Modularity, *Surcharge*

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

Didier Rémy

INRIA-Rocquencourt

Janvier 27, 2009

Overloading Why?

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 α may have an implementation depending on the type of α. For instance, a marshaling function of type $\forall\alpha.\alpha \rightarrow$ *string* may execute different code for each base type α.

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 How?

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 — Resolution strategies

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.

Overloading — Static resolution

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 — Static resolution

In SML

Definitions are primitive (numerical operators, record accesses).

Typechecking fails if overloading cannot be resolved at outermost let-definitions. For example, `let` $\mathit{twice}\ x = x + x$ 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.

Overloading Static resolution

Limits

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

Indeed, functions such as $\lambda(x)\ x + x$ 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.

Overloading — Dynamic resolution

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 + instersection 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 Dynamic resolution

Pass unresolved implementations as extra arguments

- Abstract over unresolved overloaded symbols and pass them later when then 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.

Dynamic overloading — Running example

Untyped code

$$
\begin{array}{l}
\text{let rec } \mathit{plus} \ = (+) \\
\quad \text{and } \mathit{plus} \ = (\mathit{lor}) \\
\quad \text{and } \mathit{plus} \ = \lambda(x, y)\ \lambda(x', y')\ (\mathit{plus}\ x\ x', \mathit{plus}\ y\ y') \text{ in} \\
\text{let } \mathit{twice} = \lambda(x)\ \mathit{plus}\ \ x\ x \text{ in} \\
\mathit{twice}\ (1, \mathit{true})
\end{array}
$$

It should indeed evaluate to $(1 + 1, \mathit{true}\ \mathit{lor}\ \mathit{true})$, i.e. $(2, \mathit{true})$, whatever the implementation strategy.

Church style System F with type matching

Syntax

$$
\begin{array}{lcll}
a & ::= & a \mid \lambda(x)\, a \mid a\,(a) \mid \Lambda(\alpha)\, a \mid a\,(\tau) & \text{System F} \\
 & \mid & \text{match } \tau \text{ with } \langle \pi_1 \Rightarrow a_1 \ldots \mid \pi_n \Rightarrow a_n \rangle & \text{Typecase} \\
\pi & ::= & \tau \mid \exists(\alpha)\pi & \text{Type patterns}
\end{array}
$$

Reduction: as in System F, plus the redex:

$$
\frac{\tau = \tau_i[\bar{\tau}'_i/\bar{\alpha}_i]}{\text{match } \tau \text{ with } \langle \pi_1 \Rightarrow a_1 \ldots \mid \exists(\bar{\alpha}_i)\tau_i \Rightarrow a_i \ldots \mid \pi_n \Rightarrow a_n \rangle \leadsto a_i[\bar{\tau}'_i/\bar{\alpha}_i]}
$$

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

$$
\frac{\Gamma \vdash \tau \qquad \Gamma, \bar{\alpha}_i \vdash \tau_i \qquad \Gamma, \bar{\alpha}_i \vdash a_i : \tau'}{\Gamma \vdash \text{match } \tau \text{ with } \langle \pi_1 \Rightarrow a_1 \ldots \mid \exists(\bar{\alpha}_i)\tau_i \Rightarrow a_i \ldots \mid \pi_n \Rightarrow a_n \rangle \leadsto a_i : \tau'}
$$

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 match s 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 Example

Exhaustiveness is not enforced

$$
\begin{array}{l}
\text{let rec } \mathit{plus} = \\
\quad \Lambda(\alpha \qquad\qquad) \\
\qquad \text{match } \alpha \text{ with } \langle \\
\qquad \mid \mathit{int} \Rightarrow (+) \\
\qquad \mid \mathit{bool} \Rightarrow (\mathit{lor}) \\
\qquad \mid \exists(\beta \qquad\qquad, \gamma \qquad\qquad)\ \beta \times \gamma \Rightarrow \\
\qquad\quad \lambda(x, y : \beta \times \gamma)\ \lambda(x', y' : \beta \times \gamma)\ \mathit{plus}\ \beta\ x\ x', \mathit{plus}\ \gamma\ y\ y' \\
\qquad \rangle \text{ in} \\
\text{let } \mathit{twice} = \Lambda(\alpha \qquad\qquad)\ \lambda(x : \alpha)\ \mathit{plus}\ \alpha\ x\ x \text{ in} \\
\mathit{twice}\ (\mathit{int} \times \mathit{bool})\ (1, \mathit{true})
\end{array}
$$

