Type systems for programming languages

Didier Rémy

December 20, 2012

Contents

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.

Overloading 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 α by provided several overloaded definitions for different types $\tau[\tau_i]$. For instance, a marshaling function of type $\forall\alpha.\,\alpha \to \mathsf{string}$ may execute different code for each base type α.

Overloading 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 "×" 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 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 increrases 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 machinary 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, `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 (*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 detech these sitiations 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 resoled when their types are sufficiently specialized and before they are actually needed.

For example, a program context $\mathsf{let}\ f = \lambda x.\, x + x\ \mathsf{in}\ [\,]$ can be elaborated into $\mathsf{let}\ f = \lambda(+).\, \lambda x.\, x + x\ \mathsf{in}\ [\,]$. 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.

(1995) 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:

```
class Eq (X)   { equal : X → X → Bool }
inst  Eq (Int)  { equal = (==) }
inst  Eq (Char) { equal = (==) }
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 : $X \to X \to X$ 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:

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

let EqInt : Eq Int  = { equal = ( (==) : 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 →
```

$$\text{equal } X \text{ EqX } h_1\ h_2 \text{ \&\& equal } (\mathit{List}\ X)\ (\text{EqList } X \text{ EqX})\ t_1\ t_2\ \}$$

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

```
class Eq (X) ⇒ Ord (X) { lt : X → X → Bool }
inst Ord (Int) { lt = (<) }
```

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

```
type Ord (X) = { Eq : Eq (X); lt : X → X → Bool }
let EqOrd X (OrdX : Ord X) : Eq X = OrdX.Eq
let lt X (OrdX : Ord X) : X → X → Bool = OrdX.lt
let OrdInt : Ord Int = { Eq = EqInt; lt = (<) }
```

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 dictionnaries, that will carry the appropriate definitions for the unresolved overloaded symbols. For example, consider the following definition in Mini Haskell:

```
let rec search : ∀(X) Ord X ⇒ X → List X → Bool =
  Λ(X) λ(x : X) λ(l : List X)
    match l with [] → false | h::t → equal x h || search X x t
```

This code is elaborated into:

```
let rec search X (OrdX : Ord X) (x : X) (l : List X) : Bool =
  match l with [] → false
    | h::t → equal X (EqOrd X OrdX) x h || search X OrdX x t
```

Using the overloading function, as in $\mathit{search}\ \mathit{Int}\ 1\ [1; 2; 3]$ will then elaborate into the code $\mathit{search}\ \mathit{Int}$ OrdInt $1\ [1; 2; 3]$ where a dictionary 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 Int may be dropped while the dictionary argument OrdInt is retained: the code that is actually executed is thus search OrdInt $1\ [1; 2; 3]$ (where type information has been stripped off 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,

$$
\begin{array}{rcl}
p & ::= & H_1 \ldots H_p \, h_1 \ldots h_q \, M \\
H & ::= & \mathsf{class}\ \vec{P} \Rightarrow \mathsf{K}\ \alpha\ \{\rho\} \\
\rho & ::= & u_1 : \tau_1, \ldots u_n : \tau_n \\
h & ::= & \mathsf{inst}\ \forall \vec{\beta}.\ \vec{P} \Rightarrow \mathsf{K}\ (\mathsf{G}\ \vec{\beta})\ \{r\} \\
r & ::= & u_1 : M_1, \ldots u_n : M_n
\end{array}
\qquad
\begin{array}{rcl}
P & ::= & \mathsf{K}\ \alpha \\
\vec{P} & ::= & P_1, \ldots P_n \\
Q & ::= & \mathsf{K}\ \tau \\
\vec{Q} & ::= & Q_1, \ldots Q_n \\
\sigma & ::= & \forall \vec{\alpha}.\, \vec{Q} \Rightarrow T \\
T & ::= & \tau \mid Q
\end{array}
$$

Figure 7.1: Syntax of MH expressions and types

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 $\mathsf{class}\ \vec{P} \Rightarrow \mathsf{K}\ \alpha\ \{\rho\}$. It defines a new class (constructor) K, parametrized by α. 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_{K}.

Letter u ranges over *overloaded symbols*, also called *methods*. The row ρ 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 ρ (and thus each of its field type τ_i) must not contain any other free variable than α.

The class depends on a sequence of subclasses $\vec{P}$ of the form $\mathsf{K}_1\ \alpha, \ldots \mathsf{K}_n\ \alpha$, which is called a *typing context*. Each clause $\mathsf{K}_i\ \alpha$ can be read as an assumption *"given an instance of class* $\mathsf{K}_\imath$ *at type* α*"* and $\vec{P}$ as the conjunction of these assumptions. We say that classes K_i's are superclasses of K which we write $\mathsf{K}_i \prec \mathsf{K}$. They must have been previously defined. This ensures that the relation $\prec$ is acyclic. We require that all K_i's are independent, *i.e.* there doa not exists i and j such that $\mathsf{K}_j \prec \mathsf{K}_i$.

An instance definition h is of the form $\mathsf{inst}\ \forall \vec{\beta}.\ \vec{P} \Rightarrow \mathsf{K}\ (\mathsf{G}\ \vec{\beta})\ \{r\}$. It defines an instance of a class K at type $\mathsf{G}\ \vec{\beta}$ 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 τ. An instance definition *d*efines 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 α_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 $\mathsf{K}'\ \alpha'$ in typing contexts and class declarations, and to types of the form $\mathsf{K}'\ (\mathsf{G}'\ \vec{\alpha}')$ in instances are for simplicity. Generalization are possible and discussed later (§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 σ described on Figure 7.1. Notice that types T are stratified: they are either dictionary types $\mathsf{K}\ \tau$ or a regular type τ that does not contain dictionary types.

