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: numerical operations (*e.g.* *plus_int*, *plus_float*, ...); Printing functions; 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 $\forall\alpha.\alpha \to$ *string* executes different code for each base type α.

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

A typical example of a a polytypic function is the generation of random values for 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, only one of them should be used.
  Determining which one is called overloading resolution.

Main differences

- 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

- If every overloaded symbol can be statically replaced by its implementation at the appropriate type.
- This has very limited expressiveness, indeed.

Dynamic resolution

- 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. (Requires support from the runtime.)

Overloading

Static resolution

In SML

Definitions are primitive (numerical operators, record access).

Typechecking fails if overloading cannot be resolved at outermost let-definitions. For example, `let` *twice* $x = x + x$ is rejected in SML, because $+$ could be 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 may always be resolved non-ambiguously based on subtyping by choosing the most specific definition whenever several ones match.

However, at runtime an argument may have a subtype of the type known at compile time, 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.

The solution is indeed to use 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.

Pass unresolved implementations as extra arguments

- Abstract over unresolved overloaded symbols and pass them later when then can be resolved.
- This can be done based on the typing derivation.
- Then 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

Example

Untyped code

$$
\begin{array}{l}
\text{let rec } \mathit{plus} \;=\; (+) \\
\quad \text{and } \mathit{plus} \;=\; (+.) \\
\quad \text{and } \mathit{plus} \;=\; \lambda(x,y)\ \lambda(x',y')\ (\mathit{plus}\ x\ x', \mathit{plus}\ y\ y') \text{ in} \\
\text{let } \mathit{twice} = \lambda(x)\ \mathit{plus}\ \ x\ x \text{ in} \\
\mathit{twice}\ (1, 0.5)
\end{array}
$$

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:

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

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

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

Church style System F with type matching

Soundness for System F with type matching.

- Subject-reduction holds
- Progress: the type system does not ensure exhaustiveness of type matching. Hence, progress does not hold in this version. This can be fixed by either:
  - enforcing the user to provide a default case, in case of mismatch, *e.g.* using a construction, such as match s with $\langle \pi \Rightarrow a \mid a \rangle$
  - enriching types to be able to check for exhaustiveness.

Non determinism

- The reduction is non deterministic.
- We may restrict typechecking to disallow overlapping definitions.
- We may change the semantics to give priority to the first match, or to the best match (the most precise matching pattern).

Overloading with typecase

Example

The dynamic semantics is correct,
but type matching could fail!

$$
\begin{array}{l}
\textsf{let rec } \mathit{plus} = \\
\quad \Lambda(\alpha \qquad\qquad) \\
\qquad \textsf{match } \alpha \textsf{ with } \langle \\
\qquad | \ \mathit{int} \Rightarrow (+) \\
\qquad | \ \mathit{float} \Rightarrow (+.) \\
\qquad | \ \exists(\beta \qquad\qquad, \gamma \qquad\qquad)\ \beta \times \gamma \Rightarrow \\
\qquad\quad \lambda(x, y : \beta \times \gamma)\ \lambda(x', y' : \beta \times \gamma)\ \mathit{plus}\ \beta\ x\ x', \mathit{plus}\ \gamma\ y\ y' \\
\qquad \rangle \textsf{ in} \\
\textsf{let } \mathit{twice} = \Lambda(\alpha \qquad\qquad)\ \lambda(x : \alpha)\ \mathit{plus}\ \alpha\ x\ x \textsf{ in} \\
\mathit{twice}\ (\mathit{int} \times \mathit{float})\ (1, 0.5)
\end{array}
$$

