Types and Static Semantic Analysis

Stephen A. Edwards

Columbia University

Fall 2010

Part I

Types

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);
```

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:

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

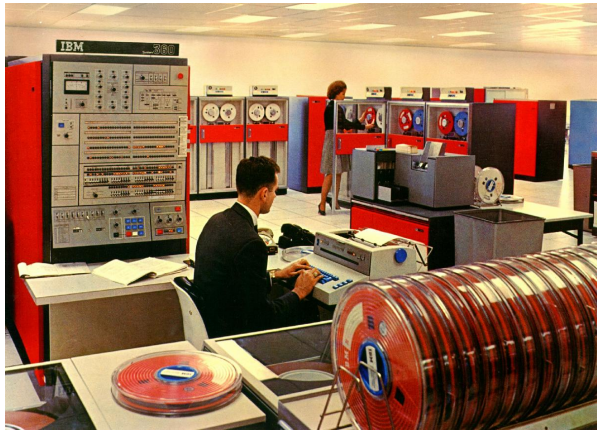
If `poly.type == CIRCLE`, use `poly.d.radius`.

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.

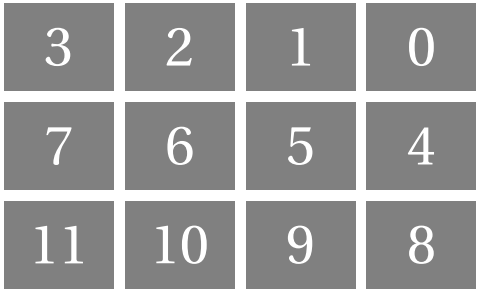


The IBM 360 (c. 1964) helped to popularize byte-addressable memory.

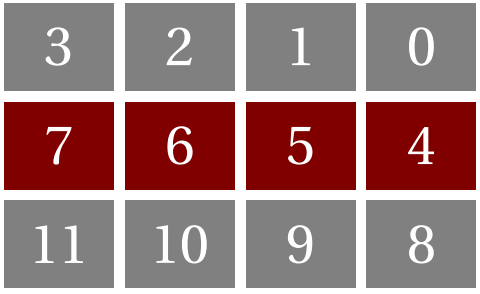
Many data types (integers, addresses, floating-point numbers) are wider than a byte.

Layout of Records and Unions

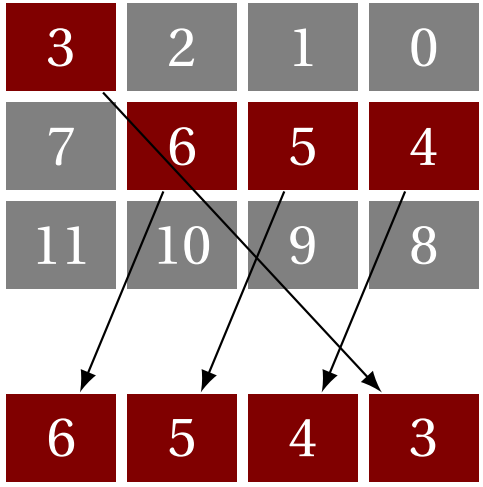
Modern memory systems read data in 32-, 64-, or 128-bit chunks:



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



It is harder to read an unaligned value: two reads plus shifting



SPARC prohibits unaligned accesses.

MIPS has special unaligned load/store instructions.

x86, 68k run more slowly with unaligned accesses.

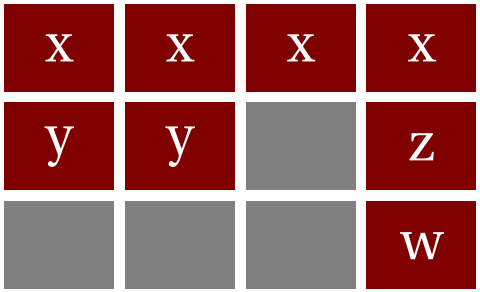
Padding

To avoid unaligned accesses, the C compiler pads the layout of unions and records.

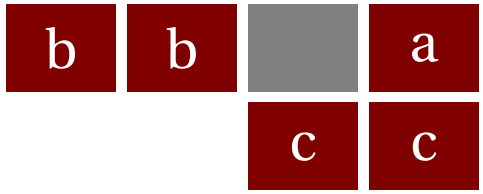
Rules:

- Each n-byte object must start on a multiple of n bytes (no unaligned accesses).
- Any object containing an n-byte object must be of size mn for some integer m (aligned even when arrayed).

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




