Simply-Typed Lambda Calculus

You guys are both my witnesses... He insinuated that ZFC set theory is superior to Type Theory!
One-Slide Summary

• A **type** is an upper bound on the range of values a program expression could take on at run-time.

• A formal **type system**, also known as a **static semantics**, describes rules for checking types.

• A **typing judgment** typically associates a **typing environment** and an expression with a type.

• The **simply-typed lambda calculus** adds type annotations for function abstractions.

• A type system is **sound** iff every expression evaluates to a value in that expression's static type.
Review!

- What is operational semantics? When would you use contextual (small-step) semantics?
- What is satisfiability modulo theories?
- What is axiomatic semantics? What is a verification condition?
Today’s (Short?) Cunning Plan

• Type System Overview
• First-Order Type Systems
• Typing Rules
• Typing Derivations
• Type Safety

Cartoon:

What does MFU2 mean on your timeline?

That’s management foul-up number two. It usually happens around the third week.

We don’t anticipate any management mistakes.

That’s MFU1.
Types

- A program variable can assume a range of values during the execution of a program.

- An upper bound of such a range is called a type of the variable.
  - A variable of type “bool” is supposed to assume only boolean values.
  - If x has type “bool” then the boolean expression “not(x)” has a sensible meaning during every run of the program.
Typed and Untyped Languages

- **Untyped languages**
  - Do *not* restrict the range of values for a given variable
  - Operations might be applied to inappropriate arguments. The behavior in such cases might be unspecified
  - The pure \( \lambda \)-calculus is an extreme case of an untyped language (however, its behavior is completely specified)

- **(Statically) Typed languages**
  - Variables are assigned (non-trivial) types
  - A type system keeps track of types
  - Types might or might not appear in the program itself
  - Languages can be explicitly typed or implicitly typed
The Purpose Of Types

• The foremost purpose of types is to prevent certain types of run-time execution errors
• Traditional trapped execution errors
  - Cause the computation to stop immediately
  - And are thus well-specified behavior
  - Usually enforced by hardware
  - e.g., Division by zero, floating point op with a NaN
  - e.g., Dereferencing the address 0 (on most systems)
• Untrapped execution errors
  - Behavior is unspecified (depends on the state of the machine = this is very bad!)
  - e.g., accessing past the end of an array
  - e.g., jumping to an address in the data segment
Execution Errors

- A program is deemed **safe** if it does **not** cause untrapped errors
  - Languages in which all programs are safe are **safe languages**
- For a given language we can designate a set of forbidden errors
  - A superset of the untrapped errors, usually including some trapped errors as well
    - e.g., null pointer dereference
- **Modern Type System Powers:**
  - prevent race conditions (e.g., Flanagan TLDI ‘05)
  - prevent insecure information flow (e.g., Li POPL ’05)
  - prevent resource leaks (e.g., Vault, Weimer)
  - help with generic programming, probabilistic languages, ...
  - ... are often combined with dynamic analyses (e.g., CCured)
Preventing Forbidden Errors - Static Checking

- Forbidden errors can be caught by a combination of static and run-time checking

- Static checking
  - Detects errors early, *before testing*
  - Types provide the necessary static information for static checking
  - e.g., ML, Modula-3, Java
  - Detecting certain errors statically is *undecidable* in most languages
Preventing Forbidden Errors - Dynamic Checking

- Required when static checking is undecidable
  - e.g., array-bounds checking
- Run-time encodings of types are still used (e.g. Lisp)
- Should be limited since it delays the manifestation of errors
- Can be done in hardware (e.g. null-pointer)
Why Typed Languages?

- **Development**
  - *Type checking catches early many mistakes*
  - Reduced debugging time
  - Typed signatures are a powerful basis for design
  - Typed signatures enable separate compilation

- **Maintenance**
  - Types act as checked specifications
  - Types can enforce abstraction

- **Execution**
  - Static checking reduces the need for dynamic checking
  - *Safe languages are easier to analyze statically*
    - the compiler can generate better code
Why Not Typed Languages?

