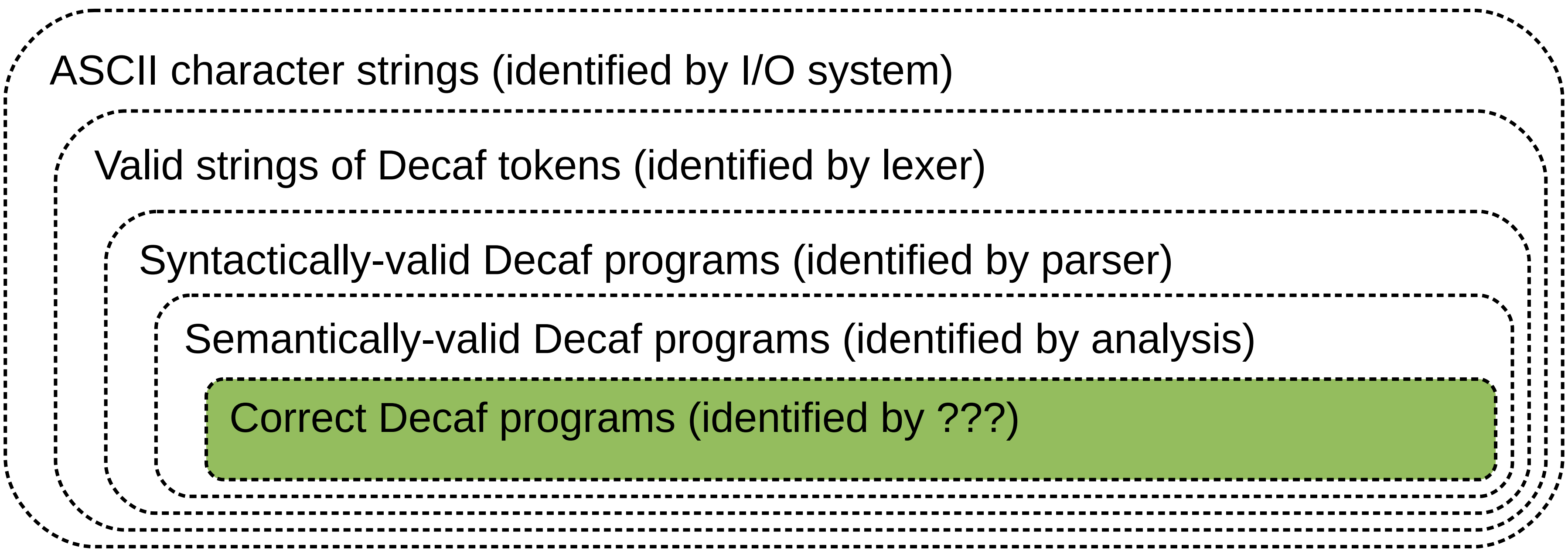
CS 480
Fall 2015

Mike Lam, Professor

Static Analysis

Overview

- Syntax: *form* of a program
  - Described using regular expressions and context-free grammars
- Semantics: *meaning* of a program
  - Much more difficult to describe clearly



Operational Semantics

- Describe a program's effects using a simpler language that is closer to the hardware

```
for (i=0; i<n; i++) {
  m *= i;
}
```

```
        i=0;
loop:   if i<n goto done
        m *= i
        i++
        goto loop
done:
```

```
for (e1; e2; e3) {
  e4
}
```

```
        e1
loop:   if e2 goto done
        e4
        e3
        goto loop
done:
```

Axiomatic Semantics

- Express programs as proof trees
  - Loops can be difficult to handle

$$\frac{\{P \wedge e1\}\ e2\ \{Q\} \qquad \{P \wedge \neg e1\}\ e3\ \{Q\}}{\{P\}\ \textbf{if}\ e1\ \textbf{then}\ e2\ \textbf{else}\ e3\ \{Q\}}\ \text{SConditional}$$

$$\frac{\dfrac{\ldots}{\{x=10 \wedge x>5\}\ y:=3\ \{x=10 \wedge y=3\}}\ \text{SAssign}}{\{x=10\}\ \textbf{if}\ x > 5\ \textbf{then}\ y := 3\ \textbf{else}\ y := 7\ \{x=10 \wedge y=3\}}\ \text{SConditional}$$

