Announcements

Homework 1 is graded. Pick it up from Cheryl Hickey (Levine 502).

We paid careful attention to problems 2 and 4.

Solutions to all problems are on the web page.

Pick up from Cheryl Hickey (Levine 502).

Homework 1 is graded.

The Lambda Calculus

If our previous language of arithmetic expressions was the simplest non-trivial programming language, then the lambda-calculus is the simplest interesting programming language.

It is the foundation of many real-world programming language designs.

The e. coly of programming language research

The lambda-calculus

The foundation of many real-world programming language designs

The e. coly of programming language research

main new feature: variable binding and lexical scope

higher order (functions as data)

Turin complete

Turing complete

Church-Turing thesis

If our previous language of arithmetic expressions was the simplest non-trivial programming language, then the lambda-calculus is the simplest interesting programming language...

The Lambda Calculus

Should be already reading TAPL, chapter 2.

Homework 2 is due at noon.

Their office hours.

If you have questions or can't read the comments see the TA during their office hours.

Solutions to all problems are on the web page.

We paid careful attention to problems 2 and 4.

Pick it up from Cheryl Hickey (Levine 502).

Homework 1 is graded.
Intuitions

Suppose we want to describe a function that adds three to any number we pass it. We might write:\n\[ \text{plus3} \times = \text{succ} \left( \text{succ} \left( \text{succ} \times \right) \right) \]\n
That is, \text{plus3} \times is \text{succ} \left( \text{succ} \left( \text{succ} \times \right) \right).

Q: What is \text{plus3} \times itself?
A: \text{plus3} \times is the function that, given \times, yields \text{succ} \left( \text{succ} \left( \text{succ} \times \right) \right).

\text{plus3} \times = \left( \text{succ} \left( \text{succ} \left( \text{succ} \times \right) \right) \right)

This function exists independent of the name \text{plus3} \times.

On this view, \text{plus3} \left( \text{succ} \times \right) \text{ is just a convenient shorthand for the function that, given } \times, \text{ yields } \text{succ} \left( \text{succ} \left( \text{succ} \times \right) \right), \text{ applied to } \text{succ} \times.

\text{plus3} \left( \text{succ} \times \right) = \left( \left( \text{succ} \left( \text{succ} \left( \text{succ} \times \right) \right) \right) \right) \text{ succ} \times
Intuitions

Suppose we want to describe a function that adds three to any number we pass it. We might write
\[\text{plus3} x = \text{succ}(\text{succ}(\text{succ}x))\]

That is, \(\text{plus3} x\) is \(\text{succ}(\text{succ}(\text{succ}x))\).

Q: What is \(\text{plus3}\)?
A: \(\text{plus3}\) is the function that, given \(x\), yields \(\text{succ}(\text{succ}(\text{succ}x))\).

This function exists independently of the name \(\text{plus3}\).

If we apply \(\text{plus3}\) to an argument like \(\text{plus3} \), the substitution rule yields a non-trivial computation:
\[\text{plus3} (\text{plus3}x) = (\text{succ}(\text{succ}(\text{succ}x)))(\text{succ}(\text{succ}(\text{succ}x))))\]

i.e., \(\text{plus3} (\text{plus3}x)\) is just a convenient shorthand for the function that, when applied to \(x\), yields \(\text{succ}(\text{succ}(\text{succ}x))\).

We have introduced two primitive syntactic forms:

\[\text{Abstraction}
\]

\[\text{Application}
\]

\[\text{Abstractions Returning Functions}
\]

Consider the following variant of \(\text{g}\):
\[\text{double} = \lambda f. \text{succ}(\text{succ}(\text{f}(\text{f}(\text{succ}(\text{succ}(\text{succ}(\text{succ}(\text{succ}(\text{succ}0))))))))))\]

I.e., \(\text{double}\) is the function that, when applied to a function \(f\), yields a function that, when applied to an argument \(y\), yields \(f(f(y))\).

CIS 500, 27 September 6
The Pure Lambda-Calculus

Syntax

Formalities

Example

Te"mplates

Axioms of the Form $\lambda x.t$ are called $\lambda$-abstractions or just abstractions

Terms in the pure $\lambda$-calculus are often called $\lambda$-terms

In this framework — the „pure Lambda-calculus“ — everything is a function.

let a rich and powerful programming language.

