

# Υλοποίηση ενός διερμηνέα για Λάμβδα Λογισμό

Ζωή Παρασκευοπούλου  
Νίκος Γιανναράκης

Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών  
Εθνικό Μετσόβιο Πολυτεχνείο

13 Νοεμβρίου 2013

# The Syntax

$e := e_1 e_2$   
| “ $\lambda$ ”  $id$  “.”  $e$   
| “*let*” [“*rec*”]  $id$  “=”  $e_1$  “*in*”  $e_2$   
| “[”  $e_1$  “,”  $e_2$  “]”  
|  $id$   
| “*true*” | “*false*”  
| “*if*”  $e_1$  “*then*”  $e_2$  “*else*”  $e_3$   
|  $e_1 op e_2$   
|  $e_1 rop e_2$   
|  $e_1 bop e_2$

$op :=$  “+” | “−”  
| “\*”  
| “\*\*”  
 $rop :=$  “=” | “<”  
| “<=” | “>”  
| “>=”  
 $bop :=$  “&&”  
| “||”

# Type System

The language is strongly typed featuring the Hindley-Milner type system. The types are implicit in the source (à la Curry) and they are automatically reconstructed using the algorithm  $W$  for type inference.

# Hindley-Milner typing

Hindley-Milner type system is a restriction of system  $F$ , featuring let polymorphism. Unlike system  $F$ , in which type reconstruction is undecidable, the types can be inferred using the algorithm  $W$ .

**Significant Limitation:** Let-polymorphism is rank-1 polymorphism, that means that functions cannot take as arguments polymorphic functions.

# Hindley-Milner typing

## Examples

```
> ./jebus annot
let const = \x. \y. x in
[const 1 true, const false 42]
-----
let const : a1 -> a2 -> a1 =
  \x. : a1. \y. : a6 . x
in
  [const 1 true, const false 42]
```

Figure 1 : Here *const* has type  
 $\forall a. \forall b. (a \rightarrow b \rightarrow a)$

```
> ./jebus annot
let id = \x. x in
let f = \g. [g 1, g true] in
  f id
Could not match type Nat with
type Bool
```

Figure 2 : *g*'s type cannot be a  
polymorphic function!

# Hindley-Milner typing

## Types

We will use  $\tau$  for simple types,  $\sigma$  for type schemes and  $\alpha$  for type variables.

$$\tau := \tau_1 \rightarrow \tau_2$$

$$| \tau_1 \times \tau_2$$

$$| \textit{Nat}$$

$$| \textit{Bool}$$

$$| \alpha$$

$$\sigma := \forall \alpha. \sigma_1$$

$$| \tau$$

$$\alpha := \alpha_1 | \alpha_2 | \dots$$

# Hindley-Milner typing

## Typing Rules

$$\frac{}{\Gamma, x : \sigma \vdash x : \sigma} \text{var}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x. e : \tau_1 \rightarrow \tau_2} \lambda$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 e_2 : \tau_2} @$$

$$\frac{\Gamma \vdash e_1 : \sigma \quad \Gamma, x : \sigma \vdash e_2 : \tau}{\Gamma \vdash \text{let } [rec] \ x = e_1 \text{ in } e_2 : \tau} \text{let}$$

$$\frac{\Gamma \vdash e : \sigma \quad \alpha \notin FV(\Gamma)}{\Gamma \vdash e : \forall a. \sigma} \text{gen}$$

$$\frac{\Gamma \vdash e : \forall a. \sigma}{\Gamma \vdash e : \sigma[\alpha \rightarrow \tau]} \text{inst}$$

$$\frac{\Gamma \vdash e : \text{Bool} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \text{if } e \text{ then } e_2 \text{ else } e_3 : \tau} \text{ite}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash [e_1, e_2] : \tau_1 \times \tau_2} \text{pair}$$

# Hindley-Milner typing

## Typing Rules (cont.)

$$\frac{\Gamma \vdash e_1 : Nat \quad \Gamma \vdash e_2 : Nat}{\Gamma \vdash e_1 \diamond e_2 : Nat \quad \diamond \in \{+, -, *, /, **\}}^{op}$$

$$\frac{\Gamma \vdash e_1 : Nat \quad \Gamma \vdash e_2 : Nat}{\Gamma \vdash e_1 \diamond e_2 : Bool \quad \diamond \in \{<, <=, ==, >, >=\}}^{rop}$$

$$\frac{\Gamma \vdash e_1 : Bool \quad \Gamma \vdash e_2 : Bool}{\Gamma \vdash e_1 \diamond e_2 : Bool \quad \diamond \in \{\&\&, ||\}}^{bop}$$

$$\frac{\Gamma \vdash e : Bool}{\Gamma \vdash not\ e : Bool}^{not}$$



# The Core Language

The internal representation is actually a pretty small language. Most of the language's expressions are defined as syntactic sugar

$$\begin{aligned} e &:= e_1 \ e_2 \\ &| \ \lambda \ id . e \\ &| \ \textit{Fix} \ e_1 \end{aligned}$$

