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

Kwangkeun Yi

SNU 4541.780, Fall 2006

Lecture 1
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

\[ e ::= \cdots \]
\[ | \ 'e \quad \text{code as data} \]
\[ | \ ,e \quad \text{code substitution} \]
\[ | \ \text{eval } e \quad \text{execute code} \]
In examples, we will use Lisp-style staging constructs + only 2 stages

\[
e ::= \ldots
\]

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

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

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

Code as data

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

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

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

prepared as

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

$(x+1)$

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

'\((x+1)\)

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

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

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

\((x+1)\)

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

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

capture-avoiding substitution

\((\lambda^* x., (spower 10) + x)\)

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

\begin{align*}
\texttt{'(x+1)}
\end{align*}

textual

intentional variable-capturing substitution at stages $> 0$

\begin{align*}
\texttt{'(λx.,(spower 10))}
\end{align*}

capture-avoiding substitution

\begin{align*}
\texttt{'(λ^∗x.,(spower 10) + x)}
\end{align*}

imperative operations with open code

\begin{align*}
cell := \texttt{'(x+1)}; \cdots cell := \texttt{'(y 1)};
\end{align*}

Features of Lisp/Scheme's quasi-quotient system
Challenge

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>(1)</th>
<th>closed code and eval</th>
<th>(2)</th>
<th>open code</th>
</tr>
</thead>
<tbody>
<tr>
<td>(3)</td>
<td>imperative operations</td>
<td>(4)</td>
<td>type inference</td>
</tr>
<tr>
<td>(5)</td>
<td>var-capturing subst.</td>
<td>(6)</td>
<td>capture-avoiding subst.</td>
</tr>
<tr>
<td>(7)</td>
<td>polymorphism</td>
<td>(8)</td>
<td>alpha equiv. at stage ( n + 1 )</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]  

\[
\begin{align*}
+1 & +2 & +3 & +4 & +5 & +6 & +7 & -8 \\
+1 & +2 & +3 & -4 & +5 & -6 & -7 & -8 \\
+1 & +2 & -3 & +4 & -5 & +6 & +7 & +8 \\
+1 & +2 & +3 & -4 & -5 & +6 & -7 & +8 \\
+1 & +2 & -3 & -4 & -5 & +6 & +7 & +8 \\
+1 & +2 & +3 & -4 & +5 & -6 & +7 & -8 \\
+1 & +2 & +3 & -4 & -5 & +6 & -7 & +8 \\
-1 & +2 & -3 & -4 & -5 & +6 & -7 & +8 \\
+1 & -2 & +3 & +4 & -5 & +6 & -7 & +8
\end{align*}
\]
code’s type: parameterized by its expected context

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

view the type environment \( \Gamma \) as a record type

\[ \Gamma = \{ x : int, \ y : int \rightarrow int, \cdots \} \]

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

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

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

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

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

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

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

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

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

code type

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

typing judgment

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

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

code type

\[ \langle x+1 \rangle : \square(\{x : int, \cdots \} \triangleright int) \]

typing judgment

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

(TSBOX)

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

(TSUNBOX)

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

(TSEVAL)

\[
\frac{\Gamma_0 \cdots \Gamma_n \vdash e : \square(\emptyset \triangleright A)}{\Gamma_0 \cdots \Gamma_n \vdash \text{eval} \ e : A} \quad \text{(for alpha-equiv. at stage 0)}
\]
Simple Type System (2/2)

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

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

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

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

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

- ML’s let-polymorphism
  - syntactic value restriction + multi-staged “expansive\(^n\)(e)”
  - expansive\(^n\)(e) = \textit{False}
    \[ \implies e \text{ never expands the store during its eval. at } \forall \text{stages} \leq n \]
  - 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: □(\{x : int\} ρ ▷ int)
  • then-branch: □(\{x : int\} ρ' ▷ int)
  • else-branch: □(ρ'' ▷ int)

• let x = ‘y in ‘(,x + w); ‘((,x 1) + z)
  x: \forall \alpha \forall \rho. □(\{y : \alpha\} \rho ▷ \alpha)
  • first x: □(\{y : int, w : int\} ρ' ▷ int)
  • second x: □(\{y : int → int, z : int\} ρ'' ▷ int → int)
Polymorphic Type System (3/4)

typing judgment

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

(TBOX)

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

(TUNBOX)

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

(TEVAL)

\[ \frac{\Delta_0 \cdots \Delta_n \vdash e : \square (\emptyset \triangleright A)}{\Delta_0 \cdots \Delta_n \vdash \text{eval } e : A} \]
(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) \[ \frac{\Delta_0 \ldots \Delta_n \vdash e_1 : A \rightarrow B \quad \Delta_0 \ldots \Delta_n \vdash e_2 : A}{\Delta_0 \ldots \Delta_n \vdash e_1 e_2 : B} \]

(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} \]

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

(TLETAPP) \[ \frac{\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} \]
Type Inference Algorithm

Unification:
- Rémy’s unification for record type \( \Gamma \)
- usual unification for new type terms such as \( \Box (\Gamma \triangleright A) \) and \( A \text{ ref} \)

Type inference algorithm:
- the same structure as top-down version \( M \) [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 $\emptyset; \emptyset \vdash e : S\alpha$.

**Complete**

If $\emptyset; \emptyset \vdash e : R\alpha$ then $\text{infer}(\emptyset, e, \alpha) = S$ and $R = TS$ for some $T$. 

Kwangkeun Yi

A Polymorphic Type System for Multi-Staged Languages
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.
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.