

# Assignment #1: Untyped $\lambda$ -Calculus

Roger Wolff, for 15-312: Principles of Programming Languages

Out: Wednesday, January 18st  
Due: Tuesday, January 24rd 11:59pm

Welcome to 15-312! First things first. We will be using Piazza for all class communications. If you have already received a welcome e-mail, there is nothing more you need to do. If not, please subscribe post-haste at <http://piazza.com/class#spring2012/15312>.

Go to the course web page to understand the whiteboard policy for collaboration regarding the homework assignments, the late policy regarding timeliness of homework submissions, and the use of Piazza.

Homework will typically consist of a theoretical section and an implementation section. For the first assignment, there is only the theoretical section. You are required to typeset your answers; see the course Web page for some guidance.

In this first assignment we are asking you to practice proving theorems by rule induction. You may find this assignment difficult. Start early, and ask us for help if you get stuck! In particular, you are encouraged to ask the TAs for help over Piazza, and/or come to office hours.

## Submission

To submit your solutions place a file named `assn1.pdf` in your handin directory:

```
/afs/andrew.cmu.edu/course/15/312/handin/<yourandrewid>/assn1/
```

## 1 Hello, $\lambda$ -calculus!

At first blush, it is hard to believe that much of anything can be computed with the untyped  $\lambda$ -calculus ( $\mathcal{L}\{\lambda\}$ ), let alone believe that it is Turing complete. In this assignment, we will practice programming in this language by learning how to add numbers.

In class, we began by representing numbers by a Church encoding called Church numerals. This represents numbers in unary. As computer science majors, we know that computing with unary numbers often leads to an exponential slowdown in computational power. Therefore, we will represent numbers by a binary Church encoding. We will first represent a single bit, then build an arbitrary length natural number by stringing bits together, and finally we will be able to show how to do some real work – adding. Let's get started.

## 1.1 Bits

We first want to represent a single bit.  $\mathcal{L}\{\lambda\}$  has no primitive values other than functions, so we need to choose one function to represent 0, and another to represent 1. Since a bit can only represent one of two values, we can use the Church booleans, as presented in class.

$$\begin{aligned}\bar{0} &= \text{false} = \lambda(-)\lambda(f)f \\ \bar{1} &= \text{true} = \lambda(t)\lambda(-)t\end{aligned}$$

We can now compute  $\text{if}(b) \ t \ \text{else} \ f$ , by  $b \ t \ f$ . It is easy to see that  $\bar{0} \ t \ f \equiv f$  and  $\bar{1} \ t \ f \equiv t$ .

We use the overline ( $\bar{\phantom{x}}$ ) to signify that we are talking about this encoding of a bit, rather than the mathematical number (1).

The bit operators NOT( $\neg$ ), AND ( $\wedge$ ), OR ( $\vee$ ) and XOR( $\oplus$ ) can be defined as:

$$\begin{aligned}\neg b &= b \ \bar{0} \ \bar{1} \\ x \wedge y &= x \ y \ \bar{0} \\ x \vee y &= x \ \bar{1} \ y \\ x \oplus y &= x \ (\neg y) \ (y)\end{aligned}$$

## 1.2 Ordered Pairs

To make our life easier, we will now take a seemingly unrelated diversion to introduce ordered pairs. We have introduced ordered pairs in class. As a reminder, ordered pairs can be represented in  $\mathcal{L}\{\lambda\}$  by the following expressions

$$\begin{aligned}\langle x, y \rangle &= \lambda(f) f \ x \ y \\ e \cdot l &= e \ (\lambda(x) \ \lambda(-) \ x) \\ e \cdot r &= e \ (\lambda(-) \ \lambda(y) \ y)\end{aligned}$$

where  $\langle x, y \rangle$  represents the ordered pair  $(x, y)$ , and  $e \cdot l$  represents the first of the pair, and  $e \cdot r$  represents the second.

**Task 1.1** [6 points]:

Prove that  $\langle x, y \rangle \cdot l \equiv x$ , by providing a derivation.

**Task 1.2** [4 points]:

Use ordered pairs to create an ordered triple. That is, give  $\mathcal{L}\{\lambda\}$  terms that represent  $\langle a, b, c \rangle$ , and  $e \cdot 1, e \cdot 2$ , and  $e \cdot 3$ .