# The Core Language

## Syntactic Sugar

- **Integers** The integers are represented internally with church encoding

$$n \equiv \lambda s. \lambda z. \underbrace{s(s \dots (s z) \dots)}_{n \text{ times}}$$

- **Arithmetical Operations**

$$e_1 + e_2 \equiv (\lambda x. \lambda y. x \text{ succ } y) e_1 e_2$$

$$e_1 - e_2 \equiv (\lambda x. \lambda y. y \text{ pred } x) e_1 e_2$$

$$e_1 * e_2 \equiv (\lambda x. \lambda y. \lambda z. x y z) e_1 e_2$$

$$e_1 ** e_2 \equiv (\lambda x. \lambda y. y x) e_1 e_2$$

# The Core Language

## Syntactic Sugar

- ▶ **Boolean Constants**

$true \equiv \lambda x. \lambda y. x$        $false \equiv \lambda x. \lambda y. y$

- ▶ **Pairs**  $[e_1, e_2] \equiv \lambda x. x e_1 e_2$

- ▶ **Provided functions for pairs**

$fst \equiv \lambda x. x \ true$  with type  $\forall a. \forall b. a \times b \rightarrow a$

$snd \equiv \lambda x. x \ false$  with type  $\forall a. \forall b. a \times b \rightarrow b$

- ▶ **Provided functions for Integers**

$succ \equiv \lambda x. \lambda s. \lambda z. s \ (n \ s) \ z$  with type  $Nat \rightarrow Nat$

$iszero \equiv \lambda x. x \ (true \ false) \ true$  with type  $Nat \rightarrow Bool$

$pred \equiv \lambda x. snd \ (x \ next \ [0, 0])$  with type  $Nat \rightarrow Nat$

where  $next \equiv \lambda x. [succ \ (fst \ x), \ (fst \ x)]$

# The Core Language

## Syntactic Sugar

### ► Boolean Operators

$not \equiv \lambda x. x \text{ false } true$

$e_1 \&\& e_2 \equiv (\lambda x. \lambda y. x \text{ y false}) e_1 e_2$

$e_1 || e_2 \equiv (\lambda x. \lambda y. x \text{ true } y) e_1 e_2$

### ► Relative Operators

$e_1 \leq e_2 \equiv (\lambda x. \lambda y. \text{iszero } (n \text{ pred } m)) e_1 e_2$

$e_1 < e_2 \equiv (\lambda x. \lambda y. \text{not } (y \text{ leq } x)) e_1 e_2$

$e_1 == e_2 \equiv (\lambda x. \lambda y. (y \text{ leq } x) \&\& (x \text{ leq } y)) e_1 e_2$

$e_1 \geq e_2 == e_2 \leq e_1$

$e_1 > e_2 == e_2 < e_1$

# The Core Language

## Syntactic Sugar

- ▶ Let Definitions

$$\text{let } x = e_1 \text{ in } e_2 \equiv (\lambda x. e_2) e_1$$

- ▶ Let rec is more tricky

$$\text{let rec } x = e_1 \text{ in } e_2 \equiv (\lambda x. e_2) (Y (\lambda x. e_1))$$

remember that  $Y \equiv \lambda f. (\lambda x. f(x x)) (\lambda x. f(x x))$

Alternatively, we can add a new construct to simulate  $Y$ 's behavior:  $\text{let rec } x = e_1 \text{ in } e_2 \equiv (\lambda x. e_2) (\mathbf{Fix} (\lambda x. e_1))$

In both cases  $e_1$  is allowed to refer to  $x$ . The difference is that, unlike  $Y$ ,  $\mathbf{Fix}$  can be typed with the following rule:

$$\frac{\Gamma \vdash e : \tau \rightarrow \tau}{\Gamma \vdash \mathbf{Fix} e : \tau} \text{fix}$$

# The Core Language

## Evaluation Strategies

Currently Jebus supports two different evaluation strategies: normal order and applicative order, with the former being a non-strict evaluation strategy and the later a strict one.

In general:

- ▶ **Normal Order** The leftmost outermost redex is always reduced first
- ▶ **Applicative Order** The leftmost innermost redex is always reduced first

Both strategies evaluate the body of an unapplied function.

# Evaluation Strategies

## Normal Order

The normal order reduction will always produce a normal form, if one exists!

$$\overline{(\lambda x. e_1) e_2 \rightarrow e_1[e_2/x]}$$

$$\frac{e_1 \rightarrow e'_1}{e_1 e_2 \rightarrow e'_1 e_2}$$

$$\frac{e \rightarrow e'}{v e \rightarrow v e'}$$

$$\frac{e \rightarrow e'}{\lambda x. e \rightarrow \lambda x. e'}$$

# Evaluation Strategies

## Applicative Order

Applicative order reduction is not normalizing!

$$\overline{(\lambda x. v_1) v_2 \rightarrow v_1[v_2/x]}$$

$$\frac{e_1 \rightarrow e'_1}{e_1 e_2 \rightarrow e'_1 e_2}$$

$$\frac{e \rightarrow e'}{v e \rightarrow v e'}$$

$$\frac{e \rightarrow e'}{\lambda x. e \rightarrow \lambda x. e'}$$



# Evaluation Strategies

## Semantics for fix

We can think fix as function that takes a function and computes its fixed point.

$$\overline{(fix \lambda x. e) \rightarrow e[fix \lambda x. e/x]}$$

$$\frac{e \rightarrow e'}{fix\ e \rightarrow fix\ e'}$$

Note that  $e[fix \lambda x. e/x] \equiv_{\beta} (\lambda x. e) (fix \lambda x. e) \equiv fix \lambda x. e$ , just like  $f(Y f) \equiv Y f$ .

