CIS 500

Software Foundations

Fall 2002

23 September

Administrivia

- Midterm 2 has been moved to Nov 13 (instead of Nov 18).
- The final will almost certainly remain on the announced date (Dec 20); the CIS 501 final may be moved
- Max Kanovich is out of town this week
  - Participants in recitation section D (Monday 3:00-4:30, Towne 307) should attend Section C (Monday 3:00-4:30, Moore 222) instead
  - Participants in section G (Tuesday 6:00-7:30) may attend any other section
  - Max's office hours are also cancelled; check the course web site for office hours of other course personnel

Operational Semantics

(review and a bit more)

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 rules

The single-step 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

Combining the previous ideas, we can write proofs about evaluation "by induction on derivation trees." E.g....

Theorem: If $\mathtt{t} \longrightarrow \mathtt{t}'$ — i.e., if $(\mathtt{t}, \mathtt{t}') \in \longrightarrow$ — then $\mathsf{size}(\mathtt{t}) > \mathsf{size}(\mathtt{t}')$.

Proof: By induction on a derivation of $\mathtt{t} \longrightarrow \mathtt{t}'$.

For each step of the induction, we assume the desired result for all smaller derivations and proceed by a case analysis (using the previous lemma) of the final evaluation rule used in constructing the derivation tree.

Aside

Q: Why are we bothering to prove all these completely obvious facts about terms and evaluation?

Aside

Q: Why are we bothering to prove all these completely obvious facts about terms and evaluation?

A: Suppose you told one of these facts to someone and they replied, "I don't believe it!" How would you convince them, aside from just saying, "Well, look at it again... isn't it obvious?"

I.e., we're trying to draw out why it is obvious.

Aside

Q: Why are we bothering to prove all these completely obvious facts about terms and evaluation?

A: Suppose you told one of these facts to someone and they replied, "I don't believe it!" How would you convince them, aside from just saying, "Well, look at it again... isn't it obvious?"

I.e., we're trying to draw out why it is obvious.

A': Facts almost this obvious have a habit of being false.

Doing the proofs is a methodology for debugging definitions.

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

$\boxed{t \longrightarrow t'}$

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

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

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

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

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

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

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

Aside

Q: Could we give the previous definition without bothering to introduce a separate category of numeric values?

Normal Forms

A *normal form* is a term t that does not evaluate to anything — i.e., such that there are no pairs of the form (t, t') in $\longrightarrow$ for any t'.

N.b.: When t is a normal form, we also say that t is "in normal form."

Normal Forms

A *normal form* is a term t that does not evaluate to anything — i.e., such that there are no pairs of the form (t, t') in $\longrightarrow$ for any t'.

Theorem: Every value v is a normal form.

Proof: ?

N.b.: When t is a normal form, we also say that t is "in normal form."

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

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.

Stuck terms model run-time errors.

Multi-step evaluation.

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

That is, it is the smallest relation such that

1. if $\mathtt{t} \longrightarrow \mathtt{t}'$ then $\mathtt{t} \longrightarrow^* \mathtt{t}'$,
2. $\mathtt{t} \longrightarrow^* \mathtt{t}$ for all $\mathtt{t}$, and
3. if $\mathtt{t} \longrightarrow^* \mathtt{t}'$ and $\mathtt{t}' \longrightarrow^* \mathtt{t}''$, then $\mathtt{t} \longrightarrow^* \mathtt{t}''$.

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 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 $\mathsf{size}(t) > \mathsf{size}(t')$

- 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

  $\mathsf{size}(t_0), \mathsf{size}(t_1), \mathsf{size}(t_2), \mathsf{size}(t_3), \mathsf{size}(t_4), \ldots$

  is an infinite, strictly decreasing sequence of natural numbers.

- 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.

More examples (time permitting)

- Nondeterministic choice
- Simple parallel composition
- A one-element memory

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
  - Variable binding and lexical scope
- 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` 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

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

That is, "`plus3 x` yields `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` yields `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

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

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

Q: What is `plus3` itself?

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

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

This function exists independent of the name `plus3`.

Intuitions

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

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

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

Q: What is `plus3` itself?

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

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

This function exists independent of the name `plus3`.

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

$$\texttt{plus3 (succ 0)} \;=\; (\lambda\texttt{x. succ (succ (succ x))) (succ 0)}$$

What's new?

We have introduced two new primitive syntactic forms:

- abstraction of a term `t` on some subterm `x`:

  λ`x. t`

  "The function that, when applied to a value `v`, yields `t` with `v` in place of `x`."

- application of a function to an argument:

  $\texttt{t}_1\ \texttt{t}_2$

  "the function $\texttt{t}_1$ applied to the argument $\texttt{t}_2$"

What's new?

We have introduced two new primitive syntactic forms:

- abstraction of a term `t` on some subterm `x`:

  λ`x. t`

  "The function that, when applied to a value `v`, yields `t` with `v` in place of `x`."

- application of a function to an argument:

  $\texttt{t}_1\ \texttt{t}_2$

  "the function $\texttt{t}_1$ applied to the argument $\texttt{t}_2$"

Note that abstractions are anonymous. For convenience in examples, we will sometimes write things like

> Let `plus3` be λ`x. succ (succ (succ x))` and consider the term `plus3 (succ 0)`

But the naming here is a metalanguage operation — the names are not part of the object language under discussion.

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

Terminology:

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

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`

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:

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

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

Operational Semantics

Computation rule:

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

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

Congruence rules:

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

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