Generalised recursion, degrees and a mixin language

Seminar in Aarhus – May 17th, 2004

Pascal Zimmer

INRIA Sophia Antipolis
General motivation

To design a base language with:

- functional core
- objects
- well-defined semantics, that can be realistically implemented
- ML-like inference of principal types

in the goal of adding other paradigms (migration, reactive)…
Outline

- semantics of object languages
- a language with recursive records and generalised recursion
- a type system with degrees
- implementation, abstract machine
- mixins
Semantics of objects 1

Auto-application semantics

- object = collection of pre-methods:

\[ o = [\ldots, l = \zeta(self) \ b, \ldots] \]

- method call:

\[ o.l \Rightarrow b \{self \leftarrow o\} \]

- specific typing
- inference of principal types impossible
Recursive record semantics

- class:

\[ C = \lambda x_1 \ldots \lambda x_n \lambda \text{self} \{ l_1 = M_1, \ldots, l_p = M_p \} \]

- object: \( o = \text{fix} (C N_1 \ldots N_n) \)
- row variables to extend the object
- no modification of the state, since \text{self} is bound to the initial object
- typing model of OCAMAL
Language proposition

- Wand’s recursive record semantics
- ML-like references to hold the state of the object
- examples:

\[
\begin{align*}
\text{point} & = \lambda x \lambda \text{self} \\
& \quad \{ \text{pos} = \text{ref } x, \\
& \quad \quad \text{move} = \lambda y (\text{self.pos} := !\text{self.pos} + y) \} \\
\quad \text{p} & = \text{fix} (\text{point } 4) \\
\text{color_point} & = \lambda x \lambda c \lambda \text{self} \\
& \quad \{ \text{point } x \text{ self, color} = \text{ref } c \}
\end{align*}
\]
Evaluating the fixpoint

- Problem: how can we evaluate the fixpoint?

\[ \text{fix} = \lambda f \ (\text{let rec } x = fx \text{ in } x) \]

- In SML, only allowed construct:

\[ \text{let rec } x = \lambda yN \text{ in } M \]

- We need a generalised recursion operator
- But some recursions are dangerous:

\[ \text{let rec } x = xV \text{ in } M \]

\[ \text{let rec } x = x + 1 \text{ in } M \]
Type system with degrees

- Boudol, 2001
- degree = boolean information in function types and in typing contexts

\[ \theta^d \rightarrow \tau \]

- 0 = “dangerous”, 1 = “sure”
- intuitively: is the value required or not when evaluating
- \( \text{(let rec } x = N \text{ in } M) \) is typable iff \( N \) is typable with a degree 1 for \( x \)
- \( \text{(let rec } x = fx \text{ in } M) \) is typable iff \( f \) has type \( \theta^1 \rightarrow \tau \) (“protective” function)
Degrees - examples

- example of protective function:

\[\text{point0} = \lambda\text{self}\]
\[\{\text{pos} = \text{ref 0},\]
\[\text{move} = \lambda y(\text{self.pos} := !\text{self.pos} + y)\}\]

- \[\text{fix} = \lambda f (\text{let rec} x = f x \text{ in } x)\]
  has type: \((\tau^1 \rightarrow \tau)^0 \rightarrow \tau\)

- \[\lambda\text{self}\{x = 0, y = \text{self.x}\}\]
  has type: \(\{\rho, x : \tau\}^0 \rightarrow \{x : \text{int}, y : \tau\}\)
  where \(\rho\) is a row variable
  with the constraint \(\rho :: \{x\}\)
Degrees - results

- subject reduction
- safety: the evaluation of a typable term never leads to an error (recursion, field access, applications...)
- algorithm for inferring principal types, extension of ML’s one
Unification and inference algorithms

- more “realistic” and efficient versions
- working on graphs (recursive types)
- unification of degrees, records, types
- polymorphism similar to ML, on degree, row or type variables; generalising for:

\[
\text{let (rec) } x = V \text{ in } M
\]

- constraints on row variables \((\rho :: L)\) and degree variables;
  example: \(\lambda f \lambda x (fx)\) has type
  \((\theta^\alpha \rightarrow \tau)^\beta \rightarrow \theta^\gamma \rightarrow \tau\) with \(\gamma \leq \alpha\)
Abstract machine

- we need to evaluate terms with the shape

\((\lambda \text{self} M) \circ\)

where \(\circ\) is a still unevaluated variable, knowing that the value of \(\text{self}\) is not needed to evaluate \(M\)

