Lambda calculus

Advanced functional programming

Wouter Swierstra

Today

- Lambda calculus – the foundation of functional programming
- What makes lambda calculus such a universal language of computation?

The Lambda Calculus

- Introduced by Church 1936 (or even earlier).
- Formal language based on variables, function abstraction and application (substitution).
- Allows to express higher-order functions naturally.
- Equivalent in computational power to a Turing machine.
- Is at the basis of functional programming languages such as Haskell.

What and why?

- A simple language with relatively few concepts.
- Easy to reason about.
- Original goal: reason about expressiveness of computations.
- Today more: core language for playing with all sorts of language features.
- Many flavours: untyped, typed, added constants and constructs.

Lambda calculus: definition

There are only three constructs:

```
e  ::=  x            (variables)
   |    e e          (application)
   |    λ x → e      (abstraction)
```

Conventions

- Note: application associates to the left:

```
a b c = (a b) c
```

- Note: only unary functions and unary application – but we write

  λ `x y` $\rightarrow$ `e`

  for

  λ `x` $\rightarrow$ `(`λ `y` $\rightarrow$ `e)`.

- Note: the function body of a lambda extends as far as possible to the right: λ `x` $\rightarrow$ `e f` should be read as

λ `x` $\rightarrow$ `(e f)`

Definitions

- We usually consider terms *equal up to renaming* (alpha equivalence);
- The central computation rule is *beta reduction*:

`(λ x → e) (a)` reduces to `e [x/a]`

Applications?

It seems as if we can do nothing useful with the lambda calculus.

There are no constants – no numbers, for instance.

But it turns out that we can **encode** recursion, numbers, booleans, and just about any other data type.

Church numerals

$$
\begin{aligned}
&\texttt{zero} \equiv \lambda\ \texttt{s}\ \texttt{z} \rightarrow \texttt{z} \\
&\texttt{one} \equiv \lambda\ \texttt{s}\ \texttt{z} \rightarrow \texttt{(s z)} \\
&\texttt{two} \equiv \lambda\ \texttt{s}\ \texttt{z} \rightarrow \texttt{(s (s z))} \\
&\texttt{three} \equiv \lambda\ \texttt{s}\ \texttt{z} \rightarrow \texttt{(s (s (s z)))} \\
&\ldots
\end{aligned}
$$

So far, so good, but can we calculate with these numbers?

Addition

$$\texttt{suc} \equiv \lambda\ \texttt{n} \to \lambda\ \texttt{s}\ \texttt{z} \to \texttt{(s (n s z))}$$

$$\texttt{add} \equiv \lambda\ \texttt{m}\ \texttt{n} \to \texttt{m suc n}$$

Does this work as expected?

Addition

$$\texttt{suc} \equiv \lambda\ \texttt{n} \to \lambda\ \texttt{s}\ \texttt{z} \to \texttt{(s (n s z))}$$

$$\texttt{add} \equiv \lambda\ \texttt{m}\ \texttt{n} \to \texttt{m suc n}$$

Does this work as expected?

```
suc two
```

Addition

$$\texttt{suc} \equiv \lambda\ \texttt{n} \to \lambda\ \texttt{s}\ \texttt{z} \to \texttt{(s (n s z))}$$

$$\texttt{add} \equiv \lambda\ \texttt{m}\ \texttt{n} \to \texttt{m suc n}$$

Does this work as expected?

```
suc two
```

```
(λ n → (λ s z → (s (n s z)))) two
```

