Υλοποίηση ενός διερμηνέα για Λάμβδα Λογισμό

Ζωή Παρασκευοπούλου
Νίκος Γιανναράκης

Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Εθνικό Μετσόβιο Πολυτεχνείο

13 Νοεμβρίου 2013

The Syntax

$$
\begin{aligned}
e := {} & e_1 \ e_2 \\
& \mid \text{``}\boldsymbol{\lambda}\text{''}\ id\ \text{``.''}\ e \\
& \mid \text{``}\textbf{\textit{let}}\text{''}\ [\text{``}\textbf{\textit{rec}}\text{''}]\ id\ \text{``} = \text{''}\ e_1\ \text{``}\textbf{\textit{in}}\text{''}\ e_2 \\
& \mid \text{``[''}e1\text{``,''}\ e2\text{``]''} \\
& \mid id \\
& \mid \text{``}\textbf{\textit{true}}\text{''} \mid \text{``}\textbf{\textit{false}}\text{''} \\
& \mid \text{``}\textbf{\textit{if}}\text{''}\ e_1\ \text{``}\textbf{\textit{then}}\text{''}\ e_2\ \text{``}\textbf{\textit{else}}\text{''}\ e_3 \\
& \mid e_1\ op\ e_2 \\
& \mid e_1\ rop\ e_2 \\
& \mid e_1\ bop\ e_2
\end{aligned}
$$

$$
\begin{aligned}
op := {} & \text{``} + \text{''} \mid \text{``} - \text{''} \\
& \mid \text{``} * \text{''} \\
& \mid \text{``} ** \text{''} \\
rop := {} & \text{``} = \text{''} \mid \text{``} < \text{''} \\
& \mid \text{``} <= \text{''} \mid \text{``} > \text{''} \\
& \mid \text{``} >= \text{''} \\
bop := {} & \text{``\&\&''} \\
& \mid \text{``}||\text{''}
\end{aligned}
$$

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 τ for simple types, σ for type schemes and α for type variables.

$$
\begin{aligned}
\tau :=\ & \tau_1 \to \tau_2 \\
& |\tau_1 \times \tau_2 \\
& |Nat \\
& |Bool \\
& |\alpha
\end{aligned}
\qquad\qquad
\begin{aligned}
\sigma :=\ & \forall \alpha.\sigma_1 \\
& |\ \tau \\
\\
\alpha :=\ & \alpha_1|\alpha_2|....
\end{aligned}
$$

Hindley-Milner typing

Typing Rules

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

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

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 \; e_2 : \tau_2}\, @$$

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

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

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

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

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

Hindley-Milner typing

Typing Rules (cont.)

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

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

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

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

The Core Language

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

$$
\begin{aligned}
e := {} & e_1 \;\; e_2 \\
\mid {} & \boldsymbol{\lambda}\, id \,\boldsymbol{.}\, e \\
\mid {} & \boldsymbol{Fix}\; e_1
\end{aligned}
$$

The Core Language

Syntactic Sugar

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

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

- **Arithmetical Operations**

$$e_1 + e_2 \equiv (\lambda\ x.\ \lambda\ y.\ x\ succ\ y)\, e_1\ \ e_2$$
$$e_1 - e_2 \equiv (\lambda\ x.\ \lambda\ y.\ y\ pred\ x)\, e_1\ \ e_2$$
$$e_1 * e_2 \equiv (\lambda\ x.\ \lambda\ y.\ \lambda\ z.\ x\ y\ z)\, e_1\ \ e_2$$
$$e_1 * * e_2 \equiv (\lambda\ x.\ \lambda\ y.\ y\ x)\, e_1\ \ e_2$$

The Core Language

Syntactic Sugar

- **Boolean Constants**
  $\mathit{true} \equiv \lambda\ x.\ \lambda\ y.\ x \qquad \mathit{false} \equiv \lambda\ x.\ \lambda\ y.\ y$
- **Pairs** $[e_1,\ e_2] \equiv \lambda\ x.\ x\ e_1\ \ e_2$
- **Provided functions for pairs**
  $\mathit{fst} \equiv \lambda\ x.\ x\ \mathit{true}$ with type $\forall\ a.\forall b.\ a \times b \to a$
  $\mathit{snd} \equiv \lambda\ x.\ x\ \mathit{false}$ with type $\forall\ a.\forall b.\ a \times b \to b$
- **Provided functions for Integers**
  $\mathit{succ} \equiv \lambda\ x.\ \lambda\ s.\ \lambda\ z.\ s\ (n\ s)\ z$ with type $\mathit{Nat} \to \mathit{Nat}$
  $\mathit{iszero} \equiv \lambda\ x.\ x\ (\mathit{true}\ \mathit{false})\ \mathit{true}$ with type $\mathit{Nat} \to \mathit{Bool}$
  $\mathit{pred} \equiv \lambda\ x.\ \mathit{snd}\ (x\ \mathit{next}\ [0, 0])$ with type $\mathit{Nat} \to \mathit{Nat}$
  where $\mathit{next} \equiv \lambda\ x.\ [\mathit{succ}\ (\mathit{fst}\ x),\ (\mathit{fst}\ x)]$

The Core Language

Syntactic Sugar

- **Boolean Operators**

  $not \equiv \lambda\ x.\ x\ false\ true$

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

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

- **Relative Operators**

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

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

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

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

  $e_1 > e_2 == e_2 < e_1$

The Core Language

Syntactic Sugar

- **Let Definitions**
  $let\, x \;=\; e_1 \; in \; e_2 \equiv (\lambda\, x.\; e_2)\; e_1$
- **Let rec is more tricky**
  $let\, rec\, x \;=\; e_1 \; in \; e_2 \equiv (\lambda\, x.\; e_2)\; (\boldsymbol{Y}\, (\lambda\, x.\; e_1))$
  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: $let\, rec\, x \;=\; e_1 \; in \; e_2 \equiv (\lambda\, x.\; e_2)\; (\boldsymbol{Fix}\, (\lambda\, x.\; e_1))$
  In both cases e_1 is allowed to refer to x. The difference is that, unlike Y, $\boldsymbol{Fix}$ can be typed with the following rule:

$$\frac{\Gamma \vdash e : \tau \rightarrow \tau}{\Gamma \vdash Fix\; e : \tau}\, fix$$

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!

