COMP 181
Type checking
October 21, 2009

Agenda
- Next week
  - OOPSLA: Object-oriented Programming Systems Languages and Applications
  - One of the top PL conferences
- Monday (Oct 26th)
  - In-class midterm
  - Review topics today
- Wednesday (Oct 28th)
  - Probably no class – maybe new programming assignment
  - Midterm reviews
    - https://www.eecs.tufts.edu/cscourse/feedback.php

Midterm topics
- Lexical analysis, scanning
  - Regular expressions
  - DFAs and NFAs
  - Conversion from RE to NFA to DFA
- Parsing
  - Grammars, precedence and associativity
  - Top-down parsers, LL grammars, First and Follow sets
  - Bottom-up parsers, LR parsing, shifting and reducing
  - LR items, viable prefix DFA, closure operation
  - Shift/reduce and reduce/reduce conflicts
- Static checks
  - Uniqueness checks
  - Symbol tables
  - Types

Today: type checking
- Big topic
  - Type systems
  - Type inference
  - Non-standard type systems for program analysis
  - Theory of type systems
- Focus
  - Role of types in compilation
  - Imperative and object-oriented languages
- What is a type?
  - Def: A type is a collection of values and a set of operations on those values

Purpose of types
- Identify and prevent errors
  - Avoid meaningless or harmful computations
  - Meaningless: (x < 6) + 1 = “bathtub”
  - Harmful?
- Program organization and documentation
  - Separate types for separate concepts
  - Type indicates programmer intent
- Support implementation
  - Allocate right amount of space for variables
  - Select right machine operations
  - Optimization: e.g., use fewer bits when possible
- Key idea: types can be checked

Type errors
- Problem:
  - Underlying memory has no concept of type
  - Everything is just a string of bits:
    - The floating point number 3.375
    - The 32-bit integer 1,079,508,992
    - Two 16-bit integers 16472 and 0
    - Four ASCII characters: @ X NUL NUL
  - Without type checking:
    - Machine will let you store 3.375 and later load 1,079,508,992
    - Violates the intended semantics of the program
**Type system**

- **Idea:**
  - Provide clear interpretation for bits in memory
  - Imposes constraints on use of variables, data
  - Expressed as a set of rules
  - Automatically check the rules
  - Report errors to programmer

- **Key questions:**
  - What types are built into the language?
  - Can the programmer build new types?
  - What are the typing rules?
  - When does type checking occur?
  - How strictly are the rules enforced?

**Expressiveness**

- Consider this Scheme function:

```
(define myfunc (lambda (x)
    (if (list? x) (myfunc(first x))
        (+ x 1)))
```

- What is the type of `x`?
  - Sometimes a list, sometimes an atom
  - Downside?

- What would happen in static typing?
  - Cannot assign a type to `x` at compile time
  - Cannot write this function
  - Static typing is **conservative**

**Types and compilers**

- **What is the role of the compiler?**
- **Example:** we want to generate code for

```
a = b + c * d;
arr[i] = *p + 2;
```

- What does the compiler need to know?

- **Duties:**
  - Enforce type rules of the language
  - Choose operations to be performed
    - Can we do this in one machine instruction?
  - Provide concrete representation – bits
  - What if cannot perform the check at compile-time?

**Type checks**

- **What is a type?**
- **More importantly: why do I care?**

  From *Types and Programming Languages*

  “A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute.”

- **Idea**
  - Divide possible values into groups – types
  - Disallow certain behaviors based on membership

**Type systems**

From language specifications:

- “The result of a unary & operator is a pointer to the object referred to by the operand. If the type of the operand is "T", the type of the result is “pointer to T”.

- “If both operands of the arithmetic operators addition, subtraction and multiplication are integers, then the result is an integer”
Properties of types

These excerpts imply:

- Types have structure
  "Pointer to T" and "Array of Pointer to T"

- Expressions have types
  Types are derived from operands by rules

- Goal: determine types for all parts of a program

Type expressions

(Not to be confused with types of expressions)

- Build a description of a type from:
  - Basic types – also called “primitive types”
    Vary between languages: int, char, float, double
  - Type constructors
    Functions over types that build more complex types
  - Type variables
    Unspecified parts of a type – polymorphism, generics
  - Type names
    An “alias” for a type expression – typedef in C

Type constructors

- Arrays
  - If T is a type, then array(T) is a type denoting an array with elements of type T
  - May have a size component: array(I, T)

- Products or records
  - If T₁ and T₂ are types, then T₁×T₂ is a type denoting pairs of two types
  - May have labels for records/structs
    (*name*, char *) × (*age*, int)

Example

- Static type checker for C
  - Defined over the structure of the program

Rules:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>E₁ + E₂</td>
<td>if type(E₁) is int and type(E₂) is int result type is int else ...other cases...</td>
</tr>
</tbody>
</table>

Question:

How do we get declared types of identifiers, functions?

