A lexically scoped type system for multi-stage languages

Morten Rhiger

Roskilde University, Denmark

IFIP WG 2.11 meeting, June 25, 2012

This talk: $\lambda^{[]}$

A type system for multi-stage calculus, which

- is hygienic,
- evaluates under λs,
- supports `run`,
- supports mutable state, and
- defines one new type of code, with one introduction rule and one elimination rule.

Approach

- A context-aware code type.

  (Records where code values are built.)

- Lexically scoped terms and types.

  (Controls where code values can be be used.)

Outline

- Monomorphic $\lambda^{[]}$
- Subtyping
- Polymorphism
- Standard examples and pitfalls

Syntax of $\lambda^{[]}$

$$e \;::=\; b \mid x \mid \lambda x{:}t\,.\,e \mid e\;e \mid \underbrace{\uparrow e \mid \downarrow e}$$

quasiquote&unquote or bracket&escape

$$t \;::=\; B \mid t \to t \mid [t]t \mid \underbrace{x \mid t, t \mid \emptyset}$$

Sets of variables (a distinguished kind)

The code type of $\lambda^{[]}$

$[\ x_1, \cdots, x_n\ ]\ t$ — type of code fragment

variables allowed free in code fragment

The code type of $\lambda^{[]}$

An intuition

From Contextual Modal Types

$$[x_1 : t_1, \cdots, x_n : t_n]t$$

Kim et al (2005), Rhiger (2005), Nanevski et al (2008)

to lexically scoped modal types:

$$[x_1 : t_1, \cdots, x_n : t_n]t$$

The code type of $\lambda^{[]}$

An intuition

From Contextual Modal Types

$$[x_1 : t_1, \cdots, x_n : t_n]t$$

Kim et al (2005), Rhiger (2005), Nanevski et al (2008)

to lexically scoped modal types:

$$\begin{array}{c} \lambda x_1 : t_1. \cdots \lambda x_n : t_n. \\ \vdots \\ [x_1 : t_1, \cdots, x_n : t_n]t \end{array}$$

The code type of $\lambda^{[]}$

An intuition

From Contextual Modal Types

$$[x_1 : t_1, \cdots, x_n : t_n]t$$

Kim et al (2005), Rhiger (2005), Nanevski et al (2008)

to lexically scoped modal types:

$$\begin{array}{l} \lambda x_1 : t_1. \cdots \lambda x_n : t_n. \\ \qquad\qquad \vdots \\ [x_1 \quad , \cdots , x_n \quad]t \end{array}$$

The code type of $\lambda^{[]}$

Another intuition (yet unexplored)

$$[\underbrace{x_1, \cdots, x_n}]t$$

A classifier?

Taha&Nielsen (2003)

Hygiene = Lexical scope

when evaluating under λs

Hygiene enforces abstractions:

$$\text{let } f(c) = \uparrow(\lambda x.\downarrow c)$$
$$\text{in } \uparrow(\lambda x.\downarrow(f\,(\uparrow x))) \quad \longmapsto \quad \uparrow(\lambda x_1.\,\lambda x_2.\,x_1)$$

Enforcing lexical scope

- In semantics (well known):

  Variable convention (Barendregt): α-convert (i.e., consistently rename) when necessary, to make bound and free variable differ.

- In type systems ($\lambda^{[]}$):

  Ditto

Contexts of $\lambda^{[]}$

Several namespaces, one for each stage:

$$\underbrace{\gamma_0, \cdots, \gamma_{n-1}}_{\text{past stages}}, \underbrace{\gamma_n}_{\text{present stage}} ; \underbrace{\gamma_{n+1}, \cdots, \gamma_m}_{\text{future stages}} \vdash \cdots$$

$$\gamma \quad \in \quad \text{term-variables} \rightarrow_{\text{fin}} \text{types}$$

$$\Gamma \quad ::= \quad \gamma, \cdots, \gamma$$

Typing $\lambda^{[]}$

$$\boxed{\Gamma\ ;\ \Gamma' \vdash e : t}$$

$$\frac{\gamma(x) = t}{\Gamma, \gamma\ ;\ \Gamma' \vdash x : t} \quad \text{(Var)}$$

