Type Systems for Programming Languages

- Type systems lead a double life
- They are an essential part of modern programming languages
- They are a fundamental concept from logic and proof theory
- As a result, they form the most important channel for connecting theoretical computer science to practical programming language design.
What are type systems used for?

• Error detection via *type checking*
• Support for structuring large (or even medium) sized programs
• Documentation
• Efficiency
• Safety
A Language of Booleans and Integers

Terms $e ::= \text{true} \mid \text{false} \mid n \mid e \leq e \mid e + e \mid e \land e \mid \neg e$

Some terms make sense:

- $3 + 4$
- $3 + 4 \leq 5$
- $(3 + 4 \leq 7) \land (7 \leq 3 + 4)$

Some terms don’t:

- $4 \land \text{true}$
- $3 \leq \text{true}$
- $\text{true} + 7$
Types \( \tau \) ::= \text{bool} \mid \mathbb{N}

Terms \( e \) ::= \text{true} \mid \text{false} \mid n \mid e \leq e \mid e + e \mid e \land e

- How to connect term (like 3 + 4) with a type (like \( \mathbb{N} \))?
- Via a \textit{typing judgement} \( e : \tau \)
- A two-place relation saying that “the term \( e \) has the type \( \tau \)”
- So \( _\vdash _ \) is an infix relation symbol
- How do we define this?
Typing Rules

- **Num**
  - \( n : \mathbb{N} \) \( n : \mathbb{N} \)

- **True**
  - \( \text{true} : \text{bool} \) \( \text{true} : \text{bool} \)

- **False**
  - \( \text{false} : \text{bool} \) \( \text{false} : \text{bool} \)

- **Plus**
  - \( e : \mathbb{N} \quad e' : \mathbb{N} \quad e + e' : \mathbb{N} \)

- **And**
  - \( e : \text{bool} \quad e' : \text{bool} \quad e \land e' : \text{bool} \)

- **LEQ**
  - \( e : \mathbb{N} \quad e' : \mathbb{N} \quad e \leq e' : \text{bool} \)

- **Above the line**: premises
- **Below the line**: conclusion
An Example Derivation Tree

\[
\begin{array}{c}
\begin{array}{c}
3 : \mathbb{N} \\
\end{array} \quad \text{Num} \quad \begin{array}{c}
4 : \mathbb{N} \\
\end{array} \quad \text{Plus} \quad \begin{array}{c}
5 : \mathbb{N} \\
\end{array} \quad \text{Num} \\
\hline
3 + 4 : \mathbb{N} \\
\hline
3 + 4 \leq 5 : \text{bool}
\end{array}
\]
Adding Variables

Types \( \tau ::= \text{bool} \mid \mathbb{N} \)

Terms \( e ::= \ldots \mid x \mid \text{letv } x = e \text{ in } e' \)

- Example: letv \( x = 5 \) in \((x + x) \leq 10\)
- But what type should \( x \) have: \( x : ? \)
- To handle this, the typing judgement must know what the variables are.
- So we change the typing judgement to be \( \Gamma \vdash e : \tau \), where \( \Gamma \) associates a list of variables to their types.
Contexts

\[ \Gamma ::= \cdot \mid \Gamma, x : \tau \]

- \( \Gamma \vdash n : \mathbb{N} \)
- \( \Gamma \vdash \text{true} : \text{bool} \)
- \( \Gamma \vdash \text{false} : \text{bool} \)

- \( \Gamma \vdash e : \mathbb{N} \quad \Gamma \vdash e' : \mathbb{N} \)
  \[ \Gamma \vdash e + e' : \mathbb{N} \]

- \( \Gamma \vdash e : \text{bool} \quad \Gamma \vdash e' : \text{bool} \)
  \[ \Gamma \vdash e \land e' : \text{bool} \]

- \( \Gamma \vdash e : \mathbb{N} \quad \Gamma \vdash e' : \mathbb{N} \)
  \[ \Gamma \vdash e \leq e' : \text{bool} \]

- \( x : \tau \in \Gamma \)
  \[ \Gamma \vdash x : \tau \]