Denotational Semantics

- Describes a program's results using functions
  - Must also track system state

```
eval :: (Program, State) → (Value, State)

eval(e1 + e2, S) =
    let (v1, S')  = eval(e1, S)  in
    let (v2, S'') = eval(e2, S') in
    (v1 + v2, S'')

eval(while e1 do e2, S) =
    let (v, S') = eval(e1, S) in
    if not v then
        (v, S')
    else let (_, S'') = eval(e2, S')
        eval(while e1 do e2, S'')
```

Semantics

- Three main approaches:
  - *Operational* semantics: programs are actions
  - *Axiomatic* semantics: programs are proofs
  - *Denotational* semantics: programs are functions

Semantics

- Goal: reject incorrect programs
- Problem: checking the semantics of a program is hard!
  - In general, we won't be able to check for full correctness
  - However, it turns out that some aspects of semantics can be robustly encoded using types and type systems

Type Systems

- A **type** is an abstract category that characterizes a range of data values
  - Base types: integer, character, boolean, floating-point
  - Enumerated types (finite list of constants)
  - Pointer types (“address of X”)
  - Array or list types (“list of X”)
  - Compound/record types (named collections of other types)
  - Function types: (type1, type2, type3) $\rightarrow$ type4
- Two types are **name-equivalent** if their names are identical
- Two types are **structurally-equivalent** if
  - They are the same basic type or
  - They are recursively structurally-equivalent

Type Systems

- A **type system** is a set of type rules
  - Rules about valid types, type compatibility, and how data values can be used
  - The system is “strongly typed” if every expression can be assigned an unambiguous type
  - The system is “statically typed” if all types can be assigned at compile time
  - The system is “dynamically typed” if some types can only be discovered at runtime
- Benefits of a robust type system
  - Earlier error detection
  - Better documentation
  - Increased modularization

Type Checking

- **Type inference** is the process of assigning types to expressions
  - This information must be “inferred” if it is not explicit
- **Type checking** is the process of ensuring that a program has no type-related errors
  - Ensure that operations are supported by a variable's type
  - Ensure that operands are of compatible types
  - This could happen at compile time (for static type systems) or at run time (for dynamic type systems)
  - A type error is usually considered a bug

Type Checking

- **Sound** vs. **complete** type checking
  - A “sound” system has no false positives
    - All errors reported are true errors
  - A “complete” system has no false negatives
    - All true errors are reported
- Most type checking is sound but not complete
  - The lack of type errors does not mean the program is correct
  - However, the presence of a type error generally does mean that the program is NOT correct

Type Conversions

- Implicit vs. explicit
  - **Implicit** conversions are performed automatically by the compiler ("coercions")
    - E.g., double x = 2;
  - **Explicit** conversions are specified by the programmer ("casts")
    - E.g., int x = (int)1.5;
- Narrowing vs. widening
  - **Widening** conversions preserve information
    - E.g., int → long
  - **Narrowing** conversions may lose information
    - E.g., float → int

Polymorphism

- **Polymorphism**: literally “taking many forms”
  - A polymorphic construct supports multiple types
  - Subtype polymorphism: object inheritance
  - Function polymorphism: overloading
  - Parametric polymorphism: generic type identifiers
    - E.g., templates in C++ or generics in Java
  - During type inference, create type variables, and unify type variables with concrete types
    - Some type variables might remain unbound
    - E.g., `map : ((a → b), [a] ) → [b]`

Symbols

- A **symbol** is a single name in a program
  - What type of value is it?
  - If it is a variable:
    - How big is it?
    - Where is it stored?
    - How long must its value be preserved?
    - Who is responsible for allocating, initializing, and de-allocating it?
  - If it is a function:
    - What parameters does it take?
    - What does it return?

