CIS 500
Software Foundations
Fall 2006

October 2

Homework

Results of my email survey:

▶ There was one badly misdesigned (PhD) problem and a couple of others that were less well thought through than they could have been. These generated the great majority of specific complaints.
▶ Besides these, most people felt the homeworks were somewhat—but not outrageously—too long.
▶ People seemed more or less happy with the pace of the course... but no one wanted it faster! :-)
▶ “PhD questions” are an issue for mixed groups

Conclusion:

▶ Basically hold course
▶ Make homeworks a little shorter and tighter
▶ Change grading scheme for “PhD problems”
  ▶ Non-PhD students in “PhD groups” will be graded the same as those in non-PhD groups
▶ Slow down a little more on harder bits of material during lectures
  ▶ I need your help for this!

Midterm

▶ Wednesday, October 11th
▶ Topics:
  ▶ Basic OCaml
  ▶ TAPL Chapters 3–9
  ▶ Inductive definitions and proofs
  ▶ Operational semantics
  ▶ Untyped lambda-calculus
  ▶ Simple types

More About Bound Variables
Substitution

Our definition of evaluation is based on the “substitution” of values for free variables within terms.

\[(\lambda \cdot t_{12}) \cdot v_2 \mapsto [x \mapsto v_2] \cdot t_{12} \quad \text{(E-AppAbs)}\]

But what is substitution, exactly? How do we define it?

Substitution, take two

\[
\begin{align*}
[x \mapsto a]x &= s & \text{if } x = y \\
[x \mapsto a]y &= y \\
[x \mapsto a] (\lambda y \cdot t_1) &= \lambda y \cdot ([x \mapsto a]t_1) \\
[x \mapsto a] (t_1 \cdot t_2) &= ([x \mapsto a]t_1) ([x \mapsto a]t_2)
\end{align*}
\]

What is wrong with this definition?

Substitution

For example, what does

\[(\lambda x \cdot x \cdot (\lambda y \cdot x \cdot y)) \cdot (\lambda x \cdot x \cdot x)\]

reduce to?

Note that this example is not a “complete program” — the whole term is not closed. We are mostly interested in the reduction behavior of closed terms, but reduction of open terms is also important in some contexts:

- program optimization
- alternative reduction strategies such as “full beta-reduction”

Formalizing Substitution

Consider the following definition of substitution:

\[
\begin{align*}
[x \mapsto a]x &= s & \text{if } x = y \\
[x \mapsto a]y &= y \\
[x \mapsto a] (\lambda y \cdot t_1) &= \lambda y \cdot ([x \mapsto a]t_1) \\
[x \mapsto a] (t_1 \cdot t_2) &= ([x \mapsto a]t_1) ([x \mapsto a]t_2)
\end{align*}
\]

What is wrong with this definition?

It substitutes for free and bound variables!

\[(x \mapsto y) (\lambda x \cdot x) = \lambda x \cdot y\]

This is not what we want!

Substitution, take two

\[
\begin{align*}
[x \mapsto a]x &= s & \text{if } x = y \\
[x \mapsto a]y &= y \\
[x \mapsto a] (\lambda y \cdot t_1) &= \lambda y \cdot ([x \mapsto a]t_1) \\
[x \mapsto a] (t_1 \cdot t_2) &= ([x \mapsto a]t_1) ([x \mapsto a]t_2)
\end{align*}
\]

What is wrong with this definition?

It suffers from variable capture!

\[(x \mapsto y) (\lambda y \cdot x) = \lambda x \cdot x\]

This is also not what we want.
Substitution, take three

\[ [x \mapsto s]x = s \]
\[ [x \mapsto s]y = y \quad \text{if } x \neq y \]
\[ [x \mapsto s](\lambda y.t_1) = \lambda y.([x \mapsto s]t_1) \quad \text{if } x \neq y, y \notin \text{FV}(s) \]
\[ [x \mapsto s](\lambda x.t_1) = \lambda x.([x \mapsto s]t_1) \]
\[ [x \mapsto s](t_1\ t_2) = ([x \mapsto s]t_1)([x \mapsto s]t_2) \]

What is wrong with this definition?

Now substitution is a partial function!

E.g., \([x \mapsto y](\lambda y.x)\) is undefined.

But we want an result for every substitution.

Substitution, for alpha-equivalence classes

Now consider substitution as an operation over alpha-equivalence classes of terms.

\[ [x \mapsto s]x = s \]
\[ [x \mapsto s]y = y \quad \text{if } x \neq y \]
\[ [x \mapsto s](\lambda y.t_1) = \lambda y.([x \mapsto s]t_1) \quad \text{if } x \neq y, y \notin \text{FV}(s) \]
\[ [x \mapsto s](\lambda x.t_1) = \lambda x.([x \mapsto s]t_1) \]
\[ [x \mapsto s](t_1\ t_2) = ([x \mapsto s]t_1)([x \mapsto s]t_2) \]

Examples:

- \([x \mapsto y](\lambda y.x)\) must give the same result as \([x \mapsto y](\lambda z.x)\).
  We know the latter is \(\lambda z.y\), so that is what we will use for the former.
- \([x \mapsto y](\lambda x.z)\) must give the same result as \([x \mapsto y](\lambda w.z)\).
  We know the latter is \(\lambda w.z\) so that is what we use for the former.

