MPRI, Typage

Didier Rémy
(With much course meterial from François Pottier)

October 19, 2010

INRIA

Plan of the course

Introduction

Simply-typed λ-calculus

Polymorphism and System F

Type reconstruction

Existential types

Overloading

Overloading

Contents

What is overloading?

Overloading occurs when at some program point, several definitions for a same identifier are visible simultaneously.

An interpretation of the program (and a fortiori a run of the program) must choose the definition that applies at this point. This is called *overloading resolution*, which may use very 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.

Why use overloading?

Naming convenience

It 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.*

Modularity

To avoid clashing, the naming discipline (including name mangling conventions) must be know globally. Isolated identifiers with no particular naming convention may still interfere between different developments and cannot be used together unless fully qualified.

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.

Why use overloading?

Type dependent functions

A function defined on $T[X]$ for all v may have an implementation depending on the type of X. For instance, a marshaling function of type $\forall X.X \rightarrow \mathit{string}$ may *execute different code* for *each base type* X.

Ad hoc polymorphism

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 integer zero or the empty list. $\times$ could mean the either integer product or string concatenation.

Why use overloading?

Type dependent functions

A function defined on $T[X]$ for all v may have an implementation depending on the type of X. For instance, a marshaling function of type $\forall X. X \rightarrow \mathit{string}$ may *execute different code* for *each* base type X.

Polytypic polymorphism

Overloading definitions depend solely on the *type structure* (on whether it is a sum, a product, *etc.*) and thus derived mechanically for all types from their definitions on base types.

Typical examples of polytypic functions are marshaling functions or the generation of random values for arbitrary types, *e.g.* as used in Quickcheck for Haskell.

Different forms of overloading

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

What are the restrictions on overloading definitions?

- None, *i.e.* arbitrary definitions can be overloaded!
- Can just functions definitions or any definition be overloaded? *e.g.* 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?

How is overloading tamed?

How is overloading resolution defined?

- up to subtyping?
- *static* or *dynamic*?

Static resolution (rather simple)

- Overloaded symbols can/must be statically replaced by their implementation at the appropriate type.
- This *does* not increase expressiveness, but may still significantly reduce verbosity.

How is overloading tamed?

How is overloading resolution defined?

- up to subtyping?
- *static* or *dynamic*?

Dynamic resolution (more involved)

This is required when the choice of the implementation depends on the dynamic of the program execution. For example, the resolution at a program point in a polymorphic function may depend on the type of its arguments so that different calls can make different choices.

The resolution is driven by information made available at runtime:

- this can be full or partial type information, or extra values (tags, dictionaries, *etc.*) correlated to types instead of types themselves.
- this information may be attached to normal values or passed as extra arguments.

Static resolution

Examples

In SML

Overloaded definitions are primitive (for numerical operators), and automatic (for record access).

Typechecking fails if overloading cannot be resolved at outermost let-definitions. For example, **let** twice $x = x + x$ is rejected in SML, at toplevel as $+$ could be the addition on either integers or floats.

Static resolution

Examples

In SML

Overloaded definitions are primitive (for numerical operators), and automatic (for record access).

Typechecking fails if *overloading* cannot be resolved at outermost let-definitions. For example, **let** twice $x = x + x$ is *rejected* in SML, at toplevel as $+$ could be the addition on either integers or floats.

In Java?

Static resolution

Examples

In 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, hence the method is overloaded.

Overloading is resolved at compile time by choosing the most specific definition. There is always a best choice—according to static knowledge.

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. *This is often a source of confusion for programmers.*

Static resolution Limits

Static overloading does not fit 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 every 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.

How is dynamic resolution implemented?

Three main techniques for dynamic resolution

- Pass types at runtime and dispatch on the runtime type, using a general typecase construct.
- Tag values with their types—or usually an approximation of their types—and dispatch on the tags of values.
  (This is one possible approach to object-orientation where objects may be tagged with the class they belong to.)
- Pass the appropriate implementations at runtime as extra arguments, usually grouped in *dictionaries* of implementations.

Dynamic resolution — Type passing semantics

Runtime type dispatch

- Use an explicitly typed calculus (*e.g.* System F)
- Add a typecase function.
- The runtime cost of typecase may be high, unless type patterns are significantly restricted.
- By default, one pays even when overloading is not used.
- Monomorphization may be used to reduce type matching statically.
- Ensuring exhaustiveness of type matching is difficult.

ML& (Castagna)

- System F + intersection types + subtyping + type matching
- An expressive type system that keeps track of exhaustiveness; type matching functions are first-class and can be extended or overridden.
- Allows patterns overlapping with a best match resolution strategy.

Dynamic resolution Type erasing semantics

Passing unresolved implementations as extra arguments

- Abstract over unresolved overloaded symbols and pass them around as extra arguments.
  Hopefully, overloaded symbols can be resoled when their types are sufficiently specialized and before they are actually needed.
  In short, $\text{let } f = \lambda x.\, x + x \text{ in}$ can be elaborated into $\text{let } f = \lambda(+).\, \lambda x.\, x + x \text{ in}$ and its application to a f 1.0 to a float is elaborated into f (+.) 1.0.
- This can be done based on 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.

Dynamic resolution

Type erasing semantics

This has been explored under different facets in the context of ML:

- Type classes, introduced in [1989] by Wadler and Blott are the most popular and widely framework of this kind.
- Other contemporary proposals were proposed by Rouaix [1990] and Kaes [1992].
- Tentative simplifications of type classes have been made [Odersky et al., 1995] but did not take over, because of their restrictions.
- Recent works on type classes is still going [Morris and Jones, 2010]

We present Mini-Haskell that contains the essence on Haskell.

Contents

Mini-Haskell

Mini Haskell is a simplification of Haskell to avoid most of the difficulties of type classes but keeping their essence:

- single parameter type classes
- 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.

Our version of Mini-Haskell is explicit typed. We present:

- Some examples in Mini-Haskell
- Elaboration of Mini-Haskell into System *F*
- An implicitly-typed version with type inference.

Mini-Haskell Example — Simple Classes and instances

Mini-Haskell class declarations and instance definitions

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

This code:

- declares a class (dictionary) of type Eq(X) that contains definitions for equal : X → X → X
- creates two concrete instances (dictionaries) of type Eq(Int) and Eq(Char),
- creates a function that given a dictionary for EQ(X) builds a dictionary for List(X).