### 1.3 Lists

In order to represent an arbitrary length number, we will do so by using a list of bits. The good news is that we have created bits. The bad news is that we need to build lists. Let's do that now.

The elimination form (how to make use of lists) will be an implementation of fold (foldr in ML). This is an example of the visitor pattern, where we will give a list two arguments: a base case,  $b$ , of what to do with an empty list, and an aggregator function,  $f$ , of what to do with a non-empty list. The function  $f$  will in turn, take two arguments: the head of the list, and the result of the fold operation on the tail. Lists will be defined as follows:

$$\begin{aligned} \text{Nil} &= \lambda(b) \lambda(f) b \\ h :: t &= \lambda(b) \lambda(f) f h (t b f) \\ [e] &= e :: \text{Nil} \end{aligned}$$

$\text{Nil}$  is the empty list, and when applied with the base and the aggregator function, simply returns the base.  $\text{cons} (::)$  is a function that accepts a head element and a tail list, and produces a list, which is a function that when applied with two arguments will perform a fold on itself as described above. For convenience,  $[e]$  will construct a list of length 1 consisting of element  $e$ .

Let's define some typical functions that can be performed on a list.

$$\text{len} = \lambda(l) l \bar{0} (\lambda(-) \lambda(b) \neg b)$$

$\text{len}$  is a function that returns the length of a list. Of course, since we are using lists to build numbers,  $\text{len}$  cannot return a number. Instead, it will return a bit representing the length of the list mod 2. The base case is  $\bar{0}$ , as the empty list has length 0. The aggregator function is a bitwise increment, which without overflow is simply negation.

#### Task 1.3 [6 points]:

Some more useful functions for our purposes are  $\text{concat}(x, y)$ , abbreviated  $x@y$ , which concatenates two lists,  $x$  and  $y$ , and  $\text{rev}(x)$ , which reverses list  $x$ . Write the  $\mathcal{L}\{\lambda\}$  expressions for  $x@y$ , and  $\text{rev}(x)$ . Make sure to explain how your code works.

### 1.4 Binary Numbers

Now that we have bits, products, and lists, we are finally ready to do some real work, adding. We will encode binary (natural) numbers as a list of bits. Numbers will be encoded with the most significant bit (msb) first, at the head of the list. We write  $\overline{n}_2$  to mean this binary encoding of the number  $n$ . We will represent  $\overline{0}_2$  by the empty list. A non-zero number will be kept in a canonical form where there are no leading zero bits. For example,  $\overline{2}_2 \equiv [\bar{1}]@[\bar{0}]$ .

Starting off simply, let's build a function that will add a single bit,  $b$ , to a number,  $n$ .

$$\begin{aligned} \text{addbit}' &= \lambda(n) \lambda(b) n \langle \text{Nil}, b \rangle (\lambda(x) \lambda(e) \langle (x \oplus e \cdot r) :: (e \cdot l), x \wedge e \cdot r \rangle) \\ \text{addbit} &= \lambda(n) \lambda(b) ((\text{addbit}' n b) \cdot r) (\bar{1} :: ((\text{addbit}' n b) \cdot l))((\text{addbit}' n b) \cdot l) \end{aligned}$$

`addbit'` does the real work here, computing a product. As the computation progresses, the left projection will be computed to be the lower  $i$  bits of  $(n+1)_2$ , where  $i$  is the number of bits that have already been processed. The right projection will be the carry bit. `addbit` uses `addbit'` to do the work, and then optionally prepends a  $\bar{1}$  if a carry is needed, otherwise just reads off the answer.

**Task 1.4** [15 points]:

Compute the sum of two arbitrary length natural numbers. As before, you are free to use the constructs and functions presented so far, and are encouraged to build helper functions as needed. Be sure to document your code.

## 2 Parallel Dynamic Semantics for $\lambda$ -calculus

The dynamics of  $\mathcal{L}\{\lambda\}$  has been presented equationally in class (see Appendix A.3 ). We can write any expression we like, but we are at a loss actually how to go about proceeding with a computation. Of particular worry is the ( $\equiv$ SYM) rule. In order to make use of this rule, we need to guess at the  $u'$  before being able to make progress.

