A Polymorphic Modal Type System for Lisp-like Multi-Staged Languages

Ik-Soon Kim\textsuperscript{1}  Kwangkeun Yi\textsuperscript{1}  Cristiano Calcagno\textsuperscript{2}

\textsuperscript{1}Seoul National University  \textsuperscript{2}Imperial College

POPL’06, 1/12/2006 @ Charleston
1. Introduction and Challenge
2. Contribution and Ideas
3. Simple Type System
4. Polymorphic Type System
5. Conclusion
program texts (code) as first class objects
“meta programming”

A general concept that subsumes
- macros
- Lisp/Scheme’s quasi-quotation
- partial evaluation
- runtime code generation
divides a computation into stages

- program at stage 0: conventional program
- program at stage \( n + 1 \): code as data at stage \( n \)

<table>
<thead>
<tr>
<th>Stage</th>
<th>Computation</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>usual + code + eval</td>
<td>usual + code</td>
</tr>
<tr>
<td>&gt; 0</td>
<td>code substitution</td>
<td>code</td>
</tr>
</tbody>
</table>
In examples, we will use Lisp-style staging constructs + only 2 stages

\[ e ::= \cdots \]

\[ \text{'}e\text{'} \quad \text{code as data} \]

\[ ,e \quad \text{code substitution} \]

\[ \text{eval } e \quad \text{execute code} \]
In examples, we will use Lisp-style staging constructs + only 2 stages

\[
ed ::= \ldots
| \ 'e' \quad \text{code as data}
| , e \quad \text{code substitution}
| \text{eval } e \quad \text{execute code}
\]

Code as data

let NULL = '0
let body = '(if e = ,NULL then abort() ...)
in eval body
Specializer/Partial evaluator

\[
\text{power}(x,n) = \text{if } n=0 \text{ then } 1 \text{ else } x \times \text{power}(x,n-1)
\]

v.s. \[
\text{power}(x,3) = x \times x \times x
\]

prepared as

\[
\text{let } \text{spower}(n) = \text{if } n=0 \text{ then '1 else '('x*,(spower (n-1)))}
\]

\[
\text{let } \text{fastpower10} = \text{eval '}(\lambda x.,(\text{spower 10}))
\]

\[
\text{in } \text{fastpower10 2}
\]
Review: Practice of Multi-Staged Programming

Features of Lisp/Scheme’s quasi-quotation system
Review: Practice of Multi-Staged Programming

- open code

\[(x+1)\]

Features of Lisp/Scheme's quasi-quotation system
open code

\( (x+1) \)

intentional variable-capturing substitution at stages \( > 0 \)

\( (\lambda x., (spower 10)) \)

Features of Lisp/Scheme's quasi-quotiation system
open code
‘(x+1)

intentional variable-capturing substitution at stages > 0
‘(λx., (spower 10))

capture-avoiding substitution
‘(λ*x., (spower 10) + x)

Features of Lisp/Scheme’s quasi-quotation system
open code

\((x+1)\)

intentional variable-capturing substitution at stages \(> 0\)

\(\text{`(\lambda x.,(spower 10))')}\)

capture-avoiding substitution

\(\text{`(\lambda^* x.,(spower 10) + x)'}\)

imperative operations with open code

\text{cell := `(x+1); · · · cell := `(y 1);}

Features of Lisp/Scheme’s quasi-quotation system
A static type system that supports the practice.

Should allow programmers both
  - type safety and
  - the expressiveness of Lisp/Scheme’s quasi-quote operators

Existing type systems support only part of the practice.
Our Contribution

A type system for ML + Lisp’s quasi-quote system

- supports multi-staged programming practice
  - open code: `(x+1)`
  - unrestricted imperative operations with open code
  - intentional var-capturing substitution at stages $> 0$
  - capture-avoiding substitution at stages $> 0$

- conservative extension of ML’s let-polymorphism

- principal type inference algorithm
Comparison

<table>
<thead>
<tr>
<th></th>
<th>closed code and eval</th>
<th>open code</th>
<th>imperative operations</th>
<th>type inference</th>
<th>var-capturing subst.</th>
<th>capture-avoiding subst.</th>
<th>polymorphism</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Our system**

[Rhiger 2005]

[Calcagno et al. 2004]

[Ancona & Moggi 2004]

[Taha & Nielson 2003]

[Chen & Xi 2003]

[Nanevsky & Pfenning 2002]

MetaML/Ocaml[2000,2001]

[Davies 1996]

[Davies & Pfenning 1996,2001]
- code’s type: parameterized by its expected context
  \( \Box (\Gamma \triangleright int) \)

- view the type environment \( \Gamma \) as a record type
  \( \Gamma = \{ x : int, y : int \to int, \cdots \} \)

- stages by the stack of type environments (modal logic S4)
  \( \Gamma_0 \cdots \Gamma_n \vdash e : A \)

