Meta-programming with Names and Necessity

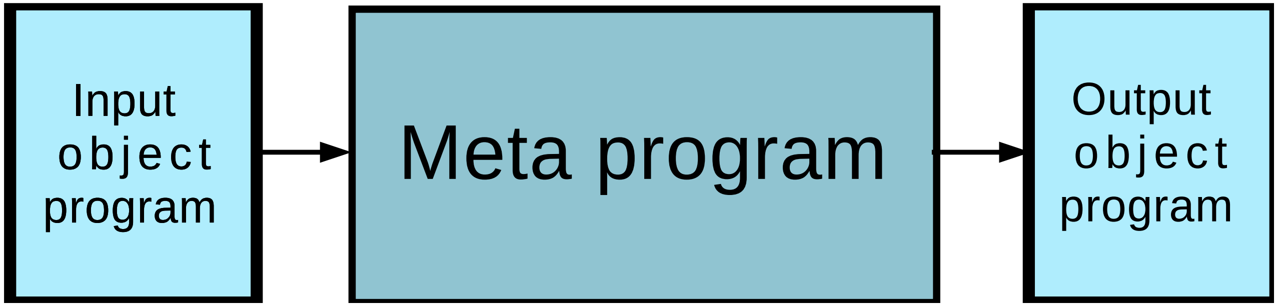
Aleksandar Nanevski

Carnegie Mellon University

ICFP, Pittsburgh, 05 October 2002

Meta-programming

- Manipulation of (source) programs of an object language



- Examples: compilers, partial evaluators, symbolic computation systems, meta-logical frameworks . . .

Typed meta-programming

- Typed meta- and object-language
- Well-typed meta-programs can construct only well-typed source object-language programs
  - Source object-language programs $\longleftrightarrow$ "higher-order" syntax trees
- Object-language types $\subseteq$ Meta-language types
- Here, we name the inclusion as a modal type constructor

$$\Box : \text{object-language types} \to \text{meta-language types}$$

- Example: $\Box A \longleftrightarrow$ type of (source) object-language programs of type A

Representation of source programs

- Must handle programs with *binding structure*
  - built-in notion of equivalence modulo α-renaming variables
- Enable type-safe evaluation of *closed* object-language programs.
- Admit programs with *free variables* (as already noticed by MetaML community).
- Provide a way to destruct source object-language programs and recurse over their structure! (and this is why we need extra expressiveness over MetaML).

Outline

- Introduction ✓
- Background on S4-necessity
- Combining necessity with names
- Theorems
- Future work and conclusions

$\lambda^{\Box}$-calculus

- Proof-term calculus for *necessity* fragment of intuitionistic modal S4 (Pfenning and Davies '00)
- Types
$$A ::= b \mid A_1 \to A_2 \mid \Box A$$
- $\Box A \longleftrightarrow$ values of this type encode *closed source* (i.e. syntactic) expressions of type A
- Typing judgment
$$\Delta; \Gamma \vdash e : A$$
- Two kinds of variables:
  - context Γ for ordinary variables (binding compiled code)
  - context Δ for expression variables (binding source expressions)

$\lambda^{\Box}$-calculus (cont'd)

- Terms

$$e ::= c \mid x \mid \lambda x{:}A.\ e \mid e_1\ e_2 \mid \mathbf{box}\ e \mid \mathbf{let\ box}\ u = e_1\ \mathbf{in}\ e_2$$

- **box** behaves like **quote** in Lisp
- Local reduction

$$\mathbf{let\ box}\ u = \mathbf{box}\ e_1\ \mathbf{in}\ e_2 \longrightarrow [e_1/u]e_2$$

Example 1

- $sum(n)$ produces *source* expression $1 + 2 + \cdots + n$

```
- fun sum (n : int) : □int =
      if n = 1 then (box 1) else
        let box u = sum (n − 1)
            box m = lift n in box (u + m) end;

- val S = sum 5;
val S = box (1 + 2 + 3 + 4 + 5)              (* syntax *)
```

- S can be pattern-matched against and/or evaluated:

```
- let box u = S in u;
val it = 15
```

Necessity limitations

- How to manipulate expressions with binding structure?
- Code analysis restricted
  - subterms of a closed term are not necessarily closed
- Allowing only closed expressions $\longrightarrow$ output expressions will contain unnecessary redexes
- Need a type of *open syntactic expressions* or *code schemas*

Outline

- Introduction ✓
- Background on S4-necessity ✓
- Combining necessity with names
- Theorems
- Future work and conclusions

