Type systems for programming languages

Didier Rémy

Academic year 2020-2021
Version of October 20, 2020
## Contents

1 Introduction ................................................................. 7
   1.1 Overview of the course ............................................ 7
   1.2 Requirements ...................................................... 9
   1.3 About Functional Programming ................................. 9
   1.4 About Types ....................................................... 9
   1.5 Acknowledgment .................................................. 11

2 The untyped \(\lambda\)-calculus ......................................... 13
   2.1 Syntax ..................................................................... 13
   2.2 Semantics .................................................................. 15
      2.2.1 Strong v.s. weak reduction strategies .................. 15
      2.2.2 Call-by-value semantics ................................... 16
   2.3 Answers to exercises ............................................... 18

3 Simply-typed lambda-calculus .......................................... 21
   3.1 Syntax ..................................................................... 21
   3.2 Dynamic semantics ................................................ 21
   3.3 Type system .......................................................... 22
   3.4 Type soundness ...................................................... 25
      3.4.1 Proof of subject reduction ................................. 26
      3.4.2 Proof of progress ............................................. 28
   3.5 Simple extensions .................................................. 30
      3.5.1 Unit .............................................................. 30
      3.5.2 Boolean ........................................................ 30
      3.5.3 Pairs ............................................................ 31
      3.5.4 Sums ............................................................. 32
      3.5.5 Modularity of extensions .................................. 32
      3.5.6 Recursive functions ........................................ 33
      3.5.7 A derived construct: let-bindings ....................... 33
   3.6 Exceptions ........................................................... 35
      3.6.1 Semantics ....................................................... 35
CONTENTS

3.6.2 Typing rules .................................................. 36
3.6.3 Variations .................................................. 37
3.7 References .................................................... 39
  3.7.1 Language definition ....................................... 39
  3.7.2 Type soundness ............................................ 41
  3.7.3 Tracing effects with a monad ............................. 42
  3.7.4 Memory deallocation ..................................... 43
3.8 Omitted proofs and answers to exercises ....................... 44

4 Polymorphism and System F .................................. 49
  4.1 Polymorphism ................................................ 49
  4.2 Polymorphic λ-calculus ..................................... 51
    4.2.1 Types and typing rules ................................ 51
    4.2.2 Semantics ............................................. 52
    4.2.3 Extended System F with datatypes ..................... 54
  4.3 Type soundness ............................................. 58
  4.4 Type erasing semantics ..................................... 62
    4.4.1 Implicitly-typed System F ............................... 62
    4.4.2 Type instance ......................................... 64
    4.4.3 Type containment in System F .......................... 66
    4.4.4 A definition of principal typings ...................... 68
    4.4.5 Type soundness for implicitly-typed System F ......... 69
  4.5 References ................................................ 72
    4.5.1 A counter example .................................... 73
    4.5.2 Internalizing configurations ........................... 74
  4.6 Damas and Milner's type system .............................. 77
    4.6.1 Definition ........................................... 77
    4.6.2 Syntax-directed presentation ........................... 79
    4.6.3 Type soundness for ML ................................ 82
  4.7 Omitted proofs and answers to exercises ...................... 84

5 Type reconstruction ......................................... 91
  5.1 Introduction ............................................... 91
  5.2 Type inference for simply-typed λ-calculus .................. 92
    5.2.1 Constraints ......................................... 93
    5.2.2 A detailed example .................................... 94
    5.2.3 Soundness and completeness of type inference ......... 96
    5.2.4 Constraint solving ...................................... 96
  5.3 Type inference for ML ....................................... 98
    5.3.1 Milner’s Algorithm J ................................... 98
    5.3.2 Constraints ........................................... 99
5.3.3 Constraint solving by example ........................................ 103
5.3.4 Type reconstruction .................................................. 106
5.4 Type annotations .......................................................... 109
  5.4.1 Explicit binding of type variables ................................. 110
  5.4.2 Polymorphic recursion .............................................. 113
  5.4.3 mixed-prefix .......................................................... 114
5.5 Equi- and iso-recursive types ......................................... 115
  5.5.1 Equi-recursive types ................................................ 115
  5.5.2 Iso-recursive types ................................................ 117
  5.5.3 Algebraic data types ................................................ 118
5.6 HM(X) ........................................................................... 119
5.7 Type reconstruction in System F ....................................... 121
  5.7.1 Type inference based on Second-order unification ............ 121
  5.7.2 Bidirectional type inference ........................................ 122
  5.7.3 Partial type inference in MLF ...................................... 124
5.8 Proofs and Solution to Exercises ....................................... 124

6 Existential types .............................................................. 127
  6.1 Towards typed closure conversion .................................. 128
  6.2 Existential types .......................................................... 130
    6.2.1 Existential types in Church style (explicitly typed) ........ 130
    6.2.2 Implicitly-typed existential types ............................... 133
    6.2.3 Existential types in ML ............................................. 135
    6.2.4 Existential types in OCaml ........................................ 136
  6.3 Typed closure conversion ................................................. 137
    6.3.1 Environment-passing closure conversion ....................... 137
    6.3.2 Closure-passing closure conversion .............................. 139
    6.3.3 Mutually recursive functions .................................... 141

7 Overloading ................................................................. 145
  7.1 An overview .............................................................. 145
    7.1.1 Why use overloading? .............................................. 145
    7.1.2 Different forms of overloading ................................. 146
    7.1.3 Static overloading .................................................. 147
    7.1.4 Dynamic resolution with a type passing semantics ........ 147
    7.1.5 Dynamic overloading with a type erasing semantics ....... 148
  7.2 Mini Haskell .............................................................. 149
    7.2.1 Examples in MH ..................................................... 149
    7.2.2 The definition of Mini Haskell .................................. 150
    7.2.3 Semantics of Mini Haskell ...................................... 152
    7.2.4 Elaboration of expressions ...................................... 154
<table>
<thead>
<tr>
<th>Contents</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.2.5 Summary of the elaboration</td>
<td>155</td>
</tr>
<tr>
<td>7.2.6 Elaboration of dictionaries</td>
<td>157</td>
</tr>
<tr>
<td>7.3 Implicitly-typed terms</td>
<td>159</td>
</tr>
<tr>
<td>7.4 Variations</td>
<td>165</td>
</tr>
<tr>
<td>8 Logical Relations</td>
<td>171</td>
</tr>
<tr>
<td>8.1 Introduction</td>
<td>171</td>
</tr>
<tr>
<td>8.2 Normalization of simply-typed λ-calculus</td>
<td>171</td>
</tr>
<tr>
<td>8.3 Proofs and Solution to Exercises</td>
<td>173</td>
</tr>
</tbody>
</table>
Chapter 7

Overloading

7.1 An overview

Overloading occurs when several definitions of an identifier may be visible simultaneously at the same occurrence in a program. An interpretation of the program (and a fortiori a run of the program) must choose the definition that applies at this occurrence. This is called overloading resolution. Overloading resolution may use quite different strategies and techniques. All sorts of identifiers may be subject to overloading: variables, labels, constructors, types, etc.

Overloading must be distinguished from shadowing of identifiers by normal scoping rules, where in this case, a definition is just temporarily inaccessible by another one, but only the last definition is visible.

7.1.1 Why use overloading?

There are several reasons to use overloading.

Overloading may just be a naming convenience that allows reusing the same identifier for similar but different operations. This avoids name mangling such as suffixing similar names by type information: printing functions, e.g. `print_int`, `print_string`, etc.; numerical operations, e.g. `(+)`, `.+` etc.; or numerical constants e.g. `0`, `0.`, etc. In this respect, it may help with modularity. In the absence of overloading, the naming discipline (including name mangling conventions) must be known globally to avoid name clashes, which breaks compositionality. Isolated identifiers with no particular naming convention may still interfere between different developments and cannot be used together unless fully qualified. This problem does not disappear with overloading but it may be minimized—as long as overloading is not ambiguous. Hence, in some sense, overloading allows to think more abstractly, in terms of operations rather than of particular implementations. For instance, calling `to_string` conversion lets the system check whether one definition is available according to the type of the argument.
Overloaded definitions may also be used to provided type dependent functions. That is, a function may be defined for all types \( \tau[\alpha] \) but with an implementation depending on the type of \( \alpha \) by provided several overloaded definitions for different types \( \tau[\tau_i] \). For instance, a marshaling function of type \( \forall \alpha. \alpha \to \text{string} \) may execute different code for each base type \( \alpha \).

Overloaded definitions may be *ad hoc*, i.e. completely unrelated for each type—or just share a same type schema. For example 0 could mean either the integer zero or the empty list; and “\( \times \)” could mean either the integer product or string concatenation.

