Foundations of Software
Fall 2015

Week 3

Review (and more details)

Recall: Simple Arithmetic Expressions

The set $\mathcal{T}$ of terms is defined by the following abstract grammar:

`t ::=`	*terms*
`true`	*constant true*
`false`	*constant false*
`if t then t else t`	*conditional*
`0`	*constant zero*
`succ t`	*successor*
`pred t`	*predecessor*
`iszero t`	*zero test*

Recall: Inference Rule Notation

More explicitly: The set $\mathcal{T}$ is the *smallest* set *closed* under the following rules.

$$\texttt{true} \in \mathcal{T} \qquad \texttt{false} \in \mathcal{T} \qquad \texttt{0} \in \mathcal{T}$$

$$\frac{\texttt{t}_1 \in \mathcal{T}}{\texttt{succ t}_1 \in \mathcal{T}} \qquad \frac{\texttt{t}_1 \in \mathcal{T}}{\texttt{pred t}_1 \in \mathcal{T}} \qquad \frac{\texttt{t}_1 \in \mathcal{T}}{\texttt{iszero t}_1 \in \mathcal{T}}$$

$$\frac{\texttt{t}_1 \in \mathcal{T} \qquad \texttt{t}_2 \in \mathcal{T} \qquad \texttt{t}_3 \in \mathcal{T}}{\texttt{if t}_1\texttt{ then t}_2\texttt{ else t}_3 \in \mathcal{T}}$$

Generating Functions

Each of these rules can be thought of as a *generating function* that, given some elements from $\mathcal{T}$, generates some other element of $\mathcal{T}$. Saying that $\mathcal{T}$ is closed under these rules means that $\mathcal{T}$ cannot be made any bigger using these generating functions — it already contains everything "justified by its members."

$$\texttt{true} \in \mathcal{T} \qquad \texttt{false} \in \mathcal{T} \qquad \texttt{0} \in \mathcal{T}$$

$$\frac{\texttt{t}_1 \in \mathcal{T}}{\texttt{succ t}_1 \in \mathcal{T}} \qquad \frac{\texttt{t}_1 \in \mathcal{T}}{\texttt{pred t}_1 \in \mathcal{T}} \qquad \frac{\texttt{t}_1 \in \mathcal{T}}{\texttt{iszero t}_1 \in \mathcal{T}}$$

$$\frac{\texttt{t}_1 \in \mathcal{T} \qquad \texttt{t}_2 \in \mathcal{T} \qquad \texttt{t}_3 \in \mathcal{T}}{\texttt{if t}_1 \texttt{ then t}_2 \texttt{ else t}_3 \in \mathcal{T}}$$

Let's write these generating functions explicitly.

$$\begin{array}{lcl}
F_1(U) & = & \{\texttt{true}\} \\
F_2(U) & = & \{\texttt{false}\} \\
F_3(U) & = & \{\texttt{0}\} \\
F_4(U) & = & \{\texttt{succ t}_1 \mid \texttt{t}_1 \in U\} \\
F_5(U) & = & \{\texttt{pred t}_1 \mid \texttt{t}_1 \in U\} \\
F_6(U) & = & \{\texttt{iszero t}_1 \mid \texttt{t}_1 \in U\} \\
F_7(U) & = & \{\texttt{if t}_1 \texttt{ then t}_2 \texttt{ else t}_3 \mid \texttt{t}_1, \texttt{t}_2, \texttt{t}_3 \in U\}
\end{array}$$

Each one takes a set of terms U as input and produces a set of "terms justified by U" as output.

If we now define a generating function for the whole set of inference rules (by combining the generating functions for the individual rules),

$$F(U) \quad = \quad F_1(U) \cup F_2(U) \cup F_3(U) \cup F_4(U) \cup F_5(U) \cup F_6(U) \cup F_7(U)$$

then we can restate the previous definition of the set of terms $\mathcal{T}$ like this:

Definition:

- A set U is said to be "closed under F" (or "F-closed") if $F(U) \subseteq U$.
- The set of terms $\mathcal{T}$ is the smallest F-closed set.
  (I.e., if $\mathcal{O}$ is another set such that $F(\mathcal{O}) \subseteq \mathcal{O}$, then $\mathcal{T} \subseteq \mathcal{O}$.)

Our alternate definition of the set of terms can also be stated using the generating function F:

$$\begin{array}{lcl}
\mathcal{S}_0 & = & \emptyset \\
\mathcal{S}_{i+1} & = & F(\mathcal{S}_i)
\end{array}$$

$$\mathcal{S} \quad = \quad \bigcup_i \mathcal{S}_i$$

Compare this definition of $\mathcal{S}$ with the one we saw last time:

$$\begin{array}{lcll}
\mathcal{S}_0 & = & \emptyset & \\
\mathcal{S}_{i+1} & = & & \{\texttt{true}, \texttt{false}, \texttt{0}\} \\
& & \cup & \{\texttt{succ t}_1, \texttt{pred t}_1, \texttt{iszero t}_1 \mid \texttt{t}_1 \in \mathcal{S}_i\} \\
& & \cup & \{\texttt{if t}_1 \texttt{ then t}_2 \texttt{ else t}_3 \mid \texttt{t}_1, \texttt{t}_2, \texttt{t}_3 \in \mathcal{S}_i\}
\end{array}$$

$$\mathcal{S} \quad = \quad \bigcup_i \mathcal{S}_i$$

We have "pulled out" F and given it a name.