- \( \Gamma \vdash e : \tau \quad \Gamma, x : \tau \vdash e' : \tau' \)
  \[ \Gamma \vdash \text{let} v \ x = e \ \text{in} \ e' : \tau' \]
• We have: a type system, associating elements from one grammar (the terms) with elements from another grammar (the types)
• We *claim* that this rules out “bad” terms
• But does it really?
• To prove, we must show *type safety*
We have introduced variables into our language, so we should introduce a notion of substitution as well:

\[
\begin{align*}
[e/x]\text{true} & \quad = \quad \text{true} \\
[e/x]\text{false} & \quad = \quad \text{false} \\
[e/x]n & \quad = \quad n \\
[e/x](e_1 + e_2) & \quad = \quad [e/x]e_1 + [e/x]e_2 \\
[e/x](e_1 \leq e_2) & \quad = \quad [e/x]e_1 \leq [e/x]e_2 \\
[e/x](e_1 \land e_2) & \quad = \quad [e/x]e_1 \land [e/x]e_2 \\
[e/x]z & \quad = \quad \begin{cases} e & \text{when } z = x \\ z & \text{when } z \neq x \end{cases} \\
[e/x](\text{letv } z = e_1 \text{ in } e_2) & \quad = \quad \text{letv } z = [e/x]e_1 \text{ in } [e/x]e_2 \quad (\ast)
\end{align*}
\]

\((\ast)\) \(\alpha\)-rename to ensure \(z\) does not occur in \(e!\)
1. (Weakening) If $\Gamma, \Gamma' \vdash e : \tau$ then $\Gamma, x : \tau'', \Gamma' \vdash e : \tau$.
   If a term typechecks in a context, then it will still typecheck in a bigger context.

2. (Exchange) If $\Gamma, x_1 : \tau_1, x_2 : \tau_2, \Gamma' \vdash e : \tau$ then $\Gamma, x_2 : \tau_2, x_1 : \tau_1, \Gamma' \vdash e : \tau$.
   If a term typechecks in a context, then it will still typecheck after reordering the variables in the context.

3. (Substitution) If $\Gamma \vdash e : \tau$ and $\Gamma, x : \tau \vdash e' : \tau'$ then $\Gamma \vdash [e/x]e' : \tau'$.
   Substituting a type-correct term for a variable will preserve type correctness.
A Proof of Weakening

- Proof goes by *structural induction*
- Suppose we have a derivation tree of $\Gamma, \Gamma' \vdash e : \tau$
- By case-analysing the root of the derivation tree, we construct a derivation tree of $\Gamma, x : \tau'', \Gamma' \vdash e : \tau$, assuming inductively that the theorem works on subtrees.
Proving Weakening, 1/4

\[
\begin{align*}
\frac{}{\Gamma, \Gamma' \vdash n : \mathbb{N}} & \quad \text{NUM} \\
\frac{}{\Gamma, x : \tau'', \Gamma' \vdash n : \mathbb{N}} & \quad \text{NUM}
\end{align*}
\]

By assumption

By rule \textsc{Num}

- Similarly for \textsc{True} and \textsc{False} rules
\[
\Gamma, \Gamma' \vdash e_1 : \mathbb{N} \quad \Gamma, \Gamma' \vdash e_2 : \mathbb{N} \\
\frac{}{\Gamma, \Gamma' \vdash e_1 + e_2 : \mathbb{N}} \text{ Plus}
\]

By assumption

Subderivation 1

Subderivation 2

Induction on subderivation 1

Induction on subderivation 2

By rule \text{Plus}

- Similarly for \text{LEQ} and \text{AND} rules
\[
\begin{align*}
\Gamma, \Gamma' \vdash e_1 : \tau_1 & \quad \Gamma, \Gamma', z : \tau_1 \vdash e_2 : \tau_2 \\
\Gamma, \Gamma' \vdash \text{letv } z = e_1 \text{ in } e_2 : \tau_2
\end{align*}
\]

