Dynamically Typed Programming Languages

Part 1: The Untyped λ-Calculus

Jim Royer

CIS 352

April 16, 2019

Reference

- *Practical Foundations for Programming Languages, 2/e, "Part VI: Dynamic Types"*, by Robert Harper, Cambridge University Press, 2016, pages 183–210.
  https://www.cs.cmu.edu/%7Erwh/pfpl/2nded.pdf

  Note: *Harper doesn't much care for "dynamically typed languages" and his criticisms are dead on, but these languages do have some advantages.*

- *Lambda calculus definition*, from Wikipedia.
  https://en.wikipedia.org/wiki/Lambda_calculus_definition

Important!!!

dynamically typed $\neq$ dynamically scoped

- *Dynamically typed* roughly means
  we may not find out the *type* of a value until runtime.
- *Dynamically scoped* roughly means
  we may not find out a variable's binding until runtime.
- Dynamically Typed Programming Languages include:
  Lisp, Scheme, Racket, Python, Clojure, Erlang, JavaScript, Julia,
  Lua, Perl, R, Ruby, Smalltalk, ...
  (See `https://en.wikipedia.org/wiki/Dynamic_programming_language#Examples`.)

  The "premier example" is the (untyped) λ-calculus.

The λ-Calculus

The λ-Calculus: Beginnings

- Alonzo Church originally developed the λ-calculus as part of a grand formalism for the foundations of mathematics.
  But …

The λ-Calculus: Beginnings

- Alonzo Church originally developed the λ-calculus as part of a grand formalism for the foundations of mathematics.
  But ...
- Stephen Kleene and Barkely Rosser showed that system was inconsistent.
  So ...

The λ-Calculus: Beginnings

- Alonzo Church originally developed the λ-calculus as part of a grand formalism for the foundations of mathematics.
  But …
- Stephen Kleene and Barkely Rosser showed that system was inconsistent.
  So …
- The formalism was cut back to the part that was about defining functions, λ-calculus.
  Amazingly …

The λ-Calculus: Beginnings

- Alonzo Church originally developed the λ-calculus as part of a grand formalism for the foundations of mathematics.
  But . . .
- Stephen Kleene and Barkely Rosser showed that system was inconsistent.
  So . . .
- The formalism was cut back to the part that was about defining functions, λ-calculus.
  Amazingly . . .
- The λ-calculus turns out to be *Turing-complete*. Why "amazingly"?

The λ-Calculus: Beginnings

- Alonzo Church originally developed the λ-calculus as part of a grand formalism for the foundations of mathematics.
  But . . .
- Stephen Kleene and Barkely Rosser showed that system was inconsistent.
  So . . .
- The formalism was cut back to the part that was about defining functions, λ-calculus.
  Amazingly . . .
- The λ-calculus turns out to be *Turing-complete*. Why "amazingly"?
- Because on the surface there is not very much to the λ-calculus.

Definitions, 1

Concrete Syntax*

$$
\begin{array}{rcll}
E & ::= & X & \textit{(variable occurrence)} \\
 & | & (E\ E) & \textit{(function application)} \\
 & | & \lambda X \,.\, E & \textit{(function abstraction)} \\
X & ::= & \text{identifiers} &
\end{array}
$$

*Note:

- We sometimes add extra parens around lambda-expressions. E.g., we can write $((\lambda x.(y\ x))\ z)$ for $(\lambda x.(y\ x)\ z)$.
- Also, by convention application associates to the left. E.g., we can write $w\ x\ y\ z$ for $(((w\ x)\ y)\ z)$ as in Haskell.
- Harper uses a LISPy concrete syntax for $\lambda x.e$. E.g.: $\lambda(x)e$

Free and bound variables

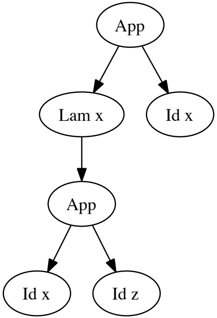
$$E \ ::= \ X \ | \ (E \ E) \ | \ \lambda X.E$$

Abstract Syntax (in Haskell)

```
type Name = String
data Exp  = Id Name | App Exp Exp | Lam Name Exp
```

$$((\lambda \overset{\color{red}{I}}{x} .(\overset{\color{green}{B}}{x} \ \overset{\color{blue}{F}}{z})) \ \ \overset{\color{blue}{F}}{x})$$

I	=	binding occurrence
B	=	bound occurence
F	=	free occurence



Computations

Definition

- An expression of the form: $((\lambda x.e_0)\ e_1)$ is called a *β-redex*.
- $((\lambda x.e_0)\ e_1)$ *β-reduces* to $e_0[e_1/x]$.
- e *β-reduces* to e' when e' is the result of replacing some subexpression of e of the form $((\lambda x.e_0)\ e_1)$ with $e_0[e_1/x]$.
- e is in *β-normal form* when it contains no β-redexes.