Note that our two definitions of terms characterize the same set from different directions:

- "from above," as the intersection of all F-closed sets;
- "from below," as the limit (union) of a series of sets that start from $\emptyset$ and get "closer and closer to being F-closed."

Proposition 3.2.6 in the book shows that these two definitions actually define the same set.

Warning: Hard hats on for the next slide!

Structural Induction

The principle of structural induction on terms can also be re-stated using generating functions:

Suppose T is the smallest F-closed set.

If, for each set U,
from the assumption "$P(u)$ holds for every $u \in U$"
we can show "$P(v)$ holds for any $v \in F(U)$,"
then $P(t)$ holds for all $t \in T$.

Structural Induction

The principle of structural induction on terms can also be re-stated using generating functions:

Suppose T is the smallest F-closed set.

If, for each set U,
from the assumption "$P(u)$ holds for every $u \in U$"
we can show "$P(v)$ holds for any $v \in F(U)$,"
then $P(t)$ holds for all $t \in T$.

Why?

Structural Induction

Why? Because:

- We assumed that T was the *smallest* F-closed set, i.e., that $T \subseteq O$ for any other F-closed set O.
- But showing

  for each set U,
  given $P(u)$ for all $u \in U$
  we can show $P(v)$ for all $v \in F(U)$

  amounts to showing that "the set of all terms satisfying P" (call it O) is itself an F-closed set.
- Since $T \subseteq O$, every element of T satisfies P.

Structural Induction

Compare this with the structural induction principle for terms from last lecture:

If, for each term s,
given $P(r)$ for all immediate subterms r of s
we can show $P(s)$,
then $P(t)$ holds for all t.

Recall, from the definition of $\mathcal{S}$, it is clear that, if a term $\mathtt{t}$ is in $\mathcal{S}_i$, then all of its immediate subterms must be in $\mathcal{S}_{i-1}$, i.e., they must have strictly smaller depths. Therefore:

If, for each term s,
given $P(r)$ for all immediate subterms r of s
we can show $P(s)$,
then $P(t)$ holds for all t.

Slightly more explicit proof:

- Assume that for each term $\mathtt{s}$, given $P(\mathtt{r})$ for all immediate subterms of $\mathtt{s}$, we can show $P(\mathtt{s})$.
- Then show, by induction on i, that $P(\mathtt{t})$ holds for all terms $\mathtt{t}$ with depth i.
- Therefore, $P(\mathtt{t})$ holds for all $\mathtt{t}$.

Operational Semantics and Reasoning

Recall: Abstract Machines

An *abstract machine* consists of:

- a set of *states*
- a *transition relation* on states, written $\longrightarrow$

For the simple languages we are considering at the moment, the term being evaluated is the whole state of the abstract machine.

Recall: Syntax for Booleans

Terms and values

`t ::=`		*terms*
	`true`	*constant true*
	`false`	*constant false*
	`if t then t else t`	*conditional*
`v ::=`		*values*
	`true`	*true value*
	`false`	*false value*

Recall: Operational Semantics for Booleans

The evaluation relation $\mathtt{t} \longrightarrow \mathtt{t}'$ is the smallest relation closed under the following rules:

$$\mathtt{if\ true\ then\ t_2\ else\ t_3} \longrightarrow \mathtt{t_2} \qquad \text{(E-IfTrue)}$$

$$\mathtt{if\ false\ then\ t_2\ else\ t_3} \longrightarrow \mathtt{t_3} \qquad \text{(E-IfFalse)}$$

$$\frac{\mathtt{t_1} \longrightarrow \mathtt{t_1'}}{\mathtt{if\ t_1\ then\ t_2\ else\ t_3} \longrightarrow \mathtt{if\ t_1'\ then\ t_2\ else\ t_3}} \qquad \text{(E-If)}$$

Derivations

We can record the "justification" for a particular pair of terms that are in the evaluation relation in the form of a tree.

(on the board)

Terminology:

- These trees are called *derivation trees* (or just *derivations*).
- The final statement in a derivation is its *conclusion*.
- We say that the derivation is a *witness* for its conclusion (or a *proof* of its conclusion) — it records all the reasoning steps that justify the conclusion.

Observation

Lemma: Suppose we are given a derivation tree $\mathcal{D}$ witnessing the pair $(\mathtt{t}, \mathtt{t}')$ in the evaluation relation. Then either

1. the final rule used in $\mathcal{D}$ is E-IfTrue and we have $\mathtt{t} = \mathtt{if\ true\ then\ t_2\ else\ t_3}$ and $\mathtt{t}' = \mathtt{t_2}$, for some $\mathtt{t_2}$ and $\mathtt{t_3}$, or
2. the final rule used in $\mathcal{D}$ is E-IfFalse and we have $\mathtt{t} = \mathtt{if\ false\ then\ t_2\ else\ t_3}$ and $\mathtt{t}' = \mathtt{t_3}$, for some $\mathtt{t_2}$ and $\mathtt{t_3}$, or
3. the final rule used in $\mathcal{D}$ is E-If and we have $\mathtt{t} = \mathtt{if\ t_1\ then\ t_2\ else\ t_3}$ and $\mathtt{t}' = \mathtt{if\ t_1'\ then\ t_2\ else\ t_3}$, for some $\mathtt{t_1}$, $\mathtt{t_1'}$, $\mathtt{t_2}$, and $\mathtt{t_3}$; moreover, the immediate subderivation of $\mathcal{D}$ witnesses $(\mathtt{t_1}, \mathtt{t_1'}) \in \longrightarrow$.