As the preceding examples suggest, once we have $\lambda$-abstraction and

application, we can throw away all the other Lambda-calculus and still have

References

Formalities

Syntax

Example

Formalities

Syntax

Example
Syntactic conventions

Since $\lambda$-calculus provides only one-argument functions, all multi-argument functions must be written in curried style. The following conventions make the linear forms of terms easier to read and write:

- Application associates to the left. E.g., $(tu)v$ means $t(uv)$, not $t(u)(v)$.
- Bodies of $\lambda$-abstractions extend as far to the right as possible. The scope of this binding is the body $t$.
- The $\lambda$-abstraction term $x.t$ binds the variable $x$.
- Occurrences of $x$ that are not within the scope of an abstraction binding $x$ are said to be free.
- Scope

Scope

Structural induction

What is the structural induction principle for lambda calculus terms?

Scope

Syntactic conventions

Syntactic conventions
### Values

**Values**

A term of the form \((\_x.t)v\) is called a redex (short for reducible expression). A term of the form \((\_x.t)\) is a \(\_\) abstraction applied to a value.

---

### Operational Semantics

**Computation rule:**

\[
(\_x.t)_1v_2 \rightarrow !\[x_7v_2\] t_1v_2 \quad \text{(E-AppAbs)}
\]

**Notation:**

\[
[x_7v_2] t_1v_2 \text{ is the term that results from substituting free occurrences of } x \text{ in } t_1v_2 \text{ with } v_1.
\]

---

**Congruence rules:**

\[
\begin{align*}
(\_x.t_1)_1 & \rightarrow \![x_7v_2]\ t_1v_2 & (\text{E-App1}) \\
(\_x.t_2)_1! & \rightarrow \![x_7v_2]\ t_2v_1 & (\text{E-App2})
\end{align*}
\]

---

### Terminology

- **A term of the form** \((\_x.t)v\) **is called a redex** (short for reducible expression).
Induction principle

What is the induction principle for the small-step evaluation relation?

We can show a property $P$ is true for all derivations of $\xi \rightarrow^* \xi'$, when

$P$ holds for all derivations that end with a use of E-App1 assuming that $P$ holds for all subderivations.

$P$ holds for all derivations that end with a use of E-App2 assuming that $P$ holds for all subderivations.

CIS 500, 27 September

Alternative evaluation strategies

Strictly speaking, the language we have defined is called the pure, call-by-value lambda-calculus.

The evaluation strategy we have chosen — call by value — reflects standard conventions found in most mainstream languages.

Some other common ones:

- Call by name (cf. Haskell)
- Normal order (leftmost/outermost)
- Full (non-deterministic) beta-reduction

CIS 500, 27 September
Above, we wrote a function `double` that returns a function as an argument. 

The Church Booleans

\[ \text{true} = \lambda x. \lambda y. x \]

\[ \text{false} = \lambda x. \lambda y. y \]

Function on Booleans

\[ \text{not} = \lambda x. x \text{false} \text{true} \text{false} \]

That is, `not` is a function that, given a boolean value \( v \), returns `false` if \( v \) is `true` and `true` if \( v \) is `false`.

Function on Booleans and Equal

\[ \text{and} = \lambda x. \lambda y. x \text{false} \text{true} y \]

That is, `and` is a function that, given two boolean values \( x \) and \( y \), returns `false` if \( x \) is `false` and \( y \) is `true`, and `true` if \( x \) is `true` and \( y \) is `true`.

Multiple arguments

Above, we wrote a function `double` that returns a function as an argument.
Pairs

\[ \text{pair} = \text{f. s. b. bfs} \]

\[ \text{fst} = \text{p. ptru} \]

\[ \text{snd} = \text{p. pfls} \]

That is, each number \( u \) is represented by a term \( c_n \) that takes two arguments, \( s \) and \( z \) (for "successor" and "zero"), and applies \( s \) \( n \) times to \( z \).

\[
\begin{align*}
\text{c}_0 &= \text{s. z.z} \\
\text{c}_1 &= \text{s. z.s(z)} \\
\text{c}_2 &= \text{s. z.s(s(z))} \\
\text{c}_3 &= \text{s. z.s(s(s(z)))}
\end{align*}
\]

Idea: represent the number \( n \) by a function that repeats some action \( n \) times.

\[
\begin{align*}
\text{successor:} & \\
\text{fun} & \text{c} & \text{f. s. z. c} & \text{f. s. z. (s. (c z)}) & \cdots \\
\end{align*}
\]