**LET**  \hspace{1cm} By assumption

\[
\begin{align*}
\Gamma, \Gamma' \vdash e_1 : \tau_1 & \quad \text{Subderivation 1} \\
\Gamma, \Gamma', z : \tau_1 \vdash e_2 : \tau_2 & \quad \text{Subderivation 2} \\
\Gamma, x : \tau'', \Gamma' \vdash e_1 : \tau_1 & \quad \text{Induction on subderivation 1} \\
\end{align*}
\]

\[
\begin{align*}
\text{Extended context} & \quad \Gamma, \underbrace{x : \tau''} \quad \Gamma', z : \tau_1 \quad \vdash e_2 : \mathbb{N} \quad \text{Induction on subderivation 2} \\
\Gamma, x : \tau'', \Gamma' \vdash \text{letv } z = e_1 \text{ in } e_2 : \tau_2 & \quad \text{By rule LET}
\end{align*}
\]
\[
\frac{z : \tau \in \Gamma, \Gamma'}{
\Gamma, \Gamma' \vdash \text{let } z = e_1 \text{ in } e_2 : \tau_2}
\]
\(\text{VAR}\) 

By assumption

\[
\begin{align*}
z : \tau & \in \Gamma, \Gamma' & \text{By assumption} \\
z : \tau & \in \Gamma, x : \tau'', \Gamma' & \text{An element of a list is also in a bigger list} \\
\Gamma, x : \tau'', \Gamma' & \vdash z : \tau & \text{By rule } \text{VAR}
\end{align*}
\]
\[
\frac{
\Gamma, x_1 : \tau_1, x_2 : \tau_2, \Gamma' \vdash n : \mathbb{N}
}{\text{Num}}
\]

By assumption

\[
\frac{
\Gamma, x_2 : \tau_2, x_1 : \tau_1, \Gamma' \vdash n : \mathbb{N}
}{\text{Num}}
\]

By rule \texttt{Num}

- Similarly for \texttt{TRUE} and \texttt{FALSE} rules
\[ \Gamma, x_1 : \tau_1, x_2 : \tau_2, \Gamma' \vdash e_1 : \mathbb{N} \quad \Gamma, x_1 : \tau_1, x_2 : \tau_2, \Gamma' \vdash e_2 : \mathbb{N} \]
\[ \Gamma, \Gamma' \vdash e_1 + e_2 : \mathbb{N} \]

By assumption

\[ \Gamma, x_1 : \tau_1, x_2 : \tau_2, \Gamma' \vdash e_1 : \mathbb{N} \quad \text{Subderivation 1} \]
\[ \Gamma, x_1 : \tau_1, x_2 : \tau_2, \Gamma' \vdash e_2 : \mathbb{N} \quad \text{Subderivation 2} \]

\[ \Gamma, x_2 : \tau_2, x_1 : \tau_1, \Gamma' \vdash e_1 : \mathbb{N} \quad \text{Induction on subderivation 1} \]
\[ \Gamma, x_2 : \tau_2, x_1 : \tau_1, \Gamma' \vdash e_2 : \mathbb{N} \quad \text{Induction on subderivation 2} \]
\[ \Gamma, x_2 : \tau_2, x_1 : \tau_1, \Gamma' \vdash e_1 + e_2 : \mathbb{N} \quad \text{By rule PLUS} \]

- Similarly for \text{LEQ} and \text{AND} rules
\[
\begin{align*}
\Gamma, x_1 : \tau_1, x_2 : \tau_2, \Gamma' \vdash e_1 : \tau' \\
\Gamma, x_1 : \tau_1, x_2 : \tau_2, \Gamma', z : \tau' \vdash e_2 : \tau_2 \\
\Gamma, \Gamma' \vdash \text{letv } z = e_1 \text{ in } e_2 : \tau_2
\end{align*}
\]

By assumption

\[
\begin{align*}
\Gamma, x_1 : \tau_1, x_2 : \tau_2, \Gamma' \vdash e_1 : \tau' \\
\Gamma, x_1 : \tau_1, x_2 : \tau_2, \Gamma', z : \tau' \vdash e_2 : \tau_2 \\
\Gamma, x_2 : \tau_2, x_1 : \tau_1, \Gamma' \vdash e_1 : \tau_1
\end{align*}
\]

