A Semantics for Lazy Types

Bachelor Thesis

Georg Neis
Advisor: Andreas Rossberg
Programming Systems Lab
Saarland University

June 8, 2006

Motivation

- Open and distributed programming: dynamic loading and linking
  - Type checking at runtime
  - Laziness

Motivation

- Open and distributed programming: dynamic loading and linking
  - Type checking at runtime
  - Laziness
- Dynamic type checking: comparison of type expressions
- Reduction to normal-form
- Due to laziness: may interleave with term-level reduction!

Motivation

- Open and distributed programming: dynamic loading and linking
  - Type checking at runtime
  - Laziness
- Dynamic type checking: comparison of type expressions
- Reduction to normal-form
- Due to laziness: may interleave with term-level reduction!
- Task: develop calculus (based on $F\omega$) that models this

An example from Alice ML

```
(* A *)
type t = int
val x : t = 5

(* B *)
import type t; val x : t from "A"
type u = t
val y : u*u = (x,x)

(* C *)
import type u = int; val y : u*u from "B"
type v = int
val z = #1 y
```

Lazy evaluation in Alice ML

- Provided by **lazy futures**
- Example: (fn f $\Rightarrow$ f 1) (lazy (fn x $\Rightarrow$ x) (fn n $\Rightarrow$ n+41))

Modeling lazy evaluation in the untyped lambda calculus

- $e ::= x \mid \lambda x.e \mid e_1\ e_2 \mid \text{let } x = e_1 \text{ in } e_2 \mid \text{lazy } x = e_1 \text{ in } e_2$

Modeling lazy evaluation in the untyped lambda calculus

- $e ::= x \mid \lambda x.e \mid e_1\ e_2 \mid \text{let } x = e_1 \text{ in } e_2 \mid \text{lazy } x = e_1 \text{ in } e_2$
- Translation of the previous example:
  (fn f ⇒ f 1) (lazy (fn x ⇒ x) (fn n ⇒ n+41))
  $\leadsto (\lambda f.f\ 1)\ (\text{lazy } y = (\lambda x.x)\ (\lambda n.n + 41) \text{ in } y)$

Modeling lazy evaluation in the untyped lambda calculus

Reduction:

- $(\lambda f.f\ 1)$ (lazy y = $(\lambda x.x)\ (\lambda n.n + 41)$ in y) $\longrightarrow$

Modeling lazy evaluation in the untyped lambda calculus

Reduction:

- $(\lambda f.f\ 1)$ (lazy y = $(\lambda x.x)\ (\lambda n.n + 41)$ in y) $\longrightarrow$
- lazy y = $(\lambda x.x)\ (\lambda n.n + 41)$ in $(\lambda f.f\ 1)\ y \longrightarrow$

Modeling lazy evaluation in the untyped lambda calculus

Reduction:

- $(\lambda f.f\ 1)$ (lazy y = $(\lambda x.x)\ (\lambda n.n + 41)$ in y) $\longrightarrow$
- lazy y = $(\lambda x.x)\ (\lambda n.n + 41)$ in $(\lambda f.f\ 1)\ y \longrightarrow$
- lazy y = $(\lambda x.x)\ (\lambda n.n + 41)$ in $y\ 1 \longrightarrow$

Modeling lazy evaluation in the untyped lambda calculus

Reduction:

- $(\lambda f.f\ 1)\ (\text{lazy } y = (\lambda x.x)\ (\lambda n.n + 41) \text{ in } y) \longrightarrow$
- $\text{lazy } y = (\lambda x.x)\ (\lambda n.n + 41) \text{ in } (\lambda f.f\ 1)\ y \longrightarrow$
- $\text{lazy } y = (\lambda x.x)\ (\lambda n.n + 41) \text{ in } y\ 1 \longrightarrow$
- $\text{let } y = (\lambda x.x)\ (\lambda n.n + 41) \text{ in } y\ 1 \longrightarrow$

Modeling lazy evaluation in the untyped lambda calculus

Reduction:

- $(\lambda f.f\ 1)$ (lazy y = $(\lambda x.x)$ $(\lambda n.n + 41)$ in y) $\longrightarrow$
- lazy y = $(\lambda x.x)$ $(\lambda n.n + 41)$ in $(\lambda f.f\ 1)$ y $\longrightarrow$
- lazy y = $(\lambda x.x)$ $(\lambda n.n + 41)$ in y 1 $\longrightarrow$
- let y = $(\lambda x.x)$ $(\lambda n.n + 41)$ in y 1 $\longrightarrow$
- let y = $(\lambda n.n + 41)$ in y 1 $\longrightarrow$

Modeling lazy evaluation in the untyped lambda calculus

Reduction:

- $(\lambda f.f\ 1)$ (lazy y = $(\lambda x.x)$ $(\lambda n.n + 41)$ in y) $\longrightarrow$
- lazy y = $(\lambda x.x)$ $(\lambda n.n + 41)$ in $(\lambda f.f\ 1)$ y $\longrightarrow$
- lazy y = $(\lambda x.x)$ $(\lambda n.n + 41)$ in y 1 $\longrightarrow$
- let y = $(\lambda x.x)$ $(\lambda n.n + 41)$ in y 1 $\longrightarrow$
- let y = $(\lambda n.n + 41)$ in y 1 $\longrightarrow$
- $(\lambda n.n + 41)$ 1 $\longrightarrow$

Modeling lazy evaluation in the untyped lambda calculus

Reduction:

- $(\lambda f.f\ 1)$ (lazy y = $(\lambda x.x)$ $(\lambda n.n + 41)$ in y) $\longrightarrow$
- lazy y = $(\lambda x.x)$ $(\lambda n.n + 41)$ in $(\lambda f.f\ 1)$ y $\longrightarrow$
- lazy y = $(\lambda x.x)$ $(\lambda n.n + 41)$ in y 1 $\longrightarrow$
- let y = $(\lambda x.x)$ $(\lambda n.n + 41)$ in y 1 $\longrightarrow$
- let y = $(\lambda n.n + 41)$ in y 1 $\longrightarrow$
- $(\lambda n.n + 41)$ 1 $\longrightarrow$
- $1 + 41 \longrightarrow 42$

Modeling lazy evaluation in the untyped lambda calculus

Contexts:

- $L ::= \_ \mid \text{lazy } x = e \text{ in } L$
- $E ::= \_ \mid E\ e \mid v\ E \mid \ldots$

Modeling lazy evaluation in the untyped lambda calculus

Contexts:

- $L ::= \_ \mid \text{lazy } x = e \text{ in } L$
- $E ::= \_ \mid E\ e \mid v\ E \mid \ldots$

- Suspend:
  $LE[\textbf{lazy x = e}_1 \textbf{ in } e_2] \longrightarrow L[\textbf{lazy x = e}_1 \textbf{ in } E[e_2]]$

Modeling lazy evaluation in the untyped lambda calculus

Contexts:

- $L ::= \_ \mid \text{lazy } x = e \text{ in } L$
- $E ::= \_ \mid E\ e \mid v\ E \mid \ldots$

- Suspend:
  $LE[\textbf{lazy x = e}_\mathbf{1}\textbf{ in } e_2] \longrightarrow L[\textbf{lazy x = e}_\mathbf{1}\textbf{ in } E[e_2]]$
- Trigger1:
  $L_1[\textbf{lazy } x = e_1 \text{ in } L_2E[x\ v]] \longrightarrow L_1[\textbf{let } x = e_1 \text{ in } L_2E[x\ v]]$

Existential types

- To model a simple form of modules
- Example:

```
structure S = struct
  type t = bool
  val n = 42
end : sig
  type t
  val n : int
end
```

$\rightsquigarrow \langle \mathsf{bool}, 42 \rangle : \exists \alpha . \mathsf{int}$

Existential types

- Accessing a module: let $\langle \alpha, x \rangle$ = e_1 in e_2

Existential types

- Accessing a module: let $\langle\alpha, x\rangle$ = e_1 in e_2
- Reduction:
  $LE[\text{let } \langle\alpha, x\rangle = \langle\tau, v\rangle{:}\tau' \text{ in } e] \longrightarrow LE[e[\alpha := \tau][x := v]]$

Existential types

- Accessing a module: let $\langle \alpha, x \rangle$ = e_1 in e_2
- Reduction:
  $LE[\text{let } \langle \alpha, x \rangle = \langle \tau, v \rangle{:}\tau' \text{ in } e] \longrightarrow LE[e[\alpha := \tau][x := v]]$
- Lazy variant for loading a module: lazy $\langle \alpha, x \rangle$ = e_1 in e_2

Typecase

- Comparison of two types τ_1 and τ_2
- tcase $e{:}\tau_1$ of $x{:}\tau_2$ then e_1 else e_2

Typecase

- Comparison of two types τ_1 and τ_2
- tcase e:τ_1 of x:τ_2 then e_1 else e_2
- Reduction:
  - $LE[\text{tcase } v{:}\nu \text{ of } x{:}\nu \text{ then } e_1 \text{ else } e_2] \longrightarrow LE[e_1[x := e]]$
  - $LE[\text{tcase } v{:}\nu \text{ of } x{:}\nu' \text{ then } e_1 \text{ else } e_2] \longrightarrow LE[e_2] \qquad (\nu \neq \nu')$

Lazy types

- Consider: lazy $\langle \alpha, x \rangle$ = *e* in tcase *x*:α of *y*:int then *y* else 0
- $\alpha \stackrel{?}{=}$ int

Lazy types

- Consider: lazy $\langle \alpha, x \rangle$ = e in tcase x:α of y:int then y else 0
- $\alpha \stackrel{?}{=}$ int
- lazy $\langle \alpha, x \rangle$ = e in tcase x:α of y:int then y else 0 $\longrightarrow$
  let $\langle \alpha, x \rangle$ = e in tcase x:α of y:int then y else 0