Induction on Derivations

We can now write proofs about evaluation "by induction on derivation trees."

Given an arbitrary derivation $\mathcal{D}$ with conclusion $\mathtt{t} \longrightarrow \mathtt{t}'$, we assume the desired result for its immediate sub-derivation (if any) and proceed by a case analysis (using the previous lemma) of the final evaluation rule used in constructing the derivation tree.

E.g....

Induction on Derivations — Example

Theorem: If $\mathtt{t} \longrightarrow \mathtt{t}'$, i.e., if $(\mathtt{t}, \mathtt{t}') \in \longrightarrow$, then $\mathit{size}(\mathtt{t}) > \mathit{size}(\mathtt{t}')$.
Proof: By induction on a derivation $\mathcal{D}$ of $\mathtt{t} \longrightarrow \mathtt{t}'$.

1. Suppose the final rule used in $\mathcal{D}$ is E-IfTrue, with $\mathtt{t} = \mathtt{if\ true\ then\ t_2\ else\ t_3}$ and $\mathtt{t}' = \mathtt{t_2}$. Then the result is immediate from the definition of *size*.
2. Suppose the final rule used in $\mathcal{D}$ is E-IfFalse, with $\mathtt{t} = \mathtt{if\ false\ then\ t_2\ else\ t_3}$ and $\mathtt{t}' = \mathtt{t_3}$. Then the result is again immediate from the definition of *size*.
3. Suppose the final rule used in $\mathcal{D}$ is E-If, with $\mathtt{t} = \mathtt{if\ t_1\ then\ t_2\ else\ t_3}$ and $\mathtt{t}' = \mathtt{if\ t_1'\ then\ t_2\ else\ t_3}$, where $(\mathtt{t_1}, \mathtt{t_1'}) \in \longrightarrow$ is witnessed by a derivation $\mathcal{D}_1$. By the induction hypothesis, $\mathit{size}(\mathtt{t_1}) > \mathit{size}(\mathtt{t_1'})$. But then, by the definition of *size*, we have $\mathit{size}(\mathtt{t}) > \mathit{size}(\mathtt{t}')$.

Normal forms

A *normal form* is a term that cannot be evaluated any further — i.e., a term $\mathtt{t}$ is a normal form (or "is in normal form") if there is no $\mathtt{t}'$ such that $\mathtt{t} \longrightarrow \mathtt{t}'$.

A normal form is a state where the abstract machine is halted — i.e., it can be regarded as a "result" of evaluation.

Normal forms

A *normal form* is a term that cannot be evaluated any further — i.e., a term $\mathtt{t}$ is a normal form (or "is in normal form") if there is no $\mathtt{t}'$ such that $\mathtt{t} \longrightarrow \mathtt{t}'$.

A normal form is a state where the abstract machine is halted — i.e., it can be regarded as a "result" of evaluation.

Recall that we intended the set of *values* (the boolean constants `true` and `false`) to be exactly the possible "results of evaluation." Did we get this definition right?

Values = normal forms

Theorem: A term $\mathtt{t}$ is a value iff it is in normal form.
Proof:
The $\Longrightarrow$ direction is immediate from the definition of the evaluation relation.

Values = normal forms

Theorem: A term $\mathtt{t}$ is a value iff it is in normal form.
Proof:
The $\Longrightarrow$ direction is immediate from the definition of the evaluation relation.
For the $\Longleftarrow$ direction,

Values = normal forms

Theorem: A term $\mathtt{t}$ is a value iff it is in normal form.
Proof:
The $\Longrightarrow$ direction is immediate from the definition of the evaluation relation.
For the $\Longleftarrow$ direction, it is convenient to prove the contrapositive:
If $\mathtt{t}$ is *not* a value, then it is *not* a normal form.

Values $=$ normal forms

Theorem: A term t is a value iff it is in normal form.
Proof:
The $\Longrightarrow$ direction is immediate from the definition of the evaluation relation.
For the $\Longleftarrow$ direction, it is convenient to prove the contrapositive: If t is *not* a value, then it is *not* a normal form. The argument goes by induction on t.
Note, first, that t must have the form if t_1 then t_2 else t_3 (otherwise it would be a value). If t_1 is true or false, then rule E-IfTrue or E-IfFalse applies to t, and we are done.
Otherwise, t_1 is not a value and so, by the induction hypothesis, there is some t_1' such that $t_1 \longrightarrow t_1'$. But then rule E-If yields

$$\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \longrightarrow \texttt{if } t_1' \texttt{ then } t_2 \texttt{ else } t_3$$

i.e., t is not in normal form.

