Types

Lecture 17
CS 169

Announcements

- Working on requirements this week...
- Work on design, implementation...

Outline

- Type concepts
  - Where do types come from and why?
- Introduction to type checking
- Then:
  - Detecting races using type checking
  - Are type casts a necessary evil?

Java Types

- The types are just class names
  - And a few types that aren't classes
    - Arrays, interfaces, basic types (int, bool, etc.)
- The user declares types for identifiers
- The compiler infers types for expressions
  - Infers a type for every expression

C Types

- C Types have structure
- Base types
  - int, char, float
- Compound types
  - \( T ::= \text{Array}[T] | \text{struct}([L: T, ...], L: T) | \text{pointer}(T) | ... \)
- Ugly C syntax: int[], const char *, ...

What Are Types?

- Types are specifications
- Explains the intended interpretation of a bit pattern
  - Int
    - It's a 32-bit integer
  - Char
    - It's an ASCII character
  - The same bit pattern can be interpreted either way
    - Without types we don't know what the programmer wanted
- Misuse of types can be detected
  - Stops you from making many simple mistakes
Analogy: Plugs and Sockets

- Lots of different kinds of electrical plugs/sockets
- Evolved to prevent electrocution/fires from mismatched electrical components
- Can’t plug a 9V wire into a 220V outlet
- No guarantee the whole system works, but at least gross power errors are prevented

Why Do We Need Type Systems?

Consider the assembly language fragment

\[ \text{addi } r1, r2, r3 \]

What are the types of \( r1, r2, r3 \)?

Types and Operations

- Certain operations are legal for values of each type
  - It does make sense to add two integers
  - It doesn’t make sense to add a function pointer and an integer in C
  - But both have the same assembly language implementation!

Type Systems

- A language’s type system specifies which operations are valid for which types
- The goal of type checking is to ensure that operations are used with the correct types
  - Enforces intended interpretation of values
- Type systems provide a concise formalization of the semantic checking rules

What Can Types do For Us?

- Can detect certain kinds of errors
- Memory errors:
  - Reading from an invalid pointer, etc.
- Violation of abstraction boundaries:

  ```
  class FileSystem {
      open(x : String) : File {
          ...
      }
  }

  class Client {
      bar(fs : FileSystem) {
          File f <- fs.open("foo")
          ...
      } // Client cannot see inside f!
      ...
  }
  ```

Type Checking Overview

- Three kinds of languages:
  - Untyped: No type checking (machine code)
  - Dynamically typed: Almost all checking of types is done as part of program execution (Scheme)
  - Statically typed: All or almost all checking of types is done as part of compilation (C, Java)
Untyped Languages

• Some languages have no type system
• Bit patterns have no interpretation except in the mind of the programmer
• Assembly code
  - But even here there is usually an int/float distinction

Dynamically Typed Languages

• Type checking at runtime is simple
• Just check that operators are applied to arguments of the expected type
  - + checks that its arguments are numbers
  - reverse checks that its argument is a list

Dynamic Typing: Implications

• This has performance implications
  - All data must carry type tags at runtime
  - All operations must check the tags
  - Generally a 25% performance overhead
• This has correctness implications
  - Type errors are detected late: during execution
  - Errors arise in awkward places
    - Example: add(a,b) = a + b ...
    - Cannot catch bad data at function interface
    - Error will arise deep within the logic of the function body

Dynamic Typing: Benefits

• Some programming styles are much easier in a dynamically typed language
• Best example: eval
  - Read a String s from the network
  - Call eval(s)
    - eval interprets the string as a program and runs it
    - Expressive, and dangerous!
• No good way to do this in a statically typed language
  - Except if checking types at load-time (e.g., class loading)

The Type Wars

• Competing views on static vs. dynamic typing
• Dynamic typing proponents say:
  - Static type systems are restrictive
  - Rapid prototyping easier in a dynamic type system
• Static typing proponents say:
  - Static checking catches many programming errors at compile time
  - Avoids overhead of runtime type checks

Static Typing

• Check types at compile time
  - One statement/function/module at a time
  - Compiler guarantees types will never be violated during program execution
• Can find bugs even before we run code
• Most common today: C, C++, Java
Expressions and Statements

- Consider the following language of expressions and statements (like in C or Java):

  **expressions**
  
  \[ E ::= n \mid x \mid false \mid E_1 + E_2 \mid not \ E_1 \]

  **types**
  
  \[ T ::= int \mid bool \]

  **statements**
  
  \[ S ::= x := E \mid while(E)\ S \mid \{S_1; \ldots; S_n\} \mid \{T\ x; S\} \mid \text{if}(E)\ S_1\ \text{else}\ S_2 \]

Type Checking and Type Inference

- Type checker can be written as a recursive function that scans the program text
- Type checking: \( \text{check}(E, T) \)
  - Return "true" if \( E \) has type \( T \), and error if \( E \) is ill-typed or has some other type
- Type inference: \( \text{infer}(E) \)
  - Returns the type of \( E \), or error if \( E \) is ill-typed
  - The actual workhorse
  
  \[ \text{check}(E, T) = \text{if } \text{infer}(E) == T \text{ then true else error} \]

Rules of Inference

- Type inference is easy for constants
  
  \[ \text{infer}(n) = \text{int} \]
  \[ \text{infer}(\text{false}) = \text{bool} \]