Bound variable names shouldn’t matter

It’s annoying that the “spelling” of bound variable names is causing trouble with our definition of substitution.

Intuition tells us that there shouldn’t be a difference between the functions \(\lambda x.x\) and \(\lambda y.y\). Both of these functions do exactly the same thing.

Because they differ only in the names of their bound variables, we’d like to think that these are the same function.

We call such terms alpha-equivalent.

Alpha-equivalence classes

In fact, we can create equivalence classes of terms that differ only in the names of bound variables.

When working with the lambda calculus, it is convenient to think about these equivalence classes, instead of raw terms.

For example, when we write \(\lambda x.x\) we mean not just this term, but the class of terms that includes \(\lambda y.y\) and \(\lambda z.z\).

We can now freely choose a different representative from a term’s alpha-equivalence class, whenever we need to, to avoid getting stuck.

Review

So what does \((\lambda x.x) (\lambda y.y)\) reduce to?
Types

Plan

▶ For today, we’ll go back to the simple language of arithmetic and boolean expressions and show how to equip it with a (very simple) type system
▶ The key property of this type system will be soundness: Well-typed programs do not get stuck
▶ Next time, we’ll develop a simple type system for the lambda-calculus
▶ We’ll spend a good part of the rest of the semester adding features to this type system

Outline

1. begin with a set of terms, a set of values, and an evaluation relation
2. define a set of types classifying values according to their "shapes"
3. define a typing relation \( t : T \) that classifies terms according to the shape of the values that result from evaluating them
4. check that the typing relation is sound in the sense that,
   4.1 if \( t : T \) and \( t \xrightarrow{\square} \star v \), then \( v : T \)
   4.2 if \( t : T \), then evaluation of \( t \) will not get stuck

Review: Arithmetic Expressions – Syntax

\[
\begin{align*}
t & ::= \text{terms} \\
& \quad \text{true} \\
& \quad \text{false} \\
& \quad \text{if } t \text{ then } t \text{ else } t \\
& \quad 0 \\
& \quad \text{succ } t \\
& \quad \text{pred } t \\
& \quad \text{iszero } t \\
\end{align*}
\]

\[
\begin{align*}
v & ::= \text{values} \\
& \quad \text{true} \\
& \quad \text{false} \\
& \quad \text{nv} \\
\end{align*}
\]

\[
\begin{align*}
v & ::= \text{numeric values} \\
& \quad 0 \\
& \quad \text{succ } nv \\
\end{align*}
\]

Evaluation Rules

\[
\begin{align*}
\text{if } true \text{ then } t_2 \text{ else } t_3 & \rightarrow t_2 \quad (E\text{-IfTrue}) \\
\text{if } false \text{ then } t_2 \text{ else } t_3 & \rightarrow t_3 \quad (E\text{-IFFalse}) \\
\text{if } t_1 \rightarrow t_1' \text{ then } t_2 \text{ else } t_3 \rightarrow \text{if } t_1' \text{ then } t_2 \text{ else } t_3 \quad (E\text{-If}) \\
\text{succ } t_1 & \rightarrow \text{succ } t_1' \quad (E\text{-Succ}) \\
\text{pred } 0 & \rightarrow 0 \quad (E\text{-PredZero}) \\
\text{pred } (\text{succ } nv_1) & \rightarrow nv_1 \quad (E\text{-PredSucc}) \\
\text{pred } t_1 & \rightarrow \text{pred } t_1' \quad (E\text{-Pred}) \\
\text{iszero } 0 & \rightarrow \text{true} \quad (E\text{-IsZeroZero}) \\
\text{iszero } (\text{succ } nv_1) & \rightarrow \text{false} \quad (E\text{-IsZeroSucc}) \\
\text{iszero } t_1 & \rightarrow \text{iszero } t_1' \quad (E\text{-IsZero})
\end{align*}
\]
Types

In this language, values have two possible "shapes": they are either booleans or numbers.

\[
T ::= \text{types} \\
\quad \text{Bool type of booleans} \\
\quad \text{Nat type of numbers}
\]

