An untyped λ-calculus, *UL*

Principles of Programming Languages

Mark Armstrong

Fall 2020

1 Preamble

1.1 TODO Notable references

- Benjamin Pierce, "Types and Programming Languages"
  - Chapter 5, The Untyped Lambda-Calculus

1.2 TODO Table of contents

- Preamble

2 Introduction

In this section we construct our first simple programming language, an untyped λ-calculus (lambda calculus).

More specifically, we construct a λ-calculus without (static) type checking (enforcement), but including the natural numbers and booleans.

2.1 What is the λ-calculus?

The λ-calculus is, put simply, a notation for forming and applying functions.

- Because the function (procedure, method, subroutine) abstraction gives us a means of representing control flow, if we have a means of representing data, the λ-calculus is a Turing-complete model of computation.

2.2 History

The (basic) λ-calculus as we know it was famously invented by Alonzo Church in the 1920s.

- This was one culmination of a great deal of work by mathematicians investigating the foundations of mathematics.

As mentioned, the λ-calculus is a Turing-complete model of computation.

- Other models proposed around the same time include
  - the Turing machine itself (due to Alan Turing), and
  - the general recursive functions (due to Stephen Cole Kleene.)
- Hence the "Church" in the "Church-Turing thesis".

The λ-calculus has since seen widespread use in the study and design of programming languages.

- It is useful both as a simple programming language, and
- as a mathematical object about which statements can be proved.

2.3 Descendents of the λ-calculus

Pure functional programming languages are clearly descended from the λ-calculus; the λ-calculus embodies their model of computation.

- Additionally, it is common to have a "lambda" operator which allows definition of anonymous functions.
  - This is so even outside of pure functional languages,
    * and it is becoming common in primarily imperative languages as well!

Imperative languages instead (traditionally) use a model of computation based on the *Von-Neumann* architecture,

- which matches our real-world computing devices!
  - Hence imperative languages are naturally lower-level; one level of abstraction closer to the real computer that functional languages, which must be translated to imperative code in order to run.
- This model of computation is a natural extension of the Turing machine, rather than the λ-calculus or recursive functions.

3 The basics

In our discussion of abstractions, we mentioned the abstraction of the function/method/procedure/subroutine.

- The functional abstraction provides a means to represent control flow.

  In its pure version, every term in the λ-calculus is a function.

- In order for such a system to be at all useful, it must of course support higher-order functions; functions may be applied to functions.

- Values such as booleans and natural numbers are *encoded* (represented) by functions.

3.1 Informal definition of terms

The pure untyped λ-calculus has just three sort of terms;

- variables such as x, y, z,
- λ-*abstractions*, of the form $\lambda x \to t$,
  - (it is also common to use in place of $\to$; we prefer $\to$ as it emphasises that these are functions)
  - where x is a variable and t is a λ-term, and
- applications of the form tu
  - where t and u are λ-terms.

3.2 Informal meaning of terms

The meaning of each term is, informally:

- A λ-abstraction $\lambda x \to t$ represents a function of one argument, which, when applied to a term u, substitutes all free occurrences of x in t with u.
- An application applies the term u to the function (term) t.
- A variable on its own (a free variable) has no further meaning.
  - Variables are intended to be *bound*.
  - "Top-level" free variables have no meaning (on their own).
    - Until we construct a new term by λ-abstracting them.

3.3 Variable binding

Recall the notion of free and bound variables.

- A *variable binder* is an operator which operates on some number of *variables* as well as *terms*.
  - Examples include quantifiers such as $\forall \_ \mid \_ \bullet \_$, $\exists \_ \mid \_ \bullet \_$ and $\sum \_ \mid \_ \bullet \_$, and substitution $\_[\_ \to \_]$.
  - By convention, the bodies of variable binders extend as far to the right as possible;
    - so for instance $\forall x \mid Px \bullet Qx \land Ry$ is read as $(\forall x \mid Px \bullet (Qx \land Ry))$.
  - But substitution binds tighter than any other operation;
    - so for instance $x + y[y \coloneqq z]$ is read as $x + (y[y \coloneqq z])$

3.4 Free and bound variables

For simplicity, let us assume here that variable binders act on a single variable and a single term.

