Announcements

- I have commented on most of your proposals

Quick Review

- λ-calculus is as expressive as a Turing machine
- We can encode a multitude of data types in the untyped λ-calculus
- To simplify programming it is useful to add types to the language
- We now start the study of type systems in the context of the typed λ-calculus

Today's Plan

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

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 λ-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 with a NaN
    – e.g., Dereferencing the address 0 (on most systems)

• Untrapped execution errors
  – Behavior is unspecified
    – 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
  – For a given program language we can designate a set of forbidden errors
  – Usually enforced by hardware (e.g., ML, Modula-3, Java)

Preventing Forbidden Errors:

• 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

• Dynamic Checking
  – Run-time checking of types are still required
  – Should be limited since it delays the manifestation of errors
  – e.g., array-bounds checking
  – Can be done in hardware (e.g., null-pointer)

Preventing Forbidden Errors:

• Static Checking
  – Detects errors early, before testing
  – Types provide the necessary static information for static checking
  – e.g., ML, Modula-3, Java

• Dynamic Checking
  – Run-time checking of types are still required
  – Should be limited since it delays the manifestation of errors
  – e.g., array-bounds checking
  – 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. 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>Static</td>
<td>Java, ML</td>
<td>Python, Perl, Ruby, Bash, Lisp</td>
</tr>
<tr>
<td>Dynamic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Safe</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unsafe</td>
<td>C, C++, Pascal, ...</td>
<td>Assembly</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 informal descriptions of the type systems (e.g., in language reference manuals)

Why Formal Type Systems?

• Many typed languages have informal descriptions of the type systems (e.g., in language reference manuals)
• A fair amount of careful analysis is required to avoid false claims of type safety
• A formal presentation of a type system is a 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

• Recall: judgment?
  - A statement about the world
  - A statement that can be proven
• 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

\[ \Gamma \vdash 1 : \text{int} \]

- Examples:

\[ \vdash x : \tau \]

\[ \Gamma, x : \tau \vdash e : \tau' \]

\[ \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau' \]

\[ \Gamma \vdash e_1 \downarrow e_2 \]

Typing Derivations

- A typing derivation is a derivation of a typing judgment (big surprise)

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

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

\[ \Gamma \vdash e_1 : \tau_2 \rightarrow \tau \quad \Gamma \vdash e_2 : \tau_2 \]

\[ \Gamma \vdash e_1 e_2 : \tau \]

Proving Type Soundness: Intuition

- A typing judgment
- 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 \( e \in \tau \) iff \( \forall v. (e \Downarrow v \Rightarrow v \in \tau) \)
- Prove type soundness
  - If \( \Gamma \vdash e : \tau \) then \( e \in \tau \)
  - or equivalently
  - If \( \Gamma \vdash e : \tau \) and \( e \Downarrow v \) then \( v \in \tau \)
- This implies safe execution (since the result of an unsafe execution is not in \( \tau \) for any \( \tau \))

Simply-Typed Lambda Calculus

- Syntax:

  \[ e ::= x | \lambda x: \tau. e | e_1 e_2 | n | e_1 + e_2 | \text{iszero } e | \text{true} | \text{false} | \text{if } e \text{ then } e_1 \text{ else } e_3 \]

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

- Notice the \( : \)

Static Semantics of \( F_1 \)

- Function rules

\[ \vdash x : \tau \quad \Gamma, x : \tau \vdash e : \tau' \]

\[ \Gamma \vdash e_1 : \tau_2 \quad \Gamma \vdash e_2 : \tau_2 \]

\[ \Gamma \vdash e_1 e_2 : \tau \]
More Static Semantics of $F_1$

- Base type rules

\[
\begin{align*}
\Gamma &\vdash n : \text{int} & \Gamma &\vdash e_1 + e_2 : \text{int} \\
\Gamma &\vdash e_1 : \text{int} & \Gamma &\vdash e_2 : \text{int} \\
\Gamma &\vdash \text{true} : \text{bool} & \Gamma &\vdash \text{not } e : \text{bool} \\
\Gamma &\vdash e_1 : \text{bool} & \Gamma &\vdash e_2 : \tau & \Gamma &\vdash e_3 : \tau \\
\Gamma &\vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau
\end{align*}
\]

Typing Derivation in $F_1$

- Consider the term $\lambda x : \text{int}. \lambda b : \text{bool}. \text{if } b \text{ then } f x \text{ else } x$
  - With the initial typing assignment $f : \text{int} \rightarrow \text{int}$

- Write the type derivation