- Static type checking imposes constraints on the programmer
  - Some valid programs might be rejected
  - But often they can be made well-typed easily
  - Hard to step outside the language (e.g. OO programming in a non-OO language, but cf. Ruby, OCaml, etc.)

- Dynamic safety checks can be costly
  - 50% is a possible cost of bounds-checking in a tight loop
    - In practice, the overall cost is much smaller
  - Memory management must be automatic ⇒ need a garbage collector with the associated run-time costs
  - Some applications are justified in using weakly-typed languages (e.g., by external safety proof)
Safe Languages

- There are typed languages that are not safe ("weakly typed languages")
- All safe languages use types (static or dynamic)

<table>
<thead>
<tr>
<th></th>
<th>Typed</th>
<th>Untyped</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Static</td>
<td>Dynamic</td>
</tr>
<tr>
<td>Safe</td>
<td>ML, Java, Ada, C#, Haskell, ...</td>
<td>Lisp, Scheme, Ruby, Perl, Smalltalk, PHP, Python, ...</td>
</tr>
<tr>
<td>Unsafe</td>
<td>C, C++, Pascal, ...</td>
<td>?</td>
</tr>
</tbody>
</table>

- We focus on statically typed languages
Properties of Type Systems

• How do types differ from other program annotations?
  - Types are more precise than comments
  - Types are more easily mechanizable than program specifications

• Expected properties of type systems:
  - Types should be enforceable
  - Types should be checkable algorithmically
  - Typing rules should be transparent
    • Should be easy to see why a program is not well-typed
Why Formal Type Systems?

• Many typed languages have \textit{informal descriptions} of the type systems (e.g., in language reference manuals)

• A fair amount of careful analysis is required to \textit{avoid false claims} of type safety

• A formal presentation of a type system is a \textit{precise specification of the type checker}
  - And allows formal proofs of type safety

• But even informal knowledge of the principles of type systems help
Formalizing a Language

1. Syntax
   - Of expressions (programs)
   - Of types
   - Issues of binding and scoping

2. Static semantics (typing rules)
   - Define the typing judgment and its derivation rules

3. Dynamic Semantics (e.g., operational)
   - Define the evaluation judgment and its derivation rules

4. Type soundness
   - Relates the static and dynamic semantics
   - State and prove the soundness theorem
Typing Judgments

- **Judgment** (recall)
  - A statement $J$ about certain formal entities
  - Has a truth value $\vdash J$
  - Has a derivation $\vdash J$ (= “a proof”)

- A common form of **typing judgment**:
  \[
  \Gamma \vdash e : \tau
  \]
  (e is an expression and $\tau$ is a type)

- $\Gamma$ (Gamma) is a set of **type assignments for the free variables of e**
  - Defined by the grammar $\Gamma ::= \cdot | \Gamma, x : \tau$
  - Type assignments for variables not free in $e$ are not relevant
  - e.g., $x : \text{int}, y : \text{int} \vdash x + y : \text{int}$
Typing rules

- **Typing rules** are used to derive typing judgments

- Examples:

\[
\Gamma \vdash 1 : \text{int} \\
\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \\
\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}}
\]
Typing Derivations

• A **typing derivation** is a derivation of a typing judgment (big surprise there ...)

• Example:

\[
\begin{array}{c}
\Gamma \vdash x : \text{int} & \quad & \Gamma \vdash 1 : \text{int} \\
\hline
\Gamma \vdash x : \text{int} & \quad & \Gamma \vdash x + 1 : \text{int} \\
\hline
\Gamma \vdash x + (x + 1) : \text{int}
\end{array}
\]

• We say \( \Gamma \vdash e : \tau \) to mean there exists a derivation of this typing judgment (= “we can prove it”)

• **Type checking**: given \( \Gamma \), \( e \) and \( \tau \) find a derivation

• **Type inference**: given \( \Gamma \) and \( e \), find \( \tau \) and a derivation
Proving Type Soundness

