A Semantics for Lazy Types

Bachelor's Thesis

Georg Neis
Advisor: Andreas Rossberg
Programming Systems Lab
Saarland University

October 11, 2006

topic of the thesis

- goal: model the combination of lazy linking and dynamic type checking
- example: Alice ML
  - components: dynamic modules
  - `import` *spec* `from` *url*
  - dynamic type checking at link-time
  - lazy futures
  - import statement $\leadsto$
    `lazy unpack (acquire` *url* `) : sig` *spec* `end`

our model

- based on F_ω (with pairs and existential types)

$$\begin{array}{lrl}
\text{terms} & e ::= & x \mid \lambda x{:}\tau.e \mid e_1\ e_2 \mid \lambda\alpha{:}\kappa.e \mid e\ \tau \mid \\
& & \langle e_1, e_2\rangle \mid \text{let } \langle x_1, x_2\rangle = e_1 \text{ in } e_2 \mid \\
& & \langle \tau_1, e\rangle{:}\tau_2 \mid \text{let } \langle \alpha, x\rangle = e_1 \text{ in } e_2
\end{array}$$

$$\begin{array}{lrl}
\text{types} & \tau ::= & \alpha \mid \tau_1 \rightarrow \tau_2 \mid \tau_1 \times \tau_2 \mid \\
& & \forall\alpha{:}\kappa.\tau \mid \exists\alpha{:}\kappa.\tau \mid \lambda\alpha{:}\kappa.\tau \mid \\
& & \tau_1\ \tau_2 \mid \langle \tau_1, \tau_2\rangle \mid \tau.1 \mid \tau.2
\end{array}$$

$$\begin{array}{lrl}
\text{kinds} & \kappa ::= & \Omega \mid \kappa_1 \rightarrow \kappa_2 \mid \kappa_1 \times \kappa_2
\end{array}$$

our model

intensional type analysis

- typecase for comparing type expressions
- $e ::= \cdots \mid \text{tcase } e_0{:}\tau \text{ of } x{:}\tau' \text{ then } e_1 \text{ else } e_2$
- operational semantics:
$$\begin{array}{lr} E[\text{tcase } v{:}\tau \text{ of } x{:}\tau' \text{ then } e_1 \text{ else } e_2] \longrightarrow E[e_1[x := v]] & (\tau \text{ equals } \tau') \\ E[\text{tcase } v{:}\tau \text{ of } x{:}\tau' \text{ then } e_1 \text{ else } e_2] \longrightarrow E[e_2] & (\text{otherwise}) \end{array}$$
- equivalence checking explicit

our model

lazy evaluation

- $e ::= \cdots \mid \text{lazy } \langle \zeta, x \rangle = e_1 \text{ in } e_2$
- lazy variant of opening existential packages
- we distinguish between regular (α) and lazy (ζ) type variables
- operational semantics:
  - $L ::= \_ \mid \text{lazy } \langle \zeta, x \rangle = e \text{ in } L$
  - $LE[\text{lazy } \langle \zeta, x \rangle = e_1 \text{ in } e_2] \longrightarrow L[\text{lazy } \langle \zeta, x \rangle = e_1 \text{ in } E[e_2]]$

lazy terms

- triggering evaluation: turning lazy into let
- strict positions

lazy terms

- triggering evaluation: turning lazy into let
- strict positions
- for lazy term variables:
  - $S ::= \_\ e \mid \_\ \tau \mid \text{let } \langle x_1, x_2 \rangle = \_ \text{ in } e \mid \text{let } \langle \alpha, x \rangle = \_ \text{ in } e$
  - $L_1[\text{lazy } \langle \zeta, x \rangle = e \text{ in } L_2 ES[x]] \longrightarrow$
    $L_1[\text{let } \langle \alpha, x \rangle = e \text{ in } (L_2 ES[x])[\zeta := \alpha]]$

lazy types

- "value" of run-time types only needed by tcase
- due to lazy linking, type expressions may contain lazy type variables
- comparison needs to know the types they represent
- example: $LE[\text{tcase } x{:}(\zeta \times \zeta) \text{ of } x'{:}(\text{int} \times \text{int}) \text{ then } e_1 \text{ else } e_2]$