Mini-Haskell Example — Simple Classes and instances

Mini-Haskell class declarations and instance definitions

```
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 → X → X
- creates two concrete instances (dictionaries) of type Eq(Int) and Eq(Char),
- creates a function that given a dictionary for EQ(X) builds a dictionary for List(X).

Example Elaboration into explicit dictionaries

```
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 }
```

Becomes:

```
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 = primtEqChar }
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 eqX h₁ h₂ && equal (List X) (eqList X eqX) t₁ t₂ }
```

Example

Class Inheritance

Classes may themselves depend on other classes (called superclasses):

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

This declares a new class (dictionary) Ord (X) that contains a method Ord(X) that depends on a dictionary Eq(X) and contains a method lt : X → X → Bool.

This instance definition builds a dictionary Ord(Int) from the existing dictionary Eq 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 = (<) }
```

Mini Haskell Overloading

Then, we can define an overloaded function and use it:

```
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 t

let b = search Int 1 [1; 2; 3];;
```

This elaborates 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

let b = search Int ordInt 1 [1; 2; 3];;
```

Contents

Mini Haskell

Class and instance declarations are restricted to the toplevel. Their scope is the whole program.

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

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

We restrict to single parameter classes.

Mini Haskell

Source programs p are of the form:

$$
\begin{array}{lll}
p ::= H, \dots H\ H\ h\ \dots\ h\ t & P ::= \mathsf{K}\ X & Q ::= \mathsf{K}\ T \\
H ::= \mathsf{class}\ \vec{P} \Rightarrow \mathsf{K}\ X\ \{\rho\} & \rho ::= \varnothing \mid \rho; u : T & \\
h ::= \mathsf{inst}\ \forall \vec{X}. \vec{P} \Rightarrow \mathsf{K}\ (\mathsf{F}\ \vec{X})\ \{r\} & r ::= \varnothing \mid r; u = t &
\end{array}
$$

Letter u ranges over overloaded symbols.

Class constructors K may appear in Q but not in T. Only regular type constructors F may appear in T.

The sequence $\vec{P}$ in class and instance definitions is called a typing context. Each clause $\vec{P}$ is of the form $\mathsf{K}'\ X'$ and can be read as an assumption *given a dictionary of type X...*

The restriction to types of the form $\mathsf{K}'\ X'$ in typing contexts and class declarations, and to types of the form $\mathsf{K}'\ (\mathsf{F}'\ \vec{X}')$ in instances are for simplicity. Generalization are discussed later.

Target language

The target language is System F with record types, let-bindings, and let-rec.

Record types are provided as data types. Record labels are overloaded identifiers u.

We may use overloaded symbols as variables. This amounts to reserving a subset of variables x_u indexed by overloaded symbols, but just writing u as a shortcut for x_u.

We use letter s instead of t for elaborated terms, to distinguish them from source terms, but they are really terms of F.

Target language

We use $\Rightarrow$ as an annotated variant of $\rightarrow$.

Polymorphic types in the target language are of the form:

$$\begin{aligned} S &::= \forall \vec{X}. Q \Rightarrow \ldots Q \Rightarrow R \\ R &::= T \mid Q \end{aligned}$$

For convenience, we may still write $Q_1, \ldots Q_n \Rightarrow T$ to mean $Q_1 \Rightarrow \ldots Q_n \Rightarrow T$.

In the target language, record fields are of type R.

Class declarations

$$\text{class } \mathsf{K}_1\ X, \ldots\ \mathsf{K}_n\ X \Rightarrow \mathsf{K}\ X\ \{\rho\}$$

A class declaration H defines a class constructor K. 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.

We say that classes K_i's are superclasses of K and we write $\mathsf{K}_i \prec \mathsf{K}$. They must have been previously defined. so that the relation $\prec$ is acyclic. (Each dictionary of class K will then contain a sub-dictionary for each superclass K_i.)

We also request that all K_i's are independent, *i.e.* there does not exists i and j such that $\mathsf{K}_j \prec \mathsf{K}_i$. (If $\mathsf{K}_j \prec \mathsf{K}_i$, then K_i dictionary already contains a sub-dictionary for K_j, to which K has access via K_i so it does need not have one itself.)

Class declarations

$$\text{class } \mathsf{K}_1\, X, \ldots\, \mathsf{K}_n\, X \Rightarrow \mathsf{K}\, X\ \{\rho\}$$

The row type ρ is of the form

$$u_1 : T_1, \ldots\, u_m : T_m$$

and declares *overloaded symbols* u_i (also called *methods*) of class K. An overloaded must not be declared twice in the same class and must be declared only in one class.

Types T_i's must be closed with respect to X.

Each dictionary of class will contain a definition for each of its method.

Class declarations

Elaboration

$$\text{class } \mathsf{K}_1\ X, \ldots\ \mathsf{K}_n\ X \Rightarrow \mathsf{K}\ X\ \{\rho\}$$

Its elaboration consists in a record type declaration to represent the dictionary and the definition of accessors for each field of the record.

The row ρ only lists methods. We extend it with all sub-dictionaries fields and define ρ^{K} to be $u^{\mathsf{K}}_{\mathsf{K}_1} : \mathsf{K}_1\ X, \ldots\ u^{\mathsf{K}}_{\mathsf{K}_n} : \mathsf{K}_n\ X, \rho$. We introduce:

- a record type definition $\mathsf{K}\ X \approx \{\rho^{\mathsf{K}}\}$, and
- for each $u : R_u$ in ρ^{K},
  - let s_u be $\Lambda X.\lambda z{:}\mathsf{K}\ X.\,(x.u)$.
  - let S_u be $\forall X.\,\mathsf{K}\ X \Rightarrow R_u$, *i.e.* the type of t_u
  - let C_u be the program context $\text{let } u : S_u = s_u \text{ in } [\,]$.

Then $[\![H]\!]$ is the composition of all program contexts C_u when $u : \rho_u$ ranges in ρ.

Class declarations

Elaboration

The elaboration $[\![\vec{H}]\!]$ of the sequence of class definitions $\vec{H}$ is the composition of the elaboration of each.

Record type definitions are collected in the program prelude.

We write $\Gamma_{\vec{H}}$ for the typing context composed of $(u : S_u)$ for all u appearing in $\vec{H}$ (which is well-formed).

Instance declarations