Class declaration The elaboration of a class declaration H_{K} of the form $\mathsf{class}\ \mathsf{K}_1\ \alpha, \ldots\ \mathsf{K}_n\ \alpha \Rightarrow \mathsf{K}\ \alpha\ \{\rho\}$ 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 $\mathsf{K}_j \prec \mathsf{K}_i$, 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 ρ of the class definition only lists the class methods. Hence, we extend it with fields for sub-dictionaries and define the record type:

$$\mathsf{K}\ \alpha \approx \{\rho^{\mathsf{K}}\} \qquad \text{where } \rho^{\mathsf{K}} \text{ is } u^{\mathsf{K}}_{\mathsf{K}_1} : \mathsf{K}_1\ \alpha, \ldots\ u^{\mathsf{K}}_{\mathsf{K}_n} : \mathsf{K}_n\ \alpha, \rho.$$

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

Then, for each $u : T_u$ in ρ^{K}, we define the program context:

$$\mathcal{R}_u \triangleq \mathsf{let}\ u : \sigma_u = N_u\ \mathsf{in}\ [\,] \qquad \text{where} \quad \sigma_u \triangleq \forall\alpha.\, \mathsf{K}\ \alpha \Rightarrow T_u \text{ and } N_u \triangleq \Lambda\alpha.\lambda z{:}\mathsf{K}\ \alpha.\, (z.u)$$

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 $[\![H_{\mathsf{K}}]\!]$ of a single class declaration H_{K} is the composition:

$$[\![H_{\mathsf{K}}]\!] \triangleq \mathcal{R}_{u_1} \circ \ldots \mathcal{R}_{u_n} \qquad \text{where} \quad \mathsf{K}\ \alpha \approx \{u_1 : T_1, \ldots u_n : T_n\}$$

that defines accessors for each field of the class dictionary. We also define the typing environment Γ_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 $[\![H_1]\!] \circ \ldots [\![H_p]\!]$ 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 $\mathsf{inst}\ \forall \vec{\beta}.\ \vec{P} \Rightarrow \mathsf{K}\ (\mathsf{G}\ \vec{\beta})\ \{r\}$, The typing context $\vec{P}$ describes the dictionaries that must be available on type parameters $\vec{\beta}$ for constructing the dictionary $\mathsf{K}\ (\mathsf{G}\ \vec{\beta})$, but that cannot yet be built because they depend on some unknown type β 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 $\mathsf{K}\ \alpha$ requires the existence of a dictionary $\mathsf{K}'\ \alpha$; then, an instance declaration $\mathsf{K}\ \mathsf{G}$ (where G is nullary) need not be parametrized over a dictionary of type $\mathsf{K}'\ \mathsf{G}$, 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 $\mathsf{K}_1\ \alpha_1, \ldots \mathsf{K}_k\ \alpha_k$ to *canonical* ones defined as satisfying the two following conditions: (1) α_i is some β_j in $\vec{\beta}$; and (2) if $\mathsf{K}_i \prec \mathsf{K}_j$ or $\mathsf{K}_j \prec \mathsf{K}_i$ or $\mathsf{K}_i = \mathsf{K}_j$. then α_i and α_j are different. The latter condition avoids having two dictionaries $\mathsf{K}_i\ \beta$ and $\mathsf{K}_j\ \beta$ when, *e.g.*, $\mathsf{K}_i \prec \mathsf{K}_j$ since the former is contained in the latter.

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

$$\sigma_h \triangleq \forall \beta_1 \ldots \beta_p.\, \mathsf{K}_1\ \alpha_1 \Rightarrow \ldots \mathsf{K}_k\ \alpha_k \Rightarrow \mathsf{K}\ (\mathsf{G}\ \vec{\beta})$$

(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 $\mathsf{K}\ (\mathsf{G}\ \vec{\beta})$, given p dictionaries (where p may be *zero*) of respective types $\mathsf{K}_1\ \beta_1, \ldots \mathsf{K}_k\ \beta_k$ and is defined as:

$$\begin{array}{rcl} N^h & \triangleq & \Lambda\beta_1.\ \ldots \Lambda\beta_p.\ \lambda(z_1{:}\mathsf{K}_1\ \alpha_1).\ \ldots \lambda(z_k{:}\mathsf{K}_k\ \alpha_k). \\ & & \qquad \{u^{\mathsf{K}}_{\mathsf{K}'_1} = q_1, \ldots\ u^{\mathsf{K}}_{\mathsf{K}'_n} = q_n,\ u_1 = N^h_1, \ldots\ u_m = N^h_m\} \end{array}$$

The types of fields are as prescribed by the class definition K, but specialized at type $\mathsf{G}\ \vec{\beta}$. That is, q_i is a dictionary expression of type $\mathsf{K}'_i\ (\mathsf{G}\ \vec{\beta})$ whose exact definition is postponed until the elaboration of dictionaries in §7.2.6. The term N^h_i 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^h_i sees the type variables $\beta_1, \ldots \beta_p$ and the dictionary parameters $z_1 : \mathsf{K}_1\ \alpha_1, \ldots z_k : \mathsf{K}_k\ \alpha_k$.

The elaboration of all instance definitions is the program context:

$$[\![\vec{h}]\!] \triangleq \mathsf{let\ rec}\ (\vec{z}_h : \vec{\sigma}_h) = \vec{N}^h\ \mathsf{in}\ [\,]$$

that recursively binds all instance definitions in the hole.

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

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

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 $\Gamma \vdash M \rightsquigarrow N : \sigma$ where Γ is a System F typing context, M is the source expression, N is the elaborated expression and σ its type in Γ. In particular, $\Gamma \vdash M \rightsquigarrow N : \sigma$ 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 Let, we require σ to be canonical, *i.e.* of the form $\forall\vec{\alpha}.\,\vec{P} \Rightarrow T$ where $\vec{P}$ is itself empty or canonical (see page 147). Rules App and Abs do not apply to overloaded expressions of type σ but only to simple expressions of type τ.

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

The premise $\Gamma \vdash q : Q$ of rule 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 Γ and Q are given and q is inferred. This is described in detail in §7.2.6.

$$\textsc{Var}\ \frac{x:\sigma \in \Gamma}{\Gamma \vdash x \rightsquigarrow x : \sigma} \qquad \textsc{Inst}\ \frac{\Gamma \vdash M \rightsquigarrow N : \forall\alpha.\,\sigma}{\Gamma \vdash M\ \tau \rightsquigarrow N\ \tau : [\alpha \mapsto \tau]\sigma} \qquad \textsc{Gen}\ \frac{\Gamma, \alpha \vdash M \rightsquigarrow N : \sigma}{\Gamma \vdash \Lambda\alpha.M \rightsquigarrow \Lambda\alpha.N : \forall\alpha.\,\sigma}$$

$$\textsc{Let}\ \frac{\Gamma \vdash M_1 \rightsquigarrow N_1 : \sigma \qquad \Gamma, x:\sigma \vdash M_2 \rightsquigarrow N_2 : \tau}{\Gamma \vdash \mathsf{let}\ x:\sigma = M_1\ \mathsf{in}\ M_2 \rightsquigarrow \mathsf{let}\ x:\sigma = N_1\ \mathsf{in}\ N_2 : \tau} \qquad \textsc{Abs}\ \frac{\Gamma, x:\tau' \vdash M \rightsquigarrow N : \tau}{\Gamma \vdash \lambda x{:}\tau'.\,M \rightsquigarrow \lambda x{:}\tau'.\,N : \tau' \to \tau}$$