Overloading with typecase

Example

The domain may be restricted by a type constraint

$$
\begin{array}{l}
\text{let rec } \mathit{plus} = \\
\quad \Lambda(\alpha\langle \mathit{Plus}\ \alpha\rangle) \\
\qquad \text{match } \alpha \text{ with } \langle \\
\qquad \mid \mathit{int} \Rightarrow (+) \\
\qquad \mid \mathit{bool} \Rightarrow (\mathit{lor}) \\
\qquad \mid \exists(\beta\langle \mathit{Plus}\ \beta\rangle,\ \gamma\langle \mathit{Plus}\ \gamma\rangle)\ \beta \times \gamma \Rightarrow \\
\qquad\quad \lambda(x, y : \beta \times \gamma)\ \lambda(x', y' : \beta \times \gamma)\ \mathit{plus}\ \beta\ x\ x', \mathit{plus}\ \gamma\ y\ y' \\
\qquad \rangle \text{ in} \\
\text{let } \mathit{twice} = \Lambda(\alpha\langle \mathit{Plus}\ \alpha\rangle)\ \lambda(x : \alpha)\ \mathit{plus}\ \alpha\ x\ x \text{ in} \\
\mathit{twice}\ (\mathit{int} \times \mathit{bool})\ (1, \mathit{true})
\end{array}
$$

Overloading with typecase — Example

The type predicate *Plus* α is defined by induction

$$\begin{array}{l}
\textit{Plus int}; \quad \textit{Plus bool}; \\
\textit{Plus}\ \alpha \Rightarrow \textit{Plus}\ \beta \Rightarrow \textit{Plus}\ (\alpha \times \beta) \\
\mathsf{let\ rec}\ \textit{plus} = \\
\quad \Lambda(\alpha\langle \textit{Plus}\ \alpha\rangle) \\
\qquad \mathsf{match}\ \alpha\ \mathsf{with}\ \langle \\
\qquad \mid \textit{int} \Rightarrow (+) \\
\qquad \mid \textit{bool} \Rightarrow (\textit{lor}) \\
\qquad \mid \exists(\beta\langle \textit{Plus}\ \beta\rangle,\ \gamma\langle \textit{Plus}\ \gamma\rangle)\ \beta \times \gamma \Rightarrow \\
\qquad\quad \lambda(x, y : \beta \times \gamma)\ \lambda(x', y' : \beta \times \gamma)\ \textit{plus}\ \beta\ x\ x', \textit{plus}\ \gamma\ y\ y' \\
\qquad \rangle\ \mathsf{in} \\
\mathsf{let}\ \textit{twice} = \Lambda(\alpha\langle \textit{Plus}\ \alpha\rangle)\ \lambda(x : \alpha)\ \textit{plus}\ \alpha\ x\ x\ \mathsf{in} \\
\textit{twice}\ (\textit{int} \times \textit{bool})\ (1, \textit{true})
\end{array}$$

Typecase Typing rules

Checking for satisfiability

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

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

can be read as deduction rules:

$$\frac{}{\textit{Plus int}}\ \textsc{PlusInt} \qquad \frac{}{\textit{Plus bool}}\ \textsc{PlusBool} \qquad \frac{\textit{Plus}\ \alpha \qquad \textit{Plus}\ \beta}{\textit{Plus}\ (\alpha \times \beta)}\ \textsc{PlusProd}$$

We can build (infer) the following derivation:

$$\textsc{PlusProd}\ \frac{\overset{\textsc{PlusInt}}{\textit{Plus int}} \qquad \overset{\textsc{PlusBool}}{\textit{Plus bool}}}{\textit{Plus}\ (\textit{int} \times \textit{bool})} \quad \triangleq \quad \textsc{PlusProd} \quad \textsc{PlusInt} \quad \textsc{PlusBo}$$