Subderivation 1

Subderivation 2

Extended context

\[
\begin{align*}
\Gamma, x_2 : \tau_2, x_1 : \tau_1, \Gamma', z : \tau_1 \vdash e_2 : \mathbb{N}
\end{align*}
\]

Induction on s.d. 1

Induction on s.d. 2

\[
\begin{align*}
\Gamma, x_2 : \tau_2, x_1 : \tau_1, \Gamma' \vdash \text{letv } z = e_1 \text{ in } e_2 : \tau_2
\end{align*}
\]

By rule \text{LET}
Proving Exchange, 4/4

\[
\begin{align*}
z : \tau & \in \Gamma, x_1 : \tau_1, x_2 : \tau_2, \Gamma' \\
\Gamma, \Gamma' & \vdash z : \tau \quad \text{VAR} \quad \text{By assumption}
\end{align*}
\]

\[
\begin{align*}
z : \tau & \in \Gamma, x_1 : \tau_1, x_2 : \tau_2, \Gamma' \quad \text{By assumption} \\
z : \tau & \in \Gamma, x_2 : \tau_2, x_1 : \tau_1, \Gamma' \quad \text{An element of a list is also in a permutation of the list} \\
\Gamma, x_2 : \tau_2, x_1 : \tau_1, \Gamma' & \vdash z : \tau \quad \text{By rule VAR}
\end{align*}
\]
A Proof of Substitution

• Proof also goes by *structural induction*
• Suppose we have derivation trees $\Gamma \vdash e : \tau$ and $\Gamma, x : \tau \vdash e' : \tau'$.
• By case-analysing the root of the derivation tree of $\Gamma, x : \tau \vdash e' : \tau'$, we construct a derivation tree of $\Gamma \vdash [e/x]e' : \tau'$, assuming inductively that substitution works on subtrees.
Substitution 1/4

\[
\begin{align*}
\Gamma, x : \tau & \vdash n : \mathbb{N} \quad \text{Num} \\
\Gamma & \vdash e : \tau \quad \text{By assumption} \\
\Gamma & \vdash e : \tau \quad \text{By assumption} \\
\Gamma & \vdash n : \mathbb{N} \quad \text{By rule Num} \\
\Gamma & \vdash [e/x]n : \mathbb{N} \quad \text{Def. of substitution}
\end{align*}
\]

- Similarly for \texttt{TRUE} and \texttt{FALSE} rules.
\[
\begin{align*}
\Gamma, x : \tau & \vdash e_1 : \mathbb{N} & \Gamma, x : \tau & \vdash e_2 : \mathbb{N} \\
\hline
\Gamma, x : \tau & \vdash e_1 + e_2 : \mathbb{N}
\end{align*}
\]

By assumption: (1)

By assumption: (2)

Subderivation of (1): (3)

Subderivation of (1): (4)

Induction on (2), (3): (5)

Induction on (2), (4): (6)

By rule \texttt{PLUS} on (5), (6)

Def. of substitution

• Similarly for \texttt{LEQ} and \texttt{AND} rules
Proving Substitution, 3/4

\[
\begin{align*}
\Gamma, x : \tau & \vdash e_1 : \tau' & \Gamma, x : \tau, z : \tau' & \vdash e_2 : \tau_2 \\
\frac{}{\Gamma, x : \tau \vdash \text{letv } z = e_1 \text{ in } e_2 : \tau_2} & \text{LET} & \text{By assumption: (1)}
\end{align*}
\]

\[
\begin{align*}
\Gamma & \vdash e : \tau & \text{By assumption: (2)} \\
\Gamma, x : \tau & \vdash e_1 : \tau' & \text{Subderivation of (1): (3)} \\
\Gamma, x : \tau, z : \tau' & \vdash e_2 : \tau_2 & \text{Subderivation of (1): (4)} \\
\Gamma & \vdash [e/x]e_1 : \tau' & \text{Induction on (2) and (3): (4)} \\
\Gamma, z : \tau' & \vdash e : \tau & \text{Weakening on (2): (5)} \\
\Gamma, z : \tau', x : \tau & \vdash e_2 : \tau_2 & \text{Exchange on (4): (6)} \\
\Gamma, z : \tau' & \vdash [e/x]e_2 : \tau_2 & \text{Induction on (5) and (6): (7)} \\
\Gamma & \vdash \text{letv } z = [e/x]e_1 \text{ in } [e/x]e_2 : \tau_2 & \text{By rule LET on (6), (7)} \\
\Gamma & \vdash [e/x](\text{letv } z = e_1 \text{ in } e_2) : \tau_2 & \text{By def. of substitution}
\end{align*}
\]
Proving Substitution, 4a/4

