Generalities

Implementation strategies

System OML

Qualified types

Type classes

Design space
Modularity, *Surcharge*

*MPRI* course 2-4-2, Part 3, Lesson 2

Didier Rémy

INRIA-Rocquencourt

Janvier 27, 2009
<table>
<thead>
<tr>
<th>Generalities</th>
<th>Implementation</th>
<th>OML</th>
<th>Qualified types</th>
<th>Type classes</th>
<th>Design space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generalities</td>
<td>Implementation strategies</td>
<td>System OML</td>
<td>Qualified types</td>
<td>Type classes</td>
<td>Design space</td>
</tr>
</tbody>
</table>

**Didier Rémy** (INRIA-Rocquencourt)
Overloading

Naming convenience
Avoid suffixing similar names by type information: printing functions; numerical operations (e.g. `plus_int`, `plus_float`, ...); numerical values?

Type dependent functions or ad hoc polymorphism
A function defined on $\tau[\alpha]$ for all $\alpha$ may have an implementation depending on the type of $\alpha$. For instance, a marshaling function of type $\forall \alpha. \alpha \rightarrow \text{string}$ may execute different code for each base type $\alpha$.

These definitions may be ad hoc (unrelated for each type), or polytypic, i.e. depending solely on the type structure (is it a sum, a product, etc.) and thus derived mechanically for all types from the base cases.

A typical example of a polytypic function is the generation of random values for arbitrary types, e.g. as used in Quickcheck for Haskell.
Overloading

Common to all forms of overloading

- At some program point (static context), an overloaded symbol $u$ has several visible definitions $a_1, \ldots, a_n$.
- In a given runtime of the program, only one of them will be used.
  Determining which one should be used is called *overloading resolution*.

Many variants of overloading

- How is overloading resolved? (see next slide)
- Is resolution done up to subtyping?
- Are overloading definitions primitive, automatic, or user-definable?
- What are the restrictions in the way definitions can be combined?
  - Can the definitions overlap? (Then, how is overlapping resolved)
  - Can overloading be on the return type?
- Can overloading definitions have a local scope?
Overloading

Static resolution (rather simple)
- If every overloaded symbol can be statically replaced by its implementation at the appropriate type.
- This does not increase expressiveness, but may reduce verbosity.

Dynamic resolution (more involved)
- Pass types at runtime and dispatch on the runtime type (typecase).
- Pass the appropriate implementations at runtime as extra arguments, eventually grouped in dictionaries.
  (Alternatively, one may pass runtime information that designates the appropriate implementation in a global structure.)
- Tag values with their types—or an approximation of their types—and dispatch on the tags of values.
In SML

Definitions are primitive (numerical operators, record accesses).
Typechecking fails if overloading cannot be resolved at outermost let-definitions. For example, let $\text{twice } x = x + x$ is rejected in SML, at toplevel as $+$ could be the addition on either integers or floats.

In Java
Overloading

**In SML**

Definitions are primitive (numerical operators, 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**

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 current 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.
Limits

Static overloading does not fit well with first-class functions and polymorphism.

Indeed, functions such as $\lambda(x) \times + x$ are rejected and must therefore 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.
Runtime type dispatch

- Use an explicitly typed calculus (i.e. Church style System F)
- Add a typecase function.
- Type matching may be expensive, unless type patterns are 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: it keeps track of exhaustiveness; type matching functions as first-class and can be extended or overriden.
- Best match resolution strategy.
Overloading

Pass unresolved implementations as extra arguments

- Abstract over unresolved overloaded symbols and pass them later when they can be resolved.
  In short, let $f = \lambda(x) \ x + x$ can be elaborated into
  let $f = \lambda(+) \ \lambda(x) \ x + x$ and its application to a float $f \ 1.0$ elaborated into $f \ (+.) \ 1.0$.

- This can be done based on the typing derivation.

- After elaboration, types may be erased (Curry’s style System F)

- Monomorphisation or other simplifications may reduce the number of abstractions and applications introduced by overloading resolution.
Generalities

Implementation strategies

System OML

Qualified types

Type classes

Design space
Dynamic overloading

Untyped code

```
let rec plus = (+)
    and plus = (lor)
    and plus = λ(x, y) λ(x′, y′) (plus x x′, plus y y′) in
let twice = λ(x) plus x x in
twice (1, true)
```

It should indeed evaluate to \((1 + 1, \text{true lor true})\), i.e. \((2, \text{true})\), whatever the implementation strategy.
Church style System F with type matching

