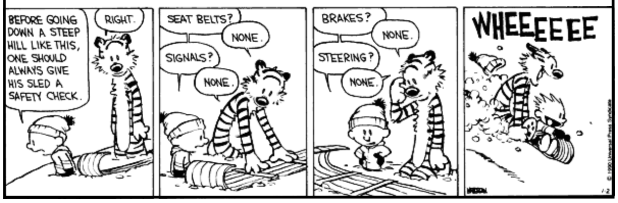
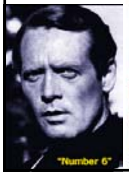
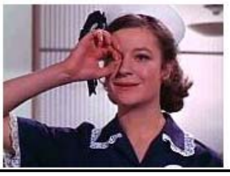
Simply-Typed Lambda Calculus

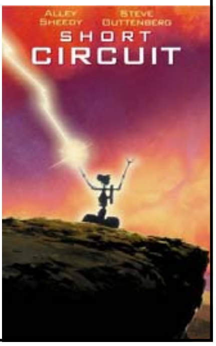


Homework Five Is Alive

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





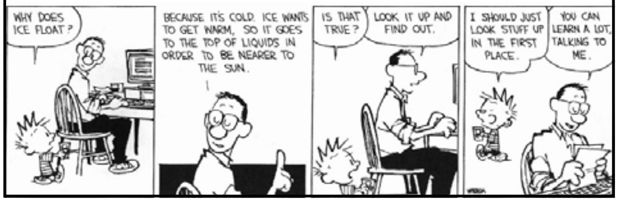


Lecture Schedule

- 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

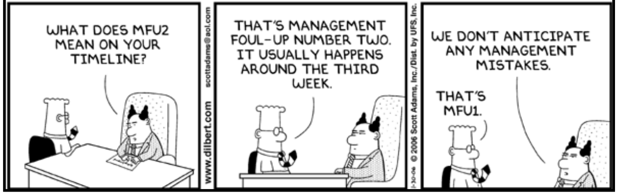
Back to School

- 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?



Today's (Short?) 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)

Safe Languages

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

	Typed		Untyped
	Static	Dynamic	
Safe	ML, Java, Ada, C#, Haskell, ...	Lisp, Scheme, Ruby, Perl, Smalltalk, PHP, Python, …	λ-calculus
Unsafe	C, C++, Pascal, ...	?	Assembly

- 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 $\models$ J
  - Has a derivation $\vdash$ J (= "a proof")
- A common form of typing judgment:

  $\Gamma \vdash e : \tau$ (e is an expression and τ is a type)
- Γ (Gamma) is a set of type assignments for the free variables of e
  - Defined by the grammar $\Gamma ::= \cdot \mid \Gamma, x : \tau$
  - Type assignments for variables not free in e are not relevant
  - e.g, $x : int, y : int \vdash x + y : int$

Typing rules

- Typing rules are used to derive typing judgments
- Examples:

$$\frac{}{\Gamma \vdash 1 : \texttt{int}}$$

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{int}}$$

#13

Typing Derivations

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

$$\frac{\frac{}{x : \texttt{int} \vdash x : \texttt{int}} \quad \frac{\frac{}{x : \texttt{int} \vdash x : \texttt{int}} \quad \frac{}{x : \texttt{int} \vdash 1 : \texttt{int}}}{x : \texttt{int} \vdash x + 1 : \texttt{int}}}{x : \texttt{int} \vdash x + (x + 1) : \texttt{int}}$$

- We say $\Gamma \vdash$ e : τ to mean there exists a derivation of this typing judgment (= “we can prove it”)
- Type checking: given Γ, e and τ find a derivation
- Type inference: given Γ and e, find τ and a derivation

#14

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 $\in \|$ int $\|$ and true $\in \|$ bool $\|$)
- Define what it means for an expression to have a type

  e $\in$ | τ | iff $\forall$v. (e $\Downarrow$ v $\Rightarrow$ v $\in \| \tau \|$)
- Prove type soundness

  If $\cdot \vdash$ e : τ then e $\in$ | τ |

  or equivalently

  If $\cdot \vdash$ e : τ 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 τ)

