COMP 520 - Compilers

Lecture 11 (Tue Feb 21, 2017)

Contextual analysis: Type Checking

• Please pick up from the back of the class
  – WA3 graded papers

• Reading
  – (Thu 2/23) Contextual Analysis (secn 5.3)
Topics

• Type checking
  – examples of type checking
  – role of types in programming languages
  – structural vs. name equivalence

• A general framework for type checking
  – definitions
    • type synthesis
    • type constraints
  – examples
Type analysis

- Basic examples
  - assignment statements
    - do target and value type agree? \( x = 1 + 2; \)
  - Expressions
    - what is the type of the result? \( X + 3 \neq 4 \)
    - What are the types of the intermediate expressions?
- function/procedure calls
  - do arguments types agree with parameter types?
  - does a function return a result of the appropriate type?
- type definitions and variable declarations
  - is the type well-formed?
    - does a class type refer to an identified class?
    - void []
- Systematically answering such questions is called “type checking”
Type analysis

Where do we need to use type analysis

- automatic conversions/coercions
  - convert byte or short to int or long
  - convert byte, short, int, long to float or double
  - automatic boxing/unboxing of int to/from Integer in Java

- overload resolution
  - which definition of “+” should be used?

- inheritance
  - which methods are available on an object?
  - can the invocation of an overridden method be statically determined?

- type inference
  - variables or parameters without type declarations (e.g. perl)
  - can a type be inferred for the missing declaration?
Types in modern programming languages

- **What is a type?**
  - a set of possible values (and their representation)
  - a set of permissible operations

- **Purpose of types**
  - safety and correctness
    - apply only permissible operations on values with correct representation
  - improve readability and comprehensibility
  - provide consistency checks on programs
  - provide information to improve efficiency of execution
    - eliminate run-time type checks
    - efficient space (re)use
Type safety

- Type safety is also known as “strong typing”
  - all operations applied to values with a known representation
    - pointer dereference can not be applied to arbitrary integers
    - arithmetic operations are applied to values of correct representation
    - appropriate methods are applied to objects
  - strongly typed languages guarantee to detect any situation where this is not the case
    - Java, Triangle, Ada
  - Weakly typed languages may not detect such situations
    - C (old C, not C99 and later)
When does type checking take place?

- **Compile time**
  - statically typed
    - Java, Triangle, C++, Fortran, Haskell, ...

- **Run-time**
  - dynamically typed
    - JavaScript, Perl, Python, PHP, Ruby, ...
    - Java casts

- **Never**
  - untyped
    - Assembler (but even this is changing towards strong typing)
Type wars

• Static vs. dynamic typing
  – static typing
    • catches many common programming errors at compile time
    • avoids run-time overhead of dynamic typing
  – dynamic typing
    • static type systems are restrictive
    • type declarations are wordy and slow the programmer down

• In practice
  – static type systems are restrictive so an escape system is added
    • e.g. C casts (void *) defeat typing
    • unclear whether this is the best or worst of the two worlds
  – static type systems are getting better
    • overloading, generics, type inference, virtual method invocation
    • dynamic typing used where static typing is too restrictive
      – “casts” with type checks and conversions
Type agreement

• In many modern languages we can define named types
  
  ```
  type Height = Integer
  type Weight = Integer
  var h : Height, w : Weight
  ...
  h := 130;  w := 150;  h := h + w ...
  ```

  is this OK?

• When are two types equivalent?
  – Structural equivalence
    • when they are the same following substitution of type definitions
      – example languages: C, Triangle
  – Name equivalence
    • only when they are the same named type
      – example languages: Ada, Pascal, (C++), (Java)

• The form of type equivalence has fundamental bearing on type checking
miniJava type checking

• Fairly simple – bottom up
  – leaves of the AST are Terminals: Identifiers, Literals, and Operators
    • We can assign each of these a specific TypeDenoter (BaseType, ClassType, or ArrayType)
      – The specific types are manifest (Literals) or extracted from the declaration of an Identifier

  – Expression, Reference, and Declaration nodes compute their type from their children

  – specific Statement nodes make some checks for type agreement
    • AssignStmt
    • IfStmt

  – special types
    • ERROR, UNSUPPORTED
Simple approach to type checking

• Define a set of possible types
  – set of base types and some ways to build new types

• Define a representation of programs
  – simple class of ASTs

• Define a type-assignment algorithm that
  – labels all nodes of an AST with zero or more types
  – handles many forms of overloading
    • essentially all languages have some form of overloading
      – addition: operation on integers or floats?