- A typing judgment is either true or false
- Define what it means for a value to have a type
  \[ v \in \| \tau \| \]
  (e.g. \( 5 \in \| \text{int} \| \) and \( \text{true} \in \| \text{bool} \| \))
- Define what it means for an expression to have a type
  \[ e \in \mid \tau \mid \text{ iff } \forall v. (e \downarrow v \Rightarrow v \in \| \tau \|) \]
- Prove type soundness
  \[ \text{If } \vdash e : \tau \text{ then } e \in \mid \tau \mid \]
  or equivalently
  \[ \text{If } \vdash e : \tau \text{ and } e \downarrow v \text{ then } v \in \| \tau \| \]
- This implies safe execution (since the result of a unsafe execution is not in \( \| \tau \| \) for any \( \tau \))
Upcoming Exciting Episodes

• We will give formal description of first-order type systems (no type variables)
  - Function types (simply typed λ-calculus)
  - Simple types (integers and booleans)
  - Structured types (products and sums)
  - Imperative types (references and exceptions)
  - Recursive types (linked lists and trees)

• The type systems of most common languages are first-order

• Then we move to second-order type systems
  - Polymorphism and abstract types
This 1988 animated movie written and directed by Isao Takahata for Studio Ghibli was considered by Roger Ebert to be one of the most powerful anti-war films ever made. It features Seita and his sister Setsuko and their efforts to survive outside of society during the firebombing of Tokyo.
Computer Science

- This American-Canadian Turing-award winner is known for major contributions to the fields of complexity theory and proof complexity. He is known for formalizing the polynomial-time reduction, NP-completeness, P vs. NP, and showing that SAT is NP-complete. This was all done in the seminal 1971 paper *The Complexity of Theorem Proving Procedures*. 
Q: Student

- This piece of diving equipment with an air-inflatable bladder changes its average density for use in SCUBA diving. It typically requires manual adjustment throughout the dive and can be augmented by breath control.
• This 1985 falling-blocks computer game was invented by Alexey Pajitnov (Алексей Пажитнов) and inspired by pentominoes.
Simply-Typed Lambda Calculus

• Syntax:

Terms

\[ e ::= x \mid \lambda x: \tau. \; e \mid e_1 \; e_2 \]

\[ \mid n \mid e_1 + e_2 \mid \text{iszero} \; e \]

\[ \mid \text{true} \mid \text{false} \mid \text{not} \; e \]

\[ \mid \text{if} \; e_1 \; \text{then} \; e_2 \; \text{else} \; e_3 \]

Types

\[ \tau ::= \text{int} \mid \text{bool} \mid \tau_1 \rightarrow \tau_2 \]

• \( \tau_1 \rightarrow \tau_2 \) is the function type

• \( \rightarrow \) associates to the right

• Arguments have typing annotations \( :\tau \)

• This language is also called \( F_1 \)
Static Semantics of F₁

• The typing judgment

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

• Some (simpler) typing rules:

\[ x : \tau \in \Gamma \quad \Rightarrow \quad \Gamma \vdash x : \tau \]

\[ \Gamma, x : \tau \vdash e : \tau' \quad \Rightarrow \quad \Gamma \vdash \lambda x : \tau. e : \tau \rightarrow \tau' \]

\[ \Gamma \vdash e_1 : \tau_2 \rightarrow \tau \quad \Gamma \vdash e_2 : \tau_2 \quad \Rightarrow \quad \Gamma \vdash e_1 e_2 : \tau \]
More Static Semantics of F₁

\[
\begin{align*}
\Gamma \vdash n : \text{int} & \quad \Gamma \vdash e_1 : \text{int} & \quad \Gamma \vdash e_2 : \text{int} \\
\hline
\Gamma \vdash e_1 + e_2 : \text{int} & \quad \Gamma \vdash e : \text{bool} \\
\hline
\Gamma \vdash \text{true} : \text{bool} & \quad \Gamma \vdash \text{not } e : \text{bool} \\
\hline
\Gamma \vdash e_1 : \text{bool} & \quad \Gamma \vdash e_t : \tau & \quad \Gamma \vdash e_f : \tau \\
\hline
\Gamma \vdash \text{if } e_1 \text{ then } e_t \text{ else } e_f : \tau
\end{align*}
\]