\[
\frac{z : \tau' \in \Gamma, x : \tau}{\Gamma, x : \tau \vdash z : \tau'} \quad \text{VAR}
\]

\[\Gamma \vdash e : \tau \quad \text{By assumption}\]

Case \(x = z:\)
\[\Gamma \vdash [e/x]x : \tau \quad \text{By def. of substitution}\]
\[
\frac{z : \tau' \in \Gamma, x : \tau}{\Gamma, x : \tau \vdash z : \tau'} \quad \text{VAR}
\]

By assumption

\[
\Gamma \vdash e : \tau \quad \text{By assumption}
\]

Case \(x \neq z\):

\[
z : \tau' \in \Gamma \quad \text{since } x \neq z \text{ and } z : \tau' \in \Gamma, x : \tau
\]

\[
\Gamma, z : \tau' \vdash z : \tau' \quad \text{By rule VAR}
\]

\[
\Gamma, z : \tau' \vdash [e/x]z : \tau' \quad \text{By def. of substitution}
\]
Operational Semantics

- We have a language and type system
- We have a proof of substitution
- How do we say what \textit{value} a program computes?
- With an \textit{operational semantics}
- Define a grammar of \textit{values}
- Define a two-place relation on terms $e \rightsquigarrow e'$
- Pronounced as “$e$ steps to $e'$”
An operational semantics

Values \( \nu \ ::= \ n \mid \text{true} \mid \text{false} \)

\[
\frac{e_1 \rightsquigarrow e_1'}{e_1 \land e_2 \rightsquigarrow e_1' \land e_2} \quad \text{AndCong} \\
\frac{\text{true} \land e \rightsquigarrow e}{\quad \text{AndTrue}}
\]

\[
\frac{\text{false} \land e \rightsquigarrow \text{false}}{\quad \text{AndFalse}}
\]

(similar rules for \( \leq \) and \(+\))

\[
\frac{e_1 \rightsquigarrow e_1'}{\text{let} \nu z = e_1 \text{ in } e_2 \rightsquigarrow \text{let} \nu z = e_1' \text{ in } e_2} \quad \text{LetCong}
\]

\[
\frac{\text{let} \nu z = v \text{ in } e_2 \rightsquigarrow [v/z]\text{e}_2}{\quad \text{LetStep}}
\]
Reduction Sequences

- A reduction sequence is a sequence of transitions $e_0 \leadsto e_1$, $e_1 \leadsto e_2$, ..., $e_{n-1} \leadsto e_n$.

- A term $e$ is stuck if it is not a value, and there is no $e'$ such that $e \leadsto e'$

<table>
<thead>
<tr>
<th>Successful sequence</th>
<th>Stuck sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(3 + 4) \leq (2 + 3)$</td>
<td>$(3 + 4) \land (2 + 3)$</td>
</tr>
<tr>
<td>$\leadsto 7 \leq (2 + 3)$</td>
<td>$\leadsto 7 \land (2 + 3)$</td>
</tr>
<tr>
<td>$\leadsto 7 \leq 5$</td>
<td>$\leadsto ???$</td>
</tr>
<tr>
<td>$\leadsto \text{false}$</td>
<td></td>
</tr>
</tbody>
</table>

Stuck terms are erroneous programs with no defined behaviour.
Type Safety

A program is safe if it never gets stuck.

1. (Progress) If $\cdot \vdash e : \tau$ then either $e$ is a value, or there exists $e'$ such that $e \leadsto e'$.

2. (Preservation) If $\cdot \vdash e : \tau$ and $e \leadsto e'$ then $\cdot \vdash e' : \tau$.

