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:

\[ t \rightarrow \text{terms:} \]

- `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

Recap: Values

The *values* of a language are a subset of the terms that are possible results of evaluation.

\[ v \rightarrow \text{values:} \]

- `true` true value
- `false` false value
- `nv` numeric value
- `0` zero value
- `succ nv` successor value

Values are *normal forms*: they cannot be evaluated further.
Recap: One Step Evaluation Rel. (1)

$t \rightarrow t'$ is an **evaluation relation** on terms. Read:

$t$ evaluates to $t'$ in one step.

The evaluation relation constitute an **operational semantics** for the example language.

\[
\begin{align*}
\text{if true then } t_2 \text{ else } t_3 & \rightarrow t_2 \quad \text{(E-IFTRUE)} \\
\text{if false then } t_2 \text{ else } t_3 & \rightarrow t_3 \quad \text{(E-IFFALSE)} \\
\end{align*}
\]

\[
\begin{align*}
& t_1 \rightarrow t'_1 \\
& \quad \text{if } t_1 \text{ then } t_2 \text{ else } t_3 \quad \text{(E-IF)} \\
& \quad \rightarrow \quad \text{if } t'_1 \text{ then } t_2 \text{ else } t_3
\end{align*}
\]

Recap: One Step Evaluation Rel. (2)

\[
\begin{align*}
& t_1 \rightarrow t'_1 \\
& \quad \text{succ } t_1 \rightarrow \text{succ } t'_1 \quad \text{(E-SUCC)} \\
& \quad \text{pred } 0 \rightarrow 0 \quad \text{(E-PREDZERO)} \\
& \quad \text{pred } (\text{succ } n v_1) \rightarrow n v_1 \quad \text{(E-PREDSUCC)} \\
& \quad t_1 \rightarrow t'_1 \\
& \quad \text{pred } t_1 \rightarrow \text{pred } t'_1 \quad \text{(E-PRED)}
\end{align*}
\]

Recap: One Step Evaluation Rel. (3)

\[
\begin{align*}
& \text{iszero } 0 \rightarrow \text{true} \quad \text{(E-ISZEROZERO)} \\
& \text{iszero } (\text{succ } n v_1) \rightarrow \text{false} \quad \text{(E-ISZEROSUCC)} \\
& \quad t_1 \rightarrow t'_1 \quad \text{(E-ISZERO)} \\
& \quad \text{iszero } t_1 \rightarrow \text{iszero } t'_1 \\
\end{align*}
\]

Recap: One Step Evaluation Rel. (4)

**Evaluation of:**

\[
\text{if (iszero (pred (succ 0))) then (pred 0) else (succ 0)}
\]

**Step 1:**

\[
\begin{align*}
& \text{pred (succ 0)} \rightarrow 0 \quad \text{E-PREDSUCC} \\
& \text{iszero (pred (succ 0))} \rightarrow \text{iszero 0} \quad \text{E-ISZERO} \\
& \text{if (iszero (pred (succ 0))) then (pred 0) else (succ 0)} \rightarrow \text{if (iszero 0) then (pred 0) else (succ 0)}
\end{align*}
\]
Recap: One Step Evaluation Rel. (5)

Step 2:

\[ \text{iszero 0} \rightarrow \text{true} \]  \hspace{1cm} \text{E-ISZEROZERO}  
\[ \text{if (iszero 0) then (pred 0) else (succ 0)} \rightarrow \text{if true then (pred 0) else (succ 0)} \]  \hspace{1cm} \text{E-IF}  

Step 3:

\[ \text{if true then (pred 0) else (succ 0)} \rightarrow \text{pred 0} \]  \hspace{1cm} \text{E-IFTRUE}  

Step 4:

\[ \text{pred 0} \rightarrow 0 \]  \hspace{1cm} \text{E-PREDZERO}  

Stuck Terms (1)

- Certain “obviously nonsensical” states are \textit{stuck}: the term cannot be evaluated further, but it is \textit{not a value}. For example:

\[ \text{if 0 then pred 0 else 0} \]

- Definition: A term is \textit{stuck} if it is a normal form but not a value.