Overloading with typecase

Example

The domain may be restricted by a type constraint

$$
\begin{array}{l}
\textsf{let rec } \mathit{plus} = \\
\quad \Lambda(\alpha\langle \mathit{Plus}\ \alpha\rangle) \\
\qquad \textsf{match } \alpha \textsf{ with } \langle \\
\qquad | \ \mathit{int} \Rightarrow (+) \\
\qquad | \ \mathit{float} \Rightarrow (+.) \\
\qquad | \ \exists(\beta\langle \mathit{Plus}\ \beta\rangle,\ \gamma\langle \mathit{Plus}\ \gamma\rangle)\ \beta \times \gamma \Rightarrow \\
\qquad\quad \lambda(x, y : \beta \times \gamma)\ \lambda(x', y' : \beta \times \gamma)\ \mathit{plus}\ \beta\ x\ x', \mathit{plus}\ \gamma\ y\ y' \\
\qquad \rangle \textsf{ in} \\
\textsf{let } \mathit{twice} = \Lambda(\alpha\langle \mathit{Plus}\ \alpha\rangle)\ \lambda(x : \alpha)\ \mathit{plus}\ \alpha\ x\ x \textsf{ in} \\
\mathit{twice}\ (\mathit{int} \times \mathit{float})\ (1, 0.5)
\end{array}
$$

Overloading with typecase

Example

The predicate *Plus* α is defined by induction

$$
\begin{array}{l}
\mathit{Plus\ int};\quad \mathit{Plus\ float};\\
\mathit{Plus}\ \alpha \Rightarrow \mathit{Plus}\ \beta \Rightarrow \mathit{Plus}\ (\alpha \times \beta)\\
\mathsf{let\ rec}\ \mathit{plus} =\\
\quad \Lambda(\alpha\langle \mathit{Plus}\ \alpha\rangle)\\
\qquad \mathsf{match}\ \alpha\ \mathsf{with}\ \langle\\
\qquad \mid \mathit{int} \Rightarrow (+)\\
\qquad \mid \mathit{float} \Rightarrow (+.)\\
\qquad \mid \exists(\beta\langle \mathit{Plus}\ \beta\rangle,\ \gamma\langle \mathit{Plus}\ \gamma\rangle)\ \beta \times \gamma \Rightarrow\\
\qquad\quad \lambda(x, y : \beta \times \gamma)\ \lambda(x', y' : \beta \times \gamma)\ \mathit{plus}\ \beta\ x\ x', \mathit{plus}\ \gamma\ y\ y'\\
\qquad \rangle\ \mathsf{in}\\
\mathsf{let}\ \mathit{twice} = \Lambda(\alpha\langle \mathit{Plus}\ \alpha\rangle)\ \lambda(x : \alpha)\ \mathit{plus}\ \alpha\ x\ x\ \mathsf{in}\\
\mathit{twice}\ (\mathit{int} \times \mathit{float})\ (1, 0.5)
\end{array}
$$

Typecase

Typing rules

Verifying the predicate

Clauses are restricted forms of horn clauses. For instance, given the context Γ equal to

$$\mathit{Plus}\ \mathit{int};\quad \mathit{Plus}\ \mathit{float};\quad \mathit{Plus}\ \alpha \Rightarrow \mathit{Plus}\ \beta \Rightarrow \mathit{Plus}\ (\alpha \times \beta)$$

We can infer:

$$\mathit{Plus}\ (\mathit{int} \times \mathit{float})$$

(and it would be easy to build a witness of the proof) $p_\times\ p_{int}\ p_{float}$

The inference rules can also be read backward for proof search: for example, to prove $\mathit{Plus}\ (\mathit{int} \times \mathit{float})$, a unique rule applies, leaving with the two subgoals *Plus int* and *Plus float* that happens to be axioms.

Dictionary passing — Running Example

In fact, *Plus* $(int \times float)$ proves that *plus* is defined for type *float*. We may thus partially apply *plus* to $int \times float$, and reduce it.
We get (using strong β-reduction on types):

$$plus\ (int \times float) \rightsquigarrow$$
$$\lambda(x, y : int \times float)\ \lambda(x', y' : int \times float)\ plus\ int\ x\ y, plus\ float\ x'\ y' \rightsquigarrow$$
$$\lambda(x, y : int \times float)\ \lambda(x', y' : int \times float)\ (+)\ x\ y, (+.)\ x'\ y'$$

