

Foundations of Software Fall 2023

Week 3

1

Review (and more details)

2

Recall: Simple Arithmetic Expressions

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

$t ::=$	<i>terms</i>
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

3

Recall: Inference Rule Notation

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

$$\begin{array}{c} \text{true} \in \mathcal{T} \qquad \text{false} \in \mathcal{T} \qquad 0 \in \mathcal{T} \\[1em] \frac{t_1 \in \mathcal{T}}{\text{succ } t_1 \in \mathcal{T}} \qquad \frac{t_1 \in \mathcal{T}}{\text{pred } t_1 \in \mathcal{T}} \qquad \frac{t_1 \in \mathcal{T}}{\text{iszero } t_1 \in \mathcal{T}} \\[1em] \frac{t_1 \in \mathcal{T} \quad t_2 \in \mathcal{T} \quad t_3 \in \mathcal{T}}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \in \mathcal{T}} \end{array}$$

4

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.”

$$\begin{array}{c} \text{true} \in \mathcal{T} \qquad \text{false} \in \mathcal{T} \qquad 0 \in \mathcal{T} \\ \hline \frac{t_1 \in \mathcal{T}}{\text{succ } t_1 \in \mathcal{T}} \qquad \frac{t_1 \in \mathcal{T}}{\text{pred } t_1 \in \mathcal{T}} \qquad \frac{t_1 \in \mathcal{T}}{\text{iszero } t_1 \in \mathcal{T}} \\ \hline \frac{t_1 \in \mathcal{T} \quad t_2 \in \mathcal{T} \quad t_3 \in \mathcal{T}}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \in \mathcal{T}} \end{array}$$

5

Let's write these generating functions explicitly.

$$\begin{aligned} F_1(U) &= \{\text{true}\} \\ F_2(U) &= \{\text{false}\} \\ F_3(U) &= \{0\} \\ F_4(U) &= \{\text{succ } t_1 \mid t_1 \in U\} \\ F_5(U) &= \{\text{pred } t_1 \mid t_1 \in U\} \\ F_6(U) &= \{\text{iszero } t_1 \mid t_1 \in U\} \\ F_7(U) &= \{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \mid t_1, t_2, t_3 \in U\} \end{aligned}$$

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

6

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

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

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 $\mathcal{T} \subseteq \mathcal{O}$.)

7

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

$$\begin{aligned} S_0 &= \emptyset \\ S_{i+1} &= F(S_i) \end{aligned}$$

$$S = \bigcup_i S_i$$

Compare this definition of S with the one we saw last time:

$$\begin{aligned} S_0 &= \emptyset \\ S_{i+1} &= \{\text{true, false, 0}\} \\ &\cup \{\text{succ } t_1, \text{pred } t_1, \text{iszero } t_1 \mid t_1 \in S_i\} \\ &\cup \{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \mid t_1, t_2, t_3 \in S_i\} \end{aligned}$$

$$S = \bigcup_i S_i$$

We have “pulled out” F and given it a name.

8

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.

9

Warning: Hard hats on for the next slide!

10

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 $t \in T$.

11

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?

11

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.
(recall: O is F -closed if $F(O) \subseteq O$)

- ▶ Since $T \subseteq O$, every element of T satisfies P .

12

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 .*

13

Recall, from the definition of \mathcal{S} , it is clear that, if a term 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 s , given $P(r)$ for all immediate subterms of s , we can show $P(s)$.
- ▶ Then show, by induction on i , that $P(t)$ holds for all terms t with depth i .
- ▶ Therefore, $P(t)$ holds for all t .

14

Operational Semantics and Reasoning

15

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.

16

Recall: Syntax for Booleans

Terms and values

```
t ::=
    true
    false
    if t then t else t
```

terms
constant true
constant false
conditional

```
v ::=
    true
    false
```

values
true value
false value

17

Recall: Operational Semantics for Booleans

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

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

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

$$\frac{t_1 \longrightarrow t'_1}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \longrightarrow \text{if } t'_1 \text{ then } t_2 \text{ else } t_3} \text{ (E-IF)}$$

18

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.

