Types and Static Semantic Analysis

COMS W4115

Prof. Stephen A. Edwards
Fall 2004
Columbia University
Department of Computer Science

Data Types

What is a type?
A restriction on the possible interpretations of a segment of memory or other program construct.

Useful for two reasons:
Runtime optimization: earlier binding leads to fewer runtime decisions. E.g., Addition in C efficient because type of operands known.
Error avoidance: prevent programmer from putting round peg in square hole. E.g., In Java, can’t open a complex number, only a file.

Are Data Types Necessary?

No: many languages operate just fine without them.
Assembly languages usually view memory as undifferentiated array of bytes. Operators are typed, registers may be, data is not.
Basic idea of stored-program computer is that programs be indistinguishable from data.
Everything’s a string in Tcl including numbers, lists, etc.

C’s Types: Base Types/Pointers

Base types match typical processor

Typical sizes: 8 16 32 64
char short int long
float double

Pointers (addresses)
int *i; /* i is a pointer to an int */
char **j; /* j is a pointer to a pointer to a char */

C’s Types: Arrays, Functions

Arrays
char c[10]; /* c[0] ... c[9] are chars */
double a[10][3][2]; /* array of 10 arrays of 3 arrays of 2 doubles */

Functions
/* function of two arguments returning a char */
char foo(int, double);

C’s Types: Structs and Unions

Structures: each field has own storage
struct box {
    int x, y, h, w;
    char *name;
};

Unions: fields share same memory
union token {
    int i;
    double d;
    char *s;
};

Composite Types: Records

A record is an object with a collection of fields, each with a potentially different type. In C,

struct rectangle {
    int n, s, e, w;
    char *label;
    color col;
    struct rectangle *next;
};

struct rectangle r;
r.n = 10;
r.label = "Rectangle";

Applications of Records

Records are the precursors of objects:
Group and restrict what can be stored in an object, but not what operations they permit.
Can fake object-oriented programming:

struct poly { ... };

struct poly *poly_create();
void poly_destroy(struct poly *p);
void poly_draw(struct poly *p);
void poly_move(struct poly *p, int x, int y);
int poly_area(struct poly *p);

C’s Types: Variant Records

A record object holds all of its fields. A variant record holds only one of its fields at once. In C,

union token {
    int i;
    float f;
    char *string;
};

union token t;
t.i = 10;
t.f = 3.14159; /* overwrites t.i */
t.string = "Hello World";

char *s = t.string; /* returns gibberish */
Applications of Variant Records

A primitive form of polymorphism:

```c
struct poly {
    int x, y;
    int type;
    union { int radius;
              int size;
              float angle; } d;
};
```

- If `poly.type == CIRCLE`, use `poly.d.radius`.
- If `poly.type == SQUARE`, use `poly.d.size`.
- If `poly.type == LINE`, use `poly.d.angle`.

Layout of Records and Unions

Modern processors have byte-addressable memory.

- 16-bit integer:
  - 0: 1 0
- 32-bit integer:
  - 3 2 1 0

- Many data types (integers, addresses, floating-point numbers) are wider than a byte.

Reading an aligned 32-bit value is fast: a single operation.

- 3 2 1 0
- 7 6 5 4
- 11 10 9 8

C’s Type System: Enumerations

```c
enum weekday {sun, mon, tue, wed, thu, fri, sat};
enum days {sun, wed, sat};
enum class {mon, wed}; /* error: mon, wed redefined */
```

Statically-Typed Languages

- Statically-typed: compiler can determine types.
- Dynamically-typed: types determined at run time.

Is Java statically-typed?

```java
class Foo {
    public void x() { ... }
}
class Bar extends Foo {
    public void x() { ... }
}
void baz(Foo f) {
    f.x();
}
```
Polymorphism

Say you write a sort routine:

```c
void sort(int a[], int n) {
    int i, j;
    for ( i = 0 ; i < n-1 ; i++ )
        for ( j = i + 1 ; j < n ; j++ )
            if (a[j] < a[i]) {
                int tmp = a[i];
                a[i] = a[j];
                a[j] = tmp;
            }
}
```