$$\text{inst } \forall \vec{Y}.\, \vec{P} \Rightarrow \mathsf{K}\,(\mathsf{F}\;\vec{Y})\;\{r\}$$

This defines an instance of a class K. The clauses $\vec{P}$ are called a *typing context*. They describe the dictionaries that must be available on type parameters $\vec{Y}$ so that the dictionary $\mathsf{K}\,(\mathsf{F}\;\vec{Y})$ can be built.

The typing context $\vec{P}$ is not related to the superclasses of the class K: For example, class K′ may be a superclass of K, so the creation of the dictionary $\mathsf{K}\,X$ requires a dictionary $\mathsf{K}'\,X$ while an instance declaration K F (where F is nullary) need not (and in fact cannot, as the syntax does not allow it) request a dictionary of type K′ F.

Indeed, either such a dictionary can already be built, hence the instance does not require it, or it will never be possible to build one (remember that instance definitions are recursively defined so all of them are already visible) and the program must be rejected.

Instance declarations

$$\text{inst } \forall \vec{Y}.\, K_1\ Y_1, \ldots\ K_p\ Y_p \Rightarrow K\ (F\ \vec{Y})\ \{r\}$$

In summary, the typing context describes dictionaries that cannot yet be built because they depend on some unknown type Y in $\vec{Y}$.

We assume that the typing context $K_1\ Y_1, \ldots\ K_p\ Y_p$ is such that

- *each* Y_i is in $\vec{Y}$
- Y_i and Y_j *may be equal, except* if $K_i \prec K_j$ *or* $K_j \prec K_i$ *or* $K_i = K_j$.

The reason for the latter condition is, as for class declarations, that it would be useless to require both dictionaries $K_i\ Y$ and $K_j\ Y$ when, *e.g.*, $K_i \prec K_j$ since the former is contained in the latter.

Such typing contexts are said to be *canonical*.

Instance declarations

$$\text{inst } \forall \vec{Y}.\, K_1\ Y_1, \ldots K_p\ Y_p \Rightarrow K\,(F\ \vec{Y})\ \{r\}$$

This instance definition h is elaborated into a triple (z_h, s_h, S_h) where z_h is an identifier to refer to the elaborated body s_h of type S_h.

The type S_h is $\forall \vec{Y}. K_1\ Y_1 \Rightarrow \ldots K_p\ Y_p \Rightarrow K\,(F\ \vec{Y})$

The expression s_h builds a dictionary of type $K\,(F\ \vec{Y})$, given p dictionaries (which may be none) of respective types $K_1\ Y_1, \ldots K_p\ Y_p$:

$$\lambda(z_1 : K_1\ X_1) \ldots . \lambda(z_p : K_p\ X_p).\, \{u^{K}_{K'_1} = q_1, \ldots u^{K}_{K'_n} = q_n,\ u_1 = s_1, \ldots u_m = s_m\}$$

The types of fields are as prescribed by the class definition K:

- q_i is a dictionary of type $K'_i\ (F\ \vec{Y})$
- s_i is the elaboration of t_i where r is $u_1 = t_1, \ldots u_m = t_m$.

(For clarity, we write z instead of x when it binds a dictionary.)

Elaboration of whole programs

The elaboration of all class instances $[\![\vec{h}]\!]$ is the program context

$$\text{let rec } (\vec{z}_h : \vec{S}_h) = \vec{s}_h \text{ in } [\,]$$

The elaboration of the whole program $\vec{H}\,\vec{h}\,t$ is

$$[\![\vec{H}\,\vec{h}\,t]\!] \triangleq \text{let } \vec{u} : \vec{S}_u = \vec{s}_u \text{ in let rec } (\vec{z}_h : \vec{S}_h) = \vec{s}_h \text{ in } s$$

Hence, the expression s and all expressions s_h are typed (and elaborated) in the environment Γ_0 equal to $\Gamma_{\vec{H}}, \Gamma_{\vec{h}}$ where

- [reminder] $\Gamma_{\vec{H}}$ declares functions to access components of dictionaries (both sub-dictionaries and definitions of overloaded symbols).
- $\Gamma_{\vec{h}}$ equal to $(\vec{z}_h : \vec{S}_h)$ declares functions to build dictionaries.

Elaboration of expressions

The elaboration of expressions is defined by a judgment

$$\Gamma \vdash t \rightsquigarrow s : S$$

where Γ is a System F typing context, t is the source expression, s is the elaborated expression and S its type in Γ. In particular, $\Gamma \vdash t \rightsquigarrow s : S$ implies $\Gamma \vdash s : S$ in F.

We write q for dictionary terms, *i.e.* the following subset of F terms:

$$q ::= u \mid z \mid q\ T \mid q\ q$$

(u and z are just particular cases of x)

The elaboration of dictionaries is the judgment $\Gamma \vdash q : S$ which is just typability in System F—but restricted to dictionary expressions.

Elaboration of expressions

Most rules just wrap the elaboration of their subexpressions:

Var
$$\frac{x : S \in \Gamma}{\Gamma \vdash x \rightsquigarrow x : S}$$

Inst
$$\frac{\Gamma \vdash t \rightsquigarrow s : \forall X.S}{\Gamma \vdash t\ T \rightsquigarrow s\ T : [X \mapsto T]S}$$

Gen
$$\frac{\Gamma, X \vdash t \rightsquigarrow s : S}{\Gamma \vdash \Lambda X.t \rightsquigarrow \Lambda X.s : \forall X.S}$$

Let
$$\frac{\Gamma \vdash t_1 \rightsquigarrow s_1 : S \qquad \Gamma, x : S \vdash t_2 \rightsquigarrow s_2 : T}{\Gamma \vdash \text{let } x : S = t_1 \text{ in } t_2 \rightsquigarrow \text{let } x : S = s_1 \text{ in } s_2 : T}$$

App
$$\frac{\Gamma \vdash t_1 \rightsquigarrow s_1 : T_2 \to T_1 \qquad \Gamma \vdash t_2 \rightsquigarrow s_2 : T_2}{\Gamma \vdash t_1\ t_2 \rightsquigarrow s_1\ s_2 : T_1}$$

Abs
$$\frac{\Gamma, x : T' \vdash t \rightsquigarrow s : T}{\Gamma \vdash \lambda x{:}T'.t \rightsquigarrow \lambda x{:}T'.s : T' \to T}$$

In rule Let, we require S to be canonical, *i.e.* of the form $\forall \vec{X}.\vec{P} \Rightarrow R$ with $\vec{P}$ is itself empty or canonical. See also this restriction.

Rules App and Abs do not applies to overloaded expressions of type S.

Elaboration of overloaded expressions

The interesting rules are the elaboration of missing abstractions or applications of dictionaries.

OAbs
$$\frac{\Gamma, x : Q \vdash t \rightsquigarrow s : S \qquad x \# t}{\Gamma \vdash t \rightsquigarrow \lambda x{:}Q.\, s : Q \Rightarrow S}$$

OApp
$$\frac{\Gamma \vdash t \rightsquigarrow s : Q \Rightarrow S \qquad \Gamma \vdash q : Q}{\Gamma \vdash t \rightsquigarrow s\, q : S}$$

Rule Oabs pushes in the context Γ dictionary abstractions as prescribed by the expected type. These might be used (in addition to dictionary accessors and instance definitions already in Γ) to elaborate dictionaries as described by the premise $\Gamma \vdash q : Q$ of rule OAbs.

The judgment $\Gamma \vdash q : Q$ is just well-typedness in System F for dictionary expressions. There is an algorithmic reading of the rule, described further, where Γ and Q are given and q is inferred.