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

Dictionary passing

Running example

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

$$\begin{array}{l} \mathit{plus}\ (\mathit{int} \times \mathit{bool}) \leadsto \\ \lambda(x, y : \mathit{int} \times \mathit{bool})\ \lambda(x', y' : \mathit{int} \times \mathit{bool})\ \mathit{plus}\ \mathit{int}\ x\ y, \mathit{plus}\ \mathit{bool}\ x'\ y' \leadsto \\ \lambda(x, y : \mathit{int} \times \mathit{bool})\ \lambda(x', y' : \mathit{int} \times \mathit{bool})\ (+)\ x\ y, (\mathit{lor})\ x'\ y' \end{array}$$

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

$$\begin{array}{l} \Lambda(\beta)\ \Lambda(\gamma)\ \lambda(\mathit{plus}_\beta : \beta \to \beta \to \beta)\ \lambda(\mathit{plus}_\gamma : \gamma \to \gamma \to \gamma) \\ \qquad \lambda(x, y : \beta \times \gamma)\ \lambda(x', y' : \beta \times \gamma)\ \mathit{plus}_\beta\ \ x\ y, \mathit{plus}_\gamma\ \ x'\ y' \end{array}$$

then the last branch of the type case is equal to $\mathit{plus}_{\exists(\beta,\gamma)\beta\times\gamma}\ \beta\ \gamma\ (\mathit{plus}\ \beta)\ (\mathit{plus}\ \gamma)$ and is

$$\mathit{plus}\ (\mathit{int} \times \mathit{bool}) \leadsto \mathit{plus}_{\exists(\beta,\gamma)\beta\times\gamma}\ \mathit{int}\ \mathit{bool}\ (\mathit{plus}_{\mathit{int}})\ (\mathit{plus}_{\mathit{bool}})$$

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

Dictionary passing Running example

$$
\begin{array}{l}
\text{let rec } \mathit{plus}_{\mathit{int}} = (+) \\
\quad \text{and } \mathit{plus}_{\mathit{bool}} = (\mathit{lor}) \\
\quad \text{and } \mathit{plus}_{\exists \beta\gamma.\beta\times\gamma} = \\
\qquad\qquad \Lambda(\beta)\ \Lambda(\gamma) \\
\qquad\qquad\quad \lambda(\mathit{plus}_\beta : \beta \to \beta \to \beta)\ \lambda(\mathit{plus}_\gamma : \gamma \to \gamma \to \gamma) \\
\qquad\qquad\qquad \lambda(x : \beta)\ \lambda(y : \gamma)\ \mathit{plus}_\beta\ x, \mathit{plus}_\gamma\ y \text{ in} \\
\text{let } \mathit{twice} = \\
\qquad\qquad \Lambda(\alpha) \\
\qquad\qquad\quad \lambda(\mathit{plus}_\alpha : \alpha \to \alpha \to \alpha)\ \lambda(x : \alpha)\ \mathit{plus}_\alpha\ x\ x \text{ in} \\
\text{let } \mathit{plus}_{\mathit{int}\times\mathit{bool}} = \mathit{plus}_{\exists(\beta,\gamma)\beta\times\gamma}\ \mathit{int}\ \mathit{bool}\ \mathit{plus}_{\mathit{int}}\ \ \mathit{plus}_{\mathit{bool}} \text{ in} \\
\mathit{twice}\ \mathit{plus}_{\mathit{int}\times\mathit{bool}}\ \ (1, \mathit{true})
\end{array}
$$

