CIS 500

Software Foundations

Fall 2004

29 September

Announcements

Upcoming CIS Colloquia related to programming languages

Tuesdays, 3:00-4:30, Levine 101

- Oct 19 - Andy Gordon, MSR Cambridge
- Nov 16 - Greg Morrisett, Harvard University
- Nov 23 - Jeanette Wing, CMU

Today

- Encoding recursion
- Proving properties by induction
- Variable substitution and alpha-equivalence
- Program equivalence

Recursion in the Lambda Calculus

Iterated Application

Suppose `f` is some λ-abstraction, and consider the following term:

```
Y_f   =   (λx. f (x x)) (λx. f (x x))
```

Now the "pattern of divergence" becomes more interesting:

```
Y_f
=
(λx. f (x x)) (λx. f (x x))
⟶
f ((λx. f (x x)) (λx. f (x x)))
⟶
f (f ((λx. f (x x)) (λx. f (x x))))
⟶
f (f (f ((λx. f (x x)) (λx. f (x x)))))
⟶
...
```


Delaying Divergence

```
poisonpill   =   λy. omega
```

Note that `poisonpill` is a value — it it will only diverge when we actually apply it to an argument. This means that we can safely pass it as an argument to other functions, return it as a result from functions, etc.

```
(λp. fst (pair p fls) tru) poisonpill
⟶
fst (pair poisonpill fls) tru
⟶*
poisonpill tru
⟶
omega
⟶
...
```


Iterated Application

Suppose `f` is some λ-abstraction, and consider the following term:

```
Y_f   =   (λx. f (x x)) (λx. f (x x))
```


Y_f is still not very useful, since (like `omega`), all it does is diverge.

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

A delayed variant of `omega`

Here is a variant of `omega` in which the delay and divergence are a bit more tightly intertwined:

```
omegav  =  λy. (λx. (λy. x x y)) (λx. (λy. x x y)) y
```

Note that `omegav` is a normal form. However, if we apply it to any argument `v`, it diverges:

```
omegav v
=
(λy.  (λx. (λy. x x y)) (λx. (λy. x x y)) y) v
⟶
(λx. (λy. x x y)) (λx. (λy. x x y)) v
⟶
(λy.  (λx. (λy. x x y)) (λx. (λy. x x y)) y) v
=
omegav v
```


Another delayed variant

Suppose `f` is a function. Define

```
Zf   =  λy. (λx. f (λy. x x y)) (λx. f (λy. x x y)) y
```

This term combines the "added `f`" from $\mathtt{Y_f}$ with the "delayed divergence" of `omegav`.

If we now apply $\mathtt{Z_f}$ to an argument `v`, something interesting happens:

```
Zf v
=
(λy. (λx. f (λy. x x y)) (λx. f (λy. x x y)) y) v
⟶
(λx. f (λy. x x y)) (λx. f (λy. x x y)) v
⟶
f (λy. (λx. f (λy. x x y)) (λx. f (λy. x x y)) y) v
=
f Zf v
```

Since $\mathtt{Z_f}$ and `v` are both values, the next computation step will be the reduction of `f` $\mathtt{Z_f}$ — that is, before we "diverge," `f` gets to do some computation.

Now we are getting somewhere.

Recursion

Let

```
f  =  λfct.
        λn.
          if n=0 then 1
          else n * (fct (pred n))
```

`f` looks just the ordinary factorial function, except that, in place of a recursive call in the last time, it calls the function `fct`, which is passed as a parameter.

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

A Generic Z

If we define

```
Z  =  λf. Z_f
```

i.e.,

```
Z  =  λf. λy. (λx. f (λy. x x y)) (λx. f (λy. x x y)) y
```

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

```
Z f  ⟶  Z_f
```


Technical note:

The term `Z` here is essentially the same as the `fix` discussed the book.

```
Z    =  λf. λy. (λx. f (λy. x x y)) (λx. f (λy. x x y)) y
fix  =  λf. (λx. f (λy. x x y)) (λx. f (λy. x x y))
```

`Z` is hopefully slightly easier to understand, since it has the property that `Z f v` $\longrightarrow^*$ `f (Z f) v`, which `fix` does not (quite) share.

