Typeful Staged Computations

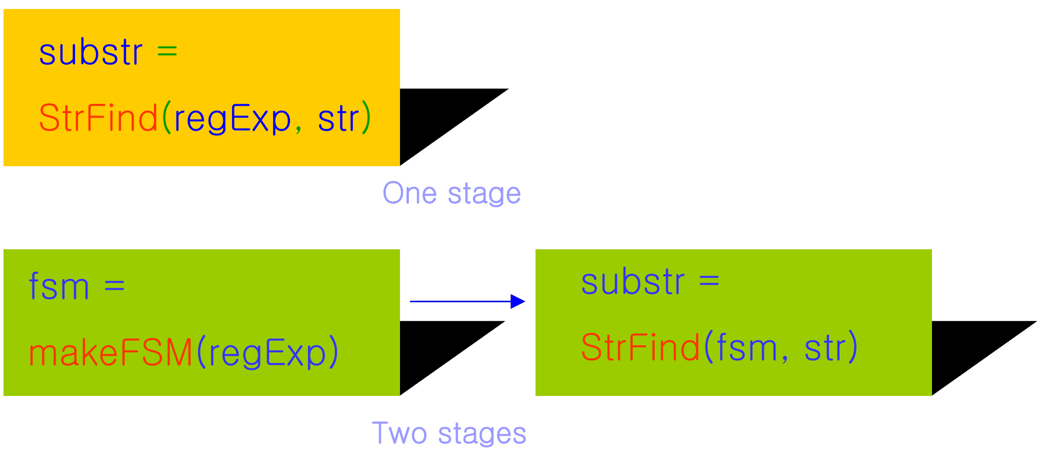
Ik-Soon Kim

LiComR Winter School 2004

Feb 12, 2004

Staged Computations

- Explicit division of a computation into stages.
- A common technique in algorithm design.
- It is concerned with *how* a value is computed



Staged Computation Examples

- Partial Evaluation:
  - Specialization of a program based on partial input data
- Run-Time Code Generation:
  - Dynamic generation of code during the evaluation of a program
  - Gains high efficiency
  - Difficult to locate bugs since code is changeable
- Macro systems
  - Translates input source language into another one
  - Provides a convenient and efficient way to write programs

Language Constructs for Staged Computations

- Explicit annotation of codes

```
fun x -> x + 1        <=>        code( fun x -> x + 1 )
```

- Run-time composition of codes

```
let c = code (fun x -> x + 1)
in  code (fun x -> comp(c) (x) + 2)
=> code (fun x -> (fun x -> x + 1)(x) + 2)
```

- Run-time evaluation of codes

```
let inc = eval( code(fun x -> x + 1) )
in  inc (y)
```

Programming Languages for Staged Computations

- Lisp

```
code        `(lambda (x) (+ x 1))
compose     `(lambda (x) (+ (,y x) 1))
eval        (eval `(lambda x -> x + 1))
```

- `C: an extension of ANSI C

```
code        void  cspec hello= `{printf("Hello");}
compose     void cspec greet = `{@hello;}
eval        compile(greet, void)
```

Programming Languages for Staged Computations

- MetaOCAML

```
code        <fun x -> x + 1>
compose     <fun x -> (~y)(x) + 1>
eval        run (<fun x -> x + 1>)
```

Types in Staged Computations

- In staged computations, programs are no more static ones
- Since programs are changeable, it is more difficult to write safe programs
- Type system is crucial for safe staged computation programs.
- Type systems for previous languages are not satisfactory
  - ‘C is not type safe like C language
  - lisp is a dynamic type language
  - MetaOCAML may raise exceptions during run-time code generation

Modal Types

- Proposed by Davies and Pfenning
- Allows only closed terms as codes

Syntax $e ::= x$
$\quad | \lambda x.e$
$\quad | e_1 e_2$
$\quad | u$
$\quad |$ box e
$\quad |$ let box u = e_1 in e_2

Modal Types

- code

```
box e   where e is a closed term
```

- compose

```
let box u = box (fun x –> x + 1)
in  box (fun x –> u + 1)
```

- eval

```
let box u = box (fun x –> x + 1)
in  u
```

- lift v

```
lift x  => box(10000) if x = 10000
```

Modal Type Example

- evaluate a polynomial for a coefficient list and some value x

```
fun evalPoly (nil, x) = 0
  |   evalPoly (a::p, x) = a + (x * evalPoly(x,p))
```

```
evalPoly( [1, 2, 3], x)
   => 1 + (x * (2 + x * (3 + x * 0)))
```

Modal Type Example

- Specialize a polynomial function:

```
fun specPoly (nil) = box (fun x => 0)
  |  specPoly (a::p) =
      let box f = specPoly p
          box v = lift a
      in  box (fun x => v + (x * f x))
```

```
specPoly( [1, 2, 3] ) =>
    box(fun x => 1+ x * (f2 x))
      f2 = box(fun x => 2 + x * (f3 x))
      f3 = box(fun x => 3 + x * (f4 x))
      f4 = box(fun x => 0)
```

