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 \quad \textsuperscript{2}Imperial College

based on the slides at POPL’06, 1/12/2006 @ Charleston
1. Introduction and Challenge
2. Contribution and Ideas
3. Simple Type System
4. Polymorphic Type System
5. To Read to Catch Up
6. Challenges Ahead
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
Multi-Staged Programming (2/2)

- 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

\[
e ::= \ldots
\]

<table>
<thead>
<tr>
<th><code>e</code></th>
<th>code as data</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>, e</code></td>
<td>code substitution</td>
</tr>
<tr>
<td><code>eval e</code></td>
<td>execute code</td>
</tr>
</tbody>
</table>
In examples, we will use Lisp-style staging constructs + only 2 stages

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

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

\[
\text{let } \text{spower}(n) = \text{if } n=0 \text{ then } '1 \text{ else } '(x*,(\text{spower}(n-1)))
\]
\[
\text{let } \text{fastpower10} = \text{eval } '(
\lambda x.,(\text{spower } 10))
\]
\[
\text{in } \text{fastpower10} 2
\]
Features of Lisp/Scheme's quasi-quotation system
• open code

‘(x+1)

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

‘(x+1)

intentional variable-capturing substitution at stages > 0

‘(λx.,(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
Review: Practice of Multi-Staged Programming

- open code
  \[\text{'(x+1)}\]
- intentional variable-capturing substitution at stages \(\geq 0\)
  \[\text{'(\(\lambda x.\), (spower 10)})}\]
- capture-avoiding substitution
  \[\text{'(\(\lambda^* x.\), (spower 10) + x)}\]
- imperative operations with open code
  \[
  \text{cell := '}(x+1); \ldots \text{cell := '}(y 1);
  \]

Features of Lisp/Scheme's quasi-quotiation 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

1. closed code and eval
2. open code
3. imperative operations
4. type inference
5. var-capturing subst.
6. capture-avoiding subst.
7. polymorphism

<table>
<thead>
<tr>
<th>Our system</th>
<th>+1 +2 +3 +4 +5 +6 +7</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Rhiger 2005]</td>
<td>+1 +2 +3 –4 +5 −6 −7</td>
</tr>
<tr>
<td>[Calcagno et al. 2004]</td>
<td>+1 +2 −3 +4 −5 +6 +7</td>
</tr>
<tr>
<td>[Ancona &amp; Moggi 2004]</td>
<td>+1 +2 +3 −4 −5 +6 −7</td>
</tr>
<tr>
<td>[Taha &amp; Nielson 2003]</td>
<td>+1 +2 −3 −4 −5 +6 +7</td>
</tr>
<tr>
<td>[Chen &amp; Xi 2003]</td>
<td>+1 +2 +3 −4 +5 −6 +7</td>
</tr>
<tr>
<td>[Nanevsky &amp; Pfenning 2002]</td>
<td>+1 +2 +3 −4 −5 +6 −7</td>
</tr>
<tr>
<td>MetaML/Ocaml[2000,2001]</td>
<td>+1 +2 −3 +4 −5 +6 +7</td>
</tr>
<tr>
<td>[Davies 1996]</td>
<td>−1 +2 −3 −4 −5 +6 −7</td>
</tr>
<tr>
<td>[Davies &amp; Pfenning 1996,2001]</td>
<td>+1 −2 +3 +4 −5 +6 −7</td>
</tr>
</tbody>
</table>
Ideas

- code’s type: parameterized by its expected context

\[ \square(\Gamma \triangleright int) \]

- 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 \( \square(\Gamma \triangleright int) \)
  - monomorphic store types

Natural ideas worked.
Multi-Staged Language

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

| box \( e \)                      | code as data            | \( ^c e \)          |
| unbox_k \( e \)                | code substitution       | \( \ldots, e \)     |
| eval \( e \)                   | execute code            |                     |
| \( \lambda^*x.e \)             | gensym                   |                     |
| \ldots                         |                         |                     |

Evaluation

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

where

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

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

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

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

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

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

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

Type \( A, B \) ::= \( \iota \) | \( A \to B \) | \( \Box (\Gamma \triangleright A) \)

code type

\( (x+1): \Box (\{x : int, \cdots \} \triangleright int) \)

typing judgment

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

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

code type

\[ \text{\texttt{\textquotesingle} (x+1)} : \square(\{x : \text{int}, \cdots \} \to \text{int}) \]

typing judgment

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

\begin{align*}
\text{(TSBOX)} & \quad \frac{\Gamma_0 \cdots \Gamma_n \Gamma \vdash e : A}{\Gamma_0 \cdots \Gamma_n \vdash \text{\texttt{box}} \ e \ : \ \square(\Gamma \to A)} \\
\text{(TSUNBOX)} & \quad \frac{\Gamma_0 \cdots \Gamma_n \vdash e : \square(\Gamma_{n+k} \to A)}{\Gamma_0 \cdots \Gamma_n \cdots \Gamma_{n+k} \vdash \text{\texttt{unbox}}_k e : A} \\
\text{(TSEVAL)} & \quad \frac{\Gamma_0 \cdots \Gamma_n \vdash e : \square(\emptyset \to A)}{\Gamma_0 \cdots \Gamma_n \vdash \text{\texttt{eval}} \ e : A} \quad \text{(for alpha-equiv. at stage 0)}
\end{align*}
Simple Type System (2/2)