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

```
Z_f 3
⟶*
f Z_f 3
=
(λfct. λn. ...) Z_f 3
⟶  ⟶
if 3=0 then 1 else 3 * (Z_f (pred 3))
⟶*
3 * (Z_f (pred 3)))
⟶
3 * (Z_f 2)
⟶*
3 * (f Z_f 2)
...
```


For example:

```
fact  =  Z ( λfct.
               λn.
                 if n=0 then 1
                 else n * (fct (pred n)) )
```

Proofs about the Lambda Calculus

Two induction principles

Like before, we have mentioned two ways to prove properties are true of the untyped lambda calculus.

- Structural induction
- Induction on derivation of $\mathtt{t} \longrightarrow \mathtt{t}'$.

Let's do an example of the latter.

Induction principle

Recall the induction principle for the small-step evaluation relation.

We can show a property $\boldsymbol{P}$ is true for all derivations of $\mathtt{t} \longrightarrow \mathtt{t}'$, when

- $\boldsymbol{P}$ holds for all derivations that use the rule E-AppAbs.
- $\boldsymbol{P}$ holds for all derivations that end with a use of E-App1 assuming that $\boldsymbol{P}$ holds for all subderivations.
- $\boldsymbol{P}$ holds for all derivations that end with a use of E-App2 assuming that $\boldsymbol{P}$ holds for all subderivations.

Example

We can formally define the set of free variables in a λ-term as follows:

$$\mathbf{FV}(\mathtt{x}) = \{\mathtt{x}\}$$
$$\mathbf{FV}(\lambda\mathtt{x}.\mathtt{t}_1) = \mathbf{FV}(\mathtt{t}_1)/\{\mathtt{x}\}$$
$$\mathbf{FV}(\mathtt{t}_1\ \mathtt{t}_2) = \mathbf{FV}(\mathtt{t}_1) \cup \mathbf{FV}(\mathtt{t}_2)$$

Theorem: if $\mathtt{t} \longrightarrow \mathtt{t}'$ then $\mathbf{FV}(\mathtt{t}) \supseteq \mathbf{FV}(\mathtt{t}')$.

Induction on derivation

We want to prove, for all derivations of $\texttt{t} \to \texttt{t}'$, that $\mathbf{FV}(\texttt{t}) \supseteq \mathbf{FV}(\texttt{t}')$.

We have three cases.

- The derivation of $\texttt{t} \to \texttt{t}'$ could just be a use of E-AppAbs. In this case, $\texttt{t}$ is $(\lambda\texttt{x.u})\texttt{v}$ which steps to $[\texttt{x}\mapsto\texttt{v}]\texttt{u}$.

$$\begin{aligned} \mathbf{FV}(\texttt{t}) &= \mathbf{FV}((\lambda\texttt{x.u})\texttt{v}) \\ &= \mathbf{FV}(\texttt{u})/\{\texttt{x}\} \cup \mathbf{FV}(\texttt{v}) \\ &\supseteq \mathbf{FV}([\texttt{x}\mapsto\texttt{v}]\texttt{u}) \\ &= \mathbf{FV}(\texttt{t}') \end{aligned}$$

- The derivation could end with a use of E-App1. In other words, we have a derivation of $\texttt{t}_1 \to \texttt{t}_1'$ and we use it to show that $\texttt{t}_1\ \texttt{t}_2 \to \texttt{t}_1'\ \texttt{t}_2$. By induction $\mathbf{FV}(\texttt{t}_1) \supseteq \mathbf{FV}(\texttt{t}_1')$.

$$\begin{aligned} \mathbf{FV}(\texttt{t}) &= \mathbf{FV}(\texttt{t}_1\ \texttt{t}_2) \\ &= \mathbf{FV}(\texttt{t}_1) \cup \mathbf{FV}(\texttt{t}_2) \\ &\supseteq \mathbf{FV}(\texttt{t}_1') \cup \mathbf{FV}(\texttt{t}_2) \\ &= \mathbf{FV}(\texttt{t}_1'\ \texttt{t}_2) \\ &= \mathbf{FV}(\texttt{t}') \end{aligned}$$

- The derivation could end with a use of E-App2. Here, we have a derivation of $\texttt{t}_2 \to \texttt{t}_2'$ and we use it to show that $\texttt{t}_1\ \texttt{t}_2 \to \texttt{t}_1\ \texttt{t}_2'$. This case is analogous to the previous case.