#15

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

Simply-Typed Lambda Calculus

- Syntax:

Notice :τ

Terms	e ::= x	| λx:τ. e	| e_1 e_2
	| n	| e_1 + e_2	| iszero e
	| true	| false	| not e
	| if e_1 then e_2 else e_3		
Types	τ ::= int | bool | $\tau_1 \to \tau_2$		

- $\tau_1 \to \tau_2$ is the function type
- $\to$ associates to the right
- Arguments have typing annotations :τ
- This language is also called F_1

#17

Static Semantics of F_1

- The typing judgment

$$\Gamma \vdash e : \tau$$

- Some (simpler) typing rules:

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x : \tau . e : \tau \to \tau'}$$

$$\frac{\Gamma \vdash e_1 : \tau_2 \to \tau \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 e_2 : \tau}$$

#18

More Static Semantics of F_1

$$\frac{}{\Gamma \vdash n : \texttt{int}} \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{int}}$$

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

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{bool}} \qquad \frac{\Gamma \vdash e : \texttt{bool}}{\Gamma \vdash \texttt{not}\ e : \texttt{bool}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \quad \Gamma \vdash e_t : \tau \quad \Gamma \vdash e_f : \tau}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_t\ \texttt{else}\ e_f : \tau}$$

Typing Derivation in F_1

- Consider the term

  λx : int. λb : bool. if b then f x else x

  - With the initial typing assignment f : int → Int
  - Where Γ = f : int → int, x : int, b : bool

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash b : \texttt{bool} \qquad \dfrac{\Gamma \vdash f : \texttt{int} \to \texttt{int} \quad \Gamma \vdash x : \texttt{int}}{\Gamma \vdash f\ x : \texttt{int}} \qquad \Gamma \vdash x : \texttt{int}}{f : \texttt{int} \to \texttt{int}, x : \texttt{int}, b : \texttt{bool} \vdash \texttt{if}\ b\ \texttt{then}\ f\ x\ \texttt{else}\ x : \texttt{int}}}{f : \texttt{int} \to \texttt{int}, x : \texttt{int} \vdash \lambda b : \texttt{bool}.\ \texttt{if}\ b\ \texttt{then}\ f\ x\ \texttt{else}\ x : \texttt{bool} \to \texttt{int}}}{f : \texttt{int} \to \texttt{int} \vdash \underbrace{\lambda x : \texttt{int}.\lambda b : \texttt{bool}.\ \texttt{if}\ b\ \texttt{then}\ f\ x\ \texttt{else}\ x} : \texttt{int} \to \texttt{bool} \to \texttt{int}}$$

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

  · ⊢ λx. x : int → int

  · ⊢ λx. x : bool → bool

Operational Semantics of F_1

- Judgment:

  e ⇓ v

- Values:

  v ::= n | true | false | λx:τ. e

- The evaluation rules …
  - Audience participation time: raise your hand and give me an evaluation rule.

Opsem of F_1 (Cont.)

- Call-by-value evaluation rules (sample)

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

$$\frac{}{\lambda x : \tau.e \Downarrow \lambda x : \tau.e}$$

$$\frac{e_1 \Downarrow \lambda x : \tau.e_1' \quad e_2 \Downarrow v_2 \quad [v_2/x]e_1' \Downarrow v}{e_1\ e_2 \Downarrow v}$$

$$\frac{}{n \Downarrow n} \qquad \frac{e_1 \Downarrow n_1 \quad e_2 \Downarrow n_2 \quad n = n_1 + n_2}{e_1 + e_2 \Downarrow n}$$

$$\frac{e_1 \Downarrow \texttt{true} \quad e_t \Downarrow v}{\texttt{if}\ e_1\ \texttt{then}\ e_t\ \texttt{else}\ e_f \Downarrow v}$$

$$\frac{e_1 \Downarrow \texttt{false} \quad e_f \Downarrow v}{\texttt{if}\ e_1\ \texttt{then}\ e_t\ \texttt{else}\ e_f \Downarrow v}$$

Evaluation is undefined for ill-typed programs !

Type Soundness for F_1

- Theorem: **If · ⊢ e : τ and e ⇓ v then · ⊢ v : τ**
  - 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.