Conversely, overloaded definitions may depend solely on the *type structure* (i.e. on whether the argument is a sum, a product, *etc.* ) so that definitions can be derived mechanically for all types from their definitions on base types. Such overloaded functions are called polytypic functions. Typical examples are marshaling functions, or the generation of random values for arbitrary types as used in the [Quickcheck](https://github.com/Quickcheck-NL) tool for Haskell, *etc.* Still, polytypic definition often need to be specialize at some particular types. For example, one may use a polytypical definition of printing, so that printing is available at all types, but define specialized versions of printing at some particular types.

### 7.1.2 Different forms of overloading

There are many variants of overloading. They can be classified by how overloading is *introduced* and *resolved*.

The first elements of classification are the restrictions on overloading definitions. Can arbitrary definitions be overloaded? For instance, can numerical values be overloaded? Are all overloaded definitions of the same symbol instances of a common type scheme? Are these type schemes arbitrary? Are overloaded definitions primitive (pre-existing), automatic (generated mechanically from other definitions), or user-defined? Can overloaded definitions overlap? Can overloaded definitions have a local scope?

However, the main element of classification remains the resolution strategy—which may indirectly constraint the way overloading is introduced. We distinguish between *static* and *dynamic* resolutions strategies.

Static resolution of overloading has a very simple semantics since the meaning of the program can be determined statically by deciding for each overloaded symbol which actual definition of the symbol should be used. Hence, it replaces each occurrence of an overloaded symbol by an actual implementation at the appropriate type. Therefore static overloading does not increase expressiveness per say, since the user could have chosen the appropriate implementation in the first place. Still, static overloading may significantly reduce verbosity—and increase modularity and abstraction, as explained above.

Conversely, dynamic resolution increases expressiveness, as the choice of the implementation may now depend on the dynamic of the program execution. However, it is also much more involved, since the semantics of the language usually need extra machinery to support the dynamic resolution. For example, the resolution of some occurrence of a polymorphic
function may depend on the type of its arguments, so that different calls of the function at different types can make different choices. The resolution is driven by information made available at runtime: it could at worse require full type information. In some restrictions, partial type information may be sufficient, and sometimes some type-related information can be used instead of types themselves, such as tags, dictionaries, etc. These can be attached to values (as tags in object oriented languages), or passed as extra arguments at runtime (as dictionaries in Haskell).

7.1.3 Static overloading

The language SML has a very limited form of overloading where overloaded definitions are primitive: they include an exhaustive list of overloaded definitions for numerical operators, plus automatically generated overloaded definitions for all record accessors. The resolution is static and fails if overloading cannot be unambiguously resolved at outermost let-definitions. For example, \texttt{let twice \(x = x + x\)} is rejected in SML at toplevel, since \(+\) could be either the addition on either integers or floats.

In the language Java, overloading is not primitive but automatically generated by subtyping: when a class extends another one and a method is redefined, the older definition is still visible, but at another type, hence the method is overloaded. This overloading is then statically resolved by choosing the most specific definition. There is always a best choice—according to static knowledge. This static resolution of overloading in Java comes in complement to the dynamic dispatch of method calls. This is often a source of confusion for programmers who often expect a dynamic resolution of overloading and as a result misunderstand the semantics of their programs. For instance, an argument may have a runtime type that is a subtype of the best known compile-time type, and perhaps a more specific definition could have been used if overloading were resolved dynamically.

However convenient, static resolution of overloading is quite limited. Moreover, it does not fit very well with first-class functions and polymorphism. Indeed, with static overloading, \(\lambda x. x + x\) is rejected when \(+\) is overloaded, as it cannot be resolved. The function must be manually specialized at some type for which \(+\) is defined. This argues in favor of some form of dynamic overloading that allows to delay resolution of overloaded symbols at least until polymorphic functions have been sufficiently specialized.

7.1.4 Dynamic resolution with a type passing semantics

The most ambitious approach to dynamic overloading is to pass types at runtime and dispatch on the runtime type, using a general typecase construct.

Runtime type dispatch is the most general approach as it does not impose much restriction on the introduction of overloaded definitions It uses an explicitly-typed calculus (\textit{e.g.} System F)—with a type passing semantics—extended with a typecase construct. However,
the runtime cost of typecase may be high, unless type patterns are significantly restricted. Moreover, one pays even when overloading is not used, since types are always passed around, even when overloading is not used, unless the compiler uses aggressive program analyzes to detect these situations and optimize type computations away. Monomorphization may also be used to allow more static resolution in such cases. Ensuring exhaustiveness of type matching is often a difficult task in this context.

The ML& calculus by Castagna (1997) offers a general overloading mechanism based on type dispatch. It is an extension of System F with intersection types, subtyping, and type matching. An expressive type system keeps track of exhaustiveness; type matching functions are first-class and can be extended or overridden. The language allows overlapping definitions with a best match resolution strategy.

### 7.1.5 Dynamic overloading with a type erasing semantics

To avoid the expensive cost of typecase, one may restrict the overloaded definitions, so that full type information is not needed and only an approximation of types, such as tags, may be used for overloading resolution. This is one possible approach to object-orientation in the method as overloading functions paradigm where object classes are used to dynamically select the appropriate method. This is also an approach used in some scheme dialects known as generics.

In fact, one may get more freedom by detaching tags from values and passing tags—or almost equivalently passing the actually implementations grouped into dictionaries—as extra runtime arguments. A side advantage of this approach is that the semantics can be described without changing the runtime environment, i.e. the representation of values, as an elaboration process that introduces abstractions and applications for implementations of overloaded symbols. Schematically, one transforms unresolved overloaded symbols into extra abstractions and passes actual implementations (or abstractions of implementations) around as extra arguments. Hopefully, overloaded symbols can be resolved when their types are sufficiently specialized and before they are actually needed.

For example, a program context `let f = λx. x + x in []` can be elaborated into `let f = λ(+). λx. x + x in []`. If `f 1.0` is placed in the hole of this original program context, it can then be elaborated to `f (+) 1.0`, which can be placed in the hole of the elaborated program context. Elaboration can be performed after typechecking by translating the typing derivation. After elaboration, types are no longer needed and can be erased. Monomorphization or other simplifications may reduce the number of abstractions and applications introduced by overloading resolution.

This technique has been widely explored—under different facets—in the context of ML: Type classes, introduced very early by Wadler and Blott (1989) are still the most popular and widely used framework. Other contemporary solutions have been proposed by Rouaix (1990) and Kaes (1992). Simplifications of type classes have also been proposed by Odersky et al.
but did not take over, because of their restrictions. Recent works on type classes is still going on Morris and Jones (2010).

In the rest of this chapter we introduce a tiny language called Mini Haskell that models the essence of Haskell type classes; at the end we also discuss implicit arguments as a less structured but simpler way of introducing dynamic overloading in a programming language.

### 7.2 Mini Haskell

Mini Haskell—or MH for short—is a simplification of Haskell to avoid most of the difficulties of type classes but keeping their essence: it is restricted to single parameter type classes and no overlapping instance definitions; it is close in expressiveness and simplicity to A second look at overloading by Odersky et al. but closer to Haskell in style—it can be easily generalized by lifting restrictions without changing the framework.

The language MH is explicitly typed. In this section, we first present some examples in MH, and then describe the language and its elaboration into System F. We introduce an implicitly-typed version of MH and its elaboration in the next section.

#### 7.2.1 Examples in MH

An equality class and several instances many be defined in Mini Haskell as follows:

```haskell
class Eq (X) { equal : X → X → Bool }
inst Eq (Int) { equal = primEqInt }
inst Eq (Char) { equal = primEqChar }
inst Λ(X) Eq (X) ⇒ Eq (List (X))
{ equal = λ(l1 : List X) λ(l2 : List X) match l1, l2 with
  | [], [] → true | [], _ | _, [] → false
  | h1::t1, h2::t2 → equal X h1 h2 && equal (List X) t1 t2 }
```

This code declares a class (dictionary) of type Eq(X) that contains definitions for equal : X → X → Bool and creates two concrete instances (dictionaries) of type Eq(Int) and Eq(Char), and a function that, given a dictionary for Eq(X), builds a dictionary for type List(X). This code can be elaborated by explicitly building dictionaries as records of functions:

```haskell
type Eq (X) = { equal : X → X → Bool }
let equal X (EqX : Eq X) : X → X → Bool = EqX.equal

let EqInt : Eq Int = { equal = ( primEqInt : Int → Int → Bool ) }
let EqChar : Eq Char = { equal = primEqChar }

let EqList X (EqX : Eq X) : Eq (List X) =
{ equal = λ(l1 : List X) λ(l2 : List X) match l1, l2 with
  | [], [] → true | [], _ | _, [] → false
  | h1::t1, h2::t2 →
```
equal $X$ $\text{Eq}X$ $h_1$ $h_2$ && equal ($\text{List} X$) ($\text{EqList} X \text{Eq}X$) $t_1$ $t_2$

