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

Kwangkeun Yi

SNU 4541.780, Fall 2006

Lecture 1

Outline

1. Introduction and Challenge
2. Contribution and Ideas
3. Simple Type System
4. Polymorphic Type System
5. Conclusion

Multi-Staged Programming (1/2)

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

Stage	Computation	Value
0	usual + code + eval	usual + code
>0	code substitution	code

Multi-Staged Programming Examples (1/2)

In examples, we will use Lisp-style staging constructs + only 2 stages

$$
\begin{array}{rcll}
e & ::= & \cdots & \\
 & \mid & `\, e & \text{code as data} \\
 & \mid & ,\, e & \text{code substitution} \\
 & \mid & \texttt{eval}\ e & \text{execute code}
\end{array}
$$

Multi-Staged Programming Examples (1/2)

In examples, we will use Lisp-style staging constructs + only 2 stages

$$
\begin{array}{rcll}
e & ::= & \cdots & \\
 & | & `\, e & \text{code as data} \\
 & | & ,\, e & \text{code substitution} \\
 & | & \texttt{eval}\; e & \text{execute code}
\end{array}
$$

Code as data

```
let NULL = `0
let body = `(if e = ,NULL then abort() ...)
in eval body
```

Specializer/Partial evaluator

```
power(x,n) = if n=0 then 1 else x * power(x,n-1)
```

v.s.
```
power(x,3) = x*x*x
```

prepared as

```
let spower(n) = if n=0 then `1 else `(x*,(spower (n-1)))
let fastpower10 = eval `(λx.,(spower 10))
in fastpower10 2
```

Review: Practice of Multi-Staged Programming

Features of Lisp/Scheme's quasi-quotation system

- open code

```
`(x+1)
```

Features of Lisp/Scheme's quasi-quotation system

Review: Practice of Multi-Staged Programming

- open code

```
`(x+1)
```

- intentional variable-capturing substitution at stages > 0

$$\texttt{`(}\lambda\texttt{x.,(spower 10))}$$

Features of Lisp/Scheme's quasi-quotation system

Review: Practice of Multi-Staged Programming

- open code

```
`(x+1)
```

- intentional variable-capturing substitution at stages > 0

$$\texttt{`(}\lambda\texttt{x.,(spower 10))}$$

- capture-avoiding substitution

$$\texttt{`(}\lambda^*\texttt{x.,(spower 10) + x)}$$

Features of Lisp/Scheme's quasi-quotation system

Review: Practice of Multi-Staged Programming

- open code

```
`(x+1)
```

- intentional variable-capturing substitution at stages > 0

$$\texttt{`(}\lambda\texttt{x.,(spower 10))}$$

- capture-avoiding substitution

$$\texttt{`(}\lambda^{*}\texttt{x.,(spower 10) + x)}$$

- imperative operations with open code

$$\texttt{cell := `(x+1);}\ \cdots\ \texttt{cell := `(y 1);}$$

Features of Lisp/Scheme's quasi-quotation system

Challenge

A static type system that supports the practice.

Should allow programmers both

- type safety and
- the expressivenss 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	(8)	alpha equiv. at stage $n+1$

Our system	$+1$ $+2$ $+3$ $+4$ $+5$ $+6$ $+7$ -8
[Rhiger 2005]	$+1$ $+2$ $+3$ -4 $+5$ -6 -7 -8
[Calcagno et al. 2004]	$+1$ $+2$ -3 $+4$ -5 $+6$ $+7$ $+8$
[Ancona & Moggi 2004]	$+1$ $+2$ $+3$ -4 -5 $+6$ -7 $+8$
[Taha & Nielson 2003]	$+1$ $+2$ -3 -4 -5 $+6$ $+7$ $+8$
[Chen & Xi 2003]	$+1$ $+2$ $+3$ -4 $+5$ -6 $+7$ -8
[Nanevsky & Pfenning 2002]	$+1$ $+2$ $+3$ -4 -5 $+6$ -7 $+8$
MetaML/Ocaml[2000,2001]	$+1$ $+2$ -3 $+4$ -5 $+6$ $+7$ $+8$
[Davies 1996]	-1 $+2$ -3 -4 -5 $+6$ -7 $+8$
[Davies & Pfenning 1996,2001]	$+1$ -2 $+3$ $+4$ -5 $+6$ -7 $+8$

