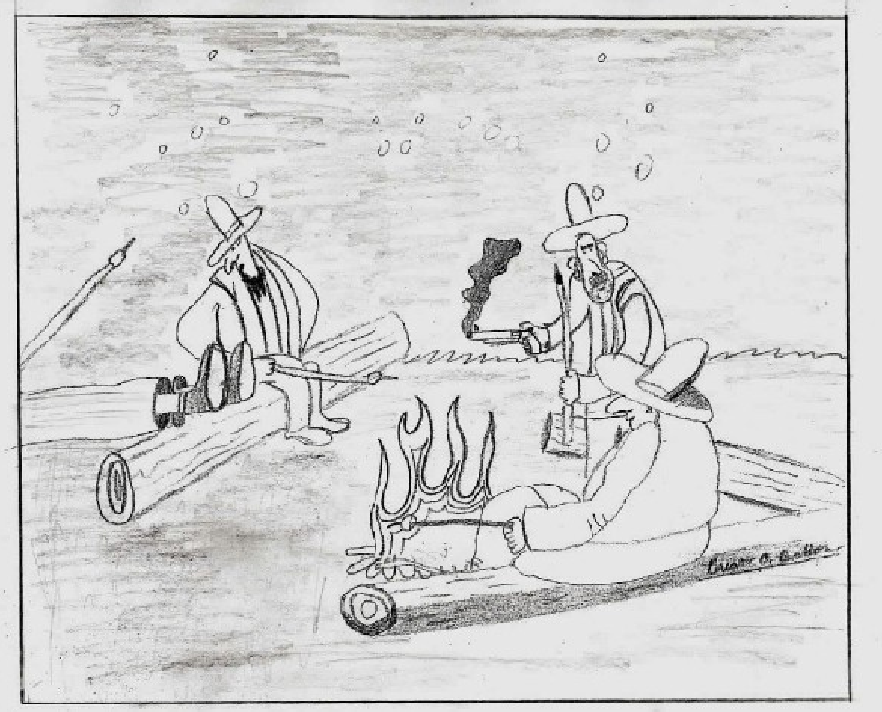
Simply-Typed Lambda Calculus



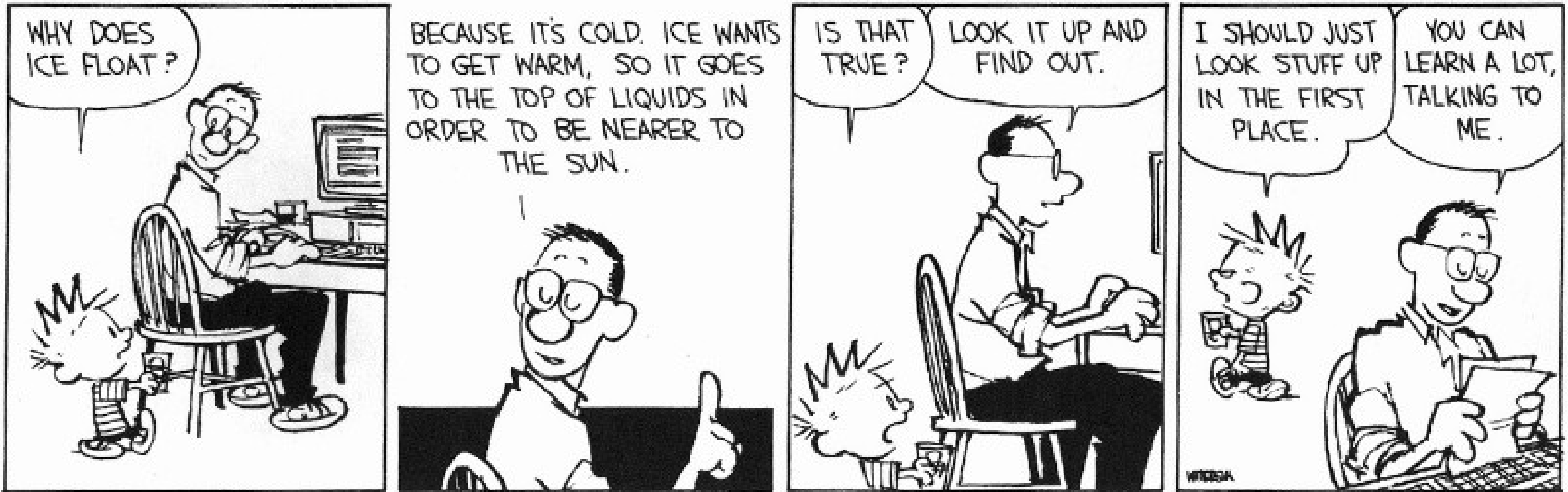
You guys are both my witnesses... He insinuated that ZFC set theory is superior to Type Theory!

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.