Lazy types

- Consider: lazy $\langle\alpha, x\rangle$ = e in tcase x:α of y:int then y else 0
- $\alpha \stackrel{?}{=}$ int
- lazy $\langle\alpha, x\rangle$ = e in tcase x:α of y:int then y else 0 $\longrightarrow$
  let $\langle\alpha, x\rangle$ = e in tcase x:α of y:int then y else 0
- $\textsc{Trigger2}$:
  L_1[lazy $\langle\alpha, x\rangle$ = e in $L_2 ET[\alpha]$] $\longrightarrow$
  L_1[let $\langle\alpha, x\rangle$ = e in $L_2 ET[\alpha]$]

An example from Alice ML

```
(* A *)
type t = int
val x : t = 5

(* B *)
import type t; val x : t from "A"
type u = t
val y : u*u = (x,x)

(* C *)
import type u = int; val y : u*u from "B"
type v = int
val z = #1 y
```

Translation into the calculus

```
(* A *)
type t = int
val x : t = 5
```

$\rightsquigarrow$

$$a \quad = \quad \lambda\_{:}1.\ \langle \mathsf{int}, 5\rangle{:}\exists\alpha.\alpha$$

Translation into the calculus

```
(* B *)
import type t; val x : t from "A"
type u = t
val y : u*u = (x,x)
```

$\rightsquigarrow$

$$\begin{array}{rcl} b & = & \lambda\_{:}1.\ \mathsf{lazy}\ \langle t, x\rangle = a\ () \\ & & \qquad\quad \mathsf{in}\ \langle t, \langle x, x\rangle\rangle{:}\exists\alpha.\alpha\times\alpha \end{array}$$

Translation into the calculus

```
(* C *)
import type u = int; val y : u*u from "B"
type v = int
val z = #1 y
```

$\rightsquigarrow$

$$
\begin{array}{rcl}
c & = & \lambda\_{:}1.\ \mathsf{lazy}\ \langle \_, y\rangle = \\
& & \qquad \mathsf{let}\ \langle u, y\rangle = b\ () \\
& & \qquad \mathsf{in\ tcase}\ y : u \times u\ \mathsf{of}\ y' : \mathsf{int} \times \mathsf{int} \\
& & \qquad\qquad \mathsf{then}\ \langle \mathsf{int}, y'\rangle{:}\exists\alpha.\mathsf{int} \times \mathsf{int} \\
& & \qquad\qquad \mathsf{else}\ \bot \\
& & \quad \mathsf{in}\ \langle \mathsf{int}, \mathsf{fst}\ y\rangle{:}\exists\alpha.\mathsf{int}
\end{array}
$$

Design Space

- Three ways of type-level reduction – different degrees of laziness
- Consider: $\mathsf{int} \times ((\lambda\beta.\nu)\ \alpha) \stackrel{?}{=} \mathsf{bool} \times \mathsf{int}$ (where α lazy)

Design Space

- Three ways of type-level reduction – different degrees of laziness
- Consider: $\mathsf{int} \times ((\lambda\beta.\nu)\ \alpha) \stackrel{?}{=} \mathsf{bool} \times \mathsf{int}$ (where α lazy)
  1. Naive: use the trigger rule

Design Space

- Three ways of type-level reduction – different degrees of laziness
- Consider: $\mathsf{int} \times ((\lambda\beta.\nu)\ \alpha) \stackrel{?}{=} \mathsf{bool} \times \mathsf{int}$ (where α lazy)
  1. Naive: use the trigger rule
  2. Clever: perform the application

Design Space

- Three ways of type-level reduction – different degrees of laziness
- Consider: $\mathsf{int} \times ((\lambda\beta.\nu)\ \alpha) \stackrel{?}{=} \mathsf{bool} \times \mathsf{int}$ (where α lazy)
  1. Naive: use the trigger rule
  2. Clever: perform the application
  3. Smart: weak-head reduction

References

- Martín Abadi, Luca Cardelli, Benjamin Pierce, and Didier Rémy. Dynamic typing in polymorphic languages. Journal of Functional Programming, January 1995.
- John Maraist, Martin Odersky, and Philip Wadler. A call-by-need lambda calculus. Journal of Functional Programming, May 1998.
- Benjamin Pierce. Types and Programming Languages. The MIT Press, February 2002.
- Zena Ariola and Matthias Felleisen. The call-by-need lambda calculus. Journal of Functional Programming, 1997.
- Matthias Berg. Polymorphic lambda calculus with dynamic types, FoPra Thesis, October 2004. http://www.ps.uni-sb.de/∼berg/fopra.html
- Andreas Rossberg, Didier Le Botlan, Guido Tack, Thorsten Brunklaus, and Gert Smolka. Alice through the looking glass. Trends in Functional Programming, volume 5, Intellect, 2005