We would like to get rid of this rule and provide a more directed way of computing with  $\mathcal{L}\{\lambda\}$  terms. The following rules define a parallel dynamics.

$$\frac{}{M \Rightarrow M} (\Rightarrow \text{REF}) \qquad \frac{M \Rightarrow M' \quad N \Rightarrow N'}{M N \Rightarrow M' N'} (\Rightarrow \text{APP})$$

$$\frac{M \Rightarrow M'}{\lambda(x.M) \Rightarrow \lambda(x.M')} (\Rightarrow \lambda) \qquad \frac{M \Rightarrow M' \quad N \Rightarrow N'}{\lambda(x.M) N \Rightarrow [N'/x]M'} (\Rightarrow \beta)$$

The only real work that can be done is a  $\beta$ -reduction ( $\Rightarrow \beta$ ), but these dynamics allow us to perform as many reductions as we like in one step. The reflexive rule ( $\Rightarrow$  REF) is needed only to simplify the dynamics, so that other parallel rules like application ( $\Rightarrow$ APP) can make progress if only one of the two subexpressions can do work.

Let's take a look at a couple of examples.

$$\underline{((\lambda(x) x) (\lambda(y) y y))} \underline{((\lambda(z) z) w)} \Rightarrow \underline{(\lambda(y) y y) w} \Rightarrow w w$$

We underline subexpressions that we work on in parallel. In this example, we use ( $\Rightarrow$ APP) to reduce both sides of an application. Observe that it is not possible to perform any additional parallelism due to the inherent dependency of this calculation. We know the first term will eventually evaluate to a function (after all, there is nothing else in the language), but we don't know what the function will look like. We cannot perform an additional  $\beta$ -reduction until that term has been evaluated enough to reveal its structure. Sequential execution can clearly be seen here as a specific instance of parallelism, where the use of ( $\Rightarrow$  REF) can ensure that at most one subexpression is making progress at any given time. Taking the same original expression, we

can serialize the computation if we wish.

$$((\lambda(x) x) (\lambda(y) y y)) ((\lambda(z) z) w) \Rightarrow ((\lambda(x) x) (\lambda(y) y y)) (w) \Rightarrow (\lambda(y) y y) w \Rightarrow w w$$

**Task 2.1** [5 points]:

Using as much parallelism as possible, how many steps of computation are necessary to reduce the following expression to its *normal form*, one that cannot be further reduced to a different expression.

$$(\lambda(x) ((\lambda(q) q q) x)) ((\lambda(z) z) w)$$

For each step,  $M \Rightarrow N$ , provide a derivation legitimating that reduction.

As a first semblance of correctness for our transition system, we will prove a lemma showing that substitution preserves the  $\Rightarrow$  property.

Formally, substitution in  $\mathcal{L}\{\lambda\}$  is defined as

$$\begin{aligned} [N/x]x &= N \\ [N/x]y &= y \\ [N/x]\lambda(y) M &= \lambda(y) [N/x]M \\ [N/x](M_1 M_2) &= [N/x]M_1 [N/x]M_2 \end{aligned}$$

**Lemma 1.** (*Substitution*) If  $M \Rightarrow M'$  and  $N \Rightarrow N'$ , then  $[N/x]M \Rightarrow [N'/x]M'$ .

**Task 2.2** [12 points]:

Prove Lemma 1 by induction on the form of  $M$  and the derivation of  $M \Rightarrow M'$ . There are 5 cases to consider. There is one case for each of the rules of  $\Rightarrow$ , with the exception of ( $\Rightarrow REF$ ), which requires two cases — one for  $M = M' = x$ , and one for  $M = M' = y$ .

For example, let's take ( $\Rightarrow APP$ ), where the last step in the derivation  $M_1 \Rightarrow M'_1$  and  $M_2 \Rightarrow M'_2$  are the premises, and  $M_1 M_2 \Rightarrow M'_1 M'_2$  is the conclusion. Here we are saying that  $M = M_1 M_2$ . We are also given  $N \Rightarrow N'$ , and we wish to prove that  $[N/x](M_1 M_2) \Rightarrow [N'/x](M'_1 M'_2)$ .