Induction on derivation

We want to prove, for all derivations of $\texttt{t} \to \texttt{t}'$, that $\mathbf{FV}(\texttt{t}) \supseteq \mathbf{FV}(\texttt{t}')$.

We have three cases.

- The derivation could end with a use of E-App1. In other words, we have a derivation of $\texttt{t}_1 \to \texttt{t}_1'$ and we use it to show that $\texttt{t}_1\ \texttt{t}_2 \to \texttt{t}_1'\ \texttt{t}_2$. By induction $\mathbf{FV}(\texttt{t}_1) \supseteq \mathbf{FV}(\texttt{t}_1')$.

$$\begin{aligned} \mathbf{FV}(\texttt{t}) &= \mathbf{FV}(\texttt{t}_1\ \texttt{t}_2) \\ &= \mathbf{FV}(\texttt{t}_1) \cup \mathbf{FV}(\texttt{t}_2) \\ &\supseteq \mathbf{FV}(\texttt{t}_1') \cup \mathbf{FV}(\texttt{t}_2) \\ &= \mathbf{FV}(\texttt{t}_1'\ \texttt{t}_2) \\ &= \mathbf{FV}(\texttt{t}') \end{aligned}$$

Substitution

Our definition of evaluation was based on the substitution of values for free variables within terms.

E-AppAbs

$(\lambda x.t_{12})\ v_2 \longrightarrow [x \mapsto v_2]t_{12}$

But what is substitution, really? How do we define it?

Formalizing Substitution

Consider the following definition of substitution:

$[x \mapsto s]x = s$

$[x \mapsto s]y = y \qquad \text{if } x \neq y$

$[x \mapsto s](\lambda y.t_1) = \lambda y.\ ([x \mapsto s]t_1)$

$[x \mapsto s](t_1\ t_2) = ([x \mapsto s]t_1)([x \mapsto s]t_2)$

What is wrong with this definition?

It substitutes for free and bound variables!

$[x \mapsto y](\lambda x.\ x) = \lambda x.y$

This is not what we want.

More about bound variables

Formalizing Substitution

Consider the following definition of substitution:

$[x \mapsto s]x = s$

$[x \mapsto s]y = y \qquad \text{if } x \neq y$

$[x \mapsto s](\lambda y.t_1) = \lambda y.\ ([x \mapsto s]t_1)$

$[x \mapsto s](t_1\ t_2) = ([x \mapsto s]t_1)([x \mapsto s]t_2)$

What is wrong with this definition?

Substitution, take two

$[x \mapsto s]x = s$

$[x \mapsto s]y = y$ if $x \neq y$

$[x \mapsto s](\lambda y.t_1) = \lambda y.\ ([x \mapsto s]t_1)$ if $x \neq y$

$[x \mapsto s](\lambda x.t_1) = \lambda x.\ t_1$

$[x \mapsto s](t_1\ t_2) = ([x \mapsto s]t_1)([x \mapsto s]t_2)$

What is wrong with this definition?

It suffers from variable capture!

$[x \mapsto y](\lambda y.x) = \lambda x.\ x$

This is also not what we want.

Substitution, take three

$[x \mapsto s]x = s$

$[x \mapsto s]y = y$ if x is not y

$[x \mapsto s](\lambda y.t_1) = \lambda y.\ ([x \mapsto s]t_1)$ if $x \neq y$, $y \notin FV(s)$

$[x \mapsto s](\lambda x.t_1) = \lambda x.\ t_1$

$[x \mapsto s](t_1\ t_2) = ([x \mapsto s]t_1)([x \mapsto s]t_2)$

What is wrong with this definition?

Now substition is a partial function!

$[x \mapsto y](\lambda y.x)$ is undefined.

But we want an answer for every substitution.

Substitution, take two

$[x \mapsto s]x = s$

$[x \mapsto s]y = y$ if $x \neq y$

$[x \mapsto s](\lambda y.t_1) = \lambda y.\ ([x \mapsto s]t_1)$ if $x \neq y$

$[x \mapsto s](\lambda x.t_1) = \lambda x.\ t_1$

$[x \mapsto s](t_1\ t_2) = ([x \mapsto s]t_1)([x \mapsto s]t_2)$

What is wrong with this definition?

