Homework Five Is Alive
- Homework 5 has not been returned
- Waiting on a few students who want to turn it in later
- There will be no Number Six

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?

Today’s Cunning Plan
- Type System Overview
- First-Order Type Systems
  - Typing Rules
  - Typing Derivations
  - Type Safety

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)
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
    - It 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 $\models J$
  - 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}
\quad x : \tau \in \Gamma
\quad \Gamma \vdash x : \tau
\quad \Gamma \vdash e_1 : \text{int}
\quad \Gamma \vdash e_2 : \text{int}
\quad \Gamma \vdash e_1 + e_2 : \text{int}
```

Typing Derivations

- A **typing derivation** is a derivation of a typing judgment (big surprise there ...)
- Example:

```
x : \text{int} \vdash x : \text{int}
x : \text{int} \vdash 1 : \text{int}
x : \text{int} \vdash x + 1 : \text{int}
x : \text{int} \vdash x + (x + 1) : \text{int}
```

- 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
  - 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 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
- The type systems of most common languages are first-order
- The we move to second-order type systems
  - Polymorphism and abstract types

Simply-Typed Lambda Calculus

- Syntax:
  - Terms $e ::= x \mid \lambda x: \tau. e \mid e_1 e_2 \mid n \mid e_1 + e_2 \mid \text{iszero } e \mid \text{true} \mid \text{false} \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3$
  - 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
- This language is also called $F_1$

Static Semantics of $F_1$

- The typing judgment $\Gamma \vdash e : \tau$
- Some (simpler) typing rules:
  
  \[
  \begin{align*}
  x : \tau & \in \Gamma & \Rightarrow \Gamma, x : \tau \vdash e : \tau' \\
  \Gamma \vdash x : \tau & & \Rightarrow \Gamma \vdash \lambda x : \tau. e : \tau' \to \tau' \\
  \Gamma \vdash e_1 : \tau_2 \to \tau & & \Gamma \vdash e_2 : \tau_2 \Rightarrow \Gamma \vdash e_1 e_2 : \tau \\
  \end{align*}
  \]

More Static Semantics of $F_1$

- 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 $f : \text{int} \to \text{int}$
  
  \[
  \begin{align*}
  \Gamma & : \text{f} : \text{int} \to \text{int} \quad \Gamma & : x : \text{int} \\
  \Gamma & : b : \text{bool} & \Gamma & : f x : \text{int} \quad \Gamma & : x : \text{int} \\
  \Gamma & : b : \text{bool} & \Gamma & : f x : \text{int} \quad \Gamma & : f x : \text{int} \\
  \end{align*}
  \]
  
  \[
  \begin{align*}
  \Gamma & : \text{f} : \text{int} \to \text{int} \quad \Gamma & : \text{x} : \text{int} \quad \Gamma & : \text{b} : \text{bool} \\
  \Gamma & : \text{f} : \text{int} \to \text{int} \quad \Gamma & : \text{x} : \text{int} \quad \Gamma & : \text{b} : \text{bool} \\
  \Gamma & : \text{true} : \text{bool} \quad \Gamma & : \text{not } e : \text{bool} \\
  \end{align*}
  \]

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 $f : \text{int} \to \text{int}$
  
  \[
  \begin{align*}
  \Gamma & : \text{f} : \text{int} \to \text{int} \quad \Gamma & : x : \text{int} \\
  \Gamma & : b : \text{bool} & \Gamma & : f x : \text{int} \quad \Gamma & : x : \text{int} \\
  \Gamma & : b : \text{bool} & \Gamma & : f x : \text{int} \quad \Gamma & : f x : \text{int} \\
  \end{align*}
  \]

Where $\Gamma = f : \text{int} \to \text{int}, x : \text{int}, b : \text{bool}$
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 evaluation rule.

Operational Semantics of $F_1$ (Cont.)

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

$$\begin{array}{c}
\lambda x : \tau. e_1 \\ e_2 \Downarrow v_2 \\
\vdash e_1 [v_2/x] \Downarrow v
\end{array}$$
$$\begin{array}{c}
e_1 \Downarrow n_1 \\
e_2 \Downarrow n_2 \\
n = n_1 + n_2
\end{array}$$

- Evaluation is **undefined** for ill-typed programs!

Type Soundness for $F_1$

- 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 Approaches To Type Safety

- Theorem: If $\vdash e : \tau$ and $e \Downarrow v$ then $\vdash v : \tau$

- To address the issue of $[v/x]e'$:
  - This is it!

- Try to prove by induction on $e$
  - Won’t work because $[v/x]e'$, in the evaluation of $e_1$, $e_2$
  - Same problem with induction on $\vdash e : \tau$

- Try to prove by induction on $\tau$
  - Won’t work because $e_1$ has a “bigger” type than $e_1$, $e_2$

- Try to prove by induction on $e \Downarrow v$
  - To address the issue of $[v/x]e'$
  - This is it!
Type Soundness Proof

- Consider the case
  \[ e_1 \vdash \lambda x : \tau_2 \cdot e'_1 \quad e_2 \vdash \nu_2 \quad [v_2/x]e'_1 \vdash v \]
  \[ e_1 \vdash \nu \]