Syntax

\[
\begin{align*}
a & ::= a \mid \lambda(x)\ a \mid a\ (a) \mid \Lambda(\alpha)\ a \mid a\ (\tau) \\
& \mid \text{match } \tau \text{ with } \langle \pi_1 \Rightarrow a_1 \ldots \mid \pi_n \Rightarrow a_n \rangle \\
\pi & ::= \tau \mid \exists(\alpha)\pi
\end{align*}
\]

System F

Typecase

Typing rules: as in System F, plus...

\[
\begin{align*}
\Gamma \vdash \tau & \quad \Gamma, \bar{\alpha}_i \vdash \tau_i \\
\Gamma, \bar{\alpha}_i \vdash a_i : \tau' \\
\Gamma \vdash \text{match } \tau \text{ with } \langle \pi_1 \Rightarrow a_1 \ldots \mid \exists(\bar{\alpha}_i)\tau_i \Rightarrow a_i \ldots \mid \pi_n \Rightarrow a_n \rangle \leadsto a_i[\bar{\tau}_i/\bar{\alpha}_i]
\end{align*}
\]
Church style System F with type matching

Soundness for System F with type matching.

- Subject-reduction holds
- Progress does not hold in the simplest version: the type system cannot ensure exhaustiveness of type matching.
- Solutions:
  - add a default case, with a construction, such as
    \[
    \text{match } s \text{ with } \langle \pi \Rightarrow a \mid a \rangle
    \]
  - use a richer type system that ensures exhaustiveness.

What to do with overlapping definitions?

- Let the reduction be nondeterministic.
- Restrict typechecking to disallow overlapping definitions.
- Change the semantics to give priority to the first match, or to the best match (the most precise matching pattern).
Overloading with typecase

Exhaustiveness is not enforced

let rec plus =
Λ(α)
match α with ⟨
| int ⇒ (+)
| bool ⇒ (lor)
| ∃(β, γ) β × γ ⇒
  λ(x, y : β × γ) λ(x', y' : β × γ) plus β × x', plus γ y y'
⟩ in
let twice = Λ(α) λ(x : α) plus α x x in
twice (int × bool) (1, true)
Overloading with typecase

The domain may be restricted by a type constraint

```
let rec plus =
   Λ(α<Plus α>)
   match α with ⟨
   | int ⇒ (+)
   | bool ⇒ (lor)
   | ∃(β<Plus β>, γ<Plus γ>) β × γ ⇒
     λ(x, y : β × γ) λ(x', y' : β × γ) plus β x x', plus γ y y'
   ⟩ in
let twice = Λ(α<Plus α>) λ(x : α) plus α x x in
twice (int × bool) (1, true)
```
Overloading with typecase

The type predicate $Plus \alpha$ is defined by induction

$Plus \ int; \ Plus \ bool;

$Plus \ \alpha \Rightarrow Plus \ \beta \Rightarrow Plus \ (\alpha \times \beta)$

let rec $plus =$

$\Lambda(\alpha \langle Plus \ \alpha \rangle)$

match $\alpha$ with \\
| int $\Rightarrow$ (+) \\
| bool $\Rightarrow$ (lor) \\
| $\exists(\beta \langle Plus \ \beta \rangle, \ \gamma \langle Plus \ \gamma \rangle) \ \beta \times \gamma \Rightarrow$ \\
| $\lambda(x, y : \beta \times \gamma) \ \lambda(x', y' : \beta \times \gamma) \ plus \ \beta \times x', \ plus \ \gamma \ y \ y'$ \\

in

let $twice =$ $\Lambda(\alpha \langle Plus \ \alpha \rangle) \ \lambda(x : \alpha) \ plus \ \alpha \times x \ \times \ \in$

$twice \ (int \times bool) \ (1, \ true)$
Checking for satisfiability

Overloaded declarations are restricted forms of horn clauses. For instance, the context $\Gamma$ equal to

$$\text{Plus int;} \quad \text{Plus bool;} \quad \text{Plus } \alpha \Rightarrow \text{Plus } \beta \Rightarrow \text{Plus } (\alpha \times \beta)$$

can be read as deduction rules:

\[
\text{PlusInt}\quad \text{PlusBool}\quad \text{PlusProd}
\]

\[
\begin{align*}
\text{Plus int} & \quad \text{Plus bool} \\
\text{Plus } (\alpha \times \beta) & \\
\end{align*}
\]

We can build (infer) the following derivation:

\[
\text{PlusProd}\quad \text{PlusInt} \quad \text{PlusBool}
\]

\[
\begin{align*}
\text{Plus int} & \quad \text{Plus bool} \\
\text{Plus } (int \times bool) & \\
\end{align*}
\]

which can be concisely represented as the proof term on the right.
Dictionary passing