To sort doubles, only need to change a few types:

```c
void sort(double a[], int n) {
    int i, j;
    for ( i = 0 ; i < n-1 ; i++ )
        for ( j = i + 1 ; j < n ; j++ )
            if (a[j] < a[i]) {
                double tmp = a[i];
                a[i] = a[j];
                a[j] = tmp;
            }
}
```

C++ Templates

C++ templates are essentially language-aware macros. Each instance generates a different refinement of the same code.

```cpp
template <class T> void sort(T a[], int n) {
    int i, j;
    for ( i = 0 ; i < n-1 ; i++ )
        for ( j = i + 1 ; j < n ; j++ )
            if (a[j] < a[i]) {
                T tmp = a[i];
                a[i] = a[j];
                a[j] = tmp;
            }
}
int a[10];
sort<int>(a, 10);
```

Faking Polymorphism with Objects

```cpp
class Sortable {
    bool lessthan(Sortable s) = 0;
};
void sort(Sortable a[], int n) {
    int i, j;
    for ( i = 0 ; i < n-1 ; i++ )
        for ( j = i + 1 ; j < n ; j++ )
            if ( a[j].lessthan(a[i]) ) {
                Sortable tmp = a[i];
                a[i] = a[j];
                a[j] = tmp;
            }
}
```

This sort works with any array of objects derived from Sortable. Same code is used for every type of object. Types resolved at run-time (dynamic method dispatch). Does not run as quickly as the C++ template version.

```
```

Arrays

Most languages provide array types:

```c
char i[10];         /* C */
character(10) i     /* FORTRAN */
i : array (0..9) of character; -- Ada
```

```pascal```
var i : array [0 .. 9] of char; { Pascal }
```
Allocating Fixed-Size Arrays

Local arrays with fixed size are easy to stack.

```c
void foo()
{
    int a;
    int b[10];
    int c;
}
```

Return address
- FP

Variables remain constant offset from frame pointer.

Allocating Variable-Sized Arrays

Variable-sized local arrays aren't as easy.

```c
void foo(int n)
{
    int a;
    int b[n];
    int c;
}
```

Return address
- FP

Doesn't work: generated code expects a fixed offset for c. Even worse for multi-dimensional arrays.

Allocating Variable-Sized Arrays

As always: add a level of indirection

```c
void foo(int n)
{
    int a;
    int b[n];
    int c;
}
```

Return address
- FP

Variables remain constant offset from frame pointer.

Lexical analysis: Make sure tokens are valid

```
if i 3 "This" /* valid */
@a123 /* invalid */
```

Syntactic analysis: Makes sure tokens appear in correct order

```
for i := 1 to 5 do 1 + break /* valid */
if i 3 /* invalid */
```

Semantic analysis: Makes sure program is consistent

```
let v := 3 in v + 8 end /* valid */
let v := "f" in v(3) + v end /* invalid */
```

Name vs. Structural Equivalence

```
let
type a = { x: int, y: int }
type b = a
var i : a := a { x = 1, y = 2 }
var j : b := b { x = 0, y = 0 }
in
i := j
end
```

Not legal because `a` and `b` are considered distinct types.

```
let
type a = { x: int, y: int }
type b = a
var i : a := a { x = 1, y = 2 }
var j : b := b { x = 0, y = 0 }
in
i := j
end
```

legal because `b` is an alias for type `a`.

( `x: int, y: int` ) creates a new type, not the `type` keyword.

## Things to Check

Make sure variables and functions are defined.

```
let var i := 10
in i(10,20) /* Error: i is a variable */
end
```

Verify each expression's types are consistent.

```
let var i := 10
    var j := "Hello"
in i + j /* Error: i is int, j is string */
end
```

- Used identifiers must be defined
- Function calls must refer to functions
- Identifier references must be to variables
- The types of operands for unary and binary operators must be consistent.
- The first expression in an `if` and `while` must be a Boolean.
- It must be possible to assign the type on the right side of an assignment to the lvalue on the left.
- ...
Static Semantic Analysis
Basic paradigm: recursively check AST nodes.

Implementing Static Semantics
Recursive walk over the AST.
Analysis of a node returns its type or signals an error.
Implicit "environment" maintains information about what symbols are currently in scope.
TigerSemant.g is a tree grammar that does this.

