COMP 3351 Programming Languages

Christian Grothoff
christian@grothoff.org

http://grothoff.org/christian/

\footnote{Based on notes by Prof. Jens Palsberg, UCLA}
Today

• \(\lambda\)-calculus exercises
• Type soundness: definition
• Type system for the \(\lambda\)-calculus
• Type soundness: proof structure
• Type soundness proof for \(\lambda\)-calculus
• Featherweight Java
Calculate (execute)

\((\lambda a.\lambda b.\lambda c. b) \ 5 \ \lambda a.4\)
Calculate (execute)

\((\lambda a.\lambda b.\lambda c. a \ b \ c)\lambda x. x \ \ \lambda a.4 \ 5\)
Calculate (execute)

\[
((\lambda f. \lambda x. fx)(\lambda x. x \ x))(\lambda x. x)
\]
Calculate (execute)

\[(\lambda z. \lambda t. z \ t)\ 4 \ \lambda z. z\]
Type Soundness

• A program is a closed expression. \((a \ b)\) is not a program (because it contains free variables).

• A value is either a \(\lambda\)-abstraction \((\lambda x. e)\) or a constant \((c)\).

• A type system for a programming language is **sound** if well-typed programs cannot cause type errors.

• A “type error” generally corresponds to a program that has not been reduced to a value but still cannot continue to execute.

• The type system must define “well-typed”.
A Type Error!

\[(\lambda z. \lambda t. z \ t)4 \quad \lambda z. z \rightarrow^* 4 \quad \lambda z. z\]

The program is “stuck”.
Types for the $\lambda$-Calculus!

For the $\lambda$-calculus, we need two kinds of types: function types and an integer type.

Types are generated from the grammar:

$$t ::= t_1 \rightarrow t_2 \mid \text{Int}$$

Note that there are infinitely many types. Notice also that each type can be viewed as a tree. The size of the tree can be used to define a partial order over types.
Type Environments

A type environment $\Gamma$ is a partial function with finite domain which maps elements of $Var$ to types:

$$\Gamma = [x \mapsto \text{Int}, y \mapsto \text{Int} \to \text{Int}]$$
Examples: Types for Values

We write $\Gamma \vdash e : t$ to denote that expression $e$ has type $t$ in the type environment $\Gamma$:

\[
\emptyset \vdash 4 : \text{Int}
\]
\[
\emptyset \vdash \lambda x. (\text{succ } x) : \text{Int} \rightarrow \text{Int}
\]
Type Rules for the $\lambda$-calculus

\[ \Gamma \vdash x : t \quad \text{if} \quad \Gamma(x) = t \quad (1) \]
Type Rules for the $\lambda$-calculus

$$
\frac{
\Gamma[x:s] \vdash e : t
}{
\Gamma \vdash \lambda x. e : s \rightarrow t
} \quad (2)
$$
Type Rules for the $\lambda$-calculus

\[ \Gamma \vdash e_1 : s \rightarrow t \quad \Gamma \vdash e_2 : s \]
\[ \Gamma \vdash e_1 e_2 : t \]
Type Rules for the $\lambda$-calculus

$$\Gamma \vdash c : \text{Int}$$ (4)
Type Rules for the $\lambda$-calculus

\[
\Gamma \vdash e : \text{Int} \\
\Gamma \vdash \text{succ } e : \text{Int}
\]
Well-typed expressions

- An expression $e$ is well-typed if there exist $\Gamma$ and $t$ so that $\Gamma \vdash e : t$ is derivable.
Example: Type Derivation

\[
\begin{align*}
\emptyset[f : s \to t][x : s] & \vdash f : s \to t \\
\emptyset[f : s \to t][x : s] & \vdash x : s \\
\emptyset & \vdash \lambda x. fx : s \to t \\
\emptyset & \vdash \lambda f. \lambda x. fx : (s \to t) \to (s \to t)
\end{align*}
\]
Example: Failing Type Derivation

$$\emptyset \vdash \lambda x. e : \text{Int}$$

$$\emptyset \vdash \text{succ} (\lambda x. e) : \text{Int}$$
Type Soundness: Proof Structure

• Preservation
  – Substitution (with equal type) preserves type
  – Execution preserves type

• Progress
  – Certain types correspond to values (base case)
  – Closed expressions of other types can make progress
  – Progress does not change closedness

⇒ Well-typed programs cannot “go wrong”.
Substitution

If $\Gamma[x : s] \vdash e : t$ and $\Gamma \vdash M : s$ then $\Gamma \vdash e[x := M] : t$. 
Proof by Induction

- Each term $e$ in the $\lambda$-calculus can be associated with a (finite) “size” based on the syntax tree for the calculus.
- We will assume that the substitution lemma holds for a “smaller” term while we try to show that it holds for a “larger” term.
Larger?

- $\lambda x. e$ is larger than $e$
- $e_1 e_2$ is larger than $e_1$ and/or $e_2$ (individually)
- $\text{succ } e$ is larger than $e$
Substitution

To show:

If $\Gamma[x : s] \vdash e : t$ and $\Gamma \vdash M : s$ then $\Gamma \vdash e[x := M] : t$.