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

Dictionary passing — Running example

$$
\begin{array}{l}
\text{let} \quad plus_{int} = (+) \ \text{in} \\
\text{let} \quad plus_{bool} = (lor) \ \text{in} \\
\text{let} \quad plus_{\exists\beta\gamma.\beta\times\gamma} = \\
\quad \Lambda(\beta)\ \Lambda(\gamma) \\
\quad\quad \lambda(plus_\beta : \beta \to \beta \to \beta)\ \lambda(plus_\gamma : \gamma \to \gamma \to \gamma) \\
\quad\quad\quad \lambda(x : \beta)\ \lambda(y : \gamma)\ plus_\beta\ x, plus_\gamma\ y \ \text{in} \\
\text{let}\ twice = \\
\quad \Lambda(\alpha) \\
\quad\quad \lambda(plus_\alpha : \alpha \to \alpha \to \alpha)\ \lambda(x : \alpha)\ plus_\alpha\ x\ x \ \text{in} \\
\text{let}\ plus_{int\times bool} = plus_{\exists(\beta,\gamma)\beta\times\gamma}\ int\ bool\ plus_{int}\ \ plus_{bool}\ \ \text{in} \\
twice\ plus_{int\times bool}\ \ (1, true)
\end{array}
$$

Definitions are non-recursive

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

Dictionary passing Example

After type inference, before translation

$$
\begin{array}{l}
\text{def } \mathit{Plus}\ \alpha = \mathit{plus} : \alpha \to \alpha \to \alpha \text{ in} \\
\text{let rec } \mathit{plus}\!: \mathit{int} \to \mathit{int} \to \mathit{int} = (+) \\
\quad \text{and } \mathit{plus}\!: \mathit{bool} \to \mathit{bool} \to \mathit{bool} = (\mathit{lor}) \\
\quad \text{and } \mathit{plus}\!: \forall \beta \langle \mathit{Plus}\ \beta \rangle\ \forall \gamma \langle \mathit{Plus}\ \gamma \rangle\ (\beta \times \gamma) \to (\beta \times \gamma) \to (\beta \times \gamma) = \\
\qquad\qquad \Lambda(\beta \langle \mathit{Plus}\ \beta \rangle)\ \Lambda(\gamma \langle \mathit{Plus}\ \gamma \rangle) \\
\qquad\qquad\quad \lambda(x, y : \beta \times \gamma)\ \lambda(x', y' : \beta \times \gamma)\ \mathit{plus}\ \beta\ x\ x', \mathit{plus}\ \gamma\ y\ y' \text{ in}
\end{array}
$$

$$
\begin{array}{l}
\text{let } \mathit{twice} = \\
\qquad\qquad \Lambda(\alpha \langle \mathit{Plus}\ \alpha \rangle) \\
\qquad\qquad\quad \lambda(x : \alpha)\ \mathit{plus}\ \alpha\ x\ x \text{ in} \\
\mathit{twice}\ (\mathit{int} \times \mathit{bool})\ (1, \mathit{true})
\end{array}
$$

Dictionary passing — Example

Alternatively, inlining the constraint (running code)

$$
\begin{array}{l}
\text{let rec } \mathit{plus} : \mathit{int} \to \mathit{int} \to \mathit{int} = (+) \\
\quad \text{and } \mathit{plus} : \mathit{bool} \to \mathit{bool} \to \mathit{bool} = (\mathit{lor}) \\
\quad \text{and } \mathit{plus} : \forall \beta \langle \mathit{plus} : \beta \to \beta \to \beta \rangle \;\; \forall \gamma \langle \mathit{plus} : \gamma \to \gamma \to \gamma \rangle \\
\qquad\qquad\qquad (\beta \times \gamma) \to (\beta \times \gamma) \to (\beta \times \gamma) = \\
\qquad\qquad \Lambda(\beta \langle \mathit{plus} : \beta \to \beta \to \beta \rangle)\; \Lambda(\gamma \langle \mathit{plus} : \gamma \to \gamma \to \gamma \rangle) \\
\qquad\qquad\quad \lambda(x, y : \beta \times \gamma)\; \lambda(x', y' : \beta \times \gamma)\; \mathit{plus}\; \beta\; x\; x', \mathit{plus}\; \gamma\; y\; y' \text{ in}
\end{array}
$$

$$
\begin{array}{l}
\text{let } \mathit{twice} = \\
\qquad\qquad \Lambda(\alpha \langle \mathit{plus} : \alpha \to \alpha \to \alpha \rangle) \\
\qquad\qquad\quad \lambda(x : \alpha)\; \mathit{plus}\; \alpha\; x\; x \text{ in} \\
\mathit{twice}\; \mathit{plus}_{(\mathit{int} \times \mathit{bool})}\; (1, \mathit{true})
\end{array}
$$

