# MPRI 2.4, Functional programming and type systems
# Metatheory of System F

Didier Rémy

# Plan of the course

Metatheory of System F

ADTs, Recursive types, Existential types, GATDs

Going higher order with $F^{\omega}$!

Logical relations

Side effects, References, Value restriction

Type reconstruction

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 new definition may just shadow an older one and temporarily become the only one 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.* $+$, $+_\bullet$); or numerical constants *e.g.* $0$, $0_\bullet$

**Modularity**

To avoid name clashing, the naming discipline (including name mangling conventions) must be known 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 $\tau[\alpha]$ for all $\alpha$ may have an implementation depending on the type of $\alpha$. For instance, a marshalling function of type $\forall\alpha.\alpha \to string$ may execute a different code for each base type $\alpha$.

## Ad hoc polymorphism

Overloaded definitions may be *ad hoc*, *i.e.* completely unrelated for each type, or just share a same type schema.

For instance, $0$ could mean either the integer zero or the empty list. The symbol $\times$ could mean either integer product or string concatenation.

# Why use overloading?

## Type dependent functions

A function defined on $\tau[\alpha]$ for all $\alpha$ may have an implementation depending on the type of $\alpha$. For instance, a marshalling function of type $\forall\alpha.\alpha \to string$ may execute a different code for each base type $\alpha$.

## Polytypic polymorphism

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

Typical examples of polytypic functions are marshalling 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 or any definition be overloaded? *e.g.* can numerical values be overloaded?
- Are all overloaded definitions of the same name 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 resolved?

**How is overloading resolution defined?**

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

**Static resolution (rather simple)**

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

# How is overloading resolved?

**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 argument so that different calls can make different choices.

The resolution is driven by information made available at runtime:

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

# Static resolution



**In SML**

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

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 Java programmers.*

**class** A { **int** bin(A y) { **return** 1; } }
**class** B **extends** A { **int** bin(B y) { **return** 2; } }
A aa = **new** A(); B bb = **new** B(); A ab = bb;

$x.bin(y)$? when $x, y \in \{aa, bb, ab\}$

# Static resolution



It does not fit well with first-class functions and polymorphism:

For example, $\lambda x.\, x + x$ is rejected when + is overloaded, as it cannot be statically resolved. The function must be specialized at some type at which $+$ is defined.

This argues in favor of some form of dynamic overloading:
dynamic overloading allows to delay resolution of overloaded symbols 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 these tags.
  (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

**Dispatch on runtime type**

- 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 overlapping definitions 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 resolved when their types are sufficiently specialized and before they are actually needed.
  In short, *let* $f = \lambda x.\, x + x$ *in* $a$ can be elaborated into
  *let* $f = \lambda(+).\, \lambda x.\, x + x$ *in* $a$. Then, the application of $f$ to a float in $a$ *e.g.* $f\ 1.0$ can be 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 explored 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.
- Other works have tried to relax some restrictions [Morris and Jones, 2010]

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

Type-classes overloading style can also be largely mimicked with implicit module arguments [White et al., 2014] with a few drawbacks but also many advantages.

# Contents





# Mini-Haskell

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

- single parameter type classes
- no overlapping instance definitions

It is close to *A second look at overloading* by Odersky et al. in terms of expressiveness and simplicity—but closer to Haskell in style: it can be easily generalized by lifting restrictions without changing the framework.

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

- Some examples in Mini-Haskell.
- Elaboration of Mini-Haskell into (the ML subset of) System F.
- An implicitly-typed version with type inference.

# Mini-Haskell Example — Implicitly/Explicitly Typed

**class** *Eq* X { equal : X → X → Bool }

**inst** *Eq* Int { equal = primEqInt }
**inst** *Eq* Char { equal = primEqChar }
**inst** Λ(X) *Eq* X ⇒ *Eq* (List (X))
{ equal = λ(l$_1$ : List X) λ(l$_2$ : List X) match l$_1$, l$_2$ with
| [],[] → true | [], _ | _,[] → false
| h$_1$::t$_1$, h$_2$::t$_2$ → equal X h$_1$ h$_2$ && equal (List X) t$_1$ t$_2$ }

This code:

- declares a class (dictionary) of type Eq(X) that contains definitions for equal : X → X → Bool,
- creates two concrete instances (dictionaries) of type *Eq* Int and *Eq* Char,
- may create a concrete instance of type *Eq* (List(X)) for any instance of type *Eq*(X)

# Example — Elaboration into explicit dictionaries

**class** *Eq* X { equal : X → X → Bool }

**inst** *Eq* Int { equal = primEqInt }
**inst** *Eq* Char { equal = primEqChar }
**inst** Λ(X) *Eq* X ⇒ *Eq* (List (X))
{ equal = λ(l$_1$ : List X) λ(l$_2$ : List X) match l$_1$, l$_2$ with
| [],[] → true | [], _ | [], _ → false
| h$_1$::t$_1$, h$_2$::t$_2$ → equal X h$_1$ h$_2$ && equal (List X) t$_1$ t$_2$ }

Becomes:

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

**let** *EqInt* : *Eq* Int = { equal = primEqInt }
**let** *EqChar* : *Eq* Char = { equal = primEqChar }
**let** *EqList* X (*EqX* : *Eq* X) : *Eq* (List X)
{ equal = λ(l$_1$ : List X) λ(l$_2$ : List X) match l$_1$, l$_2$ with
| [],[] → true | [], _ | [], _ → false
| h$_1$::t$_1$, h$_2$::t$_2$ →
equal X *EqX* h$_1$ h$_2$ && equal (List X) (*EqList* X *EqX*) t$_1$ t$_2$ }

# Example — Class Inheritance

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

$$\textbf{class}\ \mathit{Eq}\ \mathsf{X} \Rightarrow \mathit{Ord}\ (\mathsf{X})\ \{\ \mathsf{lt} : \mathsf{X} \to \mathsf{X} \to \mathsf{Bool}\ \}$$
$$\textbf{inst}\quad \mathit{Ord}\ \mathsf{Int}\ \{\ \mathsf{lt} = (<)\ \}$$

This declares a new class (dictionary) $\mathit{Ord}\ \mathsf{X}$ that depends on a dictionary $\mathit{Eq}\ \mathsf{X}$ and contains a method $\mathsf{lt} : \mathsf{X} \to \mathsf{X} \to \mathsf{Bool}$.

The instance definition builds a dictionary $\mathit{Ord}\ \mathsf{Int}$ from the existing dictionary $\mathit{Eq}\ \mathsf{Int}$ and the primitive $(<)$ for $\mathsf{lt}$.

The two declarations are elaborated into:

$$\textbf{type}\ \mathit{Ord}\ \mathsf{X} = \{\ \mathit{Eq} : \mathit{Eq}\ \mathsf{X};\ \mathsf{lt} : \mathsf{X} \to \mathsf{X} \to \mathsf{Bool}\ \}$$
$$\textbf{let}\quad \mathit{EqOrd}\ \mathsf{X}\ (\mathit{OrdX} : \mathit{Ord}\ \mathsf{X}) : \mathit{Eq}\ \mathsf{X} = \mathit{OrdX}.\mathit{Eq}$$
$$\textbf{let}\quad \mathsf{lt}\ \ \mathsf{X}\ (\mathit{OrdX} : \mathit{Ord}\ \mathsf{X}) : \mathsf{X} \to \mathsf{X} \to \mathsf{Bool} = \mathit{OrdX}.\mathsf{lt}$$

$$\textbf{let}\ \mathit{OrdInt} : \mathit{Ord}\ \mathsf{Int} = \{\ \mathit{Eq} = \mathit{EqInt};\ \mathsf{lt} = (<)\ \}$$

# Mini Haskell — Overloading

An overloaded function search is defined as follows:

```
let rec leq : ∀(X) Ord X ⇒ X → List X → Bool =
  Λ(X) λ(x : X) λ(l : ListX)
    match l with [] → true
   | h::t → (lt x h || equal x h) && leq x t

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

This elaborates into:

```
let rec leq X (OrdX : Ord X) (x : X) (l : ListX) : Bool =
  match l with | [] → true
  | h::t → (lt X OrdX x h || equal X (EqOrd X OrdX) x h)
             && leq X OrdX x t

