# Types and Static Semantic Analysis

Stephen A. Edwards

Columbia University

Fall 2014

# Types

# Types

*A restriction on the possible interpretations of a segment of memory or other program construct.*

Two uses:

**Safety:** avoids data being treated as something it isn’t

**Optimization:** eliminates certain runtime decisions

# Types of Types

| Type | Examples |
|---|---|
| Basic | Machine words, floating-point numbers, addresses/pointers |
| Aggregate | Arrays, structs, classes |
| Function | Function pointers, lambdas |

# Basic Types

Groups of data the processor is designed to operate on.

On an ARM processor,

| Type | Width (bits) |
|---|---|
| **Unsigned/two's-complement binary** | |
| Byte | 8 |
| Halfword | 16 |
| Word | 32 |
| **IEEE 754 Floating Point** | |
| Single-Precision scalars & vectors | 32, 64, .., 256 |
| Double-Precision scalars & vectors | 64, 128, 192, 256 |

# Derived types

**Array**: a list of objects of the same type, often fixed-length

**Record**: a collection of named fields, often of different types

**Pointer/References**: a reference to another object

**Function**: a reference to a block of code

# C’s Declarations and Declarators

Declaration: list of specifiers followed by a comma-separated list of declarators.

basic type

```c
static unsigned int (*f[10])(int, char*);
```

specifiers declarator

Declarator’s notation matches that of an expression: use it to return the basic type.

Largely regarded as the worst syntactic aspect of C: both pre- (pointers) and post-fix operators (arrays, functions).

# C Declarations: specifiers + initializing declarators

*declaration*
    *declaration−specifiers init−declarator−list*$_{opt}$ ;   # int a = 3, b;

*declaration−specifiers*   # List of specifiers
    *storage−class−specifier declaration−specifiers*$_{opt}$   # static, typedef
    *type−specifier declaration−specifiers*$_{opt}$   # int, struct
    *type−qualifier declaration−specifiers*$_{opt}$   # const, volatile

*init−declarator−list*   # Comma−separated list of new names
    *init−declarator*
    *init−declarator−list* , *init−declarator*

*init−declarator*   # A new name given a type and optional initial value
    *declarator*
    *declarator* = *initializer*

```
int a, b[10],  /* "a" is an integer; "b" is an array */
    *c;        /* "c" is a pointer */
static const char d = 'b', /* initialized static constant character */
                  e[5] = { 0, 8, 12, 34, 1 };
```

# Storage Classes, Type Specifiers, and Type Qualifiers

```
storage−class−specifier # Where to put the object
    typedef             # Name a type instead of an object
    extern              # Defined elsewhere; linked in
    static              # Not on stack/restricted scope
    auto                # On stack: default
    register            # In a register: ignored
type−specifier                     # What the object can hold
    void                           # For functions that return nothing
    char                           # Character                8 bits
    short                          # Short integer           16 bits
    int                            # Machine word (default)  32 bits
    long                           # Longer                  64 bits
    float                          # Single−precision FP     32 bits
    double                         # Double−precision FP     64 bits
    signed                         # Allows negative numbers: default
    unsigned                       # Never negative
    struct−or−union−specifier      # Objects with multiple fields
    enum−specifier                 # Objects that hold names
    typedef−name                   # A user−defined type (an identifier)
type−qualifier          # How to treat data in the object
    const               # May not be modified after creation
    volatile            # Do not optimize accesses
```

# C Declarations: Structs and Unions

*struct–or–union–specifier*
    *struct–or–union identifier*$_{opt}$ { *struct–declaration–list* } # New struct
    *struct–or–union identifier* # Refer to an existing one

*struct–or–union*
    **struct** # Enough storage for every field
    **union** # Enough storage for largest field only

*struct–declaration–list* # List of named fields with types
    *struct–declaration–list*$_{opt}$ *struct–declaration*

