

# A Simple Take on Typed Abstract Syntax in Haskell-like Languages

Morten Rhiger

BRICS

University of Aarhus, Denmark

Joint work with Olivier Danvy

FLOPS 2001

Tokyo, March 9, 2001

## Our work

1. We implement typed abstract syntax for  $\lambda$ -terms in Haskell so that only well-typed  $\lambda$ -terms are accepted by Haskell.
2. We apply the result to normalization by evaluation (a program transformation) showing that it preserves types and yields normal forms.

## Motivation: Representing $\lambda$ -terms in Haskell

```
data Term = VAR String
          | LAM String Term
          | APP Term Term
```

## Motivation: untyped $\lambda$ -terms

Untyped  $\lambda$ -terms can be represented in Haskell: For example

```
LAM "x" (APP (VAR "x") (VAR "x"))
```

represents

$$\lambda x. x x$$

## Goals and Means

**Goal:** Typed abstract syntax. (Making e.g.

```
LAM "x" (APP (VAR "x") (VAR "x"))
```

untyped in Haskell.)

**Means:**

1. Higher-order abstract syntax
2. “Phantom types”

# Plan

## **Part I:** Typed abstract syntax

- Higher-order abstract syntax.
- “Phantom types”.

## **Part II:** Application to normalization by evaluation

## **Conclusion**

## Higher-order abstract syntax

```
data Term = LAM (Term → Term)
          | APP Term Term
```

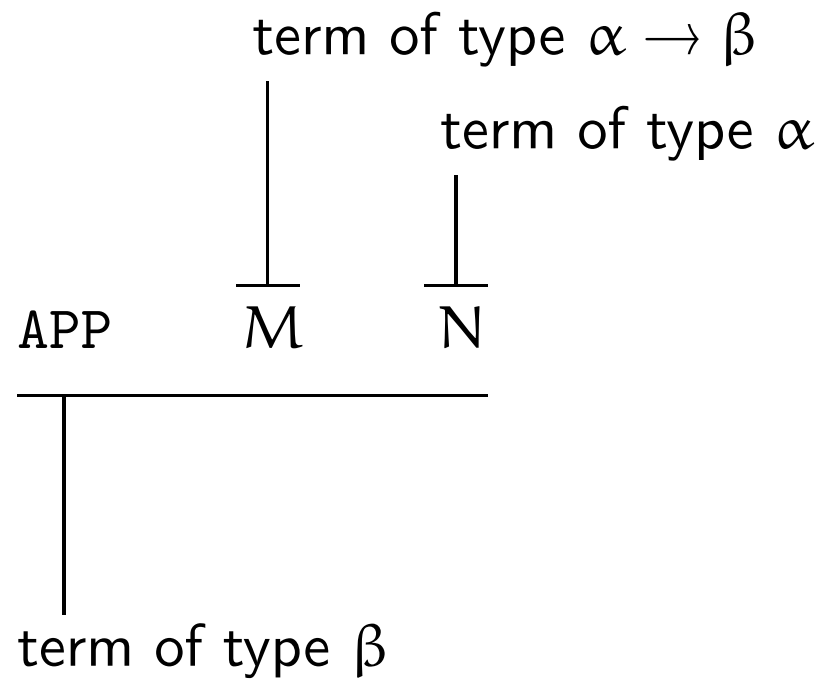
Higher-order abstract syntax implements  $\lambda$ -bindings with Haskell-bindings.