let b = leq Int OrdInt 1 [1; 2; 3];;
```

That is, the code in green is inferred.

# Contents





# Mini Haskell

We restrict to single parameter classes.

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

# Mini Haskell

In practice, a program is composed of *interleaved*

- class declarations,
- instance definitions,
- function definitions,

given *in any order* and

- ending with an expression.

Instance and function definitions are interpreted recursively.
Hence, their definition order does not matter.

For simplification, we assume that instance definitions do not depend on function definitions, which may then come last as part of the expression in a recursive let-binding.

# Mini Haskell

In practice, a program is composed of *sequences of*

- class declarations,
- instance definitions,

given *in this order* and

- ending with an expression.

Instance definitions are interpreted recursively; their 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.

# Mini Haskell

Source programs $p$ are of the form:

$$p ::= H_1 \ldots H_p \; h_1 \ldots h_q \; M$$

$$H ::= \mathit{class}\, \vec{P} \Rightarrow K\,\alpha\,\{\rho\}$$
$$\rho ::= u_1 : \tau_1, \ldots u_m : \tau_m$$

$$h ::= \mathit{inst}\, \forall \vec{\beta}.\, \vec{P} \Rightarrow K\,(G\,\vec{\beta})\,\{r\}$$
$$r ::= u_1 = M_1, \ldots u_k = M_k$$

$$P ::= K\,\alpha \qquad Q ::= K\,\tau \qquad \sigma ::= \forall \vec{\alpha}.\, \vec{Q} \Rightarrow T \qquad T ::= \tau \mid Q$$

Letter $u$ ranges over overloaded symbols.

Class constructors $K$ may appear in $Q$ but not in $\tau$.
Only regular type constructors $G$ may appear in $\tau$.

We write $\forall \vec{\alpha}.\, Q_1 \Rightarrow \ldots Q_m \Rightarrow T$ for $\forall \vec{\alpha}.\, Q_1, \ldots Q_m \Rightarrow T$
and see $\Rightarrow$ as an annotated version of $\rightarrow$.

# Mini Haskell

Source programs $p$ are of the form:

$$p ::= H_1 \ldots H_p \; h_1 \ldots h_q \; M$$

$$H ::= \mathit{class}\ \vec{P} \Rightarrow K\,\alpha\,\{\rho\}$$
$$\rho ::= u_1 : \tau_1, \ldots u_m : \tau_m$$

$$h ::= \mathit{inst}\ \forall\vec{\beta}.\ \vec{P} \Rightarrow K\,(G\,\vec{\beta})\,\{r\}$$
$$r ::= u_1 = M_1, \ldots u_k = M_k$$

$$P ::= K\,\alpha \qquad Q ::= K\,\tau \qquad \sigma ::= \forall\vec{\alpha}.\vec{Q} \Rightarrow T \qquad T ::= \tau \mid Q$$

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

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

# Target language

System F, extended with record types, let-bindings, and let-rec.

Records are provided as data types. They are used to represent dictionaries. Record labels represent overloaded symbols $u$.

We may also use overloaded symbols $u$ 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 $N$ instead of $M$ for elaborated terms, to distinguish them from source terms.

# Class declarations

$$H \triangleq \mathit{class}\ K_1\ \alpha, \ldots K_p\ \alpha \Rightarrow K\ \alpha\ \{\rho\}$$

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 $K$.

Classes $K_i$'s are superclasses of $K$ and we write $K_i \prec K$.

Class definitions must respect the order $\prec$ *(acyclic)*

The dictionary of $K$ will contain a sub-dictionary for each superclass $K_i$.

All $K_i$'s are independent in a *typing context*: there does not exists $i$ and $j$ such that $K_j \prec K_i$.

Indeed, if $K_j \prec K_i$, then $K_i$ dictionary would contain a sub-dictionary for $K_j$, to which $K$ has access via $K_i$ so $K$ does not itself need dictionary $K_j$.

# Class declarations

$$H \triangleq \mathit{class}\ K_1\ \alpha, \ldots K_p\ \alpha \Rightarrow K\ \alpha\ \{\rho\}$$

The row type $\rho$ is of the form

$$u_1 : \tau_1, \ldots\ u_m : \tau_m$$

and declares *overloaded symbols* $u_i$ (also called *methods*) of class $K$.

An overloaded symbol cannot be declared twice in the same class and must be declared only in one class.

Types $\tau_i$'s must be closed with respect to $\alpha$.

Each class instance will contain a definition for each method.

# Class declarations — Elaboration

$$H \triangleq \mathit{class}\ K_1\ \alpha, \ldots K_p\ \alpha \Rightarrow K\ \alpha\ \{\rho\}$$

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

The row $\rho$ only lists methods $u_1 : \tau_1, \ldots u_m : \tau_m$. We extend it with sub-dictionary fields and define $\rho^K$ to be $\rho, u^K_{K_1} : K_1\ \alpha, \ldots\ u^K_{K_p} : K_p\ \alpha$. Thus $\rho^K$ is of the form $u_1 : T_1, \ldots u_n : T_n$. We introduce:

- a record type definition $K\ \alpha \approx \{u_1 : T_1, \ldots u_n : T_n\}$,
- for each $i$ in $1..n$ we define the accessor to field $u_i$:
  - let $N_i$ be $\Lambda\alpha.\, \lambda z{:}K\ \alpha.\, (z.u_i)$.
  - let $\sigma_i$ be $\forall\alpha.\, K\ \alpha \Rightarrow T_i$, *i.e.* the type of $N_i$
  - let $\mathcal{R}_i$ be the program context $\mathit{let}\ u_i : \sigma_i = N_i\ \mathit{in}\ [\,]$.

Then, $[\![H]\!]$ is $\mathcal{R}_1 \circ \ldots \mathcal{R}_n$ and we write $\Gamma_H$ for the typing environment $u_1 : \sigma_1 \ldots u_p : \sigma_p$ in the hole of $[\![H]\!]$.

# Class declarations

## Elaboration

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

$$[\![H_1 \dots H_p]\!] \triangleq [\![H_1]\!] \circ \dots [\![H_p]\!] \triangleq \mathit{let}\ \vec{u} : \vec{\sigma}_u = \vec{N}_u\ \mathit{in}\ [\,]$$

Record type definitions are collected in the program prelude.

We write $\Gamma_{H_1 \dots H_p}$ for $\Gamma_{H_1}, \dots \Gamma_{H_p}$.

# Instance definitions

$$h \triangleq \mathit{inst}\ \forall \vec{\beta}.\ K'_1\ \beta_1, \ldots K'_k\ \beta_k \Rightarrow K\,(G\,\vec{\beta})\ \{r\}$$

It defines an instance of a class $K$.

The *typing context* $K'_1\ \beta_1, \ldots K'_k\ \beta_k$ describes the dictionaries that must be available on type parameters $\vec{\beta}$ to build the dictionary $K\,(G\,\vec{\beta})$.

*This is not related to the superclasses of the class K*:

For example, in

**inst** $\mathbf{\Lambda(X)}$ *Eq* X ⇒ *Eq* (List (X))

An instance of class *Eq* at type X is needed to build an instance of class *Eq* at type List(X), but *Eq* is not a superclass of itself.