In fact, \( \text{Plus} \ (\text{int} \times \text{bool}) \) proves that \( \text{plus} \) is defined for type \( \text{int} \times \text{bool} \). Partially applying \( \text{plus} \) to \( \text{int} \times \text{bool} \), and reducing it, we get:

\[
\text{plus} \ (\text{int} \times \text{bool}) \leadsto \\
\lambda(x, y : \text{int} \times \text{bool}) \lambda(x', y' : \text{int} \times \text{bool}) \text{plus int y, plus bool x' y'} \leadsto \\
\lambda(x, y : \text{int} \times \text{bool}) \lambda(x', y' : \text{int} \times \text{bool}) (\mathbb{+}) x y, (\mathbb{lor}) x' y'
\]

Unfortunately, this reduction duplicates code. Instead, we abstract each definition of \( \text{plus} \) over the types it depends on types: If \( \text{plus} \exists(\beta, \gamma)\beta \times \gamma \) is

\[
\lambda(\beta) \lambda(\gamma) \lambda(\text{plus}_\beta : \beta \to \beta \to \beta) \lambda(\text{plus}_\gamma : \gamma \to \gamma \to \gamma) \\
\lambda(x, y : \beta \times \gamma) \lambda(x', y' : \beta \times \gamma) \text{plus}_\beta x y, \text{plus}_\gamma x' y'
\]

then the last branch of the type case is equal to

\[
\text{plus} \exists(\beta, \gamma)\beta \times \gamma \beta \gamma \ (\text{plus } \beta) \ (\text{plus } \gamma)
\]

and is

\[
\text{plus} \ (\text{int} \times \text{bool}) \leadsto \text{plus} \exists(\beta, \gamma)\beta \times \gamma \text{ int bool } \ (\text{plus } \text{int}) \ (\text{plus } \text{bool})
\]

built by passing arguments to existing functions, without code duplication.
Dictionary passing

```ocaml
let rec plus_int = (+)
and plus_bool = (lor)
and plus βγ.β×γ =
  Λ(β) Λ(γ)
  λ(plusβ : β → β → β) λ(plusγ : γ → γ → γ)
  λ(x : β) λ(y : γ) plusβ x, plusγ y in

let twice =
  Λ(α)
  λ(plusα : α → α → α) λ(x : α) plusα x x in

let plus_int×bool = plus βγ.β×γ int bool plus_int plus_bool in

twice plus_int×bool (1, true)
```

- Overloaded implementations and definitions are abstracted over unresolved overloaded symbols;
- Derived implementations are built on demand after type inference.
Dictionary passing

```ocaml
let plus_int = (+) in
let plus_bool = (lor) in
let plus βγ.β×γ = Λ(β) Λ(γ)
  λ(plusβ : β → β → β) λ(plusγ : γ → γ → γ)
  λ(x : β) λ(y : γ) plusβ x, plusγ y in
let twice =
  Λ(α)
  λ(plusα : α → α → α) λ(x : α) plusα x x in
let plus_int×bool = plus βγ.β×γ int bool plus_int plus_bool in
```

- overloaded implementations and definitions are abstracted over unresolved overloaded symbols;
- derived implementations are built on demand after type inference.
Dictionary passing

After type inference, before translation

```plaintext
def Plus α = plus : α → α → α in
let rec plus: int → int → int = (+)
    and plus: bool → bool → bool = (lor)
and plus: ∀β⟨Plus β⟩ ∀γ⟨Plus γ⟩ (β × γ) → (β × γ) → (β × γ) =
    Λ(β⟨Plus β⟩) Λ(γ⟨Plus γ⟩)
    λ(x, y : β × γ) λ(x', y' : β × γ) plus β × x', plus γ y y' in
let twice =
    Λ(α⟨Plus α⟩)
    λ(x : α) plus α x x in
twice (int × bool) (1, true)
```
Alternatively, inlining the constraint (running code)

```
let rec plus: int → int → int = (+)
and plus: bool → bool → bool = (lor)
and plus: ∀β⟨plus : β → β → β⟩ ∀γ⟨plus : γ → γ → γ⟩
   (β × γ) → (β × γ) → (β × γ) =
   Λ(β⟨plus : β → β → β⟩) Λ(γ⟨plus : γ → γ → γ⟩)
   λ(x, y : β × γ) λ(x′, y′ : β × γ) plus β × x′, plus γ y y′ in

let twice =
   Λ(α⟨plus : α → α → α⟩)
   λ(x : α) plus α × x in

twice plus(int × bool) (1, true)
```
Alternatively, inlining the constraint (source code)