Addition

$$\texttt{suc} \equiv \lambda\ \texttt{n} \to \lambda\ \texttt{s}\ \texttt{z} \to \texttt{(s (n s z))}$$

$$\texttt{add} \equiv \lambda\ \texttt{m}\ \texttt{n} \to \texttt{m suc n}$$

Does this work as expected?

```
suc two
```

$$(\lambda\ \texttt{n} \to (\lambda\ \texttt{s}\ \texttt{z} \to \texttt{(s (n s z))))\ two}$$

$$\lambda\ \texttt{s}\ \texttt{z} \to \texttt{(s (two s z))}$$

Addition

$$\texttt{suc} \equiv \lambda\ \texttt{n} \to \lambda\ \texttt{s}\ \texttt{z} \to \texttt{(s (n s z))}$$

$$\texttt{add} \equiv \lambda\ \texttt{m}\ \texttt{n} \to \texttt{m suc n}$$

Does this work as expected?

```
suc two
```

$$\texttt{(}\lambda\ \texttt{n} \to \texttt{(}\lambda\ \texttt{s}\ \texttt{z} \to \texttt{(s (n s z)))) two}$$

$$\lambda\ \texttt{s}\ \texttt{z} \to \texttt{(s (two s z))}$$

$$\lambda\ \texttt{s}\ \texttt{z} \to \texttt{(s (s (s z)))}$$

Church Booleans

$$\texttt{true} \equiv \lambda\ \texttt{t}\ \texttt{f} \to \texttt{t}$$

$$\texttt{false} \equiv \lambda\ \texttt{t}\ \texttt{f} \to \texttt{f}$$

$$\texttt{ifthenelse} \equiv \lambda\ \texttt{c}\ \texttt{t}\ \texttt{e} \to \texttt{c}\ \texttt{t}\ \texttt{e}$$

Church Booleans

$$\texttt{true} \equiv \lambda\ \texttt{t f} \to \texttt{t}$$

$$\texttt{false} \equiv \lambda\ \texttt{t f} \to \texttt{f}$$

$$\texttt{ifthenelse} \equiv \lambda\ \texttt{c t e} \to \texttt{c t e}$$

The function `ifthenelse` is almost the identity function.

$$\texttt{and} \equiv \lambda\ \texttt{x y} \to \texttt{ifthenelse x y false}$$

$$\texttt{and} \equiv \lambda\ \texttt{x y} \to \texttt{x y false}$$

$$\texttt{or} \equiv \lambda\ \texttt{x y} \to \texttt{ifthenelse x true y}$$

$$\texttt{or} \equiv \lambda\ \texttt{x y} \to \texttt{x true y}$$

Church Booleans

$$\texttt{true} \equiv \lambda\ \texttt{t}\ \texttt{f} \to \texttt{t}$$

$$\texttt{false} \equiv \lambda\ \texttt{t}\ \texttt{f} \to \texttt{f}$$

$$\texttt{ifthenelse} \equiv \lambda\ \texttt{c}\ \texttt{t}\ \texttt{e} \to \texttt{c}\ \texttt{t}\ \texttt{e}$$

The function `ifthenelse` is almost the identity function.

$$\texttt{and} \equiv \lambda\ \texttt{x}\ \texttt{y} \to \texttt{ifthenelse x y false}$$

$$\texttt{and} \equiv \lambda\ \texttt{x}\ \texttt{y} \to \texttt{x y false}$$

$$\texttt{or} \equiv \lambda\ \texttt{x}\ \texttt{y} \to \texttt{ifthenelse x true y}$$

$$\texttt{or} \equiv \lambda\ \texttt{x}\ \texttt{y} \to \texttt{x true y}$$

The function `isZero` takes a number and returns a Bool.

$$\texttt{isZero} \equiv \lambda\ \texttt{n} \to \texttt{(n (}\lambda\ \texttt{x} \to \texttt{false) true)}$$

Pairs

$$\texttt{pair} \equiv \lambda\ \texttt{x y} \rightarrow (\lambda\ \texttt{p} \rightarrow (\texttt{p x y}))$$

$$\texttt{fst} \equiv \lambda\ \texttt{p} \rightarrow (\texttt{p } (\lambda\ \texttt{x y} \rightarrow \texttt{x}))$$

$$\texttt{snd} \equiv \lambda\ \texttt{p} \rightarrow (\texttt{p } (\lambda\ \texttt{x y} \rightarrow \texttt{y}))$$

The function `pair` remembers its two parameters and returns them when asked by its third parameter.