Examples

$$
\begin{array}{rcll}
((\lambda x.(plus\ x\ z))\ y) & \rightarrow_\beta & (plus\ y\ z) & \\
((\lambda w.(plus\ w\ z))\ (times\ x\ y)) & \rightarrow_\beta & (plus\ (times\ x\ y)\ z) & \\
((\lambda x.(\lambda z.(plus\ x\ z)))\ y) & \rightarrow_\beta & (\lambda z.(plus\ y\ z)) & \\
((\lambda x.(\lambda y.(plus\ x\ y)))\ y) & \not\rightarrow_\beta & (\lambda y.(plus\ y\ y)) & \textit{variable capture!!} \\
((\lambda x.(\lambda y.(plus\ x\ y)))\ y) & \rightarrow_\beta & (\lambda z.(plus\ y\ z)) & \\
(x\ (\lambda y.y)) & \not\rightarrow_\beta & \textit{anything} & \textit{as it is in normal form} \\
\Omega =_{def} ((\lambda x.(x\ x))\ (\lambda x.(x\ x))) & \rightarrow_\beta & ((\lambda x.(x\ x))\ (\lambda x.(x\ x))) = \Omega &
\end{array}
$$

β-reduction's less glamorous siblings

α-conversion (Bound vars don't mater for meaning)

$\lambda x.e \ \equiv_\alpha \ \lambda y.(e[y/x])$ where $x \neq y$.
(I.e., Renaming bound vars doesn't change meaning.)

η-conversion (Extensionality)

$((\lambda x.e)\ x) \ \equiv_\eta \ e$ when $x \notin \mathit{freeVars}(e)$.
(I.e., λ-terms producing the same result on all args are equivalent.)

Normal Order Evaluation *(think preorder)*

The Normal Order Evaluation Strategy

Always do the leftmost possible beta-reduction.
Repeat until (if ever) you reach a normal form. [Draw the parse tree!]

Normal Order Evaluation Strategy is equivalent to:

$$
\begin{array}{lll}
\textbf{function } \mathit{nor}(M) & & \\
\quad \textbf{if} & M = ((\lambda x.N)\ P) & \textbf{then } \mathit{nor}(N[P/x]) \\
\quad \textbf{else if} & M = (N\ P)\ \ \&\ \ N \to_{n.o.} N' & \textbf{then } \mathit{nor}(\ (N'\ P)\ ) \\
\quad \textbf{else if} & M = (N\ P)\ \ \&\ \ P \to_{n.o.} P' & \textbf{then } \mathit{nor}(\ (N\ P')\ ) \\
\quad \textbf{else if} & M = \lambda x.N\ \ \&\ \ N \to_{n.o.} N' & \textbf{then } \mathit{nor}(\lambda x.N') \quad (\ddagger) \\
\quad \textbf{else} & (\text{* } M \text{ is in } \beta\text{-n.f. *}) & \textbf{return } M
\end{array}
$$

Theorem

If e has a normal form, normal order evaluation will eventually reach it.

($\ddagger$) Drop this line to get to ***call-by-name***.

Applicative Order Evaluation *(think postorder)*

The Applicative Order Evaluation Strategy

Always do the innermost (to the left) possible beta-reduction.
Repeat until (if ever) you run out of beta-redexes. [Draw the parse tree!]

Normal Order Evaluation Strategy is equivalent to:

function $nor(M)$
  if $M = (N\ P)$ & $N \rightarrow_{n.o.} N'$ **then** $nor(\ (N'\ P)\ )$
  else if $M = (N\ P)$ & $P \rightarrow_{n.o.} P'$ **then** $nor(\ (N\ P')\ )$
  else if $M = ((\lambda x.N)\ P)$ **then** $nor(N[P/x])$
  else if $M = \lambda x.N$ & $N \rightarrow_{n.o.} N'$ **then** $nor(\lambda x.N')$ (‡)
  else (* M is in β-n.f. *) **return** M

Fact

If e has a normal form, applicative order evaluation may not find it.