\[
\begin{align*}
\text{let rec } & \text{plus: int } \rightarrow \text{ int } \rightarrow \text{ int } = (+) \\
& \text{and plus: bool } \rightarrow \text{ bool } \rightarrow \text{ bool } = (\text{lor}) \\
& \text{and plus: } \forall \beta \langle \text{plus : } \beta \rightarrow \beta \rightarrow \beta \rangle \ \forall \gamma \langle \text{plus : } \gamma \rightarrow \gamma \rightarrow \gamma \rangle \\
& \quad (\beta \times \gamma) \rightarrow (\beta \times \gamma) \rightarrow (\beta \times \gamma) = \\
& \quad \lambda(x, y) \ \lambda(x', y') \quad \text{plus } \ x \ x', \ plus \ y \ y' \ \text{in} \\
\text{let twice } = \\
& \quad \lambda(x) \quad \text{plus } \ x \ x \ \text{in} \\
\text{twice } & \text{ (1, true)}
\end{align*}
\]
Generalities

Implementation strategies

System \textbf{OML}

Qualified types

Type classes

Design space
System **OML**

A restrictive form of overloading

**Short description**

- System **OML** is a simple but monolithic system for overloading
  - Its specification is concise.
  - It is not a framework, i.e. everything is hard-wired in the design.
- Non overlapping definitions, hence (quasi)-untyped semantics and principal types.
- Single argument resolution.
- Dictionary passing semantics.
- Overloaded definitions need not have a common type scheme.
  e.g. one may overload $u : \text{int} \rightarrow \text{bool}$ and $u : \text{string} \rightarrow \text{int} \rightarrow \text{int}$

See Odersky et al. (1995)
System OML

\[\begin{align*}
z & ::= \ x \mid u \\
v & ::= \ z \mid \lambda(x) \ a \\
a & ::= \ v \mid a \ a \mid \text{let } x = a \ \text{in } a \\
p & ::= \ a \mid \text{def } u : \sigma = v \ \text{in } p
\end{align*}\]

Symbols
Value forms
Expressions
Overloaded definitions

- We distinguish overloaded symbols \( u \) from other variables.
- Expressions are as usual, but a program \( p \) starts with a sequence of toplevel overloaded definitions:
  \[
  \text{def } u_1 : s_1 = v_1 \ \text{in } \ldots \ \text{def } u_n : s_n = v_n \ \text{in } a
  \]
  which should be understood as if recursively defined:
  \[
  \text{let rec } u_1 : s_1 = v_1 \ \text{and } \ldots \ u_n : s_n = v_n \ \text{in } a
  \]
  The notation reflects more the way they will be compiled, by abstracting over all unresolved overloaded symbols.
- Only values can be overloaded.
**System OML**

### Types

\[ \tau ::= \alpha \mid \tau \rightarrow \tau \mid c(\bar{\tau}) \]  

\[ \rho_\alpha ::= \emptyset \mid u : \alpha \rightarrow \tau ; \rho_\alpha \]  

\[ \sigma ::= \tau \mid \forall \alpha \langle \rho_\alpha \rangle \sigma \]

### Comments

- Types are as in ML. However, each polymorphic variable of a type scheme is restricted by a (possibly empty) constraint.

- Type constraints \( \rho_\alpha \) are record-like types whose labels are distinct overloaded symbols. Intuitively, a constraint for \( \alpha \) specifies the types of overloaded symbols that can be applied to a value of type \( \alpha \).

- When \( \rho_\alpha \) is empty, we recover ML type schemes.
System OML

Overloaded definitions

Type schemes of overloaded definitions

They must be closed and of the form $\sigma_c$

$$\forall \alpha_1 \langle \rho_{\alpha_1} \rangle \ldots \forall \alpha_n \langle \rho_{\alpha_n} \rangle \ c(\bar{\alpha}_1' \ldots \alpha_2') \rightarrow \tau$$

where $\alpha'_1 \ldots \alpha'_n$ is a permutation of $\alpha_1 \ldots \alpha_n$.

Important

- The choice of an overloaded definition is fully determined by the topmost constructor of the first argument.
- This helps having principal types and a deterministic semantics.
- This also facilitates overloading resolution and coverage checking.
System OML

Typing contexts

\[ \Gamma ::= \ z : \sigma \mid u : \sigma \]

The typing relation

\[ \Gamma \vdash a : \sigma \]

Contain ML typing rules

\textsc{Var}

\begin{align*}
\text{z} : \sigma & \in \Gamma \\
\hline
\Gamma \vdash \text{z} : \sigma
\end{align*}

\textsc{Let}

\begin{align*}
\Gamma & \vdash a : \sigma \\
\Gamma, x : \sigma & \vdash a' : \tau \\
\hline
\Gamma & \vdash \text{let } x = a \text{ in } a' : \tau
\end{align*}

