Theory of Types and Programming Languages Fall 2022

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: $\mathcal{T}$ is the *smallest* set *closed* under the 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 rule can be thought of as a "generating function"

given some elements from $\mathcal{T}$,

it generates some other element of $\mathcal{T}$

Saying $\mathcal{T}$ is **closed under** these rules means that $\mathcal{T}$ cannot be made any bigger using these generating functions.

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

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

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

Let's write these generating functions explicitly.

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

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

We can define a generating function for the whole set of inference rules (by combining generating functions of 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 restate the previous definition of the set of terms $\mathcal{T}$ as:

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:

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

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

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

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

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

Our two definitions 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.

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 $\mathcal{O}_P$) is itself an F-closed set.
- Since $T \subseteq \mathcal{O}_P$, 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.*

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: Operational Semantics for Booleans

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

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

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

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

Digression

Suppose we wanted to change our evaluation strategy so that the `then` and `else` branches of an `if` get evaluated (in that order) before the guard. How would we need to change the rules?

Digression

Suppose we wanted to change our evaluation strategy so that the `then` and `else` branches of an `if` get evaluated (in that order) before the guard. How would we need to change the rules?

Suppose, moreover, that if the evaluation of the `then` and `else` branches leads to the same value, we want to immediately produce that value ("short-circuiting" the evaluation of the guard). How would we need to change the rules?

Digression

Suppose we wanted to change our evaluation strategy so that the `then` and `else` branches of an `if` get evaluated (in that order) before the guard. How would we need to change the rules?

Suppose, moreover, that if the evaluation of the `then` and `else` branches leads to the same value, we want to immediately produce that value ("short-circuiting" the evaluation of the guard). How would we need to change the rules?

Of the rules we just invented, which are computation rules and which are congruence rules?

Recall: Evaluation, more explicitly

$\longrightarrow$ is the smallest two-place relation closed under the rules:

$$((\texttt{if true then } \texttt{t}_2 \texttt{ else } \texttt{t}_3), \texttt{t}_2) \in \longrightarrow$$

$$((\texttt{if false then } \texttt{t}_2 \texttt{ else } \texttt{t}_3), \texttt{t}_3) \in \longrightarrow$$

$$\frac{(\texttt{t}_1, \texttt{t}_1') \in \longrightarrow}{((\texttt{if } \texttt{t}_1 \texttt{ then } \texttt{t}_2 \texttt{ else } \texttt{t}_3), (\texttt{if } \texttt{t}_1' \texttt{ then } \texttt{t}_2 \texttt{ else } \texttt{t}_3)) \in \longrightarrow}$$

Recall: Evaluation, more explicitly

$\longrightarrow$ is the smallest two-place relation closed under the rules:

$$((\texttt{if true then } \texttt{t}_2 \texttt{ else } \texttt{t}_3), \texttt{t}_2) \quad \in \quad \longrightarrow$$

$$((\texttt{if false then } \texttt{t}_2 \texttt{ else } \texttt{t}_3), \texttt{t}_3) \quad \in \quad \longrightarrow$$

$$\frac{(\texttt{t}_1, \texttt{t}_1') \quad \in \quad \longrightarrow}{((\texttt{if } \texttt{t}_1 \texttt{ then } \texttt{t}_2 \texttt{ else } \texttt{t}_3), (\texttt{if } \texttt{t}_1' \texttt{ then } \texttt{t}_2 \texttt{ else } \texttt{t}_3)) \quad \in \quad \longrightarrow}$$

Exercise: write the generating function corresponding to these rules

Recall: Numbers

Boolean and numeric values:

`v ::=`		*values*
	`true`	*constant true*
	`false`	*constant false*
	`nv`	*numeric value*
`nv ::=`		*numeric values*
	`0`	*zero value*
	`succ nv`	*successor value*

Evaluation rules for numbers

$\boxed{\texttt{t} \longrightarrow \texttt{t}'}$

$$\frac{\texttt{t}_1 \longrightarrow \texttt{t}_1'}{\texttt{succ t}_1 \longrightarrow \texttt{succ t}_1'} \qquad \text{(E-Succ)}$$

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

$$\texttt{pred (succ nv}_1\texttt{)} \longrightarrow \texttt{nv}_1 \qquad \text{(E-PredSucc)}$$

$$\frac{\texttt{t}_1 \longrightarrow \texttt{t}_1'}{\texttt{pred t}_1 \longrightarrow \texttt{pred t}_1'} \qquad \text{(E-Pred)}$$

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

$$\texttt{iszero (succ nv}_1\texttt{)} \longrightarrow \texttt{false} \qquad \text{(E-IszeroSucc)}$$

$$\frac{\texttt{t}_1 \longrightarrow \texttt{t}_1'}{\texttt{iszero t}_1 \longrightarrow \texttt{iszero t}_1'} \qquad \text{(E-IsZero)}$$