Numbers

New syntactic forms

t ::=	...	*terms*
	0	*constant zero*
	succ t	*successor*
	pred t	*predecessor*
	iszero t	*zero test*
v ::=	...	*values*
	nv	*numeric value*
nv ::=		*numeric values*
	0	*zero value*
	succ nv	*successor value*

New evaluation rules $\boxed{t \longrightarrow t'}$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{succ } t_1 \longrightarrow \texttt{succ } t_1'} \quad \text{(E-Succ)}$$

$$\texttt{pred 0} \longrightarrow \texttt{0} \quad \text{(E-PredZero)}$$

$$\texttt{pred (succ } nv_1\texttt{)} \longrightarrow nv_1 \quad \text{(E-PredSucc)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{pred } t_1 \longrightarrow \texttt{pred } t_1'} \quad \text{(E-Pred)}$$

$$\texttt{iszero 0} \longrightarrow \texttt{true} \quad \text{(E-IszeroZero)}$$

$$\texttt{iszero (succ } nv_1\texttt{)} \longrightarrow \texttt{false} \quad \text{(E-IszeroSucc)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{iszero } t_1 \longrightarrow \texttt{iszero } t_1'} \quad \text{(E-IsZero)}$$

Values are normal forms

Our observation a few slides ago that all values are in normal form still holds for the extended language.

Is the converse true? I.e., is every normal form a value?

Values are normal forms, but we have stuck terms

Our observation a few slides ago that all values are in normal form still holds for the extended language.

Is the converse true? I.e., is every normal form a value?
No: some terms are *stuck*.

Formally, a stuck term is one that is a normal form but not a value.
What are some examples?

Stuck terms model run-time errors.

Multi-step evaluation.

The *multi-step evaluation* relation, $\longrightarrow^*$, is the reflexive, transitive closure of single-step evaluation.

I.e., it is the smallest relation closed under the following rules:

$$\frac{\mathtt{t} \longrightarrow \mathtt{t}'}{\mathtt{t} \longrightarrow^* \mathtt{t}'}$$

$$\mathtt{t} \longrightarrow^* \mathtt{t}$$

$$\frac{\mathtt{t} \longrightarrow^* \mathtt{t}' \qquad \mathtt{t}' \longrightarrow^* \mathtt{t}''}{\mathtt{t} \longrightarrow^* \mathtt{t}''}$$

Termination of evaluation

Theorem: For every $\mathtt{t}$ there is some normal form $\mathtt{t}'$ such that $\mathtt{t} \longrightarrow^* \mathtt{t}'$.
Proof:

Termination of evaluation

Theorem: For every $\mathtt{t}$ there is some normal form $\mathtt{t}'$ such that $\mathtt{t} \longrightarrow^* \mathtt{t}'$.
Proof:

- First, recall that single-step evaluation strictly reduces the size of the term:

  if $t \longrightarrow t'$, *then* $size(t) > size(t')$

- Now, assume (for a contradiction) that

  $t_0,\ t_1,\ t_2,\ t_3,\ t_4,\ \ldots$

  is an infinite-length sequence such that

  $t_0 \longrightarrow t_1 \longrightarrow t_2 \longrightarrow t_3 \longrightarrow t_4 \longrightarrow \cdots.$

- Then

  $size(t_0) > size(t_1) > size(t_2) > size(t_3) > \ldots$

- But such a sequence cannot exist — contradiction!

Termination Proofs

Most termination proofs have the same basic form:

Theorem: *The relation $R \subseteq X \times X$ is terminating — i.e., there are no infinite sequences x_0, x_1, x_2, etc. such that $(x_i, x_{i+1}) \in R$ for each i.*

Proof:

1. *Choose*
   - *a well-founded set $(W, <)$ — i.e., a set W with a partial order $<$ such that there are no infinite descending chains $w_0 > w_1 > w_2 > \ldots$ in W*
   - *a function f from X to W*
2. *Show $f(x) > f(y)$ for all $(x, y) \in R$*
3. *Conclude that there are no infinite sequences x_0, x_1, x_2, etc. such that $(x_i, x_{i+1}) \in R$ for each i, since, if there were, we could construct an infinite descending chain in W.*

The Lambda Calculus

The lambda-calculus

- If our previous language of arithmetic expressions was the simplest nontrivial programming language, then the lambda-calculus is the simplest *interesting* programming language...
  - Turing complete
  - higher order (functions as data)
- Indeed, in the lambda-calculus, *all* computation happens by means of function abstraction and application.
- The *e. coli* of programming language research
- The foundation of many real-world programming language designs (including ML, Haskell, Scheme, Lisp, ...)

Intuitions

Suppose we want to describe a function that adds three to any number we pass it. We might write

```
plus3 x   =   succ (succ (succ x))
```

That is, "`plus3 x` is `succ (succ (succ x))`."

Intuitions

Suppose we want to describe a function that adds three to any number we pass it. We might write

$$\texttt{plus3 x} \quad = \quad \texttt{succ (succ (succ x))}$$

That is, "`plus3 x` is `succ (succ (succ x))`."

Q: What is `plus3` itself?

Intuitions

Suppose we want to describe a function that adds three to any number we pass it. We might write

$$\texttt{plus3 x} \quad = \quad \texttt{succ (succ (succ x))}$$

That is, "`plus3 x` is `succ (succ (succ x))`."

Q: What is `plus3` itself?

A: `plus3` is the function that, given `x`, yields `succ (succ (succ x))`.

Intuitions

Suppose we want to describe a function that adds three to any number we pass it. We might write

$$\texttt{plus3 x} \quad = \quad \texttt{succ (succ (succ x))}$$

That is, "`plus3 x` is `succ (succ (succ x))`."

Q: What is `plus3` itself?

A: `plus3` is the function that, given `x`, yields `succ (succ (succ x))`.

$$\texttt{plus3} \quad = \quad \lambda\texttt{x. succ (succ (succ x))}$$

This function exists independent of the name `plus3`.