Code schemas

- Syntactic expressions with "indeterminates" (also called "atoms", "symbols" or "names")
- Treatment of indeterminates (names) inspired by Nominal Logic and FreshML (Pitts and Gabbay '01)
- Names occurring in a boxed syntactic expression are listed in its type

  $\Box(A[\vec{X}]) \longleftrightarrow$ closed syntactic expressions of type A with indeterminates $\vec{X}$
- Example: assuming $X, Y : int$ are names, then

$$\textbf{box}\ (X^3 + 3X^2Y + 3XY^2 + Y^3) : \Box(int[X, Y])$$

Support of a term

- Support of a term $\longleftrightarrow$ set of names which should be defined before the term can be evaluated
- Example: assuming $X, Y : int$ are names, then

term	type	support
$X^2 + Y^2$	int	$\{X, Y\}$
$\mathbf{box}\ (X^2 + Y^2)$	$\Box(int\,[X, Y])$	$\emptyset$
$\langle X^2, \mathbf{box}\ Y^2 \rangle$	$int \times \Box(int[Y])$	$\{X\}$

- Support of a term can be arbitrarily extended

Typing code schemas

- Types $A ::= b \mid A_1 \to A_2 \mid \Box(A[\vec{X}])$
- Typing judgment

$$\Sigma; \Delta; \Gamma \vdash e : A\,[\vec{X}]$$

- $\vec{X}$ is the support of e, and $\vec{X} \subseteq \Sigma$
- Context Γ for ordinary variables
- Context Δ for expression variables with their support
- Context Σ for names

Typing code schemas (cont'd)

- $\Box$-Introduction rule

$$\frac{\Delta; \cdot \vdash e : A}{\Delta; \Gamma \vdash \mathbf{box}\ e : \Box A}$$

- $\Box$-Elimination rule

$$\frac{\Delta; \Gamma \vdash e_1 : \Box A \qquad (\Delta, u{:}A); \Gamma \vdash e_2 : B}{\Delta; \Gamma \vdash \mathbf{let\ box}\ u = e_1\ \mathbf{in}\ e_2 : B}$$

Typing code schemas (cont'd)

- $\Box$-Introduction rule

$$\frac{\Sigma; \Delta; \cdot \vdash e : A\,[\vec{X}]}{\Sigma; \Delta; \Gamma \vdash \mathbf{box}\, e : \Box(A[\vec{X}])}$$

- $\Box$-Elimination rule

$$\frac{\Sigma; \Delta; \Gamma \vdash e_1 : \Box(A[\vec{X}])\,[\vec{Y}] \qquad \Sigma; (\Delta, u{:}A[\vec{X}]); \Gamma \vdash e_2 : B\,[\vec{Y}]}{\Sigma; \Delta; \Gamma \vdash \mathbf{let\ box}\, u = e_1\, \mathbf{in}\, e_2 : B\,[\vec{Y}]}$$

Typing code schemas (cont'd)

- $\Box$-Introduction rule

$$\frac{\Sigma; \Delta; \cdot \vdash e : A\,[\vec{X}] \qquad \vec{Y} \subseteq \mathbf{dom}(\Sigma)}{\Sigma; \Delta; \Gamma \vdash \mathbf{box}\; e : \Box(A[\vec{X}])\;[\vec{Y}]}$$

- $\Box$-Elimination rule

$$\frac{\Sigma; \Delta; \Gamma \vdash e_1 : \Box(A[\vec{X}])\;[\vec{Y}] \qquad \Sigma; (\Delta, u{:}A[\vec{X}]); \Gamma \vdash e_2 : B\,[\vec{Y}]}{\Sigma; \Delta; \Gamma \vdash \mathbf{let\ box}\; u = e_1 \;\mathbf{in}\; e_2 : B\,[\vec{Y}]}$$

Typing code schemas (cont'd)

- Terms $e ::= \cdots \mid X \mid \ldots$
- Name rule

$$\frac{\vec{Y} \subseteq \mathbf{dom}(\Sigma)}{(\Sigma, X{:}A); \Delta; \Gamma \vdash X : A\,[X, \vec{Y}]}$$

Explicit name substitution

- Terms $e ::= \ldots \mid \{X \doteq e_1\}\, e_2 \mid \ldots$
- Example

  - let box u = box $(X^2 + 2XY + Y^2)$
    in
    box $(\{Y \doteq 2\}\, u)$
    end

  - val it = box $(X^2 + 2X * 2 + 2^2)$: $\Box(int[X])$

- Notice: the term constructor $\{X \doteq e_1\}\, e_2$ does not bind X

Example 2

- Given n, generate the function $\lambda x.\ \underbrace{x * \cdots * x * 1}_{n}$