Dictionary passing — Example

Alternatively, inlining the constraint (source code)

let rec $plus$: $int \to int \to int = (+)$
  and $plus$: $bool \to bool \to bool = (lor)$
  and $plus$: $\forall\beta\langle plus : \beta \to \beta \to \beta\rangle\ \ \forall\gamma\langle plus : \gamma \to \gamma \to \gamma\rangle$
    $(\beta \times \gamma) \to (\beta \times \gamma) \to (\beta \times \gamma) =$

    $\lambda(x, y \qquad)\ \lambda(x', y' \qquad)\ plus \quad x\ x', plus \quad y\ y'$ in

let $twice =$

    $\lambda(x \quad)\ plus \quad x\ x$ in

$twice \qquad (1, true)$

System OML — A restrictive form of overloading

Short description

See Odersky et al. (1995)

- 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 $u : \mathit{int} \to \mathit{bool}$ and $u : \mathit{string} \to \mathit{int} \to \mathit{int}$

System OML — Syntax

z	$::=$	$x \mid u$	Symbols
v	$::=$	$z \mid \lambda(x)\ a$	Value forms
a	$::=$	$v \mid a\ a \mid \text{let } x = a \text{ in } a$	Expressions
p	$::=$	$a \mid \text{def } u : \sigma = v \text{ in } p$	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 — Type constraints

Types

$$\begin{array}{lll} \tau & ::= & \alpha \mid \tau \to \tau \mid c(\bar{\tau}) \\ \rho_\alpha & ::= & \emptyset \mid u : \alpha \to \tau; \rho_\alpha \\ \sigma & ::= & \tau \mid \forall \alpha \langle \rho_\alpha \rangle\ \sigma \end{array}$$

types

α-constraints

type schemes

Comments

- Types are as in ML. However, each polymorphic variable of a type scheme is restricted by a (possibly empty) constraint.
- Type constraints ρ_α are record-like types whose labels are distinct overloaded symbols. Intuitively, a constraint for α specifies the types of overloaded symbols that can be applied to a value of type α.
- When ρ_α is empty, we recover ML type schemes.

System OML — Overloaded definitions

Type schemes of overloaded definitions

They must be closed and of the form σ_c

$$\forall \alpha_1 \langle \rho_{\alpha_1} \rangle \ldots \forall \alpha_n \langle \rho_{\alpha_n} \rangle \; c(\bar{\alpha}'_1 \ldots \alpha'_2) \to \tau$$

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 rules

Typing contexts

$$\Gamma \quad ::= \quad z : \sigma \mid u : \sigma$$

The typing relation

$$\Gamma \vdash a : \sigma$$

Contain ML typing rules

$$\textsc{Var}\quad \frac{z : \sigma \in \Gamma}{\Gamma \vdash z : \sigma} \qquad \textsc{Let}\quad \frac{\Gamma \vdash a : \sigma \qquad \Gamma, x : \sigma \vdash a' : \tau}{\Gamma \vdash \mathsf{let}\ x = a\ \mathsf{in}\ a' : \tau}$$

$$\textsc{Arrow-Intro}\quad \frac{x \notin \Gamma \qquad \Gamma, x : \tau \vdash a : \tau'}{\Gamma \vdash \lambda(x)\ a : \tau \to \tau'} \qquad \textsc{Arrow-Elim}\quad \frac{\Gamma \vdash a_1 : \tau_2 \to \tau_2 \qquad \Gamma \vdash a_2 : \tau_2}{\Gamma \vdash a_2\ a_1 : \tau_1}$$

System OML

Typing rules

Overloaded definitions

$$\dfrac{\Gamma \vdash u \# \sigma_\pi \qquad \Gamma \vdash a : \sigma_\pi \qquad \Gamma, u : \sigma_\pi \vdash p : \sigma}{\Gamma \vdash \mathsf{def}\ u : \sigma_\pi = a\ \mathsf{in}\ p : \sigma}\ \textsc{Def}$$

We write $\Gamma \vdash u \# \sigma_\pi$ to mean that for all $u : \sigma' \in \Gamma$, σ' and σ_π have different topmost type constructors.