We have: Since $\Gamma[x : s] \vdash e : t$, one of our five type-rules must have been used in the last step of the type derivation.
Case 1: \( e \equiv y \)

- Case 1a: \( y \equiv x \). Then \( y[x := M] = M \). Since \( \Gamma[x : s] \vdash e : t \) we conclude \( s = t \). From \( \Gamma \vdash M : s \) and \( s = t \) we conclude \( \Gamma \vdash M : t \).

- Case 1b: \( y \not\equiv x \). Then \( y[x := M] = y \); from \( \Gamma[x : s] \vdash y : t \) we conclude \( \Gamma(y) = t \) and thus \( \Gamma \vdash y : t \).
Case 2: $e \equiv \lambda y.e_1$

• Case 2a: $y \equiv x$. Then $(\lambda y.e_1)[x := M] \equiv \lambda y.e_1$. Since $x$ does not occur free in $\lambda y.e_1$ we can use the derivation from $\Gamma[x : s] \vdash \lambda y.e_1 : t$ to produce a derivation of $\Gamma \vdash \lambda y.e_1 : t$.

• Case 2b: $y \not\equiv x$. Then $(\lambda y.e_1)[x := M] \equiv \lambda z.e_1[y := z][x := M]$ with $z$ fresh. (continued)
Case 2b: $e \equiv \lambda y. e_1$, $y \neq x$

The last step in the derivation of $\Gamma[x : s] \vdash e : t$ is of the form:

$$
\Gamma[x : s][y : t_2] \vdash e_1 : t_1
$$

$$
\frac{
\Gamma[x : s] \vdash \lambda y. e_1 : t_2 \to t_1
}{
\Gamma[x : s] \vdash e_1[\lambda y. e_1][y := z] : t_1.
}
$$

Hence $\Gamma[x : s][z : t_2] \vdash e_1[y := z] : t_1$. Note that $e_1$ and consequently $e_1[y := z]$ are “smaller” than $\lambda y. e_1$ and hence by induction hypothesis $\Gamma[z : t_2] \vdash e_1[y := z][x := M] : t_1$. With type rule (2) we can derive $\Gamma \vdash \lambda z. e_1[y := z][x := M] : t_2 \to t_1$. 


Case 3:  $e \equiv e_1 e_2$

The last step in the derivation of $\Gamma[x : s] \vdash e : t$ is of the form:

$$
\begin{align*}
\Gamma[x : s] &\vdash e_1 : t_2 \rightarrow t \\
\Gamma[x : s] &\vdash e_2 : t_2 \\
\hline
\Gamma[x : s] &\vdash e_1 \; e_2 : t
\end{align*}
$$

Using the induction hypothesis we get $\Gamma \vdash e_1[x := M] : t_2 \rightarrow t$ and $\Gamma \vdash e_2[x := M] : t_2$; with rule $\textbf{(3)} \; \Gamma \vdash e_1[x := M] \; e_2[x := M] : t$ follows.
Case 4: $e \equiv c$

Obviously $c[x := M] \equiv c$. The entire derivation of $\Gamma[x : s] \vdash e : t$ is of the form $\Gamma[x : s] \vdash c : \text{Int}$. From rule (4) we have $\Gamma \vdash c : \text{Int}$. 
Case 5: $e \equiv \text{succ } e_1$

Proof is similar to case 3.
Type Preservation

If $\Gamma \vdash e : t$ and $e \rightarrow_V e'$, then $\Gamma \vdash e' : t$. 
Proof by Induction

- We use induction over the derivation of $\Gamma \vdash e : t$.
- In the proof, we assume that the theorem holds for a derivation of depth $n - 1$ and show it for a derivation of depth $n$.
- The theorem is obvious for derivations of depth 0 since $e \rightarrow_v e'$ is impossible for those.
Case 1: $e \equiv x$

$e \rightarrow_V e'$ is not possible.
Case 2: \( e \equiv \lambda x. e_1 \)

\( e \rightarrow_V e' \) is not possible.
Case 3: $e \equiv e_1 e_2$

There are three subcases depending on which of the possible ways $e \rightarrow_V e'$ was used to make progress.

If either $e_1 e_2 \rightarrow_V e'_1 e_2$ or $e_1 e_2 \rightarrow_V e_1 e'_2$ were used, $\Gamma \vdash e' : t$ follows from the induction hypothesis and rule (3).
Case 3c: $e \equiv (\lambda x.e_1)v$

Suppose

$$(\lambda x.e_1)v \rightarrow_V e_1[x := v]$$

was used. Then the last part of the derivation of $\Gamma \vdash e : t$ is of the form:

$$
\frac{
\frac{
\Gamma[x:s] \vdash e_1 : t
}{
\Gamma \vdash \lambda x.e_1 : s \rightarrow t
}
\Gamma \vdash v : s
}{
\Gamma \vdash (\lambda x.e_1)v : t
}
$$

Using the substitution lemma, $\Gamma[x : s] \vdash e_1 : t$ and $\Gamma \vdash v : s$ we get $\Gamma \vdash e_1[x := v] : t$. 
Case 4: $e \equiv c$

$e \rightarrow_V e'$ is not possible.
Case 5: \( e \equiv \text{succ } e_1 \)

Again we look at two subcases depending on how \( e \rightarrow_V e' \) happened.