\textsc{Arrow-Intro}

\begin{align*}
x & \notin \Gamma \\
\Gamma, x : \tau & \vdash a : \tau' \\
\hline
\Gamma & \vdash \lambda(x) a : \tau \rightarrow \tau'
\end{align*}

\textsc{Arrow-Elim}

\begin{align*}
\Gamma & \vdash a_1 : \tau_2 \rightarrow \tau_2 \\
\Gamma & \vdash a_2 : \tau_2 \\
\hline
\Gamma & \vdash a_2 \ a_1 : \tau_1
\end{align*}
System OML

Overloaded definitions

\[
\begin{align*}
\text{DEF} & \quad \Gamma \vdash u \not\# \sigma_\pi \\
& \quad \Gamma \vdash a : \sigma_\pi \\
& \quad \Gamma, u : \sigma_\pi \vdash p : \sigma \\
\hline
& \quad \Gamma \vdash \text{def } u : \sigma_\pi = a \text{ in } p : \sigma
\end{align*}
\]

We write \( \Gamma \vdash u \not\# \sigma_\pi \) to mean that for all \( u : \sigma' \in \Gamma \), \( \sigma' \) and \( \sigma_\pi \) have different topmost type constructors.

This implies, in particular, that overloading definitions of \( \Gamma \) are never overlapping.
Introduction and elimination of polymorphism

\[ \text{All-Intro} \]
\[
\Gamma, \rho_\alpha \vdash v : \sigma \\
\hline
\Gamma \vdash \forall \alpha \langle \rho_\alpha \rangle \sigma
\]

\[ \text{All-Elim} \]
\[
\Gamma \vdash \forall \alpha \langle \rho_\alpha \rangle \sigma \\
\Gamma \vdash \rho_\alpha [\tau/\alpha] \\
\hline
\Gamma \vdash a : \sigma[\tau/\alpha]
\]

As in ML, we restrict generalization to value forms.

Overloaded symbols

Overloaded symbols are introduced in \( \Gamma \) by rules Def or All-Intro. They can be retrieved by rule Var and used directly, or indirectly via rule All-Elim.
An example of typing is given below together with the translation to ML.
System OML

Compilation to ML

Judgment $\Gamma \vdash p : \sigma \triangleright M$

We compile a program $p$ into an ML expression $M$ (which is also an OML expression) based on the typing derivation.

The definition of the translation is by an instrumenting the typing rules.

Easy cases

**VAR**

\[
\frac{z : \sigma \in \Gamma}{\Gamma \vdash x : \sigma \triangleright x}\]

**LET**

\[
\frac{\Gamma \vdash a : \sigma \triangleright M \quad \Gamma, x : \sigma \vdash a' : \tau \triangleright M'}{\Gamma \vdash \text{let } x = a \text{ in } a' : \tau \triangleright \text{let } x = M \text{ in } M'}\]

**Arrow-Intro**

\[
\frac{x \notin \Gamma \quad \Gamma, x : \tau \vdash a : \tau' \triangleright M}{\Gamma \vdash \lambda(x) \ a : \tau \rightarrow \tau' \triangleright \lambda(x) \ M}\]

**Arrow-Elim**

\[
\frac{\Gamma \vdash a_1 : \tau_2 \rightarrow \tau_1 \triangleright M_1 \quad \Gamma \vdash a_2 : \tau_2 \triangleright M_2}{\Gamma \vdash a_1 \ a_2 : \tau_1 \triangleright M_1 \ M_2}\]
Introducing and using overloaded definitions

\[ \text{DEF} \]

\[
\Gamma \vdash u \# \sigma_{\pi} \\
\Gamma \vdash a : \sigma_{\pi} \triangleright M_{\pi} \\
\Gamma, u : \sigma_{\pi} \vdash p : \sigma \triangleright M \\
\Gamma \vdash \text{def } u : \sigma_{\pi} = a \text{ in } p : \sigma \triangleright \text{let } x_{\sigma_{\pi}}^{u} = M_{\pi} \text{ in } M
\]

\[ \text{VAR-OVER} \]

\[
\Gamma \vdash u : \sigma \in \Gamma \\
\Gamma \vdash u : \sigma \triangleright x_{\sigma}^{u}
\]