Ideas

- code's type: parameterized by its expected context

$$\Box(\Gamma \triangleright \mathit{int})$$

- view the type environment Γ as a record type

$$\Gamma = \{x : \mathit{int},\ y : \mathit{int} \rightarrow \mathit{int}, \cdots\}$$

- 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 Γ in code type $\Box(\Gamma \triangleright \mathit{int})$
  - monomorphic store types

$\boxed{\text{Natural ideas worked.}}$

Multi-Staged Language

$$
\begin{array}{lllll}
e & ::= & c \mid x \mid \lambda x.e \mid e\ e & & \\
 & \mid & \texttt{box}\ e & \text{code as data} & `\ e \\
 & \mid & \texttt{unbox}_k\ e & \text{code substitution} & ,\ldots,e \\
 & \mid & \texttt{eval}\ e & \text{execute code} & \\
 & \mid & \lambda^* x.e & \text{gensym} & \\
 & \mid & \ldots & &
\end{array}
$$

Evaluation

$$\mathcal{E} \vdash e \xrightarrow{n} r$$

where

$\mathcal{E}$: value environment

n: a stage number

r: a value or `err`

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

(EBOX) $$\frac{\mathcal{E} \vdash e \xrightarrow{n+1} v}{\mathcal{E} \vdash \texttt{box}\ e \xrightarrow{n} \texttt{box}\ v}$$

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

(EEVAL) $$\frac{\mathcal{E} \vdash e \xrightarrow{0} \texttt{box}\ v \quad \mathcal{E} \vdash v \xrightarrow{0} v'}{\mathcal{E} \vdash \texttt{eval}\ e \xrightarrow{0} v'}$$

Simple Type System (1/2)

$$\textit{Type} \quad A, B \quad ::= \quad \iota \mid A \to B \mid \Box(\Gamma \triangleright A)$$

code type

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

typing judgment

$$\Gamma_0 \cdots \Gamma_n \vdash e : A$$

Simple Type System (1/2)

$$\textit{Type} \quad A, B \quad ::= \quad \iota \mid A \to B \mid \Box(\Gamma \triangleright A)$$

code type

$$\texttt{‘(x+1):} \quad \Box(\{x : \texttt{int}, \cdots\} \triangleright \texttt{int})$$

typing judgment

$$\Gamma_0 \cdots \Gamma_n \vdash e : A$$

(TSBOX) $$\frac{\Gamma_0 \cdots \Gamma_n \Gamma \vdash e : A}{\Gamma_0 \cdots \Gamma_n \vdash \texttt{box}\, e : \Box(\Gamma \triangleright A)}$$

(TSUNBOX) $$\frac{\Gamma_0 \cdots \Gamma_n \vdash e : \Box(\Gamma_{n+k} \triangleright A)}{\Gamma_0 \cdots \Gamma_n \cdots \Gamma_{n+k} \vdash \texttt{unbox}_k e : A}$$

(TSEVAL) $$\frac{\Gamma_0 \cdots \Gamma_n \vdash e : \Box(\varnothing \triangleright A)}{\Gamma_0 \cdots \Gamma_n \vdash \texttt{eval}\ \ e : A}$$ (for alpha-equiv. at stage 0)

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

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

(TSAPP) $$\frac{\Gamma_0 \cdots \Gamma_n \vdash e_1 : A \to B \qquad \Gamma_0 \cdots \Gamma_n \vdash e_2 : A}{\Gamma_0 \cdots \Gamma_n \vdash e_1 e_2 : B}$$

Polymorphic Type System (1/4)

A combination of

- ML's let-polymorphism
  - syntactic value restriction + multi-staged "$\text{expansive}^n(e)$"
  - $\text{expansive}^n(e) =$ *False*