This implies, in particular, that overloading definitions of Γ are never overlapping.

System OML — Typing rules

Introduction and elimination of polymorphism

$$\textsc{All-Intro}\quad \frac{\Gamma, \rho_\alpha \vdash v : \sigma}{\Gamma \vdash \forall\alpha\langle\rho_\alpha\rangle\ \sigma} \qquad\qquad \textsc{All-Elim}\quad \frac{\Gamma \vdash \forall\alpha\langle\rho_\alpha\rangle\ \sigma \qquad \Gamma \vdash \rho_\alpha[\tau/\alpha]}{\Gamma \vdash a : \sigma[\tau/\alpha]}$$

As in ML, we restrict generalization to value forms.

Overloaded symbols

Overloaded symbols are introduced in Γ by rules DEF or ALL-INTRO. They can be retreived by rule VAR and used directly, or indirectly via rule ALL-ELIM.

Typing

Example

An example of typing is given below together with the translation to ML.

System OML — Compilation to ML

Judgment $\Gamma \vdash p : \sigma \rhd \mathcal{M}$

We compile a program p into an ML expression $\mathcal{M}$ (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

$$\text{VAR}\quad \frac{z : \sigma \in \Gamma}{\Gamma \vdash x : \sigma \rhd x}$$

$$\text{LET}\quad \frac{\Gamma \vdash a : \sigma \rhd \mathcal{M} \qquad \Gamma, x : \sigma \vdash a' : \tau \rhd \mathcal{M}'}{\Gamma \vdash \text{let } x = a \text{ in } a' : \tau \rhd \text{let } x = \mathcal{M} \text{ in } \mathcal{M}'}$$

$$\text{ARROW-INTRO}\quad \frac{x \notin \Gamma \qquad \Gamma, x : \tau \vdash a : \tau' \rhd \mathcal{M}}{\Gamma \vdash \lambda(x)\ a : \tau \to \tau' \rhd \lambda(x)\ \mathcal{M}}$$

$$\text{ARROW-ELIM}\quad \frac{\Gamma \vdash a_1 : \tau_2 \to \tau_1 \rhd \mathcal{M}_1 \qquad \Gamma \vdash a_2 : \tau_2 \rhd \mathcal{M}_2}{\Gamma \vdash a_1\ a_2 : \tau_1 \rhd \mathcal{M}_1\ \mathcal{M}_2}$$

System OML Compilation to ML

Introducing and using overloaded definitions

DEF
$$\frac{\Gamma \vdash u \# \sigma_\pi \qquad \Gamma \vdash a : \sigma_\pi \triangleright \mathcal{M}_\pi \qquad \Gamma, u : \sigma_\pi \vdash p : \sigma \triangleright \mathcal{M}}{\Gamma \vdash \text{def } u : \sigma_\pi = a \text{ in } p : \sigma \triangleright \text{let } x^u_{\sigma_\pi} = \mathcal{M}_\pi \text{ in } \mathcal{M}}$$

VAR-OVER
$$\frac{u : \sigma \in \Gamma}{\Gamma \vdash u : \sigma \triangleright x^u_\sigma}$$

Introducing and using polymorphism

ALL-INTRO
$$\frac{\Gamma, u_1 : \tau_1, \dots u_n : \tau_n \vdash a : \sigma \triangleright \mathcal{M} \qquad \alpha \notin \Gamma}{\Gamma \vdash \forall \alpha \langle u_1 : \tau_1, \dots u_n : \tau_n \rangle\ \sigma \triangleright \lambda(x^u_{\tau_1})\ \dots \lambda(x^u_{\tau_n})\ \mathcal{M}}$$