- with “due” restrictions
  - let-polymorphism for syntactic values
  - monomorphic \( \Gamma \) in code type \( \Box (\Gamma \triangleright int) \)
  - monomorphic store types

Natural ideas worked.
Multi-Staged Language

\[ e ::= c \mid x \mid \lambda x.e \mid e \cdot e \]

\mid \text{box } e \quad \text{code as data} \quad \text{'} e

\mid \text{unbox}_k e \quad \text{code substitution} \quad , \ldots , e

\mid \text{eval } e \quad \text{execute code}

\mid \lambda^* x.e \quad \text{gensym}

\mid \ldots

Evaluation

\[ \mathcal{E} \vdash e \xrightarrow{n} r \]

where

\[ \mathcal{E}: \text{value environment} \]

\[ n: \text{a stage number} \]

\[ r: \text{a value or } \text{err} \]
Operational Semantics (stage $n \geq 0$)

- at stage 0: normal evaluation + code + eval
- at stage $> 0$: code substitution

(EBOX) \[
\frac{\mathcal{E} \vdash e \overset{n+1}{\to} v}{\mathcal{E} \vdash \text{box } e \overset{n}{\to} \text{box } v}
\]

(EUNBOX) \[
\frac{\mathcal{E} \vdash e \overset{0}{\to} \text{box } v \quad k > 0}{\mathcal{E} \vdash \text{unbox}_k e \overset{k}{\to} v}
\]

(EEVAL) \[
\frac{\mathcal{E} \vdash e \overset{0}{\to} \text{box } v \quad \mathcal{E} \vdash v \overset{0}{\to} v'}{\mathcal{E} \vdash \text{eval } e \overset{0}{\to} v'}
\]
Simple Type System (1/2)

\[ Type \quad A, B \quad ::= \quad \iota \mid A \rightarrow B \mid \square(\Gamma \triangleright A) \]

code type

\[ \text{\textquotesingle}(x+1) : \quad \square(\{x : \textit{int}, \cdots \} \triangleright \textit{int}) \]

typing judgment

\[ \Gamma_0 \cdots \Gamma_n \vdash e : A \]
Simple Type System (1/2)

\[
\text{Type} \quad A, B \ ::= \ \iota \mid A \to B \mid \Box (\Gamma \triangleright A)
\]

code type

\[
\text{'}(x+1): \ \Box (\{x: \text{int}, \cdots\} \triangleright \text{int})
\]

typing judgment

\[
\Gamma_0 \cdots \Gamma_n \vdash e : A
\]

(TSBOX) \[ \Gamma_0 \cdots \Gamma_n \Gamma \vdash e : A \quad \frac{}{\Gamma_0 \cdots \Gamma_n \vdash \text{box} \ e : \Box (\Gamma \triangleright A)} \]

(TSUNBOX) \[ \Gamma_0 \cdots \Gamma_n \vdash e : \Box (\Gamma_{n+k} \triangleright A) \quad \frac{}{\Gamma_0 \cdots \Gamma_n \cdots \Gamma_{n+k} \vdash \text{unbox}_k \ e : A} \]

(TSEVAL) \[ \Gamma_0 \cdots \Gamma_n \vdash e : \Box (\emptyset \triangleright A) \quad \frac{}{\Gamma_0 \cdots \Gamma_n \vdash \text{eval} \ e : A} \quad \text{(for alpha-equiv. at stage 0)} \]
(TSCON) \[ \Gamma_0 \cdots \Gamma_n \vdash c : \iota \]

(TSVAR) \[ \frac{\Gamma_n(x) = A}{\Gamma_0 \cdots \Gamma_n \vdash x : A} \]

(TSABS) \[ \frac{\Gamma_0 \cdots (\Gamma_n + x : A) \vdash e : B}{\Gamma_0 \cdots \Gamma_n \vdash \lambda x.e : A \to B} \]

(TSGENS) \[ \frac{\Gamma_0 \cdots (\Gamma_n + w : A) \vdash [x^n \mapsto w] e : B \quad \text{fresh } w}{\Gamma_0 \cdots \Gamma_n \vdash \lambda^* x.e : A \to B} \]

(TSAPP) \[ \frac{\Gamma_0 \cdots \Gamma_n \vdash e_1 : A \to B \quad \Gamma_0 \cdots \Gamma_n \vdash e_2 : A}{\Gamma_0 \cdots \Gamma_n \vdash e_1 e_2 : B} \]
A combination of

- ML’s let-polymorphism
  - syntactic value restriction + multi-staged “expansive$^n(e)$”
  - expansive$^n(e) = False$
    \implies e$ never expands the store during its eval. at $\forall$ stages $\leq n$
  
  e.g.) ‘(\lambda x.e) : can be expansive
        ‘(\lambda x.eval y) : unexpansive