Classes may themselves depend on other classes (called superclasses), which realizes a form of class inheritance.

```haskell
class Eq ($X$) \Rightarrow Ord ($X$) { let : $X \rightarrow X \rightarrow \text{Bool} }
inst Ord (Int) { let = (<) }
```

The class definition declares a new class (dictionary) $\text{Ord} (X)$ that contains a method $\text{Ord}(X)$ that depends on a dictionary $\text{Eq}(X)$ and contains a method $\text{lt} : X \rightarrow X \rightarrow \text{Bool}$. The instance definition builds a dictionary $\text{Ord}(Int)$ from the existing dictionary $\text{Eq Int}$ and the primitive ($<$) for $\text{lt}$. The two declarations are elaborated into:

```haskell
type Ord ($X$) = { Eq : $\text{Eq} (X)$; let : $X \rightarrow X \rightarrow \text{Bool} }$
let EqOrd $X$ ($\text{OrdX} : \text{Ord} X$) : $\text{Eq} X = \text{OrdX.Eq}$
let EqOrd $X$ ($\text{OrdX} : \text{Ord} X$) : $X \rightarrow X \rightarrow \text{Bool} = \text{OrdX.lt}$
let OrdInt : Ord Int = { Eq = EqInt; let = (<) }
```

So far, we have just defined type classes and some instances. We may write a function that uses these overloaded definitions. When overloading cannot be resolved statically, the function will be abstracted other one or several additional arguments, called dictionaries, that will carry the appropriate definitions for the unresolved overloaded symbols. For example, consider the following definition in Mini Haskell:

```haskell
let rec search : $\forall$ ($X$) $\text{Ord} X \Rightarrow X \rightarrow \text{List} X \rightarrow \text{Bool} =$
$\Lambda (X) \lambda (x : X) \lambda (l : \text{List} X)$
match $l$ with $\text{[]} \rightarrow \text{false}$ $\mid h :: t \rightarrow \text{equal} X h$ $\mid |$$ search X x t$
```

This code is elaborated into:

```haskell
let rec search $X$ ($\text{OrdX} : \text{Ord} X$) ($x : X$) ($l : \text{List} X$) : $\text{Bool} =$
match $l$ with $\text{[]} \rightarrow \text{false}$ $\mid h :: t \rightarrow \text{equal} X (\text{EqOrd} X \text{OrdX}) x h$ $\mid |$$ search X \text{OrdX} x t$
```

Using the overloading function, as in $\text{search Int} 1 [1; 2; 3]$ will then elaborate into the code $\text{search Int OrdInt} 1 [1; 2; 3]$ where a dictionary $\text{OrdInt}$ of the appropriate type has been built and passed as an additional argument. Here, the target language is the explicitly-typed System $F$, which has a type erasing semantics, hence the type argument $\text{Int}$ may be dropped while the dictionary argument $\text{OrdInt}$ is retained: the code that is actually executed is thus $\text{search OrdInt} 1 [1; 2; 3]$ (where type information has been stripped off $\text{OrdInt}$ itself).

### 7.2.2 The definition of Mini Haskell

Class declarations and instance definitions are restricted to the toplevel. Their scope is the whole program. In practice, a program $p$ is a sequence of class declarations and instance and function definitions given in any order and ending with an expression. For simplification,
we assume that instance definitions do not depend on function definitions, which may then come last as part of the expression in a recursive let-binding.

Instance definitions are interpreted recursively and their definition order does not matter. We may assume, w.l.o.g., that instance definitions come after all class declarations. The order of class declaration matters, since they may only refer to other class constructors that have been previously defined.

For sake of simplification, we restrict to single parameter classes. The syntax of MH programs is defined in Figure 7.1. Letter $p$ ranges over source programs. A program $p$ is a sequence $H_1 \ldots H_p h_1 \ldots h_q M$, of class declaration $H_1 \ldots H_p$, followed by a sequence of instance definitions $h_1 \ldots h_q$, and ending with an expression $M$.

A class declaration $H$ is of the form $\text{class } \vec{P} \Rightarrow K \alpha \{ \rho \}$. It defines a new class (constructor) $K$, parametrized by $\alpha$. Every class (constructor) $K$ must be defined by one and only one class declaration. So we may say that $H$ is the declaration of $K$ and write $H \triangleleft K$.

Letter $u$ ranges over overloaded symbols, also called methods. The row $\rho$ of the form $u_1 : \tau_1, \ldots, u_n : \tau_n$ declares overloaded symbols $u_i$ of class $K$. An overloaded symbol cannot be declared twice in a program; it cannot be repeated twice in the same class (hence the map $i \mapsto u_i$ is injective) and cannot be declared in two different classes. The row $\rho$ (and thus each of its field type $\tau_i$) must not contain any other free variable than $\alpha$.

The class depends on a sequence of subclasses $\vec{P}$ of the form $K_1 \alpha, \ldots, K_n \alpha$, which is called a typing context. Each clause $K_i \alpha$ can be read as an assumption “given an instance of class $K_i$ at type $\alpha$” and $\vec{P}$ as the conjunction of these assumptions. We say that classes $K_i$’s are superclasses of $K$ which we write $K_i \triangleleft K$. They must have been previously defined. This ensures that the relation $\triangleleft$ is acyclic. We require that all $K_i$’s are independent, i.e. there do not exists $i$ and $j$ such that $K_j \triangleleft K_i$.

An instance definition $h$ is of the form $\text{inst } \forall \vec{\beta}. \vec{P} \Rightarrow K (\vec{G} \vec{\beta}) \{ r \}$. It defines an instance of a class $K$ at type $\vec{G} \vec{\beta}$ where $G$ is a datatype constructor, i.e. neither an arrow type nor a class constructor. A class constructor $K$ may appear in $Q$ but not in $\tau$. An instance definition defines the methods of a class at the required type: $r$ is a record of methods
$u_1 = M_1, \ldots u_n = M_n$.

An instance definition is also parametrized by a typing context $\vec{P}$ of the form $K_1 \alpha_1, \ldots K_k \alpha_k$ where variables $\alpha_i$’s are included in $\vec{\beta}$. This typing context is is not related to the typing context of its class declaration $H_K$, but to the set of classes that the implementations of the methods depend on.

**Restrictions** The restriction to types of the form $K' \alpha'$ in typing contexts and class declarations, and to types of the form $K' (G' \vec{\alpha}')$ in instances are for simplicity. Generalization are possible and discussed later ($\S 7.4$).

### 7.2.3 Semantics of Mini Haskell

The semantics of Mini Haskell is given by elaborating source programs into System $F$ extended with record types and recursive definitions. Record types are provided as data types. They are used to represent dictionaries. Record labels are used to encode overloaded identifiers $u$. We may use overloaded symbols as variables as well: this amounts to reserving a subset of variables $x_u$ indexed by overloaded symbols and writing $u$ as a shortcut for $x_u$. We use letter $N$ instead of $M$ for elaborated terms, to distinguish them from source terms. For convenience, we write $\Rightarrow$ in System $F$ as an alias for $\rightarrow$, which we use when the argument is a (record representing a) dictionary. Type schemes in the target language take the form $\sigma$ described on Figure 7.1. Notice that types $T$ are stratified: they are either dictionary types $K \tau$ or a regular type $\tau$ that does not contain dictionary types.

**Class declaration** The elaboration of a class declaration $H_K$ of the form class $K_1 \alpha, \ldots K_n \alpha \Rightarrow K \alpha \{ \rho \}$ consists of several parts. It first declares a record type that will be used as a dictionary to carry both the methods and the dictionaries of its immediate superclasses. A class need not contain subdictionaries recursively, since if $K_j \prec K_i$, then a dictionary for $K_i$ already contains a sub-dictionary for $K_j$, to which $K$ has access via $K_i$ so it does need not have one itself. The row $\rho$ of the class definition only lists the class methods. Hence, we extend it with fields for sub-dictionaries and define the record type:

$$K \alpha \approx \{ \rho^K \}$$

where $\rho^K$ is $u_{K_1}^K : K_1 \alpha, \ldots u_{K_n}^K : K_n \alpha, \rho$.

This record type declaration is collected to appear in the program *prelude*.