1st strategy

- type equivalence checking through normalize-and-compare
- $LE[\text{tcase } v{:}\nu \text{ of } x{:}\nu \text{ then } e_1 \text{ else } e_2] \longrightarrow LE[e_1[x := v]]$
- $LE[\text{tcase } v{:}\nu \text{ of } x{:}\nu' \text{ then } e_1 \text{ else } e_2] \longrightarrow LE[e_2] \quad (\nu \neq \nu')$

$$
\begin{array}{lrl}
\text{normal forms} & \nu ::= & p \mid \nu_1 \to \nu_2 \mid \nu_1 \times \nu_2 \mid \forall\alpha{:}\kappa.\nu \mid \\
 & & \exists\alpha{:}\kappa.\nu \mid \lambda\alpha{:}\kappa.\nu \mid \langle \nu_1, \nu_2 \rangle \\
\text{normal paths} & p ::= & \alpha \mid p\ \nu \mid p.1 \mid p.2
\end{array}
$$

1st strategy

- normalization: applicative order reduction
- elimination of lazy type variables as they are encountered

$$
\begin{array}{rl}
C ::= & \text{tcase } v{:}\_ \text{ of } x{:}\tau \text{ then } e_1 \text{ else } e_2 \mid \\
& \text{tcase } v{:}\nu \text{ of } x{:}\_ \text{ then } e_1 \text{ else } e_2 \\
T ::= & \_ \mid T \to \tau \mid \nu \to T \mid T \times \tau \mid \nu \times T \mid \\
& \forall\alpha{:}\kappa.T \mid \exists\alpha{:}\kappa.T \mid \lambda\alpha{:}\kappa.T \mid \\
& T\ \tau \mid \nu\ T \mid \langle T, \tau\rangle \mid \langle \nu, T\rangle \mid T.1 \mid T.2
\end{array}
$$

$$
\begin{array}{rcl}
LECT[(\lambda\alpha{:}\kappa.\nu)\ \nu'] & \longrightarrow & LECT[\nu[\alpha := \nu']] \\
LECT[\langle\nu_1, \nu_2\rangle.1] & \longrightarrow & LECT[\nu_1] \\
LECT[\langle\nu_1, \nu_2\rangle.2] & \longrightarrow & LECT[\nu_2] \\
L_1[\text{lazy } \langle\zeta, x\rangle = e_1 \text{ in } L_2ECT[\zeta]] & \longrightarrow & L_1[\text{let } \langle\alpha, x\rangle = e_1 \text{ in} \\
& & \quad (L_2ECT[\zeta])[\zeta := \alpha]]
\end{array}
$$

degree of laziness

- laziness can be improved
- consider:
  - lazy $\langle \zeta, x \rangle$ = e in tcase x:$((\lambda\alpha{:}\Omega.\mathsf{int})\ \zeta)$ of x':int then e_1 else e_2
  - lazy $\langle \zeta, x \rangle$ = e in tcase x:$(\zeta \to \mathsf{int})$ of x':(int$\times$int) then e_1 else e_2
- ideas for a lazier strategy:
  - call-by-name
  - shape-comparison during normalization

2nd strategy

- algorithm:
  1. reduce the types to weak head normal form

     $$\begin{aligned} \omega &::= q \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2 \mid \forall\alpha{:}\kappa.\tau \mid \exists\alpha{:}\kappa.\tau \mid \lambda\alpha{:}\kappa.\tau \mid \langle\tau_1, \tau_2\rangle \\ q &::= \alpha \mid q\ \tau \mid q.1 \mid q.2 \end{aligned}$$

  2. compare their heads
  3. if different, abort by reducing to the else-branch
  4. otherwise descend and repeat this procedure

