SNU 4541.574
Programming Language Theory

Ack: BCP's slides

More About Bound Variables

Substitution

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

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

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

Substitution

For example, what does

$$\texttt{(}\lambda\texttt{x. x (}\lambda\texttt{y. x y)) (}\lambda\texttt{x. x y x)}$$

reduce to?

Formalizing Substitution

Consider the following definition of substitution:

$$
\begin{array}{ll}
[\mathtt{x} \mapsto \mathtt{s}]\mathtt{x} = \mathtt{s} & \\
[\mathtt{x} \mapsto \mathtt{s}]\mathtt{y} = \mathtt{y} & \text{if } \mathtt{x} \neq \mathtt{y} \\
[\mathtt{x} \mapsto \mathtt{s}](\lambda\mathtt{y.t}_1) = \lambda\mathtt{y.}\ ([\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_1) & \\
[\mathtt{x} \mapsto \mathtt{s}](\mathtt{t}_1\ \mathtt{t}_2) = ([\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_1)([\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_2) &
\end{array}
$$

What is wrong with this definition?

Formalizing Substitution

Consider the following definition of substitution:

$$
\begin{aligned}
&[\mathtt{x} \mapsto \mathtt{s}]\mathtt{x} = \mathtt{s} && \\
&[\mathtt{x} \mapsto \mathtt{s}]\mathtt{y} = \mathtt{y} && \text{if } \mathtt{x} \neq \mathtt{y} \\
&[\mathtt{x} \mapsto \mathtt{s}](\lambda\mathtt{y}.\mathtt{t}_1) = \lambda\mathtt{y}.\ ([\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_1) && \\
&[\mathtt{x} \mapsto \mathtt{s}](\mathtt{t}_1\ \mathtt{t}_2) = ([\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_1)([\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_2) &&
\end{aligned}
$$

What is wrong with this definition?

It substitutes for free and *bound* variables!

$$[\mathtt{x} \mapsto \mathtt{y}](\lambda\mathtt{x}.\ \mathtt{x}) = \lambda\mathtt{x}.\mathtt{y}$$

This is not what we want!

Substitution, take two

$$
\begin{array}{ll}
[\mathtt{x} \mapsto \mathtt{s}]\mathtt{x} = \mathtt{s} & \\
[\mathtt{x} \mapsto \mathtt{s}]\mathtt{y} = \mathtt{y} & \text{if } \mathtt{x} \neq \mathtt{y} \\
[\mathtt{x} \mapsto \mathtt{s}](\lambda\mathtt{y}.\mathtt{t_1}) = \lambda\mathtt{y}.\ ([\mathtt{x} \mapsto \mathtt{s}]\mathtt{t_1}) & \text{if } \mathtt{x} \neq \mathtt{y} \\
[\mathtt{x} \mapsto \mathtt{s}](\lambda\mathtt{x}.\mathtt{t_1}) = \lambda\mathtt{x}.\ \mathtt{t_1} & \\
[\mathtt{x} \mapsto \mathtt{s}](\mathtt{t_1}\ \mathtt{t_2}) = ([\mathtt{x} \mapsto \mathtt{s}]\mathtt{t_1})([\mathtt{x} \mapsto \mathtt{s}]\mathtt{t_2}) &
\end{array}
$$

What is wrong with this definition?

Substitution, take two

$$
\begin{array}{ll}
[\mathtt{x} \mapsto \mathtt{s}]\mathtt{x} = \mathtt{s} & \\
[\mathtt{x} \mapsto \mathtt{s}]\mathtt{y} = \mathtt{y} & \text{if } \mathtt{x} \neq \mathtt{y} \\
[\mathtt{x} \mapsto \mathtt{s}](\lambda \mathtt{y}.\mathtt{t}_1) = \lambda \mathtt{y}.\ ([\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_1) & \text{if } \mathtt{x} \neq \mathtt{y} \\
[\mathtt{x} \mapsto \mathtt{s}](\lambda \mathtt{x}.\mathtt{t}_1) = \lambda \mathtt{x}.\ \mathtt{t}_1 & \\
[\mathtt{x} \mapsto \mathtt{s}](\mathtt{t}_1\ \mathtt{t}_2) = ([\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_1)([\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_2) &
\end{array}
$$

What is wrong with this definition?

