Type Systems

Lecture 1

Neel Krishnaswami
University of Cambridge
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 \( e \ ::= \) true \mid false \mid n \mid e \leq e \mid e + e \mid e \land e \mid \neg e

Some terms make sense:

\begin{itemize}
  \item 3 + 4
  \item 3 + 4 \leq 5
  \item (3 + 4 \leq 7) \land (7 \leq 3 + 4)
\end{itemize}

Some terms don’t:

\begin{itemize}
  \item 4 \land true
  \item 3 \leq true
  \item true + 7
\end{itemize}
Types for Booleans and Integers

Types \( \tau ::= \) bool \mid \mathbb{N}  
Terms \( e ::= \) true \mid 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 typing judgement \( e : \tau \)
- A two-place relation saying that “the term \( e \) has the type \( \tau \)”
- So \( _ : _ \) is an infix relation symbol
- How do we define this?
Typing Rules

\[
\begin{align*}
&\text{NUM} & n : \mathbb{N} & \quad \text{TRUE} & \quad \text{FALSE} \\
& & & & \\
& e : \mathbb{N} & e' : \mathbb{N} & \quad e + e' : \mathbb{N} & \quad e : \text{bool} & e' : \text{bool} & \quad e \land e' : \text{bool} \\
& & & & & & \\
& e : \mathbb{N} & e' : \mathbb{N} & \quad e \leq e' : \text{bool}
\end{align*}
\]

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

\[
\begin{array}{c}
\text{3 : } \mathbb{N} \\
\text{4 : } \mathbb{N}
\end{array}
\quad
\begin{array}{c}
\text{PLUS} \\
\text{LEQ}
\end{array}

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

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

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

- Example: let \( x = 5 \) 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

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

\[ \begin{align*}
\Gamma &\vdash n : \mathbb{N} \\
\Gamma &\vdash \text{true} : \text{bool} \\
\Gamma &\vdash \text{false} : \text{bool} \\
\Gamma &\vdash e : \mathbb{N} \quad \Gamma &\vdash e' : \mathbb{N} \\
\Gamma &\vdash e + e' : \mathbb{N} \\
\Gamma &\vdash e : \text{bool} \quad \Gamma &\vdash e' : \text{bool} \\
\Gamma &\vdash e \land e' : \text{bool} \\
\Gamma &\vdash e \leq e' : \text{bool} \\
x : \tau \in \Gamma &\quad \Gamma &\vdash e : \tau \\
\Gamma &\vdash x : \tau \\
\Gamma, x : \tau &\vdash e' : \tau' \\
\Gamma &\vdash \text{let } x = e \text{ in } e' : \tau' 
\end{align*} \]
• 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*
Prelude: Substitution

We have introduced variables into our language, so we should introduce a notion of substitution as well:

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

\(\ast\) \(\alpha\)-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 $\Gamma \vdash [e/x]e' : \tau'$. Subsituting 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

\[ \Gamma, \Gamma' \vdash n : \mathbb{N} \quad \text{By assumption} \]

\[ \Gamma, x : \tau'', \Gamma' \vdash n : \mathbb{N} \quad \text{By rule Num} \]

- Similarly for TRUE and 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

\[
\Gamma, \Gamma' \vdash e_1 : \mathbb{N} \quad \Gamma, \Gamma' \vdash e_2 : \mathbb{N} \\
\Gamma, x : \tau'', \Gamma' \vdash e_1 : \mathbb{N} \\
\Gamma, x : \tau'', \Gamma' \vdash e_2 : \mathbb{N} \\
\Gamma, x : \tau'', \Gamma' \vdash e_1 + e_2 : \mathbb{N}
\]

Subderivation 1
Subderivation 2
Induction on subderivation 1
Induction on subderivation 2
By rule PLUS

• Similarly for LEQ and AND rules
Proving Weakening, 3/4

\[ \Gamma, \Gamma' \vdash e_1 : \tau_1 \quad \Gamma, \Gamma', z : \tau_1 \vdash e_2 : \tau_2 \quad \text{LET} \]
\[ \Gamma, \Gamma' \vdash \text{let } z = e_1 \text{ in } e_2 : \tau_2 \quad \text{By assumption} \]

\[ \begin{align*}
\Gamma, \Gamma' & \vdash e_1 : \tau_1 \\
\Gamma, \Gamma', z : \tau_1 & \vdash e_2 : \tau_2 \\
\Gamma, x : \tau'', \Gamma' & \vdash e_1 : \tau_1 \\
\Gamma, x : \tau'', \Gamma' & \vdash \text{let } z = e_1 \text{ in } e_2 : \tau_2
\end{align*} \]

Subderivation 1

Subderivation 2

Induction on subderivation 1

Extended context

\[ \begin{align*}
\Gamma, x : \tau'', \Gamma' \vdash e_2 : \tau_2 \\
\Gamma, x : \tau'', \Gamma' \vdash \text{let } z = e_1 \text{ in } e_2 : \tau_2
\end{align*} \]