*struct–declaration* # Field declarations: name and type, no init
    *specifier–qualifier–list struct–declarator–list* ;

```
struct { int x, y; } a; /* "a" is a struct with fields x and y */
struct foo { int w;     /* declare struct foo, fields w and z */
             char z; }; /* no storage requested (no declarator) */
struct foo c;           /* "c" holds a struct foo */
```

# C Declarations: Structs and Unions

*specifier−qualifier−list* # Note: no extern, static, etc.
*type−specifier specifier−qualifier−list*$_{opt}$ # int, struct
*type−qualifier specifier−qualifier−list*$_{opt}$ # const, volatile

*struct−declarator−list* # Comma−separated list of field names
*struct−declarator*
*struct−declarator−list* , *struct−declarator*

*struct−declarator*
*declarator* # Named field
*declarator*$_{opt}$ : *constant−expression* # Named field with bit width

```
struct foo {
  unsigned int c : 3, d : 2; /* c is 3 bits; d is 2 */
  unsigned int a;            /* a is word length */
  double f;                  /* field f: double-precision */
  struct foo *fptr;          /* pointer to a struct foo */
};
```

# Structs

Structs 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);
```

# Unions: Variant Records

A `struct` holds all of its fields at once. A `union` holds only one of its fields at any time (the last written).

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

union token t;
t.i = 10;
t.f = 3.14159;        /* overwrite t.i */
char *s = t.string;   /* return 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`.

# 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? Should it be?

# C Declarations: Enums

*enum−specifier*

  **enum** *identifier*$_{opt}$ { *enumerator−list* }

  **enum** *identifier*

*enumerator−list*

  *enumerator*

  *enumerator−list* , *enumerator*

*enumerator*

  *enumeration−constant*

  *enumeration−constant* = *constant−expression*

*enumeration−constant*

  *identifier*

Enumeration constants in the same scope must be distinct; values need not be.

```
enum foo { A = 5, B, C = 3, D, E }; /* New enum, no storage */
enum foo a;                         /* a holds A, B, C, etc. */
enum { F = 42, G = 5 } b;           /* b holds F, G */
```

# C Declarations: Declarators

*declarator*
  *pointer$_{opt}$ direct−declarator*

*direct−declarator*
  *identifier* # name to define
  ( *declarator* ) # override precedence
  *direct−declarator* [ *constant−expression$_{opt}$* ] # array
  *direct−declarator* ( *parameter−type−list$_{opt}$* ) # function (typed args)
  *direct−declarator* ( *identifier−list $_{opt}$* ) # old−style function (names)

*pointer*
  ∗ *type−qualifier−list$_{opt}$* # e.g., ∗a, ∗const b
  ∗ *type−qualifier−list$_{opt}$ pointer* # e.g., ∗const ∗c

*type−qualifier−list*
  *type−qualifier−list$_{opt}$ type−qualifier* # const, volatile

```
int a[5];           /* array of 5 integers */
int *b[6];          /* array of 6 integer pointers */
int (*c)[6];        /* pointer to array of 6 integers */
int f(int, float);  /* f: function of two arguments returning int */
int *g(int);        /* g: function returning a pointer to an integer */
int (*h)(int);      /* h: pointer to a function returning an integer */
```

# C Declarations: Formal Function Arguments

*parameter−type−list*
  *parameter−list*
  *parameter−list* , ...   # Ellipses : variable number of arguments after this

*parameter−list*   # Comma−separated list of parameters
  *parameter−declaration*
  *parameter−list* , *parameter−declaration*

*parameter−declaration*
  *declaration−specifiers declarator*   # argument with name
  *declaration−specifiers abstract−declarator*$_{opt}$   # argument type only

```
int f( int (*)(int, float) ); /* argument is function pointer */
int g( char c );              /* argument given a name */
```

# Type Expressions

C's declarators are unusual: they always specify a name along with its type.

Languages more often have *type expressions*: a grammar for expressing a type.

Type expressions appear in three places in C:

```
(int *) a                          /* Type casts */
sizeof(float [10])                 /* Argument of sizeof() */
int f(int, char *, int (*)(int))   /* Function argument types */
```

# C's Type Expressions

*type−name* # e.g., int, int *, const unsigned char (*)(int, float [])
  *specifier−qualifier−list abstract−declarator*$_{opt}$

*specifier−qualifier−list* # Note: no extern, static, etc.
  *type−specifier specifier−qualifier−list*$_{opt}$ # int, struct
  *type−qualifier specifier−qualifier−list*$_{opt}$ # const, volatile

*abstract−declarator* # Declarator that does not define a name
  *pointer*
  *pointer*$_{opt}$ *direct−abstract−declarator*

*direct−abstract−declarator*
  ( *abstract−declarator* ) # override precedence
  *direct−abstract−declarator*$_{opt}$ [ *constant−expression*$_{opt}$ ] # array
  *direct−abstract−declarator*$_{opt}$ ( *parameter−type−list*$_{opt}$ ) # function

# Representing Declarators and Type Expressions

Simplified from the AST of CIL, a C front end in OCaml:

```
type typeSpecifier =
    Tvoid | Tchar | Tshort | Tint | Tlong | Tfloat | Tdouble
  | Tnamed of string
  | Tstruct of string * field_group list option
  | Tunion of string * field_group list option
  | Tenum of string * enum_item list option
and cvspec = CV_CONST | CV_VOLATILE
and storage = NO_STORAGE | AUTO | STATIC | EXTERN | REGISTER
type spec_elem =      (* A single type specifier *)
    SpecTypedef
  | SpecCV of cvspec
  | SpecStorage of storage
  | SpecType of typeSpecifier

type decl_type =      (* A declarator *)
 | JUSTBASE
 | ARRAY of decl_type * expression
 | PTR of decl_type
 | PROTO of decl_type * single_name list
and name = string * decl_type             (* declarator with type *)
and single_name = specifier * name
and name_group = spec_elem list * name list (* int a, *b *)
```

# Semantic Checking: Static vs. Dynamic

Consider the C assignment statement

```
b = a;
```

What makes this assignment valid? What would make it invalid?

When are these conditions checked? When the program is compiled or when it is running?

# Static Semantic Analysis

# Static Semantic Analysis

Lexical analysis: Make sure tokens are valid

```
if i 3 "This"                     /* valid Java tokens */
#a1123                            /* not a token */
```

Syntactic analysis: Makes sure tokens appear in correct order

```
for ( i = 1 ; i < 5 ; i++ ) 3 + "foo";  /* valid Java syntax */
for break                               /* invalid syntax */
```

Semantic analysis: Makes sure program is consistent

```
int v = 42 + 13;      /* valid in Java (if v is new) */
return f + f(3);      /* invalid */
```

# What To Check

Examples from Java:

Verify names are defined and are of the right type.

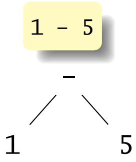
```
int i = 5;
int a = z;    /* Error: cannot find symbol */
int b = i[3]; /* Error: array required, but int found */
```

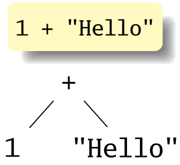
Verify the type of each expression is consistent.