λ`x. t` is written "`fun x` $\rightarrow$ `t`" in OCaml and "`x` $\Rightarrow$ `t`" in Scala.

So `plus3 (succ 0)` is just a convenient shorthand for "the function that, given `x`, yields `succ (succ (succ x))`, applied to `succ 0`."

$$\begin{array}{c}\texttt{plus3 (succ 0)} \\ = \\ (\lambda\texttt{x. succ (succ (succ x))) (succ 0)}\end{array}$$

Abstractions over Functions

Consider the λ-abstraction

```
g   =  λf. f (f (succ 0))
```

Note that the parameter variable `f` is used in the *function* position in the body of `g`. Terms like `g` are called *higher-order* functions.
If we apply `g` to an argument like `plus3`, the "substitution rule" yields a nontrivial computation:

```
g plus3
  =    (λf. f (f (succ 0))) (λx. succ (succ (succ x)))
  i.e. (λx. succ (succ (succ x)))
         ((λx. succ (succ (succ x))) (succ 0))
  i.e. (λx. succ (succ (succ x)))
         (succ (succ (succ (succ 0))))
  i.e. succ (succ (succ (succ (succ (succ (succ 0))))))
```

Abstractions Returning Functions

Consider the following variant of `g`:

```
double   =  λf. λy. f (f y)
```

I.e., `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)`.

Example

```
       double plus3 0
  =    (λf. λy. f (f y))
         (λx. succ (succ (succ x)))
         0
  i.e. (λy. (λx. succ (succ (succ x)))
               ((λx. succ (succ (succ x))) y))
         0
  i.e. (λx. succ (succ (succ x)))
               ((λx. succ (succ (succ x))) 0)
  i.e. (λx. succ (succ (succ x)))
               (succ (succ (succ 0)))
  i.e. succ (succ (succ (succ (succ (succ 0))))))
```

The Pure Lambda-Calculus

As the preceding examples suggest, once we have λ-abstraction and application, we can throw away all the other language primitives and still have left a rich and powerful programming language.

In this language — the "pure lambda-calculus"— *everything* is a function.

- Variables always denote functions
- Functions always take other functions as parameters
- The result of a function is always a function

Formalities

Syntax

`t`	::=		*terms*
		`x`	*variable*
		`λx.t`	*abstraction*
		`t t`	*application*

*Term*inology:

- terms in the pure λ-calculus are often called λ-*terms*
- terms of the form `λx. t` are called λ-*abstractions* or just *abstractions*

Syntactic conventions

Since λ-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., $t\ u\ v$ *means* $(t\ u)\ v$, *not* $t\ (u\ v)$

- Bodies of λ- abstractions extend as far to the right as possible

  E.g., $\lambda x.\ \lambda y.\ x\ y$ *means* $\lambda x.\ (\lambda y.\ x\ y)$, *not* $\lambda x.\ (\lambda y.\ x)\ y$

Scope

The λ-abstraction term `λx.t` *binds* the variable `x`.

The *scope* of this binding is the *body* `t`.

Occurrences of `x` inside `t` are said to be *bound* by the abstraction.

Occurrences of `x` that are *not* within the scope of an abstraction binding `x` are said to be *free*.

Test:

`λx. λy. x y z`

Scope

The λ-abstraction term $\lambda x.t$ *binds* the variable x.

The *scope* of this binding is the *body* t.

Occurrences of x inside t are said to be *bound* by the abstraction.

Occurrences of x that are *not* within the scope of an abstraction binding x are said to be *free*.

Test:

$$\lambda x.\ \lambda y.\ x\ y\ z$$
$$\lambda x.\ (\lambda y.\ z\ y)\ y$$

Values

v ::=		*values*
	$\lambda x.t$	*abstraction value*

Operational Semantics

Computation rule:

$$(\lambda x.t_{12})\ v_2 \longrightarrow [x \mapsto v_2]t_{12} \qquad \text{(E-AppAbs)}$$

Notation: $[x \mapsto v_2]t_{12}$ *is "the term that results from substituting free occurrences of* x *in* t_{12} *with* v_2*."*

Operational Semantics

Computation rule:

$$(\lambda x.t_{12})\ v_2 \longrightarrow [x \mapsto v_2]t_{12} \qquad \text{(E-AppAbs)}$$

Notation: $[x \mapsto v_2]t_{12}$ *is "the term that results from substituting free occurrences of* x *in* t_{12} *with* v_2*."*

Congruence rules:

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \qquad \text{(E-App1)}$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \qquad \text{(E-App2)}$$

Terminology

A term of the form $(\lambda x.t)\ v$ — that is, a λ-abstraction applied to a *value* — is called a *redex* (short for "reducible expression").

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

Classical Lambda Calculus

Full beta reduction

The classical lambda calculus allows full beta reduction.

- The argument of a β-reduction to be an arbitrary term, not just a value.
- Reduction may appear anywhere in a term.

Full beta reduction

The classical lambda calculus allows full beta reduction.

- The argument of a β-reduction to be an arbitrary term, not just a value.
- Reduction may appear anywhere in a term.

Computation rule:

$$(\lambda \mathtt{x}.\mathtt{t}_{12})\ \mathtt{t}_2 \longrightarrow [\mathtt{x} \mapsto \mathtt{t}_2]\mathtt{t}_{12} \qquad \text{(E-AppAbs)}$$

Full beta reduction

The classical lambda calculus allows full beta reduction.