$$\textsc{App}\ \frac{\Gamma \vdash M_1 \rightsquigarrow N_1 : \tau_2 \to \tau_1 \qquad \Gamma \vdash M_2 \rightsquigarrow N_2 : \tau_2}{\Gamma \vdash M_1\ M_2 \rightsquigarrow N_1\ N_2 : \tau_1} \qquad \textsc{OAbs}\ \frac{\Gamma, x:Q \vdash M \rightsquigarrow N : \sigma \qquad x \# M}{\Gamma \vdash M \rightsquigarrow \lambda x{:}Q.\,N : Q \Rightarrow \sigma}$$

$$\textsc{OApp}\ \frac{\Gamma \vdash M \rightsquigarrow N : Q \Rightarrow \sigma \qquad \Gamma \vdash q : Q}{\Gamma \vdash M \rightsquigarrow N\ q : \sigma}$$

Figure 7.2: Elaboration of expressions

By construction, elaboration produces well-typed expressions: that is $\Gamma_{\vec{H}\vec{h}} \vdash M \rightsquigarrow N : \tau$ implies that is $\Gamma_{\vec{H}\vec{h}} \vdash N : \tau$.

7.2.5 Summary of the elaboration

An instance declaration h of the form:

$$\mathsf{inst}\ \forall\vec{\beta}.\ \mathsf{K}_1\ \alpha_1, \ldots \mathsf{K}_k\ \alpha_k \Rightarrow \mathsf{K}\ \vec{\tau}\ \{u_1 = M_1, \ldots l; u_m = M_m\}$$

is translated into

$$\lambda(z_1{:}\mathsf{K}_1\ \alpha_1) \ldots . \lambda(z_p{:}\mathsf{K}_k\ \alpha_k).\,\{u^{\mathsf{K}}_{\mathsf{K}'_1} = q_1, \ldots\ u^{\mathsf{K}}_{\mathsf{K}'_n} = q_n,\ u_1 = N_1, \ldots\ u_m = N_m\}$$

where $u^{\mathsf{K}}_{\mathsf{K}'_i} : \tau_i$ are the superclasses fields, Γ^h is $\vec{\beta}, \mathsf{K}_1\ \alpha_1, \ldots \mathsf{K}_k\ \alpha_k$, and the following elaboration judgments $\Gamma_{\vec{H}\vec{h}}, \Gamma^h \vdash q_i : \tau_i$ and $\Gamma_{\vec{H}\vec{h}}, \Gamma^h \vdash M_i \rightsquigarrow N_i : \tau_i$ hold. Finally, given the program p equal to $\vec{H}\ \vec{h}\ M$, we elaborate M as N such that $\Gamma_{\vec{H}\vec{h}} \vdash M \rightsquigarrow N : \forall\bar{\alpha}.\,\tau$.

Notice that $\forall\bar{\alpha}.\,\tau$ 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 Let so that either σ is of the form $\forall\bar{\alpha}.\,\tau$ 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:

$$\mathsf{let}\ x_1 = \lambda y.\, \mathsf{let}\ x_2 = V_1\ V_2\ \mathsf{in}\ M_2\ \mathsf{in}\ [x_1 \mapsto x_1\ q]M_1 \quad \textbf{(1)}$$
$$\mathsf{let}\ x_1 = \mathsf{let}\ x_2 = \lambda y.\, V_1\ V_2\ \mathsf{in}\ [x_2 \mapsto x_2\ q]M_2\ \mathsf{in}\ M_1 \quad \textbf{(2)}$$

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 types 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 $[\![p]\!]$ 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.

$$\text{D-OVar}\ \frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \qquad \text{D-Inst}\ \frac{\Gamma \vdash q : \forall \alpha.\, \sigma}{\Gamma \vdash q\ \tau : [\alpha \mapsto \tau]\sigma} \qquad \text{D-App}\ \frac{\Gamma \vdash q_1 : Q_1 \Rightarrow Q_2 \qquad \Gamma \vdash q_2 : Q_1}{\Gamma \vdash q_1\ q_2 : Q_2}$$

Figure 7.3: Typing rules for dictionaries

For instance, two different elaborations 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 τ. The judgment can be read as: in type environment Γ, 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*, *i.e.* dictionary expressions that do not use dictionary parameters or projections. Thus, their derivation may only use D-OVar-Inst. They are typed in the environment $\Gamma_{\vec{H}\vec{h}}$, which does not contain free *type* variables. They actually do not access dictionaries, and only use the environment $\Gamma_{\vec{h}}$. Hence, all occurrences of D-OVar-Inst are of the form:

$$\text{D-OVar-Inst}\ \frac{z : \forall \vec{\beta}.\, P_1 \Rightarrow \ldots\ P_n \Rightarrow \mathsf{K}\ (\mathsf{G}\ \vec{\beta}) \in \Gamma_{\vec{h}} \qquad \Gamma_{\vec{h}} \vdash q_i : [\vec{\beta} \mapsto \vec{\tau}]P_i}{\Gamma_{\vec{h}} \vdash z\ \vec{\tau}\ \vec{q} : \mathsf{K}\ (\mathsf{G}\ \vec{\tau})}$$

$$\frac{z : \forall \vec{\beta}.\, P_1 \Rightarrow \ldots\ P_n \Rightarrow \mathsf{K}\ (\mathsf{G}\ \vec{\beta}) \in \Gamma \qquad \forall i \in 1..n,\ \Gamma \vdash q_i : [\vec{\beta} \mapsto \vec{\tau}]P_i}{\Gamma \vdash z\ \vec{\tau}\ \vec{q} : \mathsf{K}\ (\mathsf{G}\ \vec{\tau})}\ \textsc{D-OVar-Inst}$$

$$\frac{u : \forall \alpha.\, \mathsf{K}'\ \alpha \Rightarrow \mathsf{K}\ \alpha \in \Gamma \qquad \Gamma \vdash q : \mathsf{K}'\ \tau}{\Gamma \vdash u\ \tau\ q : \mathsf{K}\ \tau}\ \textsc{D-Proj} \qquad \frac{z : \mathsf{K}\ \alpha \in \Gamma}{\Gamma \vdash z : \mathsf{K}\ \alpha}\ \textsc{D-Var}$$

Figure 7.4: Algorithmic typing rules for dictionaries

and the premise $\Gamma \vdash q_i : [\vec{\beta} \mapsto \vec{\tau}]P_i$ is itself recursively built in the same way with this single rule. This rule can be read as a recursive definition, where Γ 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 $z : \forall \vec{\beta}.\, P_1 \Rightarrow \ldots\ P_n \Rightarrow \mathsf{K}\ (\mathsf{G}\ \vec{\beta})$ in Γ. The binding z can only be a binding z^h introduced as the elaboration of some class instance h at type $\Gamma\vec{\beta}$, Hence, it suffices that *instance definitions never overlap* for z^h to be uniquely determined; if recursively each q_i is unique, then $z\ \vec{\tau}\ \vec{q}$ also is. Under this hypothesis, the elaboration is always unique and therefore coherent.