19

Observation

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

1. the final rule used in \mathcal{D} is E-IFTRUE and we have $t = \text{if true then } t_2 \text{ else } t_3$ and $t' = t_2$, for some t_2 and t_3 , or
2. the final rule used in \mathcal{D} is E-IFFALSE and we have $t = \text{if false then } t_2 \text{ else } t_3$ and $t' = t_3$, for some t_2 and t_3 , or
3. the final rule used in \mathcal{D} is E-IF and we have $t = \text{if } t_1 \text{ then } t_2 \text{ else } t_3$ and $t' = \text{if } t'_1 \text{ then } t_2 \text{ else } t_3$, for some t_1, t'_1, t_2 , and t_3 ; moreover, the immediate subderivation of \mathcal{D} witnesses $(t_1, t'_1) \in \rightarrow$.

20

Induction on Derivations

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

Given an arbitrary derivation \mathcal{D} with conclusion $t \rightarrow t'$, 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....

21

Induction on Derivations — Example

Theorem: If $t \rightarrow t'$, i.e., if $(t, t') \in \rightarrow$, then $\text{size}(t) > \text{size}(t')$.

Proof: By induction on a derivation \mathcal{D} of $t \rightarrow t'$.

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

22

Normal forms

A *normal form* is a term that cannot be evaluated any further — i.e., a term t is a normal form (or “is in normal form”) if there is no t' such that $t \rightarrow t'$.

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

23

Normal forms

A *normal form* is a term that cannot be evaluated any further — i.e., a term t is a normal form (or “is in normal form”) if there is no t' such that $t \rightarrow t'$.

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?

23

Values = normal forms

Theorem: A term t is a value iff it is in normal form.

Proof:

The \Rightarrow direction is immediate from the definition of the evaluation relation.

24

Values = normal forms

Theorem: A term t is a value iff it is in normal form.

Proof:

The \Rightarrow direction is immediate from the definition of the evaluation relation.

For the \Leftarrow direction,

24

Values = normal forms

Theorem: A term t is a value iff it is in normal form.

Proof:

The \Rightarrow direction is immediate from the definition of the evaluation relation.

For the \Leftarrow direction, it is convenient to prove the contrapositive:

If t is *not* a value, then it is *not* a normal form.

24

Values = normal forms

Theorem: A term t is a value iff it is in normal form.

Proof:

The \Rightarrow direction is immediate from the definition of the evaluation relation.

For the \Leftarrow 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 $t_1 \rightarrow t'_1$. But then rule E-IF yields

`if t_1 then t_2 else $t_3 \rightarrow$ if t'_1 then t_2 else t_3`

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

24

Numbers

New syntactic forms

$t ::= \dots$
`0`
`succ t`
`pred t`
`iszero t`

terms
constant zero
successor
predecessor
zero test

$v ::= \dots$
`nv`

values
numeric value

$nv ::=$
`0`
`succ nv`

numeric values
zero value
successor value

25

New evaluation rules

$t \rightarrow t'$

$$\frac{t_1 \rightarrow t'_1}{\text{succ } t_1 \rightarrow \text{succ } t'_1} \quad (\text{E-SUCC})$$

$$\text{pred } 0 \rightarrow 0 \quad (\text{E-PREDZERO})$$

$$\text{pred } (\text{succ } nv_1) \rightarrow nv_1 \quad (\text{E-PREDSUCC})$$

$$\frac{t_1 \rightarrow t'_1}{\text{pred } t_1 \rightarrow \text{pred } t'_1} \quad (\text{E-PRED})$$

$$\text{iszero } 0 \rightarrow \text{true} \quad (\text{E-ISZEROZERO})$$

$$\text{iszero } (\text{succ } nv_1) \rightarrow \text{false} \quad (\text{E-ISZEROSUCC})$$

$$\frac{t_1 \rightarrow t'_1}{\text{iszero } t_1 \rightarrow \text{iszero } t'_1} \quad (\text{E-ISZERO})$$

26

Are all normal forms still values?

Clicker question: Propose a term that is in normal form but is *not* a value.

URL: ttpoll.eu
 Session ID: [cs452](#)

27

Stuck terms

Formally, a stuck term is one that is a normal form but not a value.

Stuck terms model run-time errors.

