This Lecture

- Recapitulation: our example language, stuck terms, type systems.
- Basic typing rules
- Safety = Progress + Preservation
- Extensions: typing let-expressions and functions

Much of this lecture follows parts of the first few chapters of B. C. Pierce 2002 *Types and Programming Languages* closely.

Recap: Example Language

Abstract syntax for the example language:

\[ t \rightarrow \text{terms:} \]
\[ \begin{array}{ll}
\text{true} & \text{constant true} \\
\text{false} & \text{constant false} \\
\text{if } t \text{ then } t \text{ else } t & \text{conditional} \\
0 & \text{constant zero} \\
\text{succ } t & \text{successor} \\
\text{pred } t & \text{predecessor} \\
\text{iszero } t & \text{zero test}
\end{array} \]

Recap: Example Language

Abstract syntax for the example language:

\[ v \rightarrow \text{values:} \]
\[ \begin{array}{ll}
\text{true} & \text{true value} \\
\text{false} & \text{false value} \\
\text{nv} & \text{numeric value} \\
0 & \text{zero value} \\
\text{succ } \text{nv} & \text{successor value}
\end{array} \]

Values are normal forms: they cannot be evaluated further.

Recap: Example Language

Abstract syntax for the example language:

\[ t \rightarrow \text{terms:} \]
\[ \begin{array}{ll}
\text{true} & \text{constant true} \\
\text{false} & \text{constant false} \\
\text{if } t \text{ then } t \text{ else } t & \text{conditional} \\
0 & \text{constant zero} \\
\text{succ } t & \text{successor} \\
\text{pred } t & \text{predecessor} \\
\text{iszero } t & \text{zero test}
\end{array} \]

Recap: Example Language

Abstract syntax for the example language:

\[ v \rightarrow \text{values:} \]
\[ \begin{array}{ll}
\text{true} & \text{true value} \\
\text{false} & \text{false value} \\
\text{nv} & \text{numeric value} \\
0 & \text{zero value} \\
\text{succ } \text{nv} & \text{successor value}
\end{array} \]

Values are normal forms: they cannot be evaluated further.

Recap: Example Language

Abstract syntax for the example language:

\[ t \rightarrow \text{terms:} \]
\[ \begin{array}{ll}
\text{true} & \text{constant true} \\
\text{false} & \text{constant false} \\
\text{if } t \text{ then } t \text{ else } t & \text{conditional} \\
0 & \text{constant zero} \\
\text{succ } t & \text{successor} \\
\text{pred } t & \text{predecessor} \\
\text{iszero } t & \text{zero test}
\end{array} \]

Recap: Example Language

Abstract syntax for the example language:

\[ v \rightarrow \text{values:} \]
\[ \begin{array}{ll}
\text{true} & \text{true value} \\
\text{false} & \text{false value} \\
\text{nv} & \text{numeric value} \\
0 & \text{zero value} \\
\text{succ } \text{nv} & \text{successor value}
\end{array} \]

Values are normal forms: they cannot be evaluated further.

Recap: Example Language

Abstract syntax for the example language:

\[ t \rightarrow \text{terms:} \]
\[ \begin{array}{ll}
\text{true} & \text{constant true} \\
\text{false} & \text{constant false} \\
\text{if } t \text{ then } t \text{ else } t & \text{conditional} \\
0 & \text{constant zero} \\
\text{succ } t & \text{successor} \\
\text{pred } t & \text{predecessor} \\
\text{iszero } t & \text{zero test}
\end{array} \]

Recap: Example Language

Abstract syntax for the example language:

\[ v \rightarrow \text{values:} \]
\[ \begin{array}{ll}
\text{true} & \text{true value} \\
\text{false} & \text{false value} \\
\text{nv} & \text{numeric value} \\
0 & \text{zero value} \\
\text{succ } \text{nv} & \text{successor value}
\end{array} \]

Values are normal forms: they cannot be evaluated further.

Recap: Example Language

Abstract syntax for the example language:

\[ t \rightarrow \text{terms:} \]
\[ \begin{array}{ll}
\text{true} & \text{constant true} \\
\text{false} & \text{constant false} \\
\text{if } t \text{ then } t \text{ else } t & \text{conditional} \\
0 & \text{constant zero} \\
\text{succ } t & \text{successor} \\
\text{pred } t & \text{predecessor} \\
\text{iszero } t & \text{zero test}
\end{array} \]

Recap: Example Language

Abstract syntax for the example language:

\[ v \rightarrow \text{values:} \]
\[ \begin{array}{ll}
\text{true} & \text{true value} \\
\text{false} & \text{false value} \\
\text{nv} & \text{numeric value} \\
0 & \text{zero value} \\
\text{succ } \text{nv} & \text{successor value}
\end{array} \]

Values are normal forms: they cannot be evaluated further.

