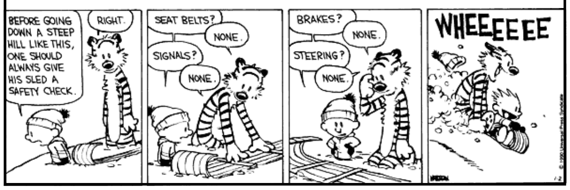
Simply-Typed Lambda Calculus

Meeting 18, CSCI 5535, Spring 2009



Announcements

- I have commented on most of your proposals

2

Quick Review

- Tell me about λ-calculus

• Evaluation strategies
  - some guarantee finding normal forms

- Functions!
- "β-reduction" – function application
- binding – scope of variables
- combinators: closed term (no free vars)
- renaming of bound variables = α-renaming (α-equivalence)
  → avoid: de Bruijn notation
- normal form: some term that can't be reduced further
- I = λx.x

3

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

4

Today's Plan

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

5

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

6

Typed and Untyped Languages

- Untyped languages
  - e.g., assembly language
  - "assign anything to anything"
  - "meaningfulness of the program is up to programmer"
- (Statically) Typed languages
  - Find type errors at compile time

7

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

8

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

9

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)
  - help with generic programming, probabilistic languages, …
  - … are often combined with dynamic analyses (e.g., CCured)

10

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

11

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)

12

Why Typed Languages?

- Performance
  - → Know more information
  - → e.g. int vs. float
  - → don't need to check at run-time
- Development
  - → catch errors early
  - → documentation
  - → helps w/ refactoring

13

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

14

Why Not Typed Languages?

- Flexible
  - "Types hamper your style"
  - Types limit the programs you write
- Want more flexibility
  - ⇒ complicated type systems

15

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)

16

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	Java, ML	Python, PERL, Ruby, Basic, Lisp	λ-calc
Unsafe	C, C++, C#	?	assembly

17

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

18

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

19

Why Formal Type Systems?

- Many typed languages have informal descriptions of the type systems (e.g., in language reference manuals)

20

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

21

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

22

Typing Judgments

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

23

Typing Judgments

- Recall: judgment
  - A statement J about certain formal entities
- 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, $x : \mathbf{int}, y : \mathbf{int} \vdash x + y : \mathbf{int}$

24

Typing rules

- Typing rules are used to derive typing judgments

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

- Examples:

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

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

25

Typing Derivations

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

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

- $\Gamma \vdash e : \tau$ means 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

26

Proving Type Soundness: Intuition

- A typing judgment
- Define what it means for a value to have a type
  v ∈ ∥ τ ∥
  (e.g. 5 ∈ ∥ int ∥ and true ∈ ∥ bool ∥)
- Define what it means for an expression to have a type
  e ∈ | τ | iff ∀v. (e ⇓ v ⇒ v ∈ ∥ τ ∥)
- Prove type soundness
  If · ⊢ e : τ then e ∈ | τ |
  or equivalently
  If · ⊢ e : τ and e ⇓ v then v ∈ ∥ τ ∥
- This implies safe execution (since the result of a unsafe execution is not in ∥ τ ∥ for any τ)

27

Simply-Typed Lambda Calculus

- Syntax: (Notice the :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
- This language is also called F_1

28

Static Semantics of F_1

- Function rules

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

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

29

Static Semantics of F_1

- Function rules

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

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

30

More Static Semantics of F_1

- Base type rules

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

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

$$\frac{\Gamma \vdash e_1 : \mathsf{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \tau}$$

31

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

- Write the type derivation

32

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

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

33