## Types of higher-order syntax constructors

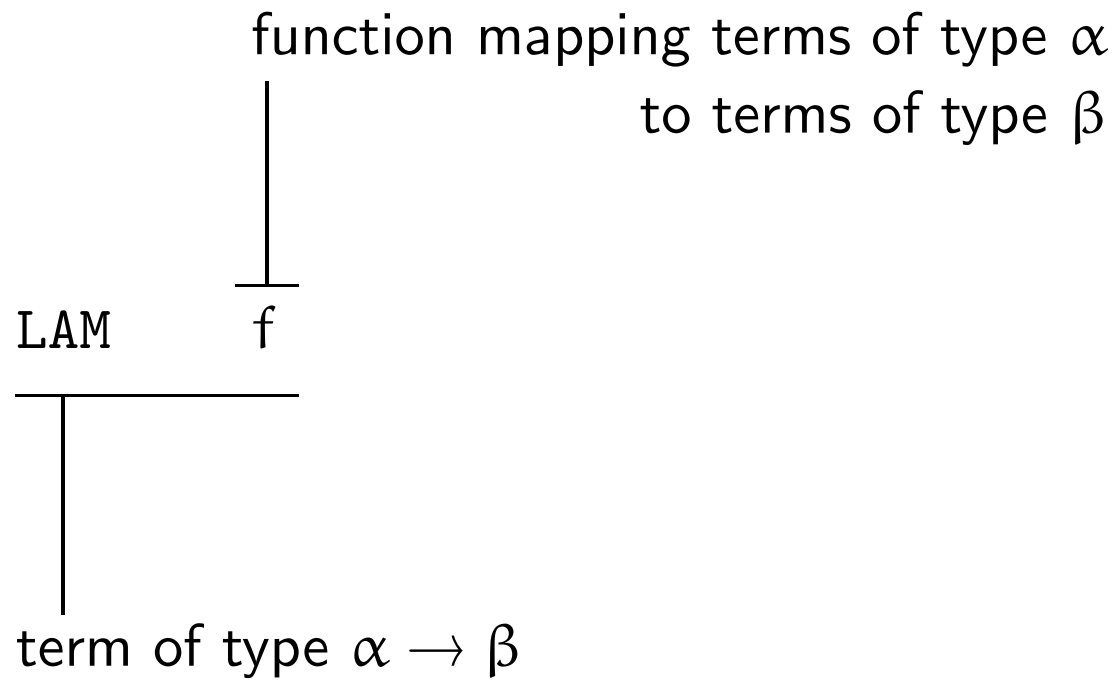
APP :: Term  $\rightarrow$  (Term  $\rightarrow$  Term)

LAM :: (Term  $\rightarrow$  Term)  $\rightarrow$  Term

# Observation 1



## Observation 2



## Phantom types

We carry the type of the represented term ( $\alpha$ ) in the type of its representation (Exp):

`type Exp  $\alpha$  = Term`

$\alpha$  is a “phantom type” (Finne et al., ICFP 1999).

Phantom types are also used to typecheck foreign function interface (Leijen & Meijer, USENIX 1999).

## Restricted types of higher-order syntax constructors

`app` :: `Exp (α → β) → (Exp α → Exp β)`

`lam` :: `(Exp α → Exp β) → Exp (α → β)`

Hiding the underlying representation:

```
newtype Exp a = EXP Term
```

```
app (EXP f) (EXP a) = EXP (APP f a)
```

```
lam f = EXP (LAM (\x ->
```

```
    let EXP t = f (EXP x) in t))
```

## Example of typed abstract syntax

- Haskell rejects this expression:

```
lam (\x -> app x x)
```

- Haskell infers the type  $\text{Exp}(\alpha \rightarrow \alpha)$  for this expression:

```
lam (\x -> x)
```

## Known problems with higher-order abstract syntax

1. The Haskell function space is too big (nontermination).
2. No fold function for higher-order abstract syntax.

These problems are irrelevant here:

1. We only consider simply typed, terminating programs.
2. We only produce (i.e., output) representations of terms.

## Our working hypothesis

We assume that for any  $\lambda$ -term of type  $t$ ,  
Haskell infers that its representation has type `Exp t`.

## Part II: Normalization by evaluation

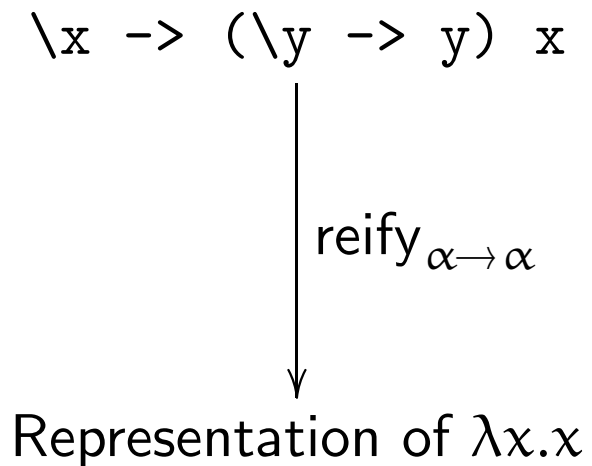
Simply-typed Haskell value  $v$  of type  $t$

$\text{reify}_t$



Representation of the normal form of  $v$

## Example of normalization by evaluation



(See, e.g., Danvy's paper at FLOPS 1998.)

# Application #1: Normalization by evaluation preserves types

Goal: To implement normalization by evaluation in Haskell such that

$$\text{reify}_t \quad :: \quad t \rightarrow \text{Exp } t$$

## Implementing normalization by evaluation

Two functions, reify and reflect, defined by structural induction on types:

$$t ::= b \mid t \rightarrow t$$

In Haskell: Overloading is implemented using type classes.