By construction, elaboration produces well-typed expressions: that is $\Gamma_0 \vdash t \rightsquigarrow s : T$ implies that is $\Gamma_0 \vdash s : T$.

Resuming the elaboration

An instance declaration h of the form:

$$\mathsf{inst}\ \forall \vec{Y}.\ \mathsf{K}_1\ Y_1, \ldots\ \mathsf{K}_p\ Y_p \Rightarrow \mathsf{K}\ \vec{T}\ \{u_1 = t_1, \ldots l; u_m = t_m\}$$

is translated into

$$\lambda(z_1 : \mathsf{K}_1\ X_1) \ldots . \lambda(z_p : \mathsf{K}_p\ X_p).\ \{u^{\mathsf{K}}_{\mathsf{K}'_1} = q_1, \ldots\ u^{\mathsf{K}}_{\mathsf{K}'_n} = q_n,\ u_1 = s_1, \ldots\ u_m = s_m\}$$

where:

- $u^{\mathsf{K}}_{\mathsf{K}'_i} : T_i$ are the superclasses fields
- Γ_h is $\vec{Y}, \mathsf{K}_1\ Y_1, \ldots\ \mathsf{K}_p\ Y_p$
- $\Gamma_0, \Gamma_h \vdash q_i : T_i$
- $\Gamma_0, \Gamma_h \vdash t_i \rightsquigarrow s_i : T_i$

Finally, given the program p equal to $\vec{H}\ \vec{h}\ t$, we elaborate t as s such that $\Gamma_0 \vdash t \rightsquigarrow s : \forall \bar{X}.T$.

Notice that $\forall \bar{X}.T$ is an unconstrained type scheme. Why?

Resuming the elaboration

Otherwise, s could elaborate into an abstraction over dictionaries, *i.e.* it would be a value and never applied!

Where else should we be careful that the *intended* semantics is preserved?

Let-monomorphization

Otherwise, s could elaborate into an abstraction over dictionaries, *i.e.* it would be a value and never applied!

Where else should we be careful that the *intended* semantics is preserved?

In a call-by-value setting, we must not elaborate applications into abstractions, since it would delay and perhaps duplicate the order of evaluations.

For that purpose, we must restrict rule Let so that either S is of the form $\forall \bar{X}.T$ or t_1 is a value or a variable.

What about call-by-name? and Haskell?

Let-monomorphization

In call-by-name, an application is not evaluated until it is needed. Hence adding an abstraction in front of an application should not change the evaluation order $t_1\ t_2$.

We must in fact compare:

$$\text{let } x_1 = \lambda y.\,\text{let } x_2 = v_1\ v_2 \text{ in } t_2 \text{ in } [x_1 \mapsto x_1\ q]t_1 \qquad \textbf{(1)}$$
$$\text{let } x_1 = \text{let } x_2 = \lambda y.\,v_1\ v_2 \text{ in } [x_2 \mapsto x_2\ q]t_2 \text{ in } t_1 \qquad \textbf{(2)}$$

The order of evaluation of $v_1\ v_2$ is preserved.

However, Haskell 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 t_2 in (2).

Let-monomorphization

The final result will still be the same in both cases because Haskell is pure, but the intended semantics is changed regarding the efficiency.

Hence, Haskell may also use monomorphization in this case. This is a delicate design choice

(Of course, monomorphization reduces polymorphism, hence the set of typable programs.)

Resuming the elaboration — Sources of failures

The elaboration may fail for several reasons:

- The input expression does not obey one of the restrictions we have requested.
- A typing may occur during elaboration of an expression.
- The impossibility to build appropriate dictionaries.

If elaboration fails, the program p is rejected, indeed.

When elaboration succeeds

When the elaboration of p succeeds it returns $[\![p]\!]$, well-typed in F.

Then, the semantics of p is given by that of $[\![p]\!]$.

When elaboration succeeds

When the elaboration of p succeeds it returns $[\![p]\!]$, well-typed in F.

Then, the semantics of p is given by that of $[\![p]\!]$.

Hum...

When elaboration succeeds

When the elaboration of p succeeds it returns $[\![p]\!]$, well-typed in F.

Then, the semantics of p is given by that of $[\![p]\!]$.

Hum... Although terms are explicitly-typed, their elaboration may not be unique! Indeed, they might be several ways to build dictionaries of some given type (see below for details).

In the worst case, a source program may elaborate to completely unrelated programs. In the best case, all possible elaborations are *equivalent* programs and we say that the elaboration is *coherent*: the programs has a deterministic semantics given by elaboration.

But what does it mean for programs be equivalent?

On program equivalence

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 would reduce to the same value whenever applied to the same 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.

On program equivalence

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

Sufficient conditions for coherence

Since terms are explicitly typed, the only source of non-determinism is the elaboration of dictionaries.

One way to ensure coherence is that two dictionaries *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.

Hint: Here is a simple intuition. Consider an occurrence of an overloaded identifier u in a context C that is elaborated into $u\,\vec{T}\,q$ where q may not be unique, but of some type $K\,T$ determined by C. During evaluation, this expression will eventually reduce to $(\lambda z{:}K\,T'.z.u)\,q_v$ where s_u is the accessors for u, q_v is a dictionary value of type T' and T' is a specialization of T that is fully determined by C and, in particular, independent of the choice of q. Since there is a unique dictionary q_v of type $K\,T'$, the actual code executed for this occurrence of u is $q_v.u$ and independent of the possible choices for q.

Elaboration of dictionaries

Elaboration of dictionaries is just typechecking in System F.

More precisely, it infers a dictionary q given Γ and Q so that $\Gamma \vdash q : Q$ holds.

The relevant subset of rules for dictionary expressions are:

$$\text{D-OVar}\quad \frac{u : S \in \Gamma}{\Gamma \vdash u : S} \qquad \text{D-Inst}\quad \frac{\Gamma \vdash s : \forall X.S}{\Gamma \vdash s\,T : [X \mapsto T]S} \qquad \text{D-Var}\quad \frac{z : Q \in \Gamma}{\Gamma \vdash z : Q}$$

$$\text{D-App}\quad \frac{\Gamma \vdash q_1 : Q_1 \Rightarrow Q_2 \qquad \Gamma \vdash q_2 : Q_1}{\Gamma \vdash q_1\,q_2 : Q_2}$$

Can we give a type-directed presentation?

Elaboration of dictionaries

Elaboration is driven by the type of the expected dictionary and the bindings available in the typing environment, which may be:

- dictionary constructors x^h given by instance definitions;
- dictionary accessors u^K given by class declarations;