Unfortunately, this reduction duplicates code. Thus, we abstract each definition of *plus* other what only depend on types: If $plus_{\exists(\beta,\gamma)\beta\times\gamma}$ is

$$\Lambda(\beta)\ \Lambda(\gamma)\ \lambda(plus_\beta : \beta \to \beta \to \beta)\ \lambda(plus_\gamma : \gamma \to \gamma \to \gamma)$$
$$\lambda(x, y : \beta \times \gamma)\ \lambda(x', y' : \beta \times \gamma)\ plus_\beta\ x\ y, plus_\gamma\ x'\ y'$$

then the last branch is equal to $plus_{\exists(\beta,\gamma)\beta\times\gamma}\ \beta\ \gamma\ (plus\ \beta)\ (plus\ \gamma)$ and is

$$plus\ (int \times float) \rightsquigarrow plus_{\exists(\beta,\gamma)\beta\times\gamma}\ int\ float\ (plus_{int})\ (plus_{float})$$

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

Dictionary passing — Running example

We recursively define all implementations, abstracting unresolved overloaded symbols away; derived implementations are build when they can be resolved by application of basic implementations.

$$
\begin{array}{l}
\textsf{let rec } \mathit{plus}_{\mathit{int}} = (+) \\
\quad \textsf{and } \mathit{plus}_{\mathit{float}} = (+.) \\
\quad \textsf{and } \mathit{plus}_{\exists\beta\gamma.\beta\times\gamma} = \\
\qquad \Lambda(\beta)\ \Lambda(\gamma) \\
\qquad\quad \lambda(\mathit{plus}_\beta : \beta \to \beta \to \beta)\ \lambda(\mathit{plus}_\gamma : \gamma \to \gamma \to \gamma) \\
\qquad\qquad \lambda(x : \beta)\ \lambda(y : \gamma)\ \mathit{plus}_\beta\ x, \mathit{plus}_\gamma\ y \ \textsf{in} \\
\textsf{let } \mathit{twice} = \\
\qquad \Lambda(\alpha) \\
\qquad\quad \lambda(\mathit{plus}_\alpha : \alpha \to \alpha \to \alpha)\ \lambda(x : \alpha)\ \mathit{plus}_\alpha\ x\ x \ \textsf{in} \\
\textsf{let } \mathit{plus}_{\mathit{int}\times\mathit{float}} = \mathit{plus}_{\exists(\beta,\gamma)\beta\times\gamma}\ \mathit{int}\ \mathit{float}\ \mathit{plus}_{\mathit{int}}\ \ \mathit{plus}_{\mathit{float}}\ \ \textsf{in} \\
\mathit{twice}\ \mathit{plus}_{\mathit{int}\times\mathit{float}}\ \ (1, 0.5)
\end{array}
$$

Dictionary passing — Running example

We recursively define all implementations, abstracting unresolved overloaded symbols away; derived implementations are build when they can be resolved by application of basic implementations.

Definitions are non-recursive

$$
\begin{array}{l}
\text{let} \quad plus_{int} = (+) \text{ in} \\
\text{let} \quad plus_{float} = (+.) \text{ in} \\
\text{let} \quad plus_{\exists\beta\gamma.\beta\times\gamma} = \\
\qquad \Lambda(\beta)\ \Lambda(\gamma) \\
\qquad\quad \lambda(plus_\beta : \beta \to \beta \to \beta)\ \lambda(plus_\gamma : \gamma \to \gamma \to \gamma) \\
\qquad\qquad \lambda(x : \beta)\ \lambda(y : \gamma)\ plus_\beta\ x, plus_\gamma\ y \text{ in} \\
\text{let } twice = \\
\qquad \Lambda(\alpha) \\
\qquad\quad \lambda(plus_\alpha : \alpha \to \alpha \to \alpha)\ \lambda(x : \alpha)\ plus_\alpha\ x\ x \text{ in} \\
\text{let } plus_{int\times float} = plus_{\exists(\beta,\gamma)\beta\times\gamma}\ int\ float\ plus_{int}\ \ plus_{float}\ \text{ in} \\
twice\ plus_{int\times float}\ \ (1, 0.5)
\end{array}
$$