Then, for each $u : T_u$ in $\rho^K$, we define the program context:

$$\mathcal{R}_u \triangleq \text{let } u : \sigma_u = N_u \text{ in } []$$

where $\sigma_u \triangleq \forall \alpha. K \alpha \Rightarrow T_u$ and $N_u \triangleq \Lambda \alpha. \lambda z : K \alpha. (z.u)$

Let the composition $\mathcal{R}_1 \circ \mathcal{R}_2$ of two contexts be the context $\mathcal{R}_1[\mathcal{R}_2]$ obtained by placing $\mathcal{R}_2$ in the hole of $\mathcal{R}_1$. The elaboration $[H_K]$ of a single class declaration $H_K$ is the composition:

$$[H_K] \triangleq \mathcal{R}_{u_1} \circ \ldots \mathcal{R}_{u_n}$$

where $K \alpha \approx \{ u_1 : T_1, \ldots u_n : T_n \}$
that defines accessors for each field of the class dictionary. We also define the typing environment \( \Gamma_H \) as an abbreviation for \( u_1 : \sigma_{u_1}, \ldots, u_n : \sigma_{u_n} \).

The elaboration \( [H_1 \ldots H_p] \) of all class definitions is the composition \( [H_1] \circ \ldots \circ [H_p] \) of the elaboration of each. We also define \( \Gamma_{H_1 \ldots H_n} \) as the concatenation \( \Gamma_{H_1} \ldots \Gamma_{H_n} \) of individual typing environments.

**Instance definition** In an instance declaration \( h \) of the form \( \text{inst } \forall \beta. \vec{P} \Rightarrow K (G \, \vec{\beta}) \{ r \} \), The typing context \( \vec{P} \) describes the dictionaries that must be available on type parameters \( \vec{\beta} \) for constructing the dictionary \( K (G \, \vec{\beta}) \), but that cannot yet be built because they depend on some unknown type \( \beta \) in \( \vec{\beta} \).

As mentioned above \( \vec{P} \) is not related to the typing context of the class declaration \( H_K \). To see this, assume that class \( K' \) is an immediate superclass of \( K \), so that the creation of the dictionary \( K : \alpha \) requires the existence of a dictionary \( K' : \alpha \); then, an instance declaration \( K : \Gamma \) (where \( \Gamma \) is nullary) need not be parametrized over a dictionary of type \( K' : \Gamma \), as either such a dictionary can already be built, hence the instance definition does not require it, or it will never be possible to build one, as instance definitions are recursively defined so all of them are already visible—and the program must be rejected.

We restrict typing context \( K_1 : \alpha_1, \ldots, K_k : \alpha_k \) to canonical ones defined as satisfying the two following conditions: (1) \( \alpha_i \) is some \( \beta_j \) in \( \vec{\beta} \); and (2) if \( K_i \) and \( K_j \) are related, i.e. \( K_i < K_j \) or \( K_j < K_i \) or \( K_i = K_j \). then \( \alpha_i \) and \( \alpha_j \) are different. The latter condition avoids having two dictionaries \( K_i : \beta \) and \( K_j : \beta \) when, e.g., \( K_i < K_j \) since the former is contained in the latter.

The elaboration of an instance declaration \( h \) is a triple \((z_h, N^h, \sigma_h)\) where \( z_h \) is an identifier to refer to the elaborated body \( N^h \) of type 

\[
\sigma_h \triangleq \forall \beta_1, \ldots, \beta_p. K_1 : \alpha_1 \Rightarrow \cdots K_k : \alpha_k \Rightarrow K (G \, \vec{\beta})
\]

(Variables \( \alpha_1, \ldots, \alpha_k \) are among \( \beta_1, \ldots, \beta_p \) and may contain repetitions, as explained above.)

The expression \( N^h \) builds a dictionary of type \( K (G \, \vec{\beta}) \), given \( k \) dictionaries (where \( k \) may be zero) of respective types \( K_1 : \beta_1, \ldots, K_k : \beta_k \) and is defined as:

\[
N^h \triangleq \Lambda \beta_1, \ldots, \beta_p. \lambda(z_1 : K_1 : \alpha_1) \ldots \lambda(z_k : K_k : \alpha_k).
\{ u_{K_1}^h = q_1, \ldots, u_{K_k}^h = q_n, \ u_1 = N_1^h, \ldots, u_m = N_m^h \}
\]

The types of fields are as prescribed by the class definition \( K \), but specialized at type \( G \, \vec{\beta} \). That is, \( q_i \) is a dictionary expression of type \( K'_i (G \, \vec{\beta}) \) whose exact definition is postponed until the elaboration of dictionaries in §7.2.6. The term \( N_i^h \) is the elaboration of \( M_i \) where \( u_1 = M_1, \ldots, u_m = M_m \) is \( r \); it is described in the next section (§7.2.4). For clarity, we write \( z \) instead of \( x \) when a variable binds a dictionary or a function building a dictionary. Notice that the expressions \( q_i \) and \( N_i^h \) sees the type variables \( \beta_1, \ldots, \beta_p \) and the dictionary parameters \( z_1 : K_1 : \alpha_1, \ldots, z_k : K_k : \alpha_k \).
The elaboration of all instance definitions is the program context:

\[
[\vec{h}] \triangleq \text{let rec } (\vec{z}_h : \vec{\sigma}_h) = \vec{N}_h \text{ in } []
\]

that recursively binds all instance definitions in the hole.

**Program** Finally, the elaboration of a complete program \(\vec{H} \vec{h} M\) is

\[
[\vec{H} \vec{h} M] \triangleq ([\vec{H}] \circ [\vec{h}])[M] = \text{let } \vec{u} : \vec{\sigma}_u = \vec{N}_u \text{ in } \text{let rec } (\vec{z}_h : \vec{\sigma}_h) = \vec{N}_h \text{ in } \vec{N}
\]

Hence, the expression \(N\), which is the elaboration of \(M\), and all expressions \(N_h\) are typed (and elaborated) in the environment \(\Gamma_{\vec{H} \vec{h}}\) equal to \(\Gamma_{\vec{H}}, \Gamma_{\vec{h}}\): the environment \(\Gamma_{\vec{H}}\) declares functions to access components of dictionaries (both sub-dictionaries and definitions of overloaded symbols) while the environment \(\Gamma_{\vec{h}}\), declares functions to build dictionaries.

### 7.2.4 Elaboration of expressions

The elaboration of expressions is defined by a judgment \(\Gamma \vdash M \leadsto N : \sigma\) where \(\Gamma\) is a System F typing context, \(M\) is the source expression, \(N\) is the elaborated expression and \(\sigma\) its type in \(\Gamma\). In particular, \(\Gamma \vdash M \leadsto N : \sigma\) implies \(\Gamma \vdash N : \sigma\) in System F.

We write \(q\) for dictionary terms, which are the following subset of System-F terms:

\[
q ::= u \mid z \mid q \tau \mid q q
\]

Variables \(u\) and \(z\) are just particular cases of variables \(x\). Variable \(u\) is used for methods (and access to subdictionaries), while variable \(z\) is used for dictionary parameters and for class instances, i.e. dictionaries or functions building dictionaries.

The rules for elaboration of expressions are described in Figure 7.2. Most of them just wrap the elaboration of their sub-expressions. In rule \textsf{LET}, we require \(\sigma\) to be canonical, i.e. of the form \(\forall \alpha. \vec{P} \Rightarrow T\) where \(\vec{P}\) is itself empty or canonical (see page 153). Rules \textsf{APP} and \textsf{ABS} do not apply to overloaded expressions of type \(\sigma\) but only to simple expressions of type \(\tau\).

The interesting rules are the elaboration of overloaded expressions, and in particular of missing abstractions (Rule \textsf{OABS}) and applications (Rule \textsf{OAPP}) of dictionaries. Rule \textsf{OABS} pushes dictionary abstractions in the context \(\Gamma\) as prescribed by the expected type. On the opposite, Rule \textsf{OAPP} searches for an appropriate dictionary-building function and applies it to the required sub-directionary.

The premise \(\Gamma \vdash q : Q\) of rule \textsf{OAPP} also triggers the elaboration of dictionaries. This judgment is just the typability in System F—but restricted to dictionary expressions. That is, it searches for a well-typed dictionary expression. The restriction to dictionary expressions ensures that under reasonable conditions the search is decidable—and coherent. The elaboration of dictionaries reads the typing rules of System F restricted to dictionaries as an algorithm, where \(\Gamma\) and \(Q\) are given and \(q\) is inferred. This is described in detail in §7.2.6.
By construction, elaboration produces well-typed expressions: that is $\Gamma \vdash M \leadsto N : \tau$ implies that is $\bar{H} h \vdash M \leadsto N : \tau$.