28

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:

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

29

Termination of evaluation

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

Proof:

30

Termination of evaluation

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

Proof:

- ▶ First, recall that single-step evaluation strictly reduces the size of the term:
if $t \longrightarrow t'$, then $\text{size}(t) > \text{size}(t')$
- ▶ Now, assume (for a contradiction) that
 $t_0, t_1, t_2, t_3, t_4, \dots$
is an infinite-length sequence such that
 $t_0 \longrightarrow t_1 \longrightarrow t_2 \longrightarrow t_3 \longrightarrow t_4 \longrightarrow \dots$
- ▶ Then
 $\text{size}(t_0) > \text{size}(t_1) > \text{size}(t_2) > \text{size}(t_3) > \dots$
- ▶ But such a sequence cannot exist — contradiction!

30

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 $w_0 > w_1 > w_2 > \dots$ 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 .

31

Reading for next week

- ▶ Chapter 4 – Untyped Lambda Calculus (20 pages!)

32

The Lambda Calculus

33

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, ...)

34

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))`.”

35

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))`.”

Q: What is `plus3` itself?

35

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))`.”

Q: What is `plus3` itself?

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

35

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))`.”

Q: What is `plus3` itself?

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

```
plus3 = λx. succ (succ (succ x))
```

This function exists independent of the name `plus3`.

`λx. t` is written “`fun x → t`” in OCaml and “`x ⇒ t`” in Scala.

35

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

$$\begin{aligned} \text{plus3 (succ 0)} \\ = \\ (\lambda x. \text{succ (succ (succ x))}) (\text{succ 0}) \end{aligned}$$

36

Abstractions over Functions

Consider the λ -abstraction

$$g = \lambda f. f (f (\text{succ } 0))$$

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:

$$\begin{aligned} g \text{ plus3} \\ = & (\lambda f. f (f (\text{succ } 0))) (\lambda x. \text{succ (succ (succ x))}) \\ \text{i.e. } & (\lambda x. \text{succ (succ (succ x))}) \\ & ((\lambda x. \text{succ (succ (succ x))}) (\text{succ } 0)) \\ \text{i.e. } & (\lambda x. \text{succ (succ (succ x))}) \\ & (\text{succ (succ (succ (succ 0)))}) \\ \text{i.e. } & \text{succ (succ (succ (succ (succ (succ (succ 0))))))} \end{aligned}$$

37

Abstractions Returning Functions

Consider the following variant of `g`:

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

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)`.

38

Example

$$\begin{aligned} & \text{double plus3 0} \\ = & (\lambda f. \lambda y. f (f y)) \\ & (\lambda x. \text{succ (succ (succ x))}) \\ & 0 \\ \text{i.e. } & (\lambda y. (\lambda x. \text{succ (succ (succ x))}) \\ & ((\lambda x. \text{succ (succ (succ x))}) y)) \\ & 0 \\ \text{i.e. } & (\lambda x. \text{succ (succ (succ x))}) \\ & ((\lambda x. \text{succ (succ (succ x))}) 0) \\ \text{i.e. } & (\lambda x. \text{succ (succ (succ x))}) \\ & (\text{succ (succ (succ 0))}) \\ \text{i.e. } & \text{succ (succ (succ (succ (succ (succ 0))))} \end{aligned}$$

39

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

40

Formalities

41

Syntax

$t ::=$	<i>terms</i>
x	<i>variable</i>
$\lambda x. t$	<i>abstraction</i>
$t\ t$	<i>application</i>

Terminology:

- ▶ terms in the pure λ -calculus are often called λ -terms
- ▶ terms of the form $\lambda x. t$ are called λ -abstractions or just *abstractions*

42

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

43

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:

$\lambda x. \lambda y. x y z$

44

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:

$\lambda x. \lambda y. x y z$

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

44

Values

$v ::=$
 $\lambda x. t$

values
abstraction value

45

Operational Semantics

Computation rule:

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

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

46

Operational Semantics

Computation rule:

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

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:

$$\frac{t_1 \longrightarrow t'_1}{t_1 \ t_2 \longrightarrow t'_1 \ t_2} \quad (\text{E-APP1})$$

$$\frac{t_2 \longrightarrow t'_2}{v_1 \ t_2 \longrightarrow v_1 \ t'_2} \quad (\text{E-APP2})$$

46

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”).