Introducing and using polymorphism

\[ \text{ALL-INTRO} \]

\[
\Gamma, u_{1} : \tau_{1}, \ldots, u_{n} : \tau_{n} \vdash a : \sigma \triangleright M \quad \alpha \notin \Gamma \\
\Gamma \vdash \forall \alpha \langle u_{1} : \tau_{1}, \ldots, u_{n} : \tau_{n} \rangle \sigma \triangleright \lambda(x_{\tau_{1}}^{u_{1}}) \ldots \lambda(x_{\tau_{n}}^{u_{n}}) M
\]

\[ \text{ALL-ELIM} \]

\[
\Gamma \vdash a : \forall \alpha \langle u_{1} : \tau_{1}, \ldots, u_{n} : \tau_{n} \rangle \sigma \triangleright M \\
\Gamma \vdash (u_{1} : \tau_{1}, \ldots, u_{n} : \tau_{n})[\tau / \alpha] \\
\Gamma \vdash a : \sigma[\tau / \alpha] \triangleright M \ x_{\tau_{1}[\tau / \alpha]}^{u_{1}} \ldots x_{\tau_{n}[\tau / \alpha]}^{u_{n}}
\]
The previous example, twice

The typing derivation is as follows. We write $\tau^1$ for $\tau$ and $\tau^{n+1}$ for $\tau \rightarrow \tau^n$; $\Gamma$ for $x : \alpha$, plus: $\alpha^3$; and $\Gamma_0$ for some non conflicting context.

\[
\begin{align*}
\Gamma_0 \Gamma \vdash \text{plus} : \alpha^3 \triangleright x_{\alpha^3} & \quad \Gamma_0 \Gamma \vdash x : \alpha \triangleright x \\
\Gamma_0 \Gamma \vdash \text{plus} \times x : \alpha \triangleright x_{\alpha^3} \times x \\
\Gamma_0, \text{plus: } \alpha^3 \vdash \lambda(x) \text{ plus } x \times x : \alpha \rightarrow \alpha \triangleright \lambda(x) x_{\alpha^3}^\text{plus} \times x \\
\Gamma_0 \vdash \lambda(x) \text{ plus } x \times : \forall \alpha \langle \text{plus: } \alpha^3 \rangle \alpha \rightarrow \alpha \triangleright \lambda(x) x_{\alpha^3}^\text{plus} \lambda(x) x_{\alpha^3}^\text{plus} \times x
\end{align*}
\]
Compilation of OML

Let $\Gamma_0$ stand for

$$\text{plus: } \text{int}^3, \text{plus: } \text{bool}^3, \text{plus: } \forall \beta \langle \text{plus: } \beta^3 \rangle \forall \gamma \langle \text{plus: } \gamma^3 \rangle (\beta \times \gamma)^3$$

and $\Gamma_1$ be $\Gamma_0$, $\text{twice : } \forall \alpha \langle \text{plus: } \alpha^3 \rangle \alpha \to \alpha$. We have:

\[
\begin{align*}
\text{All-Elim} \\
\Gamma_1 \vdash \text{plus: } \forall \beta \langle \text{plus: } \beta^3 \rangle \forall \gamma \langle \text{plus: } \gamma^3 \rangle (\beta \times \gamma)^3 \triangleright x_{\sigma}^	ext{plus} \\
\Gamma_1 \vdash \text{plus: } \text{int}^3 \triangleright x_{\text{int}^3}^\text{plus} & \quad \Gamma_1 \vdash \text{plus: } \text{bool}^3 \triangleright x_{\text{bool}^3}^\text{plus} \\
\hline
\Gamma_1 \vdash \text{plus: } (\text{int} \times \text{bool})^3 \triangleright x_{\sigma}^\text{plus} \times x_{\text{int}^3}^\text{plus} \times x_{\text{bool}^3}^\text{plus}
\end{align*}
\]

Therefore,

\[
\begin{align*}
\text{All-Elim} \\
\Gamma_0 \vdash \text{twice : } (\text{int} \times \text{bool})^2 \triangleright \text{twice } (x_{\sigma}^\text{plus} \times x_{\text{int}^3}^\text{plus} \times x_{\text{bool}^3}^\text{plus}) \\
\Gamma_0 \vdash \text{twice } (1, \text{true}) : (\text{int} \times \text{bool}) \triangleright \text{twice } (x_{\sigma}^\text{plus} \times x_{\text{int}^3}^\text{plus} \times x_{\text{bool}^3}^\text{plus}) (1, \text{true})
\end{align*}
\]
Properties

Type preservation

The translation is type preserving. This result is easy to establish.

Coherence

The translation is based on derivations and returns different programs for different derivations. Does the semantics depend on the typing derivation?

Fortunately, this is not the case. Two translations of the same program based on two different typing derivations are observationally equivalent. We say that the semantics is coherent.

This result is difficult and tedious and has in fact only been proved for variants of the language. So far, it is only a conjecture for OML.
System OML

Principal types

There are principal types in OML, thanks to the restriction on the type schemes of overloaded functions.