### 7.2.5 Summary of the elaboration

An instance declaration $h$ of the form:

\[
\text{inst } \forall \tilde{\alpha}. K_1 \alpha_1, \ldots K_k \alpha_k \Rightarrow K \tilde{\tau} \{ u_1 = M_1, \ldots; u_m = M_m \}
\]

is translated into

\[
\lambda(z_1:K_1\alpha_1) \ldots \lambda(z_p:K_k\alpha_k). \{ u_{K_1}^{K_1} = q_1, \ldots; u_{K_k}^{K_k} = q_n; u_1 = N_1, \ldots; u_m = N_m \}
\]

where $u_{K_i}^{K_j} : \tau_i$ are the superclasses fields, $\Gamma^h$ is $\tilde{\beta}, K_1 \alpha_1, \ldots K_k \alpha_k$, and the following elaboration judgments $\bar{H} h, \Gamma^h \vdash q_i : \tau_i$ and $\bar{H} h, \Gamma^h \vdash M_i \leadsto N_i : \tau_i$ hold. Finally, given the program $p$ equal to $\bar{H} h M$, we elaborate $M$ as $N$ such that $\bar{H} h \vdash M \leadsto N : \forall \tilde{\alpha}. \tau$.

Notice that $\forall \tilde{\alpha}. \tau$ is an unconstrained type scheme. Otherwise, $N$ could elaborate into an abstraction over dictionaries, which could turn a computation into a function that is not reduced: this would not preserve the intended semantics.

More generally, we must be careful to preserve the intended semantics of source programs. For this reason, in a call-by-value setting, we must not elaborate applications into abstractions, since this could delay and perhaps duplicate the order of evaluations. We just pick the obvious solution, that is to restrict rule $\text{LET}$ so that either $\sigma$ is of the form $\forall \tilde{\alpha}. \tau$ or $M_1$ is a value or a variable.
In a language with a call-by-name semantics, an application is not evaluated until it is needed. Hence adding an abstraction in front of an application should not change the evaluation order $M_1 M_2$. We must in fact compare:
\[
\text{let } x_1 = \lambda y. \text{let } x_2 = V_1 V_2 \text{ in } M_2 \text{ in } [x_1 \mapsto x_1 q]M_1 \tag{1}
\]
\[
\text{let } x_1 = \text{let } x_2 = \lambda y. V_1 V_2 \text{ in } [x_2 \mapsto x_2 q]M_2 \text{ in } M_1 \tag{2}
\]
The order of evaluation of $V_1 V_2$ is preserved. However, the Haskell language is call-by-need and not call-by-name! Hence, applications are delayed as in call-by-name but shared and only reduced once. The application $V_1 V_2$ will be reduced once in (1), but as many times as there are occurrences of $x_2$ in $M_2$ in (2).

The final result will still be the same in both cases if the language has no side effects, but the intended semantics may be changed regarding the complexity.

**Coherence** The elaboration may fail for several reasons: The input expression may not obey one of the restrictions we have requested; a typing may occur during elaboration of an expression; or or some dictionary cannot be build. If elaboration fails, the program $p$ is rejected, of course.

When the elaboration of $p$ succeeds, it should return a term $\llbracket p \rrbracket$ that is well-typed in $F$ and that defines the semantics of $p$. However, although terms are explicitly-typed, their elaboration may not be unique! Indeed, they might be several ways to build dictionaries of some given type, as we shall see below (§7.2.6).

We may distinguish two situations: in the worst case, a source program may elaborate to several completely unrelated programs; in the better case, all possible elaborations may in fact be equivalent programs: we say that the elaboration is coherent and the programs has a deterministic semantics given by any of its elaboration.

Opening a parenthesis, what does it mean for programs be equivalent? There are several notions of program equivalence:

- If programs have a denotational semantics, the equivalence of programs should be the equality of their denotations.

- As a subcase, two programs having a common reduct should definitely be equivalent. However, this will in general not be complete: values may contain functions that are not identical, but perhaps reduce to the same value whenever applied to equivalent arguments.

- This leads to the notion of observational equivalence. Two expressions are observationally equivalent (at some observable type, such as integers) if their are indistinguishable whenever they are put in arbitrary (well-typed) contexts of the observable type.

End of parenthesis.
7.2. MINI HASKELL

For instance, two different elaboration algorithms that consistently change the representation of dictionaries (e.g. by ordering records in reverse order), may be equivalent if we cannot observe the representation of dictionaries.

Returning to the coherence problem, the only source of non-determinism in Mini Haskell is the elaboration of dictionaries. Hence, to ensure coherence, it suffices that two dictionary values of the same type are always equal. This does not mean that there is a unique way of building dictionaries, but that all ways are equivalent as they eventually return the same dictionary.

7.2.6 Elaboration of dictionaries

The elaboration of dictionaries is based on typing rules of System F—but restricted to a subset of the language. The relevant typing rules are given in Figure 7.3. However, elaboration significantly differs from type inference since the judgment $\Gamma \vdash q : Q$ is used for inferring $q$ rather than $\tau$. The judgment can be read as: in type environment $\Gamma$, a dictionary of type $Q$ can be constructed by the dictionary expression $q$. As for type inference, elaboration of dictionaries is simplified by finding an appropriate syntax-directed presentation of the typing rules—but directed by the structure of the type of the expected dictionary instead of expressions.

Elaboration is also driven by the bindings available in the typing environment. These may be dictionary constructors $z^h$, given by instance definitions; dictionary accessors $u^K$, given by class declarations; dictionary arguments $z$, given by the local typing context. This suggests the presentation of the typing rules in Figure 7.4.

**Dictionary values** Let us first consider the elaboration of dictionary values. They are typed in the environment $\Gamma_{\vec{h}}$, which does not contain free type variables. Hence, rule $[\text{D-VAR}]$ does not apply. Moreover, dictionaries stored in other dictionaries had to be built in the first place, hence rule $[\text{D-PRO}]$ should never be needed. That is, dictionary values can be built with only instances of $[\text{D-OVAR-INST}]$ of the form:

\[
\frac{\text{D-OVAR-INST}}{\Gamma_{\vec{h}} \vdash z : \forall \vec{\beta}. P_1 \Rightarrow \ldots \Rightarrow P_n \Rightarrow K (\vec{G} \vec{\beta}) \in \Gamma_{\vec{h}} \quad \Gamma_{\vec{h}} \vdash q_i : [\vec{\beta} \mapsto \vec{\tau}] P_i}{\Gamma_{\vec{h}} \vdash z \vec{\tau} \vec{q} : K (\vec{G} \vec{\tau})}
\]
where the premises \( \Gamma \vdash q_i : [\beta \mapsto \tau]P_i \) are themselves recursively built in the same way. This rule can be read as a recursive definition, where \( \Gamma \) is constant, \( Q \) is the input type of the dictionary, and \( q \) is the output dictionary. This reading is deterministic if there is no choice in finding \( z : \forall \beta. P_1 \Rightarrow \ldots P_n \Rightarrow K(G(\beta)) \in \Gamma \). The binding \( z \) can only be a binding \( z_h \) introduced as the elaboration of some class instance \( h \) at type \( \Gamma \beta \). Hence, it suffices that \textit{instance definitions never overlap} for \( z_h \) to be uniquely determined; if recursively each \( q_i \) is unique, then \( z \bar{\tau} q \) also is. Under this hypothesis, the elaboration is always unique and therefore coherent.

\textbf{Definition 3 (Overlapping instances)} Two instances \( \text{inst } \forall \beta_1. P \Rightarrow K(G_1(\beta_1)) \{r_1\} \) and \( \text{inst } \forall \beta_2. P \Rightarrow K(G_2(\beta_2)) \{r_2\} \) of a class \( K \) overlap if the type schemes \( \forall \beta_1. K(G_1(\bar{\tau}_1)) \) and \( \forall \beta_2. K(G_2(\bar{\tau}_2)) \) have a common instance, i.e. in the current setting, if \( G_1 \) and \( G_2 \) are equal.

Overlapping instances are an inherent source of incoherence, as it means that for some type \( Q \) (in the common instance), a dictionary of type \( Q \) may (possibly) be built using two different implementations.

**Dictionary expressions** Dictionary expressions may compute on dictionaries: they may extract sub-dictionaries or build new dictionaries from other dictionaries received as argument. Indeed, in overloaded code, the exact type is not fully known at compile type, hence dictionaries must be passed as arguments, from which superclass dictionaries may be extracted (actually must be extracted, as we forbade to pass a class and one of its super class dictionaries simultaneously).