- Progress means that well-typed programs are not stuck: they can always take a step of progress (or are done).
- Preservation means that if a well-typed program takes a step, it will stay well-typed.
- So a well-typed term won’t reduce to a stuck term: the final term will be well-typed (due to preservation), and well-typed terms are never stuck (due to progress).
(Progress) If $\cdot \vdash e : \tau$ then either $e$ is a value, or there exists $e'$ such that $e \rightsquigarrow e'$.

- To show this, we do structural induction on the derivation of $\cdot \vdash e : \tau$.
- For each typing rule, we show that either $e$ is a value, or can step.
Progress: Values

\[
\begin{align*}
\text{\underline{\text{NUM}}} \\
\vdash n : \mathbb{N} & \quad \text{By assumption} \\
\end{align*}
\]

\[n \text{ is a value} \quad \text{Def. of value grammar}\]

Similarly for boolean literals...
Progress: Let-bindings

\[
\begin{array}{c}
\cdot \vdash e_1 : \tau \\
x : \tau \vdash e_2 : \tau'
\end{array}
\]
\[
\frac{}{\cdot \vdash \text{letv } x = e_1 \text{ in } e_2 : \tau'} \quad \text{LET}
\]

By assumption: (1)

Subderivation of (1): (2)

Subderivation of (1): (3)

Induction on (2)

Case \(e_1 \rightsquigarrow e_1'\) or \(e_1\) value

\[
\text{letv } x = e_1 \text{ in } e_2 \rightsquigarrow \text{letv } x = e_1' \text{ in } e_2
\]

By rule \text{LETCONG}

Case \(e_1\) value:

\[
\text{letv } x = e_1 \text{ in } e_2 \rightsquigarrow [e_1/x]e_2
\]

By rule \text{LETSTEP}
Type Preservation

(Preservation) If $\vdash e : \tau$ and $e \rightsquigarrow e'$ then $\vdash e' : \tau$.

1. We will use structural induction again, but on which derivation?

2. Two choices: (1) $\vdash e : \tau$ and (2) $e \rightsquigarrow e'$

3. The right choice is induction on $e \rightsquigarrow e'$

4. We will still need to deconstruct $\vdash e : \tau$ alongside it!
Type Preservation: Let Bindings 1

\[ e_1 \leadsto e_1' \]

\[ \text{let} v \ x = e_1 \ \text{in} \ e_2 \leadsto \text{let} v \ x = e_1' \ \text{in} \ e_2 \]

By assumption: (1)

\[ \cdot \vdash e_1 : \tau \quad x : \tau \vdash e_2 : \tau' \]

\[ \cdot \vdash \text{let} v \ x = e_1 \ \text{in} \ e_2 : \tau' \]

By assumption: (2)

\[ e_1 \leadsto e_1' \]

Subderivation of (1): (3)

\[ \cdot \vdash e_1 : \tau \]

Subderivation of (2): (4)

\[ x : \tau \vdash e_2 : \tau' \]

Subderivation of (2): (5)

\[ \cdot \vdash e_1' : \tau \]

Induction on (3), (4): (6)

\[ \cdot \vdash \text{let} v \ x = e_1' \ \text{in} \ e_2 : \tau' \]

Rule LET on (6), (4)
letv \( x = v_1 \) in \( e_2 \) \( \leadsto \) \( [v_1/x]e_2 \)

By assumption: (1)

\[ v_1 : \tau \quad x : \tau \vdash e_2 : \tau' \]

By assumption: (2)

Subderivation of (2): (3)

Subderivation of (2): (4)

Substitution on (3), (4)
Conclusion

Given a language of program terms and a language of types:

- A type system ascribes types to terms
- An operational semantics describes how terms evaluate
- A type safety proof connects the type system and the operational semantics
- Proofs are intricate, but not difficult
1. Give cases of the operational semantics for $\leq$ and $\mathbf{+}$.
2. Extend the progress proof to cover $e \land e'$.
3. Extend the preservation proof to cover $e \land e'$.

(This should mostly be review of IB *Semantics of Programming Languages.*)