Substitution, take three

$[x \mapsto s]x = s$

$[x \mapsto s]y = y$ if x is not y

$[x \mapsto s](\lambda y.t_1) = \lambda y.\ ([x \mapsto s]t_1)$ if $x \neq y$, $y \notin FV(s)$

$[x \mapsto s](\lambda x.t_1) = \lambda x.\ t_1$

$[x \mapsto s](t_1\ t_2) = ([x \mapsto s]t_1)([x \mapsto s]t_2)$

What is wrong with this definition?

Alpha-equivalence classes

In fact, we can create equivalence classes of terms that differ only in the names of bound variables.

When working with the lambda calculus, it is convenient to think about these equivalence classes, instead of raw terms.

For example, when we write $\lambda x.x$ we mean not just this term, but the class of terms that includes $\lambda y.y$ and $\lambda z.z$.

Unfortunately, we have to be more clever when implementing the lambda calculus in ML... (cf. TAPL chapters 6 and 7)

Equivalence of Lambda Terms

Bound variable names shouldn't matter

It's annoying that that the names of bound variables are causing trouble with our definition of substitution.

Intuition tells us that there shouldn't be a difference between the functions $\lambda x.x$ and $\lambda y.y$. Both of these functions will do the same thing.

Because they differ only in the names of their bound variables, we'd like to think that these are the same function.

We call such terms alpha-equivalent.

Substitution, for alpha-equivalence classes

Now consider substitution as an operation over alpha-equivalence classes of terms:

$[x \mapsto s]x = s$	
$[x \mapsto s]y = y$	if $x \neq y$
$[x \mapsto s](\lambda y.t_1) = \lambda y.\ ([x \mapsto s]t_1)$	if $x \neq y$, $y \notin FV(s)$
$[x \mapsto s](t_1\ t_2) = ([x \mapsto s]t_1)([x \mapsto s]t_2)$	

Examples:

- $[x \mapsto y](\lambda y.x)$ must give the same result as $[x \mapsto y](\lambda z.x)$. We know the latter is $\lambda z.y$, so that is what we will use for the former.
- $[x \mapsto y](\lambda x.z)$ must give the same result as $[x \mapsto y](\lambda w.z)$. We know the latter is $\lambda w.z$ so that is what we use for the former.

Why is program equivalence important?

Representing Numbers

We have seen how certain terms in the lambda-calculus can be used to represent natural numbers.

c_0 = λs. λz. z
c_1 = λs. λz. s z
c_2 = λs. λz. s (s z)
c_3 = λs. λz. s (s (s z))

Other lambda-terms represent common operations on numbers:

```
scc = λn. λs. λz. s (n s z)
```


Program Equivalence

- Syntactic equivalence - Are the terms the same "letter by letter"? Not that useful.
- Alpha-equivalence - Are the terms equivalent up to renaming of bound variables?
- Beta/eta-equivalence - Can we use specific program transformations to convert one term into another?
- Behavioral equivalence - If both terms are placed in the same context, will they produce the same result?

Why is program equivalence important?

- Used to catch cheaters in low-level programming classes.
- Used to prove the correctness of embeddings. (Why should we believe that Church encodings represent natural numbers?)
- Used to prove the correctness of compiler optimizations.
- Used to show that updates to a program do not break it.

The naive approach

One possibility:

For each n, the term `scc` c_n evaluates to c_{n+1}.

A better approach

Recall the intuition behind the church numeral representation:

- a number n is represented as a term that "does something n times to something else"
- `scc` takes a term that "does something n times to something else" and returns a term that "does something $n+1$ times to something else"

I.e., what we really care about is that `scc` c_2 behaves the same as c_3 when applied to two arguments.

Representing Numbers

We have seen how certain terms in the lambda-calculus can be used to represent natural numbers.

```
c0 = λs. λz. z
c1 = λs. λz. s z
c2 = λs. λz. s (s z)
c3 = λs. λz. s (s (s z))
```

Other lambda-terms represent common operations on numbers:

```
scc = λn. λs. λz. s (n s z)
```

In what sense can we say this representation is "correct"?