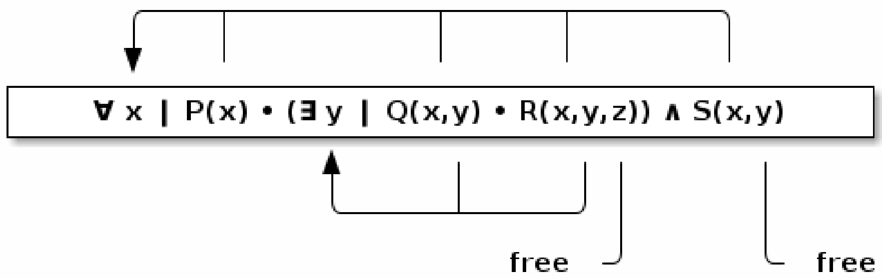
- Let $B\_ \bullet \_$ range over the set of variable binders in a language.
- An occurrence of a variable x in a term t that is *not* in a subterm of the form $Bx \bullet u$ is called *free*.
- In a term t with a subterm of the form $Bx \bullet u$, all free occurrences of the variable x that occur within u are *bound* by that instance of the binder B.
  - Note: instances of x which are bound elsewhere are not bound by that B.

3.5 Open and closed terms; combinators

- A λ-term which contains free variables is called an *open term*.
- A λ-term with no free variables is called a *closed term*.
  - Such λ-terms are also called *combinators*.

3.6 Picturing variable bindings

For instance, in the language of predicate logic, we can view the variables bound like so.



3.7 Representing functions with multiple arguments

You may have noticed that our method for constructing function in the λ-calculus (the λ-abstraction) only allows us to construct single-argument functions.

- That is, we do not have terms such as $\lambda(x, y) \to t$.
- This may seem restrictive,
- but it turns out to be sufficient. And it keeps the language simpler theoretically.

3.8 Currying

Rather than complicating our set of terms by admitting functions of multiple arguments, we use the technique of *currying* functions.

- Consider a function $f : A \times B \to C$.
- We can substitute a new function $f' : A \to (B \to C)$ for f.
  - (By convention, function arrows associate to the right, so this is equivalent to $f : A \to B \to C$.)
  - So f' is a function which takes an A and *produces a function* of type $B \to C$.

  * We also say that f' is *partially applied* to a value of A.
  * We usually don't give this new function a name.
  * We can consider this new function as having a *fixed* value for the A argument that was provided.
  * (So we must be able to represent higher-order functions to use Currying.)

3.9 Examples of λ-terms

λ `x` $\to$ `x`

is a familiar function; it is the *identity* function. We will use the name `id` to refer to this function.

λ `x` $\to$ λ `y` $\to$ `x`

is a function which ignores its second argument, and just returns the first; this is sometimes called `const`.

λ `x` $\to$ λ `y` $\to$ `y`

is then a function which ignores its first argument.

λ `f` $\to$ λ `x` $\to$ `f x`

is a function which applies its second argument to its first; we might call this just `apply`.

4 The syntax and semantics of *UL*

We now discuss the formal semantics of the untyped λ-calculus; that is, we

- give a grammar for its syntax, and
- define operational semantics for the language.