  \- fun $exp\ (n : int) : \Box(int[X])$ =
  if $n = 0$ then box 1 else
  let box u = $exp\ (n-1)$ in box ($X * u$) end

  \- val $poly$ = $exp\ 2$;
  val $poly$ = box ($X * X * 1$)

  \- let box u = $poly$ in box $(\lambda x.\ \{X \doteq x\}\ u)$ end;
  val it = box$(\lambda x.\ x * x * 1)$

Name creation

- Dynamic introduction of names into computation (version of gensym)
- Terms $e ::= \ldots \mid \mathbf{new}\ X{:}A\ \mathbf{in}\ e \mid \ldots$
- Type system ensures the value of e does not depend on X
- Typing rule

$$\frac{(\Sigma, X{:}A); \Delta; \Gamma \vdash e : B\,[\vec{Y}] \qquad\qquad X \not\in B[\vec{Y}]}{\Sigma; \Delta; \Gamma \vdash \mathbf{new}\ X{:}A\ \mathbf{in}\ e : B\,[\vec{Y}]}$$

Name abstraction

- Used to express that a term depends on one name, no matter which (inspired by FreshML and Nominal Logic of Pitts and Gabbay)
- Terms $e ::= \cdots \mid X . e \mid \ldots$ Types $A ::= \cdots \mid \underset{X:A_1}{\text{И}} A_2$
- $X . e$ pairs up X and the value of e into a closure
- Example: polynomial p with one indeterminate

- new $X : int$ in
  let val p = box$(X^2 + 1)$ in
  $X . p$
  end
end
val it = X . box$(X^2 + 1)$: $\underset{Y:int}{\text{И}} \Box(int[Y])$

Name concretion

- Provides a fresh name in place of the abstracted one
- Terms $e ::= \cdots \mid e \mathbin{@} X \mid \ldots$
- Elimination form for abstraction
- Example

  - val p = X . box$(X^2 + Z^2)$: $\underset{X:int}{\text{И}} \Box(int[X, Z])$
  - val q = Y . box$(Y^2 + Z^2)$: $\underset{X:int}{\text{И}} \Box(int[X, Z])$

  - new W : int in
  $p \mathbin{@} W$ = $q \mathbin{@} W$
  end;
  val it = true

- Expressions $p \mathbin{@} Z$ and $q \mathbin{@} Z$ are not be well-typed, as Z is not fresh for p and q.

Example 3

- Given source for $f{:}int \to int$, generate source for f^2
- Use pattern-matching to check if f is a lambda

- fun $square2\ (F : \Box(int \to int))$ =
  case F of
  box $(\lambda x.\ [E \,@\, x]) \Rightarrow \quad (*\ \ E{:}\ \text{И}_{X:int}\ \Box(int[X])\ \ *)$
  new $X : int$ in
  let box u = $E \,@\, X$ in box $(\lambda x.\ \{X \doteq x\}\ (u * u))$
  $|$ box $(E) \Rightarrow$ box $\lambda x.\ (E\ x){*}(E\ x)$

- $square2$ (box $\lambda x.\ x$);
val it = box $(\lambda y.\ y * y)$

- Thanks to pattern-matching, no redexes in the result

Outline

- Introduction ✓
- Background on S4-necessity ✓
- Combining necessity with names ✓
- Theorems
- Future work and conclusions

Substitution principles

1. Ordinary substitution principle
   if $\Sigma; \Delta; \Gamma \vdash e_1 : A\,[C]$ and $\Sigma; \Delta; \Gamma, x{:}A \vdash e_2 : B\,[C]$, then
   $\Sigma; \Delta; \Gamma \vdash [e_1/x]e_2 : B\,[C]$

2. Modal substitution principle
   if $\Sigma; \Delta; \cdot \vdash e_1 : A\,[C]$ and $\Sigma; \Delta, u{:}A[C]; \Gamma \vdash e_2 : B\,[C]$, then
   $\Sigma; \Delta; \Gamma \vdash [e_1/u]e_2 : B\,[C]$

3. Name substitution principle
   if $\Sigma, X{:}A; \Delta; \Gamma \vdash e_1 : A\,[C]$ and
   $\Sigma, X{:}A; \Delta; \Gamma \vdash e_2 : B\,[X, C]$, then
   $\Sigma, X{:}A; \Delta; \Gamma \vdash \{X/e_1\}e_2 : B\,[C]$