Dictionary passing

Example

After type inference, before translation

$$
\begin{array}{l}
\textsf{def } \mathit{Plus}\ \alpha = \mathit{plus} : \alpha \to \alpha \to \alpha \textsf{ in} \\
\textsf{let rec } \mathit{plus}\colon \mathit{int} \to \mathit{int} \to \mathit{int} = (+) \\
\quad \textsf{and } \mathit{plus}\colon \mathit{float} \to \mathit{float} \to \mathit{float} = (+.) \\
\quad \textsf{and } \mathit{plus}\colon \forall\beta\langle \mathit{Plus}\ \beta\rangle\ \forall\gamma\langle \mathit{Plus}\ \gamma\rangle\ (\beta \times \gamma) \to (\beta \times \gamma) \to (\beta \times \gamma) = \\
\qquad \Lambda(\beta\langle \mathit{Plus}\ \beta\rangle)\ \Lambda(\gamma\langle \mathit{Plus}\ \gamma\rangle) \\
\qquad\quad \lambda(x, y : \beta \times \gamma)\ \lambda(x', y' : \beta \times \gamma)\ \mathit{plus}\ \beta\ x\ x', \mathit{plus}\ \gamma\ y\ y' \textsf{ in} \\
\textsf{let } \mathit{twice} = \\
\qquad \Lambda(\alpha\langle \mathit{Plus}\ \alpha\rangle) \\
\qquad\quad \lambda(x : \alpha)\ \mathit{plus}\ \alpha\ x\ x \textsf{ in} \\
\mathit{twice}\ (\mathit{int} \times \mathit{float})\ (1, 0.5)
\end{array}
$$

Dictionary passing

Example

Alternatively, inlining the constraint

$$\begin{array}{l}
\textsf{let rec } \mathit{plus}\colon \mathit{int} \to \mathit{int} \to \mathit{int} = (+) \\
\quad \textsf{and } \mathit{plus}\colon \mathit{float} \to \mathit{float} \to \mathit{float} = (+.) \\
\quad \textsf{and } \mathit{plus}\colon \forall\beta\langle \mathit{plus} : \beta \to \beta \to \beta\rangle \;\; \forall\gamma\langle \mathit{plus} : \gamma \to \gamma \to \gamma\rangle \\
\qquad\qquad (\beta \times \gamma) \to (\beta \times \gamma) \to (\beta \times \gamma) = \\
\qquad\quad \Lambda(\beta\langle \mathit{plus} : \beta \to \beta \to \beta\rangle)\, \Lambda(\gamma\langle \mathit{plus} : \gamma \to \gamma \to \gamma\rangle) \\
\qquad\qquad \lambda(x, y : \beta \times \gamma)\, \lambda(x', y' : \beta \times \gamma)\, \mathit{plus}\ \beta\ x\ x', \mathit{plus}\ \gamma\ y\ y' \textsf{ in} \\
\textsf{let } \mathit{twice} = \\
\qquad\quad \Lambda(\alpha\langle \mathit{plus} : \alpha \to \alpha \to \alpha\rangle) \\
\qquad\qquad \lambda(x : \alpha)\, \mathit{plus}\ \alpha\ x\ x \textsf{ in} \\
\mathit{twice}\ \mathit{plus}_{(\mathit{int} \times \mathit{float})}\ (1, 0.5)
\end{array}$$

Dictionary passing

Example

Alternatively, inlining the constraint

$$\begin{array}{l}
\textsf{let rec } \mathit{plus}\colon \mathit{int} \to \mathit{int} \to \mathit{int} = (+) \\
\quad \textsf{and } \mathit{plus}\colon \mathit{float} \to \mathit{float} \to \mathit{float} = (+.) \\
\quad \textsf{and } \mathit{plus}\colon \forall\beta\langle \mathit{plus} : \beta \to \beta \to \beta\rangle \;\; \forall\gamma\langle \mathit{plus} : \gamma \to \gamma \to \gamma\rangle \\
\qquad\qquad (\beta \times \gamma) \to (\beta \times \gamma) \to (\beta \times \gamma) = \\
\\
\qquad\qquad \lambda(x, y \qquad) \; \lambda(x', y' \qquad\quad) \; \mathit{plus} \quad x\; x', \mathit{plus} \quad y\; y' \textsf{ in} \\
\textsf{let } \mathit{twice} = \\
\\
\qquad\qquad \lambda(x \quad) \; \mathit{plus} \quad x\; x \textsf{ in} \\
\mathit{twice} \qquad\qquad (1, 0.5)
\end{array}$$

System OML

A restrictive form of overloading

See Odersky et al. (1995)

Characteristics

- System OML is a simple but monolithic system for overloading
  - Its specification is concise.
  - It is not a framework as opposed to other proposals.
- Non overlapping definitions, hence (quasi)-untyped semantics and principal types.
- Single argument resolution.
- Dictionary passing semantics.
- Overloaded definitions of need not have a common type scheme.