Example

\[
\text{fst(pairvw)} = \text{fst((s. b. bvs)w)} \]

\[
\begin{align*}
\text{reduce the underlined reduct} & \\
\text{reduce the underlined reduct} & \\
\text{by definition} & \\
\text{reduce the underlined reduct} & \\
\text{by definition} & \\
\text{reduce the underlined reduct} & \\
\text{reduce the underlined reduct} & \\
\text{by definition} & \\
\text{reduce the underlined reduct} & \\
\text{reduce the underlined reduct} & \\
\text{by definition} & \\
\end{align*}
\]

Church numerals

\[
\begin{align*}
\text{successor:} & \\
\text{fun} & \text{c} & \text{f. s. z. c} & \text{f. s. z. (s. (c z)}) & \cdots \\
\end{align*}
\]

\[
\begin{align*}
\text{addition:} & \\
\text{fun} & \text{m. n. s. z.m(nsv)} & \\
\text{multiplication:} & \\
\text{fun} & \text{m. n.m(plusn)c} & \\
\text{zero test:} & \\
\text{fun} & \text{m.m(x.fls)tru} & \\
\end{align*}
\]

\[ \text{predecessor?} \]

\[
\begin{align*}
\text{Example} & \\
\text{fun} & \text{f. p. f.ps} & \text{f. p. f.ps) (f. p. f.ps)} & \cdots & \\
\end{align*}
\]

\[
\begin{align*}
\text{pairs} & \\
\text{fun} & \text{a. b. (c. a.p (b.p))} & \\
\text{fun} & \text{a. b. (c. a.p (b.p))} & \\
\text{fun} & \text{a. b. (c. a.p (b.p))} & \\
\text{fun} & \text{a. b. (c. a.p (b.p))} & \\
\end{align*}
\]
Function on Church Numerals

Successor:
\[ \text{scc} = \lambda n. \lambda s. \lambda z. n \cdot s \cdot (n \cdot s \cdot z) \]

Addition:
\[ \text{plus} = \lambda m. \lambda n. \lambda s. \lambda z. m \cdot s \cdot (n \cdot s \cdot z) \]

Multiplication:
\[ \text{times} = \lambda m. \lambda n. m \cdot (\lambda s. \lambda z. n \cdot s \cdot z) \]

Zero test:
\[ \text{iszro} = \lambda m. \lambda n. \lambda s. \lambda z. m \cdot (\lambda x. \lambda f. \lambda l. m \cdot (x \cdot f) \cdot l) \cdot f \cdot l \]

What about predecessor?
What about predecessor?

Functions on Church Numerals

Successor:
\[ \text{succ} = \lambda x. \lambda f. \lambda z. f(x(z)) \]

Addition:
\[ \text{plus} = \lambda m. \lambda n. \lambda f. \lambda z. m(f(n(z))) \]

Multiplication:
\[ \text{times} = \lambda m. \lambda n. \lambda f. \lambda z. m(n(f(z))) \]

Zero test:
\[ \text{iszero} = \lambda m. \lambda f. \lambda z. f(z) \]

What about predecessor?
Recall: A normal form is a term that cannot take an evaluation step. A stuck term is a normal form that is not a value.

Are there any stuck terms in the pure λ-calculus?

A stuck term is a normal form that is not a value. A normal term is a term that cannot take an evaluation step.

Prove it.

Do every term evaluate to a normal form?

Prove it.
Divergence

\[ \omega = (x. \omega)(x. \omega) \]

Suppose \( \omega \) is some \( \lambda \)-abstraction, and consider the\ following term:

\[ \frac{1}{x} = (x. \omega)(x. \omega) \]

Note that \( \omega \) evaluates in one step to itself.

\[ \omega = (x. \omega)(x. \omega) \]

So evaluation of \( \omega \) never reaches a normal form: It \textit{diverges}.

Recursion in the Lambda Calculus

Iterated Application

Suppose \( f \) is some \( \lambda \)-abstraction, and consider the following term:

\[ Y_f = (x. f(xx))(x. f(xx)) \]

Now the pattern of divergence becomes more interesting:

\[ Y_f = (x. f(xx))(x. f(xx)) \]

\[ f((x. f(xx))(x. f(xx))) \]

\[ f(f((x. f(xx))(x. f(xx)))) \]

\[ f(f(f((x. f(xx))(x. f(xx))))) \]

...
Suppose \( f \) is some abstraction, and consider the following term:

\[
Y_f = (\lambda x. f(xx))(\lambda x. f(xx))
\]

Now the "pattern of divergence" becomes more interesting:

\[
Y_f = (\lambda x. f(xx))(\lambda x. f(xx))
\]

\[
! f((\lambda x. f(xx))(\lambda x. f(xx)))
\]

\[
! f(f((\lambda x. f(xx))(\lambda x. f(xx))))
\]

\[
! f(f(f((\lambda x. f(xx))(\lambda x. f(xx)))))
\]

\[
! f(f(f(f((\lambda x. f(xx))(\lambda x. f(xx)))))
\]

\[
Y_f \text{ is still not very useful, since } (\lambda x. f(xx)) \text{ always diverges.}
\]

Is there any way we could "slow it down"?