(‡) Drop this line to get to ***call-by-value***.

Aside: Are there other evaluation strategies?

More than you want to know. For starts, see:

https://en.wikipedia.org/wiki/Evaluation_strategy

The λ-calculus as a RISC assembly language, 1

Church Booleans

$$
\begin{aligned}
\mathit{true} &=_{def} \lambda t.\lambda f.t \\
\mathit{false} &=_{def} \lambda t.\lambda f.f \\
\mathit{test} &=_{def} \lambda b.\lambda m.\lambda n.((l\ m)\ n) \\
\mathit{and} &=_{def} \lambda b.\lambda c.((b\ c)\ \mathit{false})
\end{aligned}
$$

Examples

- $\mathit{test}\ \mathit{true}\ u\ v \ \to_\beta^* \ u$
- $\mathit{test}\ \mathit{false}\ u\ v \ \to_\beta^* \ v$
- $\mathit{and}\ \mathit{true}\ \mathit{true} \ \to_\beta^* \ \mathit{true}$
- $\mathit{and}\ \mathit{false}\ \mathit{true} \ \to_\beta^* \ \mathit{false}$

$\vdots$

Church Pairs

$$
\begin{aligned}
\mathit{pair} &=_{def} \lambda f.\lambda s.\lambda b.((b\ f)\ s) \\
\mathit{fst} &=_{def} \lambda p.(p\ \mathit{true}) \\
\mathit{snd} &=_{def} \lambda p.(p\ \mathit{false})
\end{aligned}
$$

Examples

- $\mathit{fst}\ (\mathit{pair}\ u\ v) \ \to_\beta^* \ u$
- $\mathit{snd}\ (\mathit{pair}\ u\ v) \ \to_\beta^* \ u$

The λ-calculus as a RISC assembly language, 2

Church Numerals

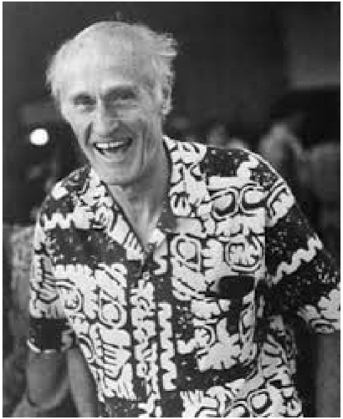
$$
\begin{aligned}
c_0 &=_{def} \lambda s.\lambda z.z \\
c_1 &=_{def} \lambda s.\lambda z.(s\ z) \\
c_2 &=_{def} \lambda s.\lambda z.(s\ (s\ z)) \\
c_3 &=_{def} \lambda s.\lambda z.(s\ (s\ (s\ z))) \\
&\vdots \\
successor &=_{def} \lambda n.\lambda s.\lambda z.(s\ (n\ s\ z)) \\
plus &=_{def} \lambda m.\lambda n.\lambda s.\lambda z.m\ s\ (n\ s\ z) \\
times &=_{def} \lambda m.\lambda n.(m\ (plus\ n)\ c_0) \\
&\vdots \\
Y &=_{def} \lambda f.(\lambda x.f(x\,x))(\lambda x.f(x\,x))
\end{aligned}
$$

"Object oriented integers"

- specify what successor (s) is
- specify what zero (z) is
- c_n = apply s n-times to z
- predecessor is difficult
  ($pred\ c_0 \to_\beta c_0$ &
  $pred\ c_{n+1} \to_\beta c_n$)

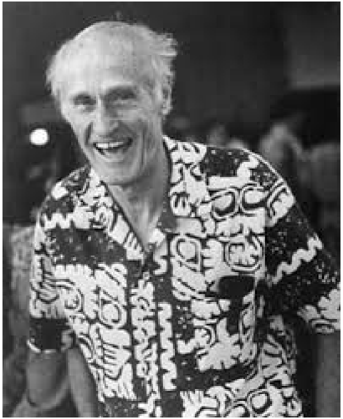
From Barendregt's *Impact of the Lambda calculus*

Kleene did find a way to lambda define the predecessor function in the untyped lambda calculus, by using an appropriate data type (pairs of integers) as auxiliary device. In [69], he described how he found the solution while being anesthetized by laughing gas (N^2O) for the removal of four wisdom teeth.



http://www-users.mat.umk.pl/~adwid/materialy/doc/church.pdf

Kleene did find a way to lambda define the predecessor function in the untyped lambda calculus, by using an appropriate data type (pairs of integers) as auxiliary device. In [69], he described how he found the solution while being anesthetized by laughing gas (N^2O) for the removal of four wisdom teeth.



http://www-users.mat.umk.pl/~adwid/materialy/doc/church.pdf

So yes, drugs were involved in all of this.