System OML — Syntax

We distinguish overloaded symbols u from other variables. Expressions are as usual, but a program p starts with a sequence of toplevel overloaded definitions $\text{def } u : \sigma = v \text{ in } p$

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

Although given in sequence, overloaded definitions

$$\text{def } u_1 : s_1 = v_1 \text{ in } \ldots \text{def } u_n : s_n = v_n \text{ in } a$$

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 abstracted over all unresolved overloading symbols.

Note that overloaded definitions are values.

System OML — Type constraints

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 labels. Intuitively, a constraint for α specifies the types of overloaded symbols that can be applied to a value of type α.

$$\begin{array}{lcll} \tau & ::= & \alpha \mid \tau \to \tau \mid c(\bar{\tau}) & \text{types} \\ \rho_\alpha & ::= & \emptyset \mid u : \alpha \to \tau; \rho_\alpha & \alpha\text{-constraints} \\ \sigma & ::= & \tau \mid \forall \alpha \langle \rho_\alpha \rangle\ \sigma & \text{type schemes} \end{array}$$

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 $\alpha_1 \ldots \alpha_n$. This ensures that

- The result type is fully determined by the first argument.
  (This helps having principle types and a deterministic semantics)
- The definition is parametric in all values of domain in $\exists(\bar{\alpha})c(\bar{\alpha})$.
  This facilitates overloading resolution, coverage checking, and type inference.

System OML — Typing rules

Typing contexts

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

Typing judgments $\Gamma \vdash a : \sigma$ contain the ML typing rules

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

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

System OML — Typing rules

Overloaded definitions

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

We write $\Gamma \vdash u \# \sigma_\pi$ to mean that σ' and σ_π do not have the same toplevel type constructor for all $u : \sigma' \in \Gamma$, which implies in particular that they are not overlapping.

System OML — Typing rules

Introduction and elimination of polymorphism

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

As in ML, we restrict generalization to value expressions.

Overloaded symbols

$$\textsc{Var-Over}\ \frac{u : \sigma \in \Gamma}{\Gamma \vdash u : \sigma}$$

This rules allows to retrieve overloaded symbols that are either provided by overloaded definitions, or assumptions introduced by constrained polymorphism All-Intro (and discharged at some All-Elim).

Typing

Example

See Translation to ML which contains withing the translation an example of typing.

System OML — Compilation to ML

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

We compile a program p into an ML expression $\mathcal{M}$ (a particular case expressions of the source language) 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

$$\frac{\Gamma \vdash a : \sigma \triangleright \mathcal{M} \qquad \Gamma, x : \sigma \vdash a' : \tau \triangleright \mathcal{M}'}{\Gamma \vdash \mathsf{let}\ x = a\ \mathsf{in}\ a' : \tau \triangleright \mathsf{let}\ x = \mathcal{M}\ \mathsf{in}\ \mathcal{M}'}$$

ARROW-INTRO

$$\frac{x \notin \Gamma \qquad \Gamma, x : \tau \vdash a : \tau' \triangleright \mathcal{M}}{\Gamma \vdash \lambda(x)\ a : \tau \to \tau' \triangleright \lambda(x)\ \mathcal{M}}$$

ARROW-ELIM

$$\frac{\Gamma \vdash a_1 : \tau_2 \to \tau_1 \triangleright \mathcal{M}_1 \qquad \Gamma \vdash a_2 : \tau_2 \triangleright \mathcal{M}_2}{\Gamma \vdash a_1\ a_2 : \tau_1 \triangleright \mathcal{M}_1\ \mathcal{M}_2}$$

System OML — Compilation to ML

Introducing and using overloaded definitions

DEF

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

VAR-OVER

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

Introducing and using polymorphism

ALL-INTRO

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

ALL-ELIM

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

Compilation of OML — Example

The previous example, twice

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

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

Compilation of OML — Example (Cont.)