- dictionary arguments, given by the local typing context.

Hence, the typing rules may be factorized as follows:

D-OVar-Inst
$$\frac{x : \forall \vec{Y}. P_1 \Rightarrow \ldots P_n \Rightarrow K\,(F\,\vec{Y}) \in \Gamma \qquad \Gamma \vdash q_i : [\vec{Y} \mapsto \vec{T}]P_i}{\Gamma \vdash x\,\vec{T}\,\vec{q} : K\,(F\,\vec{T})}$$

D-Proj
$$\frac{u : \forall X. K'\,X \Rightarrow K\,X \in \Gamma \qquad \Gamma \vdash q : K'\,T}{\Gamma \vdash z\,T\,q : K\,T}$$

D-Var
$$\frac{z : K\,X \in \Gamma}{\Gamma \vdash z : K\,X}$$

Elaboration of dictionary values

Dictionary values are typed in Γ_0, which does not contain free type variables, hence, only the first rule applies:

D-OVar-Inst

$$\frac{x : \forall \vec{Y}. P_1 \Rightarrow \ldots P_n \Rightarrow \mathsf{K}\,(\mathsf{F}\,\vec{Y}) \in \Gamma \qquad \Gamma \vdash q_i : [\vec{Y} \mapsto \vec{T}] P_i}{\Gamma \vdash u\,\vec{T}\,\vec{q} : \mathsf{K}\,(\mathsf{F}\,\vec{T})}$$

This rule for the judgment $\Gamma \vdash q : T$ can be read as an algorithm where Γ and Q are inputs (and Γ is constant) and q is an output.

Provided there is no choice in finding $x : \forall \vec{Y}. P_1 \Rightarrow \ldots P_n \Rightarrow \mathsf{K}\,(\mathsf{F}\,\vec{Y})$ in Γ.

Since each such clause is coming from an instance definition h, their is no choice in the application of this rule if *instance definitions never overlap*.

This assumption ensures coherence.

Overlapping instances

Two instances inst $\forall \vec{Y}_i . \vec{P} \Rightarrow \mathsf{K}\,(\mathsf{F}_i\ \vec{Y}_i)\ \{r_i\}$ for i in $\{1, 2\}$ of a class K overlap if the type schemes $\forall \vec{Y}_i . \mathsf{K}\,(\mathsf{F}_i\ \vec{T}_i)$ have a common instance, *i.e.* in the present setting, if F_1 and F_2 are equal.

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

Elaboration of dictionary arguments

Dictionary *expressions*, as opposed to dictionary values, will also be built by *extracting* dictionaries from other dictionaries.

Why?

Elaboration of dictionary arguments

Dictionary expressions, as opposed to dictionary values, will also be built by extracting dictionaries from other dictionaries.

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 (and must, as we forbid to pass both a class and one of its super class dictionary simultaneously) be extracted.

Technically, they are typed in an extension of the typing context Γ_0 which may contain typing assumptions $z : K' \, Y$ about dictionaries received as arguments. Hence rules D-Proj and D-Var may also apply.

Elaboration of dictionary arguments

The elaboration of dictionaries uses the three rules:

D-OVar-Inst

$$\frac{x : \forall \vec{Y}. P_1 \Rightarrow \ldots P_n \Rightarrow \mathsf{K}\,(\mathsf{F}\,\vec{Y}) \in \Gamma \qquad \Gamma \vdash q_i : [\vec{Y} \mapsto \vec{T}]P_i}{\Gamma \vdash x\,\vec{T}\,\vec{q} : \mathsf{K}\,(\mathsf{F}\,\vec{T})}$$

D-Proj

$$\frac{u : \forall X. \mathsf{K}'\,X \Rightarrow \mathsf{K}\,X \in \Gamma \qquad \Gamma \vdash q : \mathsf{K}'\,T}{\Gamma \vdash z\,T\,q : \mathsf{K}\,T}$$

D-Var

$$\frac{z : \mathsf{K}\,X \in \Gamma}{\Gamma \vdash z : \mathsf{K}\,X}$$

They can be read as a backtracking algorithm.

Elaboration of dictionary arguments Termination

The proof search always terminates, since premises have smaller Q than the conclusion when using the lexicographic order of first the height of T, then the reverse order of class inheritance.

If no rule applies, we fail. If rule D-Var applies, the derivation ends with success.

If rule D-Proj applies, the premise is called with a smaller problem since the height is unchanged and $\mathsf{K}' \vec{T}$ with $\mathsf{K}' \prec \mathsf{K}$.

If D-Ovar-Inst applies, the premises are called at type $\mathsf{K}_i\, T_j$ where T_j is subtype of $\vec{T}$, hence of a strictly smaller height.

Elaboration of dictionary arguments Non determinism

For instance, in the introduction, we defined two instances eqInt and ordInt, while the later contains an instance of the former.

Hence, a dictionary of type eqInt may be obtained:

- 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 may recover determinism by fixing a simple and sensible strategy for elaboration. Restrict the use of rule D-Proj to cases where Q is P—when D-OVar-Inst does not apply. However, the extra flexibility is harmless and perhaps useful freedom for the compiler.

Typing dictionaries

Example

In the introductory example Γ_0 is:

$$\begin{array}{lcl}
\mathsf{equal} & : & \forall X. Eq\ X \Rightarrow X \rightarrow X \rightarrow Bool, \\
\mathsf{eqInt} & : & EqInt \\
\mathsf{eqList} & : & \forall X. Eq\ X \Rightarrow Eq\ (List\ x) \\
\mathsf{eqOrd} & : & \forall X. Ord\ X \Rightarrow Eq\ X, \\
\mathsf{lt} & : & \forall X. Ord\ X \Rightarrow X \rightarrow X \rightarrow Bool,
\end{array}$$

When typing search, we have to infer a dictionary for equal in the context $\Gamma, X, \mathsf{ordX} : Ord\ X$.

We find:

$$\Gamma, X, \mathsf{ordX} : Ord\ X \vdash \mathsf{eqOrd}\ \ \mathsf{X}\ \ \mathsf{ordX} : \mathsf{Eq}\ \ \mathsf{X}$$

Contents

What can be left implicit?

Class declarations?

What can be left implicit?

Class declarations must remain explicit:

- They define the structure of dictionaries: a record type definition and its accessors.
- They define the type scheme of overloaded symbols and the class they belong to.

The type of instance declarations?

What can be left implicit?

Class declarations must remain explicit:

- They define the structure of dictionaries: a record type definition and its accessors.
- They define the type scheme of overloaded symbols and the class they belong to.

The type of instance declarations must also remain explicit:

- These are polymorphic recursive definitions, hence their types are mandatory.

What can be left implicit?

Class declarations must remain explicit:

- They define the structure of dictionaries: a record type definition and its accessors.
- They define the type scheme of overloaded symbols and the class they belong to.

The type of instance declarations must also remain explicit:

- These are polymorphic recursive definitions, hence their types are mandatory.

However, all core language expressions (in instance declarations and the final one) can be left explicit.

Example