Back to Harper

Term formation *à la* Harper

There is just one type: ok.

$$\frac{}{\Gamma,\ x : \mathsf{ok} \vdash x : \mathsf{ok}}$$

$$\frac{\Gamma,\ x : \mathsf{ok} \vdash e : \mathsf{ok}}{\Gamma \vdash \lambda x.e : \mathsf{ok}}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{ok} \qquad \Gamma \vdash e_2 : \mathsf{ok}}{\Gamma \vdash (e_1\ e_2) : \mathsf{ok}}$$

Term equivalence *à la* Harper

$\Gamma \vdash u \equiv u'$ is meant to assert that u and u' (with free variables in Γ) are behaviorally equivalent.

$$\frac{}{\Gamma,\ u : \mathsf{ok} \vdash u \equiv u} \qquad \frac{\Gamma \vdash u \equiv u'}{\Gamma \vdash u' \equiv u} \qquad \frac{\Gamma \vdash u \equiv u' \qquad \Gamma \vdash u' \equiv u''}{\Gamma \vdash u \equiv u''}$$

$$\frac{\Gamma \vdash u_1 \equiv u_1' \qquad \Gamma \vdash u_2 \equiv u_2'}{\Gamma \vdash (u_1\, u_2) \equiv (u_1'\, u_2')} \qquad \frac{\Gamma, x : \mathsf{ok} \vdash u \equiv u'}{\Gamma \vdash \lambda x.u \equiv \lambda x.u'}$$

$$\frac{\Gamma, x : \mathsf{ok} \vdash u_2 : \mathsf{ok} \qquad \Gamma \vdash e_1 : \mathsf{ok}}{\Gamma \vdash ((\lambda x.e_2)\, e_1) \equiv e_2[e_1/x]}$$

Scott's Theorem

$u \equiv u'$ is undecidable.

So, if there was any doubt, we are firmly in Turing's tar-pit.

Aside: Notes on the Rules

- The first three rules say that $\equiv$ is an equiv. rel.
- The fourth rule states that applying $\equiv$ things, yields $\equiv$ results.
- The fifth rule states that abstracting on $\equiv$ things, yields $\equiv$ results.
- The last rule says that $u \to_\beta u'$ implies $u \equiv u'$.
  I.e., an evaluation step (β-reduction) preserves meaning.

Observation: The λ-calculus is seriously strange

The λ-calculus has but one "feature," the higher-order function.

Everything is a function,
and hence every expression may be applied to an argument,
which must itself be a function,
with the result also being a function.

To borrow a turn of phrase, in the λ-calculs it's functions all the way down.

Harper, page 127

In terms of the type ok, we have:

$$\mathsf{ok} \cong (\mathsf{ok} \to \mathsf{ok})$$

By Cantor's Theorem, the above makes no sense in standard set theory.

Observation: The λ-calculus is seriously strange

> ***The λ-calculus has but one "feature," the higher-order function.***
>
> ***Everything is a function,***
> ***and hence every expression may be applied to an argument,***
> ***which must itself be a function,***
> ***with the result also being a function.***
>
> ***To borrow a turn of phrase, in the λ-calculs it's functions all the way down.***
>
> *Harper, page 127*

In terms of the type ok, we have:

$$\mathsf{ok} \cong (\mathsf{ok} \to \mathsf{ok})$$

By Cantor's Theorem, the above makes no sense in standard set theory.
However, it turns out to make sense in a computability context.

Aside: Set Theoretic Notes

By Cantor, if $card(A) > 1$, then

$$card(A) < card(A^A)$$

where

$$A^A = \text{the set of (total) functions from } A \text{ to } A.$$

How to make computational sense of $\mathsf{ok} \cong (\mathsf{ok} \to \mathsf{ok})$

Briefly, as a recursive type. E.g., in Haskell

```
data Ok = F (Ok -> Ok)
```

We are skipping Harper's treatment of recursive types, but:

- It isn't too hard to represent the λ-calculus using a type like `Ok`.
- But it also is a pain. (See Harper.)
- …and it breaks as soon as you add any other data type to the lambda calculus, which is exactly what we want to do.
- So there is another approach. (Next time!)