A delayed variant of \( \omega \):

\[
\omega_v = (\lambda x. (\lambda y. xx)(\lambda y. y))(\lambda x. (\lambda y. xx)(\lambda y. y))
\]

Note that \( \omega_v \) is a normal form. However, if we apply it to any argument \( v \):

\[
\omega_v v = (\lambda x. (\lambda y. xx)(\lambda y. y))(\lambda x. (\lambda y. xx)(\lambda y. y))
\]

Now let \( \text{postscript} \) be a value — it will only diverge when we actually apply it to an argument. This means that we can safely pass it as an argument:

\[
\text{postscript} \downarrow
\]

Delayed Divergence
Another delayed variant

Suppose \( f \) is a function. Define

\[
Z_f = \lambda y. (\lambda x. f y x y) (\lambda y. (\lambda x. f y x y) y)
\]

This term combines the "boxed" \( f \) with the "delayed divergence" of \( y \) (\( \forall y. f y y \)).

\[
\lambda y. (\lambda x. f y x y) y
\]

Suppose \( f \) is a function. Define

Another delayed variant.

We can use \( Z \) to "tie the knot" in the definition of \( f \) and obtain a real recursive factorial function:

\[
Z_f 3 = (\lambda y. (\lambda x. fct y x y) (\lambda y. (\lambda x. fct y x y) y)) 3
\]

Let

N.b.: For brevity, this example uses "real" numbers and booleans in syntax, etc. It can easily be translated into the pure lambda-calculus (using Church numerals, etc.).
Induction in the Lambda Calculus

If we define $Z$ to be

$$\lambda f.Z(f \cdot Z)$$

then we can obtain the behavior of $Zf$ for any $f$ we like, simply by applying $Z$ to $f$.

Let $\text{fact} = Z(\lambda n.\text{if}\ n=0\ \text{then} 1\ \text{else} \ n*\text{fact}(\text{pred}\ n))$.

For example:

\begin{align*}
\text{fact}\ 0 &= Z(\lambda n.\text{if}\ n=0\ \text{then} 1\ \text{else} \ n*\text{fact}(\text{pred}\ n)) \Rightarrow 1 \\
\text{fact}\ 1 &= Z(\lambda n.\text{if}\ n=0\ \text{then} 1\ \text{else} \ n*\text{fact}(\text{pred}\ n)) \Rightarrow 1*1 = 1 \\
\text{fact}\ 2 &= Z(\lambda n.\text{if}\ n=0\ \text{then} 1\ \text{else} \ n*\text{fact}(\text{pred}\ n)) \Rightarrow 1*1*2 = 2 \\
\text{fact}\ 3 &= Z(\lambda n.\text{if}\ n=0\ \text{then} 1\ \text{else} \ n*\text{fact}(\text{pred}\ n)) \Rightarrow 1*1*2*3 = 6 \\
\end{align*}
Two induction principles

Like before, there are two ways to prove properties are true of the untyped lambda calculus.
Induction on derivation

We want to prove, for all derivations of $t \rightarrow t_0$, that $FV(t) = FV(t_0)$. We have three cases.

1. The derivation of $t \rightarrow t_0$ could just be an use of E-App Abs. In this case, $t$ is $(\lambda x.t_0)v$ which steps to $[x \rightarrow v]t_0$. This case is analogous to the previous case.

2. By induction $FV(t_0)$ can be written as $FV(t_2) = (a) \land \{x\}/FV(t_2) - (3)\ FV(t_2)$. We have three cases.

   a. $\lambda x.t_0$. In this case, $t$ is $(\lambda x.t_0)v$ which steps to $[x \rightarrow v]t_0$. This case is analogous to the previous case.

   b. $t_1 \rightarrow t_2$. This case is analogous to the previous case.

   c. $t_1 \rightarrow t_2$. This case is analogous to the previous case.

We want to prove for all derivations of $t \rightarrow t_0$, that $FV(t) \subseteq FV(t_0)$.
The derivation could end with a use of $E$-APP. Here, we have a derivation analogous to the previous case. This case is