# Instance definitions

$$h \triangleq \mathit{inst}\ \forall \vec{\beta}.\ K'_1\ \beta_1, \ldots K'_k\ \beta_k \Rightarrow K\,(G\,\vec{\beta})\ \{r\}$$

The typing context describes dictionaries that cannot yet be built because they depend on some unknown type $\beta$ in $\vec{\beta}$.

We assume that the typing context is such that:

- each $\beta_i$ is in $\vec{\beta}$
- $\beta_i$ and $\beta_j$ may be equal, except if $K_i$ and $K_j$ are related (*i.e.* $K_i \prec K_j$ or $K_j \prec K_i$ or $K_i = K_j$)
  The reason is, as for class declarations, that it would be useless to require both dictionaries $K_i\ \beta$ and $K_j\ \beta$ when they are equal or one is contained in the other.

Such typing contexts are said to be *canonical*.

# Instance declarations Elaboration

$$h \triangleq \mathit{inst}\ \forall\vec{\beta}.\ K'_1\ \beta_1, \ldots K'_k\ \beta_k \Rightarrow K\,(G\,\vec{\beta})\ \{r\}$$

This instance definition $h$ is elaborated into a triple $(z_h, N^h, \sigma_h)$ where $z_h$ is an identifier to refer to the elaborated body $N^h$ of type $\sigma_h$.

The type $\sigma_h$ is $\forall\vec{\beta}.\, K'_1\ \beta_1 \Rightarrow \ldots K'_k\ \beta_k \Rightarrow K\,(G\,\vec{\beta})$

The expression $N^h$ builds a dictionary of type $K\,(G\,\vec{\beta})$, given $k \geq 0$ dictionaries of respective types $K'_1\ \beta_1, \ldots K'_k\ \beta_k$:

$$\Lambda\vec{\beta}.\,\lambda(z_1 : K'_1\ \beta_1).\ \ldots \lambda(z_k : K'_k\ \beta_k).$$
$$\{u_1 = N^h_1, \ldots u_m = N^h_m,\ u^K_{K_1} = q_1, \ldots u^K_{K_p} = q_p\}$$

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

- $N^h_i$ is the elaboration of $M_i$ where $r$ is $u_1 = M_1, \ldots u_m = M_m$.
- $q_i$ is a dictionary of type $K_i\ (G\,\vec{\beta})$ (the $i$'th subdictionary of $K$)

(We write $z$ for a variable $x$ that binds a dictionary.)

# Elaboration of whole programs

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

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

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

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

Hence, the expression $N$ and all expressions $N^h$ are typed (and elaborated) in the environment $\Gamma_0$ equal to $\Gamma_{\vec{H}}$, $\Gamma_{\vec{h}}$ where

- $\Gamma_{\vec{H}}$ declares functions to access components of dictionaries (both sub-dictionaries and definitions of overloaded symbols).
- $\Gamma_{\vec{h}}$ equal to $(\vec{z}_h : \vec{\sigma}_h)$ declares functions to build dictionaries (*i.e.* all class instances).

# Elaboration of expressions

The elaboration of expressions is defined by a judgment

$$\Gamma \vdash M \rightsquigarrow N : \sigma$$

where $\Gamma$ is a System-F typing context, $M$ is the source expression, $N$ is the elaborated expression and $\sigma$ its type in $\Gamma$.

In particular, $\Gamma \vdash M \rightsquigarrow N : \sigma$ implies $\Gamma \vdash N : \sigma$ in $F$.

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

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

($u$ and $z$ are just particular cases of $x$)

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

# Elaboration of expressions

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

$$\textsc{Let}\ \frac{\Gamma \vdash M_1 \rightsquigarrow N_1 : \sigma \qquad \Gamma, x : \sigma \vdash M_2 \rightsquigarrow N_2 : \tau}{\Gamma \vdash \mathit{let}\ x : \sigma = M_1\ \mathit{in}\ M_2 \rightsquigarrow \mathit{let}\ x : \sigma = N_1\ \mathit{in}\ N_2 : \tau}$$

$$\textsc{App}\ \frac{\Gamma \vdash M_1 \rightsquigarrow N_1 : \tau_2 \to \tau_1 \qquad \Gamma \vdash M_2 \rightsquigarrow N_2 : \tau_2}{\Gamma \vdash M_1\ M_2 \rightsquigarrow N_1\ N_2 : \tau_1} \qquad \textsc{Abs}\ \frac{\Gamma, x : \tau' \vdash M \rightsquigarrow N : \tau}{\Gamma \vdash \lambda x{:}\tau'.\, M \rightsquigarrow \lambda x{:}\tau'.\, N : \tau' \to \tau}$$

In rule Let, $\sigma$ must be canonical, *i.e.* of the form $\forall \vec{\alpha}.\, \vec{P} \Rightarrow T$ where $\vec{P}$ is itself empty or canonical (see the definition and also this restriction).

Rules App and Abs do not apply to overloaded expressions of type $\sigma$.

# Elaboration of overloaded expressions

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

$$\textsc{OAbs}\ \frac{\Gamma, x : Q \vdash M \rightsquigarrow N : \sigma \qquad x \# M}{\Gamma \vdash M \rightsquigarrow \lambda\, x : Q.\, N : Q \Rightarrow \sigma} \qquad \textsc{OApp}\ \frac{\Gamma \vdash M \rightsquigarrow N : Q \Rightarrow \sigma \qquad \Gamma \vdash q : Q}{\Gamma \vdash M \rightsquigarrow N\ q : \sigma}$$

Rule OAbs pushes dictionary abstractions $Q$ in the context $\Gamma$ as prescribed by the expected type of the argument $x$.

These may then be used (in addition to dictionary accessors and instance definitions already in $\Gamma$) to elaborate dictionaries as described by the premise $\Gamma \vdash q : Q$ of rule OApp.

# Elaboration of overloaded expressions

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

$$\textsc{OAbs}\ \frac{\Gamma, x : Q \vdash M \rightsquigarrow N : \sigma \qquad x \# M}{\Gamma \vdash M \rightsquigarrow \lambda\, x : Q.\, N : Q \Rightarrow \sigma} \qquad \textsc{OApp}\ \frac{\Gamma \vdash M \rightsquigarrow N : Q \Rightarrow \sigma \qquad \Gamma \vdash q : Q}{\Gamma \vdash M \rightsquigarrow N\ q : \sigma}$$

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

# Elaboration of overloaded expressions

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

$$\textsc{OAbs}\ \frac{\Gamma, x : Q \vdash M \rightsquigarrow N : \sigma \qquad x \# M}{\Gamma \vdash M \rightsquigarrow \lambda\, x : Q.\, N : Q \Rightarrow \sigma} \qquad \textsc{OApp}\ \frac{\Gamma \vdash M \rightsquigarrow N : Q \Rightarrow \sigma \qquad \Gamma \vdash q : Q}{\Gamma \vdash M \rightsquigarrow N\ q : \sigma}$$

By construction, elaboration produces well-typed expressions: that is $\Gamma_0 \vdash M \rightsquigarrow N : \tau$ implies that is $\Gamma_0 \vdash N : \tau$.

# Resuming the elaboration

An instance declaration $h$ of the form:

$$\mathit{inst}\ \forall\vec{\beta}.\ K_1'\ \beta_1, \ldots K_k'\ \beta_k \Rightarrow K\ (G\ \vec{\tau})\ \{u_1 = M_1, \ldots u_m = M_m\}$$

is translated into

$$\Lambda\vec{\beta}.\ \lambda(z_1 : K_1'\ \beta_1).\ \ldots \lambda(z_k : K_k'\ \beta_k).$$
$$\{u_1 = N_1^h, \ldots u_m = N_m^h,\ u_{K_1}^K = q_1, \ldots u_{K_p}^K = q_p\}$$

where:

- $u_{K_i}^K : Q_i$ are the superclasses fields, $u_i : \tau_i$ are the method fields
- $\Gamma_h$ is $\vec{\beta}, K_1'\ \beta_1, \ldots K_k'\ \beta_k$
- $\Gamma_0, \Gamma_h \vdash q_i : Q_i$
- $\Gamma_0, \Gamma_h \vdash M_i \rightsquigarrow N_i : \tau_i$

Finally, given the program $p$ equal to $\vec{H}\ \vec{h}\ M$, we elaborate $M$ as $N$ such that $\Gamma_0 \vdash M \rightsquigarrow N : \forall\bar{\alpha}.\tau$.

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

# Let-monomorphization

Otherwise, $N$ 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 $\textsc{Let}$ so that either $\sigma$ is of the form $\forall \bar{\alpha}.\tau$ or $M_1$ 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 $M_1\ M_2$.

We must in fact compare:

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

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 their evaluation is shared and only reduced once.

The application $V_1\ V_2$ will be reduced once in $(2)$, but as many types as there are occurrences of $x_2$ in $M_2$ in $(1)$.

# 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 error may occur during elaboration of an expression.
- Some required dictionary cannot be built.

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

Hum... Although terms are explicitly-typed, their elaboration may not be unique! Indeed, there 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 program then 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 *zobservational 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 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.

# Elaboration of dictionaries

Elaboration of dictionaries is just typing in System F.

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

The relevant subset of rules for dictionary expressions are:

D-OVar
$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}$$

D-Inst
$$\frac{\Gamma \vdash q : \forall \alpha.\, \sigma}{\Gamma \vdash q\ \tau : [\alpha \mapsto \tau]\sigma}$$

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

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:

- a dictionary constructor $z_h$ given by an instance definition $h$;
- a dictionary accessor $u_K^{K'}$ given by a class declaration $K'$;
- a dictionary argument $z$, given by the local typing context.

Hence, the typing rules may be reorganized as follows:

D-OVar-Inst

$$\frac{z_h : \forall \vec{\beta}.\, P_1 \Rightarrow \ldots\, P_n \Rightarrow K\,(G\,\vec{\beta}) \in \Gamma \qquad \Gamma \vdash q_i : [\vec{\beta} \mapsto \vec{\tau}]P_i}{\Gamma \vdash z_h\ \vec{\tau}\ \vec{q} : K\,(G\,\vec{\tau})}$$

D-Proj

$$\frac{u_K^{K'} : \forall \alpha.\, K'\ \alpha \Rightarrow K\,\alpha \in \Gamma \qquad \Gamma \vdash q : K'\ \tau}{\Gamma \vdash u_K^{K'}\ \tau\ q : K\,\tau}$$

D-Var

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

# Elaboration of dictionary *values*

Dictionary values are typed in $\Gamma_0$, which does not contain free type variables, hence, the last rule does not apply.

Dictionary stored in other dictionaries must have been built in the first place. Hence, all dictionary values can be built with the *unique* rule:

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

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

There is no choice in finding $z_h : \forall \vec{\beta}.\, P_1 \Rightarrow \ldots\ P_n \Rightarrow K\,(G\,\vec{\beta}) \in \Gamma$, since each such clause is coming from an instance definition $h$, and we requested that *instance definitions never overlap*.

This ensures *uniqueness of dictionary* **values**, hence *coherence*.

# Overlapping instances

Two instances *inst* $\forall \vec{\beta}_i.\ \vec{P} \Rightarrow K\ (G_i\ \vec{\beta}_i)\ \{r_i\}$ for $i$ in $\{1, 2\}$ of a class $K$ overlap if the type schemes $\forall \vec{\beta}_i.\, K\ (G_i\ \vec{\tau}_i)$ have a common instance, *i.e.* in the present setting, if $G_1$ and $G_2$ 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? Indeed, in overloaded code, the exact type is not fully known at compile time, 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 $\Gamma_0$ which may contain typing assumptions $z : K'\ \beta$ 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 (reminder):

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

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

$$\textsc{D-Var}\quad \frac{z : K\ \alpha \in \Gamma}{\Gamma \vdash z : K\ \alpha}$$

They can be read as a prolog-like *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 $\tau$, 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 $K'\ \vec{\tau}$ with $K' \prec K$.
- If D-Ovar-Inst applies, the premises are called at type $K_i\ \tau_j$ where $\tau_j$ is subtype (i.e. subterm) of $\vec{\tau}$, 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 *OrdInt*.*Eq*, 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 enforce 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 $\Gamma_0$ is:

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

When elaborating the body of leq, we have to infer a dictionary for *EqOrd* X *OrdX* in the local context X, *OrdX* : *Ord* X. Thus, $\Gamma$ is $\Gamma_0, \alpha, z : Ord\ \alpha$ and *EqOrd* is $u_{Eq}^{Ord}$. We have:

$$\textsc{d-Proj}\ \frac{\textsc{d-OVar-Inst}\quad \Gamma \vdash u_{\textit{Eq}}^{\textit{Ord}}\ \alpha : \textit{Ord}\ \alpha \to \textit{Eq}\ \alpha \qquad \textsc{d-Var}\quad \Gamma \vdash z : \textit{Ord}\ \alpha}{\Gamma \vdash u_{\textit{Eq}}^{\textit{Ord}}\ \alpha\ z : \textit{Eq}\ \alpha}$$

# Contents





# 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 implicit, in particular dictionary applications, but also abstractions over unresolved dictionaries.

# Example