ALL-ELIM
$$\frac{\Gamma \vdash a : \forall \alpha \langle u_1 : \tau_1, \dots u_n : \tau_n \rangle\ \sigma \triangleright \mathcal{M} \qquad \Gamma \vdash (u_1 : \tau_1, \dots u_n : \tau_n)[\tau/\alpha]}{\Gamma \vdash a : \sigma[\tau/\alpha] \triangleright \mathcal{M}\ x^{u_1}_{\tau_1[\tau/\alpha]} \dots\ x^{u_n}_{\tau_n[\tau/\alpha]}}$$

Compilation of OML — Example

The previous example, twice

The typing derivation is as follows. We write τ^1 for τ and τ^{n+1} for $\tau \to \tau^n$; Γ for $x : \alpha, \text{plus: } \alpha^3$; and Γ_0 for some non conflicting context.

$$\dfrac{\dfrac{\dfrac{\Gamma_0\Gamma \vdash \text{plus } : \alpha^3 \triangleright \mathsf{x}^{\text{plus}}_{\alpha^3} \qquad \Gamma_0\Gamma \vdash x : \alpha \triangleright x}{\Gamma_0\Gamma \vdash \text{plus } x\, x : \alpha \triangleright \mathsf{x}^{\text{plus}}_{\alpha^3}\ x\ x}}{\Gamma_0, \text{plus: } \alpha^3 \vdash \lambda(x)\ \text{plus } x\, x : \alpha \to \alpha \triangleright \lambda(x)\ \mathsf{x}^{\text{plus}}_{\alpha^3}\ x\ x}}{\Gamma_0 \vdash \lambda(x)\ \text{plus } x\, x : \forall\alpha\langle \text{plus: } \alpha^3\rangle\ \alpha \to \alpha \triangleright \lambda(\mathsf{x}^{\text{plus}}_{\alpha^3})\ \lambda(x)\ \mathsf{x}^{\text{plus}}_{\alpha^3}\ x\ x}$$

Compilation of OML — Example (Cont.)

Let Γ_0 stand for

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

and Γ_1 be $\Gamma_0, \mathit{twice} : \forall \alpha \langle \text{plus: } \alpha^3 \rangle\ \alpha \to \alpha$. We have:

$$\frac{\begin{array}{c}\textsc{All-Elim}\\ \Gamma_1 \vdash \text{plus: } \forall \beta \langle \mathit{plus} : \beta^3 \rangle\ \forall \gamma \langle \text{plus: } \gamma^3 \rangle\ (\beta \times \gamma)^3 \triangleright \mathsf{x}^{\text{plus}}_{\sigma} \\ \Gamma_1 \vdash \text{plus: } \mathit{int}^3 \triangleright \mathsf{x}^{\text{plus}}_{\mathit{int}^3} \qquad \Gamma_1 \vdash \text{plus: } \mathit{bool}^3 \triangleright \mathsf{x}^{\text{plus}}_{\mathit{bool}^3}\end{array}}{\Gamma_1 \vdash \text{plus: } (\mathit{int} \times \mathit{bool})^3 \triangleright \mathsf{x}^{\text{plus}}_{\sigma}\ \mathsf{x}^{\text{plus}}_{\mathit{int}^3}\ \mathsf{x}^{\text{plus}}_{\mathit{bool}^3}}$$

Therefore,

$$\frac{\begin{array}{c}\textsc{All-Elim}\\ \Gamma_0 \vdash \mathit{twice} : (\mathit{int} \times \mathit{bool})^2 \triangleright \mathit{twice}\ (\mathsf{x}^{\text{plus}}_{\sigma}\ \mathsf{x}^{\text{plus}}_{\mathit{int}^3}\ \mathsf{x}^{\text{plus}}_{\mathit{bool}^3})\end{array}}{\Gamma_0 \vdash \mathit{twice}\ (1, \mathit{true}) : (\mathit{int} \times \mathit{bool}) \triangleright \mathit{twice}\ (\mathsf{x}^{\text{plus}}_{\sigma}\ \mathsf{x}^{\text{plus}}_{\mathit{int}^3}\ \mathsf{x}^{\text{plus}}_{\mathit{bool}^3})\ (1, \mathit{true})}$$

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 — Type inference

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 asumption (an overoaded 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 asumptions.