Recap: Example Language

Abstract syntax for the example language:

\[ t \rightarrow \text{terms:} \]
\[ \begin{array}{ll}
\text{true} & \text{constant true} \\
\text{false} & \text{constant false} \\
\text{if } t \text{ then } t \text{ else } t & \text{conditional} \\
0 & \text{constant zero} \\
\text{succ } t & \text{successor} \\
\text{pred } t & \text{predecessor} \\
\text{iszero } t & \text{zero test}
\end{array} \]

Recap: Example Language

Abstract syntax for the example language:

\[ v \rightarrow \text{values:} \]
\[ \begin{array}{ll}
\text{true} & \text{true value} \\
\text{false} & \text{false value} \\
\text{nv} & \text{numeric value} \\
0 & \text{zero value} \\
\text{succ } \text{nv} & \text{successor value}
\end{array} \]

Values are normal forms: they cannot be evaluated further.

Recap: Example Language

Abstract syntax for the example language:

\[ t \rightarrow \text{terms:} \]
\[ \begin{array}{ll}
\text{true} & \text{constant true} \\
\text{false} & \text{constant false} \\
\text{if } t \text{ then } t \text{ else } t & \text{conditional} \\
0 & \text{constant zero} \\
\text{succ } t & \text{successor} \\
\text{pred } t & \text{predecessor} \\
\text{iszero } t & \text{zero test}
\end{array} \]
Certain “obviously nonsensical” states are stuck; the term cannot be evaluated further, but it is not a value. For example: if 0 then pred 0 else 0

Definition: A term is stuck if it is a normal form but not a value.

Why stuck???
- The program is not well-defined according to the dynamic semantics of the language.
- We are attempting to break the abstractions of the language.

We let the notion of getting stuck model run-time errors.

We will define a typing relation between terms and types: $t : T$

Read:
- $t$ has type $T$

A term that has a type, i.e., is related to a type by such a typing relation, is said to be well-typed.

The typing relation will be defined by (schematic) typing rules, in the same way we defined the evaluation relation.

The most basic property of a type system: safety, or “well typed programs do not go wrong”, where “wrong” means entering a “stuck state”.

This breaks down into two parts:
- Progress: A well-typed term is not stuck.
- Preservation: If a well-typed term takes a step of evaluation, then the resulting term is also well-typed.

Together, these two properties say that a well-typed term can never reach a stuck state during evaluation.
Formally:

- **THEOREM [PROGRESS]**: Suppose that \( t \) is a well-typed term (i.e., \( t : T \)), then either \( t \) is a value or else there is some \( t' \) such that \( t \rightarrow t' \).
  
  **PROOF:** By induction on a derivation of \( t : T \).

- **THEOREM [PRESERVATION]**: If \( t : T \) and \( t \rightarrow t' \), then \( t' : T \).
  
  **PROOF:** By induction on a derivation of \( t : T \).

(Strong form: exact type \( T \) preserved.)

**Exceptions (1)**

What about terms like

- division by zero
- head of empty list

that usually are considered well-typed?

If the type system does not rule them out, we need to differentiate those from stuck terms, or we can no longer claim that "well-typed programs do not go wrong"!

**Exceptions (2)**

Idea: allow exceptions to be raised, and make it well-defined what happens when exceptions are raised.

For example:

- introduce a term `error`
- introduce evaluation rules like
  
  \[
  \text{head } [\ ] \rightarrow \text{error}
  \]
- typing rule: `error : T`

**Exceptions (3)**

- introduce propagation rules to ensure that the entire program evaluates to `error` once the exception has been raised (unless there is some exception handling mechanism), e.g.:
  \[
  \text{pred error} \rightarrow \text{error}
  \]
- change the Progress theorem slightly to allow for exceptions:

  **THEOREM [PROGRESS]:** Suppose that \( t \) is a well-typed term (i.e., \( t : T \)), then either \( t \) is a value or \( \text{error} \), or else there is some \( t' \) with \( t \rightarrow t' \).

**Extension: Let-bound Variables (1)**

Syntactic extension:

\[
\begin{align*}
\text{terms:} & \quad \text{...}
\mid \cdot \, x & \quad \text{variable}
\mid \text{let } x = t \in t & \quad \text{let-expression}
\end{align*}
\]

New evaluation rules:

\[
\begin{align*}
\text{let } x = v_1 \in t_2 & \rightarrow [x \mapsto v_1] t_2 \quad \text{(E-LETV)}
\mid t_1 & \rightarrow t'_1 \quad \text{(E-LET)}
\end{align*}
\]

**Extension: Let-bound Variables (2)**

We now need a **typing context** or **type environment** to keep track of types of variables (an abstract version of a "symbol table").

The typing relation thus becomes a **ternary relation**:

\[
\Gamma, x : T
\]

Read: term \( t \) has type \( T \) in type environment \( \Gamma \).

**Extension: Let-bound Variables (3)**

Environment-related notation:

- Extending an environment:
  \[
  \Gamma, x : T
  \]

  The new declaration is understood to replace any earlier declaration for a variable with the same name.