47

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

48

Classical Lambda Calculus

49

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.

50

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:

$$(\lambda x. t_{12}) t_2 \longrightarrow [x \mapsto t_2] t_{12} \quad (\text{E-APPABS})$$

50

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:

$$(\lambda x. t_{12}) t_2 \longrightarrow [x \mapsto t_2] t_{12} \quad (\text{E-APPABS})$$

Congruence rules:

$$\frac{t_1 \longrightarrow t'_1}{t_1 t_2 \longrightarrow t'_1 t_2} \quad (\text{E-APP1})$$

$$\frac{t_2 \longrightarrow t'_2}{t_1 t_2 \longrightarrow t_1 t'_2} \quad (\text{E-APP2})$$

$$\frac{t \longrightarrow t'}{\lambda x. t \longrightarrow \lambda x. t'} \quad (\text{E-ABS})$$

50

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:

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

51

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:

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

Solution:

need to rename bound variables before performing the substitution.

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

51

Alpha conversion

Renaming bound variables is formalized as α -conversion.

Conversion rule:

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

Equivalence rules:

$$\frac{t_1 =_{\alpha} t_2}{t_2 =_{\alpha} t_1} \quad (\alpha\text{-SYMM})$$

$$\frac{t_1 =_{\alpha} t_2 \quad t_2 =_{\alpha} t_3}{t_1 =_{\alpha} t_3} \quad (\alpha\text{-TRANS})$$

Congruence rules: the usual ones.

52

Confluence

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

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

53

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 t, t_1, t_2 be terms such that $t \longrightarrow^* t_1$ and $t \longrightarrow^* t_2$. Then there exists a term t_3 such that $t_1 \longrightarrow^* t_3$ and $t_2 \longrightarrow^* t_3$.

53

Programming in the Lambda-Calculus

54

Multiple arguments

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

```
double = λf. λy. 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, $\lambda x. \lambda y. t$ 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, $\lambda x. \lambda y. t$ is a two-argument function.

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

55

The “Church Booleans”

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

```
tru v w  
= (λt. λf. t) v w by definition  
→ (λf. v) w      reducing the underlined redex  
→ v              reducing the underlined redex
```

```
fls v w  
= (λt. λf. f) v w by definition  
→ (λf. f) w      reducing the underlined redex  
→ w              reducing the underlined redex
```

56

Functions on Booleans

```
not = λb. 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`.

57

Functions on Booleans

`and = λb. λc. 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`.

58

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.

59

Example

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

60

Church numerals

Idea: represent the number `n` by a function that “repeats some action `n` times.”

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

That is, each number `n` is represented by a term `cn` that takes two arguments, `s` and `z` (for “successor” and “zero”), and applies `s`, `n` times, to `z`.

61

Functions on Church Numerals

Successor:

62

Functions on Church Numerals

Successor:

$scc = \lambda n. \lambda s. \lambda z. s (n s z)$

62

Functions on Church Numerals

Successor:

$scc = \lambda n. \lambda s. \lambda z. s (n s z)$

Addition:

62

Functions on Church Numerals

Successor:

$scc = \lambda n. \lambda s. \lambda z. s (n s z)$

Addition:

$plus = \lambda m. \lambda n. \lambda s. \lambda z. m s (n s z)$

62

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:

62

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
```

62

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:

62

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
```

62

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?

62

Predecessor

```
zz = pair c0 c0
```

```
ss = λp. pair (snd p) (scc (snd p))
```

```
prd = λm. fst (m ss zz)
```

63

Recursion in the Lambda-Calculus

64

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*.

65

Recursion and divergence

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

$$\text{omega} = (\lambda x. x x) (\lambda x. x x)$$

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...

65

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.

66

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?