$$\frac{\Gamma, \gamma\ ;\ \Gamma' \vdash t' :: * \quad \Gamma, \gamma + (x : t')\ ;\ \Gamma' \vdash e : t}{\Gamma, \gamma\ ;\ \Gamma' \vdash \lambda x{:}t'.\, e : t' \to t} \quad (\to \text{I})$$

$$\frac{\Gamma\ ;\ \Gamma' \vdash e_1 : t_2 \to t \quad \Gamma\ ;\ \Gamma' \vdash e_2 : t_2}{\Gamma\ ;\ \Gamma' \vdash e_1\ e_2 : t} \quad (\to \text{E})$$

$$\vdots$$

Typing $\lambda^{[\,]}$ $\quad \boxed{\Gamma\,;\,\Gamma' \vdash e : t}$

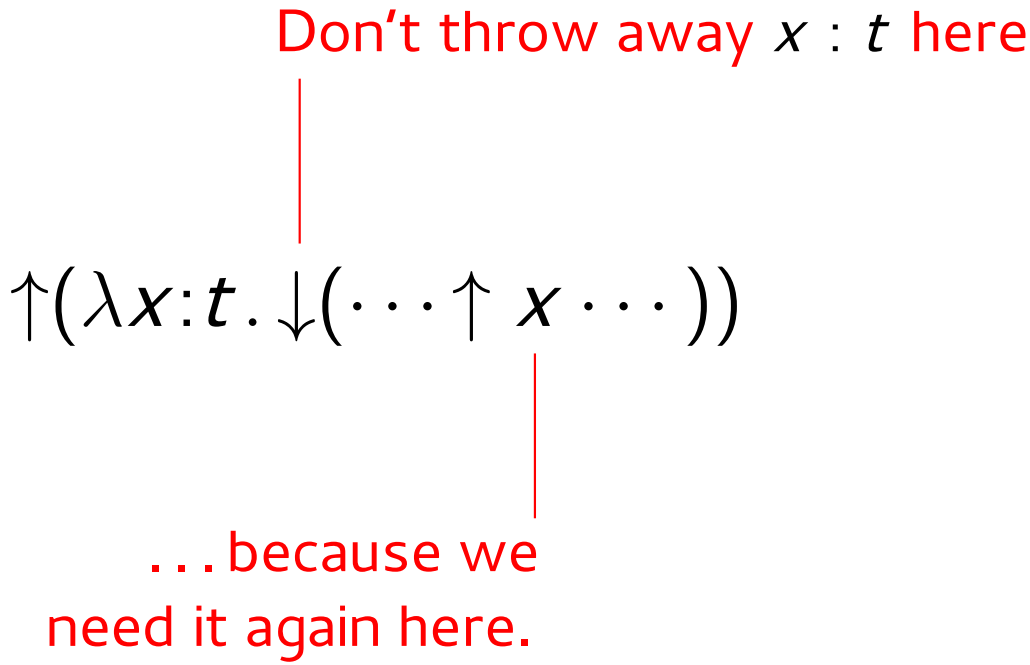
$$\vdots$$

$$\frac{\Gamma\,,\,\gamma\,;\quad \Gamma' \vdash\quad e :\quad t}{\Gamma\quad\;;\gamma\,,\,\Gamma' \vdash \uparrow e : [\overline{\gamma}]t} \qquad ([\,]\ \mathrm{I})$$

$$\frac{\Gamma\quad\;;\gamma\,,\,\Gamma' \vdash\quad e : [\overline{\gamma}]t}{\Gamma\,,\,\gamma\,;\quad \Gamma' \vdash \downarrow e :\quad t} \qquad ([\,]\ \mathrm{E})$$

where $\overline{\gamma} = \mathrm{dom}(\gamma)$, as a type.

Enforcing lexical scope



Kinding $\lambda^{[]}$

$$\boxed{\Gamma \; ; \; \Gamma' \vdash t :: \kappa}$$

Kinding guarantees that types are

- lexically scoped,
- correctly staged, and
- well formed.