If \( e \equiv \text{succ } c_1 \) and \( e' \equiv c_2 \) (where \( \langle c_2 \rangle = \langle c_1 \rangle + 1 \)) then the type derivation of \( \Gamma \vdash e : t \) was of the form \( \Gamma \vdash \text{succ } c_1 : \text{Int} \) and from rule (4) we have \( \Gamma \vdash c_2 : \text{Int} \).
Case 5b: \( e \equiv \text{succ} \ e_1 \text{ and } e_1 \rightarrow_V e_2 \)

The last part of the derivation of \( \Gamma \vdash e : t \) is then of the form:

\[
\frac{\Gamma \vdash e_1 : \text{Int}}{\Gamma \vdash \text{succ} \ e_1 : \text{Int}}
\]

From the induction hypothesis we have \( \Gamma \vdash e_2 : \text{Int} \), so using rule (5) we derive \( \Gamma \vdash \text{succ} \ e_2 : \text{Int} \).
Typable Value

If $\Gamma \vdash v : Int$, then $v$ is of the form $c$.

If $\Gamma \vdash v : s \rightarrow t$ then $v$ is of the form $\lambda x.e$.

Proof: Obvious from type rules 2 and 4.
Progress

If $e$ is a closed expression, and $\Gamma \vdash e : t$ then either $e$ is a value, or there exists $e'$ such that $e \rightarrow_V e'$. 
Proof by Induction

- Since $\Gamma \vdash e : t$ there must exist a type derivation for the term $e$

- We will assume that the progress lemma holds for a type derivation of size $n - 1$ while we try to show that it holds for a type derivation of size $n$

- There are now five subcases depending on which of the type rules was the last one used in the derivation
Case 1: $e \equiv x$

The term is not closed.
Case 2: \( e \equiv \lambda x. e \)

The term is a value.
Case 3: $e \equiv e_1 e_2$

Since $e$ is closed, $e_1$ and $e_2$ must be closed. The last step in the derivation of $\Gamma \vdash e_1 e_2 : t$ must be of the form

$$\frac{\Gamma \vdash e_1 : s \rightarrow t \quad \Gamma \vdash e_2 : s}{\Gamma \vdash e_1 e_2 : t}$$

From the induction hypothesis we have that either $e_1$ is a value or there exists $e'_1$ such that $e_1 \rightarrow_V e'_1$ (in which case we can make progress to $e'_1 e_2$). Also, either $e_2$ is a value, or there exists $e'_2$ such that $e_2 \rightarrow_V e'_2$ (in which case we can make progress to $e_1 e'_2$).
Case 3c: \[ e \equiv (\lambda x. e_3) e_2 \]

If both \( e_1 \) and \( e_2 \) are values, then according to the typeable value theorem \( e_1 \) must be of the form \( \lambda x. e_3 \) and hence

\[ e_1 e_2 \rightarrow_V e_3[x := e_2] \]
Case 4: $e \equiv c$

The term is a value.
Case 5: \( e \equiv \text{succ} \; e_1 \)

Since \( e \) is closed, \( e_1 \) is also closed. The last step in the derivation of \( \Gamma \vdash e : t \) must be of the form

\[
\Gamma \vdash e_1 : \text{Int} \\
\frac{}{\Gamma \vdash \text{succ} \; e_1 : \text{Int}}
\]

From the induction hypothesis we have that either \( e_1 \) is a value or there exists \( e'_1 \) such that \( e_1 \rightarrow_V e'_1 \).
Case 5: \( e \equiv \text{succ } e_1 \) (continued)

If \( e_1 \) is a value, then from \( \Gamma \vdash e_1 : \text{Int} \) and the typeable value lemma we have that \( e_1 \) is of the form \( c_1 \) and hence \( \text{succ } c_1 \rightarrow c_2 \) (where \( \langle c_2 \rangle = \langle c_1 \rangle + 1 \)).

Otherwise, if there exists \( e'_1 \) such that \( e_1 \rightarrow V e'_1 \), then we can make progress using \( \text{succ } e_1 \rightarrow V \text{succ } e'_1 \).
Closedness Preservation

If \( e \) is closed, and \( e \rightarrow_{V} e' \), then \( e' \) is closed.

**Proof:** Obvious.
Conclusion

Well-typed programs cannot go wrong.

*Proof:* Suppose we have a well-typed program $e$ that is stuck at an expression $e'$ with $e \rightarrow^* e'$. We know that $e'$ is closed (closedness preservation) and well-typed (type preservation). But then there exists $e''$ so that $e' \rightarrow_V e''$ (progress), a contradiction ($e'$ can not be stuck).
Questions

?
Question!

Can Java programs go wrong?