Types and Static Semantic Analysis

COMS W4115

Prof. Stephen A. Edwards
Spring 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 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);

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 *s;
};

union token t;
t.i = 10;
t.f = 3.14159; /* overwrites t.i */
t.label = "gibberish"; /* 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.

```
0
1
2
3
```

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};
```

```c
enum weekday day = mon;
```

Most languages "pad" the layout of records to ensure alignment restrictions.

```c
struct padded {
    int x; /* 4 bytes */
    char z; /* 1 byte */
    short y; /* 2 bytes */
    char w; /* 1 byte */
};
```

SPARC prohibits unaligned accesses.
MIPS has special unaligned load/store instructions.
x86, 68k run more slowly with unaligned accesses.

C's Type System

Types may be intermixed at will:

```c
struct {
    int i;
    union {
        char (*one)(int);
        char (*two)(int, int);
    } u;
    double b[20][10];
} *a[10];
```

Array of ten pointers to structures. Each structure contains an int, a 2D array of doubles, and a union that contains a pointer to a char function of one or two arguments.

C's Type System

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

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 */
```

Is Java strongly-typed?

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

Is Java statically-typed?

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

Statically-Typed Languages

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

Is Java statically-typed?
Polymorphism

Say you write a sort routine:
void sort(int a[], int n)
{
    int i, j;
    for ( i = 0 ; i < n­1 ; i++ )
        for ( j = i + 1 ; j < n ; j++ )
            if (a[j] < a[i]) {
                int tmp = a[i];
                a[i] = a[j];
                a[j] = tmp;
            }
}

Polymorphism

To sort doubles, only need to change a few types:
void sort(double a[], int n)
{
    int i, j;
    for ( i = 0 ; i < n­1 ; 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

template <class T> void sort(T a[], int n)
{
    int i, j;
    for ( i = 0 ; i < n­1 ; 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);

C++ Templates

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

template<int>(a, 10);
template<double>(b, 30);
template<char*>(c, 20);

Fast code, but lots of it.

Faking Polymorphism with Objects

class Sortable {
    bool lessthan(Sortable s) = 0;
}

void sort(Sortable a[], int n) {
    int i, j;
    for ( i = 0 ; i < n­1 ; 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;
            }
}

Faking Polymorphism with Objects

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:
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,

struct foo a[10];
a[i] is at a + i * sizeof(struct foo)

struct foo a[10][20];
a[i][j] is at a + (j + 20 * i) * sizeof(struct foo)

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

Allocating Arrays

int a[10]; /* static */

void foo(int n) {
    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;
}
```

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

Allocating Variable-Sized Arrays

Variable-sized local arrays aren't as easy.

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

Variables remain constant offset from frame pointer.

Static Semantic Analysis

Lexical analysis: Make sure tokens are valid

```c
if i 3 "This" /* valid */
#define123 /* invalid */
```

Syntactic analysis: Makes sure tokens appear in correct order

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

Semantic analysis: Makes sure program is consistent

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

Name vs. Structural Equivalence

typedef struct { int x, y; } foo;
typedef struct { int x, y; } bar;
typedef bar baz;
foo a = { 1, 2 }, c = { 4, 5 }
bar b = { 3, 4 }
baz d = { 5, 6 }
void baz() {
    c = a; /* OK: both foo */
    b = d; /* OK: baz is an alias for bar */
    b = a; /* Bad: foo not the same as bar */
}
```

Things to Check

Make sure variables and functions are defined.

```c
int i;
i(10,20); /* Error: i is a variable, not function */
```

Verify each expression's types are consistent.

```c
int i = 10;
char *s = "Hello";
i + j /* Error: i is int, j is string */
```

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
\[ \frac{1}{\text{break}} \frac{1}{5} \]
```

```
check(+) check(-)
check(1) = int
check(break) = void
FAIL: int \neq 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.

```
expr returns [Type t]
{ Type a, b, c; t = env.getVoidType();}
: "nil" { t = env.getNilType(); }
| t=lvvalue
| 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() // int
public STRING() // string
public NIL() // nil
public VOID() // ()
public NAME(String n) // type a = b
public ARRAY(Type e) // array of int
```

```
t.c coerceTo(a) is true
b.coerceTo(a) is true
a.coerceTo(nil) is false
```

**Environment.java**

```
package Semant;
public 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()
}
```

```
t.enterScope();
Object get("a"); // string
Object get("a"); // int
```

```
t.get("a"); // string
```

**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()
}
```

```
put("a", new VarEntry(env.getIntType()));
t.put("a", new VarEntry(env.getStringType()));
t.get("a"); // string
t.enterScope();
t.get("a"); // string
t.put("a", new VarEntry(env.getIntType()));
t.get("a"); // int
t.leaveScope();
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(Args f, Type r)
```

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 VarEntries and FunEntries.

Rule for an Identifier

```java
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 C-style Block

```java
| #( BLOCK
  { env.enterScope(); }
  #(DECLS (DECLS (DECL+ )+ )= )
  a=expr
  |
  env.leaveScope();
  |
  t = a;
  |
  )
```

Variable Declaration

```java
decl { Type a, b; }
: #( DECL a=type i:ID b=expr
  { /* Verify type of b is a */
    env.vars.put(i.getText(), new VarEntry(b));
  }
)
```

Partial rule for BINOP

```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 */
    } |
  }
```