It suffers from *variable capture*!

$$[\mathtt{x} \mapsto \mathtt{y}](\lambda \mathtt{y}.\mathtt{x}) = \lambda \mathtt{x}.\ \mathtt{x}$$

This is also not what we want.

Substitution, take three

$$
\begin{array}{ll}
[\mathtt{x} \mapsto \mathtt{s}]\mathtt{x} = \mathtt{s} & \\
[\mathtt{x} \mapsto \mathtt{s}]\mathtt{y} = \mathtt{y} & \text{if } \mathtt{x} \neq \mathtt{y} \\
[\mathtt{x} \mapsto \mathtt{s}](\lambda\mathtt{y}.\mathtt{t}_1) = \lambda\mathtt{y}.\ ([\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_1) & \text{if } \mathtt{x} \neq \mathtt{y},\ \mathtt{y} \notin FV(\mathtt{s}) \\
[\mathtt{x} \mapsto \mathtt{s}](\lambda\mathtt{x}.\mathtt{t}_1) = \lambda\mathtt{x}.\ \mathtt{t}_1 & \\
[\mathtt{x} \mapsto \mathtt{s}](\mathtt{t}_1\ \mathtt{t}_2) = ([\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_1)([\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_2) &
\end{array}
$$

Bound variable names shouldn't matter

It's annoying that 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 $\lambda\texttt{x.x}$ and $\lambda\texttt{y.y}$. 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 $\lambda\mathtt{x.x}$ we mean not just this term, but the class of terms that includes $\lambda\mathtt{y.y}$ and $\lambda\mathtt{z.z}$.

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

Substitution, for alpha-equivalence classes

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

$$
\begin{array}{ll}
[\mathtt{x} \mapsto \mathtt{s}]\mathtt{x} = \mathtt{s} & \\
[\mathtt{x} \mapsto \mathtt{s}]\mathtt{y} = \mathtt{y} & \text{if } \mathtt{x} \neq \mathtt{y} \\
[\mathtt{x} \mapsto \mathtt{s}](\lambda\mathtt{y}.\mathtt{t}_1) = \lambda\mathtt{y}.\ ([\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_1) & \text{if } \mathtt{x} \neq \mathtt{y},\ \mathtt{y} \notin \mathit{FV}(\mathtt{s}) \\
[\mathtt{x} \mapsto \mathtt{s}](\lambda\mathtt{x}.\mathtt{t}_1) = \lambda\mathtt{x}.\ \mathtt{t}_1 & \\
[\mathtt{x} \mapsto \mathtt{s}](\mathtt{t}_1\ \mathtt{t}_2) = ([\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_1)([\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_2) &
\end{array}
$$

Examples:

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

Review

So what does

$$(\lambda\texttt{x. x } (\lambda\texttt{y. x y})) \; (\lambda\texttt{x. x y x})$$

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 $\mathtt{t} \longrightarrow^{*} \mathtt{v}$, then `v : T`
   4.2 if `t : T`, then evaluation of `t` will not get stuck

Review: Arithmetic Expressions – Syntax

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

Evaluation Rules

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

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

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

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{succ}\ \mathtt{t}_1 \longrightarrow \mathtt{succ}\ \mathtt{t}_1'} \tag{E-Succ}$$

$$\mathtt{pred}\ \mathtt{0} \longrightarrow \mathtt{0} \tag{E-PredZero}$$

$$\mathtt{pred}\ (\mathtt{succ}\ \mathtt{nv}_1) \longrightarrow \mathtt{nv}_1 \tag{E-PredSucc}$$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{pred}\ \mathtt{t}_1 \longrightarrow \mathtt{pred}\ \mathtt{t}_1'} \tag{E-Pred}$$

$$\mathtt{iszero}\ \mathtt{0} \longrightarrow \mathtt{true} \tag{E-IszeroZero}$$

$$\mathtt{iszero}\ (\mathtt{succ}\ \mathtt{nv}_1) \longrightarrow \mathtt{false} \tag{E-IszeroSucc}$$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{iszero}\ \mathtt{t}_1 \longrightarrow \mathtt{iszero}\ \mathtt{t}_1'} \tag{E-IsZero}$$