Progress and preservation

If $\Sigma; \cdot; \cdot \vdash e : A\,[\,]$ then either

1. e is a value, or
2. there exists $\Sigma' \subseteq \Sigma$, such that $\Sigma, e \longmapsto \Sigma', e'$; furthermore, e' is unique, and $\Sigma'; \cdot; \cdot \vdash e' : A\,[\,]$

Future work

- Support polymorphism can be found in the paper
- Names of general types (currently names are simply typed)
- Type polymorphism and type-polymorphic recursion
- Polymorphic patterns and intensional type analysis
- Relation to MetaML and other meta-programming languages
- Extension to type theory with names

Conclusions

- Type of closed syntactic program representations corresponds to $\Box$ modality of intuitionistic S4.
- Not expressive enough for intensional manipulation of programs with binding structure
- Type of open source programs can be obtained by adding indeterminates (names) to the language, thus creating "polynomials" over source expressions
- Names stand for free variables of source programs making it possible to destruct and analyze the source programs
- The distinction between compiled and source code achieved through the $\Box$ modality allows for *typed* names
- Since names are typed, explicit substitution can be made primitive

Outline

- Introduction ✓
- Background on S4-necessity ✓
- Combining necessity with names ✓
- Theorems ✓
- Future work and conclusions

Related work

- Judgmental reconstruction of modal logic (Pfenning and Davies ’00)
- Nominal logic and FreshML (Pitts and Gabbay ’01)
  - Modeled in Fraenkel-Mostowsky set theory
  - Uses name abstraction to represent α-equivalence classes of terms
  - Only “first-order” syntax
  - Names limited to a type **atm**
    - can be extended to a family of types...
    - ...but still, names can be used only for bindings
  - No distinction between variables and names of type **atm**
  - Substitution must be hand-written
  - Impossible to give substitution-style operational semantics

Related work (cont'd)

- Systems with type of open syntactic expressions
  - Temporal $\lambda^{\bigcirc}$ calculus (Davies '96)
    - object program = meta program at "later time"
    - free object program variables = meta variables at "later time"
    - problems:
      no evaluation of closed expressions
      no attempt at code analysis
  - MetaML (Calcagno, Moggi, Taha, Sheard '01)
    - $\lambda^{\bigcirc}$ + type refinement for closedness
    - problems:
      no code analysis
      scope extrusion in presence of references

Intensional code analysis

- Destructing syntactic expressions (with binding) by pattern-matching
- Higher-order patterns

$$\pi ::= [E\ x_1 \cdots x_n] \mid x \mid \lambda x.\ \pi \mid (\pi_1{:}A_1 \to A_2)\ (\pi_2) \mid \ldots$$

- Pattern $[E\ x_1 \cdots x_n]$ matches a syntactic expression with free variables in the set $\{x_1, \ldots, x_n\}$, and stores it into the pattern variable E

Intensional code analysis (cont'd)

- Pattern typing judgment

$$\Sigma;\Gamma \Vdash \pi : A\,[\vec{Y}] \Longrightarrow \Gamma'$$

- Lambda abstraction rule

$$\frac{\Sigma;(\Gamma, x{:}A_1) \Vdash \pi : A_2\,[\vec{Y}] \Longrightarrow \Gamma'}{\Sigma;\Gamma \Vdash \lambda x{:}A_1.\,\pi : A_1 \to A_2\,[\vec{Y}] \Longrightarrow \Gamma'}$$

- Pattern-variable rule

$$\frac{x_i{:}A_i \in \Gamma \qquad\qquad \vec{Y} \subseteq \mathbf{dom}(\Sigma)}{\Sigma;\Gamma \Vdash [E\;\vec{x}] : A\,[\vec{Y}] \Longrightarrow E{:}\;\underset{a_1:A_1}{\text{И}} \cdots \underset{a_n:A_n}{\text{И}}\;\Box(A[\vec{Y},\vec{a}])}$$

Relationship with S4

- Syntactic expressions can be composed

$$
\begin{aligned}
&apply \equiv \\
&\qquad \lambda x.\ \lambda y.\ \textbf{let box}\ u = x\ \textbf{in let box}\ v = y\ \textbf{in box}\ (u\ v) \\
&\qquad : \Box(A \to B) \to \Box A \to \Box B
\end{aligned}
$$

- Syntactic expressions are syntactic

$$lift \equiv (\lambda x.\ \textbf{let box}\ u = x\ \textbf{in box box}\ u) : \Box A \to \Box\Box A$$