Modal Types

Types	$A, B ::= A \rightarrow B \mid \Box A$
Contexts	$\Gamma, \Delta ::= \Gamma, x : A \mid \Delta, u : A$

- $\Box A$
  - The type of code of type A
  - Related with modal logic S4
  - A is necessarily true in all accessible worlds
  - $\Box A$ in all accessible stages

- Δ … type environment for code variables
- Γ … type environment for value variables

Modal Types

$$\frac{\Gamma(x) = A}{\Delta;\Gamma \vdash x : A} \qquad \frac{\Delta(x) = A}{\Delta;\Gamma \vdash x : A}$$

$$\frac{\Delta;\Gamma, x : A \vdash e : B}{\Delta;\Gamma \vdash \lambda x.e : A \to B} \qquad \frac{\Delta;\Gamma \vdash e_1 : A \to B \quad \Delta;\Gamma \vdash e_2 : A}{\Delta;\Gamma \vdash e_1 e_2 : B}$$

$$\frac{\Delta;\bullet \vdash e : A}{\Delta;\Gamma \vdash \text{box } e : \Box A} \qquad \frac{\Delta;\Gamma \vdash e_1 : \Box A \quad \Delta, u : A;\Gamma \vdash e_2 : B}{\Delta;\Gamma \vdash \text{let box } u = e_1 \text{ in } e_2 : B}$$

Support multi-staged computations:

If $e : \Box A$, e is necessarily A in all accessible stages

let box u = e (* $\Box A$ *)

in box(… u ….box(… u …) …)

Modal Type Examples

(* $\Box(A \to B) \to \Box A \to \Box B$ *)

$\lambda x.\lambda y.$

let box $u = x$ in

let box $v = y$ in

box (u v)

(* eval: $\Box A \to A$ *)

$\lambda x.$ let box $u = x$ in u

(* quote: $\Box A \to \Box\Box A$ *)

$\lambda x.$ let box $u = x$

in box (box u)

Modal Types

- It is a severe restriction to allow only closed terms as codes

```
specPoly( [1, 2, 3] ) =>
    box(fun x => 1+ x * (f2 x))
      f2 = box(fun x => 2 + x * (f3 x))
      f3 = box(fun x => 3 + x * (f4 x))
      f4 = box(fun x => 0)
```

- For improved staged computations, open terms should be allowed as codes

```
specPoly( [1, 2, 3] ) =>

    box(fun x => 1+ x * (2 + x * (3 + x * 0)))
```

Temporal Types

- Proposed by Davies
- Allow restricted open terms in code constructs

Syntax $e ::= c$
$\mid x$
$\mid \lambda x.e$
$\mid e_1 e_2$
$\mid \text{next } e$
$\mid \text{prev } e$

Types $A, B ::= A \to B \mid \bigcirc A$

Contexts $\Gamma ::= \bullet \mid \Gamma, x : A^n$

$\Gamma \vdash^n e : A$ — e has type A at time (stage) n in context Γ

Semantics

$e \to^n v$ e evaulates to v at time (stage) n

$$\lambda x.e \to^0 \lambda x.e \qquad \frac{e_1 \to^0 \lambda x.e'_1 \quad e_2 \to^0 v_2 \quad [v_2/x]e'_1 \to^0 v_3}{e_1 e_2 \to^0 v_3}$$

$$x \to^{n+1} x \qquad \frac{e \to^{n+1} v}{\lambda x.e \to^{n+1} \lambda x.v} \qquad \frac{e_1 \to^{n+1} v_1 \quad e_2 \to^{n+1} v_2}{e_1 e_2 \to^{n+1} v_1 v_2}$$

$$\frac{e \to^{n+2} v}{\text{next } e \to^{n+1} \text{next } v} \qquad \frac{e \to^0 \text{next } v}{\text{prev } e \to^1 v} \qquad \frac{e \to^{n+1} v}{\text{prev } e \to^{n+2} \text{prev } v}$$

Temporal Types

$$\frac{\Gamma(x) = A^n}{\Gamma \vdash^n x : A}$$

$$\frac{\Gamma, x : A^n \vdash^n e : B}{\Gamma \vdash^n \lambda x.e : A \to B} \qquad \frac{\Gamma \vdash^n e_1 : A \to B \quad \Gamma \vdash^n e_2 : A}{\Gamma \vdash^n e_1 e_2 : B}$$

$$\frac{\Gamma \vdash^{n+1} e : A}{\Gamma \vdash^n \text{next } e : \bigcirc A} \qquad \frac{\Gamma \vdash^n e : \bigcirc A}{\Gamma \vdash^{n+1} \text{prev } e : A}$$

prev (next e) $\to e$ next (prev e) $\to e$

Temporal Type Examples

```
fun pow n = next(fun x → prev(
      (fun m  →
         if m=0
         then next(1)
         else next(x  * (prev (pow (m−1)))))
      n))
```

```
fun pow' n =
   if n = 0
   then box(fun x  → 1)
   else let box u = pow (n−1) in
            box(fun x  → x * (u x))
```