Kinding $\lambda^{[]}$

$$\boxed{\Gamma\ ;\ \Gamma' \vdash t :: \kappa}$$

Kinding guarantees that types are

- lexically scoped,
- correctly staged, and
- well formed.

$$\kappa ::= * \mid \text{env}$$

Kinding $\lambda^{[]}$

$$\boxed{\Gamma\ ;\ \Gamma' \vdash t :: \kappa}$$

$$\frac{\Gamma\ ;\ \Gamma' \vdash t_1 :: *\quad \Gamma\ ;\ \Gamma' \vdash t_2 :: *}{\Gamma\ ;\ \Gamma' \vdash t_1 \to t_2 :: *}\ (\mathsf{K}\to)$$

$$\vdots$$

Kinding $\lambda^{[]}$

$$\boxed{\Gamma\ ;\ \Gamma' \vdash t :: \kappa}$$

$$\vdots$$

$$\frac{x \in \mathsf{dom}(\gamma)}{\Gamma, \gamma\ ;\ \Gamma' \vdash x :: \mathsf{env}} \quad \text{(K-Var)}$$

$$\frac{\Gamma\ ;\ \Gamma' \vdash t_1 :: \mathsf{env} \quad \Gamma\ ;\ \Gamma' \vdash t_2 :: \mathsf{env}}{\Gamma\ ;\ \Gamma' \vdash t_1, t_2 :: \mathsf{env}} \quad \text{(K-Union)}$$

$$\frac{}{\Gamma, \gamma\ ;\ \Gamma' \vdash \emptyset :: \mathsf{env}} \quad \text{(K } \emptyset\text{)}$$

$$\vdots$$

Kinding $\lambda^{[]}$

$$\boxed{\Gamma\ ;\ \Gamma' \vdash t :: \kappa}$$

$$\vdots$$

$$\frac{\Gamma, \gamma\ ;\ \Gamma' \vdash t_1 :: \mathsf{env} \quad \Gamma, \gamma\ ;\ \Gamma' \vdash t_2 :: *}{\Gamma\ ;\ \gamma, \Gamma' \vdash [t_1]\, t_2 :: *} \qquad (\mathsf{K}\ [])$$

Examples:
Scope extrusion

Avoiding scope extrusion (1)

```
let c : []int ref = ref (↑2)
in  ↑(λx:int. ··· ↓(c := ↑x) ···)
```

is correctly rejected, since

```
  c  :  []int ref
↑x   :  [x]int
```

Avoiding scope extrusion (2)

```
let c : [x]int ref = ref (↑2)
in  ↑(λx:int. ··· ↓(c := ↑x) ···)
```

is correctly rejected, since

the first x is unbound.

Avoiding scope extrusion (3)

```
↑(λx:int.
    ↓(let c : [x]int ref = ref (↑2)
      in ↑(λx:int. ··· ↓(c := ↑x) ···)))
```

is correctly rejected, since

```
  c  :  [x]int ref
↑x   :  [x]int
```

and x and x are different variables.

Subtyping

Subtyping relation

$$\vdots$$

$$t, t' \leq t$$

$$\frac{t_2 \leq t_1 \quad t_1' \leq t_2'}{[t_1]t_1' \leq [t_2]t_2'}$$

($[t]t'$ is contravariant in t.)

Subsumption

$$\frac{\Gamma\,;\Gamma' \vdash e : t_2 \quad t_2 \leq t_1}{\Gamma\,;\Gamma' \vdash e : t_1} \qquad (t\text{-}\leq)$$

$$\frac{\Gamma_2\,;\Gamma'_2 \vdash e : t \quad \Gamma_1\,;\Gamma'_1 \leq \Gamma_2\,;\Gamma'_2}{\Gamma_1\,;\Gamma'_1 \vdash e : t} \qquad (\Gamma\text{-}\leq)$$

Examples: Subtyping

Subtyping example (1)

To type

$$\texttt{let c : []int = } \uparrow \texttt{1 in } \uparrow(\lambda \texttt{x:bool. } \downarrow \texttt{c})$$

coerce $\downarrow$c from `[]int` to `[x]int`.

