

# COMP 181

## Lecture 10 Static checking

October 12, 2005



## Prelude

- What is this spacecraft?

*Mars Climate Orbiter*



- What happened to it?
  - After 268 day journey, it attempted to enter into Mars orbit
  - Passed too close, crashed
- Why?

Engineering team at Lockheed using *English units*, NASA using *metric units*



Tufts University Computer Science

2

## Beyond syntax

- What's wrong with this code?  
(Note: it parses perfectly)

```
foo(int a, char * s){ ... }  
  
int bar() {  
    int f[3];  
    int i, j, k;  
    char *p;  
    float k;  
    foo(f[6], 10, j);  
    break;  
    i->val = 5;  
    j = i + k;  
    printf("%s,%s.\n", p, q);  
    goto label23;  
}
```



Tufts University Computer Science

3

## Errors

- Undeclared identifier
- Multiplied declared identifier
- Index out of bounds
- Wrong number or types of args to call
- Incompatible types for operation
- Break statement outside switch/loop
- Goto with no label



Tufts University Computer Science

4

## Kinds of checks

- Uniqueness checks
  - Certain names must be unique
  - Many languages require variable declarations
- Flow-of-control checks
  - Match control-flow operators with structures
  - Example: break applies to innermost loop/switch
- Type checks
  - Check compatibility of operators and operands



Tufts University Computer Science

5

## Static checks

Why do we care?

- Obvious: report mistakes to programmer
  - f[6] will cause a run-time failure
  - Help programmer verify intent
- How do these checks help compiler?
  - Allocate right amount of space for variables
  - Select right machine operations
  - Proper implementation of control structures



Tufts University Computer Science

6

## Today

### Focus on type systems

- Type system
  - Type expressions
  - Type inference
- Checking and conversions
  - Static vs dynamic checks
  - Type conversions
- Polymorphism



## Other semantic checks

- Based on symbol tables
  - Add entries to symbol table
  - Check references against table
  - Example: declarations
    - Add each declaration to table
    - Make sure entries are unique
    - Check that variable uses refer to entries in table

- Closely tied to notion of **scope**

*We'll address these topics when we discuss procedures*



- Duff's device



## Types

- As programmers...
  - We have an intuitive notion of types
  - What is a type?



## Types

- 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



## Types and compilers

- Compiler must understand the type system of the language
  - Enforce limitations
  - Implement checks and conversions
  - Provide concrete representation – bits
- Responsibilities vary by language
  - C/C++ – all type checking occurs in compiler
  - Java – mixed model
  - Perl, Ruby – no compiler, all dynamic checking



## 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

What do 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 names  
An "alias" for a type expression – `typedef` in C
  - Type variables  
Unspecified parts of a type – *polymorphism*
  - Type constructors  
Functions over types that build more complex types



## 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 and records
  - If  $T_1$  and  $T_2$  are types, then  $T_1 \times T_2$  is a type denoting pairs of two types
  - May have labels for records/structs  
(*"name", char \**)  $\times$  (*"age", int*)



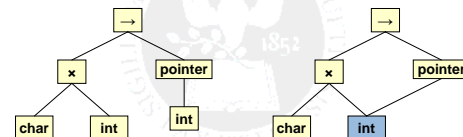
## Type constructors

- Pointers
  - If T is a type, the *pointer(T)* denotes a pointer to T
- Functions or function *signatures*
  - If D and R are types then  $D \rightarrow R$  is a type denoting a function from domain type D to range type R
  - For multiple inputs, domain is a product
  - Notice: primitive operations have signatures  
Mod % operator:  $\text{int} \times \text{int} \rightarrow \text{int}$



## Graphically

- View types as graphs
  - Node for each type
  - Often created as a DAG for efficiency



Function:  $(\text{char} \times \text{int}) \rightarrow \text{int}^*$



## Type checking

- Define language *type system*
  - Set of rules for assigning type expressions to the parts of a program
- Implemented by *type checker*
  - Derives types using rules
  - Static (compile-time) or dynamic (run-time)
- Type checking may fail
  - Handling errors depends on specific constructs



## Example

- Static type checker for C
- Assume:
  - We can get declared types of identifiers, functions

• Rules:

Expression	Type rule
$E_1 [ E_2 ]$	if $\text{type}(E_2)$ is int and $\text{type}(E_1)$ is $\text{array}(T)$ result type is T else <b>error</b>
$* E$	if $\text{type}(E)$ is $\text{pointer}(T)$ result type is T else <b>error</b>



## Example

- What about function calls?
  - Consider single argument case

Expression	Type rule
$E_1 ( E_2 )$	if $\text{type}(E_1)$ is $D \rightarrow R$ and $\text{type}(E_2)$ is D result type is R else <b>error</b>

- Extends to multiple arguments (products)
- What is fundamental operation?
  - "If two type expressions are equivalent then..."