- The argument of a β-reduction to be an arbitrary term, not just a value.
- Reduction may appear anywhere in a term.

Computation rule:

$$(\lambda \mathtt{x}.\mathtt{t}_{12})\ \mathtt{t}_2 \longrightarrow [\mathtt{x} \mapsto \mathtt{t}_2]\mathtt{t}_{12} \qquad \text{(E-AppAbs)}$$

Congruence rules:

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{t}_1\ \mathtt{t}_2 \longrightarrow \mathtt{t}_1'\ \mathtt{t}_2} \qquad \text{(E-App1)}$$

$$\frac{\mathtt{t}_2 \longrightarrow \mathtt{t}_2'}{\mathtt{t}_1\ \mathtt{t}_2 \longrightarrow \mathtt{t}_1\ \mathtt{t}_2'} \qquad \text{(E-App2)}$$

$$\frac{\mathtt{t} \longrightarrow \mathtt{t}'}{\lambda \mathtt{x}.\mathtt{t} \longrightarrow \lambda \mathtt{x}.\mathtt{t}'} \qquad \text{(E-Abs)}$$

Substitution revisited

Remember: $[x \mapsto v_2]t_{12}$ *is "the term that results from substituting free occurrences of* x *in* t_{12} *with* v_2*."*

This is trickier than it looks!
For example:

$$\begin{array}{ll} & (\lambda \mathtt{x}.\ (\lambda \mathtt{y}.\ \mathtt{x}))\ \mathtt{y} \\ \longrightarrow & [\mathtt{x} \mapsto \mathtt{y}]\lambda \mathtt{y}.\ \mathtt{x} \\ = & ??? \end{array}$$

Substitution revisited

Remember: $[x \mapsto v_2]t_{12}$ *is "the term that results from substituting free occurrences of* x *in* t_{12} *with* v_2*."*

This is trickier than it looks!
For example:

$$\begin{array}{ll} & (\lambda \mathtt{x}.\ (\lambda \mathtt{y}.\ \mathtt{x}))\ \mathtt{y} \\ \longrightarrow & [\mathtt{x} \mapsto \mathtt{y}]\lambda \mathtt{y}.\ \mathtt{x} \\ = & ??? \end{array}$$

Solution:
need to rename bound variables before performing the substitution.

$$\begin{array}{ll} & (\lambda \mathtt{x}.\ (\lambda \mathtt{y}.\ \mathtt{x}))\ \mathtt{y} \\ = & (\lambda \mathtt{x}.\ (\lambda \mathtt{z}.\ \mathtt{x}))\ \mathtt{y} \\ \longrightarrow & [\mathtt{x} \mapsto \mathtt{y}]\lambda \mathtt{z}.\ \mathtt{x} \\ = & \lambda \mathtt{z}.\ \mathtt{y} \end{array}$$

Alpha conversion

Renaming bound variables is formalized as α-conversion.
Conversion rule:

$$\frac{\mathtt{y} \notin \mathtt{fv(t)}}{\lambda\mathtt{x.\ t} =_\alpha \lambda\mathtt{y}.[\mathtt{x} \mapsto \mathtt{y}]\mathtt{t}} \qquad (\alpha)$$

Equivalence rules:

$$\frac{\mathtt{t}_1 =_\alpha \mathtt{t}_2}{\mathtt{t}_2 =_\alpha \mathtt{t}_1} \qquad (\alpha\text{-}\textsc{Symm})$$

$$\frac{\mathtt{t}_1 =_\alpha \mathtt{t}_2 \qquad \mathtt{t}_2 =_\alpha \mathtt{t}_3}{\mathtt{t}_1 =_\alpha \mathtt{t}_3} \qquad (\alpha\text{-}\textsc{Trans})$$

Congruence rules: the usual ones.

Confluence

Full β-reduction makes it possible to have different reduction paths.

Q: Can a term evaluate to more than one normal form?

Confluence

Full β-reduction makes it possible to have different reduction paths.

Q: Can a term evaluate to more than one normal form?

The answer is no; this is a consequence of the following

Theorem [Church-Rosser]
Let $\mathtt{t}$, $\mathtt{t}_1$, $\mathtt{t}_2$ be terms such that $\mathtt{t} \longrightarrow^* \mathtt{t}_1$ and $\mathtt{t} \longrightarrow^* \mathtt{t}_2$. Then there exists a term $\mathtt{t}_3$ such that $\mathtt{t}_1 \longrightarrow^* \mathtt{t}_3$ and $\mathtt{t}_2 \longrightarrow^* \mathtt{t}_3$.

Programming in the Lambda-Calculus

Multiple arguments

Consider the function `double`, which returns a function as an argument.

$$\texttt{double} \quad = \quad \lambda\texttt{f.}\ \lambda\texttt{y.}\ \texttt{f (f y)}$$

This idiom — a λ-abstraction that does nothing but immediately yield another abstraction — is very common in the λ-calculus.

In general, λ`x.` λ`y.` `t` is a function that, given a value `v` for `x`, yields a function that, given a value `u` for `y`, yields `t` with `v` in place of `x` and `u` in place of `y`.

That is, λ`x.` λ`y.` `t` is a two-argument function.

(Recall the discussion of *currying* in OCaml.)