Subtyping example (2)

To type

$$\uparrow(\lambda \texttt{x:bool.}\ \downarrow(\ \cdots\ \texttt{run}(\uparrow 2)\ \cdots\ ))$$

remove the binding

$$\texttt{x:bool}$$

from the context at $\uparrow 2$.

Passing open code across a lambda

```
↑(λx:int.
  ↓(let c : [x]int = ref (↑1)
    in  ↑(λy:bool.
            ··· ↓( c := ↑x; ··· ))))
```

is accepted (and is safe).

Polymorphism

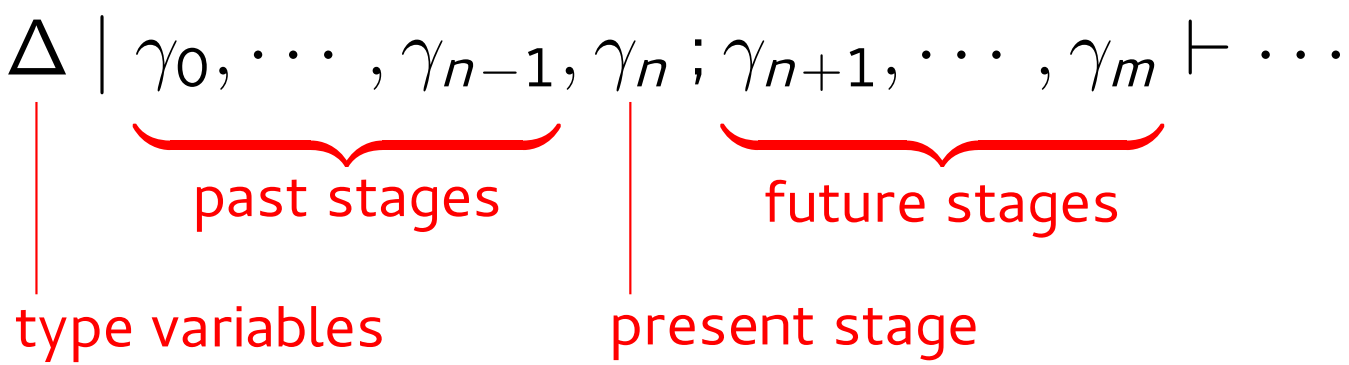
Polymorphic extension

$$e \;::=\; \cdots \mid \Lambda T :: \kappa.\, e \mid e\,[t]$$

$$t \;::=\; \cdots \mid T \mid \forall T :: \kappa.\, t$$

Extending contexts

One namespace of type variables:



$$\Delta \in \text{type-variables} \rightarrow_{\text{fin}} \text{kinds}$$

Extending kinding

$$\boxed{\Delta \mid \Gamma \,;\, \Gamma' \vdash t :: \kappa}$$

$$\vdots$$

$$\frac{\Delta(T) = \kappa}{\Delta \mid \Gamma \,;\, \Gamma' \vdash T :: \kappa} \qquad \text{(K-Tvar)}$$

$$\frac{\Delta + (T :: \kappa) \mid \Gamma \,;\, \Gamma' \vdash t :: *}{\Delta \mid \Gamma \,;\, \Gamma' \vdash \forall T :: \kappa.\, t :: *} \qquad \text{(K } \forall\text{)}$$

Extending typing

$$\boxed{\Delta \mid \Gamma \,;\, \Gamma' \vdash e : t}$$

$$\vdots$$

$$\frac{\Delta + (T :: \kappa) \mid \gamma \,;\, \Gamma' \vdash e : t}{\Delta \mid \gamma \,;\, \Gamma' \vdash \Lambda T :: \kappa.\, e : \forall T :: \kappa.\, t} \quad (\forall\ \mathrm{I})$$

at stage 0 only (in this talk)

$$\frac{\Delta \mid \Gamma \,;\, \Gamma' \vdash e : \forall T' :: \kappa.\, t \quad \Delta \mid \Gamma \,;\, \Gamma' \vdash t :: \kappa}{\Delta \mid \Gamma \,;\, \Gamma' \vdash e\,[t'] : t\{t'/T'\}} \quad (\forall\ \mathrm{E})$$