Typing Rules

\[
\begin{align*}
\text{true} : \text{Bool} & \quad (T-\text{TRUE}) \\
\text{false} : \text{Bool} & \quad (T-\text{FALSE}) \\
\text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T & \quad (T-\text{IF}) \\
\text{0} : \text{Nat} & \quad (T-\text{ZERO}) \\
\text{succ } t_1 : \text{Nat} & \quad (T-\text{SUCC}) \\
\text{pred } t_1 : \text{Nat} & \quad (T-\text{PRED}) \\
\text{iszero } t_1 : \text{Bool} & \quad (T-\text{ISZERO})
\end{align*}
\]

Typing Derivations

Every pair \((t, T)\) in the typing relation can be justified by a derivation tree built from instances of the inference rules. For example:

\[
\begin{array}{c}
\text{T-ZERO} \\
\text{0 : Nat} \\
\text{T-IsZERO} \\
\text{iszero } 0 : \text{Bool} \\
\text{T-ZERO} \\
\text{0 : Nat} \\
\text{T-PRED} \\
\text{pred } 0 : \text{Nat} \\
\text{T-IF} \\
\text{if } \text{iszero } 0 \text{ then } 0 \text{ else } \text{pred } 0 : \text{Nat}
\end{array}
\]

Proofs of properties about the typing relation often proceed by induction on typing derivations.

Imprecision of Typing

Like other static program analyses, type systems are generally imprecise: they do not predict exactly what kind of value will be returned by every program, but just a conservative (safe) approximation.

\[
\begin{align*}
\text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T & \quad (T-\text{IF})
\end{align*}
\]

Using this rule, we cannot assign a type to
\[
\text{if true then 0 else false}
\]
even though this term will certainly evaluate to a number.

Type Safety

The safety (or soundness) of this type system can be expressed by two properties:

1. **Progress**: A well-typed term is not stuck.
   \[ \text{If } t : T, \text{ then either } t \text{ is a value or } t \rightarrow t' \text{ for some } t'. \]

2. **Preservation**: Types are preserved by one-step evaluation.
   \[ \text{If } t : T \text{ and } t \rightarrow t', \text{ then } t' : T. \]
Inversion

Lemma:
1. If $\text{true} : R$, then $R = \text{Bool}$.
2. If $\text{false} : R$, then $R = \text{Bool}$.
3. If $\text{if t1 then t2 else t3} : R$, then $t1 : \text{Bool}$, $t2 : R$, and $t3 : R$.
4. If $0 : R$, then $R = \text{Nat}$.
5. If $\text{succ t1} : R$, then $R = \text{Nat}$ and $t1 : \text{Nat}$.
6. If $\text{pred t1} : R$, then $R = \text{Nat}$ and $t1 : \text{Nat}$.
7. If $\text{iszero t1} : R$, then $R = \text{Bool}$ and $t1 : \text{Nat}$.

Proof: ...

This leads directly to a recursive algorithm for calculating the type of a term...

Canonical Forms

Lemma:
1. If $v$ is a value of type $\text{Bool}$, then $v$ is either $\text{true}$ or $\text{false}$.
2. If $v$ is a value of type $\text{Nat}$, then $v$ is a numeric value.

Proof: ...

Typechecking Algorithm

\[
\text{typeof(t)} = \begin{cases} 
\text{if t = true then Bool} \\
\text{else if t = false then Bool} \\
\text{else if t = if t1 then t2 else t3 then Nat else "not typable"} \\
\text{else if t = iszero t1 then} \\
\quad \text{let T1 = typeof(t1) in} \\
\quad \text{if T1 = Nat then Bool else "not typable"} \\
\text{else if t = 0 then Nat} \\
\text{else if t = succ t1 then} \\
\quad \text{let T1 = typeof(t1) in} \\
\quad \text{if T1 = Nat then Nat else "not typable"} \\
\text{else if t = pred t1 then} \\
\quad \text{let T1 = typeof(t1) in} \\
\quad \text{if T1 = Nat then Nat else "not typable"} \\
\text{else if t = iszero t1 then} \\
\quad \text{let T1 = typeof(t1) in} \\
\quad \text{if T1 = Nat then Bool else "not typable"}
\end{cases}
\]
Theorem: Suppose $t$ is a well-typed term (that is, $t : T$ for some $T$). Then either $t$ is a value or else there is some $t'$ with $t \rightarrow t'$.

Proof: By induction on a derivation of $t : T$.

The T-True, T-False, and T-Zero cases are immediate, since $t$ in these cases is a value.

Case T-If: $t = \text{if } t_1 \text{ then } t_2 \text{ else } t_3$

$t_1 : \text{Bool}$  $t_2 : T$  $t_3 : T$
Preservation

*Theorem:* If $t : T$ and $t \rightarrow t'$, then $t' : T$.

Proof: ...

Recap: Type Systems

- Very successful example of a *lightweight formal method*
- Big topic in PL research
- Enabling technology for all sorts of other things, e.g. language-based security
- The skeleton around which modern programming languages are designed