2nd strategy

- algorithm:
  1. reduce the types to weak head normal form

     $$\begin{aligned} \omega &::= \; q \mid \tau_1 \rightarrow \tau_2 \mid \tau_1 \times \tau_2 \mid \forall\alpha{:}\kappa.\tau \mid \exists\alpha{:}\kappa.\tau \mid \lambda\alpha{:}\kappa.\tau \mid \langle\tau_1, \tau_2\rangle \\ q &::= \; \alpha \mid q\ \tau \mid q.1 \mid q.2 \end{aligned}$$

  2. compare their heads
  3. if different, abort by reducing to the else-branch
  4. otherwise descend and repeat this procedure
- first typecase rule as before:
  $LE[\text{tcase } v{:}\nu \text{ of } x{:}\nu \text{ then } e_1 \text{ else } e_2] \longrightarrow LE[e_1[x := v]]$
- $LE[\text{tcase } v{:}\tau \text{ of } x{:}\tau' \text{ then } e_1 \text{ else } e_2] \longrightarrow LE[e_2]$
  if $(\text{tcase } v{:}\tau \text{ of } x{:}\tau' \text{ then } e_1 \text{ else } e_2) = B[\omega][\omega']$ with $\omega \not\sim \omega'$
  or $(\text{tcase } v{:}\tau \text{ of } x{:}\tau' \text{ then } e_1 \text{ else } e_2) = P[q][q']$ with $q \not\sim q'$

2nd strategy

- binary contexts determine how to descend into types of the same shape

$$
\begin{aligned}
B ::= \;& \mathsf{tcase}\ v{:}\_\ \mathsf{of}\ x{:}\_\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mid \\
& B[\exists\alpha{:}\kappa.\_][\exists\alpha{:}\kappa.\_] \mid \\
& B[\_ \times \tau_1][\_ \times \tau_2] \mid B[\nu \times \_][\nu \times \_] \mid \ldots
\end{aligned}
$$

- sample decomposition of
  $\mathsf{tcase}\ v{:}(\exists\alpha{:}\Omega.\alpha \times \zeta)\ \mathsf{of}\ x{:}(\exists\alpha{:}\Omega.\alpha \times \mathsf{int})\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$
  - $B_0 = \mathsf{tcase}\ v{:}\_\ \mathsf{of}\ x{:}\_\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$
  - $B_1 = B_0[\exists\alpha{:}\Omega.\_][\exists\alpha{:}\Omega.\_]$
  - $B_2 = B_1[\alpha \times \_][\alpha \times \_]$
  - $\rightsquigarrow B_2[\zeta][\mathsf{int}]$

2nd strategy

- weak head normalization:

$$\begin{aligned} C &::= B[\_][\tau] \mid B[\omega][\_] \\ T &::= \_ \mid T\ \tau \mid T.1 \mid T.2 \end{aligned}$$

$$\begin{aligned} LECT[(\lambda\alpha{:}\kappa.\tau_1)\ \tau_2] &\longrightarrow LECT[\tau_1[\alpha := \tau_2]] \\ LECT[\langle\tau_1, \tau_2\rangle.1] &\longrightarrow LECT[\tau_1] \\ LECT[\langle\tau_1, \tau_2\rangle.2] &\longrightarrow LECT[\tau_2] \\ L_1[\text{lazy } \langle\zeta, x\rangle = e_1 \text{ in } L_2ECT[\zeta]] &\longrightarrow L_1[\text{let } \langle\alpha, x\rangle = e_1 \text{ in} \\ &\qquad\quad (L_2ECT[\zeta])[\zeta := \alpha]] \end{aligned}$$