Ask yourself: at a particular node type, what must be true?

Type Classes
package Semant;
public abstract class Type {
    public Type actual()
        public boolean coerceTo(Type t)
    }

class INT() // int

class STRING() // string

class NIL() // nil

class VOID() // ()

class NAME(String n) // type a = b

class ARRAY(Type e) // array of int

class RECORD(String n, Type t, RECORD next)

Type Classes
actual() returns the actual type of an alias, e.g.,
type a = int
type b = a
type c = b
c.actual() will return the INT type.

class NIL() corresponds to the nil keyword.
The VOID type corresponds to expressions that return no value.

Type Classes
coerceTo() answers the "can this be assigned to" question.
type a = {x:int}
type b = a
nil.coerceTo(a) is true
b.coerceTo(a) is true
a.coerceTo(nil) is false

TigerSemant.g

```
expr returns [Type t]
{ Type a, b, c; t = env.getVoidType(); }
| "nil" { t = env.getNilType(); }
| t=1value
| STRING { t = env.getStringType(); }
| NUMBER { t = env.getIntType(); }
| #( NEG a=expr
   { /* Verify expr is an int */
     if ( !(a instanceof Semant.INT))
       semantError(#expr,
           "Operand not integer");
       t = env.getIntType();
   } )
```

Type Classes
The RECORD class is a linked list representation of record types.
type point = { x: int, y: int }
new RECORD("x", intType,
            new RECORD("y", intType, null))

Environment.java
package Semant;

class Environment {
    public Table vars = new Table();
    public Table types = new Table();
    public INT getIntType()
    public VOID getVoidType()
    public NIL getNilType()
    public STRING getStringType()
    public void enterScope()
    public void leaveScope()
    }

Symbol Tables

```java
class Semant {
    public class Table {
        public Table {
            public Object get(String key) {
            }
            public void put(String key, Object value) {
            }
            public void enterScope() {
            }
            public void leaveScope() {
            }
        }
    }
}
```

Symbol Tables Operations:
- `put(String key, Object value)` inserts a new named object in the table, replacing any existing one in the current scope.
- `get(String key)` returns the object of the given name, or null if there isn’t one.

Symbol Table Scopes
- `enterScope()` pushes a new scope on a stack.
- `leaveScope()` removes the topmost one.

```java
Table t = new Table();
t.put("a", new VarEntry(env.getIntType()));
t.put("a", new VarEntry(env.getStringType()));
t.get("a"); // string
```

Symbol Table Objects
- Discriminates between variables and functions.
- Stores extra information for each.

```java
package Semant;
public VarEntry(Type t) {
    public FunEntry(RECORD f, Type r) {
        RECORD argument represents the function arguments; other is the return type.
    }
}
```

Symbol Tables and the Environment
- The environment has two symbol tables:
  - `types` for types
  - `vars` for variables and functions
- Objects stored in symbol table are `Types` and `VarEntry`s.

Rule for an Identifier
- `lvalue returns [Type t]`:
  ```java
  : i:ID {
      Entry e = (Entry) env.vars.get(i.getText());
      if ( e == null )
        semantError(i, i.getText()+' undefined');
      if ( !(e instanceof VarEntry) )
        semantError(i, i.getText()+' not variable');
      VarEntry v = (VarEntry) e;
      t = v.ty;
  }
  ```

Rule for Let
- `#(| #( "let" { env.enterScope(); } | #(DECLS (#(DECLS (decl)+ ))+ )| ) | )`:
  ```java
  : #(| "var" i:ID { a:type = "nil" { a = null; } ) b=expr |
      { /* Verify a=b if a != null */
        /* Make sure b != nil if a == null */
        env.vars.put(i.getText(), new VarEntry(b));
      }
  }
  ```

Partial rule for Var
- `decl { Type a, b; }`:
  ```java
  : #(| BINOP a=expr b=expr |
      String op = #expr.getText();
      if ( op.equals("+")) || op.equals("-") || op.equals("*")) || op.equals("/") {
        if (!a instanceof Semant.INT) || !b instanceof Semant.INT)
          semantError(#expr, op+" operands not int");
        t = a;
      } else {
        /* Check other operators */
      }
  }
  ```

Partial rule for BINOP