How do you come up with these definitions?

Church encoding for arbitrary datatypes

There is a correspondence between the so-called *fold* (or *catamorphism* or *eliminator*) for a datatype and its Church encoding.

Haskell:

```
data Nat = Suc Nat | Zero


foldNat Zero     s z  =  z
foldNat (Suc n)  s z  =  s (foldNat n s z)
```

Lambda calculus:

$$\texttt{zero} \equiv \lambda\ \texttt{s z} \rightarrow \texttt{z}$$

$$\texttt{suc n} \equiv \lambda\ \texttt{s z} \rightarrow \texttt{(s (n s z))}$$

Church encoding for arbitrary datatypes – contd.

Haskell:

```
data Bool = True | False


foldBool True  t f = t
foldBool False t f = f
```

Lambda calculus:

$$\texttt{true} \equiv \lambda\ \texttt{t}\ \texttt{f} \to \texttt{t}$$
$$\texttt{false} \equiv \lambda\ \texttt{t}\ \texttt{f} \to \texttt{f}$$

Note that `foldBool` is just `ifthenelse` again.

Church encoding for arbitrary datatypes – contd.

Haskell:

```
data Pair x y = Pair x y

foldPair (Pair x y) p = p x y
```

Lambda calculus:

$$\texttt{pair} \equiv \lambda\ \texttt{x y} \rightarrow (\lambda\ \texttt{p} \rightarrow (\texttt{p x y}))$$

Encoding vs. adding constants

The fact that we can encode certain entities in the lambda justifies that we can add them as constants to the language without changing the nature of the language.

Example (adding Booleans)

```
e  ::=  true
    |    false
    |    if e then e else e
```

Once we have new forms of expressions, we need more than just beta-reduction:

```
if true   then e1 else e2 ⟶  e1
if false  then e1 else e2 ⟶  e2
```

Towards Haskell

Haskell is based on the lambda calculus.

Yet, so far it seems hard to believe that we can desugar Haskell to some form of lambda calculus.

Binding names with `let`

Haskell allows us to bind identifiers to expressions in the language using `let`.

We have only introduced informal abbreviations for lambda terms so far such as `true` or `isZero`.

Binding names with `let` – contd.

In fact, `let` can simply be desugared to a lambda binding.

```
let x = e1 in e2  ≡ (λ x → (e2)) e1
```

Note that this does not work if `x` is a recursive binding or if you want to preserve sharing.

What about recursion, then?

Haskell example

```
fac = \ n → if n == 0 then 1 else n * fac (n - 1)
```

```
fac = fix
  (\ fac n → if n == 0 then 1
                       else n * fac (n - 1))
```

The desired function `fac` can be viewed as a fixed point of the related non-recursive function `fac'`.

Fixed points

A *fixed-point combinator* is a combinator `fix` with the property that for any `f`,

```
-- Using recursion directly
fix f = f (fix f)
```

In particular,

```
fix fac' = fac' (fix fac')
```

thus `fix fac'` is a fixed point of `fac`.

Fixed-point combinators

Many fixed-point combinators can be defined in the untyped lambda calculus.

Here is one of the smallest and most famous ones, called Y.

```
Y ≡ λ f → (λ x → (f (x x))) (λ x → (f (x x)))
```

Verification that `Y` is a fixed-point combinator

```
Y f
```

Verification that `Y` is a fixed-point combinator

```
Y f
```

$\equiv$

```
(λ x → (f (x x))) (λ x → (f (x x)))
```

Verification that Y is a fixed-point combinator

```
Y f
≡
(λ x → (f (x x))) (λ x → (f (x x)))
≡
f ((λ x → (f (x x))) (λ x → (f (x x))))
..
≡
f (Y f)
```

Recap

It is thus possible to desugar a recursive Haskell definition into the lambda calculus by translating recursion into applications of `fix`.