More examples

- More interesting cases

Rules:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>E₁ [ E₂ ]</td>
<td>if type(E₁) is int and type(E₂) is array(T) result type is T else error</td>
</tr>
<tr>
<td>* E</td>
<td>if type(E) is pointer(T) result type is T else error</td>
</tr>
</tbody>
</table>
Example

- What about function calls?
  - Consider single argument case

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E_1 (E_2)$</td>
<td>if type($E_1$) is D $\rightarrow$ R and type($E_2$) is D result type is R else error</td>
</tr>
</tbody>
</table>

- How do we perform these checks?
  - What is the core type-checking operation?
  - How do I determine if "type($E$) is D"?

  "If two type expressions are equivalent then..."

Type equivalence

- Implementation: structural equivalence
  - Same basic types
  - Same set of constructors applied

<table>
<thead>
<tr>
<th>Type equivalence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recursive test:</td>
</tr>
<tr>
<td>function equiv(s, t) if s and t are the same basic type return true if s = pointer(s_1) and t = pointer(t_1) return equiv(s_1, t_1) if s = s_1 * s_2 and t = t_1 * t_2 return equiv(s_1, t_1) &amp;&amp; equiv(s_2, t_2) ... etc...</td>
</tr>
</tbody>
</table>

Representation

- Represent types as graphs
  - Node for each type
  - Often a DAG: share the structure when possible

  ![Graph representation of types]

  Function: (char * int) $\rightarrow$ int *

Structural equivalence

- Efficient implementation
  - Recursively descend DAG until common node

  Many subtle variations in practice
  - Special rules for parameter passing
    - C: array $T[]$ is compatible with $T*$
    - Pascal, Fortran: leaving off size of array
    - Is "size" part of the type?
  - Type qualifiers: const, static, etc.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E_i = E_j$</td>
<td>if type($E_i$) == type($E_j$) result type is $E_i$ else error</td>
</tr>
</tbody>
</table>

Notions of equivalence

- Different way of handling type names

  **Structural equivalence**
  - Ignores type names
  - typedef int * numptr means numptr $\equiv$ int *
  - Not always desirable
  - Example?

  **Name equivalence**
  - Types are equivalent if they have the same name
  - Solves an important problem: recursive types

Recursive types

- Why is this a problem?

  ```
  struct cell {
    int info;
    struct cell * next;
  }
  ```

  Cycle in the type graph!
  - C uses structural equivalence for everything except structs (and unions)
    - The name "struct cell" is used instead of checking the actual fields in the struct
  - Can we have two compatible struct definitions?
Java types

- Type equivalence for Java

```java
class Foo {
    int x;
}
class Bar {
    int w;
    float y;
    float z;
}
```

- Can we pass Bar objects to a method taking a type Foo?
  - No
  - Java uses name equivalence for classes
  - What can we do in C that we can’t do in Java?

Type checking

- Consider this case:
  - What is the type of `x + i` if `x` is `float` and `i` is `int`?
  - Is this an error?
  - Compiler fixes the problem
    - Convert into compatible types
    - Automatic conversions are called **coercions**
    - Rules can be complex
      - In C, large set of rules for called **integral promotions**
      - Goal is to preserve information

Type coercions

- Rules
  - Find a common type
  - Add explicit conversion into the AST

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type rule</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>E₁ + E₂</code></td>
<td></td>
</tr>
<tr>
<td>If <code>type(E₁)</code> is <code>int</code> and <code>type(E₂)</code> is <code>int</code></td>
<td></td>
</tr>
<tr>
<td>Result type is <code>int</code></td>
<td></td>
</tr>
<tr>
<td>Else</td>
<td></td>
</tr>
<tr>
<td>If <code>type(E₁)</code> is <code>int</code> and <code>type(E₂)</code> is <code>float</code></td>
<td></td>
</tr>
<tr>
<td>Result type is <code>float</code></td>
<td></td>
</tr>
<tr>
<td>Else</td>
<td></td>
</tr>
<tr>
<td>If <code>type(E₁)</code> is <code>float</code> and <code>type(E₂)</code> is <code>int</code></td>
<td></td>
</tr>
<tr>
<td>Result type is <code>float</code></td>
<td></td>
</tr>
<tr>
<td>Else</td>
<td></td>
</tr>
</tbody>
</table>

Implementing type checkers

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type rule</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>E₁[E₂]</code></td>
<td></td>
</tr>
<tr>
<td>If <code>type(E₁)</code> is <code>int</code> and <code>type(E₂)</code> is <code>array(T)</code></td>
<td></td>
</tr>
<tr>
<td><code>type(E) = T</code></td>
<td></td>
</tr>
<tr>
<td>Else</td>
<td></td>
</tr>
<tr>
<td><code>* E</code></td>
<td></td>
</tr>
<tr>
<td>If <code>type(E)</code> is <code>pointer(T)</code></td>
<td></td>
</tr>
<tr>
<td><code>type(E) = T</code></td>
<td></td>
</tr>
<tr>
<td>Else</td>
<td></td>
</tr>
</tbody>
</table>

Does this form look familiar?

- Type checking fits into syntax-directed translation
- What kind of attribute is type?

Interesting cases

- What about `printf`?
  - `printf(const char * format, ...)`
  - Implemented with varargs
  - Format specifies which arguments should follow
  - Who checks?