```
class Eq X   { equal : X → X → Bool }
inst Eq Int  { equal = primEqInt }
inst Eq Char { equal = primEqChar }
inst Λ(X) Eq X ⇒ Eq (List (X))
  { eq = 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 leq : ∀(X) Ord X ⇒ X → List X → Bool =
  Λ(X) λ(x : X) λ(l : List X)
    match l with [] → true
    | h::t → (equal X x h || lt X x h) && leq X x t

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

# Type inference

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

Just read $\forall\vec{\alpha}.\,\vec{P} \Rightarrow \tau$ as the constraint type scheme $\forall\vec{\alpha}[\vec{P}].\,\tau$.

We extend constraints with dictionary predicates:

$$C ::= \ldots \mid K\,\tau$$

On ground types a constraint $K\,\mathbf{t}$ is satisfied if one can build a dictionary of type $K\,\mathbf{t}$ in the initial environment $\Gamma_0$ (that contains all class and instance declarations), *i.e.* formally, if there exists a dictionary expression $q$ such that $\Gamma_0 \vdash q : K\,\mathbf{t}$.

The satisfiability of class-membership constraints is thus:

$$\frac{K\,\phi\tau}{\phi \vdash K\,\tau}$$

# Reasoning with class-membership constraints

For every class declaration *class* $K_1\ \alpha, \ldots K_n\ \alpha \Rightarrow K\ \alpha\ \{\rho\}$,

$$K\ \alpha \quad \Vdash \quad K_1\ \alpha \wedge \ldots\ K_n\ \alpha \qquad (1)$$

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

*Proof of (1).* *Assume* $\phi \vdash K\ \alpha$, i.e. $\Gamma_0 \vdash q : K\ (\phi\alpha)$ *for some* $q$.

*From the class declaration, we know that* $K\ \alpha$ *is a record type definition that contains fields* $u^K_{K_i}$ *of type* $K_i\ \alpha_i$. *Hence, the dictionary value* $q$ *contains field values of types* $K_i\ (\phi\alpha)$. *Therefore, we have* $\phi \vdash K_i\ \alpha$ *for all* $i$ *in* $1..n$, *which implies* $\phi \vdash K_1\ \alpha \wedge \ldots\ K_n\ \alpha$. □

# Reasoning with class-membership constraints

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

$$K\,(G\,\vec{\beta}) \quad \equiv \quad K_1\ \beta_1 \wedge \ldots\ K_p\ \beta_p \qquad (2)$$

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

*Proof of (2) ($\dashv\!\shortmid$ direction).* *Assume* $\phi \vdash K_i\ \beta_i$. *There exists dictionaries* $q_i$ *such that* $\Gamma_0 \vdash q_i : K_i\ (\phi\beta_i)$. *Hence,* $\Gamma_0 \vdash x_h\ \vec{\beta}\ q_1\ \ldots\ q_p : K\,(G\,(\phi\vec{\beta}))$, i.e. $\phi \vdash K\,(G\,(\phi\vec{\beta}))$.

