin Odersky, Philip Wadler, and Martin Wehr. A second look at overloading. In *Functional Programming Languages and Computer Architecture*, pages 135–146, 1995.

[PL99] Francois Pessaux and Xavier Leroy. Type-based analysis of uncaught exceptions. In *Symposium on Principles of Programming Languages*, pages 276–290, 1999.

[Riv92] R. Rivest. The MD5 message-digest algorithm, 1992.

[Rob65] J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[TA95] Andrew P. Tolmach and Andrew W. Appel. A debugger for Standard ML. *Journal of Functional Programming*, 5(2):155–200, 1995.

[W3C00] W3C. Extensible markup language (XML) 1.0. Available at `http://www.w3.org/TR/2000/REC-xml-20001006`, 2000.

[Wri95] Andrew K. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4):343–355, 1995.

Contents

Index