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)` $\rightarrow$ `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)` $\rightarrow$ `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:

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

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

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

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:

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

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:

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.

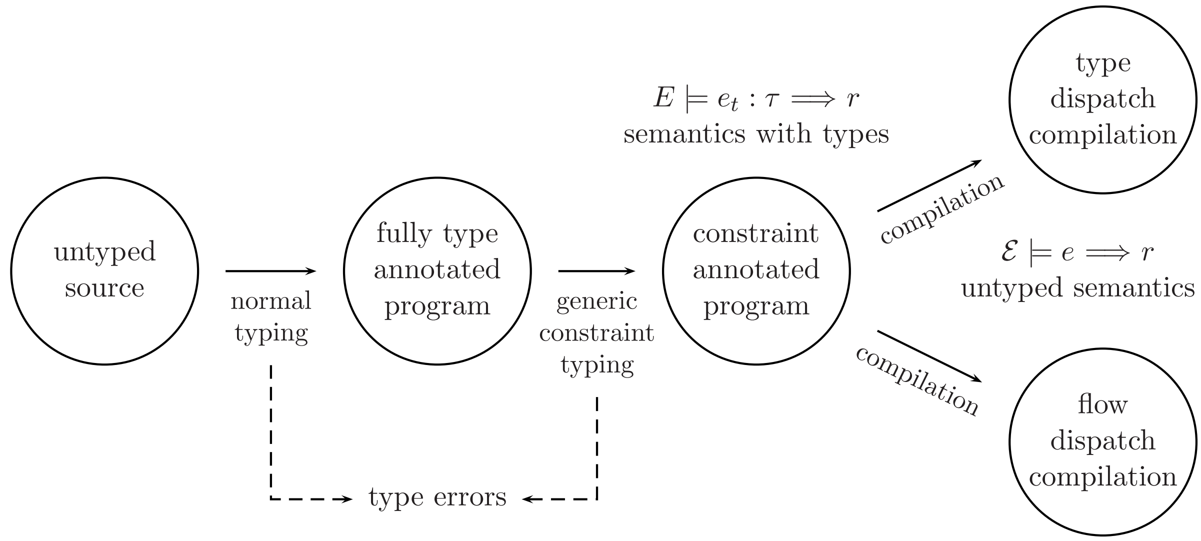

Figure 1.1: Typing and compilation phases

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.