```
class Eq (X)     { equal : X → X → Bool }
inst Eq (Int)   { equal = (==) }
inst Eq (Char) { equal = (==) }
inst          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                                  =
         λ(x    ) λ(l          )
    match l with [] → false | h::t → equal    x h || search    x t

let b = search Int 1 [1; 2; 3];;
```

Example

```
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 : List X) λ(l₂ : List X) match l₁, l₂ with
           | [],[] → true | [],_ | [],_ → false
           | h₁::t₁, h₂::t₂ → eq X h₁ h₂ && eq (List X) t₁ t₂ }

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

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

let b = search Int 1 [1; 2; 3];;
```

Type inference

The idea is to see dictionary types $\mathsf{K}\, T$, which can only appear in type schemes and not in types, as a type constraint to mean *"there exists a dictionary of type $\mathsf{K}\, X$"*.

Just read $\forall \vec{X}. \vec{P} \Rightarrow T$ a the constraint type scheme $\forall \vec{X}[\vec{P}].T$.

We extend constraints with dictionary predicates:

$$C ::= \ldots \mid \mathsf{K}\, T$$

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 Γ_0 (that contains all class and instance declarations), *i.e.* formally, if there exists a dictionary expression q such that $\Gamma_0 \vdash q : \mathsf{K}\, \mathbf{t}$.

The satisfiability of class-membership constraints is thus:

$$\frac{\mathsf{K}\, \phi T}{\phi \vdash \mathsf{K}\, T}$$

Reasoning with class-membership constraints

For every class declaration $\mathsf{class}\ \mathsf{K}_1\ X_1, \ldots\ \mathsf{K}_n\ X_n \Rightarrow \mathsf{K}\ X\ \{\rho\}$,

$$\mathsf{K}\ X \Vdash \mathsf{K}_1\ X_1 \wedge \ldots\ \mathsf{K}_n\ X_n \tag{1}$$

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

Proof of (1).

Reasoning with class-membership constraints

For every class declaration class $\mathsf{K}_1\ X_1, \ldots\ \mathsf{K}_n\ X_n \Rightarrow \mathsf{K}\ X\ \{\rho\}$,

$$\mathsf{K}\ X \Vdash \mathsf{K}_1\ X_1 \wedge \ldots\ \mathsf{K}_n\ X_n \tag{1}$$

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

Proof of (1). Assume $\phi \vdash \mathsf{K}\ X$, *i.e.* $\Gamma_0 \vdash q : \mathsf{K}\ (\phi X)$ for some dictionary q.

From the class declaration, we know that $\mathsf{K}\ X$ is a record type definition that contains fields $u^{\mathsf{K}}_{\mathsf{K}_i}$ of type $\mathsf{K}_i\ X_i$. Hence, the dictionary value q contains field values of types $\mathsf{K}_i\ (\phi X)$. Therefore, we have $\phi \vdash \mathsf{K}_i\ X$ for all i in $1..n$, which implies $\phi \vdash \mathsf{K}_1\ X \wedge \ldots\ \mathsf{K}_n\ X$. □

Reasoning with class-membership constraints

For every instance definition inst $\forall \vec{Y}. K_1\ Y_1, \dots K_p\ Y_p \Rightarrow K\ (F\ Y)\ \{r\}$,

$$K\ (F\ \vec{Y}) \equiv K_1\ Y_1 \wedge \dots K_p\ Y_p \tag{2}$$

This rule allows to decompose all class constraints into simple constraints of the form $K\ X$.

Proof of (2) ($\dashv\Vdash$ direction). *Assume $\phi \vdash K_i\ Y_i$. There exists dictionaries q_i such that $\Gamma_0 \vdash q_i : K_i\ (\phi Y_i)$. Hence, $\Gamma_0 \vdash x_h\ \vec{Y}\ q_1\ \dots\ q_p : K\ (F\ (\phi \vec{Y}))$,* i.e. $\phi \vdash K\ (F\ (\phi \vec{Y}))$.

($\Vdash$ direction). *Assume, $\phi \vdash K\ (F\ (\phi \vec{Y})$.* i.e. *there exists a dictionary q such that $\Gamma_0 \vdash q : K\ (F\ \phi \vec{Y})$. By non-overlapping of instance declarations, the only way to build such a dictionary is by an application of x_h. Hence, q must be of the form $x_h\ \vec{Y}\ q_1\ \dots\ q_p$ with $\Gamma_0 \vdash q_i : K_i\ (\phi Y_i)$, that is, $\phi \vdash K_i\ Y_i$ for every i, which implies $\phi \vdash K_1\ Y_1 \wedge \dots K_p\ Y_p$.*

Reasoning with class-membership constraints

For every instance definition inst $\forall \vec{Y}.\, K_1\ Y_1, \ldots K_p\ Y_p \Rightarrow K\ (F\ Y)\ \{r\}$,

$$K\ (F\ \vec{Y}) \equiv K_1\ Y_1 \wedge \ldots K_p\ Y_p \qquad (2)$$

This rule allows to decompose all class constraints into simple constraints of the form $K\ X$.

Notice that the equivalence 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 overlapping of instances were allowed, the $\Vdash$ direction would not hold. Then, the rewriting rule:

$$K\ (F\ \vec{Y}) \longrightarrow K_1\ Y_1 \wedge \ldots K_p\ Y_p$$

would still be sound (the right-hand side entails the left-hand side, and thus type inference would infer sound typings), i.e. but not complete (type inference could miss some typings).

Reasoning with class-membership constraints

For every class K and type constructor F for which there is no instance of K,

$$K\,(F\,\vec{Y}) \equiv \text{false} \qquad \textbf{(3)}$$

This rule allows failure to be reported as soon as constraints of the form $K\,(F\,\vec{T})$ appear and there is not instance of K for F.

Proof of (3). *The $\dashv\!\vdash$ direction is a tautology, so it suffices to prove the $\Vdash$ direction. By contradiction. Assume $\phi \vdash K\,(F\,\vec{Y})$. This implies the existence of a dictionary q such that $\Gamma_0 \vdash q : K\,(F\,(\phi\vec{Y}))$. Then, there must be some x_h in Γ whose type scheme is of the form* $\forall\vec{Y}.\vec{P} \Rightarrow K\,(F\,\vec{Y})$, i.e. *there must be an instance of class K for F.*

Notice that this rule does not work in an open world assumption. The rewriting rule

$$K\,(F\,\vec{Y}) \longrightarrow \text{false}$$

would still remain sound but incomplete.

Typing constraints

Constraint generation is unchanged.

$$\begin{array}{rcl}
(\!|x|\!) & = & \forall X[x \preceq X].X \\
(\!|\lambda x.a|\!) & = & \forall X_1 X_2\big[\mathrm{let}\ x : X_1\ \mathrm{in}\ (\!|a|\!) \preceq X_2\big].X_1 \to X_2 \\
 & & \mathrm{if}\ X_1, X_2 \# a \\
(\!|a_1\ a_2|\!) & = & \forall X_1 X_2\big[(\!|a_1|\!) \preceq X_1 \to X_2 \wedge (\!|a_2|\!) \preceq X_1\big].X_2 \\
 & & \mathrm{if}\ X_1, X_2 \# a_1, a_2 \\
(\!|\mathrm{let}\ x = w\ \mathrm{in}\ a|\!) & = & \forall X\big[\mathrm{let}\ x : (\!|w|\!)\ \mathrm{in}\ (\!|a|\!) \preceq X\big].X \\
 & & \text{if } w \text{ is a value or a variable}
\end{array}$$

A constraint type scheme can always be decomposed into one of the form $\forall \bar{X}[P_1 \wedge P_2].T$ where $\mathrm{ftv}(P_1) \in \bar{X}$ and $\mathrm{ftv}(P_2) \# \bar{X}$.

The constraints P_2 can then be extruded in the enclosing context if any, so we are in general left just with P_1.

Remember that $\Gamma \vdash a : T$ iff $\mathrm{def}\ \Gamma\ \mathrm{in}\ (\!|a|\!) \preceq T$.

Checking well-typedness

To check well-typedness of the program p equal to $\vec{H}\ \vec{h}\ a$, we must check that: each expression a_i^h and the expression a are well-typed, in the environment used to elaborate them:

This amounts to checking:

- $\Gamma_0, \Gamma_h \vdash a_i^h : T_i^h$ where T_i^h is given.
  Thus, we check that the constraints $\mathit{def}\ \Gamma_0, \Gamma_h \text{ in } (\!| a_i^h |\!) \leq T_i^h \equiv \text{true}$.
- $\Gamma_0 \vdash a : T$ for some T.
  Thus, we check that $\mathit{def}\ \Gamma_0 \text{ in } \exists X. (\!| a |\!) \leq X \equiv \text{true}$.

However, ...

Checking well-typedness

To check well-typedness of the program p equal to $\vec{H}\ \vec{h}\ a$, we must check that: each expression a_i^h and the expression a are well-typed, in the environment used to elaborate them:

This amounts to checking:

- $\Gamma_0, \Gamma_h \vdash a_i^h : T_i^h$ where T_i^h is given.
  Thus, we check that the constraints $\text{def}\, \Gamma_0, \Gamma_h \text{ in } (\!| a_i^h |\!) \leq T_i^h \equiv \text{true}$.
- $\Gamma_0 \vdash a : T$ for some T.
  Thus, we check that $\text{def}\, \Gamma_0 \text{ in } \exists X. (\!| a |\!) \leq X \equiv \text{true}$.

However, . . . Typechecking is not sufficient!

Type reconstruction should also return an explicitly-typed term t than can then be elaborated into s. That is $\Gamma \vdash a \rightsquigarrow t : T$.

Type reconstruction