4.1 A grammar for *UL*

$\langle$`term`$\rangle$ ::= `var` | λ `var` $\to$ $\langle$`term`$\rangle$ | $\langle$`term`$\rangle$ $\langle$`term`$\rangle$

In the case that we are restricted to ASCII characters, we will write abstraction as

`"lambda" var .` $\langle$`term`$\rangle$

4.2 The operational semantics of *UL*

A term of the form $(\lambda x \to t_1)t_2$ is called a *redex* (β-redex), meaning *reducible expression.*

The semantics of the λ-calculus is given by a *reduction strategy* (β-reduction strategy);

- A reduction (β-reduction) transforms a subterm of the form
  - $(\lambda x \to t_1)t_2$ (a redex) to
  - $t_1[x \coloneqq t_2]$.
    - (There are various syntactic representations of substitutions; we prefer to the substitution operation to come after the term where the substitution is carried out (t_1), and to use the "becomes" operator to imply free instance of x become t_2.
    - Pierce instead uses the form $[x \mapsto t_2]t_1$, with the substitution operation coming before the term, and using the "maps to" operator instead of "becomes".
    - You may also see forms such as $[x \backslash t_1]$ or $[t_1/x]$.)

4.3 Normal forms and values

A term which does not involve any redexes is said to be in *normal form* (β-normal form).

- Terms in β-normal form which are not variables are called *values.*
  - In the pure untyped λ-calculus, these only include λ-abstractions.
  - Later, we will add other constant values, such as `true`, `false`, `0`, etc.

In the untyped λ-calculus,

- if a term has a normal form, that normal form is unique.
  - (By the *Church–Rosser* theorem.)
- But not all terms have a normal form!

4.4 Some reduction strategies

Given an arbitrary term, there may be several subterms which are redexes,

- so we have a choice of what subterm to reduce.

A reduction strategy limits our choice of which redex to reduce.

Several strategies have been studied. We discuss just four of them.

- full β-reduction, normal order,
- call by name, and call by value.

We only give a full formal treatment to call-by-value.

The last two you may know as names of parameter passing methods from (practical) programming languages.

- There is a direct correspondance between reduction strategies and parameter passing methods.

4.5 Some reduction strategies: full β-reduction and normal order

The *full β-reduction* strategy is, essentially, to have no strategy at all.

- Under full β-reduction, and redex can be reduced at any point.
- This strategy gives rise to a reduction *relation*, not a function.
  - Since a given term may reduce to *many* other terms.

The *normal order* strategy enforces that the leftmost, outermost redex is always reduced first.

- This restriction gives rise to a function.

4.6 Some reduction strategies: call by name and call by value

The *call by name* strategy builds on the normal order strategy

- by mandating that no reductions take place inside abstractions.
- So "arguments cannot be evaluated before being applied".

The *call by value* strategy also builds on the normal order strategy,

- by mandating that a redex is reduced only when its right hand side
  - (the "argument")

  is a value (in β-normal form and not a variable).

4.7 A formal description of call by value semantics

Let us use the convention that variable names involving

- `t` represent arbitrary λ-terms, whereas variable names involving
- `v` represent terms in λ-normal form (values).

Then we may give a formal description of call-by-value semantics using inference rules.

$$\frac{\mathtt{t_1} \longrightarrow \mathtt{t_1}}{\mathtt{t_1}\ \mathtt{t_2} \longrightarrow \mathtt{t_1}\ \ \mathtt{t_2}}\ \mathtt{App}^l$$

$$\frac{\mathtt{t_2} \longrightarrow \mathtt{t_2}}{\mathtt{v_1}\ \mathtt{t_2} \longrightarrow \mathtt{v_1}\ \mathtt{t_2}}\ \mathtt{App}^r$$

$$\frac{}{(\lambda\ \mathtt{x} \rightarrow \mathtt{t})\ \mathtt{v} \longrightarrow \mathtt{t[x} \coloneqq \mathtt{v]}}\ \mathtt{AppAbs}$$

Notice how the use of `t`'s and `v`'s mandates that

- terms on the left reduce first, and
- applications only take place when the term being applied is a value.

4.8 β-reduction, α-equivalence and η-conversion

β-reduction gives us one way to equate terms;

- two terms "have the same value" if they both reduce to the same value (irreducible term.)
- So we call terms that reduce to the same value β-equivalent.
  - For instance, $(\lambda x \rightarrow x)y =_\beta y$.