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

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

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

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

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.

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

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

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

Evaluation Strategies

Fix: Example

$$
\begin{aligned}
& \textit{let rec fact} \;=\; \lambda\ x.\ \textit{if iszero}\ x\ \textit{then}\ 1\ \textit{else}\ x * \textit{fact}\ (x-1)\ \textit{in fact}\ 3 \\
\rightarrow & (\lambda\ \textit{fact}.\ \textit{fact}\ 3)\ (\textbf{\textit{fix}}\ (\lambda\ \textit{fact}.\lambda\ x.\ \textit{if iszero}\ x\ \textit{then}\ 1\ \textit{else}\ x * \textit{fact}\ (x-1))) \\
\rightarrow & (\textbf{\textit{fix}}\ (\lambda\ \textit{fact}.\lambda\ x.\ \textit{if iszero}\ x\ \textit{then}\ 1\ \textit{else}\ x * \textit{fact}\ (x-1)))\ 3 \\
\rightarrow & (\lambda\ x.\ \textit{if iszero}\ x\ \textit{then}\ 1\ \textit{else}\ x * (\textbf{\textit{fix}}\ (\lambda\ \textit{fact}.\lambda\ x.\ \textit{if iszero}\ x\ \textit{then}\ 1\ \textit{else}\ x * \textit{fact}\ (x-1)))\ (x-1))\ 3 \\
\rightarrow & \textit{if iszero}\ 3\ \textit{then}\ 1\ \textit{else}\ 3 * (\textbf{\textit{fix}}\ (\lambda\ \textit{fact}.\lambda\ x.\ \textit{if iszero}\ x\ \textit{then}\ 1\ \textit{else}\ x * \textit{fact}\ (x-1)))\ (3-1) \\
\rightarrow & 3 * (\textbf{\textit{fix}}\ (\lambda\ \textit{fact}.\lambda\ x.\ \textit{if iszero}\ x\ \textit{then}\ 1\ \textit{else}\ x * \textit{fact}\ (x-1)))\ 2 \\
\rightarrow & 3 * (\lambda\ x.\ \textit{if iszero}\ x\ \textit{then}\ 1\ \textit{else}\ x * (\textbf{\textit{fix}}\ (\lambda\ \textit{fact}.\lambda\ x.\ \textit{if iszero}\ x\ \textit{then}\ 1\ \textit{else}\ x * \textit{fact}\ (x-1)))\ (x-1))\ 2 \\
\rightarrow & 3 * \textit{if iszero}\ 2\ \textit{then}\ 1\ \textit{else}\ 2 * (\textbf{\textit{fix}}\ (\lambda\ \textit{fact}.\lambda\ x.\ \textit{if iszero}\ x\ \textit{then}\ 1\ \textit{else}\ x * \textit{fact}\ (x-1)))\ (2-3) \\
\rightarrow & 3 * 2 * (\textbf{\textit{fix}}\ (\lambda\ \textit{fact}.\lambda\ x.\ \textit{if iszero}\ x\ \textit{then}\ 1\ \textit{else}\ x * \textit{fact}\ (x-1)))\ 1 \\
\rightarrow & 3 * 2 * (\lambda\ x.\ \textit{if iszero}\ x\ \textit{then}\ 1\ \textit{else}\ x * (\textbf{\textit{fix}}\ (\lambda\ \textit{fact}.\lambda\ x.\ \textit{if iszero}\ x\ \textit{then}\ 1\ \textit{else}\ x * \textit{fact}\ (x-1)))\ (x-1))\ 1 \\
\rightarrow & 3 * 2 * \textit{if iszero}\ 1\ \textit{then}\ 1\ \textit{else}\ 1 * (\textbf{\textit{fix}}\ (\lambda\ \textit{fact}.\lambda\ x.\ \textit{if iszero}\ x\ \textit{then}\ 1\ \textit{else}\ x * \textit{fact}\ (x-1)))\ (1-1) \\
\rightarrow & 3 * 2 * 1 * (\textbf{\textit{fix}}\ (\lambda\ \textit{fact}.\lambda\ x.\ \textit{if iszero}\ x\ \textit{then}\ 1\ \textit{else}\ x * \textit{fact}\ (x-1)))\ 0 \\
\rightarrow & 3 * 2 * 1 * (\lambda\ x.\ \textit{if iszero}\ x\ \textit{then}\ 1\ \textit{else}\ x * (\textbf{\textit{fix}}\ (\lambda\ \textit{fact}.\lambda\ x.\ \textit{if iszero}\ x\ \textit{then}\ 1\ \textit{else}\ x * \textit{fact}\ (x-1)))\ (x-1))\ 0 \\
\rightarrow & 3 * 2 * 1 * \textit{if iszero}\ 0\ \textit{then}\ 1\ \textit{else}\ 0 * (\textbf{\textit{fix}}\ (\lambda\ \textit{fact}.\lambda\ x.\ \textit{if iszero}\ x\ \textit{then}\ 1\ \textit{else}\ x * \textit{fact}\ (x-1)))\ (0-1) \\
\rightarrow & 3 * 2 * 1 * 1
\end{aligned}
$$

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 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 form 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!