- Syntactic expressions can be compiled and evaluated

$$eval \equiv (\lambda x.\ \textbf{let box}\ u = x\ \textbf{in}\ u) : \Box A \to A$$

Typing abstraction and concretion

- И type constructor is a binder
- Name abstraction rule

$$\frac{(\Sigma, X{:}A); \Delta; \Gamma \vdash e : B\,[\vec{Y}]}{(\Sigma, X{:}A); \Delta; \Gamma \vdash X\,.\,e : (\underset{X'{:}A}{\text{И}}\,[X'/X]B)\,[\vec{Y}]}$$

- Name concretion rule

$$\frac{(\Sigma, X{:}A); \Delta; \Gamma \vdash e : (\underset{X'{:}A}{\text{И}}\,B)\,[\vec{Y}]}{(\Sigma, X{:}A); \Delta; \Gamma \vdash e\,@\,X : ([X/X']B)\,[\vec{Y}]}$$

Example 2

- How to generate syntactic expressions with binding structure?
- Application $exp(n)$ produces source for $\lambda x{:}int.\ x^n$

```
- fun exp (n : int) : □(int → int) =
      if n = 0 then (box λx. 1) else
        let box u = exp (n − 1) in
              box λx. x ∗ u (x)
        end
```

```
- exp 2;

val it = box λx. x ∗ (λy. y ∗ (λz. 1) y) x
```

- But we want $exp\ 2 \hookrightarrow \text{box}(\lambda x{:}int.\ x * x * 1)$!

Example 4

- Given code for $f\!:\!int \rightarrow int$, generate code for f^2
- Attempt with no code analysis

```
- fun square (F : □(int → int)) =
      let box f = F in
          box λx. (f x) * (f x)
      end
```

```
 - square (box λx. x);
val it = box (λy. (λx. x) y * (λx. x) y)
```

- Unnecessary redexes again!

Possible applications

- Distinguishing between extensional and intensional nature of programs
  - algebraic simplifications in symbolic computation
  - functions can exploit knowledge of intensional structure of arguments (examples: integration, differentiation)
  - Higher-order Abstract Syntax
- Programmer-specified (source level) optimizations in run-time code generation
  - mechanism for choosing between highly-optimized or quickly produced target programs
  - domain-specific optimizations

$\lambda^{\Box}$-calculus (cont'd)

- Hypothesis rule

$$\frac{x{:}A \in \Delta \cup \Gamma}{\Delta; \Gamma \vdash x{:}A}$$

- Local reduction

$$\textbf{let box } u = \textbf{box } e_1 \textbf{ in } e_2 \longrightarrow [e_1/u]e_2$$

- Local expansion

$$e \longrightarrow \textbf{let box } u = e \textbf{ in box } u$$

Explicit name substitution (cont'd)

- Substituted name must be in context Σ

- Typing rule

$$\frac{(\Sigma, Y{:}A); \Delta; \Gamma \vdash e_1 : A\,[\vec{X}] \qquad (\Sigma, Y{:}A); \Delta; \Gamma \vdash e_2 : B\,[Y, \vec{X}]}{(\Sigma, Y{:}A); \Delta; \Gamma \vdash \{Y \doteq e_1\}\; e_2 : B\,[\vec{X}]}$$

Typing code schemas (cont'd)

- Terms $e ::= \cdots \mid X \mid \ldots$
- Name rule

$$\frac{\vec{Y} \subseteq \mathbf{dom}(\Sigma)}{(\Sigma, X{:}A); \Delta; \Gamma \vdash X : A\,[X, \vec{Y}]}$$

- Hypotheses rules

$$\frac{}{\Delta; \Gamma, x{:}A \vdash x : A} \qquad \frac{}{\Delta, u{:}A; \Gamma \vdash u : A}$$

Typing code schemas (cont'd)

- Terms $e ::= \cdots \mid X \mid \ldots$
- Name rule

$$\frac{\vec{Y} \subseteq \mathbf{dom}(\Sigma)}{(\Sigma, X{:}A); \Delta; \Gamma \vdash X : A\,[X, \vec{Y}]}$$

- Hypotheses rules

$$\frac{\vec{X} \subseteq \mathbf{dom}(\Sigma)}{\Sigma; \Delta; (\Gamma, x{:}A) \vdash x : A\,[\vec{X}]} \qquad \frac{\vec{X} \subseteq \vec{Y} \subseteq \mathbf{dom}(\Sigma)}{\Sigma; (\Delta, u{:}A[\vec{X}]); \Gamma \vdash u : A\,[\vec{Y}]}$$