The "Church Booleans"

```
tru  =  λt. λf. t
fls  =  λt. λf. f
```

$$\begin{array}{lll} & \texttt{tru v w} & \\ = & \underline{(\lambda\texttt{t.}\lambda\texttt{f.t})\ \texttt{v}}\ \texttt{w} & \text{by definition} \\ \longrightarrow & \underline{(\lambda\texttt{f. v})\ \texttt{w}} & \text{reducing the underlined redex} \\ \longrightarrow & \texttt{v} & \text{reducing the underlined redex} \end{array}$$

$$\begin{array}{lll} & \texttt{fls v w} & \\ = & \underline{(\lambda\texttt{t.}\lambda\texttt{f.f})\ \texttt{v}}\ \texttt{w} & \text{by definition} \\ \longrightarrow & \underline{(\lambda\texttt{f. f})\ \texttt{w}} & \text{reducing the underlined redex} \\ \longrightarrow & \texttt{w} & \text{reducing the underlined redex} \end{array}$$

Functions on Booleans

$$\texttt{not} \quad = \quad \lambda\texttt{b.}\ \texttt{b fls tru}$$

That is, `not` is a function that, given a boolean value `v`, returns `fls` if `v` is `tru` and `tru` if `v` is `fls`.

Functions on Booleans

$$\texttt{and} \quad = \quad \lambda\texttt{b.}\ \lambda\texttt{c.}\ \texttt{b c fls}$$

That is, `and` is a function that, given two boolean values `v` and `w`, returns `w` if `v` is `tru` and `fls` if `v` is `fls`
Thus `and v w` yields `tru` if both `v` and `w` are `tru` and `fls` if either `v` or `w` is `fls`.

Pairs

```
pair = λf.λs.λb. b f s
fst = λp. p tru
snd = λp. p fls
```

That is, `pair v w` is a function that, when applied to a boolean value `b`, applies `b` to `v` and `w`.
By the definition of booleans, this application yields `v` if `b` is `tru` and `w` if `b` is `fls`, so the first and second projection functions `fst` and `snd` can be implemented simply by supplying the appropriate boolean.

Example

	`fst (pair v w)`	
$=$	`fst ((λf. λs. λb. b f s) v w)`	by definition
$\longrightarrow$	`fst ((λs. λb. b v s) w)`	reducing
$\longrightarrow$	`fst (λb. b v w)`	reducing
$=$	`(λp. p tru) (λb. b v w)`	by definition
$\longrightarrow$	`(λb. b v w) tru`	reducing
$\longrightarrow$	`tru v w`	reducing
$\longrightarrow^*$	`v`	as before.

Church numerals

Idea: represent the number n by a function that "repeats some action n times."

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))

That is, each number n 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`.

Functions on Church Numerals

Successor:

Functions on Church Numerals

Successor:

```
scc = λn. λs. λz. s (n s z)
```

Functions on Church Numerals

Successor:

```
scc = λn. λs. λz. s (n s z)
```

Addition:

Functions on Church Numerals

Successor:

```
scc = λn. λs. λz. s (n s z)
```

Addition:

```
plus = λm. λn. λs. λz. m s (n s z)
```

Functions on Church Numerals

Successor:

```
scc = λn. λs. λz. s (n s z)
```

Addition:

```
plus = λm. λn. λs. λz. m s (n s z)
```

Multiplication:

Functions on Church Numerals

Successor:

```
scc = λn. λs. λz. s (n s z)
```

Addition:

```
plus = λm. λn. λs. λz. m s (n s z)
```

Multiplication:

```
times = λm. λn. m (plus n) c0
```

Functions on Church Numerals

Successor:

```
scc = λn. λs. λz. s (n s z)
```

Addition:

```
plus = λm. λn. λs. λz. m s (n s z)
```

Multiplication:

```
times = λm. λn. m (plus n) c0
```

Zero test:

Functions on Church Numerals

Successor:

```
scc = λn. λs. λz. s (n s z)
```

Addition:

```
plus = λm. λn. λs. λz. m s (n s z)
```

Multiplication:

```
times = λm. λn. m (plus n) c0
```

Zero test:

```
iszro = λm. m (λx. fls) tru
```

Functions on Church Numerals

Successor:

```
scc = λn. λs. λz. s (n s z)
```

Addition:

```
plus = λm. λn. λs. λz. m s (n s z)
```

Multiplication:

```
times = λm. λn. m (plus n) c0
```

Zero test:

```
iszro = λm. m (λx. fls) tru
```

What about predecessor?

Predecessor

```
zz = pair c0 c0

ss = λp. pair (snd p) (scc (snd p))

prd = λm. fst (m ss zz)
```

Recursion in the Lambda-Calculus

Recursion and divergence

Recursion and divergence are intertwined, so we need to consider divergent terms.

```
omega  =  (λx. x x) (λx. x x)
```

Note that omega evaluates in one step to itself!
So evaluation of omega never reaches a normal form: it *diverges*.

Recursion and divergence

Recursion and divergence are intertwined, so we need to consider divergent terms.

```
omega  =  (λx. x x) (λx. x x)
```

Note that omega evaluates in one step to itself!
So evaluation of omega never reaches a normal form: it *diverges*.

Being able to write a divergent computation does not seem very useful in itself. However, there are variants of omega that are *very* useful...

Recall: Normal forms

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

Does every term evaluate to a normal form?

No, `omega` is not in normal form.

Recall: Normal forms

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

Does every term evaluate to a normal form?

No, `omega` is not in normal form.

But are there any stuck terms in the pure λ-calculus?