Definition 4 (Overlapping instances) *Two instances* $\mathsf{inst}\ \forall \vec{\beta}_1.\ \vec{P} \Rightarrow \mathsf{K}\ (\mathsf{G}_1\ \vec{\beta}_1)\ \{r_1\}$ *and* $\mathsf{inst}\ \forall \vec{\beta}_2.\ \vec{P} \Rightarrow \mathsf{K}\ (\mathsf{G}_2\ \vec{\beta}_2)\ \{r_2\}$ *of a class* K overlap *if the type schemes* $\forall \vec{\beta}_1.\mathsf{K}\ (\mathsf{G}_1\ \vec{\tau}_1)$ *and* $\forall \vec{\beta}_2.\mathsf{K}\ (\mathsf{G}_2\ \vec{\tau}_2)$ *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 $\Gamma_{\vec{H}\vec{h}}, \Gamma^h$ where Γ^h binds the local typing context, *i.e.* assumptions $z : \mathsf{K}'\ \beta$ about dictionaries received as arguments. Hence, rules D-Proj and D-Var may now apply, *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 τ and then the reverse order of class inheritance: when no rule applies, the search fails; when rule D-Var applies, the search ends

with a successful derivation; when rule D-Proj applies, the premise is called with a smaller problem since the height is unchanged and $\mathsf{K}'\ \vec{\tau}$ with $\mathsf{K}' \prec \mathsf{K}$; when D-Ovar-Inst applies, the premises are called at type $\mathsf{K}_i\ \tau_j$ where τ_j is subtype of $\vec{\tau}$, 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 EqInt and OrdInt where the later contains an instance of the former. Hence, a dictionary of type EqInt may be obtained, either directly as EqInt, or indirectly as *Eq* OrdInt, by projecting the *Eq* sub-dictionary of class *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 D-Proj to cases where Q is P–when 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 $\Gamma_{\vec{H}\vec{h}}$ is (we remind both the informal and formal names of variables):

$$\begin{array}{rcll}
\text{equal} & \triangleq & u_{equal} & : \forall\alpha.\, Eq\ \alpha \Rightarrow \alpha \to \alpha \to \mathsf{bool}, \\
\text{EqInt} & \triangleq & z_{Eq}^{Int} & : Eq\ \mathsf{int} \\
\text{EqList} & \triangleq & z_{Eq}^{List} & : \forall\alpha.\, Eq\ \alpha \Rightarrow Eq\ (\mathsf{list}\ \alpha) \\
 & & & \\
\text{EqOrd} & \triangleq & u_{Eq}^{Ord} & : \forall\alpha.\, Ord\ \alpha \Rightarrow Eq\ \alpha \\
\text{lt} & \triangleq & u_{lt} & : \forall\alpha.\, Ord\ \alpha \Rightarrow \alpha \to \alpha \to \mathsf{bool}
\end{array}$$

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

$$\text{D-Proj}\ \frac{\text{D-OVar-Inst}\ \Gamma \vdash z : u_{Eq}^{Ord} : Ord\ \alpha \to Eq\ \alpha \qquad \text{D-Var}\ \Gamma \vdash z : Ord\ \alpha}{\Gamma \vdash u_{Eq}^{Ord}\ \alpha\ z : Eq\ \alpha}$$