Recall: Derivations

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

$$\dfrac{\dfrac{\dfrac{}{\texttt{pred 0} \longrightarrow \texttt{0}}\ \textsc{E-PredZero}}{\texttt{succ (pred 0)} \longrightarrow \texttt{succ 0}}\ \textsc{E-Succ}}{\texttt{pred (succ (pred 0))} \longrightarrow \texttt{pred (succ 0)}}\ \textsc{E-Pred}$$

Recall: Derivations

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

$$\dfrac{\dfrac{\dfrac{}{\texttt{pred 0} \longrightarrow \texttt{0}}\ \textsc{E-PredZero}}{\texttt{succ (pred 0)} \longrightarrow \texttt{succ 0}}\ \textsc{E-Succ}}{\texttt{pred (succ (pred 0))} \longrightarrow \texttt{pred (succ 0)}}\ \textsc{E-Pred}$$

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.

Recall: Induction on Derivations

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

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

Recall: Induction on Derivations

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

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

Example

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

Consider one by one each possible final rule used in $\mathcal{D}$: E-IfTrue, E-IfFalse, E-If, *etc.*

Recall: 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* $t \longrightarrow t'$*.*

I.e., a state where the abstract machine is halted. It can be regarded as a "result" of evaluation.

Recall: Values are normal forms

All values are normal forms in our language of booleans and numbers.

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

Recall: Values are normal forms, but we have stuck terms

All values are normal forms in our language of booleans and numbers.

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

No: some terms are *stuck*.

Recall: Values are normal forms, but we have stuck terms

All values are normal forms in our language of booleans and numbers.

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.

Stuck terms model run-time errors.

Recall: Values are normal forms, but we have stuck terms

All values are normal forms in our language of booleans and numbers.

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.

Stuck terms model run-time errors.

What are some examples?

Recall: 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:

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

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

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

Recall: Termination of evaluation

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

Proof sketch: By an argument on the strictly reducing size of terms at each evaluation step.

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 OCaml, Haskell, Scheme, Lisp, ...)

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))`."

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?

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))`.

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 $succ\ 0$."

So `plus3 (succ 0)` is just a convenient shorthand for:

"the function that, given x,
yields $succ\ (succ\ (succ\ x))$, applied to $succ\ 0$."

Reduction.

```
plus3 (succ 0)
=
(λx. succ (succ (succ x))) (succ 0)
```

So `plus3 (succ 0)` is just a convenient shorthand for:

"the function that, given x,
yields $\mathit{succ}\ (\mathit{succ}\ (\mathit{succ}\ x))$, applied to $\mathit{succ}\ 0$."

Reduction.

```
plus3 (succ 0)
=
(λx. succ (succ (succ x))) (succ 0)
=
succ (succ (succ (succ 0)))
```

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`:

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

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)`.

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

$\mathtt{t}$ ::=		*terms*
	$\mathtt{x}$	*variable*
	$\lambda\mathtt{x.t}$	*abstraction*
	$\mathtt{t\ t}$	*application*

*Term*inology:

- terms in the pure λ-calculus are often called λ-*terms*
- terms of the form $\lambda\mathtt{x.\ t}$ 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 λ`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:

λ`x.` λ`y. x y z`

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
λx. (λy. z y) y
```

Values

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

Operational Semantics

Computation rule:

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

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:

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

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:

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

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

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:

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

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:

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

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

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

Alpha conversion

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

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

Equivalence rules:

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

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

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
```

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

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

Functions on Booleans

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

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

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

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

$$
\begin{array}{lll}
 & \texttt{fst (pair v w)} & \\
= & \texttt{fst (}\underline{\texttt{(}\lambda\texttt{f. }\lambda\texttt{s. }\lambda\texttt{b. b f s) v}}\texttt{ w)} & \text{by definition} \\
\longrightarrow & \texttt{fst (}\underline{\texttt{(}\lambda\texttt{s. }\lambda\texttt{b. b v s) w}}\texttt{)} & \text{reducing} \\
\longrightarrow & \texttt{fst (}\lambda\texttt{b. b v w)} & \text{reducing} \\
= & \underline{\texttt{(}\lambda\texttt{p. p tru) (}\lambda\texttt{b. b v w)}} & \text{by definition} \\
\longrightarrow & \underline{\texttt{(}\lambda\texttt{b. b v w) tru}} & \text{reducing} \\
\longrightarrow & \texttt{tru v w} & \text{reducing} \\
\longrightarrow^{*} & \texttt{v} & \text{as before.}
\end{array}
$$

Church numerals

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

$$c_0 = \lambda s.\ \lambda z.\ z$$
$$c_1 = \lambda s.\ \lambda z.\ s\ z$$
$$c_2 = \lambda s.\ \lambda z.\ s\ (s\ z)$$
$$c_3 = \lambda s.\ \lambda z.\ s\ (s\ (s\ z))$$

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) c_0

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:

$\texttt{times = } \lambda\texttt{m. }\lambda\texttt{n. m (plus n) c}_0$

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)
```