System OML — Type inference by example

Asume Γ_0 is neg: int^2, neg: $bool^2$, neg: $\forall\beta\langle neg : \beta^2\rangle\,(\beta \text{ list})^2$.
To solve the typing contraint $\Gamma_0 \vdash (\lambda(x)\text{ neg } x)\ 1 : \tau$, we infer

$$\frac{\Gamma_0, \text{neg}: \alpha \to \beta, x : \alpha \vdash \text{neg } x : \beta}{\Gamma_0, \text{neg}: \alpha \to \beta \vdash \lambda(x)\text{ neg } x : \alpha \to \beta}$$

- The most general judgment uses the asumption neg: $\alpha \to \beta$.

System OML — Type inference by example

Asume Γ_0 is neg: int^2, neg: $bool^2$, neg: $\forall\beta\langle neg : \beta^2\rangle\ (\beta \text{ list})^2$.
To solve the typing contraint $\Gamma_0 \vdash (\lambda(x)\ \text{neg}\ x)\ 1 : \tau$, we infer

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

(Informally)

- The most general judgment uses the asumption neg: $int \to \beta$.
- We must unify *int* with *int* in order to type the application.

System OML — Type inference by example

Asume Γ_0 is neg: int^2, neg: $bool^2$, neg: $\forall\beta\langle neg : \beta^2\rangle\,(\beta \text{ list})^2$.
To solve the typing contraint $\Gamma_0 \vdash (\lambda(x)\text{ neg } x)\ 1 : \tau$, we infer

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

(Informally)

- The most general judgment uses the asumption neg: $int \to int$.
- We must unify *int* with *int* in order to type the application.
- There is a hidden well-formedness constraint: Since Γ_0 contains a asumption $neg : int^2$, it must be merged with the other asumption $neg : int \to int'$, which forces the unification of int' with int,

System OML — Type inference by example

Asume Γ_0 is neg: int^2, neg: bool^2, neg: $\forall\beta\langle \mathit{neg} : \beta^2\rangle\,(\beta \text{ list})^2$.
To solve the typing contraint $\Gamma_0 \vdash (\lambda(x)\text{ neg } x)\ 1 : \tau$, we infer

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

- The most general judgment
- We must unify *int* with *int* in order to type the application.
- There is a hidden well-formedness constraint: Since Γ_0 contains a asumption $\mathit{neg} : \mathit{int}^2$, it must be merged with the other asumption $\mathit{neg} : \mathit{int} \to \mathit{int}'$, which forces the unification of int' with int,
- Removing repeated typing asumptions from the typing environment

System OML — Type inference by example

Asume Γ_0 is neg: int^2, neg: $bool^2$, neg: $\forall\beta\langle neg : \beta^2\rangle\,(\beta\text{ list})^2$.
To solve the typing contraint $\Gamma_0 \vdash (\lambda(x)\text{ neg } x)\ 1 : \tau$, we infer

$$\dfrac{\dfrac{\Gamma_0, x : int \vdash \text{neg } x : int}{\Gamma_0 \vdash \lambda(x)\text{ neg } x : int \to int} \qquad \dfrac{}{\Gamma_0 \vdash 1 : int}}{\Gamma_0 \vdash (\lambda(x)\text{ neg } x)\ 1 : int}$$

- The most general judgment
- We must unify *int* with *int* in order to type the application.
- There is a hidden well-formedness constraint: Since Γ_0 contains a asumption $neg : int^2$, it must be merged with the other asumption $neg : int \to int'$, which forces the unification of int' with int,
- Removing repeated typing asumptions from the typing environment
- Finally...

Qualified types

A general framework

See Jones (1992)

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* α means that α can only be instantiated at a type τ 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.

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 asumptions.

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)

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 constrats 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 frustating
- 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.

Bibliography I

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

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.

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

Bibliography II

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

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

Mark P. Jones. A theory of qualified types. In *In Fourth European Symposium on Programming*, pages 287–306. Springer-Verlag, 1992.

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

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

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

Bibliography III

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

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

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