*($\shortmid\!\vdash$ direction).* *Assume,* $\phi \vdash K\,(G\,(\phi\vec{\beta})$. i.e. *there exists a dictionary* $q$ *such that* $\Gamma_0 \vdash q : K\,(G\,\phi\vec{\beta})$. *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* $x_h\ \vec{\beta}\ q_1\ \ldots\ q_p$ *with* $\Gamma_0 \vdash q_i : K_i\ (\phi\beta_i)$, *that is,* $\phi \vdash K_i\ \beta_i$ *for every* $i$, *which implies* $\phi \vdash K_1\ \beta_1 \wedge \ldots\ K_p\ \beta_p$.

# Reasoning with class-membership constraints

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

$$K\ (G\ \vec{\beta}) \quad \equiv \quad K_1\ \beta_1 \wedge \ldots\ K_p\ \beta_p \qquad (\mathbf{2})$$

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

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:

$$K\ (G\ \vec{\beta}) \quad \longrightarrow \quad K_1\ \beta_1 \wedge \ldots\ K_p\ \beta_p$$

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 $G$ for which there is no instance of $K$,

$$K\,(G\,\vec{\beta}) \quad \equiv \quad \mathit{false} \tag{3}$$

This rule allows failure to be reported as soon as constraints of the form $K\,(G\,\vec{\tau})$ appear and there is no instance of $K$ for $G$.

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

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

$$K\,(G\,\vec{\beta}) \quad \longrightarrow \quad \mathit{false}$$

would still remain sound but incomplete.

# Typing constraints

Constraint generation is as in ML.

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

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

# 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:

- $\Gamma_0, \Gamma_h \vdash a_i^h : \tau_i^h$ where $\tau_i^h$ is given.
  Thus, we check that the constraints *def* $\Gamma_0, \Gamma_h$ *in* $(\!|a_i^h|\!) \preceq \tau_i^h \equiv$ *true*.
- $\Gamma_0 \vdash a : \tau$ for some $\tau$.
  Thus, we check that *def* $\Gamma_0$ *in* $\exists\alpha.\, (\!|a|\!) \preceq \alpha \equiv$ *true*.

However, . . . Typechecking is not sufficient!

Type reconstruction should also return an explicitly-typed term $M$ that can then be elaborated into $N$.

# Type reconstruction

As for ML the resolution strategy for constraints may be tuned to keep persistent constraints from which an explicitly typed term $M$ can be read back.

# 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.

That is, $a$ elaborates to $N$ if $\Gamma \vdash a \rightsquigarrow M : \tau$ and $\Gamma \vdash M \rightsquigarrow N : \tau$.

Hence, even if 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 reconstruction.

# 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 reconstruction). However, this does not imply that there are principal *unconstrained* type schemes.

Indeed, assume that the principal constrained type scheme is $\forall\alpha[K\,\alpha].\,\alpha \to \alpha$ and the typing environment contains two instances of $K\,G_1$ and $K\,G_2$ of class $K$. Constraint-free instances of this type scheme are $G_1 \to G_1$ and $G_2 \to G_2$ but $\forall\alpha.\,\alpha \to \alpha$ is certainly not one.

Not only neither choice is principal, but the two choices would elaborate into 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 $\Gamma_0$ that request an unconstrained type scheme $\forall \alpha. \tau$

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

# 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 $\forall\alpha[\mathit{Num}\,\alpha].\,\alpha$—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 $\forall\alpha[\mathit{Readable}\,\alpha].\,\mathit{unit} \to \alpha$—and several classes are possible for $\mathit{Readable}\,\alpha$. 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 $\mathit{let}\, x = a_1\ \mathit{in}\, a_2$ is $\mathit{let}\, x : \forall\alpha[K\,\alpha].\, \mathit{int} \to \mathit{int} = N_1\ \mathit{in}\, N_2$.

All applications of $x$ in $N_2$ will lead to an unresolved constraint $K\,\alpha$ since neither the argument nor the context of this application can determine the value of the type parameter $\alpha$. Still, a dictionary of type $K\,\tau$ must be given before $N_1$ can be executed.

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

# Coherence

## The open-world view

When there is a single instance $K\,G$ for a class $K$ that appears in an unresolved or isolated constraint $K\,\alpha$, the problem formally disappears, as all possible type reconstructions are coherent.

However, we may still not accept this situation, for modularity reasons, as an extension of the program with another non-overlapping *correct* instance declaration would make the program become ambiguous.

Formally, this amounts to saying that the program must be coherent in its current form, but also in all possible extensions with well-typed class definitions. This is taking an *open-world* view.

# On the importance of principal type reconstruction

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

As noticed earlier, some (usually arbitrary) less general elaboration would solve 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

Consider the program

**let** twice $= \lambda(x)\ x + x$ **in** twice (twice 1)

Its principal type reconstruction is:

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

which elaborates into

**let** twice X numX $= \lambda(x : X)\ x$ (plus numX) x **in**
twice Int *NumInt* (twice Int *NumInt* 1)

while, avoiding polyorphism, twice would elaborate into:

**let** twice $= \lambda(x : \mathrm{Int})\ x$ (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.
- $\mathsf{read} : \mathsf{desc} \to X$.
  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 $K\,\alpha$ have types of the form $\alpha \to \tau$.