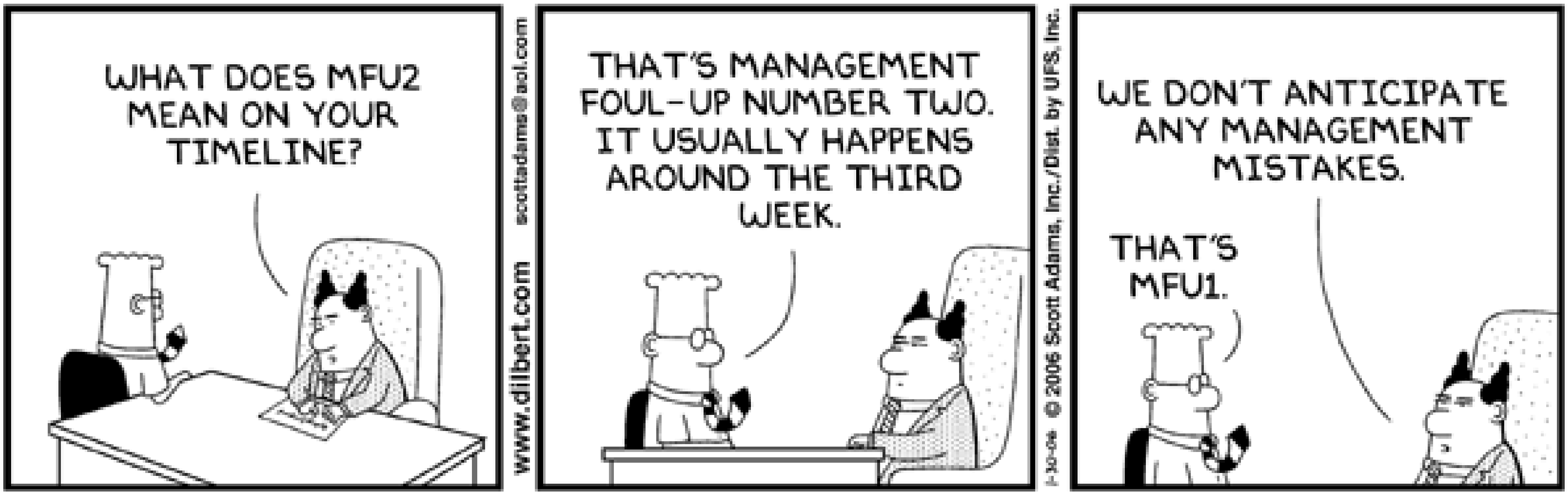
Review!

- 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



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 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)

	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 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 $\models J$
  - 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., **$x : \text{int}, y : \text{int} \vdash x + y : \text{int}$**

Typing rules

- Typing rules are used to derive typing judgments

- Examples:

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

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

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

Typing Derivations

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

$$\dfrac{\dfrac{}{x : \mathtt{int} \vdash x : \mathtt{int}} \qquad \dfrac{\dfrac{}{x : \mathtt{int} \vdash x : \mathtt{int}} \quad \dfrac{}{x : \mathtt{int} \vdash 1 : \mathtt{int}}}{x : \mathtt{int} \vdash x + 1 : \mathtt{int}}}{x : \mathtt{int} \vdash x + (x + 1) : \mathtt{int}}$$

- We say $\Gamma \vdash e : \tau$ 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

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

$$e \in | \tau | \quad \textbf{iff} \quad \forall v. (e \Downarrow v \Rightarrow v \in \| \tau \|)$$

- Prove type soundness

$$\text{If } \cdot \vdash e : \tau \qquad \text{then } e \in | \tau |$$

or equivalently

$$\text{If } \cdot \vdash e : \tau \text{ and } e \Downarrow v \qquad \text{then } v \in \| \tau \|$$

- This implies safe execution (since the result of a unsafe execution is not in $\| \tau \|$ for any τ)

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

Q: Movies (378 / 842)

- 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.

Q: Games (504 / 842)

- This 1985 falling-blocks computer game was invented by Alexey Pajitnov (Алексей Пажитнов) and inspired by pentominoes.

Simply-Typed Lambda Calculus

Notice $:\tau$

- 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

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

- Some (simpler) typing rules:

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

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

More Static Semantics of F_1

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

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

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

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

Typing Derivation in F_1

- Consider the term (also underlined below)

  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

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

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 <u>no unique type</u>

$$\cdot \vdash \lambda x.\ x : \text{int} \to \text{int}$$

$$\cdot \vdash \lambda x.\ x : \text{bool} \to \text{bool}$$

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)

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

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

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

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

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

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

Evaluation is undefined for ill-typed programs !

Type Soundness for F_1

- Thm: **If $\cdot \vdash e : \tau$ and $e \Downarrow v$ then $\cdot \vdash v : \tau$**
  - 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