In particular, on what basis can we argue that `scc` on church numerals corresponds to ordinary successor on numbers?

The naive approach... doesn't work

One possibility:

For each n, the term `scc` c_n evaluates to c_{n+1}.

Unfortunately, this is false.

E.g.:

```
scc c2  =   (λn. λs. λz. s (n s z)) (λs. λz. s (s z))
        ⟶   λs. λz. s ((λs. λz. s (s z)) s z)
        ≠   λs. λz. s (s (s z))
        =   c3
```

```
scc c2 v w   =   (λn. λs. λz. s (n s z)) (λs. λz. s (s z)) v w
             ⟶   (λs. λz. s ((λs. λz. s (s z)) s z)) v w
             ⟶   (λz. v ((λs. λz. s (s z)) v z)) w
             ⟶   v ((λs. λz. s (s z)) v w)
             ⟶   v ((λz. v (v z)) w)
             ⟶   v (v (v w))

c3 v w       =   (λs. λz. s (s (s z))) v w
             ⟶   (λz. v (v (v z))) w
             ⟶   v (v (v w)))
```


A More General Question

We have argued that, although `scc` c_2 and c_3 do not evaluate to the same thing, they are nevertheless "behaviorally equivalent."

What, precisely, does behavioral equivalence mean?

Intuition

Roughly,

> terms `s` and `t` are behaviorally equivalent

should mean:

> there is no "test" that distinguishes `s` and `t` — i.e., no way to use them in the same context and obtain different results.

Some test cases

```
tru = λt. λf. t
tru' = λt. λf. (λx.x) t
fls = λt. λf. f
omega = (λx. x x) (λx. x x)
poisonpill = λx. omega
placebo = λx. tru
Y_f = (λx. f (x x)) (λx. f (x x))
```

Which of these are behaviorally equivalent?

Observational equivalence

As a first step toward defining behavioral equivalence, we can use the notion of normalizability to define a simple way of testing terms.

> Two terms `s` and `t` are said to be observationally equivalent if either both are normalizable (i.e., they reach a normal form after a finite number of evaluation steps) or both are divergent.

I.e., our primitive notion of "observing" a term's behavior is simply running it on our abstract machine.

Aside:

- Is observational equivalence a decidable property?

Examples

- `omega` and `tru` are not observationally equivalent

Observational equivalence

As a first step toward defining behavioral equivalence, we can use the notion of normalizability to define a simple way of testing terms.

> Two terms `s` and `t` are said to be observationally equivalent if either both are normalizable (i.e., they reach a normal form after a finite number of evaluation steps) or both are divergent.

I.e., our primitive notion of "observing" a term's behavior is simply running it on our abstract machine.

Observational equivalence

As a first step toward defining behavioral equivalence, we can use the notion of normalizability to define a simple way of testing terms.

> Two terms `s` and `t` are said to be observationally equivalent if either both are normalizable (i.e., they reach a normal form after a finite number of evaluation steps) or both are divergent.

I.e., our primitive notion of "observing" a term's behavior is simply running it on our abstract machine.

Aside:

- Is observational equivalence a decidable property?
- Does this mean the definition is ill-formed?

Behavioral Equivalence

This primitive notion of observation now gives us a way of "testing" terms for behavioral equivalence

> Terms `s` and `t` are said to be behaviorally equivalent if, for every finite sequence of values $v_1, v_2, \ldots, v_n$, the applications
>
> $$s\ v_1\ v_2\ \ldots\ v_n$$
>
> and
>
> $$t\ v_1\ v_2\ \ldots\ v_n$$
>
> are observationally equivalent.

Examples

- `omega` and `tru` are not observationally equivalent
- `tru` and `fls` are observationally equivalent

Examples

These terms are behaviorally equivalent:

```
tru = λt. λf. t
tru' = λt. λf. (λx.x) t
```

So are these:

```
omega = (λx. x x) (λx. x x)
Y_f = (λx. f (x x)) (λx. f (x x))
```

These are not behaviorally equivalent (to each other, or to any of the terms above):

```
fls = λt. λf. f
poisonpill = λx. omega
placebo = λx. tru
```