  - Array bounds
    - Array sizes rarely provided in declaration
    - Cannot check statically (in general)
    - *There are fancy-dancy systems that try to do this*
    - Java: check at run-time

Overloading

- “*+” operator
  - Same syntax, same “semantics”, multiple implementations
  - C: `float versus int`
  - C++: arbitrary user implementation
    - Note: cannot change parser — what does that mean?

  - How to decide which one?
    - Use types of the operands
    - Find operator with the right type signature

  - Complex interaction with coercions
    - Need a rule to choose between conversion and overloading
Object oriented types

- What is the relationship between `Foo` and `Bar`?
  - `Bar` is a subtype of `Foo`
  - Any code that accepts a `Foo` object can also accept a `Bar` object
  - We'll talk about how to implement this later

- Modify type compatibility rules
  - To check an assignment, check subtype relationship $\leq$
  - Also for formal parameters

<table>
<thead>
<tr>
<th>Exp</th>
<th>Type rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E_1 = E_2$</td>
<td>if $\text{type}(E_2) \leq \text{type}(E_1)$ result type is $E_1$, else error</td>
</tr>
</tbody>
</table>

Java arrays

- Question: is `bar[]` a subtype of `foo[]`?
  - Answer: yes
  - Consequences?

```java
class Foo {    …    }
class Bar extends Foo {   …   }

void storeIt(Foo f, Object [] arr) {
    arr[0] = f;
}
```

Polymorphism

- Ordinary procedures
  - Accept fixed type signature
  - Example: `search(char c, string s)`

- Generic procedures
  - Work on arguments of different types

- User-defined generics
  - Define interface with type variables
  - Example: `search<El>(El c, El [] list)`

Type checking polymorphism

- Problem:
  - How do we check generics statically?
  - What do we need to know?
    - Does `El` have a method “compareTo”?
    - What is the type of `El` in this instantiation?
    - Is `A` an array of type `int`?

```java
int search<El>(El x, El [] list) {
    for (int i = 0; i < list.length; i++)
        if (x.compareTo(list[i])) return i;
    return -1;
}

…
search(5, A);    // Use of generic
```

Type checking polymorphism

- Checking uses of generics
  - Is `search(5, A)` correct? (Let’s say `int A[20]`)
  - Find a mapping from `5` and `A` to `El` and `El []`
  - Mapping: `El` is `int`
  - This process is called unification

- Unification
  - Given two type expressions, one with type variables
  - Find a substitution of type variables that turns it into the other type expression
  - Used in many other areas: theorem proving, Prolog
Polymorphism

How is polymorphism implemented?

- Type erasure
  - Convert all type variables into Object
  - Java approach
  - Pros and cons?
- Instantiation
  - Generate a new implementation for each combination of type arguments
  - C++ approach
  - Pros and cons?

Overview

- Type system
  - Set of rules for assigning types to parts of the program
  - Inference rules for operators
- Type checker
  - Start with declared types
  - Apply typing rules
  - At each step, make sure resulting types obey the rules of the language
- Why type checking?
  - Alert the programmer to errors
  - Help compiler generate a correct translation

Type inference

- Languages without declarations
  - ML, Haskell
  - Still statically typed
  - Types determined by use
- Requires type inference algorithm
  - Determine constraints from program
    - Example: "x + y" implies x and y must be numeric types
  - Results in type expressions with type variables
    - Example: "x <= number" and "z <= x"
  - Compute a consistent assignment to variables

Static checking

- Question: what can be statically checked?
  - What are the limits?
  - What kinds of things might we want to check?
- In general: undecidable
  - Asking questions about the possible behavior of a program
  - Often degenerates into the halting problem
- Static check is an approximation
  - Properties:
    - Soundness
    - Completeness
  - We'll come back to this in dataflow analysis

Back to Mars Orbiter

- Language support for units
  - Idea: make units a part of the type
  - Use polymorphism for operators
  - Type-check the computations
- Example:
  - double<kg> weight;
  - double<s> time;
  - double<m> distance;
  - double<m s^-2> gravity = 9.8 * m/(s * s);
  - double<kg m s^-2> force = weight * gravity;
- Issues:
  - Type checker must understand algebra
  - Generics are tricky (what is the type of sqrt?)

Template metaprogramming

- C++ templates are very powerful
  - More powerful than the designers realized!
- Template parameters can be primitive types
  - Must be compile-time constants
    - template<int N>
      - class Vector { int the_data[N]; ... }
  - Partial instantiation
    - Can provide partial parameters
    - Compiler chooses the best unification
      - template<1>
        - class Vector { int one_piece_of_data; ... }
Template metaprogramming

```cpp
template<bool C> class IfThen { }

class IfThen<true> {
    public:
    static inline void do() { statement1; }  // true case
};

class IfThen<false> {
    public:
    static inline void do() { statement2; }  // false case
};

// Replacement for 'if/else' statement:
IfThen<condition>::do();
```

Loops (recursion)

```cpp
class Factorial<1> {
    public:
    enum { value = 1 };  
};

template<int N>
class Factorial {
    public:
    enum { value = N * Factorial<N-1>::value };  
};

x = Factorial<8>::value;
```