```
int j = i + 53;
int k = 3 + "hello";   /* Error: incompatible types */
int l = k(42);         /* Error: k is not a method */
if ("Hello") return 5; /* Error: incompatible types */
String s = "Hello";
int m = s;             /* Error: incompatible types */
```

# How To Check: Depth-first AST Walk

Checking function: environment → node → type



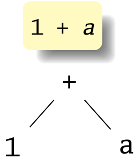

check(-)
- check(1) = int
- check(5) = int
- Success: int – int = int

check(+)
- check(1) = int
- check("Hello") = string
- FAIL: Can’t add int and string

Ask yourself: at each kind of node, what must be true about the nodes below it? What is the type of the node?

# How To Check: Symbols

Checking function: environment → node → type

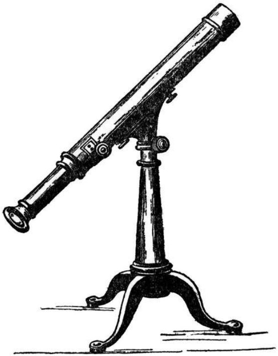



check(+)
  check(1) = int
  check(a) = int
  Success: int + int = int

The key operation: determining the type of a symbol when it is encountered.

The environment provides a “symbol table” that holds information about each in-scope symbol.

# Scope



# Basic Static Scope in C, C++, Java, etc.

A name begins life where it is declared and ends at the end of its block.

From the CLRM, "The scope of an identifier declared at the head of a block begins at the end of its declarator, and persists to the end of the block."

```
void foo()
{
   int x;



}
```

# Hiding a Definition

Nested scopes can hide earlier definitions, giving a hole.

From the CLRM, “If an identifier is explicitly declared at the head of a block, including the block constituting a function, any declaration of the identifier outside the block is suspended until the end of the block.”

```
void foo()
{
  int x;

  while ( a < 10 ) {
    int x;

  }


}
```

# Static Scoping in Java

```
public void example() {
  // x, y, z not visible

  int x;
  // x visible

  for ( int y = 1 ; y < 10 ; y++ ) {
    // x, y visible

    int z;
    // x, y, z visible
  }

  // x visible
}
```

# Basic Static Scope in O'Caml

A name is bound after the "in" clause of a "let." If the name is re-bound, the binding takes effect *after* the "in."

```
let x = 8 in


let x = x + 1 in
```

Returns the pair (12, 8):

```
let x = 8 in
  (let x = x + 2 in
     x + 2),
x
```

# Let Rec in O'Caml

The "rec" keyword makes a name visible to its definition. This only makes sense for functions.

```
let rec fib i =
  if i < 1 then 1 else
    fib (i-1) + fib (i-2)
in
  fib 5
```

```
(* Nonsensical *)
let rec x = x + 3 in
```

# Let...and in O'Caml

Let...and lets you bind multiple names at once. Definitions are not mutually visible unless marked "rec."

```
let x = 8
and y = 9 in
```

```
let rec fac n =
     if n < 2 then
       1
     else
       n * fac1 n
and fac1 n = fac (n - 1)
in
fac 5
```

# Nesting Function Definitions

```
let articles words =

  let report w =

    let count = List.length
      (List.filter ((=) w) words)
    in w ^ ": " ^
       string_of_int count

  in String.concat ", "
    (List.map report ["a"; "the"])

in articles
    ["the"; "plt"; "class"; "is";
     "a"; "pain"; "in";
     "the"; "butt"]
```

Produces “a: 1, the: 2”

```
let count words w = List.length
  (List.filter ((=) w) words) in

let report words w = w ^ ": " ^
  string_of_int (count words w) in

let articles words =
  String.concat ", "
    (List.map (report words)
     ["a"; "the"]) in

articles
    ["the"; "plt"; "class"; "is";
     "a"; "pain"; "in";
     "the"; "butt"]
```

# A Static Semantic Analyzer

# The Static Semantic Checking Function

A big function: “check: 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

Need an OCaml type to represent the type of something in your language.

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.

An environment type for a C-like language:

```
type translation_environment = {
    scope : symbol_table;                      (* symbol table for vars *)

    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 *)
    exception_scope : exception_scope;  (* sym tab for exceptions *)
    labels : label list ref;                   (* labels on statements *)
    forward_gotos : label list ref; (* forward goto destinations *)
  }
```

# A Symbol Table

Basic operation is string → type. Map or hash could do this, but a list is fine.

```
type symbol_table = {
  parent : symbol_table 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 *)
```