1. $M_1 \Rightarrow M'_1$	given
2. $M_2 \Rightarrow M'_2$	given
3. $N \Rightarrow N'$	given
4. $[N/x]M_1 \Rightarrow [N'/x]M'_1$	Induction Hypothesis on 1,3
5. $[N/x]M_2 \Rightarrow [N'/x]M'_2$	Induction Hypothesis on 2,3
6. $[N/x]M_1 [N/x]M_2 \Rightarrow [N'/x]M'_1 [N/x]M'_2$	( $\Rightarrow$ APP) and 4,5
7. $[N/x]M_1 [N/x]M_2 = [N/x](M_1 M_2)$	definition of substitution
8. $[N'/x]M'_1 [N'/x]M'_2 = [N'/x](M'_1 M'_2)$	definition of substitution
9. $[N/x](M_1 M_2) \Rightarrow [N'/x](M'_1 M'_2)$	6,7,8

Eventually, we will prove that the two dynamics, those defined by  $\Rightarrow$  and those defined by  $\equiv$  are equivalent. We will make this more precise later. For now, we will prove a useful lemma that begins to relate the two relations.

**Lemma 2.** *If  $M \Rightarrow N$  then  $M \equiv N$ .*

**Task 2.3** [8 points]:

Prove Lemma 2 by induction on the derivation of  $M \Rightarrow N$ . Remember to show cases for each of the four rules of  $\Rightarrow$ .

Our transition system is nondeterministic. As previously shown, there are expressions that may take one of several different transitions. We hope that our language is unambiguous, in that multiple executions will ultimately resolve to the same result. We will prove this later. For now, we will show that our language has the *diamond* property.

**Definition 1.**  $N \Downarrow P$  is the relation that  $N$  and  $P$  can transition to the same expression in one step. Formally,  $N \Downarrow P = \exists Q. N \Rightarrow Q \wedge P \Rightarrow Q$ .

**Lemma 3.** (*Diamond*) *If  $M \Rightarrow N$  and  $M \Rightarrow P$ , then  $N \Downarrow P$ .*

**Task 2.4** [15 points]:

Prove Lemma 3 by induction on the derivations of  $M \Rightarrow N$  and  $M \Rightarrow P$ .

There will be 6 cases to consider. Let's do a case analysis on the form of  $M$ . If  $M = x$  for some variable  $x$ , there is only one reduction that is applicable, namely  $x \Rightarrow x$ . Similarly, if  $M = \lambda(x) M'$  there is also only one reduction available,  $\lambda(x) M' \Rightarrow \lambda(x) M''$ , where  $M' \Rightarrow M''$ .

But if  $M$  is an application, then it may take one of two reductions, from rules ( $\Rightarrow$  APP) or ( $\Rightarrow \beta$ ). And since  $M$  reduces to  $N$  via one of two rules, and  $M$  reduces to  $P$  by one of two rules, there are 4 cases to consider. So you need to consider the cases where  $M$  reduces to both  $N$  and  $P$  by ( $\Rightarrow$  APP), by ( $\Rightarrow \beta$ ), and where  $M$  reduces to  $N$  by ( $\Rightarrow$  APP) and to  $P$  by ( $\Rightarrow \beta$ ). The fourth case, where where  $M$  reduces to  $N$  by ( $\Rightarrow \beta$ ) and to  $P$  by ( $\Rightarrow$  APP) is, by symmetry, equivalent to the previous one.

Due to dependencies, we cannot always compute the normal form of an expression in one parallel step. If we wish to do multiple parallel steps of work, we will denote it as  $\Rightarrow^*$ , the transitive closure of  $\Rightarrow$ . If

$M \Rightarrow^* N$ , we say that  $N$  is a *reduct* of  $M$ . Equivalently, we may say that  $M$  is *reducible* to  $N$ .

Formally, we can define  $\Rightarrow^*$  as

$$\frac{}{a \Rightarrow^* a} \text{ (*INIT)} \qquad \frac{a \Rightarrow b \quad b \Rightarrow^* c}{a \Rightarrow^* c} \text{ (*HEAD)}$$

**Definition 2.**  $N \Downarrow^* P$  is the relation that  $N$  and  $P$  have a common reduct. Formally,  $N \Downarrow^* P = \exists Q. N \Rightarrow^* Q \wedge P \Rightarrow^* Q$ .

Ultimately, we will prove that  $\equiv$  and  $\Downarrow^*$  in fact represent the same relation. We will need to first prove one more lemma first.

**Lemma 4. (Confluence)** If  $M \Rightarrow^* N$  and  $M \Rightarrow^* P$ , then  $N \Downarrow^* P$ .

Confluence is a desirable property to have in a language. In the dynamic semantics we have defined, we can have one term for which there are multiple transitions. This results in an indeterminate computation. What *confluence* tells us is that if an expression can split (transition in multiple ways), at least each of those ways is able to eventually join (transition to the same reduct).

