Type-Checking

Credit: presentation of type-checking follows Krishnamurthi (2007)
Types

A type is...

✦ a set of values that **share some property**
✦ a **promise to produce** a member of a certain set of values
✦ a **prediction about the value** an expression will yield
## Our type judgments

<table>
<thead>
<tr>
<th>Numbers</th>
<th>Booleans</th>
<th>Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma \vdash e:\text{number}$</td>
<td>$\Gamma \vdash e:\text{bool}$</td>
<td>$\Gamma(e) = \tau$</td>
</tr>
<tr>
<td><strong>Addition</strong></td>
<td></td>
<td>$\Gamma \vdash e:\tau$</td>
</tr>
<tr>
<td>$\Gamma \vdash e_1:\text{number}$</td>
<td>$\Gamma \vdash e_2:\text{number}$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\Gamma \vdash (+ e_1 e_2):\text{number}$</td>
</tr>
<tr>
<td><strong>Functions</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\Gamma(x \vdash \tau_1)$</td>
<td>$\Gamma \vdash e:\tau_2$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\Gamma \vdash (\lambda x:\tau_1 e):\tau_1 \rightarrow \tau_2$</td>
</tr>
<tr>
<td><strong>Function Applications</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\Gamma \vdash e_1:\tau_1 \rightarrow \tau_2$</td>
<td>$\Gamma \vdash e_2:\tau_3$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\Gamma \vdash (e_1 e_2):\tau_2$ if $\tau_1 = \tau_3$</td>
</tr>
</tbody>
</table>
Practice:

Check the following expression using our type judgments:

```
((lambda (y: number)
  (lambda (x: number)
    (+ x y)) 10) 9)
```
Let's enrich our language with if-expressions.

\[
\Gamma \vdash p : ??? \quad \Gamma \vdash t : ??? \quad \Gamma \vdash e : ??? \\
\Gamma \vdash (\text{if } p \; t \; e) : ???
\]
Conditionals

The predicate is the easy part: it must be boolean.

\[
\Gamma \vdash p : \text{bool} \quad \Gamma \vdash t : ??? \quad \Gamma \vdash e : ???
\]

\[\Gamma \vdash (\text{if } p \text{ t e}) : ???\]
Conditionals

What about the \texttt{if} and \texttt{else} branches?

\[
\Gamma \vdash p : \texttt{bool} \quad \Gamma \vdash t : ??? \quad \Gamma \vdash e : ??? \\
\Gamma \vdash (\texttt{if} \ p \ t \ e) : ???
\]
What about the `if` and `else` branches?

Consider:

\[
\text{(if } (= 0 \, x) \, \#f \, (\, / \, 10 \, x)\text{)}
\]

\[
\Gamma \vdash p : \text{bool} \quad \Gamma \vdash t : ??? \quad \Gamma \vdash e : ??? \\
\therefore \Gamma \vdash \text{(if } p \, t \, e \text{)} : ???
\]
What about the \texttt{if} and \texttt{else} branches?

Maybe they can be any type!

\[
\Gamma \vdash p : \texttt{bool} \quad \Gamma \vdash t : \texttt{tau1} \quad \Gamma \vdash e : \texttt{tau2}
\]

\[
\Gamma \vdash \texttt{(if p t e)} : ???
\]
Conditionals

What about the **if** and **else** branches?

Maybe they can be any type!

**Problem**: now we can’t type the whole expression!

\[
\begin{align*}
\Gamma \vdash p : \text{bool} & \quad \Gamma \vdash t : \tau_1 & \quad \Gamma \vdash e : \tau_2 \\
\Gamma \vdash (\text{if } p \text{ } \text{t} \text{ } e) : \text{??}
\end{align*}
\]
Conditionals

What about the `if` and `else` branches?

**Problem**: now we can’t type the whole expression!

**Solution**: force both branches to be the same type.

\[
\Gamma \vdash p : \text{bool} \quad \Gamma \vdash t : \tau \quad \Gamma \vdash e : \tau
\]

\[
\Gamma \vdash (\text{if } p \text{ } t \text{ } e) : \tau
\]
Conditionals

**Solution**: force both branches to be the same type.

\[
\Gamma \vdash p : \text{bool} \quad \Gamma \vdash t : \tau \quad \Gamma \vdash e : \tau \\
\hline
\Gamma \vdash (\text{if } p \text{ } t \text{ } e) : \tau
\]
Evaluating our type judgment rules

How do we evaluate our type-checking system?

