**Semantic Analysis: Types and Type Checking**

**CS 471**  
October 10, 2007

---

**Beyond Syntax**

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

```c
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;
}
```

---

**Errors**

- Undeclared identifier
- Multiply 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

---

**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

---

**Program Checking**

Why do we care?

**Obvious:**  
- Report mistakes to programmer  
- Avoid bugs:  
  [f[6] will cause a run-time failure]
- Help programmer verify intent

**How do these checks help compilers?**  
- Allocate right amount of space for variables  
- Select right machine operations  
- Proper implementation of control structures
Goals of a Semantic Analyzer
Find all possible remaining errors that would make program invalid

Static checks
− Done by the compiler
− Detect and report errors by analyzing the sources

Dynamic checks
− Done at run time
− Detect and handle errors as they occur

What are the pros and cons?
• Efficiency? Completeness? Developer vs. user experience? Language flexibility?

Types
What is a type?
• The notion varies from language to language

Consensus
• A set of values
• A set of operations allowed on those values

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

Why Do We Need Type Systems?
Consider the assembly language fragment

```
addi $r1, $r2, $r3
```

What are the types of $r1, $r2, $r3?

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

Type Checking Overview
Four kinds of languages:

• **Statically typed**: All or almost all checking of types is done as part of compilation

• **Dynamically typed**: Almost all checking of types is done as part of program execution (no compiler) as in Perl, Ruby

• **Mixed Model**: Java

• **Untyped**: No type checking (machine code)

The Type Wars
Competing views on static vs. dynamic typing

**Static typing proponents say:**
• Static checking catches many programming errors at compile time
• Avoids overhead of run-time type checks

**Dynamic typing proponents say:**
• Static type systems are restrictive
• Rapid prototyping easier in a dynamic type system

In practice, most code is written in statically typed languages with an “escape” mechanism
• Unsafe casts in C, Java
• The best or worst of both worlds?
Type Checking and Type Inference

**Type Checking** is the process of verifying fully typed programs
- Given an operation and an operand of some type, determine whether the operation is allowed

**Type Inference** is the process of filling in missing type information
- Given the type of operands, determine
  - the meaning of the operation
  - the type of the operation
- OR, without variable declarations, infer type from the way the variable is used

The two are different, but are often used interchangeably

Issues in Typing

Does the language have a type system?
- Untyped languages (e.g. assembly) have no type system at all

When is typing performed?
- Static typing: At compile time
- Dynamic typing: At runtime

How strictly are the rules enforced?
- Strongly typed: No exceptions
- Weakly typed: With well-defined exceptions

Type equivalence & subtyping
- When are two types equivalent?
  - What does "equivalent" mean anyway?
  - When can one type replace another?

Components of a Type System

**Built-in types**

**Rules for constructing new types**
- Where do we store type information?

**Rules for determining if two types are equivalent**

**Rules for inferring the types of expressions**

Component: Built-in Types

**Integer**
- usual operations: standard arithmetic

**Floating point**
- usual operations: standard arithmetic

**Character**
- character set generally ordered lexicographically
- usual operations: (lexicographic) comparisons

**Boolean**
- usual operations: not, and, or, xor

Component: Type Constructors

**Arrays**
- array(I,T) denotes the type of an array with elements of type T, and index set I
- multidimensional arrays are just arrays where T is also an array
- operations: element access, array assignment, products

**Strings**
- bitstrings, character strings
- operations: concatenation, lexicographic comparison

**Records (structs)**
- Groups of multiple objects of different types where the elements are given specific names.

**Pointers**
- addresses
- operations: arithmetic, dereferencing, referencing
- issue: equivalency

**Function types**
- A function such as "int add(real, int)" has type real-int→int
Component: Type Equivalence

Name equivalence
• Types are equiv only when they have the same name

Structural equivalence
• Types are equiv when they have the same structure

Example
• C uses structural equivalence for structs and name equivalence for arrays/pointers
• Tiger uses name equivalence for both

let type a = {x:int, y:int} let type b = {x:int, y:int} var i : a := … var j : b := …
in i := j end

Type Coercion

• If x is float, is x = 3 acceptable?
  – Disallow
  – Allow and implicitly convert 3 to float
  – "Allow" but require programmer to explicitly convert 3 to float

• What should be allowed?
  – float to int ?
  – int to float ?
  – What if multiple coercions are possible?
    • Consider 3 + "4" …

Formalizing Types: Rules of Inference

We have seen two examples of formal notation specifying parts of a compiler
• Regular expressions (for the lexer)
• Context-free grammars (for the parser)

The appropriate formalism for type checking is logical rules of inference

Rules of Inference

Inference rules have the form

If Hypothesis is true, then Conclusion is true

Type checking computes via reasoning

If E₁ and E₂ have certain types,
then E₃ has certain type

Rules of inference are a compact notation for "If-Then" statements

From English to an Inference Rule

The notation is easy to read (with practice)

Start with a simplified system and gradually add features

Building blocks
• Symbol \(\land\) is "and"
• Symbol \(\Rightarrow\) is "if-then"
• \(x:T\) is "\(x\) has type \(T\)"

From English to an Inference Rule (2)

If \(e₁\) has type \(\text{int}\) and \(e₂\) has type \(\text{int}\),
then \(e₁ + e₂\) has type \(\text{int}\)

\((e₁ \text{ has type } \text{int} \land e₂ \text{ has type } \text{int}) \Rightarrow e₁ + e₂ \text{ has type } \text{int}\)