As a corollary, this implies that if an expression reduces to a value (an expression that cannot be reduced anymore, a number say), then no matter how the execution proceeds, it will always result in the same answer. Formally, if  $M \Rightarrow^* N \not\Rightarrow^* Q$  and  $M \Rightarrow^* P \not\Rightarrow^* Q'$ , for any  $Q, Q'$  different from  $N, P$ , then by the confluence lemma,  $N = P$ . In other words, the execution may be ambiguous, but the result is not.

**Task 2.5** [4 points]:

Use the Lemma 3(diamond) to prove Lemma 4 (confluence).

Finally, we are ready to prove the equivalency between  $\equiv$  and  $\Downarrow^*$ .

**Theorem 1.**  $M \equiv N$  iff  $M \Downarrow^* N$

**Task 2.6** [6 points]:

Prove Theorem 1. The left to right direction can be proved by induction on the derivation of  $M \equiv N$ . The right to left direction can be proved by appealing to Lemma 2.

Let's recap what we've done. The equational dynamics of  $\mathcal{L}\{\lambda\}$  do not lend themselves to an executional semantics with  $\lambda$  terms. That is,  $\mathcal{L}\{\lambda\}$  may be powerful enough for use to prove that  $\overline{1_2} + \overline{1_2} \equiv \overline{2_2}$ , but they don't tell us how to compute the latter expression from the former. It may be the case that we would need to prove  $\overline{1_2} + \overline{1_2} \equiv \overline{0_2} + \overline{2_2}$  and that  $\overline{0_2} + \overline{2_2} \equiv \overline{2_2}$ , and derive our result from the transitive property of  $\equiv$ . In other words, the equational semantics require mathematical proofs.

On the other hand, we have developed a transitional dynamic semantics which is purely mechanical. It has non-determinism built in, but this is a feature, not a necessity. We can choose any arbitrary execution strategy we like, and reducing an expression takes on a purely mechanical form.

But by doing this, we have not lost any expressiveness! In a very precise sense, the two dynamics are equivalent. If only all of mathematics were this easy.

By no means was this a trivial feat. And this was only week 1. So pat yourselves on the back for a job well done. And then get ready for some more. Welcome again to 15-312. We hope you have a great semester.

## A $\lambda$ -calculus

### A.1 Syntax

Expression  $u ::= x$  variable  
 $\lambda(x) u$   $\lambda$ -abstraction  
 $u_1 u_2$  application

### A.2 Statics

$$\frac{}{\Gamma, x \text{ ok} \vdash x \text{ ok}} \text{ (VAR)} \quad \frac{\Gamma \vdash u_1 \text{ ok} \quad \Gamma \vdash u_2 \text{ ok}}{\Gamma \vdash u_1 u_2 \text{ ok}} \text{ (APP)} \quad \frac{\Gamma, x \text{ ok} \vdash u \text{ ok}}{\Gamma \vdash \lambda(x.u) \text{ ok}} \text{ (\lambda)}$$

### A.3 Equational Dynamics

$$\frac{}{\Gamma, u \text{ ok} \vdash u \equiv u} \text{ (\equiv REF)} \quad \frac{\Gamma, x \text{ ok} \vdash u \equiv u'}{\Gamma \vdash \lambda(x.u) \equiv \lambda(x.u')} \text{ (\equiv \lambda)} \quad \frac{\Gamma \vdash u \equiv u' \quad \Gamma \vdash u' \equiv u''}{\Gamma \vdash u \equiv u''} \text{ (\equiv TRANS)}$$

$$\frac{\Gamma \vdash u' \equiv u}{\Gamma \vdash u \equiv u'} \text{ (\equiv SYM)} \quad \frac{\Gamma \vdash e_1 \equiv e'_1 \quad \Gamma \vdash e_2 \equiv e'_2}{\Gamma \vdash e_1 e_2 \equiv e'_1 e'_2} \text{ (\equiv APP)} \quad \frac{\Gamma, x \text{ ok} \vdash e_2 \text{ ok} \quad \Gamma \vdash e_1 \text{ ok}}{\Gamma \vdash \lambda(x.e_2) e_1 \equiv [e_1/x]e_2} \text{ (\equiv \beta)}$$