```
struct padded {
  char a;  /* 1 byte  */
  short b; /* 2 bytes */
  short c; /* 2 bytes */
};
```



C's Type System: Enumerations

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

enum weekday day = mon;
```

Enumeration constants in the same scope must be unique:

```
enum days {sun, wed, sat};

enum class {mon, wed};  /* error: mon, wed redefined */
```

C's Type System

Types may be intermixed at will:

```
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.

Strongly-typed Languages

Strongly-typed: no run-time type clashes.

C is definitely not strongly-typed:

```
float g;

union { float f; int i } u;

u.i = 3;

g = u.f + 3.14159; /* u.f is meaningless */
```

Is Java strongly-typed?

Statically-Typed Languages

Statically-typed: compiler can determine types.

Dynamically-typed: types determined at run time.

Is Java statically-typed?

```
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:

```
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;
      }
}
```

Polymorphism

To sort doubles, only need to change two types:

```
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

```
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);
```

C++ Templates

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

```
sort<int>(a, 10);

sort<double>(b, 30);

sort<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 < 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;
      }
}
```

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)`

$\Rightarrow$ Array bounds must be known to access 2D+ arrays

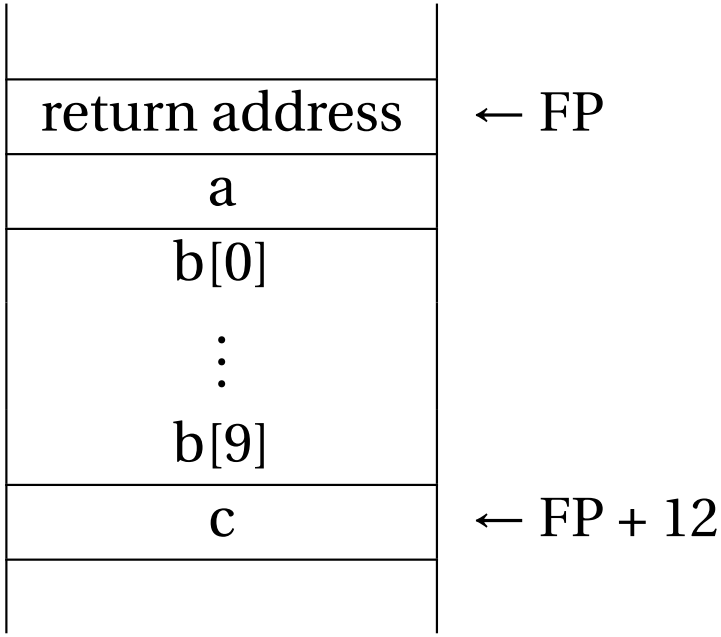
Allocating Arrays in C++

```
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.

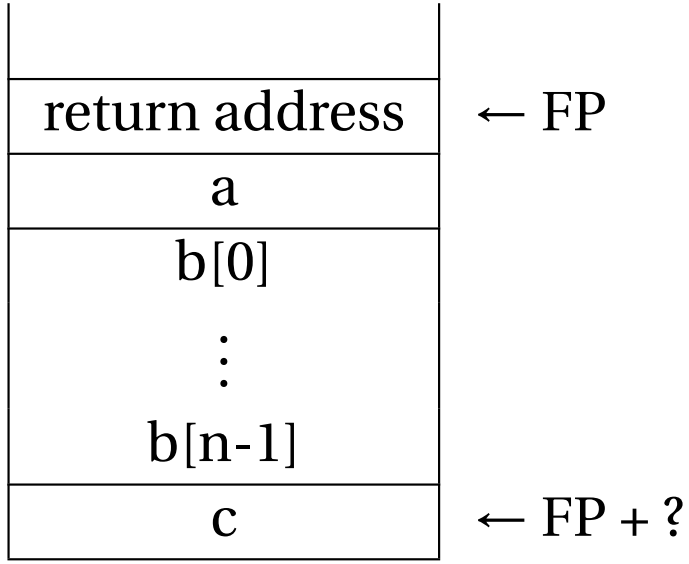
```
void foo()
{
  int a;
  int b[10];
  int c;
}
```



Allocating Variable-Sized Arrays

Variable-sized local arrays aren't as easy.