(TSCON) \[ \Gamma_0 \cdots \Gamma_n \vdash c : \iota \]

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

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

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

(TSAPP) \[ \Gamma_0 \cdots \Gamma_n \vdash e_1 : A \to B \]
\[ \Gamma_0 \cdots \Gamma_n \vdash e_2 : A \]
\[ \frac{\Gamma_0 \cdots \Gamma_n \vdash e_1 e_2 : B}{\Gamma_0 \cdots \Gamma_n \vdash e : B} \]
Lemma (Preservation)

If $\Gamma_0 \cdots \Gamma_n \vdash e : A$ and $\mathcal{E} \vdash e \xrightarrow{n} r$ for $\models \mathcal{E} : \Gamma_0$, then $\emptyset \Gamma_1 \cdots \Gamma_n \vdash r : A$. 
A combination of

- ML’s let-polymorphism
  - syntactic value restriction + multi-staged “expansive\(^n\)\(e\)”
  - \(\text{expansive}^n(e) = \text{False}\)
    \(\implies e\) never expands the store during its eval. at \(\forall\text{stages} \leq n\)
  - e.g.) \((\lambda x.e)\) : can be expansive
    \((\lambda x.\text{eval } y)\) : 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: \[\Box(\{x : int\} \rho \triangleright int)\]

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

let x = 'y in 'x + w; 'x 1) + z
\[x: \forall \alpha \forall \rho. \Box(\{y : \alpha\} \rho \triangleright \alpha)\]

- first x: \[\Box(\{y : int, w : int\} \rho' \triangleright int)\]
- second x: \[\Box(\{y : int \rightarrow int, z : int\} \rho'' \triangleright int \rightarrow int)\]
Polymorphic Type System (3/4)

typing judgment

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

(TBOX)

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

(TUNBOX)

\[ \frac{\Delta_0 \cdots \Delta_n \vdash e : \Box (\Gamma \triangleright A) \quad \Delta_{n+k} \triangleright \Gamma \quad k > 0}{\Delta_0 \cdots \Delta_n \cdots \Delta_{n+k} \vdash \text{unbox}_k e : A} \]

(TEVAL)

\[ \frac{\Delta_0 \cdots \Delta_n \vdash e : \Box (\emptyset \triangleright A)}{\Delta_0 \cdots \Delta_n \vdash \text{eval } e : A} \]
(TVAR) \[ \frac{\Delta_n(x) \succ A}{\Delta_0 \cdots \Delta_n \vdash x : A} \]

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

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

(expansive\(^n\)(e_1)) \[ \frac{\Delta_0 \cdots \Delta_n \vdash e_1 : A \quad \Delta_0 \cdots \Delta_n + x : A \vdash e_2 : B}{\Delta_0 \cdots \Delta_n \vdash \text{let}(x \ e_1) \ e_2 : B} \]

(\neg \text{expansive}\(^n\)(e_1)) \[ \frac{\Delta_0 \cdots \Delta_n \vdash e_1 : A}{\Delta_0 \cdots \Delta_n + x : GEN_A(\Delta_0 \cdots \Delta_n) \vdash e_2 : B} \]

(TLETAPP) \[ \frac{\Delta_0 \cdots \Delta_n \vdash e_1 : A}{\Delta_0 \cdots \Delta_n \vdash \text{let}(x \ e_1) \ e_2 : B} \]
Lemma (Preservation)

If $\Delta_0 \cdots \Delta_n \vdash e : A$, and $\mathcal{E} \vdash e \xrightarrow{n} r$ for $\models \mathcal{E} : \Delta_0$ then $\emptyset \Delta_1 \cdots \Delta_n \vdash r : A$. 
Unification:
- Rémy’s unification for record type $\Gamma$
- usual unification for new type terms such as $\square(\Gamma \triangleright A)$ and $A$ 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
Type Inference Algorithm

- **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 $\mathcal{M}$ [Lee and Yi 1998] of the $\mathcal{W}$
  - usual on-the-fly instantiation and unification

Sound  If $\text{infer}(\emptyset, e, \alpha) = S$ then $\emptyset \vdash e : S\alpha$.
Complete  If $\emptyset \vdash e : R\alpha$ 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 are 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 are in the technical report.

Staged programming “practice” has a sound static type system.
Preliminary basis to catch up:

- ~kwang/book/pl/root.dvi: chapter 1 and 2
- SNU 4541.664A homepage: Papers/Type-Based Analysis
  - paper no.2, no.3, and no.7

Read references in the paper.
design of closure analysis
  - challenge: how to abstract infinite code (maybe easy)
useless code analysis for staged language: type-based
  - challenge: extend Kobayashi’s type-based approach
extend the type system for exceptions
  - challenge: add proof cases for the preservation lemmas
exception analysis for staged programs
  - challenge: type-based approach
  - challenge: ai-based/constraint-based approach
research tool developments
  - MetanML interpreter: syntax design + interpreter
  - applications: self-evolving code, partial evaluator, etc
  
...