  and by inversion on the derivation of \( e_1 : \tau \)

- From IH on \( e_1 \) we have \( \vdash \nu \)
- From IH on \( e_2 \) we have \( \vdash \nu_2 : \tau_2 \)
- Need to infer that \( \vdash [v_2/x]e'_1 : \tau \) and use the IH
  - We need a substitution lemma (by induction on \( e'_1 \))

Significance of Type Soundness

- The theorem says that the result of an evaluation has the same type as the initial expression
- The theorem does not say that
  - The evaluation never gets stuck (e.g., trying to apply a non-function, to add non-integers, etc.), nor that
  - The evaluation terminates
- Even though both of the above facts are true of \( F_1 \)
- We need a small-step semantics to prove that the execution never gets stuck
- I Assert: the execution always terminates in \( F_1 \)
  - When does the lambda calculus ever not terminate?

Small-Step Contextual Semantics for \( F_1 \)

- We define redexes
  \[ r := n_1 + n_2 \quad \text{if } b \text{ then } e_1 \text{ else } e_2 \quad (\lambda x : \tau . e_1) v_2 \]
- and contexts
  \[ H := H_1 + e_2 \quad n_1 + H_2 \quad \text{if } H \text{ then } e_1 \text{ else } e_2 \quad (\lambda x : \tau . e_1) H_2 \]
- and local reduction rules
  \[ n_1 + n_2 \rightarrow n_1 \text{ plus } n_2 \quad \text{if true then } e_1 \text{ else } e_2 \rightarrow e_1 \]
  \[ \text{if false then } e_1 \text{ else } e_2 \rightarrow e_2 \quad (\lambda x : \tau . e_1) v_2 \rightarrow [v_2/x]e_1 \]
- and one global reduction rule
  \[ H[r] \rightarrow H[e] \text{ iff } r \rightarrow e \]

Decomposition Lemmas for \( F_1 \)

1. If \( \vdash e : \tau \) and \( e \) is not a (final) value then there exist (unique) \( H \) and \( r \) such that \( e = H[r] \)
   - any well-typed expression can be decomposed
   - any well-typed non-value can make progress
2. Furthermore, there exists \( \tau' \) such that \( \vdash r : \tau' \)
   - the redex is closed and well typed
3. Furthermore, there exists \( e' \) such that \( r \rightarrow e' \) and \( \vdash e' : \tau' \)
   - local reduction is type preserving
4. Furthermore, for any \( e' \), \( \vdash e' : \tau' \) implies \( \vdash H[e'] : \tau \)
   - the expression preserves its type if we replace the redex with an expression of same type

Type Safety of \( F_1 \)

- Type preservation theorem
  - If \( \vdash e : \tau \) and \( e \rightarrow e' \) then \( \vdash e' : \tau \)
  - Follows from the decomposition lemma
- Progress theorem
  - If \( \vdash e : \tau \) and \( e \) is not a value then there exists \( e' \) such that \( e \) can make progress: \( e \rightarrow e' \)
  - Progress theorem says that execution can make progress on a well typed expression
  - From type preservation we know the execution of well typed expressions never gets stuck
  - This is a (very!) common way to state and prove type safety of a language

What’s Next?

- We’ve got the basic simply-typed monomorphic lambda calculus
- Now let’s make it more complicated …
- By adding features!
Product Types: Static Semantics

• Extend the syntax with (binary) tuples
  \[ e ::= ... | (e_1, e_2) | \text{fst } e | \text{snd } e \]
  \[ \tau ::= ... | \tau_1 \times \tau_2 \]
  - This language is sometimes called $F_{1 \times}$

• Same typing judgment
  \[ \Gamma \vdash e : \tau \]

\[
\begin{align*}
\Gamma &\vdash e_1 : \tau_1 & \Gamma &\vdash e_2 : \tau_2 \\
\Gamma &\vdash (e_1, e_2) : \tau_1 \times \tau_2 \\
\Gamma &\vdash e : \tau_1 \times \tau_2 & \Gamma &\vdash e : \tau_1 \times \tau_2 \\
\Gamma &\vdash \text{fst } e : \tau_1 & \Gamma &\vdash \text{snd } e : \tau_2
\end{align*}
\]

Product Types: Dynamic Semantics and Soundness

• New form of values:
  \[ v ::= ... | (v_1, v_2) \]

• New (big step) evaluation rules:
  \[
  \begin{align*}
  e_1 \Downarrow v_1 &\quad e_2 \Downarrow v_2 \\
  (e_1, e_2) \Downarrow (v_1, v_2) \\
  \text{fst } e \Downarrow v_1 &\quad \text{snd } e \Downarrow v_2
  \end{align*}
  \]

• New contexts:
  \[ H ::= ... | (H_1, e_2) | (v_1, H_2) | \text{fst } H | \text{snd } H \]

• New redexes:
  \[ \text{fst } (v_1, v_2) \rightarrow v_1 \]
  \[ \text{snd } (v_1, v_2) \rightarrow v_2 \]

• Type soundness holds just as before

General PL Feature Plan

• The general plan for language feature design
• You invent a new feature (tuples)
• You add it to the lambda calculus
• You invent typing rules and opsem rules
• You extend the basic proof of type safety
• You declare moral victory, and milling throngs of cheering admirers wait to carry you on their shoulders to be knighted by the Queen, etc.

Homework

• Read Wright and Felleisen article
• Work on your projects!
  - Status Update Due: Thursday Mar 23