Generalised Recursion in ML and Mixins

Seminar in Aalborg – November 24th, 2004

Pascal Zimmer

pzimmer@daimi.au.dk

BRICS

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

- model initiated by Kamin, 1988;
  reference: Abadi and Cardelli, 1996
- object = collection of pre-methods:

$$o = [\ldots, l = \zeta(\mathrm{self})\ b, \ldots]$$

- method call:

$$o.l \ \Rightarrow \ b\ \{\mathrm{self} \leftarrow o\}$$

- specific typing
- inference of principal types impossible

Semantics of objects 2

Recursive record semantics

- Cardelli 1988, Wand 1994, Cook 1994
- class:

$$C = \lambda x_1 \dots \lambda x_n\ \lambda\mathrm{self}\ \{l_1 = M_1, \dots, l_p = M_p\}$$

- object: $o = \mathrm{fix}\ (C N_1 \dots N_n)$
- row variables to extend the object
- no modification of the state, since self is bound to the initial object
- typing model of OCAML

Language proposition

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

$$
\begin{aligned}
\mathit{point} = \lambda x \lambda \mathrm{self}\ & \\
& \{\mathit{pos} = \mathrm{ref}\ x, \\
& \ \ \mathit{move} = \lambda y(\mathrm{self}.\mathit{pos} := !\mathrm{self}.\mathit{pos} + y)\}
\end{aligned}
$$

$$p = \mathrm{fix}\ (\mathit{point}\ 4)$$

$$
\begin{aligned}
\mathit{color\_point} = \lambda x \lambda c \lambda \mathrm{self}\ & \\
& \{\mathit{point}\ x\ \mathrm{self}, \mathit{color} = \mathrm{ref}\ c\}
\end{aligned}
$$

Evaluating the fixpoint

- Problem: how can we evaluate the fixpoint ?

$$\text{fix} = \lambda f\ (\text{let rec } x = f\,x \text{ in } x)$$

- In SML, only allowed construct:

$$\text{let rec } x = \lambda y N \text{ in } M$$

- 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

$$\theta^d \longrightarrow \tau$$

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

Degrees - examples

- example of protective function:

$$
\begin{aligned}
point0 = \lambda\text{self}\ & \\
& \{pos = \text{ref } 0, \\
& \quad move = \lambda y(\text{self}.pos := !\text{self}.pos + y)\}
\end{aligned}
$$

- $\text{fix} = \lambda f(\text{let rec } x = f x \text{ in } x)$
  has type: $(\tau^{1} \rightarrow \tau)^{0} \rightarrow \tau$
- $\lambda\text{self}\{x = 0, y = \text{self}.x\}$
  has type: $\{\rho, x : \tau\}^{0} \rightarrow \{x : int, y : \tau\}$
  where ρ 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 infering 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:

$$\mathrm{let}\ (\mathrm{rec})\ x = V\ \mathrm{in}\ M$$

- constraints on row variables ($\rho :: L$) and degree variables;
  example: $\lambda f \lambda x (f x)$ has type
  $(\theta^\alpha \to \tau)^\beta \to \theta^\gamma \to \tau$ with $\gamma \le \alpha$

Abstract machine

- we need to evaluate terms with the shape

$$(\lambda \mathrm{self}\, M)\ o$$

  where o is a still unevaluated variable, knowing that the value of self is not needed to evaluate M
- usual machines for λ-calculus or ML do not allow the evaluation of generalised recursion

Abstract machine

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

- S: control stack
- σ: environment
- M: term to evaluate
- ξ: memory for recursive values (and references)

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

$$\begin{array}{rll} & (S :: (\sigma \lambda y M[]), \rho :: \{x \mapsto \ell\}, x, \xi) & \\ \longrightarrow & (S, \sigma :: \{y \mapsto \ell\}, M, \xi) & \text{if } \xi(\ell) = \bullet \end{array}$$

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

Method call: $M\#l$

Mixins - examples

$point = \lambda x$
mixin
var $pos = x$
meth $move \ldots$
end

$coloring = \lambda c$
mixin
var $color = c$
meth $paint \ldots$
end

$colorPoint = \lambda x \lambda c$
mixin
inherit $point\ x$
inherit $coloring\ c$
end

$\Rightarrow$ multiple inheritance

Mixins - examples

$reset =$
mixin
meth $reset(\mathrm{super}, \mathrm{self}) = \mathrm{self}.pos := 0$
end

$resetPoint = \lambda x$
mixin
inherit $point\ x$
inherit $reset$
end

$resetColorPoint = \lambda x \lambda c$
mixin
inherit $colorPoint\ x\ c$
inherit $reset$
end

$\Rightarrow$ code sharing

Mixins - examples

mixin

$$\textbf{meth}\ \mathit{reset}(\text{super}, \text{self}) \leftarrow$$
$$\lambda d\ (\text{super}\#\mathit{reset};\ \text{super}\#\mathit{paint}\ d)$$

end

- Typing determines which mixins can be instantiated and which cannot.
- By changing the initial generator, one can get initialisers.
- Mixins = first order values
  $\Rightarrow$ a huge expressive power
  still to be explored !

And after ?

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

$$\textbf{meth}\ eq(\mathrm{super}, \mathrm{self}) = \lambda p\ (\mathrm{self}.pos == p.pos)$$

- 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