- usual machines for \(\lambda\)-calculus or ML do not allow the evaluation of generalised recursion
Abstract machine

\[ \mathcal{M} = (S, \sigma, M, \xi) \]

- \(S\): control stack
- \(\sigma\): environment
- \(M\): term to evaluate
- \(\xi\): memory for recursive values (and references)

- set of 11 transition rules, among which a “magic” rule:

\[
(S :: (\sigma \lambda y M []), \rho :: \{x \mapsto \ell\}, x, \xi) \\
\rightarrow (S, \sigma :: \{y \mapsto \ell\}, M, \xi) \quad \text{if } \xi(\ell) = \bullet
\]
Abstract machine

• operational correspondence
• determinism
• no infinite “silent” reductions
• correction:
  if the starting term is typable, then both the machine and the calculus semantics go through the same reductions
MLObj

http://www-sop.inria.fr/mimosa/Pascal.Zimmer/mlobj.html

OCAML-like interpreter...
Mixins

- **goal:** use higher-order constructs to build more powerful objects
- **generator:** $\lambda s \{ \ldots \}$
- **mixin:** generator modifier

\[ C = \lambda x_1 \ldots \lambda x_n \lambda g \lambda s \{ \ldots \text{fields} \ldots \text{methods} \ldots \} \]

- **instance** ($\lambda s \{ \}$ is the initial generator):

\[ \text{fix} \ (C N_1 \ldots N_n (\lambda s \{\})) \]

- **new operator:**

\[ \text{new} = \lambda m \text{fix} \ (m (\lambda s \{\})) \]
Mixins - definition

Implemented by syntactic sugar rules.

mixin

\begin{align*}
\text{var} \ l &= N & \text{non-constant data} \\
\text{cst} \ l &= N & \text{constant data} \\
\text{meth} \ l(\text{super, self}) &= N & \text{method} \\
\text{meth} \ l(\text{super, self}) &\leftarrow N & \text{method override} \\
\text{inherit} \ N & & \text{inheritance} \\
\text{without} \ l & & \text{field suppression} \\
\text{rename} \ l \ as \ l' & & \text{field renaming} \\
\end{align*}

end

Method call: \( M \# l \)
Mixins - examples

\[ \text{point} = \lambda x \]
\[ \text{mixin} \]
\[ \text{var} \ pos = x \]
\[ \text{meth} \ move \ldots \]
\[ \text{end} \]

\[ \text{coloring} = \lambda c \]
\[ \text{mixin} \]
\[ \text{var} \ color = c \]
\[ \text{meth} \ paint \ldots \]
\[ \text{end} \]

\[ \text{colorPoint} = \lambda x \lambda c \]
\[ \text{mixin} \]
\[ \text{inherit} \ point \ x \]
\[ \text{inherit} \ coloring \ c \]
\[ \text{end} \]

\Rightarrow \text{multiple inheritance}
Mixins - examples

\[
\text{reset} = \\
\text{mixin} \\
\text{meth \ reset}(\text{super}, \text{self}) = \text{self}.\text{pos} := 0 \\
\text{end}
\]

\[
\text{resetPoint} = \lambda x \text{ \mixin} \\
\text{inherit \ point \ x} \\
\text{inherit \ reset} \\
\text{end} \\
\Rightarrow \text{code sharing}
\]

\[
\text{resetColorPoint} = \lambda x \lambda c \text{ \mixin} \\
\text{inherit \ colorPoint \ x \ c} \\
\text{inherit \ reset} \\
\text{end}
\]
Mixins - examples

mixin
meth reset(super, self) ←
    λd (super#reset; super#paint d)
end

• Typing determines which mixins can be instantiated and which cannot.
• By changing the initial generator, one can get initialisers.
• Mixins = first order values
  ⇒ a huge expressive power
  still to be explored!
And after?

- advanced functionalities: cloning, binary methods...:

\[
\text{meth } eq(\text{super, self}) = \lambda p (\text{self.pos} == p.pos)
\]

- operationally, no problem
- typing: not enough polymorphism!
- System F? type inference undecidable...
- intersection types? finite-rank inference is decidable...
  \[\Rightarrow\] 2nd part of PhD thesis: new inference algorithm for intersection types
Future

- integrate intersection types in the language MLOBJ
- polymorphic methods in MLOBJ
- study the expressivity of mixins more closely
- extend the language with other paradigms
The end