preservation property

- theorem: if $\Gamma \vdash e : \tau$ and $e \longrightarrow e'$, then $\Gamma \vdash e' : \tau$
- proof idea: cases $LE[e_1] \longrightarrow LE[e_1']$
  1. by Context Elimination: $\Gamma, \Gamma' \vdash e_1 : \tau'$ with $\Gamma \vdash L : \Gamma'$
  2. by ...: $\Gamma, \Gamma' \vdash e_1' : \tau'$
  3. By Exchange: $\Gamma \vdash LE[e_1'] : \tau'$

preservation property

- theorem: if $\Gamma \vdash e : \tau$ and $e \longrightarrow e'$, then $\Gamma \vdash e' : \tau$
- proof idea: cases $LE[e_1] \longrightarrow LE[e_1']$
  1. by Context Elimination: $\Gamma, \Gamma' \vdash e_1 : \tau'$ with $\Gamma \vdash L : \Gamma'$
  2. by ...: $\Gamma, \Gamma' \vdash e_1' : \tau'$
  3. By Exchange: $\Gamma \vdash LE[e_1'] : \tau'$
- type-level application (1st strategy): $e_1 = CT[(\lambda\alpha{:}\kappa.\nu)\ \nu')]$, $e_1' = CT[\nu[\alpha := \nu']]$ uses Type Context Elimination, Type Substitution, Type Exchange

progress property

- standard formulation: if $\cdot \vdash e : \tau$ and $e \neq L[v]$, then $e \longrightarrow e'$
- what if e is lazy $\langle \zeta, x \rangle = e_1$ in e_2?
- our formulation: if $\cdot \vdash L[e] : \tau$ where e is neither a value nor a lazy expression, then $L[e] \longrightarrow e'$ (not $L[e']$)

progress property

- sample case: $L[e] = L[\text{tcase } v{:}\tau_0 \text{ of } x{:}\tau_0' \text{ then } e_1 \text{ else } e_2]$
  - lemma for 1st strategy:
    if $\cdot \vdash LCT[\tau] : \tau'$ and τ not normal, then $LCT[\tau] \longrightarrow e$
  - lemma for 2nd strategy:
    if $\cdot \vdash LB[\tau_1][\tau_2] : \tau$, then $LB[\tau_1][\tau_2] \longrightarrow e$
    claim follows for $B = \text{tcase } v{:}\_ \text{ of } x{:}\_ \text{ then } e_1 \text{ else } e_2$

progress property

if $\cdot \vdash LB[\tau_1][\tau_2] : \tau$, then $LB[\tau_1][\tau_2] \longrightarrow e$

- proof by induction on $\mathit{weight}(B, \tau_1, \tau_2)$
- case $\tau_1 = \alpha = \tau_2$ requires another lemma to use induction hypothesis

conclusion

- model for the integration of dynamic type checking and lazy linking into a language that provides higher-order polymorphism
- two strategies for dealing with free type variables that represent yet unknown types

conclusion

- model for the integration of dynamic type checking and lazy linking into a language that provides higher-order polymorphism
- two strategies for dealing with free type variables that represent yet unknown types
- future work: subtyping?

references

- Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. **Dynamic typing in a statically typed language**. ACM Transactions on Programming Languages and Systems, 13(2):237-268, April 1991.
- Zena Ariola, Matthias Felleisen, John Maraist, Martin Odersky, and Philip Wadler. **A call-by-need lambda calculus**. 22'nd Symposium on Principles of Programming Languages, ACM Press, San Francisco, California, January 1995.
- Peter Sestoft. **Demonstrating lambda calculus reduction**. In The Essence of Computation: Complexity, Analysis, Transformation. Springer-Verlag, 2002.
- Karl Crary. **Logical Relations and a Case Study in Equivalence Checking**. Benjamin C. Pierce, editor, Advanced Topics in Types and Programming Languages, 2005.

references

- 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
- Andreas Rossberg. **The Missing Link - Dynamic Components for ML**. 11th International Conference on Functional Programming, Portland, Oregon, USA, ACM Press.
- Andreas Rossberg. **Typed open programming**. PhD thesis, Programming Systems Lab, Universität des Saarlandes, 2006. To appear.