- Rémy’s record types [Rémy 1993]
  - type environments as record types with field addition
  - record subtyping + record polymorphism
if \( e \) then ‘(x+1) else ‘1:

then-branch: \( \Box(\{x : \text{int}\} \rho \triangleright \text{int}) \)
else-branch: \( \Box(\rho'' \triangleright \text{int}) \)

let x = ‘y in ‘((,x + w); ‘((,x 1) + z)

x: \( \forall \alpha \forall \rho. \Box(\{y : \alpha\} \rho \triangleright \alpha) \)

first x: \( \Box(\{y : \text{int}, w : \text{int}\} \rho' \triangleright \text{int}) \)
second x: \( \Box(\{y : \text{int} \rightarrow \text{int}, z : \text{int}\} \rho'' \triangleright \text{int} \rightarrow \text{int}) \)
typing judgment

\[ \Delta_0 \ldots \Delta_n \vdash e : A \]

\[ \frac{\Delta_0 \ldots \Delta_n \Gamma \vdash e : A}{\Delta_0 \ldots \Delta_n \vdash \text{box } e : \square (\Gamma \triangleright A)} \]

\text{(TBOX)}

\[ \frac{\Delta_0 \ldots \Delta_n \vdash e : \square (\Gamma \triangleright A)}{\Delta_0 \ldots \Delta_n \vdash \text{unbox}_k e : A} \]

\[ \text{unbox}_k \]

\[ \frac{\Delta_0 \ldots \Delta_n \vdash \text{eval } e : A}{\Delta_0 \ldots \Delta_n \vdash e : A} \]

\text{(TEVAL)}
Polymorphic Type System (4/4)

(TVAR)  \[
\frac{\Delta_n(x) \succ A}{\Delta_0 \ldots \Delta_n \vdash x : A}
\]

(TABS)  \[
\frac{\Delta_0 \ldots (\Delta_n + x : A) \vdash e : B}{\Delta_0 \ldots \Delta_n \vdash \lambda x. e : A \rightarrow B}
\]

(TAPP)  \[
\frac{\Delta_0 \ldots \Delta_n \vdash e_1 : A \rightarrow B \quad \Delta_0 \ldots \Delta_n \vdash e_2 : A}{\Delta_0 \ldots \Delta_n \vdash e_1 e_2 : B}
\]

(TLETIMP)  \[
\begin{aligned}
&\text{expansive}^n(e_1) \\
&\frac{\Delta_0 \ldots \Delta_n \vdash e_1 : A \quad \Delta_0 \ldots \Delta_n + x : A \vdash e_2 : B}{\Delta_0 \ldots \Delta_n \vdash \text{let}(x \ e_1) \ e_2 : B}
\end{aligned}
\]

(TLETAPP)  \[
\begin{aligned}
&\neg \text{expansive}^n(e_1) \\
&\frac{\Delta_0 \ldots \Delta_n \vdash e_1 : A \quad \Delta_0 \ldots \Delta_n + x : GEN_A(\Delta_0 \ldots \Delta_n) \vdash e_2 : B}{\Delta_0 \ldots \Delta_n \vdash \text{let}(x \ e_1) \ e_2 : B}
\end{aligned}
\]
Unification:
- Rémy’s unification for record type $\Gamma$
- usual unification for new type terms such as $\Box(\Gamma \triangleright A)$ and $A$ ref

Type inference algorithm:
- the same structure as top-down version $M$ [Lee and Yi 1998] of the $W$
- usual on-the-fly instantiation and unification
Unification:
- Rémy’s unification for record type $\Gamma$
- usual unification for new type terms such as $\square(\Gamma \triangleright A)$ and $A\text{ ref}$

Type inference algorithm:
- the same structure as top-down version $\mathcal{M}$ [Lee and Yi 1998]
  - of the $\mathcal{W}$
- usual on-the-fly instantiation and unification

**Sound**
If $\text{infer}(\emptyset, e, \alpha) = S$ then $\emptyset; \emptyset \vdash e : S\alpha$.

**Complete**
If $\emptyset; \emptyset \vdash e : R\alpha$ then $\text{infer}(\emptyset, e, \alpha) = S$ and $R = TS$ for some $T$. 
A type system for ML + Lisp’s quasi-quote system

- supports multi-staged programming practice
- conservative extension to ML’s let-polymorphism
- principal type inference algorithm

Exact details, lemmas, proof sketches, and embedding relations in the paper; full proofs in the technical report.
A type system for ML + Lisp’s quasi-quote system
  - supports multi-staged programming practice
  - conservative extension to ML’s let-polymorphism
  - principal type inference algorithm

Exact details, lemmas, proof sketches, and embedding relations in the paper; full proofs in the technical report.

Staged programming “practice” has a sound static type system.
Conclusion

A type system for ML + Lisp’s quasi-quote system
- supports multi-staged programming practice
- conservative extension to ML’s let-polymorphism
- principal type inference algorithm

Exact details, lemmas, proof sketches, and embedding relations in the paper; full proofs in the technical report.

Staged programming “practice” has a sound static type system.

Thank you.