## Type equivalence

- Implementation: *structural equivalence*

- Same basic types
- Same set of constructors applied

- Recursive test:
 

```
function equiv(s, t)
  if s and t are the same basic type
    return true
  if s = pointer(s1) and t = pointer(t1)
    return equiv(s1, t1)
  if s = s1x2 and t = t1t2
    return equiv(s1, t1) && equiv(s2, t2)
  ...etc...
```

Efficiency is critical



## Structural equivalence

- Efficient implementation
  - Recursively descend tree, compare basic types
  - Recursively descend DAG until common node (Same type always represented by same 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
  - Type qualifiers: `const`, `static`, etc.



## Name equivalence

- Different way of handling type names
- Structural equivalence
  - Ignores type names
  - `typedef int * numptr → numptr ≡ int *`
  - Not always desirable
- Name equivalence
  - Types are equivalent if they have the same name
  - Solves an important problem: recursive types



## Recursive types

- Cycle in the type graph:

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

- C uses structural equivalence for everything except structs
  - The name "struct cell" is used instead of checking the actual fields in the struct
  - Can we have two compatible struct definitions?



- Type equivalence for 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

Expression	Type rule
$E_1 + E_2$	if <code>type(E<sub>1</sub>)</code> is <code>int</code> and <code>type(E<sub>2</sub>)</code> is <code>int</code> result type is <code>int</code>
	if <code>type(E<sub>1</sub>)</code> is <code>int</code> and <code>type(E<sub>2</sub>)</code> is <code>float</code> result type is <code>float</code>
	if <code>type(E<sub>1</sub>)</code> is <code>float</code> and <code>type(E<sub>2</sub>)</code> is <code>int</code> result type is <code>float</code>
	...etc...



## Overloading

- "+" operator
  - Same syntax, multiple implementations
  - C: `float` versus `int`
  - C++: arbitrary user implementation
- How to decide which one?
  - Use types of the operands
  - Find operator with the right type signature
- How does this interact with coercions?



## Object oriented types

- ```
class foo { ... }
class bar extends foo { ... }
```
- What is relationship between `foo` and `bar`?
    - Any code that accepts a `foo` object can also accept a `bar` object
    - `bar` is a subtype of `foo`
  - Modify type compatibility rules
    - Formal parameter can accept any subtype
    - Same holds for assignment



## Type inference

- Languages without declarations
  - ML, Haskell
  - Still statically typed
  - Types determined by use
- Requires *type inference* algorithm
  - Determine constraints from program
  - Results in type expressions with type variables
  - Compute a consistent assignment to variables



## Implementing type checkers

| Expression                  | Type rule                                                                                                                              |
|-----------------------------|----------------------------------------------------------------------------------------------------------------------------------------|
| $E \rightarrow E_1 [ E_2 ]$ | if $\text{type}(E_2)$ is <code>int</code> and $\text{type}(E_1)$ is <code>array(T)</code><br>$\text{type}(E) = T$<br>else <b>error</b> |
| $E \rightarrow * E$         | if $\text{type}(E)$ is <code>pointer(T)</code><br>$\text{type}(E)$ is <code>T</code><br>else <b>error</b>                              |

- Does this form look familiar?
  - Type checking fits into syntax-directed translation
  - One reason why declarations precede stmts



## Expressions

- Why compute types of all expressions?
  - Example:  $x + y + z$  *what is type of  $(x+y)$ ?*
- Already mentioned
  - Check for correctness
  - Add coercions
- Code generation
  - Most machines cannot add 3 values
  - Must break up the expression
  - Store intermediate results



## 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



## Polymorphism

- Ordinary procedures
  - Accept fixed type signature
  - Example: `search(char c, string s)`
- Generic procedures
  - Work on arguments of different types
  - Example: arithmetic operators *(most languages)*
- User-defined generics
  - Define interface with type variables
  - Example: `search(E1 c, E1 [] list)`



## Polymorphic type checking

- Problem:
  - Is `search(5, A)` correct? *(Let's say `int A[20]`)*
  - Find a mapping from 5 and A to E1 and E1 []
  - Mapping: E1 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



## Generics in Java

- New in Java 1.5
  - Type variables, like C++ templates
  - Not as powerful (on purpose)  
`boolean search(T e1, List<T> list)`
- Question:
  - How is this different from:  
`boolean search(Object e1, List list)`  
(Where list is a linked list of Object references)



## 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?*)



## Next time...

- The procedure abstraction
  - Symbol tables and scopes
  - Run-time environments
  - Storage allocation, stacks
- *Later today*: new programming assignment