Dictionaries are typically typed in the typing environment \( \Gamma_{hh}, \Gamma^h \) where \( \Gamma^h \) binds the local typing context, \textit{i.e.} assumptions \( z : K' \beta \) about dictionaries received as arguments. Hence, rules \textbf{D-Proj} and \textbf{D-Var} may now apply, \textit{i.e.} the elaboration of expressions uses the three rules of 7.4. This can still be read as a backtracking proof search algorithm. The proof search always terminates, since premises always have strictly smaller \( Q \) than the conclusion when using the lexicographic ordering of the height of \( \tau \) and then the reverse order of class inheritance: when no rule applies, the search fails; when rule \textbf{D-Var} applies, the search ends
with a successful derivation; when rule \texttt{D-Proj} applies, the premise is called with a smaller problem since the height is unchanged and \( K' \triangleright \tau \) with \( K' < K \); when \texttt{D-Ovar-Inst} applies, the premises are called at type \( K_i \tau_j \) where \( \tau_j \) is subtype of \( \tau \), hence of a strictly smaller height.

**Non determinism** However, non-overlapping of class instances is no more sufficient to prevent from non determinism. For instance, the introductory example of §7.2.1 defines two instances \texttt{EqInt} and \texttt{OrdInt} where the later contains an instance of the former. Hence, a dictionary of type \texttt{EqInt} may be obtained, either directly as \texttt{EqInt}, or indirectly as \texttt{Eq OrdInt}, by projecting the \texttt{Eq} sub-dictionary of class \texttt{Ord Int}. In fact, the latter choice could then be reduced at compile time and be equivalent to the first one.

One could force more determinism by fixing a strategy for elaboration. Restrict the use of rule \texttt{D-Proj} to cases where \( Q \) is \( P \) when \texttt{D-Ovar-Inst} does not apply. However, since the two elaborations paths are equivalent, the extra flexibility is harmless and may perhaps be useful freedom for the compiler.

**Example of elaboration** In our introductory example, the typing environment \( \Gamma_{\text{Hh}} \) is (we remind both the informal and formal names of variables):

\[
\begin{align*}
\text{equal} & \triangleq \ u_{\text{equal}} : \forall \alpha. \text{Eq} \ \alpha \Rightarrow \alpha \to \alpha \to \text{bool}, \\
\text{EqInt} & \triangleq \ z_{\text{Eq}} : \text{Eq int} \\
\text{EqList} & \triangleq \ z_{\text{List}} : \forall \alpha. \text{Eq} \ \alpha \Rightarrow \text{Eq} (\text{list} \ \alpha) \\
\text{EqOrd} & \triangleq \ u_{\text{Eq}} : \forall \alpha. \text{Ord} \ \alpha \Rightarrow \text{Eq} \ \alpha \\
\text{lt} & \triangleq \ u_{\text{lt}} : \forall \alpha. \text{Ord} \ \alpha \Rightarrow \alpha \to \alpha \to \text{bool}
\end{align*}
\]

When elaborating the body of the \texttt{search} function, we have to infer a dictionary for \texttt{EqOrd X OrdX} in the local context \( X, \text{OrdX} : \text{Ord} \ X \). Using formal notations, dictionaries are typed in the environment \( \Gamma \) equal to \( \Gamma_0, \alpha, z : \text{Ord} \ \alpha \) and \texttt{EqOrd} is \( u_{\text{Eq}}^{\text{Ord}} \). We have the following derivation:

\[
\frac{
\text{D-Ovar-Inst} \quad \text{D-Proj} \quad \text{D-var}}{
\Gamma \vdash z : u_{\text{Eq}}^{\text{Ord}} \alpha : \text{Ord} \ \alpha \Rightarrow \text{Eq} \ \alpha \quad \Gamma \vdash z : \text{Ord} \ \alpha \\
\Gamma \vdash u_{\text{Eq}}^{\text{Ord}} \alpha \ z : \text{Eq} \ \alpha}
\]

### 7.3 Implicitly-typed terms

Our presentation of Mini Haskell is explicitly typed. Since we remain within an ML-like type system where type schemes are not first-class, we may leave some type information implicit. But how much? Class declarations define both the structure of dictionaries—a record type definition and its accessors—and the type scheme of overloaded symbols. Since, we inferring type schemes is out of the scope of ML-like type inference, class declarations
must remain explicit. Instance definitions are turned into recursive polymorphic definitions, which in ML require type scheme annotations. So they instance definitions also remain explicit. Fortunately, all remaining core language expressions, i.e. the body of instance definitions and the final program expression can be left implicit.

For instance, the example program in the introduction can be rewritten more concisely.

```haskell
class Eq (X) { equal : X → X → Bool }
inst Eq (Int) { equal = primEqInt }
inst Eq (Char) { equal = primEqChar }
inst Λ(X) Eq (X) ⇒ Eq (List (X)) { Eq = \(\lambda (l_1) (l_2) \cdot \text{match } l_1, l_2 \text{ with}
\] 
  | [], [] → true | [], _ | _, _ → false 
  | h1::t1, h2::t2 → Eq h1 h2 && Eq t1 t2 }
class Eq (X) ⇒ Ord (X) { lt : X → X → Bool }
inst Ord (Int) { lt = (<) }
let rec search x l = match l with [] → false | h::t → equal x h || search x t
let b = search Int 1 [1; 2; 3];;
```

The missing type information can be rebuilt by type inference.

**Type inference** To perform type inference in Mini Haskell, the idea is to see dictionary types \(K\ τ\), which can only appear in type schemes and not in types, as a type constraint to mean “there exists a dictionary of type \(K\ α\)”. That is, we may read the type scheme

\[\forall α. K[\bar{P}]. τ\]

as the constraint type scheme

\[∀ α, K[\bar{P}] τ\]

On ground types, a constraint \(K\ t\) is satisfied if one can build a dictionary of type \(K\ t\) in the initial environment \(Γ\⃗H\⃗h\) (that contains all class and instance declarations)—formally, if there exists a dictionary expression \(q\) such that \(Γ\⃗H\⃗h\vdash q : K\ t\). Then satisfiability of class-membership constraints is (with its unfolded version on the right):

\[\frac{\text{INSTANCE}}{K\ φτ} \quad \frac{\text{INSTANCE}}{Γ\⃗H\⃗h\vdash ρ : K\ φτ} \quad \frac{\text{INSTANCE}}{ϕ \vdash K\ τ}\]

We use entailment to reason with class-membership constraints. For every class declaration

class \(K_1\ α, \ldots, K_n\ α \Rightarrow K\ α\{ρ\}\), we have:

\[K\ α \vdash K_1\ α \land \ldots \land K_n\ α\]

(K1)

This rule allows to decompose any set of simple constraints into a canonical one.

**Proof**: Assume \(ϕ \vdash K\ α\), i.e. by Rule [INSTANCE] \(Γ\⃗H\⃗h\vdash q : K\ (φα)\) for some dictionary \(q\). From the class declaration in \(Γ\⃗H\⃗h\), we know that \(K\ α\) is a record type definition that contains
fields $u_i^K$ of type $K_i \alpha_i$. Hence, the dictionary value $q_i$ contains field values of types $K_i (\phi_i)$. Therefore, we have $\phi \vdash K_i \alpha_i$ for all $i$ in $1..n$, which implies $\phi \vdash K_1 \alpha \land \ldots \land K_n \alpha$.

For every instance definition $\text{inst} \forall \vec{\beta}. K_1 \beta_1, \ldots, K_p \beta_p \Rightarrow K (G \vec{\beta}) \{r\}$, we have

$$K (G \vec{\beta}) \equiv K_1 \beta_1 \land \ldots \land K_p \beta_p \quad (K2)$$

This rule allows to decompose any class constraint into a conjunction of simple constraints (i.e. of the form $K \alpha$).

|| Proof: Let $h$ be the above instance definition. We prove both directions separately:

Case $\vdash$: Assume $\phi \vdash K_i \beta_i$ for $i$ in $\{1, \ldots, p\}$. By Rule [INSTANCE] for each $i$, there exists a dictionary $q_i$ such that $\Gamma_{\vec{H}_h} \vdash q_i : K_i (\phi_i)$. Hence, $\Gamma_{\vec{H}_h} \vdash x_h \beta q_1 \ldots q_p : K (G (\phi \vec{3}))$, i.e. by Rule [INSTANCE] $\phi \vdash K (G \vec{3})$.

Case $\vdash$: Assume, $\phi \vdash K (G \vec{3})$. i.e. there exists a dictionary $q$ such that $\Gamma_{\vec{H}_h} \vdash q : K (G (\phi \vec{3}))$. By inversion of typing (and non-overlapping of instance declarations), the only way to build such a dictionary is by an application of $x_h \beta q_1 \ldots q_p$ with $\Gamma_{\vec{H}_h} \vdash q_i : K_i (\phi_i)$. By Rule [INSTANCE] this means $\phi \vdash K_i \beta_i$ for every $i$, which implies $\phi \vdash K_1 \beta_1 \land \ldots \land K_p \beta_p$.

Notice that the equivalence (K2) still holds in an open-world assumption where new instance clauses may be added later, because another future instance definition cannot overlap with existing ones.