Induction on subderivation 2

By rule LET
Proving Weakening, 4/4

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

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

\[
\Gamma, x_2 : \tau_2, x_1 : \tau_1, \Gamma' \vdash n : \mathbb{N} \quad \text{By rule \textsc{Num}}
\]

- Similarly for \textsc{True} and \textsc{False} rules
\[
\Gamma, x_1 : \tau_1, x_2 : \tau_2, \Gamma' \vdash e_1 : \mathbb{N} \quad \Gamma, x_1 : \tau_1, x_2 : \tau_2, \Gamma' \vdash e_2 : \mathbb{N}
\]
\[
\Gamma, x_1 : \tau_1, x_2 : \tau_2, \Gamma' \vdash e_1 + e_2 : \mathbb{N} \quad \text{PLUS}
\]

By assumption

\[
\Gamma, x_1 : \tau_1, x_2 : \tau_2, \Gamma' \vdash e_1 : \mathbb{N} \
\Gamma, x_1 : \tau_1, x_2 : \tau_2, \Gamma' \vdash e_2 : \mathbb{N}
\]

Subderivation 1

Subderivation 2

\[
\Gamma, x_2 : \tau_2, x_1 : \tau_1, , \Gamma' \vdash e_1 : \mathbb{N} \
\Gamma, x_2 : \tau_2, x_1 : \tau_1, , \Gamma' \vdash e_2 : \mathbb{N}
\]

Induction on subderivation 1

Induction on subderivation 2

\[
\Gamma, x_2 : \tau_2, x_1 : \tau_1, , \Gamma' \vdash e_1 + e_2 : \mathbb{N}
\]

By rule PLUS

• Similarly for LEQ and AND rules
\[
\Gamma, x_1 : \tau_1, x_2 : \tau_2, \Gamma' \vdash e_1 : \tau'
\]
\[
\Gamma, x_1 : \tau_1, x_2 : \tau_2, \Gamma', z : \tau' \vdash e_2 : \tau_2
\]
\[
\frac{\Gamma, x_1 : \tau_1, x_2 : \tau_2, \Gamma' \vdash e_1 : \tau'}{\Gamma, \Gamma' \vdash \text{let } z = e_1 \text{ in } e_2 : \tau_2} \quad \text{LET}
\]

By assumption

Subderivation 1

Subderivation 2

Induction on s.d. 1

Extended context

\[
\Gamma, x_2 : \tau_2, x_1 : \tau_1, \Gamma' \vdash e_1 : \tau_1
\]

Induction on s.d. 2

By rule LET

\[
\Gamma, x_2 : \tau_2, x_1 : \tau_1, \Gamma', z : \tau_1 \vdash e_2 : \mathbb{N}
\]
\[
\frac{z : \tau \in \Gamma, x_1 : \tau_1, x_2 : \tau_2, \Gamma'}{\Gamma, \Gamma' \vdash z : \tau}
\]

By assumption

By assumption

An element of a list is also in a permutation of the list

By rule VAR
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

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

• Similarly for TRUE and FALSE rules
Proving Substitution, 2/4

\[
\frac{\Gamma, x : \tau \vdash e_1 : \mathbb{N} \quad \Gamma, x : \tau \vdash e_2 : \mathbb{N}}{\Gamma, x : \tau \vdash e_1 + e_2 : \mathbb{N}}
\]

By assumption: (1)

\[
\Gamma \vdash e : \tau
\]

By assumption: (2)

\[
\Gamma, x : \tau \vdash e_1 : \mathbb{N}
\]

Subderivation of (1): (3)

\[
\Gamma, x : \tau \vdash e_2 : \mathbb{N}
\]

Subderivation of (1): (4)

\[
\Gamma \vdash [e/x]e_1 : \mathbb{N}
\]

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

\[
\Gamma \vdash [e/x]e_2 : \mathbb{N}
\]

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

\[
\Gamma \vdash [e/x](e_1 + e_2) : \mathbb{N}
\]

By rule PLUS on (5), (6)

Def. of substitution

• Similarly for LEQ and AND rules
Proving Substitution, 3/4

\[ \Gamma, x : \tau \vdash e_1 : \tau' \quad \Gamma, x : \tau, z : \tau' \vdash e_2 : \tau_2 \]

\[ \Gamma, x : \tau \vdash \text{let } z = e_1 \text{ in } e_2 : \tau_2 \] \hspace{1cm} \text{LET} \quad \text{By assumption: (1)}

\[ \Gamma \vdash e : \tau \] \hspace{1cm} \text{By assumption: (2)}

\[ \Gamma, x : \tau \vdash e_1 : \tau' \] \hspace{1cm} \text{Subderivation of (1): (3)}

\[ \Gamma, x : \tau, z : \tau' \vdash e_2 : \tau_2 \] \hspace{1cm} \text{Subderivation of (1): (4)}

\[ \Gamma \vdash [e/x]e_1 : \tau' \] \hspace{1cm} \text{Induction on (2) and (3): (4)}

\[ \Gamma, z : \tau' \vdash e : \tau \] \hspace{1cm} \text{Weakening on (2): (5)}

