

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

Ik-Soon Kim¹ Kwangkeun Yi¹ Cristiano Calcagno²

¹Seoul National University ²Imperial 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

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

e	::=	...
	' e	code as data
	, e	code substitution
	eval e	execute code

Multi-Staged Programming Examples (1/2)

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

$$e ::= \dots$$

	' e	code as data
	, e	code substitution
	eval e	execute code

Code as data

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

Multi-Staged Programming Examples (2/2)

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
```

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

Review: Practice of Multi-Staged Programming

- open code

`'(x+1)`

- intentional variable-capturing substitution at stages > 0

`'(λx. , (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

`'(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

`'(x+1)`

- intentional variable-capturing substitution at stages > 0

`'(λx., (spower 10))`

- capture-avoiding substitution

`'(λ*x., (spower 10) + x)`

- imperative operations with open code

`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

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

Our system

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

- code's type: parameterized by its expected context

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

- view the type environment Γ as a record type

$$\Gamma = \{x : \text{int}, y : \text{int} \rightarrow \text{int}, \dots\}$$

- 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 $\Box(\Gamma \triangleright \text{int})$
 - monomorphic store types

Natural ideas worked.

Multi-Staged Language

$e ::=$	$c \mid x \mid \lambda x.e \mid e e$		
	$\text{box } e$	code as data	' e
	$\text{unbox}_k e$	code substitution	, ..., e
	$\text{eval } e$	execute code	
	$\lambda^* x.e$	gensym	
	...		

Evaluation

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

where

\mathcal{E} : value environment

n : a stage number

r : a value or err

Operational Semantics (stage $n \geq 0$)

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

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

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

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

Simple Type System (1/2)

Type $A, B ::= \iota \mid A \rightarrow B \mid \square(\Gamma \triangleright A)$

code type

$\text{'(x+1): } \square(\{x : \mathit{int}, \dots\} \triangleright \mathit{int})$

typing judgment

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

Simple Type System (1/2)

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

code type

$\text{'(x+1): } \Box(\{x : \text{int}, \dots\} \triangleright \text{int})$

typing judgment

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

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

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

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

Simple Type System (2/2)

$$\text{(TSCON)} \quad \Gamma_0 \cdots \Gamma_n \vdash c : \iota$$

$$\text{(TSVAR)} \quad \frac{\Gamma_n(x) = A}{\Gamma_0 \cdots \Gamma_n \vdash x : A}$$

$$\text{(TSABS)} \quad \frac{\Gamma_0 \cdots (\Gamma_n + x : A) \vdash e : B}{\Gamma_0 \cdots \Gamma_n \vdash \lambda x. e : A \rightarrow B}$$

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

$$\text{(TSAPP)} \quad \frac{\Gamma_0 \cdots \Gamma_n \vdash e_1 : A \rightarrow B \quad \Gamma_0 \cdots \Gamma_n \vdash e_2 : A}{\Gamma_0 \cdots \Gamma_n \vdash e_1 e_2 : B}$$

Polymorphic Type System (1/4)

A combination of

- ML's let-polymorphism
 - syntactic value restriction + multi-staged “ $\text{expansive}^n(e)$ ”
 - $\text{expansive}^n(e) = \text{False}$
 - $\implies e$ never expands the store during its eval. at $\forall \text{stages} \leq n$

e.g.) $(\lambda x. e) \quad :$ can be expansive
 $(\lambda x. \text{eval } y) \quad :$ 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 $'1$: $\boxed{\square(\{x : int\}\rho \triangleright int)}$
 - then-branch: $\square(\{x : int\}\rho' \triangleright int)$
 - else-branch: $\square(\rho'' \triangleright int)$
- let $x = 'y$ in $'(, x + w)$; $'((, x 1) + z)$
 $\boxed{x: \forall \alpha \forall \rho. \square(\{y : \alpha\}\rho \triangleright \alpha)}$
 - first x : $\square(\{y : int, w : int\}\rho' \triangleright int)$
 - second x : $\square(\{y : int \rightarrow int, z : int\}\rho'' \triangleright int \rightarrow int)$

Polymorphic Type System (3/4)

typing judgment

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

$$\text{(TBOX)} \quad \frac{\Delta_0 \cdots \Delta_n \Gamma \vdash e : A}{\Delta_0 \cdots \Delta_n \vdash \text{box } e : \square(\Gamma \triangleright A)}$$

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

$$\text{(TEVAL)} \quad \frac{\Delta_0 \cdots \Delta_n \vdash e : \square(\emptyset \triangleright A)}{\Delta_0 \cdots \Delta_n \vdash \text{eval } e : A}$$

Polymorphic Type System (4/4)

$$\text{(TVAR)} \quad \frac{\Delta_n(x) \succ A}{\Delta_0 \cdots \Delta_n \vdash x : A}$$

$$\text{(TABS)} \quad \frac{\Delta_0 \cdots (\Delta_n + x : A) \vdash e : B}{\Delta_0 \cdots \Delta_n \vdash \lambda x. e : A \rightarrow B}$$

$$\text{(TAPP)} \quad \frac{\Delta_0 \cdots \Delta_n \vdash e_1 : A \rightarrow B \quad \Delta_0 \cdots \Delta_n \vdash e_2 : A}{\Delta_0 \cdots \Delta_n \vdash e_1 e_2 : B}$$

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

$$\text{(TLETAPP)} \quad \frac{\neg \text{expansive}^n(e_1) \quad \Delta_0 \cdots \Delta_n \vdash e_1 : A \quad \Delta_0 \cdots \Delta_n + x : \mathbf{GEN}_A(\Delta_0 \cdots \Delta_n) \vdash e_2 : B}{\Delta_0 \cdots \Delta_n \vdash \mathbf{let} (x e_1) e_2 : B}$$

Type Inference Algorithm

- Unification:
 - Rémy's unification for record type Γ
 - usual unification for new type terms such as $\square(\Gamma \triangleright A)$ 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

Type Inference Algorithm

- Unification:
 - Rémy's unification for record type Γ
 - 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 .

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.

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.

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.