Two other notions of equality between λ-terms prove useful.

- α-equivalence stipulates that two terms which vary only in the naming of bound variables are equivalent.

- For instance, $\lambda x \to x =_\alpha \lambda y \to y$.
- This is a very useful stipulation to help avoid name clashes!

- η-conversion stipulates that
  - a term of the form $\lambda x \to f x$ can be reduced to f, (η-reduction) and conversely,
  - a term of the form f can be expanded to $\lambda x \to f x$ (η-expansion.)

4.9 Strong and weak normalisation

As we've said, a λ-term is said to be in *normal form* if it cannot be reduced.

- We can define this concept of normal form in any system in which terms reduce;
  - in particular, in all the other models of computation we will consider.

A set of terms along with a reduction strategy is then called

- *strongly normalising* if every reduction sequence is guaranteed to terminate in a normal form, and
- *weakly normalising* if for every term, there is at least one reduction sequence which terminates in a normal form.

4.10 Exercise: a term with no normal form

One combinator (closed term) of the untyped λ-calculus is the ω-*combinator*, which is also called the *divergent* combinator.

```
omega = (λ x → x x) (λ x → x x)
```

This combinator has no normal form; can you prove that?
Hint: what reductions are possible here? What is the result of that reduction?

5 λ-encodings

As mentioned previously, in the pure λ-calculus, every term is a function.

- There are no basic types of data.

So, we must have a way of representing any data as a function.

- We call these Church encodings.

We will show how to do this for

- booleans,
- pairs, and
- natural numbers,

and give some "combinators" which operate on these kinds of data.

5.1 Church booleans

We define the following terms to represent boolean values.

- `tru` represents truth, and
- `fls` represents false.

```
tru = λ t → λ f → t
fls = λ t → λ f → f
```

These choices are *somewhat* arbitrary.

- We could choose any two distinct λ-terms.
- But they are not really arbitrary; these two terms embody the idea that a boolean value is a "choice" between two options.
  - `tru`, when given two arguments, "chooses" the first.
  - `fls`, when given two arguments, "chooses" the second.

5.2 Defining `if-then-else` using Church booleans

Since the Church encoded booleans already "perform" a choice, defining an "if-then-else" construct using them is quite straightforward.

```
test = λ l → λ m → λ n → l m n
```

The intention is that

- the first argument is a Church boolean,

- the second is the "then" branch, and
- the third is the "else" branch.

Notice that `test b v w` simply reduces to `b v w`;

- the boolean `b` really "does the work" of choosing between `v` and `w`.

5.3 Exercise: is `test` really if-then-else?

Let us briefly pause to consider the semantics of `test`,

- and see if it matches the behaviour we expect from an "if-then-else" construct.

Consider the example λ-term

```
test true (id true) (id false)
```

Using call-by-value semantics, we have

```
  test true (id true) (id false)
= test true ((λ x → x) (λ x → λ y → x)) ((λ x → x) (λ x →
↪  λ y → y))
⟶ test true (λ x → λ y → x) ((λ x → x) (λ x → λ y → y))
⟶ test true (λ x → λ y → x) (λ x → λ y → y)
= test true true false
⟶ …
```

Exercise: Considering this portion of the reduction sequence, what is different about `test` and the "if-then-else" construct that you are used to?

5.4 Pairs

We now give an encoding of pairs

- (a wrapping of two terms into one),
- along with pair "deconstructors".