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

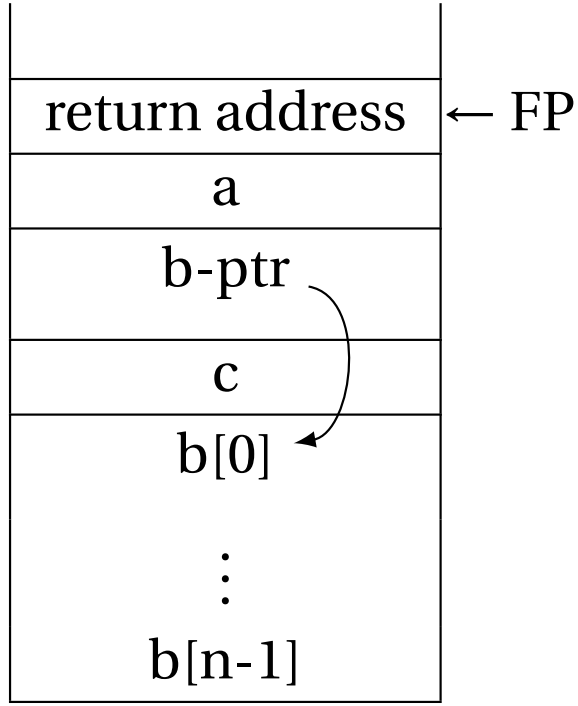


Doesn't work: generated code expects a fixed offset for c. Even worse for multi-dimensional arrays.

Allocating Variable-Sized Arrays

As always:
add a level of indirection

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



Variables remain constant offset from frame pointer.

Part II

Static Semantic Analysis

Static Semantic Analysis

Lexical analysis: Make sure tokens are valid

```
if i 3 "This"                    /* valid */
#a1123                           /* 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

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

struct b {
  int x, y;
} bar;

bar = foo;
```

Is this legal in C?

Name vs. Structural Equivalence

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

typedef struct f f_t;

f_t baz;

baz = foo;
```

Legal because `f_t` is an alias for `struct f`.

Things to Check

Make sure variables and functions are defined.

```
int i = 10;
int b = i[5]; /* Error: not an array */
```

Verify each expression's types are consistent.

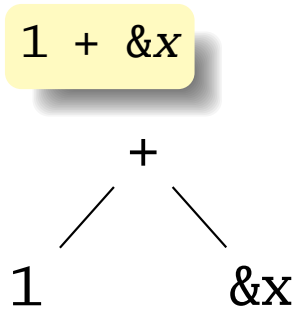
```
int i = 10;
char *j = "Hello";
int k = i * j; /* Error: bad operands */
```

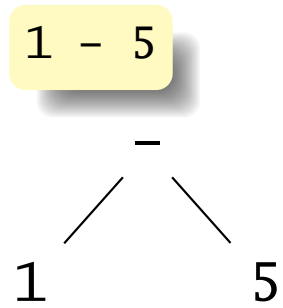
Things to Check

- 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 predicate of 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.






```
check(+)
  check(1) = int
  check(&x) = int *
  FAIL: int != int *
```

```
check(-)
  check(1) = int
  check(5) = int
  Types match, return int
```

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.

An Example of Static Semantics

A big function: “translate: ast → sast”

Converts a raw AST to a “semantically checked AST”

Names and types resolved

AST:

```
type expression =
    IntConst of int
  | Id of string
  | Call of string * expression list
  | ...
```

SAST:

```
type expr_detail =
   IntConst of int
| Id of variable_decl
| Call of function_decl * expression list
| ...

type expression = expr_detail * Type.t
```

The Type of Types

To do type checking, you need to represent the types of expressions.

An example for a language with integer, structures, arrays, and exceptions:

```
type t = (* can't call it "type" since that's reserved *)
    Void
  | Int
  | Struct of string * ((string * t) array) (* name, fields *)
  | Array of t * int                        (* type, size *)
  | Exception of string
```

Translation Environments

Whether an expression/statement/function is correct depends on its context. Represent this as an object with named fields since you will invariably have to extend it.

Here is the environment type for an elaborate C-like language:

```
type translation_environment = {
    return_type : Types.t;                (* Function's return type *)
    in_switch : bool;                  (* if we are in a switch stmt *)
    case_labels : Big_int.big_int list ref; (* known case labels *)
    break_label : label option;           (* when break makes sense *)
    continue_label : label option;  (* when continue makes sense *)
    scope : symbol_table;                  (* symbol table for vars *)
    exception_scope : exception_scope;  (*    " " for exceptions *)
    labels : label list ref;                (* labels on statements *)
    forward_gotos : label list ref; (* forward goto destinations *)
  }
```

A Symbol Table

```
type symbol_table = {
  parent : scope option;
  variables : variable_decl list
}

let rec find_variable (scope : symbol_table) name =
  try
     List.find (fun (s, _, _, _) -> s = name) scope.variables
  with Not_found ->
    match scope.parent with
      Some(parent) -> find_variable parent name
    | _ -> raise Not_found
```

Checking Expressions: Literals and Identifiers

```
(* Information about where we are *)
type translation_environment = {
   scope : symbol_table;
}

let rec expr env = function

    (* An integer constant: convert and return Int type *)
    Ast.IntConst(v) -> Sast.IntConst(v), Types.Int

    (* An identifier: verify it is in scope and return its type *)
  | Ast.Id(vname) ->
     let vdecl = try
       find_variable env.scope vname (* locate a variable by name *)
     with Not_found ->
        raise (Error("undeclared identifier " ^ vname))
     in
     let (_, typ) = vdecl in (* get the variable's type *)
     Sast.Id(vdecl), typ

  | ...
```

Checking Expressions: Binary Operators

```
(* let rec expr env = function *)

| A.BinOp(e1, op, e2) ->
   let e1 = expr env e1     (* Check left and right children *)
   and e2 = expr env e2 in

   let _, t1 = e1           (* Get the type of each child *)
   and _, t2 = e2 in

   if op <> Ast.Equal && op <> Ast.NotEqual then
     (* Most operators require both left and right to be integer *)
     (require_integer e1 "Left operand must be integer";
      require_integer e2 "Right operand must be integer")
   else
     if not (weak_eq_type t1 t2) then
       (* Equality operators just require types to be "close" *)
       error ("Type mismatch in comparison: left is " ^
              Printer.string_of_sast_type t1 ^ "\" right is \"" ^
              Printer.string_of_sast_type t2 ^ "\""
             ) loc;

   Sast.BinOp(e1, op, e2), Types.Int  (* Success: result is int *)
```

Checking Statements: Expressions, If

```
let rec stmt env = function

    (* Expression statement: just check the expression *)
    Ast.Expression(e) -> Sast.Expression(expr env e)

    (* If statement: verify the predicate is integer *)
  | Ast.If(e, s1, s2) ->

      let e = check_expr env e in  (* Check the predicate *)
      require_integer e "Predicate of if must be integer";

      Sast.If(e, stmt env s1, stmt env s2) (* Check then, else *)
```

Checking Statements: Declarations

```
(* let rec stmt env = function *)

| A.Local(vdecl) ->
    let decl, (init, _) = check_local vdecl (* already declared? *)
    in

   (* side-effect: add variable to the environment *)
    env.scope.S.variables <- decl :: env.scope.S.variables;

    init (* initialization statements, if any *)
```

Checking Statements: Blocks

```
(* let rec stmt env = function *)

| A.Block(sl) ->

    (* New scopes: parent is the existing scope, start out empty *)

    let scope' = { S.parent = Some(env.scope); S.variables = [] }
    and exceptions' =
      { excep_parent = Some(env.exception_scope); exceptions = [] }
    in

    (* New environment: same, but with new symbol tables *)
    let env' = { env with scope = scope';
                          exception_scope = exceptions' } in

    (* Check all the statements in the block *)

    let sl = List.map (fun s -> stmt env' s) sl in
    scope'.S.variables <-
       List.rev scope'.S.variables; (* side-effect *)

    Sast.Block(scope', sl) (* Success: return block with symbols *)
```