- Stating that the type of a variable is given by an environment:
  \[
  x : T \in \Gamma \quad \text{or} \quad \Gamma(x) = T
  \]

**Progress: A Proof Fragment (1)**

The relevant **typing** and **evaluation** rules for the case T-IF:

\[
\begin{align*}
\text{T-IF:} & \quad t_1 : \text{Bool} \quad t_2 : T \quad t_3 : T
\mid \text{if true then } t_2 \text{ else } t_3 \rightarrow t_2 \quad \text{(E-IFTRUE)}
\mid \text{if false then } t_2 \text{ else } t_3 \rightarrow t_3 \quad \text{(E-IFFALSE)}
\mid t_1 \rightarrow t'_1
\mid \text{if } t_1 \text{ then } t_2 \text{ else } t_3
\rightarrow \text{if } t'_1 \text{ then } t_2 \text{ else } t_3
\end{align*}
\]

**Progress: A Proof Fragment (2)**

A typical case when proving Progress by induction on a derivation of \( t : T \):

Case T-IF: \( t = \text{if } t_1 \text{ then } t_2 \text{ else } t_3 \)

\[
\begin{align*}
& t_1 : \text{Bool} \\
& t_2 : T \\
& t_3 : T
\end{align*}
\]

By ind. hyp, either \( t_1 \) is a value, or else there is some \( t'_1 \) such that \( t_1 \rightarrow t'_1 \). If \( t_1 \) is a value, then it must be either \( \text{true} \) or \( \text{false} \), in which case either \( \text{E-IFTRUE} \) or \( \text{E-IFFALSE} \) applies to \( t \). On the other hand, if \( t_1 \rightarrow t'_1 \), then by \( \text{E-IF} \), \( t \rightarrow \text{if } t'_1 \text{ then } t_2 \text{ else } t_3 \).

**Safety = Progress + Preservation (2)**

Formally:

A typical case when proving Progress by induction on a derivation of \( t : T \):

Case T-IF: \( t = \text{if } t_1 \text{ then } t_2 \text{ else } t_3 \)

\[
\begin{align*}
& t_1 : \text{Bool} \\
& t_2 : T \\
& t_3 : T
\end{align*}
\]

By ind. hyp, either \( t_1 \) is a value, or else there is some \( t'_1 \) such that \( t_1 \rightarrow t'_1 \). If \( t_1 \) is a value, then it must be either \( \text{true} \) or \( \text{false} \), in which case either \( \text{E-IFTRUE} \) or \( \text{E-IFFALSE} \) applies to \( t \). On the other hand, if \( t_1 \rightarrow t'_1 \), then by \( \text{E-IF} \), \( t \rightarrow \text{if } t'_1 \text{ then } t_2 \text{ else } t_3 \).
Extension: Let-bound Variables (4)

Updated typing rules:
\[ \Gamma \vdash \text{true} : \text{Bool} \]  \hspace{1cm} \text{(T-TRUE)}
\[ \Gamma \vdash \text{false} : \text{Bool} \]  \hspace{1cm} \text{(T-FALSE)}
\[ \Gamma \vdash t_1 : \text{Bool} \quad \Gamma \vdash t_2 : T \quad \Gamma \vdash t_3 : T \]  \hspace{1cm} \text{(T-IF)}

Extension: Let-bound Variables (5)

Updated typing rules:
\[ \Gamma \vdash 0 : \text{Nat} \]  \hspace{1cm} \text{(T-ZERO)}
\[ \Gamma \vdash t_1 : \text{Nat} \]  \hspace{1cm} \text{(T-SUCC)}
\[ \Gamma \vdash t_1 : \text{Nat} \]  \hspace{1cm} \text{(T-PRED)}
\[ \Gamma \vdash \text{iszero} t_1 : \text{Bool} \]  \hspace{1cm} \text{(T-ISZERO)}

Extension: Let-bound Variables (6)

New typing rules:
\[ x : T \in \Gamma \]  \hspace{1cm} \text{(T-VAR)}
\[ \Gamma \vdash x : T \]  \hspace{1cm} \text{(T-ABS)}
\[ \Gamma \vdash t_1 : T \quad \Gamma \vdash t_2 : T_2 \]  \hspace{1cm} \text{(T-APP)}

Extension: Functions (1)

Syntactic extension:
\[ t \to \ldots \quad \text{terms:} \]
\[ | \lambda x : T.t \quad \text{abstraction} \]
\[ | t \ q \quad \text{application} \]
\[ v \to \ldots \quad \text{values:} \]
\[ | \lambda x : T.t \quad \text{abstraction value} \]
\[ T \to \ldots \quad \text{types:} \]
\[ | T \to T \quad \text{type of functions} \]

Extension: Functions (2)

New evaluation rules:
\[ t_1 \to t'_1 \]  \hspace{1cm} \text{(E-APP1)}
\[ t_2 \to t'_2 \]  \hspace{1cm} \text{(E-APP2)}
\[ (\lambda x : T_1.t_2)v_2 \to [x \mapsto v_2]t_2 \]  \hspace{1cm} \text{(E-APPABS)}

Note:
- left to right evaluation order: first the function (E-APP1), then the argument (E-APP2)
- call-by-value: the argument fully evaluated before function "invoked" (E-APPABS).