$\text{pow } 0 \to \text{next}(\text{fun } x \to 1)$

$\text{pow } 1 \to \text{next}(\text{fun } x \to x * 1)$

$\text{pow } 2 \to \text{next}(\text{fun } x \to x * (x * 1))$

$\text{pow}'\,0 \to \text{box}(\text{fun } x \to 1) = \text{r0}$

$\text{pow}'\,1 \to \text{box}(\text{fun } x \to x * (\text{r0 } x)) = \text{r1}$

$\text{pow}'\,2 \to \text{box}(\text{fun } x \to x * (\text{r1 } x))$

Temporal Types

- Time (or stage) n is some value in linear order

$$\frac{\Gamma \vdash^{n+1} e : A}{\Gamma \vdash^{n} \text{next } e : \bigcirc A} \qquad \frac{\Gamma \vdash^{n} e : \bigcirc A}{\Gamma \vdash^{n+1} \text{prev } e : A}$$

- next time of n is only one stage n+1
- prev time of n+1 is only one n
- Code sharing is very restricted between n time and n+1 time
- Until one closed code is obtained, another closed code can not be written
- eval construct is missing

Environment Classifiers

- Proposed by (explicit) Taha and (implicit) Calcagno
- Expandsion of temporal types
- Linear time is expanded into some name sequence like $\alpha_1, \alpha_2, \ldots, \alpha_n = \Sigma, \alpha_n$ instead of 1,2,…,n
- next (e) => <e> prev(e) => ~e
- run construct is newly appended for eval

$$\frac{\Gamma \vdash^{\Sigma} e : A}{\Gamma \vdash^{\Sigma\alpha} \text{next } e : \langle A \rangle^{\alpha}} \quad \rightarrow \quad \frac{\Gamma \vdash^{\Sigma} e : A}{\Gamma \vdash^{\Sigma\alpha} \langle e \rangle : \langle A \rangle^{\alpha}}$$

$$\frac{\Gamma \vdash^{\Sigma\alpha} e : \langle A \rangle^{\alpha}}{\Gamma \vdash^{\Sigma} \text{prev } e : A} \quad \rightarrow \quad \frac{\Gamma \vdash^{\Sigma\alpha} e : \langle A \rangle^{\alpha}}{\Gamma \vdash^{\Sigma} \sim e : A}$$

Environment Classifiers

$$\frac{\Gamma(x) = A^{\Sigma}}{\Gamma \vdash^{\Sigma} x : A}$$

$$\frac{\Gamma, x : A^{\Sigma} \vdash^{\Sigma} e : B}{\Gamma \vdash^{\Sigma} \lambda x.e : A \rightarrow B} \qquad \frac{\Gamma \vdash^{\Sigma} e_1 : A \rightarrow B \quad \Gamma \vdash^{\Sigma} e_2 : A}{\Gamma \vdash^{\Sigma} e_1 e_2 : B}$$

$$\frac{\Gamma \vdash^{\Sigma} e : A}{\Gamma \vdash^{\Sigma\alpha} \text{next } e : \langle A \rangle^{\alpha}} \qquad \frac{\Gamma \vdash^{\Sigma,\alpha} e : \langle A \rangle^{\alpha}}{\Gamma \vdash^{\Sigma} \text{prev } e : A}$$

$$\frac{\Gamma \vdash^{\Sigma} e : \langle A \rangle^{\alpha} \quad \alpha \notin FV(\Gamma, \Sigma)}{\Gamma \vdash^{\Sigma} \text{run } e : A}$$

Environment Classifiers

- Can express a rather restricted open terms as codes

  <fun x → ~x+1> (good)   <x+1> (wrong)
- In explicit environment classifiers
  - Stage names should be explicitly provided by programmer

    $(\alpha)e$ or $(\alpha_1)(\ldots(\alpha_2)e\ldots)$
  - Support polymorphic type system
  - Principal type inference algorithm does not exist

- In implicit environment classifiers
  - Support polymorphic type system
  - Type inference algorithm
  - Stage names are automatically inference by type inference algorithm

Temporal Types and Environment Classifiers

- Type systems do not support imperative features

```
val a = ref <1>
val b = <fun x  →  ~(a:=<x>;<2>);
val c=!a
    c is <x>, and it is a type error !!
```

Conclusions

- Staged computation is a common and necessary technique
- Type system is crucial for safe staged computations
- For more convenient and efficient manipulation of codes, general open terms are required in staged computations

- Type system is require to
  - Express general open terms
  - Support polymorphic types
  - Support imperative features
  - Support the type inference algorithm