The above examples are indeed rejected by this definition.

# Overloading by return types

In fact, disallowing overloading by return types suffices to ensure 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 be used at the call site to dispatch to the appropriate implementation of the method.

This has been studied in [Furuse, 2003].

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 be equipped with GADT (Guarded Abstract Data Types).

# Multi-parameter type classes

Multi-parameter type classes are of the form

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

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

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:

```
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 { ... }
```

# 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 : ∀(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 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 examples that would have to be rejected if type dependencies could not be expressed.

# Associated types

Functional dependencies are being replaced by the notion of *associated types*, which allows a class to declare its *own type function*.

Correspondingly, instance definitions must provide a definition for 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

## 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 = primEqInt }

This elaborates into the creation of a generic dictionary

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

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

# 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 $K\,\tau$ instead of $K\,\alpha$? (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 instance $\mathit{inst}\ \forall\,\vec{\beta}.\ \vec{P} \Rightarrow K\,\tau\ \{\rho\}$ could be such that $\tau$ is $G\,\vec{\tau}$ and not $G\,\vec{\beta}$, then it would be more 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.

In this view, 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].

# Implicit arguments

Oliviera et al noticed that type classes could be largely emulated in Scala with implicit arguments [Oliveira et al., 2010].

This has recently be formalized by Oliviera et al in COCHIS, a calculus with implicits arguments [Schrijvers et al., 2017].

This allows *local scoping* of overloaded functions, but coherence is solved by a restriction to first-choice commitment during resolution to force it to be deterministic.

# Modules-based type classes

Modular type classes [Dreyer et al., 2007] mimic type classes at the level of modules, but with explicit abstraction and instantiations.

Modular implicits [White et al., 2014] allows for implicit module arguments. This extends the idea of modular type-classes in two directions.

- The language is more expressive
- Module arguments are inferred (left implicit).
- Module abstractions remain explicit. This allows for local scoping of overloading.

```
module type EQ =
  sig type t val eq : t → t → bool end

implicit module Eq_Int = struct
  type t = int let eq (x:int) y = x = y
end

implicit module Eq_Char = struct
  type t = char
  let eq (x:char) y = x = y end

let eq {Eq : EQ} x = Eq.eq x

implicit module Eq_List {Eq : EQ} =
  struct module rec R : EQ
      with type t = Eq.t list = struct
  type t = Eq.t list
  let eq l1 l2 = match l1, l2 with
   | [],[] → true | [], _ | _,[] → false
   | h1::t1, h2::t2 →
       eq h1 h2 && eq {R} t1 t2
end include R end
```

```
module type ORD = sig type t
  module Eq : EQ with type t = t
  val lt : t → t → bool end

implicit module Ord_Int = struct
  type t = int
  module Eq = Eq_Int
  let lt x y = (x < y) end

implicit module Eq {Ord : ORD} = Ord.Eq

let lt {Ord : ORD} x = Ord.lt x;;

implicit module Ord_List {Ord : ORD} =
  struct module rec R : ORD
      with type t = Ord.t list = struct
  type t = Ord.t list
  module Eq = Eq_List {Ord.Eq}
  let lt l1 l2 = match l1, l2 with
   | _, [] → false | [], _ → true
   | h1::t1, h2::t2 → lt h1 h2 && lt{R} t1 t2
end include R end
```

```
let leq (type a) {Ord : ORD with type t = a list} (l1 : a list) l2 =
    lt l1 l2 || eq l1 l2
```

# 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 there is not yet an 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 arbitrary 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.

A technique similar to defunctionalization can be used to replace dictionnaries by tags, which are interpreted when calling an overloaded function to dispatch to the appropriate definition [Pottier and Gauthier, 2004].

Implicit module arguments [White et al., 2014] can also mimick type-classes overloading with some drawbacks and advantages.

# Bibliography I

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

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.

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

▷ Derek Dreyer, Robert Harper, Manuel M. T. Chakravarty, and Gabriele Keller. Modular type classes. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 63–70, New York, NY, USA, 2007. ACM. ISBN 1-59593-575-4.

# Bibliography II


▷ 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 2003.

Mark P. Jones. *Qualified types: theory and practice*. Cambridge University Press, New York, NY, USA, 1995. 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 III

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.

▷ 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.

▷ Bruno C.d.S. Oliveira, Adriaan Moors, and Martin Odersky. Type classes as objects and implicits. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 341–360, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0203-6. doi: 10.1145/1869459.1869489.

# Bibliography IV


- François Pottier and Nadji Gauthier. Polymorphic typed defunctionalization. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 89–98, January 2004.
- 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.

- Tom Schrijvers, Bruno C. d. S. Oliveira, and Philip Wadler. Cochis: Deterministic and coherent implicits. Technical report, KU Leuven, May 2017.
- 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.

# Bibliography V


- 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.
- Leo White, Frédéric Bour, and Jeremy Yallop. Modular implicits. In Oleg Kiselyov and Jacques Garrigue, editors, *Proceedings ML Family/OCaml Users and Developers workshops, ML/OCaml 2014, Gothenburg, Sweden, September 4-5, 2014.*, volume 198 of *EPTCS*, pages 22–63, 2014. doi: 10.4204/EPTCS.198.2.