Types and Static Semantic Analysis

COMS W4115

Prof. Stephen A. Edwards
Fall 2007
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. Operands 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 char */
char ***k; /* k is a pointer to 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);

Composite 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 */
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;
};
```

```c
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.

```c
0 1 2 3 4
```

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

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

C's Type System: Enumerations

```c
enum weekday {sun, mon, tue, wed, thu, fri, sat};
enum weekday day = mon;
```

```c
eenum days {sun, wed, sat};
eenum class {mon, wed}; /* error: mon, wed redefined */
```

Strongly-typed Languages

Strongly-typed: no run-time type clashes.
C is definitely not strongly-typed:

```c
float g;
union { float f; int i } u;
u.i = 3;
g = u.f + 3.14159; /* u.f is meaningless */
```

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
template <typename 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].lessThan(a[i]) ) {
                T tmp = a[i];
                a[i] = a[j];
                a[j] = tmp;
            }
}
```

Faking Polymorphism with Objects

```c
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;
            }
}
```

Arrays

Most languages provide array types:

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

Array Address Calculation

In C,

```c
struct foo a[10];
```

```c
a[i] is at a + i*sizeof(struct foo)
```

```c
struct foo a[10][20];
```

```c
a[i][j] is at a + (j + 20 * i) * sizeof(struct foo)
```

⇒ Array bounds must be known to access 2D+ arrays

Allocating Arrays

```c
int a[10];      /* static */
void foo(int n)  /* static */
{
    int b[15];  /* stacked */
    int c[n];   /* stacked: tricky */
    int d[];    /* on heap */
    vector<int> e;    /* on heap */

d = new int[n*2]; /* fixes size */
    e.append(1);  /* may resize */
    e.append(2);  /* may resize */
}
Allocating Fixed-Size Arrays

Local arrays with fixed size are easy to stack.

```c
void foo(){
    int a; int b[10]; int c;
    b[0] = 0; ... b[9] = 0;  
    c = 0;  
}
```

Allocating Variable-Sized Arrays

Variable-sized local arrays aren’t as easy.

```c
void foo(int n){
    int a; int b[n]; int c;
    b[0] = 0; ... b[n-1] = 0;  
    c = 0;  
}
```

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

Name vs. Structural Equivalence

```c
struct f {
    int x, y;
} foo = { 0, 1 };  
```

```c
typedef struct f f_t;  
f_t baz;  
baz = foo;  
```

Is this legal?

Things to Check

- 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.

```
1 + break 1 - 5
+ 1 break 1 5
```

```
check(+) check(-)
check(1) = int check(1) = int
check(break) = void check(5) = int
FAIL: int ≠ void
```

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

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.

```
extr expr returns [Type t]
{ Type a, b, c; t = env.getVoidType();}
| "nil" { t = env.getNilType(); }
| t = lvalue
| 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

```
package Semant;
public abstract class Type {
  public Type actual()
  public boolean coerceTo(Type t)
}
public INT()
public STRING()
public VOID()
```

Type Classes

```
coerceTo() answers the "can this be assigned to" question.

int a;
int b;
void.coerceTo(a) is false
b.coerceTo(a) is true
a.coerceTo(void) is true
```

Environment.java

```
package Semant;
public class Environment {
  public Table vars = new Table();
  public Table types = new Table();
  public INT getIntType()
  public STRING getStringType()
  public VOID getVoidType()
  public void enterScope()
  public void leaveScope()
```

Symbol Tables

```
package 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.
Object get(String key) returns the object of the given name, or null if there isn't one.
```

Symbol Table Scopes

```
void enterScope() pushes a new scope on a stack.
void leaveScope() removes the topmost one.
```

```
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.

public VarEntry(Type t)
public FunEntry(Args f, Type r)
f is a list of types representing the arguments, r is the return type.

Symbol Tables and the Environment

The environment has two symbol tables:
- types for types
  Objects stored in symbol table are Types
- vars for variables and functions
  Objects are VarEntry and FunEntry.

Rule for an Identifier

lvalue returns [Type t]
{ Type a, b; t = env.getVoidType(); }
: 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 a New Scope

| #( "let"
  { env.enterScope(); }#(DECLS #(DECLS (decl)+ ))# )
  a=expr
  {
    env.leaveScope();
    t = a;
  }
|

Partial rule for a Declaration

dcl { Type a, b; }
: #( "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 BINOP

| #( 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 */
    }
  }
}