$$\vdots$$

Extending typing

$$\boxed{\Delta \mid \Gamma \,;\, \Gamma' \vdash e : t}$$

$$\vdots$$

$$\frac{\Delta \mid \Gamma, \gamma \,;\quad \Gamma' \vdash\quad e :\qquad\quad t}{\Delta \mid \Gamma\qquad ;\gamma, \Gamma' \vdash \uparrow e : [\overline{\gamma}, \Delta_{\mathsf{env}}]t} \qquad ([\,]\ \mathrm{I})$$

$$\frac{\Delta \mid \Gamma\qquad ;\gamma, \Gamma' \vdash\quad e : [\overline{\gamma}, \Delta_{\mathsf{env}}]t}{\Delta \mid \Gamma, \gamma \,;\quad \Gamma' \vdash \downarrow e :\qquad\quad t} \qquad ([\,]\ \mathrm{E})$$

where $\Delta_{\mathsf{env}} = \{T \mid \Delta(T) = \mathsf{env}\}$, as a type.

Type variables of kind env

- $(R :: \text{env})$ is a "placeholder" for a set of term variables:

$$\forall R :: \text{env}.\,[x, y, R]\,t \xrightarrow[\text{instantiate}]{} [x, y, a, b]\,t$$

- Any $(R :: \text{env})$ in scope must go into a code type:

$$\lambda R :: \mathit{env}.$$

$$\uparrow(\qquad\qquad)$$

$$\text{type } [R]\,t'$$

Type variables of kind env

- $(R :: \text{env})$ is a "placeholder" for a set of term variables:

$$\forall R :: \text{env}.\,[x, y, R]t \xrightarrow[\text{instantiate}]{} [x, y, a, b]t$$

- Any $(R :: \text{env})$ in scope must go into a code type:

$$\lambda R :: \mathit{env}.$$
$$\lambda x : [R]t.$$
$$\uparrow(\qquad\qquad)$$
$$\text{type } [R]t'$$

Type variables of kind env

- $(R :: \text{env})$ is a "placeholder" for a set of term variables:

$$\forall R :: \text{env}.\, [x, y, R]t \xrightarrow[\text{instantiate}]{} [x, y, a, b]t$$

- Any $(R :: \text{env})$ in scope must go into a code type:

$$\begin{array}{l} \lambda R :: \mathit{env}. \\ \quad \lambda x : [R]t. \\ \quad\quad \uparrow(\cdots \downarrow x \cdots) \end{array}$$

type $[R]t'$

Example: Polymorphism

Staged eta expansion

```
eta = ΛT :: *. ΛU :: *. ΛR :: env.
        λf : ∀S :: env. [R,S]T → [R,S]U.
          ↑(λx : T.
            ↓(f [x] (↑x)))

eta : ∀T :: *. ∀U :: *. ∀R :: env.
        (∀S :: env. [R, S]T → [R, S]U) →
          [R](T → U)
```

Using the staged eta expansion

```
powbta n [S::env] (x:[S]int) =
  if n=0 then
    ↑1
  else
    ↑(↓x * ↓(f (n-1) [S] x))

powgen n = eta [int] [int] [0] (powbta n)
```

Conclusions

$\lambda^{[]}$ demonstrates that

- α-equivalence **is compatible** with context-aware code types (see Taha&Nielsen, POPL'03),
- code types **are lightweight**,
- there is **no need for additional technologies** to handle side effects (see $\lambda_1^{\oslash}$) or `run`,
- there is **no need to restrict evaluation under dynamic λs**, e.g. by preventing it, or by replacing it by substitution (see $\lambda^{\square}$, CMT),
- standard type-system machinery (environments, subtyping, polymorphism) are sufficient.

Conclusions

$\lambda^{[]}$ demonstrates that

- α-equivalence **is compatible** with context-aware code types (see Taha&Nielsen, POPL'03),
- code types **are lightweight**,
- there is **no need for additional technologies** to handle side effects (see $\lambda_1^{\oslash}$) or `run`,
- there is **no need to restrict evaluation under dynamic λs**, e.g. by preventing it, or by replacing it by substitution (see $\lambda^{\square}$, CMT),
- standard type-system machinery (environments, subtyping, polymorphism) are sufficient.

Thanks