## Type-class declarations

```
class Reify a
  where reify  :: a -> Exp a
```

```
class Reflect a
  where reflect :: Exp a -> a
```

## Induction step

```
instance (Reflect a, Reify b) => Reify (a -> b)
  where reify v = lam (\x -> reify (v (reflect x)))
```

```
instance (Reify a, Reflect b) => Reflect (a -> b)
  where reflect e = \x -> reflect (app e (reify x))
```

Matching the canonical definition of normalization by evaluation:

$$\downarrow^{s \rightarrow t} v = \underline{\lambda}x. \downarrow^t (v @ (\uparrow_s x))$$

$$\uparrow_{s \rightarrow t} e = \lambda x. \uparrow_t (e @ (\underline{\downarrow}^s x))$$

## Induction base

In normalization by evaluation:

$$\begin{aligned} \text{reify}_\alpha &= \textit{the identity function} \\ \text{reflect}_\alpha &= \textit{the identity function} \end{aligned}$$

But

$$\begin{aligned} \text{reify}_\alpha &:: \alpha \rightarrow \text{Exp } \alpha \\ \text{reflect}_\alpha &:: \text{Exp } \alpha \rightarrow \alpha \end{aligned}$$

## Solution: Coerce and uncoerce

Hidden representation:

$$\begin{aligned}\text{Exp } \alpha &= \text{Term} \\ \text{Exp } (\text{Exp } \alpha) &= \text{Term}\end{aligned}$$

“Phantom” identity functions:

$$\begin{aligned}\text{coerce} &:: \text{Exp } (\text{Exp } \alpha) \rightarrow \text{Exp } \alpha \\ \text{uncoerce} &:: \text{Exp } \alpha \rightarrow \text{Exp } (\text{Exp } \alpha)\end{aligned}$$

## Induction base (continued)

```
instance Reify (Exp a)
  where reify e = uncoerce e
```

```
instance Reflect (Exp a)
  where reflect e = coerce e
```

Matching the definition of normalization by evaluation:

$$\begin{aligned}\downarrow^b v &= v \\ \uparrow_b e &= e\end{aligned}$$

using the phantom identity functions.

**Result #1:**  
**Normalization by evaluation preserves types**

Using the Haskell type inferencer as a theorem prover:

$\text{reify} \quad :: \quad \text{Reify } \alpha \Rightarrow \alpha \rightarrow \text{Exp } \alpha$

## **Application #2: Normalization by evaluation yields normal forms**

A long  $\beta\eta$ -normal form

1. contains no  $\beta$ -redexes, and
2. is fully  $\eta$ -expanded with respect to its type

## Long $\beta\eta$ -normal forms

Typing rules

- Normal forms:  $\Delta \vdash_{\text{nf}} e :: t$

$$\frac{\Delta, x :: t_1 \vdash_{\text{nf}} e :: t_2}{\Delta \vdash_{\text{nf}} (\lambda x :: t_1. e) :: t_1 \rightarrow t_2} \qquad \frac{\Delta \vdash_{\text{at}} e :: b}{\Delta \vdash_{\text{nf}} e :: b}$$

- Atomic forms:  $\Delta \vdash_{\text{at}} e :: t$

$$\frac{\Delta \vdash_{\text{at}} e_0 :: t_1 \rightarrow t_2 \quad \Delta \vdash_{\text{nf}} e_1 :: t_1}{\Delta \vdash_{\text{at}} (e_0 e_1) :: t_2} \qquad \frac{\Delta(x) = t}{\Delta \vdash_{\text{at}} x :: t}$$

## Data type of normal forms

```
data Nf = COERCE At
        | LAM (At → Nf)
data At = APP At Nf
```

We take the same steps as before:

- Introduce phantom-typed syntax constructors.
- Hide their representation.
- Use them to implement normalization by evaluation.

**Result #2:**  
**Normalization by evaluation yield normal forms**

Using the Haskell type inferencer as a theorem prover:

$$\text{reify} \quad :: \quad \text{Reify } \alpha \Rightarrow \alpha \rightarrow \text{Nf}' \alpha$$

## Conclusion

1. Our simple take on typed abstract syntax relies on (1) higher-order abstract syntax and (2) phantom types.
2. Using this typed abstract syntax we have shown that normalization by evaluation
  - preserves types, and
  - yields long  $\beta\eta$ -normal forms.

Thank you

## Larger example of normalization by evaluation

Given

$$s = \lambda f g x \rightarrow f x (g x)$$

$$k = \lambda x y \rightarrow x$$

what is the normal form of  $(s k)$  which has type

$$T = ((\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \alpha)$$

We have the following

$$\begin{array}{c} (s k) \\ \downarrow \text{reify}_T \\ \text{Representation of } \lambda x. \lambda y. y \end{array}$$