Types

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

`T`	::=	*types*
	`Bool`	*type of booleans*
	`Nat`	*type of numbers*

Typing Rules

$$\texttt{true} : \texttt{Bool} \qquad \text{(T-True)}$$

$$\texttt{false} : \texttt{Bool} \qquad \text{(T-False)}$$

$$\frac{\texttt{t}_1 : \texttt{Bool} \qquad \texttt{t}_2 : \texttt{T} \qquad \texttt{t}_3 : \texttt{T}}{\texttt{if } \texttt{t}_1 \texttt{ then } \texttt{t}_2 \texttt{ else } \texttt{t}_3 : \texttt{T}} \qquad \text{(T-If)}$$

$$\texttt{0} : \texttt{Nat} \qquad \text{(T-Zero)}$$

$$\frac{\texttt{t}_1 : \texttt{Nat}}{\texttt{succ } \texttt{t}_1 : \texttt{Nat}} \qquad \text{(T-Succ)}$$

$$\frac{\texttt{t}_1 : \texttt{Nat}}{\texttt{pred } \texttt{t}_1 : \texttt{Nat}} \qquad \text{(T-Pred)}$$

$$\frac{\texttt{t}_1 : \texttt{Nat}}{\texttt{iszero } \texttt{t}_1 : \texttt{Bool}} \qquad \text{(T-IsZero)}$$

Typing Derivations

Every pair $(\mathtt{t}, \mathtt{T})$ in the typing relation can be justified by a *derivation tree* built from instances of the inference rules.

$$\dfrac{\dfrac{\dfrac{}{\mathtt{0} : \mathtt{Nat}}\ \textsc{T-Zero}}{\mathtt{iszero\ 0} : \mathtt{Bool}}\ \textsc{T-IsZero} \qquad \dfrac{}{\mathtt{0} : \mathtt{Nat}}\ \textsc{T-Zero} \qquad \dfrac{\dfrac{}{\mathtt{0} : \mathtt{Nat}}\ \textsc{T-Zero}}{\mathtt{pred\ 0} : \mathtt{Nat}}\ \textsc{T-Pred}}{\mathtt{if\ iszero\ 0\ then\ 0\ else\ pred\ 0} : \mathtt{Nat}}\ \textsc{T-If}$$

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.

$$\frac{\mathtt{t}_1 : \mathtt{Bool} \qquad \mathtt{t}_2 : \mathtt{T} \qquad \mathtt{t}_3 : \mathtt{T}}{\mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 : \mathtt{T}} \qquad \text{(T-IF)}$$

Using this rule, we cannot assign a type to

$$\mathtt{if\ true\ then\ 0\ else\ false}$$

even though this term will certainly evaluate to a number.

Properties of the Typing Relation

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

   If $t : T$, then either t is a value or else $t \longrightarrow t'$ for some t'.

2. *Preservation:* Types are preserved by one-step evaluation

   If $t : T$ and $t \longrightarrow t'$, then $t' : T$.

Inversion

Lemma:

1. If $\mathtt{true} : \mathtt{R}$, then $\mathtt{R} = \mathtt{Bool}$.
2. If $\mathtt{false} : \mathtt{R}$, then $\mathtt{R} = \mathtt{Bool}$.
3. If $\mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 : \mathtt{R}$, then $\mathtt{t}_1 : \mathtt{Bool}$, $\mathtt{t}_2 : \mathtt{R}$, and $\mathtt{t}_3 : \mathtt{R}$.
4. If $\mathtt{0} : \mathtt{R}$, then $\mathtt{R} = \mathtt{Nat}$.
5. If $\mathtt{succ}\ \mathtt{t}_1 : \mathtt{R}$, then $\mathtt{R} = \mathtt{Nat}$ and $\mathtt{t}_1 : \mathtt{Nat}$.
6. If $\mathtt{pred}\ \mathtt{t}_1 : \mathtt{R}$, then $\mathtt{R} = \mathtt{Nat}$ and $\mathtt{t}_1 : \mathtt{Nat}$.
7. If $\mathtt{iszero}\ \mathtt{t}_1 : \mathtt{R}$, then $\mathtt{R} = \mathtt{Bool}$ and $\mathtt{t}_1 : \mathtt{Nat}$.

Inversion

Lemma:

1. If $\texttt{true} : \texttt{R}$, then $\texttt{R} = \texttt{Bool}$.
2. If $\texttt{false} : \texttt{R}$, then $\texttt{R} = \texttt{Bool}$.
3. If $\texttt{if}\ \texttt{t}_1\ \texttt{then}\ \texttt{t}_2\ \texttt{else}\ \texttt{t}_3 : \texttt{R}$, then $\texttt{t}_1 : \texttt{Bool}$, $\texttt{t}_2 : \texttt{R}$, and $\texttt{t}_3 : \texttt{R}$.
4. If $\texttt{0} : \texttt{R}$, then $\texttt{R} = \texttt{Nat}$.
5. If $\texttt{succ}\ \texttt{t}_1 : \texttt{R}$, then $\texttt{R} = \texttt{Nat}$ and $\texttt{t}_1 : \texttt{Nat}$.
6. If $\texttt{pred}\ \texttt{t}_1 : \texttt{R}$, then $\texttt{R} = \texttt{Nat}$ and $\texttt{t}_1 : \texttt{Nat}$.
7. If $\texttt{iszero}\ \texttt{t}_1 : \texttt{R}$, then $\texttt{R} = \texttt{Bool}$ and $\texttt{t}_1 : \texttt{Nat}$.

Proof: ...

Inversion

Lemma:

1. If $\texttt{true} : \texttt{R}$, then $\texttt{R} = \texttt{Bool}$.
2. If $\texttt{false} : \texttt{R}$, then $\texttt{R} = \texttt{Bool}$.
3. If $\texttt{if } \texttt{t}_1 \texttt{ then } \texttt{t}_2 \texttt{ else } \texttt{t}_3 : \texttt{R}$, then $\texttt{t}_1 : \texttt{Bool}$, $\texttt{t}_2 : \texttt{R}$, and $\texttt{t}_3 : \texttt{R}$.
4. If $\texttt{0} : \texttt{R}$, then $\texttt{R} = \texttt{Nat}$.
5. If $\texttt{succ } \texttt{t}_1 : \texttt{R}$, then $\texttt{R} = \texttt{Nat}$ and $\texttt{t}_1 : \texttt{Nat}$.
6. If $\texttt{pred } \texttt{t}_1 : \texttt{R}$, then $\texttt{R} = \texttt{Nat}$ and $\texttt{t}_1 : \texttt{Nat}$.
7. If $\texttt{iszero } \texttt{t}_1 : \texttt{R}$, then $\texttt{R} = \texttt{Bool}$ and $\texttt{t}_1 : \texttt{Nat}$.

Proof: ...

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

Typechecking Algorithm

```
typeof(t) = if t = true then Bool
            else if t = false then Bool
            else if t = if t1 then t2 else t3 then
              let T1 = typeof(t1) in
              let T2 = typeof(t2) in
              let T3 = typeof(t3) in
              if T1 = Bool and T2=T3 then T2
              else "not typable"
            else if t = 0 then Nat
            else if t = succ t1 then
              let T1 = typeof(t1) in
              if T1 = Nat then Nat else "not typable"
            else if t = pred t1 then
              let T1 = typeof(t1) in
              if T1 = Nat then Nat else "not typable"
            else if t = iszero t1 then
              let T1 = typeof(t1) in
              if T1 = Nat then Bool else "not typable"
```