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.

```
class Eq (X)   { equal : X → X → Bool }
inst Eq (Int)  { equal = (==) }
inst Eq (Char) { equal = (==) }
inst Λ(X) Eq (X) ⇒ Eq (List (X))
  { Eq = λ(l1) λ(l2) match l1, l2 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 rebuilt by type inference.

Type inference To perform type inference in Mini Haskell, the idea is to see dictionary types $\mathsf{K}\ \tau$, which can only appear in type schemes and not in types, as a type constraint to mean *"there exists a dictionary of type* $\mathsf{K}\ \alpha$*"*. That is, we may read the type scheme $\forall\vec{\alpha}.\,\vec{P} \Rightarrow \tau$ as the constraint type scheme $\forall\vec{\alpha}[\vec{P}].\,\tau$ where $\vec{P}$ is seen as a type predicate, say a dictionary predicate. Therefore, we extend constraints with dictionary predicates:

$$C ::= \ldots \mid \mathsf{K}\ \tau$$

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

$$\frac{\textsc{Instance}\quad \mathsf{K}\ \phi\tau}{\phi \vdash \mathsf{K}\ \tau} \qquad\qquad \frac{\textsc{Instance}\quad \Gamma_{\vec{H}\vec{h}} \vdash \rho : \mathsf{K}\ \phi\tau}{\phi \vdash \mathsf{K}\ \tau}$$

We use entailment to reason with class-membership constraints. For every class declaration $\mathsf{class}\ \mathsf{K}_1\ \alpha_1, \ldots\ \mathsf{K}_n\ \alpha_n \Rightarrow \mathsf{K}\ \alpha\ \{\rho\}$, we have:

$$\mathsf{K}\ \alpha \Vdash \mathsf{K}_1\ \alpha_1 \wedge \ldots\ \mathsf{K}_n\ \alpha_n \qquad \textbf{(K1)}$$

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

Proof: Assume $\phi \vdash \mathsf{K}\ \alpha$, *i.e.* by Rule Instance $\Gamma_{\vec{H}\vec{h}} \vdash q : \mathsf{K}\ (\phi\alpha)$ for some dictionary q. From the class declaration in $\Gamma_{\vec{H}\vec{h}}$, we know that $\mathsf{K}\ \alpha$ is a record type definition that contains

fields $u^{\mathsf{K}}_{\mathsf{K}_i}$ of type $\mathsf{K}_i\ \alpha_i$. Hence, the dictionary value q contains field values of types $\mathsf{K}_i\ (\phi\alpha)$. Therefore, we have $\phi \vdash \mathsf{K}_i\ \alpha$ for all i in $1..n$, which implies $\phi \vdash \mathsf{K}_1\ \alpha \wedge \ldots\ \mathsf{K}_n\ \alpha$.

For every instance definition $\mathsf{inst}\ \forall\vec{\beta}.\ \mathsf{K}_1\ \beta_1, \ldots\ \mathsf{K}_p\ \beta_p \Rightarrow \mathsf{K}\ (\mathsf{G}\ \vec{\beta})\ \{r\}$, we have

$$\mathsf{K}\ (\mathsf{G}\ \vec{\beta}) \equiv \mathsf{K}_1\ \beta_1 \wedge \ldots\ \mathsf{K}_p\ \beta_p \qquad \textbf{(K2)}$$

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

Proof: Let h be the above instance definition. We proof both directions separately:

Case $\dashv\Vdash$: Assume $\phi \vdash \mathsf{K}_i\ \beta_i$ for i in $\{1, \ldots p\}$. By Rule INSTANCE, for each i, there exists a dictionary q_i such that $\Gamma_{\vec{H}\vec{h}} \vdash q_i : \mathsf{K}_i\ (\phi\beta_i)$. Hence, $\Gamma_{\vec{H}\vec{h}} \vdash x_h\ \vec{\beta}\ q_1\ \ldots\ q_p : \mathsf{K}\ (\mathsf{G}\ (\phi\vec{\beta}))$, *i.e.* by Rule INSTANCE $\phi \vdash \mathsf{K}\ (\mathsf{G}\ \vec{\beta})$.

Case $\Vdash\vdash$: Assume, $\phi \vdash \mathsf{K}\ (\mathsf{G}\ \vec{\beta})$. *i.e.* there exists a dictionary q such that $\Gamma_{\vec{H}\vec{h}} \vdash q : \mathsf{K}\ (\mathsf{G}\ (\phi\vec{\beta}))$. By inversion of typing (and non-overlapping of instance declarations), the only way to build such a dictionary is by an application of z_h. Hence, q must be of the form $x_h\ \vec{\beta}\ q_1\ \ldots\ q_p$ with $\Gamma_{\vec{H}\vec{h}} \vdash q_i : \mathsf{K}_i\ (\phi\beta_i)$. By Rule INSTANCE, this means $\phi \vdash \mathsf{K}_i\ \beta_i$ for every i, which implies $\phi \vdash \mathsf{K}_1\ \beta_1 \wedge \ldots\ \mathsf{K}_p\ \beta_p$.

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 $\Vdash\vdash$ direction does not hold anymore; the rewriting rule:

$$\mathsf{K}\ (\mathsf{G}\ \vec{\beta}) \longrightarrow \mathsf{K}_1\ \beta_1 \wedge \ldots\ \mathsf{K}_p\ \beta_p$$

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:

$$\mathsf{K}\ (\mathsf{G}\ \vec{\beta}) \equiv \mathsf{false} \qquad \textbf{(K3)}$$

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

Proof: The $\dashv\Vdash$ direction is a tautology, so it suffices to prove the $\Vdash\vdash$ direction. By contradiction. Assume $\phi \vdash \mathsf{K}\ (\mathsf{G}\ \vec{\beta})$. This implies the existence of a dictionary q such that $\Gamma_{\vec{H}\vec{h}} \vdash q : \mathsf{K}\ (\mathsf{G}\ (\phi\vec{\beta}))$. Then, there must be some x_h in Γ whose type scheme is of the form $\forall\vec{\beta}.\ \vec{P} \Rightarrow \mathsf{K}\ (\mathsf{G}\ \vec{\beta})$, *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 $\mathsf{K}\ (\mathsf{G}\ \vec{\beta}) \longrightarrow \mathsf{false}$ 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 $\forall \bar{\alpha}[P_1 \wedge P_2].\,\tau$ where $\mathsf{ftv}(P_1) \in \bar{\alpha}$ and $\mathsf{ftv}(P_2) \# \bar{\alpha}$. 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 $\forall \vec{\alpha}.\,\vec{P} \Rightarrow \tau$ with a typing context $\vec{P}$.

To check well-typedness of a program $\vec{H}\ \vec{h}\ a$, 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:

- $\Gamma_{\vec{H}\vec{h}}, \Gamma^h \vdash a^h : \tau^h$ where τ^h is given. That is, that $\mathsf{def}\ \Gamma_{\vec{H}\vec{h}}, \Gamma^h\ \mathsf{in}\ (\!|\, a^h \,|\!) \preceq \tau^h \equiv \mathsf{true}$ holds;
- $\Gamma_{\vec{H}\vec{h}} \vdash a : \tau$ for some τ. That is, that $\mathsf{def}\ \Gamma_{\vec{H}\vec{h}}\ \mathsf{in}\ \exists\alpha.\,(\!|\, a \,|\!) \preceq \alpha \equiv \mathsf{true}$ 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 $\Gamma \vdash a \rightsquigarrow M : \tau$.

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 $\forall\alpha[\mathsf{K}\ \alpha].\,\alpha \to \alpha$ and the typing environment contains two instances of $\mathsf{K}\ \mathsf{G}_1$ and $\mathsf{K}\ \mathsf{G}_2$ of class K. Constraint-free

instances of this type scheme are $\mathsf{G}_1 \to \mathsf{G}_1$ and $\mathsf{G}_2 \to \mathsf{G}_2$ but $\forall\alpha.\,\alpha \to \alpha$ 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 $\Gamma_{\vec{H}\vec{h}}$ 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 $\forall\vec{\beta}[\vec{Q}].\,\tau$ in $\Gamma_{\vec{H}\vec{h}}$ or, equivalently, a type τ in $\Gamma_{\vec{H}\vec{h}}, \vec{\beta}, \vec{Q}$. Consider, for example:

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

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

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

The type of v is $\forall\alpha[\mathit{Readable}\ \alpha].\,\mathsf{unit} \to \alpha$—and several classes are possible for $\mathit{Readable}\ \alpha$. 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 $\mathsf{let}\ x = a_1\ \mathsf{in}\ a_2$ elaborates to $\mathsf{let}\ x : \forall\alpha[\mathsf{K}\ \alpha].\,\mathsf{int} \to \mathsf{int} = N_1\ \mathsf{in}\ N_2$. All applications of x in N_2 will lead to an unresolved constraint $\mathsf{K}\ \alpha$ for some fresh α since neither the argument nor the context of this application can determine the value of the type parameter α. Still, a dictionary of type $\mathsf{K}\ \tau$ 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 $\mathsf{K}\ \mathsf{G}$ for a class K that appears in an unresolved or isolated constraint $\mathsf{K}\ \alpha$, 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

$$\texttt{let}\ \mathit{twice} = \lambda(x)\ x + x\ \texttt{in}\ \mathit{twice}\ (\mathit{twice}\ 1)$$

whose principal type reconstruction is:

$$\texttt{let}\ \mathit{twice} : \forall(X)\ [\ \mathit{Num}\ X\ ]\ X \to X = \Lambda(X)\ [\mathit{Num}\ X]\ \lambda(x)\ x + x\ \texttt{in}$$
$$\mathit{twice}\ \mathit{Int}\ (\mathit{twice}\ \mathit{Int})\ 1$$

This program is coherent. It's natural elaboration is

$$\texttt{let}\ \mathit{twice}\ X\ \mathit{NumX} = \lambda(x : X)\ x\ (\mathit{plus}\ \mathit{NumX})\ x\ \texttt{in}$$
$$\mathit{twice}\ \mathit{Int}\ \mathrm{NumInt}\ (\mathit{twice}\ \mathit{Int}\ \mathrm{NumInt}\ 1)$$

However, it can also be elaborated to

$$\texttt{let}\ \mathit{twice} = \lambda(x : \mathit{Int})\ x\ (\mathit{plus}\ \mathrm{NumInt})\ x\ \texttt{in}\ \mathit{twice}\ (\mathit{twice}\ 1)$$

avoiding the generalization of twice; moreover, the overloaded application $\mathit{plus}\ \mathrm{NumInt}$ can now be statically reduced, leading to:

$$\texttt{let}\ \mathit{twice} = \lambda(x : \mathit{Int})\ x\ \mathit{Int}.(+)\ x\ \texttt{in}\ \mathit{twice}\ (\mathit{twice}\ 1)$$

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 $\mathrm{read} : \mathit{descr} \to X$, 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 $\mathsf{K}\ \alpha$ have types of the form $\alpha \to \tau$. 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 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:

$$\mathsf{class}\ \vec{P} \Rightarrow \mathsf{K}\ \vec{\alpha}\ \{\rho\}$$

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:

```
class Collection C E { find : C → E → Option(E), add : C → E → C }
inst Collection (List X) X { find = List.find, add = λ(c)λ(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:

```
let add2 c x y = add (add c x) y
add2 : ∀(C, E, E') Collection C E, Collection C E' ⇒ C → E → E' → 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 E | C → E { ... }
```

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:

```
class Collection E {
  type C : * → *
  find : C → E → Option E
  add : C → E → C
}
inst Collection Eq X ⇒ Collection X {type C = List E, ... }
inst Collection Eq X ⇒ Collection X {type C = Set E, ... }
```

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 (2010) for a recent new proposal. For example, the definitions:

```
inst Eq(X) { equal = (=) }
inst Eq(Int) { equal = (==) }
```

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

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

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

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

Implicit values

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

Implicits values have been used in the language Scala for implicit conversions Sca (but they can do more). An extension of OCaml with implicit values is beeing prototyped. Implicit values have also been proposed as an alternative to Haskell type classes Oliveira et al. (2012).

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, 2005).

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 every one agrees, some useful extensions often come with serious drawbacks, and they is not yet an agreement on the best design compromises. In Haskell, the design decisions have often been in favor of expressiveness, but then loosing 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.

Bibliography

▷ A tour of scala: Implicit parameters. Part of scala documentation.

▷ Martín Abadi and Luca Cardelli. A theory of primitive objects: Untyped and first-order systems. *Information and Computation*, 125(2):78–102, March 1996.

▷ Martín Abadi and Luca Cardelli. A theory of primitive objects: Second-order systems. *Science of Computer Programming*, 25(2–3):81–116, December 1995.

▷ Amal Ahmed and Matthias Blume. Typed closure conversion preserves observational equivalence. In *ACM International Conference on Functional Programming (ICFP)*, pages 157–168, September 2008.

▷ Lennart Augustsson. Implementing Haskell overloading. In *FPCA '93: Proceedings of the conference on Functional programming languages and computer architecture*, pages 65–73, New York, NY, USA, 1993. ACM. ISBN 0-89791-595-X.

▷ Nick Benton and Andrew Kennedy. Exceptional syntax journal of functional programming. *J. Funct. Program.*, 11(4):395–410, 2001.

▷ Richard Bird and Lambert Meertens. Nested datatypes. In *International Conference on Mathematics of Program Construction (MPC)*, volume 1422 of *Lecture Notes in Computer Science*, pages 52–67. Springer, 1998.

Nikolaj Skallerud Bjørner. Minimal typing derivations. In *In ACM SIGPLAN Workshop on ML and its Applications*, pages 120–126, 1994.

Daniel Bonniot. *Typage modulaire des multi-méthodes*. PhD thesis, École des Mines de Paris, November 2005.

▷ Daniel Bonniot. Type-checking multi-methods in ML (a modular approach). In *Workshop on Foundations of Object-Oriented Languages (FOOL)*, January 2002.

▷ Michael Brandt and Fritz Henglein. Coinductive axiomatization of recursive type equality and subtyping. *Fundamenta Informaticæ*, 33:309–338, 1998.

▷ Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. Comparing object encodings. *Information and Computation*, 155(1/2):108–133, November 1999.

Luca Cardelli. An implementation of f¡:. Technical report, DEC Systems Research Center, 1993.

Giuseppe Castagna. *Object-Oriented Programming: A Unified Foundation*. Progress in Theoretical Computer Science Series. Birkäuser, Boston, 1997.

▷ Henry Cejtin, Matthew Fluet, Suresh Jagannathan, and Stephen Weeks. The MLton compiler, 2007.

▷ Arthur Charguéraud and François Pottier. Functional translation of a calculus of capabilities. In *ACM International Conference on Functional Programming (ICFP)*, pages 213–224, September 2008.

▷ Juan Chen and David Tarditi. A simple typed intermediate language for object-oriented languages. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 38–49, January 2005.

▷ Adam Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 54–65, June 2007.

▷ Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type erasure semantics. *Journal of Functional Programming*, 12(6):567–600, November 2002.

Julien Crétin and Didier Rémy. Extending System F with Abstraction over Erasable Coercions. In *Proceedings of the 39th ACM Conference on Principles of Programming Languages*, January 2012.