The resolution strategy for constraints may be tuned to delay *garbage collection* of solved constraints, so that the explicitly-typed term can be read back from the constraint in solved form.

For example, original let-constraints can be kept using the rule

$$\text{let } x : S \text{ in } C \longrightarrow \text{let } x : S \text{ in def } y : S \text{ in } C$$

to propagate the constraint type scheme S inside C, but keeping the original binding for elaboration. (Of course, we then also disallow let-constraints to be discarded.)

Constraints in canonical forms now contain let-bindings $\text{let } x : S \text{ in } C$ where S is itself in canonical form and x does not appear in C.

From a constraint in canonical form, we can rebuild a term where all let-bindings and arguments of functions are explicitly typed, from which type applications for let-bound variables can be easily rebuilt (further instrumentation of the constraints could also remember those).

Type reconstruction

Principal

While type inference infers *principal types*, there may be several *explicitly typed terms of the same type, even if we restrict* instantiation to variables and generalization to let-bound expressions.

An example (in ML)?

Type reconstruction

Principal

While type inference infers *principal types*, there may be several explicitly typed terms of the same type, even if we restrict instantiation to variables and generalization to let-bound expressions.

An example (in ML)

$$\text{let } x = \lambda y.\, y \text{ in } x\ 1$$

can be elaborated into either one of:

$$\text{let } x : \text{int} \to \text{int} = \lambda y{:}\text{int}.\, y \text{ in } x\ 1$$
$$\text{let } x : \forall X. X \to X = \Lambda x. \lambda x{:}\text{int}.\, x \text{ in } x \text{ int } 1$$

Which one is better?

Type reconstruction — Principal

While type inference infers *principal types*, there may be several explicitly typed terms of the same type, even if we restrict instantiation to variables and generalization to let-bound expressions.

An example (in ML)

$$\text{let } x = \lambda y.\, y \text{ in } x\ 1$$

can be elaborated into either one of:

$$\text{let } x : \text{int} \to \text{int} = \lambda y{:}\text{int}.\, y \text{ in } x\ 1$$
$$\text{let } x : \forall X.\, X \to X = \Lambda x.\lambda x{:}\text{int}.\, x \text{ in } x\ \text{int}\ 1$$

Which one is better?

Monomorphic terms can be compiled more efficiently, so removing useless polymorphism may be useful.

It is best is to leave the choice to the compiler.

Type reconstruction Principal

Let t and t' be two type reconstructions of a term a. We say that t is *more general* than t' if all let-bindings are assigned more general type schemes in t than in t', *i.e.*

> *for all decompositions of t into $C[\text{let } x : S = t_1 \text{ in } t_2]$, then there is a corresponding decomposition of t'* (i.e. *one where C and C' have the same erasure) as $C'[\text{let } x : S' = t_1' \text{ in } t_2']$ where S is more general than S'.*

A *type reconstruction is principal* if it is more general than any other type reconstruction of the same term.

Core ML admits principal type reconstructions.

Type reconstruction Non principal

Exercise

Find extensions of ML that do not have principal type reconstructions.

Type reconstruction — Non principal

Exercise

Find extensions of ML that do not have principal type reconstructions.

Possible answers:

- ML with modules.

Type reconstruction Non principal

Exercise

Find extensions of ML that do not have principal type reconstructions.

Possible answers:

- ML with modules.
  Signatures must have closed type schemes.
- ML with dynamic values.

Type reconstruction Non principal

Exercise

Find extensions of ML that do not have principal type reconstructions.

Possible answers:

- ML with modules.
  Signatures must have closed type schemes.
- ML with dynamic values.
  A dynamic value is a value package with its type. For example, it can then be stored on a persistent store and retrieved later in another session (by checking the dynamic value against its type).
  What is the problem?

Type reconstruction Non principal

Exercise

Find extensions of ML that do not have principal type reconstructions.

Possible answers:

- ML with modules.
  Signatures must have closed type schemes.
- ML with dynamic values.
  A dynamic value is a value package with its type. For example, it can then be stored on a persistent store and retrieved later in another session (by checking the dynamic value against its type).

  What is the problem? To be packed into a dynamic, the value must have a closed type scheme.

Type reconstruction Non principal

Exercise

Find extensions of ML that do not have principal type reconstructions.

Possible answers:

- ML with modules.
  Signatures must have closed type schemes.
- ML with dynamic values.
  A dynamic value is a value package with its type. For example, it can then be stored on a persistent store and retrieved later in another session (by checking the dynamic value against its type).
  What is the problem? To be packed into a dynamic, the value must have a closed type scheme.
  So what should be the type of $\lambda x. \mathit{dynamic}\, x$?

Type reconstruction (Minimal)

Question: Does ML admits least general type reconstructions?

Type reconstruction (Minimal)

Question: *Does ML admits least general type reconstructions?*

No, there are examples where there are two minimal incomparable type reconstructions and others with smaller and smaller type reconstructions but no smallest one.

Exercise: *Find examples of both kinds!*

Answer: *See [Bjørner, 1994]*

Type reconstruction — Principal elaboration

The constraint framework enforces the inference of principal types, since it transforms the original constraint into an *equivalent constraint*.

However, it does not enforce type reconstruction to be principal.

Why?

Type reconstruction Principal elaboration

The constraint framework enforces the inference of principal types, since it transforms the original constraint into an *equivalent constraint*.

However, it does not enforce type reconstruction to be principal.

Indeed, in a constraint $\exists X.C$, the existentially bound type variable X may be instantiated to *any* type that satisfies the constraint C and not necessarily the most general one.

Interestingly, however, the default *strategy* for constraint resolution always *returns principal type reconstructions*.

That is, variables are never arbitrarily instantiated, although this would be allowed by the specification.

Back to 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.

Coherence — Toplevel unresolved constraints

Thanks to the several restrictions on class declarations and instance definitions, the type system has principal constrained schemes (and principal typing reconstructions). However, this does not imply that there are principal *unconstrained* type schemes.

Indeed, assume that the principal constrained type scheme is $\forall X[\mathsf{K}\,X].X \to X$ and the typing environment contains two instances of $\mathsf{K}\,\mathsf{F}_1$ and $\mathsf{K}\,\mathsf{F}_2$ of class K. Constraint-free instances of this type scheme are $\mathsf{F}_1 \to \mathsf{F}_1$ and $\mathsf{F}_2 \to \mathsf{F}_2$ but $\forall X.X \to X$ is certainly not one.

Not only neither choice is principal, but the two choices would elaborate in expressions with different (non-equivalent) semantics.

We must fail in such cases.

Coherence — Toplevel unresolved constraints

This problem may appear while typechecking the final expression a in Γ_0 that request an unconstrained type scheme $\forall X. T$

It may also occur when typechecking the body of an instance definition, which requests an explicit type scheme $\forall \vec{X}[\vec{Q}]. T$ in Γ_0 or equivalently that request a type T in $\Gamma_0, \vec{X}, \vec{Q}$.

Coherence — Example of unresolved constraints