# Evaluation Strategies

## Fix: Example

```
let rec fact = λ x. if iszero x then 1 else x * fact (x - 1) in fact 3
→(λ fact. fact 3) (fix (λ fact.λ x. if iszero x then 1 else x * fact (x - 1)))
→(fix (λ fact.λ x. if iszero x then 1 else x * fact (x - 1))) 3
→(λ x. if iszero x then 1 else x * (fix (λ fact.λ x. if iszero x then 1 else x * fact (x - 1))) (x - 1)) 3
→if iszero 3 then 1 else 3 * (fix (λ fact.λ x. if iszero x then 1 else x * fact (x - 1))) 2
→3 * (fix (λ fact.λ x. if iszero x then 1 else x * fact (x - 1))) 2
→3 * (λ x. if iszero x then 1 else x * (fix (λ fact.λ x. if iszero x then 1 else x * fact (x - 1))) (x - 1)) 2
→3 * if iszero 2 then 1 else 2 * (fix (λ fact.λ x. if iszero x then 1 else x * fact (x - 1))) 1
→3 * 2 * (fix (λ fact.λ x. if iszero x then 1 else x * fact (x - 1))) 1
→3 * 2 * (λ x. if iszero x then 1 else x * (fix (λ fact.λ x. if iszero x then 1 else x * fact (x - 1))) (x - 1)) 1
→3 * 2 * if iszero 1 then 1 else 1 * (fix (λ fact.λ x. if iszero x then 1 else x * fact (x - 1))) 0
→3 * 2 * 1 * (fix (λ fact.λ x. if iszero x then 1 else x * fact (x - 1))) 0
→3 * 2 * 1 * (λ x. if iszero x then 1 else x * (fix (λ fact.λ x. if iszero x then 1 else x * fact (x - 1))) (x - 1)) 0
→3 * 2 * 1 * if iszero 0 then 1 else 0 * (fix (λ fact.λ x. if iszero x then 1 else x * fact (x - 1))) (0 - 1)
→3 * 2 * 1 * 1
```

# Evaluation Strategies

## Normal Order vs. Applicative Order

Consider the following programs:

```
> cat ite.lam
let f = \x.
  if (iszero x) then x + 3
  else x * 3
in
  f 0
```

Figure 3 : ite.lam

```
> cat fact.lam
let rec fact = \x.
  if (iszero x) then 1
  else x * fact (x-1)
in
  fact 4
```

Figure 4 : fact.lam

# Evaluation Strategies

## Normal Order vs. Applicative Order

Applicative order needs 4 more beta reductions. Applicative order is a strict reduction strategy so both the then and the else parts will be evaluated.

```
> ./jebus eval -e=normal -t < ite.lam  
(\f . f (\f . \x . x)) ..... =>  
... =>  
...  
... =>  
\f . \x . f (f (f x))  
Performed 11 beta reductions.
```

Figure 5 : Evaluate ite.lam with normal order strategy. Only 11 beta reductions needed.

```
> ./jebus eval -e=applicative -t < ite.lam  
(\f . f (\f . \x . x)) ..... =>  
... =>  
...  
... =>  
\f . \x . f (f (f x))  
Performed 14 beta reductions.
```

Figure 6 : Evaluate ite.lam with applicative order strategy. 15 beta reductions needed.

# Evaluation Strategies

## Normal Order vs. Applicative Order

```
> ./jebus eval -e=normal -t < fact.lam
((\ . fac (\f . \x . f (f (f (f x)))))) ..... =>
... =>
...
... =>
\z . \x . z (z (z (z (z (z (z (z (z (z (z (z (z (z (z (z (z (z (z (z (z (z (z (z (z x)...))
Performed 9236 beta reductions.
```

Figure 7 : Evaluate fact.lam with normal order strategy. The program terminates after 9236 reductions.

# Evaluation Strategies

## Normal Order vs. Applicative Order

```
> ./jebus eval -e=applicative -t < fact.lam  
((\ . fac (\f . \x . f (f (f (f x)))))) ..... =>  
... =>  
...  
...
```

Figure 8 : Evaluate fact.lam with applicative order strategy. The program does not terminate.

# Useful links

- ▶ [Notes](#) from NTUA's Applications of Logic in Computer Science course
- ▶ [Chapter 5](#) from the book Syntax and Semantics of Programming Languages, Kenneth Slonneger, Barry L. Kurtz
- ▶ [Hindley-Milner Typing and Algorithm W](#) from Compiler Construction course notes, Utrech University
- ▶ [lambda library](#) from NYU Lambda Seminar
- ▶ [Simply typed lambda calculus extensions](#) from Programming Languages course notes, University of Washington

# The end!

Demo

Fork here!