Joshua Dunfield. Greedy bidirectional polymorphism. In *ML '09: Proceedings of the 2009 ACM SIGPLAN workshop on ML*, pages 15–26, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-509-3. doi: http://doi.acm.org/10.1145/1596627.1596631.

Jun Furuse. Extensional polymorphism by flow graph dispatching. In Ohori (2003), pages 376–393. ISBN 3-540-20536-5.

▷ Jun Furuse. Extensional polymorphism by flow graph dispatching. In *Asian Symposium on Programming Languages and Systems (APLAS)*, volume 2895 of *Lecture Notes in Computer Science*. Springer, November 2003b.

▷ Jacques Garrigue. Relaxing the value restriction. In *Functional and Logic Programming*, volume 2998 of *Lecture Notes in Computer Science*, pages 196–213. Springer, April 2004.

Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Thèse d'état, Université Paris 7, June 1972.

▷ Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*. Cambridge University Press, 1990.

▷ Dan Grossman. Quantified types in an imperative language. *ACM Transactions on Programming Languages and Systems*, 28(3):429–475, May 2006.

▷ Bob Harper and Mark Lillibridge. ML with callcc is unsound. Message to the TYPES mailing list, July 1991.

Robert Harper and Benjamin C. Pierce. Design considerations for ML-style module systems. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 8, pages 293–345. MIT Press, 2005.

▷ Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, April 1993.

▷ J. Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and Lambda-Calculus*. Cambridge University Press, 1986.

▷ Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of Haskell: being lazy with class. In *ACM SIGPLAN Conference on History of Programming Languages*, June 2007.

Gérard Huet. *Résolution d'équations dans des langages d'ordre* 1, 2, ..., ω. PhD thesis, Université Paris 7, September 1976.

▷ John Hughes. Why functional programming matters. *Computer Journal*, 32(2):98–107, 1989.

▷ Mark P. Jones. Simplifying and improving qualified types. In *FPCA '95: Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 160–169, New York, NY, USA, 1995a. ACM. ISBN 0-89791-719-7.

Mark P. Jones. Typing Haskell in Haskell. In *In Haskell Workshop*, 1999a.

Mark P. Jones. *Qualified types: theory and practice*. Cambridge University Press, New York, NY, USA, 1995b. ISBN 0-521-47253-9.

▷ Mark P. Jones. Typing Haskell in Haskell. In *Haskell workshop*, October 1999b.

▷ Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: an exploration of the design space. In *Haskell workshop*, 1997.

▷ Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. Practical type inference for arbitrary-rank types. *Journal of Functional Programming*, 17(01):1, 2006.

Stefan Kaes. Type inference in the presence of overloading, subtyping and recursive types. In *LFP '92: Proceedings of the 1992 ACM conference on LISP and functional programming*, pages 193–204, New York, NY, USA, 1992. ACM. ISBN 0-89791-481-3. doi: http://doi.acm.org/10.1145/141471.141540.

▷ Assaf J. Kfoury, Jerzy Tiuryn, and Pawel Urzyczyn. ML typability is DEXPTIME-complete. In *Colloquium on Trees in Algebra and Programming*, volume 431 of *Lecture Notes in Computer Science*, pages 206–220. Springer, May 1990.

▷ Peter J. Landin. Correspondence between ALGOL 60 and Church's lambda-notation: part I. *Communications of the ACM*, 8(2):89–101, 1965.

▷ Konstantin Läufer and Martin Odersky. Polymorphic type inference and abstract data types. *ACM Transactions on Programming Languages and Systems*, 16(5):1411–1430, September 1994.

▷ Didier Le Botlan and Didier Rémy. Recasting MLF. *Information and Computation*, 207(6): 726–785, 2009. ISSN 0890-5401. doi: 10.1016/j.ic.2008.12.006.

▷ Xavier Leroy. *Typage polymorphe d'un langage algorithmique*. PhD thesis, Université Paris 7, June 1992.

▷ Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 42–54, January 2006.

▷ Xavier Leroy and François Pessaux. Type-based analysis of uncaught exceptions. *ACM Trans. Program. Lang. Syst.*, 22(2):340–377, 2000. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/349214.349230.

▷ John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 47–57, January 1988.

▷ Harry G. Mairson. Deciding ML typability is complete for deterministic exponential time. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 382–401, 1990.

▷ David McAllester. A logical algorithm for ML type inference. In *Rewriting Techniques and Applications (RTA)*, volume 2706 of *Lecture Notes in Computer Science*, pages 436–451. Springer, June 2003.

Todd D. Millstein and Craig Chambers. Modular statically typed multimethods. In *ECOOP '99: Proceedings of the 13th European Conference on Object-Oriented Programming*, pages 279–303, London, UK, 1999. Springer-Verlag. ISBN 3-540-66156-5.

▷ Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.

▷ Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 271–283, January 1996.

▷ John C. Mitchell. Polymorphic type inference and containment. *Information and Computation*, 76(2–3):211–249, 1988.

▷ John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, 1988.

▷ Benoît Montagu and Didier Rémy. Modeling abstract types in modules with open existential types. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 63–74, January 2009.

J. Garrett Morris and Mark P. Jones. Instance chains: type class programming without overlapping instances. In *ICFP '10: Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, pages 375–386, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-794-3. doi: http://doi.acm.org/10.1145/1863543.1863596.

▷ Greg Morrisett and Robert Harper. Typed closure conversion for recursively-defined functions (extended abstract). In *International Workshop on Higher Order Operational Techniques in Semantics (HOOTS)*, volume 10 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 1998.

▷ Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):528–569, May 1999.

▷ Alan Mycroft. Polymorphic type schemes and recursive definitions. In *International Symposium on Programming*, volume 167 of *Lecture Notes in Computer Science*, pages 217–228. Springer, April 1984.

▷ Matthias Neubauer, Peter Thiemann, Martin Gasbichler, and Michael Sperber. Functional logic overloading. pages 233–244, 2002. doi: http://doi.acm.org/10.1145/565816.503294.

▷ Martin Odersky, Philip Wadler, and Martin Wehr. A second look at overloading. In *FPCA '95: Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 135–146, New York, NY, USA, 1995. ACM. ISBN 0-89791-719-7.

▷ Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999.

▷ Martin Odersky, Matthias Zenger, and Christoph Zenger. Colored local type inference. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 41–53, 2001.

Atsushi Ohori, editor. *Programming Languages and Systems, First Asian Symposium, APLAS 2003, Beijing, China, November 27-29, 2003, Proceedings*, volume 2895 of *Lecture Notes in Computer Science*, 2003. Springer. ISBN 3-540-20536-5.