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

Stuck Terms (2)

- We let the notion of getting stuck model \textit{run-time errors}.

Recap: Type Systems

Definitions (Pierce):

- A \textit{type system} is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute.

- A \textit{safe language} is one that protects its abstractions.

\textit{Our goal is thus a type system that rules out semantically ill-defined programs, i.e. that guarantees that a program never gets stuck!}
Why Should We Care About Safety?

- One reason: security.
- C/C++ is unsafe: buffer overruns possible.
- Buffer overruns allows input data to be executed as code.
- One of the most common security holes: Had a safe variant of C been used, one might speculate that billions of dollars would have been saved.

Today, we’re going to see how to go about proving that the design of a language is safe.

Types

At this point, there are only two types, booleans and the natural numbers:

\[ T \rightarrow \text{types:} \]

- \( \text{Bool} \) type of booleans
- \( \text{Nat} \) type of natural numbers

Typing Relation

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.

Typing Rules

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

What (if any) is the type of the following terms?

• if (iszero (succ 0)) then (succ 0) else 0
• if 0 then pred 0 else 0

Safety = Progress + Preservation (1)

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.

Safety = Progress + Preservation (2)

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.)

Progress: A Proof Fragment (1)

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

\[
\begin{align*}
  t_1 : \text{Bool} & \quad t_2 : T & \quad t_3 : T \\
  \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : & \quad T \\
  \text{if true then } t_2 \text{ else } t_3 & \rightarrow t_2 & \quad (E-IFTRUE) \\
  \text{if false then } t_2 \text{ else } t_3 & \rightarrow t_3 & \quad (E-IFFALSE)
\end{align*}
\]

\[
\begin{align*}
  t_1 & \rightarrow t'_1 \\
  \text{if } t'_1 \text{ then } t_2 \text{ else } t_3 & \rightarrow \text{if } t'_1 \text{ then } t_2 \text{ else } t_3 & \quad (E-IF)
\end{align*}
\]
Progress: A Proof Fragment (2)

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

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

\( t_1 : \text{Bool} \quad t_2 : T \quad t_3 : T \)

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}, \( t \rightarrow \text{if } t'_1 \text{ then } t_2 \text{ else } t_3 \).

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 \textit{exceptions} to be raised, and make it well-defined what happens when exceptions are raised.

For example:

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

Exceptions (3)

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

\[
\text{THEOREM [PROGRESS]: Suppose that } t \text{ is a well-typed term (i.e., } t : T), \text{ then either } t \text{ is a value or } error, \text{ or else there is some } t' \text{ with } t \rightarrow t'.
\]
Extension: Let-bound Variables (1)

Syntactic extension:

\[ t \rightarrow \ldots \]

| \[ x \] \hspace{1cm} \text{variable} |
| \[ \text{let } x = t \text{ in } t \] \hspace{1cm} \text{let-expression} |

New evaluation rules:

\[
\begin{align*}
\text{let } x = v_1 \text{ in } t_2 & \rightarrow [x \mapsto v_1]t_2 \quad \text{(E-LETV)} \\
\end{align*}
\]

\[
\begin{align*}
\text{let } x = t_1 \text{ in } t_2 & \rightarrow \text{let } x = t'_1 \text{ in } t_2 \\ 
\end{align*}
\]

Extension: Let-bound Variables (2)

We now need a \textit{typing context} or \textit{type environment} to keep track of types of variables (an abstract version of a “symbol table”).

