Formal Methods

Lecture 5

(B. Pierce's slides for the book "Types and Programming Languages")

Review (and more details)

Operational Semantics

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.

Operational semantics for Booleans

Syntax of 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*

Evaluation Relation on Booleans

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

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

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

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

Evaluation, more explicitly

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

$$((\mathtt{if\ true\ then\ } t_2 \mathtt{\ else\ } t_3), t_2) \in \longrightarrow$$

$$((\mathtt{if\ false\ then\ } t_2 \mathtt{\ else\ } t_3), t_3) \in \longrightarrow$$

$$\frac{(t_1, t_1') \in \longrightarrow}{((\mathtt{if\ } t_1 \mathtt{\ then\ } t_2 \mathtt{\ else\ } t_3), (\mathtt{if\ } t_1' \mathtt{\ then\ } t_2 \mathtt{\ else\ } t_3)) \in \longrightarrow}$$

Reasoning about Evaluation

Derivations

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

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 D witnessing the pair (t, t') in the evaluation relation. Then either

1. the final rule used in 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 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 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 D witnesses $(t_1, t_1') \in \longrightarrow$.

Induction on Derivations

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

Given an arbitrary derivation D with conclusion $t \longrightarrow t'$, 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.

Induction on Derivations — Example

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

1. Suppose the final rule used in 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 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 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 D_1. By the induction hypothesis, $size(\mathtt{t_1}) > size(\mathtt{t_1'})$. But then, by the definition of *size*, we have $size(\mathtt{t}) > size(\mathtt{t}')$.

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.

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 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 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.

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 $\text{if } t_1 \text{ then } t_2 \text{ else } t_3$ (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 $\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \longrightarrow \text{if } t_1 \text{ then } t_2 \text{ else } t_3$

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

$\boxed{t \longrightarrow t'}$

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

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

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

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

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

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

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

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

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, $\xrightarrow{*}$, is the reflexive, transitive closure of single-step evaluation.

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

$$\frac{t \longrightarrow t'}{t \xrightarrow{*} t'}$$

$$t \xrightarrow{*} t$$

$$\frac{t \xrightarrow{*} t' \qquad t' \xrightarrow{*} t''}{t \xrightarrow{*} t''}$$

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:

  $$\textit{if } t \longrightarrow t', \textit{ then } \textit{size}(t) > \textit{size}(t')$$

- 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

  $$\textit{size}(t_0) > \textit{size}(t_1) > \textit{size}(t_2) > \textit{size}(t_3) > \ldots$$

- 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 base 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))`."

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))`.

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

This function exists independent of the name `plus3`.

λ`x. t` is written "`fun x → t`" in OCaml.

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

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

```
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

Syntax

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

*Term*inology:

- 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*

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., `λx. λy. x y` *means* `λx. (λy. x y)`, *not* `λx. (λy. 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*.

```
λx. λy. x y z
λx. (λy. 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

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

Programming in the Lambda-Calculus

Multiple arguments

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

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

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

In general, `λx. λy. 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, `λx. λy. t` is a two-argument function.

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

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

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
```

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`.

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`.

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

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

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:

```
scc = λn. λs. λz. s (n s z)
```

Addition:

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

Multiplication:

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

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)
```