If class instances may overlap, the $\vdash$ direction does not hold anymore; the rewriting rule:

$$K (G \vec{\beta}) \rightarrow K_1 \beta_1 \land \ldots \land K_p \beta_p$$

remains sound (the inverse entailment holds, and thus type inference still infer sound typings), but it is incomplete (type inference could miss some typings).

We also use the following equivalence: for every class $K$ and type constructor $G$ for which there is no instance of $K$:

$$K (G \vec{\beta}) \equiv \text{false} \quad (K3)$$

This rule allows to report failure as soon as a constraint of the form $K (G \vec{\tau})$ for which there is not instance of $K$ for $G$ appears.

|| Proof: The $\vdash$ direction is a tautology, so it suffices to prove the $\rightarrow$ direction. By contradiction. Assume $\phi \vdash K (G \vec{3})$. This implies the existence of a dictionary $q$ such that $\Gamma_{\vec{H}_h} \vdash q : K (G (\phi \vec{3}))$. Then, there must be some $x_h$ in $\Gamma$ whose type scheme is of the form $\forall \vec{\beta}. P \Rightarrow K (G \vec{3})$, i.e. there must be an instance of class $K$ for $G$. |
Notice that the equivalence is only an inverse entailment in an open world assumption: when there is not instance of \( K \) at type \( G \), the rewriting rule \( K (G \beta) \mapsto \text{false} \) remains sound, but it is incomplete.

We are now fully equipped for type inference. Constraint generation is unchanged: see Figure 5.6. A constraint type scheme can then always be decomposed into one of the form \( \forall \alpha [P_1 \land P_2]. \tau \) where \( \text{ftv}(P_1) \not\subseteq \alpha \) and \( \text{ftv}(P_2) \not\subseteq \alpha \). The constraints \( P_2 \) can then be extruded to the enclosing context if any, so that we are just left with \( P_1 \), and thus a well-formed type scheme \( \forall \alpha. \bar{P} \Rightarrow \tau \) with a typing context \( \bar{P} \).

To check well-typedness of a program \( \bar{H} \bar{h} a \), we must check that: each expression \( a^h \) and the expression \( a \) are well-typed, in the environment used to elaborate them. This amounts to checking:

- \( \Gamma_{\bar{H}h}, \Gamma \vdash a^h : \tau^h \) where \( \tau^h \) is given. That is, that \( \text{def} \, \Gamma_{\bar{H}h}, \Gamma \in \{a^h\} \preceq \tau^h \equiv \text{true} \) holds;
- \( \Gamma_{\bar{H}h} \vdash a : \tau \) for some \( \tau \). That is, that \( \text{def} \, \Gamma_{\bar{H}h} \in \exists \alpha. \{a\} \preceq \alpha \equiv \text{true} \) holds.

However, typechecking is not sufficient: type reconstruction should also return an explicitly-typed term \( M \) than can in turn be elaborated into some term \( N \) of System \( F \), i.e. such that \( \Gamma \vdash a \leadsto M : \tau \).

**Type reconstruction** Type reconstruction can be performed as described in §5.3.4 by keeping persistent constraints during resolution. As in ML, there may be several ways to reconstruct programs, which we may solve by requesting explicitly-typed terms to be canonical and principal.

**Coherence** When the source language is implicitly-typed, the elaboration from the source language into System \( F \) code is the composition of type reconstruction with elaboration of explicitly typed terms.

Hence, even though the elaboration is coherent for explicitly-typed terms, this may not be true for implicitly-typed terms. There are two potential problems:

- The language has principal constrained type schemes, but the elaboration requests unconstrained type schemes.
- Ambiguities could be hidden (and missed) by non principal type reconstructions.

**Toplevel unresolved constraints** The restrictions we put on class declarations and instance definitions ensure that the type system has principal constrained schemes (and principal typing reconstructions).

However, this does not imply that there are principal *unconstrained* type schemes. For example, assume that the principal constrained type scheme is \( \forall \alpha [K \alpha]. \alpha \rightarrow \alpha \) and the typing environment contains two instances of \( K G_1 \) and \( K G_2 \) of class \( K \).
instances of this type scheme are \(G_1 \rightarrow G_1\) and \(G_2 \rightarrow G_2\) but \(\forall \alpha. \alpha \rightarrow \alpha\) is certainly not one. Not only neither choice is principal, but worse, the two choices would elaborate in expressions with different (and non-equivalent) semantics. Elaboration should fail in such cases.

This problem may appear while typechecking the final expression \(a\) in \(\Gamma_{\tilde{H}h}\) that request an unconstrained type scheme \(\forall \alpha. \tau\). It may also occur when typechecking the body of an instance definition \(h\), which requests an explicit type scheme \(\forall \beta[Q]. \tau\) in \(\Gamma_{\tilde{H}h}\) or, equivalently, a type \(\tau\) in \(\Gamma_{\tilde{H}h}; \beta, Q\). Consider, for example:

```plaintext
class Num (X) { 0 : X, (+) : X \rightarrow X \rightarrow X }
inst Num Int { 0 = Int.(0), (+) = Int.(+) }
inst Num Float { 0 = Float.(0), (+) = Float.(+) }
let zero = 0 + 0;
```

The type of \(\text{zero}\) or \(\text{zero} + \text{zero}\) is \(\forall \alpha[\text{Num} \alpha]. \alpha\) while several class instances are possible for \(\text{Num} X\). The semantics of the program is thus undetermined. Another example is:

```plaintext
class Readable (X) { read : descr \rightarrow X }
inst Readable (Int) { read = read_int }
inst Readable (Char) { read = read_char }
let v = read (open_in())
```

The type of \(v\) is \(\forall \alpha[\text{Readable} \alpha]. \text{unit} \rightarrow \alpha\) — and several classes are possible for \(\text{Readable} \alpha\). This program is also rejected.

**Inaccessible constraint variables** In the previous examples, the incoherence arise from the obligation to infer unconstrained toplevel type schemes. A similar problem may occur with isolated constraints in a type scheme. For instance, assume that \(\text{let } x = a_1 \text{ in } a_2\) elaborates to \(\text{let } x : \forall \alpha[K \alpha]. \text{int} \rightarrow \text{int} = N_1\) in \(N_2\). All applications of \(x\) in \(N_2\) will lead to an unresolved constraint \(K \alpha\) for some fresh \(\alpha\) since neither the argument nor the context of this application can determine the value of the type parameter \(\alpha\). Still, a dictionary of type \(K \tau\) must be given before \(N_1\) can be executed.

Although \(x\) may not be used in \(N_2\), in which case, all elaborations of the expression may be coherent, we may still raise an error, since an unusable local definition is certainly useless, hence probably a programmer’s mistake. The error may then be raised immediately, at the definition site, instead of at every use of \(x\).

**The open-world view** When there is a single instance \(K G\) for a class \(K\) that appears in an unresolved or isolated constraint \(K \alpha\), the problem formally disappears, as all possible type reconstructions are coherent.

However, we may still not accept this situation, for modularity reasons, as an extension of the program with another non-overlapping correct instance declaration would make the program become ambiguous.
Formally, this amounts to saying that the program must be coherent in its current form, but also in all possible extensions with well-typed class definitions. This is taking an *open-world* view.

**On the importance of principal type reconstruction** A source of incoherence is when some class constraint remains undetermined. Some (usually arbitrary) less general elaboration could cover the problem—but the source program would remain incoherent. Hence, in order to detect programs with ambiguous semantics, it is essential that type reconstruction is principal. A program can still be specialized but only after it has been proved coherent. This freedom may actually be very useful for optimizations. Consider for example, the program

```plaintext
let twice = \(x\) \(x + x\) in twice \((\text{twice} 1)\)
```

whose principal type reconstruction is:

```plaintext
let twice : \(\forall X\) \([\text{Num} X]\) \(X \rightarrow X = \Lambda (X) [\text{Num} X] \lambda x. x + x\) in twice \(\text{Int}\) \((\text{twice} \text{Int}) 1\)
```

This program is coherent. It’s natural elaboration is

```plaintext
let twice \(X\) \(\text{Num} X\) = \(\lambda x: X\) \((\text{plus} \text{Num} X) x\) in twice \(\text{Int}\) \(\text{Num} \text{Int}\) \((\text{twice} \text{Int} \text{Num} \text{Int}) 1\)
```

However, it can also be elaborated to

```plaintext
let twice = \(\lambda x: \text{Int}\) \(x \text{ (plus} \text{Num} \text{Int}) x\) in twice \((\text{twice} 1)\)
```

avoiding the generalization of twice; moreover, the overloaded application *plus NumInt* can now be statically reduced, leading to:

```plaintext
let twice = \(\lambda x: \text{Int}\) \(x \text{ Int.(+)} x\) in twice \((\text{twice} 1)\)
```

**Overloading by return types** All previous ambiguous examples are overloaded by their return types: For instance, in \(0 : X\), the value 0 has an overloaded type that is not constraint by the argument; in \(\text{read} : \text{descr} \rightarrow X\), the return type is under specified, independently of the type of the argument.