The typing relation thus becomes a \textit{ternary relation}:

\[
\Gamma \vdash t : T
\]

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:

\[
x : T \in \Gamma \quad \text{or} \quad \Gamma(x) = T
\]

Extension: Let-bound Variables (4)

Updated typing rules:

\[
\begin{align*}
\Gamma \vdash \text{true} : \text{Bool} \quad \text{(T-TRUE)} \\
\Gamma \vdash \text{false} : \text{Bool} \quad \text{(T-FALSE)} \\
\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T \\
\end{align*}
\]

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

\[
\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T \quad \text{(T-IF)}
\]

\[
\begin{align*}
\end{align*}
\]
Extension: Let-bound Variables (5)

Updated typing rules:

\[ \Gamma \vdash 0 : \text{Nat} \quad \text{(T-ZERO)} \]
\[ \Gamma \vdash t_1 : \text{Nat} \quad \text{(T-SUCC)} \]
\[ \Gamma \vdash \text{pred} t_1 : \text{Nat} \quad \text{(T-PRED)} \]
\[ \Gamma \vdash \text{iszero} t_1 : \text{Bool} \quad \text{(T-ISZERO)} \]

Extension: Let-bound Variables (6)

New typing rules:

\[ x : T \in \Gamma \]
\[ \Gamma \vdash x : T \quad \text{(T-VAR)} \]
\[ \Gamma \vdash t_1 : T_1 \]
\[ \Gamma, x : T_1 \vdash t_2 : T_2 \]
\[ \Gamma \vdash \text{let} x = t_1 \text{ in } t_2 : T_2 \quad \text{(T-LET)} \]

Extension: Let-bound Variables (7)

Recursive bindings?

Typing is straightforward if the recursively-defined entity is \textit{explicitly} typed:

\[ \Gamma, x : T_1 \vdash t_1 : T_1 \]
\[ \Gamma, x : T_1 \vdash t_2 : T_2 \]
\[ \Gamma \vdash \text{let} x = t_1 \text{ in } t_2 : T_2 \quad \text{(T-LET)} \]

If not, the question is if \( T_1 \) is uniquely defined (and in a type checker how to compute this type):

\[ \Gamma, x : T_1 \vdash t_1 : T_1 \]
\[ \Gamma, x : T_1 \vdash t_2 : T_2 \]
\[ \Gamma \vdash \text{let} x = t_1 \text{ in } t_2 : T_2 \quad \text{(T-LET)} \]

\textbf{(Evaluation} is more involved: we leave that for now.\textbf{)}

Extension: Functions (1)

Syntactic extension:

\[ t \rightarrow \ldots \]
\[ \lambda x : T . t \quad \text{abstraction} \]
\[ t \hspace{1em} t \quad \text{application} \]

\[ v \rightarrow \ldots \]
\[ \lambda x : T . t \quad \text{abstraction value} \]

\[ T \rightarrow \ldots \]
\[ T \rightarrow T \quad \text{type of functions} \]
Extension: Functions (2)

New evaluation rules:

\[
\begin{align*}
    t_1 & \rightarrow t'_1 & \quad & \text{(E-APP1)} \\
    t_1 t_2 & \rightarrow t'_1 t_2 \\
    t_2 & \rightarrow t'_2 & \quad & \text{(E-APP2)} \\
    v_1 t_2 & \rightarrow v'_1 t'_2 \\
(\lambda x : T_{11} . t_{12}) v_2 & \rightarrow [x \mapsto v_2] t_{12} & \quad & \text{(E-APPABS)}
\end{align*}
\]

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).

Extension: Functions (3)

New typing rules:

\[
\begin{align*}
    \Gamma, x : T_1 & \vdash t_2 : T_2 & \quad & \text{(T-ABS)} \\
    \Gamma & \vdash \lambda x : T_1 . t_2 : T_1 \rightarrow T_2 \\
    \Gamma & \vdash t_1 : T_{11} \rightarrow T_{12} \\
    \Gamma & \vdash t_2 : T_{11} \\
    \Gamma & \vdash t_1 t_2 : T_{12} & \quad & \text{(T-APP)}
\end{align*}
\]