\[ \Gamma, z : \tau', x : \tau \vdash e_2 : \tau_2 \] \hspace{1cm} \text{Exchange on (4): (6)}

\[ \Gamma, z : \tau' \vdash [e/x]e_2 : \tau_2 \] \hspace{1cm} \text{Induction on (5) and (6): (7)}

\[ \Gamma \vdash \text{let } z = [e/x]e_1 \text{ in } [e/x]e_2 : \tau_2 \] \hspace{1cm} \text{By rule LET on (6), (7)}

\[ \Gamma \vdash [e/x](\text{let } z = e_1 \text{ in } e_2) : \tau_2 \] \hspace{1cm} \text{By def. of substitution}
Proving Substitution, 4a/4

\[
\begin{align*}
z : \tau' & \in \Gamma, x : \tau \\
\Gamma, x : \tau & \vdash z : \tau' \quad \text{VAR} \\
\Gamma & \vdash e : \tau \quad \text{By assumption} \\
\text{Case } x = z : \\
\Gamma & \vdash [e/x]x : \tau \quad \text{By def. of substitution}
\end{align*}
\]
Proving Substitution, 4b/4

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

Case \( x \neq z \):

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

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

Values \( v \ ::= \ n | \ true | \ false \)

\[
\begin{align*}
& e_1 \leadsto e_1' \quad \text{ANDCONG} \quad \frac{e_1 \land e_2 \leadsto e_1' \land e_2}{e_1 \land e_2 \leadsto e_1'} \\
& true \land e \leadsto e \quad \text{ANDTRUE} \\
& false \land e \leadsto false \quad \text{ANDFALSE}
\end{align*}
\]

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

\[
\begin{align*}
& e_1 \leadsto e_1' \quad \text{LETCONG} \quad \frac{\text{let } z = e_1 \text{ in } e_2 \leadsto \text{let } z = e_1' \text{ in } e_2}{e_1 \leadsto e_1'} \\
& \text{let } z = v \text{ in } e_2 \leadsto [v/z]e_2 \quad \text{LETSTEP}
\end{align*}
\]
• A *reduction sequence* is a sequence of transitions \( e_0 \leadsto e_1, e_1 \leadsto e_2, \ldots, e_{n-1} \leadsto e_n \).

• 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 $e \rightsquigarrow e'$.
2. (Preservation) If $\cdot \vdash e : \tau$ and $e \rightsquigarrow e'$ 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

\[
\text{Num} \quad \vdash n : \mathbb{N}
\]

By assumption

\(n\) is a value   Def. of value grammar

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

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

\[ \frac{\vdash e_1 : \tau \quad x : \tau \vdash e_2 : \tau'}{...} \quad \text{LET} \]

By assumption: (1)

Subderivation of (1): (2)

Subderivation of (1): (3)

\( e_1 \leadsto e'_1 \) or \( e_1 \) value

Induction on (2)

Case \( e_1 \leadsto e'_1 \):

- \( \text{let}\ x = e_1\ \text{in}\ e_2 \leadsto \text{let}\ x = e'_1\ \text{in}\ e_2 \) By rule LETCONG

Case \( e_1 \) value:

- \( \text{let}\ x = e_1\ \text{in}\ e_2 \leadsto [e_1/x]e_2 \) By rule LETSTEP
(Preservation) If $\cdot \vdash e : \tau$ and $e \leadsto e'$ then $\cdot \vdash e' : \tau$.

1. We will use structural induction again, but on which derivation?
2. Two choices: (1) $\cdot \vdash e : \tau$ and (2) $e \leadsto e'$
3. The right choice is induction on $e \leadsto e'$
4. We will still need to deconstruct $\cdot \vdash e : \tau$ alongside it!
Type Preservation: Let Bindings 1

\[ e_1 \sim e'_1 \]

\[
\frac{
\text{let } x = e_1 \text{ in } e_2 \sim \text{let } x = e'_1 \text{ in } e_2
}{
\text{By assumption: (1)}
}\]

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

\[
\text{By assumption: (2)}
\]

\[
\text{Subderivation of (1): (3)}
\]

\[
\text{Subderivation of (2): (4)}
\]

\[
\text{Subderivation of (2): (5)}
\]

\[
\text{Induction on (3), (4): (6)}
\]

\[
\text{Rule LET on (6), (4)}
\]
let \( x = v_1 \) in \( e_2 \) \( \leadsto \) \( [v_1/x]e_2 \)  

By assumption: (1)

\[
\begin{align*}
\cdot &\vdash v_1 : \tau \\
\cdot &\vdash x : \tau \vdash e_2 : \tau' \\
\cdot &\vdash \text{let } x = v_1 \text{ in } e_2 : \tau'
\end{align*}
\]

By assumption: (2)

\[
\begin{align*}
\cdot &\vdash v_1 : \tau \\
x : \tau &\vdash e_2 : \tau' \\
\cdot &\vdash \left[ v_1/x \right] e_2 : \tau'
\end{align*}
\]

Subderivation of (2): (3)

Subderivation of (2): (4)

Substitution on (3), (4)
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 $\oplus$.
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.)