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

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.

Imperative languages instead 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 The 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.

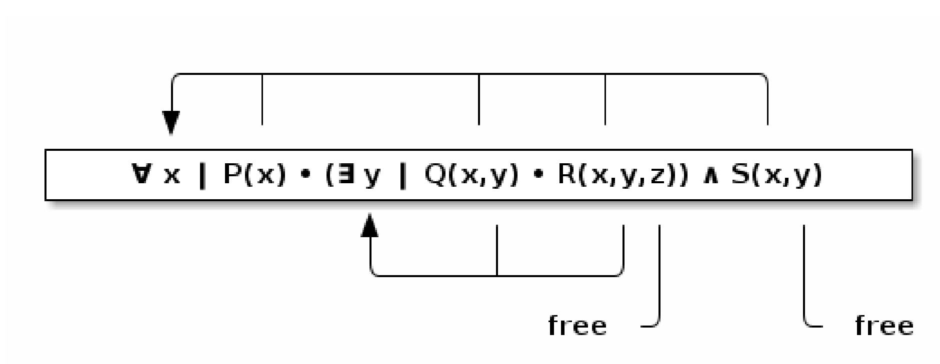
3.3 Variable binding; free and bound variables

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 $\_[\_ \to \_]$.
- (For simplicity, let us assume below 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.

3.4 Picturing variable bindings

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



3.5 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.

3.6 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 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.)

3.7 Examples of λ-terms

```
λ x → x
```

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

```
λ x → λ y → x
```

```
λ x → λ y → y
```

4 The formal 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` $\rightarrow$ $\langle$`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 $(\lambda x \rightarrow t_1)t_2$ is called a *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
  - $(\lambda x \rightarrow t_1)t_2$ (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 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.

4.4 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.

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

4.5 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")

  cannot be reduced (is a value.)

4.6 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 irreducible λ-terms (values).

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

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1}{\mathtt{t}_1\ \mathtt{t}_2 \longrightarrow \mathtt{t}_1\ \ \mathtt{t}_2}\ \mathtt{Application}^l$$

$$\frac{\mathtt{t}_2 \longrightarrow \mathtt{t}_2}{\mathtt{v}_1\ \mathtt{t}_2 \longrightarrow \mathtt{v}_1\ \mathtt{t}_2}\ \mathtt{Application}^r$$

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

4.7 α-conversion and η-conversion

:TODO:

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.)

:TODO:

4.8 Normalisation

A λ-term is said to be in *normal form* if it cannot be reduced.

:TODO:

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 Pairs

:TODO:

5.4 Church numerals

:TODO:

5.5 Addition and multiplication

:TODO:

6 Enriching the calculus

7