Why do we have this mysterious gap? I don’t know either!
Typing Derivation in $F_1$

• Consider the term (also underlined below)
  \[ \lambda x : \text{int. } \lambda b : \text{bool. } \text{if } b \text{ then } f \ x \text{ else } x \]
  - With the initial typing assignment \( f : \text{int} \rightarrow \text{Int} \)
  - Where \( \Gamma = f : \text{int} \rightarrow \text{int}, x : \text{int}, b : \text{bool} \)

\[
\frac{
\frac{
\frac{\Gamma \vdash f : \text{int} \rightarrow \text{int} \quad \Gamma \vdash x : \text{int}}{
\Gamma \vdash b : \text{bool} \quad \Gamma \vdash f \ x : \text{int} \quad \Gamma \vdash x : \text{int}}
}{f : \text{int} \rightarrow \text{int}, x : \text{int}, b : \text{bool} \vdash \text{if } b \text{ then } f \ x \text{ else } x : \text{int}}
\]

\[
\frac{
\frac{f : \text{int} \rightarrow \text{int}, x : \text{int} \vdash \lambda b : \text{bool. } \text{if } b \text{ then } f \ x \text{ else } x : \text{bool} \rightarrow \text{int}}{
f : \text{int} \rightarrow \text{int} \vdash \lambda x : \text{int. } \lambda b : \text{bool. } \text{if } b \text{ then } f \ x \text{ else } x : \text{int} \rightarrow \text{bool} \rightarrow \text{int}}
\]
Type Checking in $F_1$

- **Type checking** is *easy* because
  - Typing rules are *syntax directed*
  - Typing rules are *compositional* (what does this mean?)
  - All local variables are annotated with types

- In fact, **type inference** is *also easy* for $F_1$

- Without type annotations an expression may have **no unique type**
  - $\vdash \lambda x. \ x : \text{int} \to \text{int}$
  - $\vdash \lambda x. \ x : \text{bool} \to \text{bool}$
Operational Semantics of $F_1$

- Judgment:
  $$e \Downarrow v$$

- Values:
  $$v ::= n \mid \text{true} \mid \text{false} \mid \lambda x: \tau. \ e$$

- The evaluation rules ...
  
  - *Audience participation time*: “raise your hand” and give me an opsem evaluation rule.
Opsem of $F_1$ (Cont.)

- **Call-by-value evaluation rules (sample)**

\[
\begin{align*}
\lambda x : \tau . e & \Downarrow \lambda x : \tau . e \\
\lambda x : \tau . e_1 & \Downarrow \lambda x : \tau . e' \\
\lambda x : \tau . e_1 & \Downarrow \lambda x : \tau . e' \\
[e_2/x]e'_1 & \Downarrow v \\
\end{align*}
\]

\[
\begin{align*}
e_1 e_2 & \Downarrow v \\
\end{align*}
\]

\[
\begin{align*}
e_1 & \Downarrow n_1 \\
e_2 & \Downarrow n_2 \\
n & = \color{red}{n_1 + n_2} \\
n & \Downarrow n \\
\end{align*}
\]

\[
\begin{align*}
e_1 & \Downarrow true \\
e_t & \Downarrow v \\
\text{if } e_1 \text{ then } e_t \text{ else } e_f & \Downarrow v \\
\end{align*}
\]

\[
\begin{align*}
e_1 & \Downarrow false \\
e_f & \Downarrow v \\
\text{if } e_1 \text{ then } e_t \text{ else } e_f & \Downarrow v \\
\end{align*}
\]

Where is the Call-By-Value? How might we change it?

Evaluation is **undefined** for ill-typed programs!
Type Soundness for $F_1$

- Thm: $\frac{}{\Gamma \vdash e : \tau \text{ and } e \Downarrow v \text{ then } \Gamma \vdash v : \tau}$
  - Also called, subject reduction theorem, type preservation theorem

- This is one of the most important sorts of theorems in PL

- Whenever you make up a new safe language you are expected to prove this
  - Examples: Vault, TAL, CCured, ...

- Proof: next time!
Homework

- Read actually-exciting Leroy paper
- Homework continues