66

Towards recursion: Iterated application

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

$$Y_f = (\lambda x. f (x x)) (\lambda x. f (x x))$$

67

Towards recursion: Iterated application

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

$$Y_f = (\lambda x. f (x x)) (\lambda x. f (x x))$$

Now the “pattern of divergence” becomes more interesting:

$$\begin{aligned} Y_f &= \\ &= (\lambda x. f (x x)) (\lambda x. f (x x)) \\ &\rightarrow \\ &= f ((\lambda x. f (x x)) (\lambda x. f (x x))) \\ &\rightarrow \\ &= f (f ((\lambda x. f (x x)) (\lambda x. f (x x)))) \\ &\rightarrow \\ &= f (f (f ((\lambda x. f (x x)) (\lambda x. f (x x))))) \\ &\rightarrow \\ &\dots \end{aligned}$$

67

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

Is there any way we could “slow it down”?

68

Delaying divergence

$$\text{poisonpill} = \lambda y. \omega$$

Note that poisonpill is 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 to other functions, return it as a result from functions, etc.

$$\begin{aligned} &(\lambda p. \text{fst} (\text{pair } p \text{ fls}) \text{tru}) \text{poisonpill} \\ &\rightarrow \\ &= \text{fst} (\text{pair } \text{poisonpill} \text{ fls}) \text{tru} \\ &\rightarrow^* \\ &= \text{poisonpill } \text{tru} \\ &\rightarrow \\ &= \omega \\ &\rightarrow \\ &\dots \end{aligned}$$

69

A delayed variant of ω

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

$$\omega_{\text{gav}} = \lambda y. (\lambda x. (\lambda y. x x y)) (\lambda x. (\lambda y. x x y)) y$$

Note that ω_{gav} is a normal form. However, if we apply it to any argument v , it diverges:

$$\begin{aligned} \omega_{\text{gav}} v &= \\ &= (\lambda y. (\lambda x. (\lambda y. x x y)) (\lambda x. (\lambda y. x x y)) y) v \\ &\rightarrow \\ &= (\lambda x. (\lambda y. x x y)) (\lambda x. (\lambda y. x x y)) v \\ &\rightarrow \\ &= (\lambda y. (\lambda x. (\lambda y. x x y)) (\lambda x. (\lambda y. x x y)) y) v \\ &= \\ &= \omega_{\text{gav}} v \end{aligned}$$

70

Another delayed variant

Suppose f is a function. Define

$$z_f = \lambda y. (\lambda x. f (\lambda y. x x y)) (\lambda x. f (\lambda y. x x y)) y$$

This term combines the “added f ” from Y_f with the “delayed divergence” of ω_{gav} .

71

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

$$\begin{aligned} z_f v &= \\ & (\lambda y. (\lambda x. f (\lambda y. x x y)) (\lambda x. f (\lambda y. x x y)) y) v \\ & \rightarrow \\ & (\lambda x. f (\lambda y. x x y)) (\lambda x. f (\lambda y. x x y)) v \\ & \rightarrow \\ & f (\lambda y. (\lambda x. f (\lambda y. x x y)) (\lambda x. f (\lambda y. x x y)) y) v \\ & = \\ & f z_f v \end{aligned}$$

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.

72

Recursion

Let

$$f = \lambda \text{fct.} \lambda n. \begin{cases} \text{if } n=0 \text{ then } 1 \\ \text{else } n * (\text{fct } (\text{pred } n)) \end{cases}$$

f looks just like the ordinary factorial function, except that, in place of a recursive call in the last line, 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.).

73

We can use z_f to “tie the knot” in the definition of f and obtain a real recursive factorial function:

$$\begin{aligned} z_f 3 & \rightarrow^* \\ & f z_f 3 \\ & = \\ & (\lambda \text{fct. } \lambda n. \dots) z_f 3 \\ & \rightarrow \rightarrow \\ & \text{if } 3=0 \text{ then } 1 \text{ else } 3 * (z_f (\text{pred } 3)) \\ & \rightarrow^* \\ & 3 * (z_f (\text{pred } 3)) \\ & \rightarrow \\ & 3 * (z_f 2) \\ & \rightarrow^* \\ & 3 * (f z_f 2) \\ & \dots \end{aligned}$$

74

A Generic z

If we define

$$z = \lambda f. z_f$$

i.e.,

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

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

$$z f \longrightarrow z_f$$

75

For example:

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

76

Technical Note

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

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

$$\text{fix} = \lambda f. (\lambda x. f (\lambda y. x x y)) (\lambda x. f (\lambda y. x x y))$$

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.

77