To avoid such cases, [Odersky et al.](#) has suggested to prevent overloading by return types by requesting that overloaded symbols of a class \(K \alpha\) have types of the form \(\alpha \rightarrow \tau\). The above examples would then be rejected by this definition.

In fact, disallowing overloading by return types—in addition to our previous restrictions—suffices to ensure that all well-typed programs are coherent. Moreover, untyped programs can then be given a direct semantics (which of course coincides with the semantics obtained by elaboration). Many interesting examples of overloading actually fits in this restricted subset. However, overloading by returns types is also found useful in several cases and [Haskell](#) allows it, using default rules to resolve ambiguities. This is still an arguable design choice in the [Haskell](#) community.
7.4 Variations

Changing the representation of dictionaries  An overloaded method call $u$ of a class $K$ is elaborated into an application $u \ q$ of $u$ to a dictionary expression $q$ of class $K$. The function $u$ and the representation of the dictionary are both defined in the elaboration of the class $K$ and need not be known at the call site. This leaves some flexibility in the representation of dictionaries. For example, we have used records to represent dictionaries, but tuples would have been sufficient.

Going one step further, dictionaries need not contain the methods themselves but enough information from which the methods may be recovered. For example, dictionaries may be replaced by a derivation tree that proves the existence of the dictionary. This derivation tree may be concisely represented and passed around instead of the dictionary itself and be used and interpreted at the call site to dispatch to the appropriate implementation of the method. Such an approach has been followed by Furuse (2003b).

This change of representation can also elegantly be explained as a type preserving compilation of dictionaries called concretization and described in Pottier and Gauthier (2006). It is somehow similar to defunctionalization and also requires that the target language is equipped with GADT (Guarded Abstract Data Types).

Multi-parameter type classes  To allow multi-parameter type classes, we may extend the syntax of class definitions as follows:

\[
\text{class } \vec{P} \Rightarrow K \vec{\alpha} \{ \rho \}
\]

where free variables of $\vec{P}$ must be bound in $\vec{\alpha}$. The current framework can easily be extended to handle multi-parameter type classes. For example, Collections may be represented by a type $C$ whose elements are of type $E$ and defined as follows:

```plaintext
class Collection C E { find : C \rightarrow E \rightarrow Option(E), add : C \rightarrow E \rightarrow C }
inst Collection (List X) X { find = List.find, add = \lambda(c)\lambda(e) e::c }
inst Collection (Set X) X { ... }
```

However, the class `Collection` does not provide the intended intuition that collections are homogeneous. Indeed, we may define:

```plaintext
let add2 c x y = add (add c x) y
add2 : \forall(C, E, E') Collection C E, Collection C E' \Rightarrow C \rightarrow E \rightarrow E' \rightarrow C
```

This is accepted assuming that collections are heterogeneous. Although, this is unlikely the case, no contradiction can be assumed. However, if collections are indeed homogeneous, no instance of heterogeneous collections will ever be provided and the above code is overly general. As a result, uses of collections have unresolved often parameters, which would be resolved, if we had a way to tell the system that collections are homogeneous.

The solution is to add a clause to say that the parameter $C$ determines the parameter $E$:
class Collection $C \times E \mid C \to E \{ \ldots \}$

Then, because $C$ determines $E$, the two instances $E$ and $E'$ must be equal in $C$. Type dependencies also reduce overlapping between class declarations, since fewer instances of a class make sense. Hence they also allow example that would have to be rejected if type dependencies could not be expressed.

**Associated types** Associated types are an alternative to functional dependencies. They allow a class to declare its own type functions. Correspondingly, instance definitions must provide a definition for all associated types—in addition to values for overloaded symbols.

For example, the `Collection` class becomes a single parameter class with an associated type definition:

```haskell
class Collection $E \{ 
    type $C : \ast$
    find : $C \to E \to Option E$
    add : $C \to E \to C$
\}
```

```haskell
inst Collection Eq $X \Rightarrow Collection X \{ \text{type } C = List E, \ldots \}$
inst Collection Eq $X \Rightarrow Collection X \{ \text{type } C = Set E, \ldots \}$
```

Associated types increase the expressiveness of type classes.

**Overlapping instances** In practice, overlapping instances may be desired! This seems in contradiction with the fact that overlapping instances are a source of incoherence. For example, one could provide a generic implementation of sets provided an ordering relation on elements, but also provide a more efficient version for bit sets. When overlapping instances are allowed, further rules are needed to disambiguate the overloading resolution and preserve coherence. For instance, priority rules may be used. An interesting resolution strategy is to give priority to the most specific match.

However, the semantics depend on some particular resolution strategy and becomes more fragile. See [Jones et al. (1997)] for a discussion. See also [Morris and Jones (2011)] for a recent new proposal. For example, the definitions:

```haskell
inst Eq(X) \{ equal = (=) \}
inst Eq(Int) \{ equal = primEqInt \}
```

could elaborate into the creation of both a generic dictionary and a specialized one.

```haskell
let Eq X : Eq X= \{ equal = (=) \}
let EqInt : Eq Int = \{ equal = primEqInt \}
```

Then, `EqInt` or `Eq Int` are two dictionaries of type `Eq Int` but with different implementations.
7.4. Variations

Restriction that are harder to lift  We have made several restrictions to the definition of type classes. Some can be lifted at the price of some tolerable complication. Relaxing other restrictions, even if it could make sense in theory, would raise serious difficulties in practice.

For example, allowing constrained type schemes of the form $K\,\tau$ instead of the restricted form $K\,\alpha$ would affect many aspects of the language and it would becomes much more difficult to control the termination of constrained resolution and of the elaboration of dictionaries.

Allowing class instances of the form $\text{inst}\,\forall\vec{\beta}.\,\vec{P} \Rightarrow K\,\tau\{\rho\}$ where $\tau$ is $G\,\vec{\tau}$ and not just $G\,\vec{\beta}$, it would become difficult to check non-overlapping of class instances.

Alternative to type classes

Implicit values

Implicit values are a mechanism that allows to build values from types. The implies a way to populate an environment of definitions that can be used to build implicit values and a mechanism to introduce place holders where values should be build from their types.

Implicit values have been used in the language Scala for implicit conversions (but they can do more). An extension of OCaml with implicit values is being prototyped. Implicit values have also been proposed as an alternative to Haskell type classes (Oliveira et al. 2012) and more recently formalized in COCHIS, a calculus of implicit values (Schrijvers et al. 2017).

Module-based type classes

Modular type classes (Dreyer et al. 2007) mimic type classes at the level of modules, but with explicit module abstractions and module applications.

Modular implicits extends this idea by allowing implicit module arguments. Module arguments are thus inferred, but module abstractions remain explicit, which interestingly, allows for local scoping of overloading. They also extend the module language to increase expressiveness.

Conclusions

Methods as overloading functions  One approach to object-orientation is to see methods as overloaded functions. Then, objects carry class tags that can be used at runtime to find the best matching definition. This approach has been studied in detail by Millstein and Chambers (1999). See also Bonniot (2002, 2003).
Summary  Static overloading is not a solution for polymorphic languages. Dynamics overloading must be used instead. The implementation of type classes in the Haskell language has proved quite effective: it is a practical, general, and powerful solution to dynamic overloading. Moreover, it works relatively well in combination with ML-like type inference.

However, besides the simplest case of overloading on which everyone agrees, some useful extensions often come with serious drawbacks, and there is not yet an agreement on the best design compromises. In Haskell, the design decisions have often been in favor of expressiveness, but then losing some of the properties and the canonicity of the minimalistic initial design.

Dynamic overloading is a typical and very elegant use of elaboration. The programmer could in principle write the elaborated program manually, explicitly building and passing dictionaries around, but this would be cumbersome, tricky, error prone, and it would significantly obfuscate the code. Instead, the elaboration mechanism does this automatically, without arbitrary choices (in the minimal design) and with only local transformations that preserve the structure of the source program.

Further reading  For an all-in-one explanation of Haskell-like overloading, see The essence of Haskell by Odersky et al. See also the Jones’s monograph Qualified types: theory and practice. For a calculus of overloading see the ML& calculus proposed by Castagna (1997).

Recently, type classes have also been added to Coq Sozeau and Oury (2008). Interestingly, the elaboration of proof terms need not be coherent which makes it a simpler situation for overloading.
A tour of scala: Implicit parameters. Part of scala documentation.