Towards recursion: Iterated application

Suppose `f` is some λ-abstraction, and consider the following variant of `omega`:

$$\mathtt{Y}_f \quad = \quad (\lambda\mathtt{x.\ f\ (x\ x))\ (}\lambda\mathtt{x.\ f\ (x\ x))}$$

Towards recursion: Iterated application

Suppose `f` is some λ-abstraction, and consider the following variant of `omega`:

$$\mathtt{Y}_f \quad = \quad (\lambda\mathtt{x.\ f\ (x\ x))\ (}\lambda\mathtt{x.\ f\ (x\ x))}$$

Now the "pattern of divergence" becomes more interesting:

$$\begin{array}{c}
\mathtt{Y}_f \\
= \\
\underline{(\lambda\mathtt{x.\ f\ (x\ x))\ (}\lambda\mathtt{x.\ f\ (x\ x))}} \\
\longrightarrow \\
\mathtt{f\ (}\underline{(\lambda\mathtt{x.\ f\ (x\ x))\ (}\lambda\mathtt{x.\ f\ (x\ x))}}\mathtt{)} \\
\longrightarrow \\
\mathtt{f\ (f\ (}\underline{(\lambda\mathtt{x.\ f\ (x\ x))\ (}\lambda\mathtt{x.\ f\ (x\ x))}}\mathtt{))} \\
\longrightarrow \\
\mathtt{f\ (f\ (f\ (}\underline{(\lambda\mathtt{x.\ f\ (x\ x))\ (}\lambda\mathtt{x.\ f\ (x\ x))}}\mathtt{)))} \\
\longrightarrow \\
\ldots
\end{array}$$

$\texttt{Y}_f$ is still not very useful, since (like `omega`), all it does is diverge.

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

Delaying divergence

$$\texttt{poisonpill} \quad = \quad \lambda\texttt{y. omega}$$

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.

$$\begin{array}{c}
\underline{\texttt{(}\lambda\texttt{p. fst (pair p fls) tru) poisonpill}} \\
\longrightarrow \\
\texttt{fst (pair poisonpill fls) tru} \\
\longrightarrow^{*} \\
\underline{\texttt{poisonpill tru}} \\
\longrightarrow \\
\texttt{omega} \\
\longrightarrow \\
\cdots
\end{array}$$

A delayed variant of omega

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

$$\begin{array}{c}
\texttt{omegav} \quad = \\
\lambda\texttt{y. (}\lambda\texttt{x. (}\lambda\texttt{y. x x y)) (}\lambda\texttt{x. (}\lambda\texttt{y. x x y)) y}
\end{array}$$

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

$$\begin{array}{c}
\texttt{omegav v} \\
= \\
\underline{\texttt{(}\lambda\texttt{y. (}\lambda\texttt{x. (}\lambda\texttt{y. x x y)) (}\lambda\texttt{x. (}\lambda\texttt{y. x x y)) y) v}} \\
\longrightarrow \\
\underline{\texttt{(}\lambda\texttt{x. (}\lambda\texttt{y. x x y)) (}\lambda\texttt{x. (}\lambda\texttt{y. x x y))}}\texttt{ v} \\
\longrightarrow \\
\texttt{(}\lambda\texttt{y. (}\lambda\texttt{x. (}\lambda\texttt{y. x x y)) (}\lambda\texttt{x. (}\lambda\texttt{y. x x y)) y) v} \\
= \\
\texttt{omegav v}
\end{array}$$

Another delayed variant

Suppose `f` is a function. Define

$$\texttt{z}_f \quad = \quad \lambda\texttt{y. (}\lambda\texttt{x. f (}\lambda\texttt{y. x x y)) (}\lambda\texttt{x. f (}\lambda\texttt{y. x x y)) y}$$

This term combines the "added `f`" from $\texttt{Y}_f$ with the "delayed divergence" of `omegav`.

If we now apply z_f to an argument v, something interesting happens:

$$
\begin{array}{c}
z_f \ v \\
= \\
\underline{(\lambda y.\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))\ y)\ v} \\
\longrightarrow \\
\underline{(\lambda x.\ f\ (\lambda y.\ x\ x\ y))\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))}\ v \\
\longrightarrow \\
f\ (\lambda y.\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))\ y)\ v \\
= \\
f\ z_f\ v
\end{array}
$$

Since z_f and v are both values, the next computation step will be the reduction of f 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.).

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

$$
\begin{array}{c}
z_f\ 3 \\
\longrightarrow^* \\
f\ z_f\ 3 \\
= \\
(\lambda fct.\ \lambda n.\ \ldots)\ z_f\ 3 \\
\longrightarrow\ \ \longrightarrow \\
\texttt{if } 3{=}0 \texttt{ then } 1 \texttt{ else } 3 * (z_f\ (\texttt{pred } 3)) \\
\longrightarrow^* \\
3 * (z_f\ (\texttt{pred } 3))) \\
\longrightarrow \\
3 * (z_f\ 2) \\
\longrightarrow^* \\
3 * (f\ z_f\ 2) \\
\ldots
\end{array}
$$

A Generic z

If we define

$$z \quad = \quad \lambda f.\ z_f$$

i.e.,

$$
\begin{array}{c}
z \quad = \\
\lambda f.\ \lambda y.\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))\ y
\end{array}
$$

then we can obtain the behavior of z_f for any f we like, simply by applying z to f.

$$z\ f \quad \longrightarrow \quad z_f$$

For example:

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

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.