Monolithic type inference

Principal types can be inferred by solving unification constraints on the fly as in Damas-Milner. The main difference is to treat applications of overloaded functions by generating a fresh overloaded assumption (an overloaded variable with a type constraint) in the typing environment.

The non-overlapping of typing assumptions on overloaded variables implies that the overloaded assumptions may have to be transformed when a variable is instantiated during unification: assumptions may have to be merged triggering further unifications, or to be resolved and removed from the typing environment, but perhaps introducing other assumptions.
Asume $\Gamma_0$ is neg: $\textit{int}^2$, neg: $\textit{bool}^2$, neg: $\forall \beta \langle \text{neg} : \beta^2 \rangle (\beta \text{ list})^2$.

To solve the typing contraint $\Gamma_0 \vdash (\lambda(x) \text{ neg } x) \ 1 : \tau$, we infer

\[
\Gamma_0, \text{neg: } \alpha \rightarrow \beta, \ x : \alpha \vdash \text{neg } x : \beta
\]

\[
\Gamma_0, \text{neg: } \alpha \rightarrow \beta \vdash \lambda(x) \text{ neg } x : \alpha \rightarrow \beta
\]

- The most general judgment uses the asumption neg: $\alpha \rightarrow \beta$. 
Asume $\Gamma_0$ is $\text{neg}: \text{int}^2, \text{neg}: \text{bool}^2, \text{neg}: \forall \beta \langle \text{neg} : \beta^2 \rangle (\beta \text{ list})^2$.

To solve the typing constraint $\Gamma_0 \vdash (\lambda(x) \text{ neg } x) \ 1 : \tau$, we infer:

$$\Gamma_0, \text{neg}: \text{int} \rightarrow \beta, x : \text{int} \vdash \text{neg } x : \beta$$

$$\Gamma_0, \text{neg}: \text{int} \rightarrow \beta \vdash \lambda(x) \text{ neg } x : \text{int} \rightarrow \beta$$

$\text{(Informally)}$

- The most general judgment uses the asumption $\text{neg}: \text{int} \rightarrow \beta$.
- We must unify $\text{int}$ with $\text{int}$ in order to type the application.
### System OML

Type inference by example

Assume $\Gamma_0$ is $\text{neg}: \text{int}^2$, $\text{neg}: \text{bool}^2$, $\text{neg}: \forall \beta \langle \text{neg} : \beta^2 \rangle (\beta \text{ list})^2$.

To solve the typing constraint $\Gamma_0 \vdash (\lambda(x) \text{ neg } x) \mathbf{1} : \tau$, we infer

$$\frac{\Gamma_0, \text{neg}: \text{int} \rightarrow \text{int}, x : \text{int} \vdash \text{neg } x : \text{int}}{\Gamma_0, \text{neg}: \text{int} \rightarrow \text{int} \vdash \lambda(x) \text{ neg } x : \text{int} \rightarrow \text{int}}$$

(Informally)

- The most general judgment uses the assumption $\text{neg}: \text{int} \rightarrow \text{int}$.
- We must unify $\text{int}$ with $\text{int}$ in order to type the application.
- There is a hidden well-formedness constraint: Since $\Gamma_0$ contains a assumption $\text{neg} : \text{int}^2$, it must be merged with the other assumption $\text{neg} : \text{int} \rightarrow \text{int}'$, which forces the unification of $\text{int}'$ with $\text{int}$,
System OML

Type inference by example

Asume $\Gamma_0$ is neg: $int^2$, neg: $bool^2$, neg: $\forall \beta \langle neg : \beta^2 \rangle (\beta \text{ list})^2$. To solve the typing constraint $\Gamma_0 \vdash (\lambda(x) \ neg \ x) \ 1 : \tau$, we infer

$$
\frac{
\Gamma_0, x : int \vdash neg \ x : int
}{
\Gamma_0 \vdash \lambda(x) \ neg \ x : int \rightarrow int
}
$$

- The most general judgment
- We must unify $int$ with $int$ in order to type the application.
- There is a hidden well-formedness constraint: Since $\Gamma_0$ contains a assumption $neg : int^2$, it must be merged with the other assumption $neg : int \rightarrow int'$, which forces the unification of $int'$ with $int$,
- Removing repeated typing assumptions from the typing environment
Asume $\Gamma_0$ is neg: $\texttt{int}^2$, neg: $\texttt{bool}^2$, neg: $\forall \beta \langle \text{neg} : \beta^2 \rangle (\beta \texttt{list})^2$.