Symbol Tables

- A **symbol table** stores information about symbols during compilation
  - Aggregates information from (potentially) distant parts of code
  - Maps symbol names to symbol information
  - Often implemented using hash tables
  - Usually one symbol table per scope
    - Each table contains a pointer to its parent (next larger scope)
- Supported operations
  - Insert(name, record) – add a new symbol to the current table
  - LookUp(name) – retrieve information about a symbol

Building Symbol Tables

- Walk the AST
  - Create new symbol table for each scope
    - Global, function, block, etc.
  - Add all symbol information
    - Variable declarations, function parameters, etc.

Type Checking

- Walk the AST
  - Calculate the types of all expressions
    - Using symbol table lookups
    - May require some inference
  - Verify that all types are correct according to type rules
    - May require additional lookups

Formal Type Theory

- Type systems expressed as a set of type rules
  - Each rule has zero or more premises and a conclusion
  - Apply rules recursively to form proof trees
  - Curry-Howard correspondence ("proofs as programs")
  - Can be applied to *typed* lambda calculus

$$\text{TInt}\quad \frac{}{A \vdash n : \mathrm{int}} \qquad\qquad \frac{x : t \in A}{A \vdash x : t}\quad \text{TVar}$$

$$\text{TFun}\quad \frac{A, x : t \vdash e : t'}{A \vdash \lambda x{:}t.e : t \to t'} \qquad\qquad \frac{A \vdash e : t \to t' \qquad A \vdash e' : t}{A \vdash e\ e' : t'}\quad \text{TApp}$$

Formal Type Theory

$$\frac{}{A \vdash n : \color{blue}{int}}\ \textbf{TInt} \qquad \frac{x : \color{blue}{t} \in A}{A \vdash x : \color{blue}{t}}\ \textbf{TVar} \qquad \frac{A, x : \color{blue}{t} \vdash e : \color{blue}{t'}}{A \vdash \lambda x{:}\color{blue}{t}.e : \color{blue}{t \to t'}}\ \textbf{TFun} \qquad \frac{A \vdash e : \color{blue}{t \to t'} \quad A \vdash e' : \color{blue}{t}}{A \vdash e\ e' : \color{blue}{t'}}\ \textbf{TApp}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{+ : \qquad \in B}{B \vdash + :}\,\text{TVar} \quad \dfrac{x : \qquad \in B}{B \vdash x :}\,\text{TVar}}{B \vdash + x :}\,\text{TApp} \qquad B \vdash 3 :}{B \vdash + x\ 3 :}\,\text{TApp}}{A \vdash (\lambda x{:}\color{blue}{int}.+ x\ 3) :}\,\text{TFun} \qquad A \vdash 4 :}{A \vdash (\lambda x{:}\color{blue}{int}.+ x\ 3)\ 4 :}\,\text{TApp}$$

$A = \{\ + : \color{blue}{int \to int \to int}\ \}$ $\qquad$ $B = A, x : \color{blue}{int}$

Formal Type Theory

$$\dfrac{}{A \vdash n : int}\;\textbf{TInt} \qquad \dfrac{x : t \in A}{A \vdash x : t}\;\textbf{TVar} \qquad \dfrac{A, x : t \vdash e : t'}{A \vdash \lambda x{:}t.e : t \to t'}\;\textbf{TFun} \qquad \dfrac{A \vdash e : t \to t' \qquad A \vdash e' : t}{A \vdash e\,e' : t'}\;\textbf{TApp}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{+ : i \to i \to i \in B}{B \vdash + : i \to i \to i}\;\text{TVar} \qquad \dfrac{x : int \in B}{B \vdash x : int}\;\text{TVar}}{B \vdash +\,x : int \to int}\;\text{TApp} \qquad B \vdash 3 : int}{B \vdash +\,x\,3 : int}\;\text{TApp}}{A \vdash (\lambda x{:}int.+\,x\,3) : int \to int}\;\text{TFun} \qquad A \vdash 4 : int}{A \vdash (\lambda x{:}int.+\,x\,3)\;4 : int}\;\text{TApp}$$

$A = \{\, + : int \to int \to int \,\}$ $\qquad$ $B = A, x : int$