```
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 $\forall X[Num\ X].X$—and several classes are possible for $Num\ X$. The semantics of the program is undetermined.

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

The type of x is $\forall X[readable\ X].\mathrm{unit} \to X$—and several classes are possible for $readable\ X$. The program is rejected.

Coherence Inaccessible constraint variables

In the previous examples, the incoherence comes from the obligation to infer type schemes without constraints. A similar problem may occur with isolated constraints in a type scheme.

Assume, for instance, that the elaboration of $\text{let } x = a_1 \text{ in } a_2$ is $\text{let } x : \forall X[\mathsf{K}\ X].\,\text{int} \to \text{int} = s_1 \text{ in } s_2$.

All applications of x in s_2 will lead to an unresolved constraint $\mathsf{K}\ X$ since neither the argument nor the context of this application can determine the value of the type parameter X. Still, a dictionary of type $\mathsf{K}\ T$ must be given before s_1 can be executed.

Although x may not be used in s_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.

Coherence — The open-world view

When there is a single instance $\mathsf{K}\,\mathsf{F}$ for a class K that appears in an unresolved or isolated constraint $\mathsf{K}\,X$, 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

In the source of incoherence we have seen, some class constraint remained undetermined.

As noticed earlier some (usually arbitrary) less general elaboration would cover the problem—but the source program would remain incoherent.

Hence, in order to detect incoherent (i.e. ambiguous) programs it is essential that type reconstruction is principal.

Once a program has been checked coherent, *i.e.* with no undetermined constraint, based on a principal type reconstruction, can we still use another non principal type reconstruction for its elaboration?

On the importance of principal type reconstruction

In the source of incoherence we have seen, some class constraint remained undetermined.

As noticed earlier some (usually arbitrary) less general elaboration would cover the problem—but the source program would remain incoherent.

Hence, in order to detect incoherent (i.e. ambiguous) programs it is essential that type reconstruction is principal.

Once a program has been checked coherent, *i.e.* with no undetermined constraint, based on a principal type reconstruction, can we still use another non principal type reconstruction for its elaboration?

Yes, indeed, this will preserve the semantics.

This freedom may actually be very useful for optimizations.

On the importance of principal type reconstruction

Example

Consider the program

On the importance of principal type reconstruction

Example

Consider the program

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

Its principal type reconstruction is:

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

which *elaborates* into

On the importance of principal type reconstruction

Example

Consider the program

let twice = λ(x) x + x in twice (twice 1)

Its principal type reconstruction is:

let twice : all(X) [Num X] X → X = Λ(X) [Num X] λ(x) x + x in
twice Int (twice Int) 1

which *elaborates* into

let twice X numX = λ(x : X) x (plus numX) x in
twice Int numInt (twice Int numInt 1)

while, avoiding the generalization of twice, it would elaborate into:

let twice = λ(x : Int) x (plus numInt) x in twice (twice 1)

where moreover, the plus numInt can be statically reduced.

Overloading by return types

All previous ambiguous examples are overloaded by return types:

- $0 : X$.
  The value 0 has an overloaded type that is not constraint by the argument.
- $\mathit{read} : \mathit{desc} \to X$.
  The function read applied to some ground type argument will be under specified.

Odersky et al. [1995] suggested to prevent overloading by return types by requesting that overloaded symbols of a class $\mathsf{K}\,X$ have types of the form $X \to T$.

The above examples are indeed rejected by this definition.

Overloading by return types

In fact, disallowing overloading by return types suffices to ensures that all well-typed programs are coherent.

Moreover, untyped programs can then be given a semantics directly (which of course coincides with the semantics obtained by elaboration).

Many interesting examples of overloading fits in this schema.

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.

Contents

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 quite a lot of flexibility in the representation of dictionaries.

For example, we used record data-type definitions to represent dictionaries, but tuples would have been sufficient.

An alternative compilation of type classes

The dictionary passing semantics is quite intuitive and very easy to type in the target language.

However, dictionaries may be replaced by a derivation tree that proves the existence of the dictionary. This derivation tree can be passed around instead of the dictionary and at the call site be used to dispatch to the appropriate implementation of then method.

This has been studied in [Furuse, 2003b].

This 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). See the following course.

Multi-parameter type classes

Multi-parameter type classes are of the form

$$\text{class } \vec{P} \Rightarrow K\ \vec{X}\ \{\rho\}$$

where free variables of $\vec{P}$ are in $\vec{X}$.

The current framework can easily be extended to handle multi-parameter type classes.

Example

Collections represented by type C whose elements are of type E can be defined as follows:

$$\begin{array}{l}
\textbf{class}\ \text{Collection}\ C\ E\ \{\ \text{find} : C \to E \to \text{Option}(E),\ \text{add} : C \to E \to C\ \} \\
\textbf{inst}\ \text{Collection}\ (\text{List}\ X)\ X\ \{\ \text{find} = \text{List.find},\ \text{add} = \lambda(c)\lambda(e)\ e{::}c\ \} \\
\textbf{inst}\ \text{Collection}\ (\text{Set}\ X)\ X\ \{\ \ldots\ \}
\end{array}$$

Type dependencies

However, the class Collection does not provide the intended intuition that collections be homogeneous:

```
let add2 c x y = add (add c x) y
add2 : all(C, E, E') Collection C E, Collection C E' ⇒ C → E → E' → C
```

This definition assumes that collections may be heterogeneous. This may not be intended, and perhaps no instance of heterogeneous collections may ever be provided will ever be provided.

To statically enforce collections to be homogeneous in types, the definition can add a clause to say that the parameter C determines the parameter E:

```
class Collection C E | C → E { ... }
```

Then, add2 would enforce E and E' to be equal.

Type dependencies

Type dependencies also reduce overlapping between class declarations.

Hence they allow example that would have to be rejected if type dependencies could not be expressed.

Associated types

Functional dependencies are beeing replaced by the notion of associated types.

Associated types allow a class to declare its own type function.

Correspondingly, instance definitions must provide a definition for associated type (in addition to values for overloaded symbols as before).

For example, the Collection class becomes a single paramter 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

Associated types increase the expressivity of type classes.

Overlapping instances Example

In practice, overlapping instances may be desired! 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.

If overlapping instances are allowed, further rules are needed to disambiguate the resolution of overloading, such as giving priority to rules, or using 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.

Overlapping instances

Example

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

This elaborates into the creation of a generic dictionary

```
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

One may consider removing other restrictions on the class declarations or instance definitions. While some of these generalizations would make sense in theory, they may raise serious difficulties in practice.

For example:

- If constrained type schemes are of the form $\mathsf{K}\,T$ instead of $\mathsf{K}\,X$? (which affects all aspects of the language), then it becomes difficult to control the termination of constrained resolution and of the elaboration of dictionaries.
- If a class instances $\mathsf{inst}\;\forall \vec{\mathsf{Y}}.\,\vec{P} \Rightarrow \mathsf{K}\,T\;\{\rho\}$ could be such that T is $\mathsf{F}\,\vec{T}$ and not $\mathsf{F}\,\vec{\mathsf{Y}}$, then it becomes difficult to check non-overlapping of class instances.

Methods as overloading functions

One approach to object-orientation is to see methods as over as overloaded functions.

Object then 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.

Dynamics overloading is a powerful mechanism.

Haskell type classes are a practical, general and powerful solution to dynamic overloading,

Dynamic overloading works relatively well in combination with ML-like type inference.

However, besides the simplest case where every one agrees, useful extensions often come with some drawbacks, and they is not yet a agreement on the best design choices.

The design decisions are often in favor of expressiveness, but loosing some of the properties and the canonicity of the minimal design.

Summary

Dynamic overloading is a typical and very elegant use of elaboration.

The programmer could in principle write the elaborated program, build and pass dictionaries explicitly, but this would be cumbersome, tricky, error prone, and it would obfuscate the code.

The elaboration does this automatically, without arbibrary choices (in the minimal design) and with only local transformations that preserve the structure of the source.

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 ML& [Castagna, 1997]

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 I

(Most titles have a clickable mark "▷" that links to online versions.)

▷ 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.

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.

Bibliography II

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

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.

▷ 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*, 1999.

Bibliography III

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

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

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.

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.

Bibliography IV

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.

▷ 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.

Bibliography V

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.

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

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.

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

Bibliography VI

- 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.
- 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.
- 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.