\((e₁ : \text{int} \land e₂ : \text{int}) \Rightarrow e₁ + e₂ : \text{int}\)
From English to an Inference Rule (3)
The statement
\((e_1: \text{int} \land e_2: \text{int}) \Rightarrow e_1 + e_2: \text{int}\)
is a special case of
\((\text{Hypothesis}_1 \land \ldots \land \text{Hypothesis}_n) \Rightarrow \text{Conclusion}\)
This is an inference rule

Notation for Inference Rules
By tradition inference rules are written
\[ \frac{\text{Hypothesis}_1 \ldots \text{Hypothesis}_n}{\text{Conclusion}} \]
Our type rules have hypotheses and conclusions of the form:
\[ \vdash e : T \]
\[ \vdash \text{means "it is provable that . . ."} \]

Two Rules
\[ \frac{i \text{ is an integer}}{\vdash i: \text{int}} \]
\[ \vdash e_1: \text{int} \]
\[ \vdash e_2: \text{int} \]
\[ \vdash e_1 + e_2: \text{int} \]

Example: 1 + 2
\[ \vdash 1: \text{int} \]
1 is an integer
\[ \vdash 2: \text{int} \]
2 is an integer
\[ \vdash 1 + 2: \text{int} \]

A Few More Rules...
\[ \vdash e: \text{boolean} \]
\[ \vdash \neg e: \text{boolean} \]
\[ \vdash e_1: \text{int} \]
\[ \vdash e_2: \text{int} \]
\[ \vdash e_1 < e_2: \text{boolean} \]
\[ \vdash e_1: \text{boolean} \]
\[ \vdash e_2: \text{boolean} \]
\[ \vdash e_1 \&\& e_2: \text{boolean} \]

What’s the Point?
• Provides a concise (several pages vs. several hundred pages) representation of a language
  − Hence, you'll see a LOT of this in a PL course

But that's enough for our purposes
Variables/Identifiers

Need an environment that keeps track of types of all identifiers in scope

```java
int i, n = ...;
for (i=0; i < n; i++)
    boolean b = ...
}
```

Symbol Tables

**purpose:**
- keep track of names declared in the program
- names of variables, functions, classes, types, ...

**symbol table entry:**
- kind of name (variable, type, function, etc)
- type (int, float, etc)
- nesting level
- size
- memory location (i.e., where will it be found at runtime)

Symbol Tables

- Symbol table (also called environments)
  - Symbol table (static)
  - Environment (dynamic)
- Can be represented as set of name → type pairs (bindings) (a → string, b → int)

Functions:
Type Lookup(String id)
Void Add(String id, Type binding)

Environments

Represents a set of mappings in the symbol table

```java
function f(a:int, b:int, c:int) =
    print_int(a+c);
    let var j := a+b
    var a := "hello"
    in print(a); print_int(j)
end;
print_int(b)
```

Imperative vs. Functional Environments

Functional style – keep all copies of σ₀ σ₁ σ₂ ...

Imperative style – modify σ₁ until it becomes σ₂
- “destroys” σ₁
- can “undo” σ₂ to get back to σ₁ again
- single environment σ that becomes σ₁ σ₂ σ₃
- latest bindings destroyed, old bindings restored

**NOTE:** Functional/imperative environment management can be used regardless of whether the language is “functional” “imperative” or “object-oriented”

Implementation Options

- **Array**
  - simple, but linear LookUp time
  - However, we may use a sorted array for reserved words, since they are generally few and known in advance
- **Binary Tree**
  - O(log n) lookup time if kept balanced
- **Hash Table**
  - most common implementation
  - O(1) LookUp time
Scope Issues

One idea is to have a global symbol table and save the scope information for each entry.
- When an identifier goes out of scope, scan the table and remove the corresponding entries
  - We may even link all same-scope entries together for easier removal
- Careful: deleting from a hash table that uses open addressing is tricky
  - We must mark a slot as Deleted, rather than Empty, otherwise later LookUp operations may fail

Alternatively, we can maintain a separate, local table for each scope.

Structure Tables

Where should we store struct field names?
- Separate mini symbol table for each struct
  - Conceptually easy
- Separate table for all struct field names
  - We need to somehow uniquely map each name to its structure (e.g. by concatenating the field name with the struct name)
- No special storage
  - struct field names are stored in the regular symbol table
  - Again we need to be able to map each name to its structure

How symbol tables work (1)

```c
int x;
char y;
void p(void)
{ double x;
  ( int y[10];

  }

void q(void)
{ int y;

  }

main()
{ char x;

  }
```

How symbol tables work (2)

```c
int x;
char y;
void p(void)
{ double x;
  ( int y[10];

  }

void q(void)
{ int y;

  }

main()
{ char x;

  }
```

How symbol tables work (3)

```c
int x;
char y;
void p(void)
{ double x;
  ( int y[10];

  }

void q(void)
{ int y;

  }

main()
{ char x;

  }
```

How symbol tables work (4)

```c
int x;
char y;
void p(void)
{ double x;
  ( int y[10];

  }

void q(void)
{ int y;

  }

main()
{ char x;

  }
```
int x;
char y;

void p(void) {
    double x;
    ...
    (int y[10];
    ...
    )
    void q(void) {
        int y;
        ...
    }
    main() {
        char x;
        ...
    }
}

int x;
char y;

void p(void) {
    double x;
    ...
    (int y[10];
    ...
    )
    void q(void) {
        int y;
        ...
    }
    main() {
        char x;
        ...
    }