- Type inference for composed constructs is recursive
  
  \[ \text{infer}(E_1 + E_2) = \text{int} \]
  \[ \text{if } \text{check}(V, E_1, \text{int}) \text{ and } \text{check}(V, E_2, \text{int}) \]
  \[ \text{infer}(\text{not } E_1) = \text{bool} \]
  \[ \text{if } \text{check}(V, E_1, \text{bool}) \]

Type Inference for Variables

- How can we infer the type of a variable?
  
  \[ \text{infer}(x) = ? \]

  - Idea: we can keep track of declarations as we scan the program
    - Use a second argument to infer and check
    - \( V \) is a set of pairs of variables and their types
    - A variable environment
    
    \[ \text{Infer}(V, E) = T \]
    
    - Means \( E \) has type \( T \), assuming variables are declared as specified by \( V \)

Typing for Expressions. Revisited.

- \( \text{infer}(V, n) = \text{int} \)
- \( \text{infer}(V, \text{false}) = \text{bool} \)
- \( \text{infer}(V, E_1 + E_2) = \text{int} \)
  
  - if \( \text{check}(V, E_1, \text{int}) \text{ and } \text{check}(V, E_2, \text{int}) \)
- \( \text{infer}(V, \text{not } E) = \text{bool} \)
  
  - if \( \text{check}(V, E, \text{bool}) \)
- \( \text{infer}(V, x) = T \)
  
  - if \( (x, T) \in V \)
- \( \text{infer}(V, E) = \text{error otherwise} ! \)

Examples of Ill-Typed Expressions

The following invocations result in error. Why?

- \( \text{infer}(V, \text{false} + 1) \)
- \( \text{infer}(V, \text{not} 1) \)
- \( \text{infer}(\{(x, \text{bool})\}, x + 1) \)
- \( \text{infer}(\{(y, \text{int})\}, x + 1) \)
Typing for Statements

- We define a `scheck(V, S)` function:
  
  \[
  \text{scheck}(V, x := E) = \\
  \text{if } (x, T) \in V \land \text{check}(V, E, T) \\
  \text{scheck}(V, \text{if}(E) S_1 \text{ else } S_2) = \\
  \text{if } \text{check}(V, E, \text{bool}) \land \text{scheck}(V, S_1) \land \text{scheck}(V, S_2) \\
  \text{scheck}(V, \text{while}(E) S) = \\
  \text{if } \text{check}(V, E, \text{bool}) \land \text{scheck}(V, S) \\
  \text{scheck}(V, \{ S_1; \ldots; S_n \}) = \\
  \text{if } \text{scheck}(V, S_i) \text{ for } i = 1, \ldots, n
  \]

Local Variable Declarations

- Our language has static scope
  - A variable name refers to the closest enclosing variable declaration

  **Example:**
  
  \[
  \{ \text{int } x; \\
  x = 1; \\
  \{ \text{bool } x; x := \text{false}; \\
  x = x + 1 \}
  \}
  \]
  - Type checker must keep track of scope

Local Variable Declarations

\[
\text{scheck}(V, \{ T x; S \}) = \\
\text{if } \text{scheck}(V_1, S), \text{ where } V_1 = V - (x, \ldots) + (x, T) \\
\text{We write simply } V_1 = V + (x, T) \\
\text{We check } S \text{ with a modified set of variables} \\
\text{We replace the old declaration of } x \\
\text{Or just add a declaration if none was present}
\]

- Think how `scheck` works on previous example

Type Checking Example

\[
\begin{align*}
\text{Code} & : & \text{scheck}(V, (T_0 x; \}) \\
\text{Type env.} & : & \text{scheck}(V+(x,T_0)), \\
\text{Types} & : & \text{scheck}(V+(x,T_0), (T_1 y; x = \}) \text{scheck}(V+(x,T_0), (T_2 x; \}) \\
\text{infer}(V+(x,T_0)+(y,T_1), E_{x,y} = T_0) & : & \text{scheck}(V+(x,T_0), (T_1 y; y = \}) \text{infer}(V+(x,T_0)+(y,T_1), F_{x,y} = \text{int})
\end{align*}
\]

Notes

- The type environment gives types to the identifiers in scope
- The type environment is passed down the expression from the root towards the leaves
- Types are computed up the expression from the leaves towards the root

Know Your Type System: Const

- You have to really learn the language to know how to use the type system
  - What you don’t know can’t help you
- In C, learn to use `const`
  - Data declared `const` is read-only
  - Very useful for documenting and checking update policies in APIs
  - Not as widely used as it should be
Static Type Systems Summary

- General themes
  - Type rules are defined on the structure of expressions
  - One rule for each kind of expression
  - Types of variables are modeled by an environment
- Most rules are painfully obvious
  - It is the systematic use of the rules on large programs that is beneficial
- Type checking is conservative

Type Systems for the Software Engineer

- Types are one of the most useful tools available to programmers
- If you don’t use them, they won’t help you
- Helps to know a bit about how they work!
- Can use typecking-like techniques for checking other useful properties

Summary

- Types are the most successful specifications we have
  - Formal
  - Part of the code
  - Automatically checked
- Use your type system!
  - Spend the time to really learn the language
  - It can help you
- Choice of statically or dynamically typed languages
  - Static typing preferred
  - Some problems more dynamic than others