To solve the typing constraint $\Gamma_0 \vdash (\lambda(x) \text{neg} x) \ 1 : \tau$, we infer

\[
\Gamma_0, x : \texttt{int} \vdash \text{neg} x : \texttt{int}
\]

\[
\frac{
\Gamma_0 \vdash \lambda(x) \text{neg} x : \texttt{int} \rightarrow \texttt{int} \quad \Gamma_0 \vdash 1 : \texttt{int}
}{\Gamma_0 \vdash (\lambda(x) \text{neg} x) \ 1 : \texttt{int}}
\]

- The most general judgment
- We must unify $\texttt{int}$ with $\texttt{int}$ in order to type the application.
- There is a hidden well-formedness constraint: Since $\Gamma_0$ contains a assumption $\text{neg} : \texttt{int}^2$, it must be merged with the other assumption $\text{neg} : \texttt{int} \rightarrow \texttt{int}'$, which forces the unification of $\texttt{int}'$ with $\texttt{int}$.
- Removing repeated typing assumptions from the typing environment
- Finally...
<table>
<thead>
<tr>
<th>Generalities</th>
<th>Implementation strategies</th>
<th>System</th>
<th>OML</th>
<th>Qualified types</th>
<th>Type classes</th>
<th>Design space</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Generalities**

**Implementation strategies**

**System OML**

**Qualified types**

**Type classes**

**Design space**
Qualified types

A general framework

Qualified types are a general framework for inferring types of partial functions. Overloading is just a particular case of qualified types.

Idea: introduce predicates that restrict the set of types a variable may range over. For instance, $Plus \alpha$ means that $\alpha$ can only be instantiated at a type $\tau$ such that there exists a definition for $plus$ of type $\tau \rightarrow \tau \rightarrow \tau$.

Parameterize over the constraint domain

- Typing rules use a separate judgement to state when constraints are satisfied, which depends on the constraint domain.
- This separates the resolution of constraints from their generation.
- It also internalizes simplification and optimization of constraints.

See Jones (1992)
Generalities

Implementation strategies

System OML

Qualified types

Type classes

Design space
Types classes

What are they?

A mechanism for building overloaded definitions is a more structured way.

- Overloaded definitions are grouped into type classes.
- A type class defines a set of identifiers that belong to that class.
- An instance of a type class provides, for a specific type, definitions for all elements of the class.
- A type class may have default definitions, which are not overloaded definitions, but defaults for overloaded definitions when taking instances of that class.

Type classes are more convenient to use than plain unstructured overloading and keep types more concise, both for defining new implementations and writing assumptions.

Type classes can be compiled away into qualified types.
Module-based overloading

Modules can be used instead type classes to group overloaded definitions.

- A type component distinguishes the type at which overloaded instances are provided.
- Basic instances are basic modules.
- Derivable instances are defined as functors.
- Modules can be declared as overloading their definitions.
- The basic overloaded mechanism can then be used to resolved overloaded names.
- Functor application is implicitly used to generate derived instances.

The advantage of module-based overloading over type classes is that modules already organize name scoping and type abstraction.

However, the underlying overloading engine is essentially the same.

See Dreyer et al. (2007)
Generalities

Implementation strategies

System OML

Qualified types

Type classes

Design space
Problems and challenges

**Simplification and optimizations**

Because generalization and instantiation induces additional abstractions and applications, it is important to use them as little as necessary, while retaining principal types. This constrasts with ML where it does not matter. (Coherence implies that the semantics does not depend on the derivation, but the efficiency does, indeed.)

**Efficiency of implementation techniques**

The pros and cons of the different implementation techniques are well-understood, but they is no available detailed comparison of their respective performance, with different optimization techniques.
Remaining problems and challenges

Overlapping instances

The semantics depend on types. This does not work well with type inference. Type inference (checking coverage) may also become expensive or even undecidable.

Overloaded on return types

The semantics depends on types and type inference.

Overloading with local scope

This introduced a potential conflict in the resolution: An overloaded symbol with a local implementation can either be resolved immediately or left generic to be resolved later, in the context of use, perhaps with another implementation. This choice cannot be left implicit.
Remaining problems and challenges

**Design space**

Because some restrictions must be imposed on the shape and overlapping of type definitions, there are many variations in the design space.

See Jones et al. (1997)
Ideas to bring back home

Overloading is quite useful
- Static overloading may already significantly alleviate the notations
- However, it is too strong a restriction, which may often be frustrating
- Dynamic overloading enables polytypic programming

Overloading is well-understood
- Long, positive experience with Haskell.
- Perhaps, more restrictive forms of overloading would be acceptable.

It always require some compromises
- When definitions are overlapping, the semantics depends on typechecking. With powerful type inference, the semantics may not always be obvious to the programmer.
- There is still place for other, perhaps better compromises.