These definitions rely upon the encoding of booleans we have just given.

```
pair = λ f → λ s → λ b → b f s
fst = λ p → p tru
snd = λ p → p fls
```

We may check that, for instance, `fst (pair v w)` will indeed reduce to `v`, using call-by-value semantics.

```
  fst (pair v w)
= (λ p → p (λ x → λ y → x)) ((λ f → λ s → λ b → b f s) v
 ↪  w)
⟶ (λ p → p (λ x → λ y → x)) ((       λ s → λ b → b v s)
 ↪  w)
⟶ (λ p → p (λ x → λ y → x)) ((             λ b → b v w))
⟶ (λ b → b v w) (λ x → λ y → x)
→ (λ x → λ y → x) v w
⟶ (λ y → v) w
→ v
```

5.5 Exercise: `snd`

As an exercise, you may confirm that `snd (pair v w)` reduces to `w`, using call-by-value semantics.

5.6 Natural numbers: Church numerals

To represent natural numbers is only slightly more complicated than booleans and pairs. We give the pattern

c_0 = λ s → λ z → z
c_1 = λ s → λ z → s z
c_2 = λ s → λ z → s (s z)
...

That is, each numeral `n` is represented as the function which applies its first argument to its second argument `n` times.

Or more neatly, we define

```
zero = λ s → λ z → z
scc = λ n → λ s → λ z → s (n s z)
```

so then c_0 is `zero`, c_1 can be obtained from `scc zero` (by reducing it), c_2 can be obtained from `scc (scc zero)`, etc.

5.7 Addition and multiplication

By using the fact that

- "each numeral `n` is represented as the function which applies its first argument to its second argument `n` times",

we can fairly easily define addition and multiplication.

For addition, `m + n`,

- we begin with `n`,
- and apply `suc` `m`-many times.

```
plus = λ m → λ n → λ s → λ z → m s (n s z)
```

For multiplication, `m * n`,

- we begin with `zero`,
- and apply "`plus n`" `m`-many times.

```
times = λ m → λ n → m (plus n) zero
```

5.8 Testing for zero

In order to test if a natural number is zero, we use the same ideas,

- but now the base case is true,
- and the function we apply `m`-many times is just the constantly false function.

```
iszro = λ m → m (λ x → fls) tru
```

6 Recursion: the fixed point combinator

We have, in the previous section, encoded booleans, pairs and natural numbers in the untyped λ-calculus.

In the process,

- we defined a "control structure" combinator `test = λ l → λ m → λ n → l m n` which acts something like `if-then-else`.
- we defined functions for deconstructing pairs, `fst` and `snd`,
- and for operating on natural numbers: `scc`, `plus`, `times` and `iszro`.

But we are still lacking in "easy" ways to define new functions.

- The way we define those functions relies heavily on the encoding of the data.
- We perhaps cannot make it truly “easy” in this limited language,
- but we can get “easier”.

6.1 The ω-omega combinator: unbounded recursion

During our discussion of normal forms, we mention the “ω-combinator”, which embodies *divergence* (non-termination).

```
omega = (λ x → x x) (λ x → x x)
```

`omega` has one redex, and reducing it results in `omega` once more.

- So `omega` has no normal form, because no reduction sequence for `omega` terminates.

A generalisation of the `omega` combinator will let us define recursive functions.

6.2 The fixed-point combinator, a.k.a. the Y-combinator

The *fixed-point combinator*, or the (call-by-value) *Y-combinator*, has the form

```
fix = λ f → (λ x → f (λ y → x x y)) (λ x → f (λ y → x x
↪ y))
```

:TODO: prove that this gives rise to a fixed point

6.3 Recursive definitions via the fixed point combinator

To use `fix`, we define a function `g` of the form

```
g = λ f → ...
```

and use `f` as a *recursive call*.

Then we apply `fix g`, which computes a recursive function whose right-hand side is given by `g`.

See Pierce, chapter 5, page 66 for an example involving a definition of factorial in this manner.

7 Enriching the calculus

We may "enrich" our untyped λ-calculus

- first by adding additional values for types such as booleans and natural numbers,
  - values which are simply new constants, and not encodings as pure untyped functions,
- and by then adding a (simple) type system to obtain a (simply) typed λ-calculus.

We will do both of these in section 6 of the notes, "A typed λ-calculus, TL".