Homework Five Is Alive

- Ocaml now installed on dept linux/solaris machines in /usr/cs (e.g., /usr/cs/bin/ocamlc)
- There will be no Number Six

Lecture Schedule

- Thu Oct 13 - Today
- Tue Oct 14 - Monomorphic Type Systems
- Thu Oct 12 - Exceptions, Continuations, Rec Types
- Tue Oct 17 - Subtyping
  - Homework 5 Due
- Thu Oct 19 - No Class
- Tue Oct 24 - 2nd Order Types | Dependent Types
  - Double Lecture
  - Food!
  - Project Status Update Due
- Thu Oct 26 - No Class
- Tue Oct 31 - Theorem Proving, Proof Checking

Back to School

- What is operational semantics? When would you use contextual (small-step) semantics?
- What is denotational semantics?
- What is axiomatic semantics? What is a verification condition?

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

Today’s (Short?) Cunning Plan

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

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 $\Gamma ::= e : \tau \mid \Gamma, x : \tau$
  - 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:

\[
\begin{align*}
\Gamma &\vdash 1 : \text{int} \\
\Gamma &\vdash x : \tau \quad (x \in \Gamma) \\
\Gamma &\vdash e_1 : \text{int} \quad \Gamma &\vdash e_2 : \text{int} \\
\Gamma &\vdash e_1 + e_2 : \text{int}
\end{align*}
\]

Typing Derivations

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

- Example:

\[
\begin{align*}
\Gamma &\vdash x : \text{int} \\
\Gamma &\vdash x : \text{int} \quad \Gamma &\vdash x + 1 : \text{int} \\
\Gamma &\vdash x : \text{int} \quad \Gamma &\vdash x + (x + 1) : \text{int}
\end{align*}
\]

- 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 \(e \in |\tau|\) iff \(\forall v. (e \Downarrow v \Rightarrow v \in \| \tau \|)\)

- Prove type soundness

\[
\begin{align*}
\text{If } \Gamma &\vdash e : \tau \\
\text{or equivalently} &\text{ then } e \in |\tau| \\
\text{If } \Gamma &\vdash e : \tau \quad \text{and } e \Downarrow v \\
\text{then } v &\in \| \tau \|
\end{align*}
\]

- 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 \(\lambda\)-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

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 \(\tau ::= \text{int} \mid \text{bool} \mid \tau_1 \to \tau_2\)

- \(\tau_1 \to \tau_2\) is the function type

- \(\to\) associates to the right

- Arguments have typing annotations \(\tau\)

- This language is also called \(F_1\)

Static Semantics of \(F_1\)

• The typing judgment

\[
\begin{align*}
\Gamma &\vdash e : \tau
\end{align*}
\]

• Some (simpler) typing rules:

\[
\begin{align*}
\Gamma &\vdash x : \tau \\
\Gamma &\vdash x : \tau \quad \Gamma &\vdash \lambda x : \tau. e : \tau \to \tau' \\
\Gamma &\vdash e_1 : \tau_2 \to \tau \\
\Gamma &\vdash e_2 : \tau_2 \\
\Gamma &\vdash e_1 e_2 : \tau
\end{align*}
\]
More Static Semantics of F₁

\[ \Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int} \]

Why do we leave this mysterious gap? I don’t know either!

Typing Derivation in F₁

- 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 \( f : \text{int} \rightarrow \text{Int} \)
  - Where \( \Gamma = f : \text{int} \rightarrow \text{int}, x : \text{int}, b : \text{bool} \)

Type Checking in F₁

- 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 F₁
- Without type annotations an expression may have no unique type

  \[ \vdash \lambda x. x : \text{int} \rightarrow \text{int} \]
  \[ \vdash \lambda x. x : \text{bool} \rightarrow \text{bool} \]

Operational Semantics of F₁

- 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 evaluation rule.

Opsem of F₁ (Cont.)

- Call-by-value evaluation rules (sample)

  \[ \begin{array}{l}
  \lambda x : \tau. e \Downarrow \lambda x : \tau. e' \\
  e_1 \Downarrow \lambda x : \tau. e' \\
  e_2 \Downarrow v \\
  \end{array} \quad \begin{array}{l}
  (e_2/x)e' \Downarrow v \\
  e_1 \Downarrow v \\
  e_2 \Downarrow v \\
  n \Downarrow n \\
  e_1 \Downarrow \text{true} \\
  e_1 \Downarrow v \\
  \end{array} \]

  \[ \begin{array}{l}
  \text{if } e_1 \text{ then } e_2 \text{ else } e_f \Downarrow v \\
  e_1 \Downarrow \text{false} \\
  e_f \Downarrow v \\
  \end{array} \]

Type Soundness for F₁

- Theorem: If \( \vdash e : \tau \) and \( e \Downarrow v \) then \( \vdash v : \tau \)
  - 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 Wright and Felleisen article
- Work on your projects!
  - Status Update Due Soon
- Work on Homework 5

The reading is not optional.