Conversely, we can justify adding recursion as a construct to the lambda calculus without changing its essential nature.

General vs. structural recursion

Note that most recursive functions can actually be defined without a fixed-point combinator. We have already defined `add`:

```
add  ≡  λ m n → (m suc n)
```

In Haskell, `add` would be recursive

```
data Nat = Suc Nat | Zero


add (Suc m)  n  =  Suc (add m n)
add Zero     n  =  n
```

but can also be defined in terms of `foldNat`:

```
add m n = foldNat m Suc n
```

General vs. structural recursion – contd.

Functions defined in terms of a fold function are called *structurally recursive*.

Recursion using the fixed-point combinator is called *general recursion*.

Writing functions using general recursion is often perceived as simpler or more direct.

Structural recursion is often more well-behaved. For instance, for many datatypes it can be proved that if the arguments to the fold terminate, the structurally recursive function also terminates.

Pattern matching

In Haskell we can define functions using pattern matching:

```
data Nat = Suc Nat | Zero


pred (Suc m)  = m
pred Zero     = Zero
```

Question

How can we define `pred` for the Church numerals?

Case function

Alternatively, pattern matching via case on a natural number can be captured as a function:

```
caseNat :: Nat → (Nat → r) → r → r
caseNat (Suc n)  s z  =  s n
caseNat Zero     s z  =  z

pred = \ m → caseNat  m
             (\ m' → m')
             Zero
```

The case function can be expressed in terms of the fold for that datatype, and hence the Church encoding.

Case function – contd.

Haskell:

```
caseNat :: Nat → (Nat → r) → r → r
caseNat (Suc n)  s z  =  s n
caseNat Zero     s z  =  z

foldNat :: Nat → (s → s) → s → s
foldNat (Suc n)  s z  =  s (foldNat n s z)
foldNat Zero     s z  =  z
```

Case via fold

We call `foldNat` choosing `s ≡ (r, Nat)` – that is pairing the return type and natural number:

```
caseNat n s z ≡
  fst (foldNat n (\ (_,r) → (s r, Suc r))
                           (z, zero))
```

The second component of the pair just constructs the natural number again. This is how we can access the predecessor!

Nested patterns

Haskell allows nested patterns, too:

```
fib Zero          =  Zero
fib (Suc Zero)    =  Suc Zero
fib (Suc (Suc n)) =  add (fib n) (fib (Suc n))
```

These can easily be desugared to nested applications of `case` using only flat patterns (and hence to applications of `caseNat`):

```
fib n = case n of
          Zero    → Zero
          Suc n'  → case n' of
                      ...
```

Recap

We have seen how most Haskell constructs can be desugared to the lambda calculus:

- constructors of datatypes using the Church encoding,
- non-recursive `let` using lambda abstractions,
- general recursion using a fixed-point combinator,
- pattern matching using possibly nested applications of case functions.

Recap – contd.

Many other Haskell constructs can be expressed in terms of the ones we have already seen – for instance:

- `where`-clauses can be transformed into `let`
- `if-then-else` can be expressed as a function
- list comprehensions can be transformed into applications of `map`, `concat` and `if-then-else`
- monadic `do` notation can be transformed into applications of a limited number of functions

Even Simpler

A straightforward implementation of the lambda calculus may give rise to abitrary large reduction steps. We can represent all lambda expressions using only three combinators with the following reduction behaviour:

```
S f g  x = (f x) (g x)
K y    x = y
I      x = x
```

Translation

Given a lambda term of the form – how can we translate this to an expression using SKI?

```
data SKI = Var String | S | K | I | App SKI SKI


toSKI :: Lambda -> SKI
toSKI (Var x)     = Var x
toSKI (App t1 t2) = (toSKI t1)  `App`(toSKI t2)
toSKY (Lam x t)   = remove x (toSKI t)
```

The auxiliary function `remove` does the actual work...