▷ Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1999.

▷ Bruno C.d.S. Oliveira, Tom Schrijvers, Wontae Choi, Wonchan Lee, and Kwangkeun Yi. The implicit calculus: a new foundation for generic programming. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, pages 35–44, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1205-9. doi: 10.1145/2254064.2254070.

▷ Simon Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. Online lecture notes, January 2009.

▷ Simon Peyton Jones and Mark Shields. Lexically-scoped type variables. Manuscript, April 2004.

▷ Simon Peyton Jones and Philip Wadler. Imperative functional programming. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 71–84, January 1993.

Frank Pfenning. Partial polymorphic type inference and higher-order unification. In *LFP '88: Proceedings of the 1988 ACM conference on LISP and functional programming*, pages 153–163, New York, NY, USA, 1988. ACM. ISBN 0-89791-273-X. doi: http://doi.acm.org/10.1145/62678.62697.

▷ Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

▷ Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Transactions on Programming Languages and Systems*, 22(1):1–44, January 2000.

▷ Andrew M. Pitts. Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science*, 10:321–359, 2000.

▷ François Pottier. Notes du cours de DEA "Typage et Programmation", December 2002.

François Pottier. A typed store-passing translation for general references. In *Proceedings of the 38th ACM Symposium on Principles of Programming Languages (POPL'11)*, Austin, Texas, January 2011. Supplementary material.

▷ François Pottier and Nadji Gauthier. Polymorphic typed defunctionalization and concretization. *Higher-Order and Symbolic Computation*, 19:125–162, March 2006.

François Pottier and Jonathan Protzenko. Programming with permissions in Mezzo. Submitted for publication, October 2012.

▷ François Pottier and Didier Rémy. The essence of ML type inference. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005.

▷ François Pottier and Didier Rémy. The essence of ML type inference. Draft of an extended version. Unpublished, September 2003.

▷ Didier Rémy. Simple, partial type-inference for System F based on type-containment. In *Proceedings of the tenth International Conference on Functional Programming*, September 2005.

▷ Didier Rémy. Programming objects with ML-ART: An extension to ML with abstract and record types. In *International Symposium on Theoretical Aspects of Computer Software (TACS)*, pages 321–346. Springer, April 1994a.

▷ Didier Rémy. Type inference for records in a natural extension of ML. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming: Types, Semantics and Language Design*. MIT Press, 1994b.

▷ Didier Rémy and Jérôme Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory and Practice of Object Systems*, 4(1):27–50, 1998.

Didier Rémy and Boris Yakobowski. Efficient Type Inference for the MLF language: a graphical and constraints-based approach. In *The 13th ACM SIGPLAN International Conference on Functional Programming (ICFP'08)*, pages 63–74, Victoria, BC, Canada, September 2008. doi: http://doi.acm.org/10.1145/1411203.1411216.

▷ John C. Reynolds. Towards a theory of type structure. In *Colloque sur la Programmation*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425. Springer, April 1974.

▷ John C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing 83*, pages 513–523. Elsevier Science, 1983.

▷ John C. Reynolds. Three approaches to type structure. In *International Joint Conference on Theory and Practice of Software Development (TAPSOFT)*, volume 185 of *Lecture Notes in Computer Science*, pages 97–138. Springer, March 1985.

François Rouaix. Safe run-time overloading. In *Proceedings of the 17th ACM Conference on Principles of Programming Languages*, pages 355–366, 1990. doi: http://doi.acm.org/10.1145/96709.96746.

▷ Christian Skalka and François Pottier. Syntactic type soundness for HM(X). In *Workshop on Types in Programming (TIP)*, volume 75 of *Electronic Notes in Theoretical Computer Science*, July 2002.

Geoffrey S. Smith. Principal type schemes for functional programs with overloading and subtyping. In *Science of Computer Programming*, 1994.

▷ Matthieu Sozeau and Nicolas Oury. First-Class Type Classes. In Sofiène Tahar, Otmame Ait-Mohamed, and César Muñoz, editors, *TPHOLs 2008: Theorem Proving in Higher Order Logics, 21th International Conference*, Lecture Notes in Computer Science. Springer, August 2008.

▷ Paul A. Steckler and Mitchell Wand. Lightweight closure conversion. *ACM Transactions on Programming Languages and Systems*, 19(1):48–86, 1997.

▷ Christopher Strachey. Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, 13(1–2):11–49, April 2000.

▷ Peter J. Stuckey and Martin Sulzmann. A theory of overloading. In *ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 167–178, New York, NY, USA, 2002. ACM. ISBN 1-58113-487-8.

▷ W. W. Tait. Intensional interpretations of functionals of finite type i. *The Journal of Symbolic Logic*, 32(2):pp. 198–212, 1967. ISSN 00224812.

▷ Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. *Information and Computation*, 11(2):245–296, 1994.

Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, April 1975.

▷ Jerzy Tiuryn and Pawel Urzyczyn. The subtyping problem for second-order types is undecidable. *Information and Computation*, 179(1):1–18, 2002.

▷ Mads Tofte, Lars Birkedal, Martin Elsman, and Niels Hallenberg. A retrospective on region-based memory management. *Higher-Order and Symbolic Computation*, 17(3):245–265, September 2004.

▷ Andrew Tolmach and Dino P. Oliva. From ML to Ada: Strongly-typed language interoperability via source translation. *Journal of Functional Programming*, 8(4):367–412, July 1998.

▷ Philip Wadler. Theorems for free! In *Conference on Functional Programming Languages and Computer Architecture (FPCA)*, pages 347–359, September 1989.

▷ Philip Wadler. The Girard-Reynolds isomorphism (second edition). *Theoretical Computer Science*, 375(1–3):201–226, May 2007.

▷ Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 60–76, January 1989.

Mitchell Wand. Corrigendum: Complete type inference for simple objects. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, 1988.

▷ J. B. Wells. The essence of principal typings. In *International Colloquium on Automata, Languages and Programming*, volume 2380 of *Lecture Notes in Computer Science*, pages 913–925. Springer, 2002.

▷ J. B. Wells. The undecidability of Mitchell's subtyping relation. Technical Report 95-019, Computer Science Department, Boston University, December 1995.

▷ J. B. Wells. Typability and type checking in system F are equivalent and undecidable. *Annals of Pure and Applied Logic*, 98(1–3):111–156, 1999.

▷ Andrew K. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8 (4):343–356, December 1995.

▷ Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, November 1994.