    $\Longrightarrow$ e never expands the store during its eval. at $\forall$stages$\leq n$

e.g.)	$`(\lambda x.,e)$	:	can be expansive
	$`(\lambda x.\texttt{eval}\ y)$	:	unexpansive

- Rémy's record types [Rémy 1993]
  - type environments as record types with field addition
  - record subtyping + record polymorphism

Polymorphic Type System (2/4)

- `if` e `then ‘(x+1) else ‘1`: $\boxed{\Box(\{x : \mathit{int}\}\rho \rhd \mathit{int})}$
  - then-branch: $\Box(\{x : int\}\rho' \rhd \mathit{int})$
  - else-branch: $\Box(\rho'' \rhd \mathit{int})$
- `let x = ‘y in ‘(,x + w); ‘((,x 1) + z)`

  $\boxed{\texttt{x}\text{: } \forall\alpha\forall\rho.\Box(\{y : \alpha\}\rho \rhd \alpha)}$
  - first x: $\Box(\{y : \mathit{int},\ w : \mathit{int}\}\rho' \rhd \mathit{int})$
  - second x: $\Box(\{y : \mathit{int} \to \mathit{int},\ z : \mathit{int}\}\rho'' \rhd \mathit{int} \to \mathit{int})$

typing judgment

$$\Delta_0 \cdots \Delta_n \vdash e : A$$

(TBOX)
$$\frac{\Delta_0 \cdots \Delta_n \Gamma \vdash e : A}{\Delta_0 \cdots \Delta_n \vdash \texttt{box}\, e : \Box(\Gamma \triangleright A)}$$

(TUNBOX)
$$\frac{\Delta_0 \cdots \Delta_n \vdash e : \Box(\Gamma \triangleright A) \quad \Delta_{n+k} \succ \Gamma \quad k > 0}{\Delta_0 \cdots \Delta_n \cdots \Delta_{n+k} \vdash \texttt{unbox}_k\ e : A}$$

(TEVAL)
$$\frac{\Delta_0 \cdots \Delta_n \vdash e : \Box(\varnothing \triangleright A)}{\Delta_0 \cdots \Delta_n \vdash \texttt{eval}\ \ e : A}$$

Polymorphic Type System (4/4)

(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) $$\frac{\Delta_0 \cdots \Delta_n \vdash e_1 : A \to B \qquad \Delta_0 \cdots \Delta_n \vdash e_2 : A}{\Delta_0 \cdots \Delta_n \vdash e_1 e_2 : B}$$

(TLETIMP) $$\frac{\begin{array}{l}\text{expansive}^n(e_1) \\ \Delta_0 \cdots \Delta_n \vdash e_1 : A \qquad \Delta_0 \cdots \Delta_n + x : A \vdash e_2 : B\end{array}}{\Delta_0 \cdots \Delta_n \vdash \texttt{let}\ (x\ e_1)\ e_2 : B}$$

(TLETAPP) $$\frac{\begin{array}{l}\neg\, \text{expansive}^n(e_1) \\ \Delta_0 \cdots \Delta_n \vdash e_1 : A \\ \Delta_0 \cdots \Delta_n + x : \mathit{GEN}_A(\Delta_0 \cdots \Delta_n) \vdash e_2 : B\end{array}}{\Delta_0 \cdots \Delta_n \vdash \texttt{let}\ (x\ e_1)\ e_2 : B}$$

Type Inference Algorithm

- Unification:
  - Rémy's unification for record type Γ
  - usual unification for new type terms such as $\Box(\Gamma \triangleright A)$ and $A\,\mathsf{ref}$
- 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 Γ
  - 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 $\texttt{infer}(\emptyset, e, \alpha) = S$ then $\emptyset; \emptyset \vdash e : S\alpha$.

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

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 sketchs, and embedding relations in the paper; full proofs in the technical report.

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 sketchs, and embedding relations in the paper; full proofs in the technical report.

$\boxed{\text{Staged programming "practice" has a sound static type system.}}$

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 sketchs, and embedding relations in the paper; full proofs in the technical report.

$\boxed{\text{Staged programming ``practice'' has a sound static type system.}}$

Thank you.