• Type checking
  – following type assignment each AST node is labeled with a set of types
    • program is type correct if all nodes have a single type
    • program contains type error(s) if some node has no type assignment or more than one possible type assignment
Characterization of a set of types

- Type values constructed from
  - basic types
    - Int, Real, Bool, ...
  - parameterized types
    (in the following, a *type variable* ($\alpha$, $\beta$, ...) stands for any type)
    - tuple types
      $\alpha_1 \times \ldots \times \alpha_n$
    - function types
      $\alpha \rightarrow \beta$
    - array types
      Array($\alpha$)
  - named types
    - for name equivalence, if needed
      Complex = Real $\times$ Real
Characterization of a simple class of ASTs

- **AST structure**
  - **Leaves:** two kinds
    - **constants**
    - **identifiers (applied occurrences)**
      - denoting variables or functions (including operators)
  - **interior nodes:** two kinds
    - **tuple constructor**
    - **function application**

- **Example**
  - Concrete syntax: \( a + 10 \)
  - AST:
Type values at leaves

- Declarations provide type value(s) for AST leaves
  - a variable type is obtained from its (unique) declaration
    a: Int
  
  - constants have a manifest (unique) type
    10: Int
    5.3: Real
    true: Bool

  - functions or operators may have multiple types as a result of overloading
    +: Int × Int → Int
    +: Real × Real → Real

- The declarations are external to our simple ASTs
Generate possible type assignments

- Step 1: generate possible type assignments $\tau(v)$ for each node $v$ by bottom-up traversal of AST
  - $v$ is a leaf of the AST
    - $\tau(v)$ = set of types associated with $v$
  - $v$ is a tuple constructor $(v_1, \ldots, v_k)$
    - $\tau(v) = \{ t_1 \times \ldots \times t_k | t_1 \in \tau(v_1), \ldots, t_k \in \tau(v_k) \}$
  - $v$ is function application $f(a)$
    - $\tau(v) = \{ r | (d \rightarrow r) \in \tau(f) \text{ and } d \in \tau(a) \}$
Constrain type assignments

- Step 2: constrain type assignments $\sigma(v) \subseteq \tau(v)$ for each node $v$ by top-down traversal of AST
  - $v$ is root
    - $\sigma(v) = \tau(v)$
  - $v$ is function application $f(a)$
    - $\sigma(f) = \{ d \rightarrow r | (d \rightarrow r) \in \tau(f) \text{ and } d \in \tau(a) \text{ and } r \in \sigma(v) \}$
    - $\sigma(a) = \{ d | (d \rightarrow r) \in \tau(f) \text{ and } d \in \tau(a) \text{ and } r \in \sigma(v) \}$
  - $v$ is tuple constructor $(v_1, \ldots, v_k)$
    - $\sigma(v_i) = \{ t_i | t_1 \times \ldots \times t_i \times \ldots \times t_k \in \sigma(v) \}$
Type checking

- Type checking of an AST is successful if and only if $|\sigma(v)| = 1$ for every $v$ in the AST
  - ex: $a + 10$
    $\tau(v)$ is shown as $\{ ... \}$
    $\sigma(v) \subseteq \tau(v)$ is shown by underlining elements of $\tau(v)$
  - type checking is successful
  - overloading is resolved

```
+ (
  a { Int } 10 { Int }

{ Int × Int → Int,
 Real × Real → Real }
```
More examples

• Declarations

\[ +: \text{Real} \times \text{Real} \rightarrow \text{Real} \]
\[ +: \text{Complex} \times \text{Complex} \rightarrow \text{Complex} \]
\[ +: \text{Real} \times \text{Real} \rightarrow \text{Complex} \]
\[ =: \text{Real} \times \text{Real} \rightarrow \text{Bool} \]
\[ =: \text{Complex} \times \text{Complex} \rightarrow \text{Bool} \]

r: Real

c: Complex

• Examples

\[ r + r = r \]
\[ r = c \]
\[ (r + r) = (r + r) \]
Extensions

- **Parametric polymorphism (generic types)**
  - parameterized types that include type variables that vary over all types
    
    index: $\text{Array}(\alpha) \times \text{Int} \rightarrow \alpha$
    
    $=\alpha \times \alpha \rightarrow \text{Bool}$
  - substitute type variables in generate and constrain phases
  - ex
    - $a: \text{Array}(\text{Real})$, $i: \text{Int}$
    - type assignment for $a[i]$?
Commands

• Include commands in AST with a new type Stmt
  – parametric polymorphism: type variables $\alpha$ vary over all types
    ifCmd: $\text{Bool} \times \text{Stmt} \times \text{Stmt} \rightarrow \text{Stmt}$
    assignCmd : $\alpha \times \alpha \rightarrow \text{Stmt}$
    sequenceCmd : $\text{Stmt} \times \text{Stmt} \rightarrow \text{Stmt}$
  – ex
    • $x: \text{Int}$
    • type assignment for $x := 3; x := 4$ ?
Type inference

- No types declared for variables – types must be inferred
  - a type variable $\alpha_x$ is used to describe the type of each occurrence of program variable $x$
  - equality and membership become equations rather than true/false propositions (solved using resolution theorem proving)
    - types are inferred if there exists a unique solution for type equations at end of constrain phase
- found in various languages including Haskell
- Example
  What is the type assignment for $a$, $b$ and $i$
  $a[i] := b[i+1] * 5.5$
  Given only the types for the operators (+, *, :=, and indexing) as defined in these slides