**Idea #1:** Try out some programs
Evaluating our type judgment rules

How do we evaluate our type-checking system?

Idea #1: Try out some programs

Idea #2: Implement and try out more programs
Implementing a Racket Type-Checker
Evaluating our type judgment rules

How do we evaluate our type-checking system?

**Idea #1**: Try out some programs

**Idea #2**: Implement and try out more programs
Evaluating our type judgment rules

How do we evaluate our type-checking system?

- **Idea #1**: Try out some programs
- **Idea #2**: Implement and try out more programs
- **Idea #3**: Prove it correct
Type soundness

The goal of static typing is to learn some things about our program’s behavior without running it. This lets us catch errors early and makes our programs safer.

*Well-typed programs do not go wrong.*

Robin Milner
Type soundness

One useful property is **type soundness**. A sound type system is one that correctly predicts the type of the values that the program produces.

More formally:

**Type soundness**: for all programs $p$, if (1) the type checker assigns $p$ the type $\text{tau}$, and if (2) the semantics says that $p$ evaluates to a value $v$, then the type checker will also assign $v$ the type $\text{tau}$.
Type soundness: for all programs $p$, if (1) the type checker assigns $p$ the type $\tau$, and if (2) the semantics says that $p$ evaluates to a value $v$, then the type checker will also assign $v$ the type $\tau$.

What does this say about cases where the program doesn’t evaluate to a value, such as:

- side effects
- failure to terminate
- errors
Type soundness

Also, note the two components of this definition!

Type soundness requires participation from the run-time system (which implements the semantics).

**Type soundness:** for all programs $p$, if (1) the type checker assigns $p$ the type $\tau$, and if (2) the semantics says that $p$ evaluates to a value $v$, then the type checker will also assign $v$ the type $\tau$. 
Type safety: no primitive operation ever applies to values of the wrong type.

This can’t be guaranteed by the static type-checker alone. As we have seen, many aspects of program behavior can’t be statically checked.

Instead, this is a property of the type system + the run-time system: a safe language is one where the run-time system respects type abstractions.
Can all errors be statically caught?

We’ve seen some success in catching errors with our simple static type-checker. But are there errors that can’t be caught?
The limits of static type-checking

Some type errors can’t be caught statically. 😢

(\ 7 (- 1 1))
The limits of static type-checking

Static type-checking restricts some dynamic behavior:

```
static type checker

(if #f then (+ 1 #t ) else 1)

Is this an error?
```
Static versus Dynamic Type-Checking
Static Type Checking

- May reject a program after parsing, before running.
- Catches some bugs early (before running)
- May reject programs that would not have produced an error (over-cautious)
Dynamic Type Checking

- Only rejects syntax errors before running
- Doesn’t reject any programs unnecessarily
Which is better?

Convenience:
It’s annoying to write types... but it’s also annoying to have bugs.
Which is better?

Convenience:
It’s annoying to write types… but it’s also annoying to have bugs.

Expressiveness:
Static typing rejects some programs that run and can make some abstractions difficult to implement.
Which is better?

Convenience:
It’s annoying to write types… but it’s also annoying to have bugs.

Expressiveness:
Static typing rejects some programs that run and can make some abstractions difficult to implement.

Security:
Static typing catches more bugs… but not all bugs.
Which is better?

Convenience:
It’s annoying to write types… but it’s also annoying to have bugs.

Expressiveness:
Static typing rejects some programs that run and can make some abstractions difficult to implement.

Security:
Static typing catches more bugs… but not all bugs.

Efficiency:
Compilers can use static types to optimize code… compilers can also remove unnecessary dynamic checks.
Which is better?

Convenience:
It’s annoying to write types… but it’s also annoying to have bugs.

Expressiveness:
Static typing rejects some programs that run and can make some abstractions difficult to implement.

Security:
Static typing catches more bugs… but not all bugs.

Efficiency:
Compilers can use static types to optimize code… compilers can also remove unnecessary dynamic checks.

Code reuse:
Untyped code can be applied more flexibly… but advanced type systems also allow flexible use. Types are useful for documentation.
Summary

- It’s a complex issue!
- Static versus dynamic may be too broad of a debate
- Better questions: - **what** checks should be enforced?
  - **when** should checks be enforced?