Bracket abstraction

```
remove :: Var -> Lambda -> SKI
remove x (Var y)
  | x == y = I
  | otherwise = K `App` y
remove x (App t1 t2) =
  S `App` (remove (App t1 x))
    `App` (remove (App t2 x))
```

This is sometimes called *bracket abstraction*.

Note: there is no case for lambdas – why?

Intuition

What's going on?

```
S f g  x = (f x) (g x)
K y    x = y
I      x = x
```

`S` is *duplicating* a variable; `K` is discarding a variable; `I` is using a variable.

Intuition

What's going on?

```
S f g  x = (f x) (g x)
K y    x = y
I      x = x
```

`S` is *duplicating* a variable; `K` is discarding a variable; `I` is using a variable.

Bracket abstraction simply explains *how* to route the argument of a function to the variable's occurrences in the lambda's body.

Alternatives

Haskell Curry proposed the following combinators:

```
B x y z = x (y z)
C x y z = x z y
K x y = x
W x y = x y y
```

Here `B` ‘routes arguments’ to the left only; `C` ‘routes arguments’ to the right; and `W` duplicates its inputs.

Even Simpler

The combinator `I` is superfluous:

`S K K x` $\longrightarrow$ `(K x) (K x)` $\longrightarrow$ `x`

and hence

```
I = S K K
```

Even Simpler

In 1989 Jeroen Fokker (UU) invented:

```
X = λ f → (f S f3)
f3 = λ p _ _ → p        -- first of three
```

with which we can define K as follows:

```
K y x   →   X   X                    y   x
```

Does it reduce as expected?

Even Simpler

In 1989 Jeroen Fokker (UU) invented:

```
X = λ f → (f S f3)
f3 = λ p _ _ → p        -- first of three
```

with which we can define K as follows:

```
K y x   →  X  X                  y  x
```

Does it reduce as expected?

```
        →  X  S  f3               y  x
        →  S  S  f3        f3     y  x
        →  S  f3 (f3 f3)   y         x
        →  f3 y  (f3 f3 y) x
        →  y
```

Even simpler

Check for yourself:

```
S = X (X X)
```

Back to Haskell

This is nice – but what does this have to do with Haskell?

Back to Haskell

This is nice – but what does this have to do with Haskell?

GHC translates to an intermediate language: *GHC Core*.

GHC Core is really little more than a (typed) lambda calculus.

You can read the spec on GitHub.

Core sketch

GHC Core is based on System Fc – a typed lambda calculus extended with type coercions.

- variables, lambdas, and application;
- literals;
- let bindings;
- case expressions;
- coercions – used to implement GADTs amongst other things;
- ‘ticks’ – used for HPC to track program coverage.

Inspecting core can be useful to see how code is generated and optimized.

Generating core

```
alias ghci-core="ghci -ddump-simpl \
-dsuppress-idinfo -dsuppress-coercions \
-dsuppress-type-applications \
-dsuppress-uniques -dsuppress-module-prefixes"
```

The following Haskell code and corresponding Core:

```
f :: Int -> Int
f x = x + 1

f :: Int -> Int
f = \ (x :: Int) ->
      case x of _ { I# x1 -> I# (+# x1 1) }
```

What we haven't discussed yet

Types

Compare

```
false  ≡  λ t f -> f
zero   ≡  λ s z -> z
```

We can easily write terms that do not make sense in lambda calculus; Haskell has types to prevent that.

Overloading

In Haskell, functions can be overloaded using type classes. How can such overloading be resolved and desugared?

What we haven't discussed yet – contd.

Laziness

Haskell makes use of a particular evaluation strategy called lazy evaluation. We have not looked at evaluation strategies at all so far.

Side effects

The lambda calculus has no notion of effects, not even encapsulated effects such as Haskell offers with `IO`. So the behaviour of `IO` cannot be described by reduction to the lambda calculus.

Conclusion

Conclusion

“The lambda calculus has many applications.”