Let Γ_0 stand for

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

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

$$\dfrac{\Gamma_1 \vdash \text{plus: } \forall\beta\langle \mathit{plus} : \beta^3\rangle\ \forall\gamma\langle \text{plus: } \gamma^3\rangle\ (\beta \times \gamma)^3 \triangleright \mathrm{x}_\sigma^{\text{plus}} \quad \Gamma_1 \vdash \text{plus: } \mathit{int}^3 \triangleright \mathrm{x}_{\mathit{int}^3}^{\text{plus}} \quad \Gamma_1 \vdash \text{plus: } \mathit{float}^3 \triangleright \mathrm{x}_{\mathit{float}^3}^{\text{plus}}}{\Gamma_1 \vdash \text{plus: } (\mathit{int} \times \mathit{float})^3 \triangleright \mathrm{x}_\sigma^{\text{plus}}\ \mathrm{x}_{\mathit{int}^3}^{\text{plus}}\ \mathrm{x}_{\mathit{float}^3}^{\text{plus}}}\ \textsc{All-Elim}$$

Therefore

$$\dfrac{\Gamma_1 \vdash \mathit{twice} : (\mathit{int} \times \mathit{float})^2 \triangleright \mathit{twice}\ (\mathrm{x}_\sigma^{\text{plus}}\ \mathrm{x}_{\mathit{int}^3}^{\text{plus}}\ \mathrm{x}_{\mathit{float}^3}^{\text{plus}})}{\Gamma_1 \vdash \mathit{twice}\ (1, 0.5) : (\mathit{int} \times \mathit{float}) \triangleright \mathit{twice}\ (\mathrm{x}_\sigma^{\text{plus}}\ \mathrm{x}_{\mathit{int}^3}^{\text{plus}}\ \mathrm{x}_{\mathit{float}^3}^{\text{plus}})\ (1, 0.5)}\ \textsc{All-Elim}$$

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 be proved for different versions of the language. So far, it is only a conjecture for OML.

System OML — Type inference

Principal types

There are principal types in ML, 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 with ML is to treat applications of overloaded functions by generating a fresh type variable with a type constraint that is stored in the context as overloaded assumptions.

The non-overlapping of typing assumptions on overloaded variables implies that the assumptions may have to transformed when a variable is instantiated during unification: two assumptions may be merged triggering further unifications, or may have to be resolved, which removes them from the context, but perhaps introduces other assumptions in the context.

Qualified types

See Jones (1992)

A general framework

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

The idea is to introduce predicates that restrict the set of types a variable may range over. For instance, *Plus* α, which we have used above means that α can only be instantiated at the type τ for such that there is a definition for *plus* of type $\tau \longrightarrow \tau \longrightarrow \tau$.

The framework allows to gather predicate type constraints (much as system OML does for overloaded definitions) but to solve them indepedently. It can also be parametrized by the way constraints are solved. This introduces more flexibility and allows to internalize simplification and optimization of constraints. This is an important difference between qualified types and system OML.

Types classes

Overloading definitions are more structured

- They 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 can be used by default when defining instances. Default definitions 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.

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.
- Derived instances are defined as functors.
- Such modules can be declared as overloading their definitions.
- The basic overloaded mechanism is then 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 the basically 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.

Local overloading

There is a potential conflict with escaping resolution: An overloaded symbol with a local implementation can either be resolved immediately or left generic to be resolved later, perhaps with another implementation.

Some explicit information may be required when introduction new overloaded scopes for disambiguation.

Remaining problems and challenges

Design space

Because some restrictions must be imposed, there are many variations in the design space.

See Jones et al. (1997)

Ideas to bring back home

Overloading is quite useful

- Just static overloading may significantly enlighten the notations
- Static overloading is too strong a restriction and often frustating.
- Dynamic overloading enables polytypic programmaing

Overloading is well-understood

- Long, very possitive experience in Haskell.
- Perhaps, more restrictive forms would be acceptable.

Require some compromises

- On the overlapping definitions: the semantics depends on typechecking; typechecking may also become hard.
- 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.