Types and Static Semantic Analysis

Stephen A. Edwards

Columbia University

Spring 2016
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 in C
Basic C Types

C was designed for efficiency: basic types are whatever is most efficient for the target processor.

On an (32-bit) ARM processor,

```c
char c;  /* 8-bit binary */
short d; /* 16-bit two's-complement binary */
unsigned short d; /* 16-bit binary */
int a;    /* 32-bit two’s-complement binary */
unsigned int b; /* 32-bit binary */
float f;  /* 32-bit IEEE 754 floating-point */
double g; /* 64-bit IEEE 754 floating-point */
```
Pointers and Arrays

A pointer contains a memory address.

Arrays in C are implemented with arithmetic on pointers.

A pointer can create an *alias* to a variable:

```c
int a;
int *b = &a;  /* "pointer to integer b is the address of a" */
int *c = &a;  /* c also points to a */

*b = 5;      /* sets a to 5 */
*c = 42;     /* sets a to 42 */

printf("%d %d %d\n", a, *b, *c);  /* prints 42 42 42 */
```
Pointers Enable Pass-by-Reference

```c
void swap(int x, int y)
{
    int temp;
    temp = x;
    x = y;
    y = temp;
}

Does this work?
```
Pointers Enable Pass-by-Reference

void swap(int x, int y)
{
    int temp;
    temp = x;
    x = y;
    y = temp;
}

Does this work? Nope.

void swap(int *px, int *py)
{
    int temp;

    temp = *px; /* get data at px */
    *px = *py; /* get data at py */
    *py = temp; /* write data at py */
}

void main()
{
    int a = 1, b = 2;

    /* Pass addresses of a and b */
    swap(&a, &b);

    /* a = 2 and b = 1 */
}
Arrays and Pointers

```c
int a[10];
```

![Array and Pointers Diagram](image)

The diagram shows an array `a` with elements `a[0]` to `a[9]`. The code snippet initializes an array `a[10]` and uses pointers to access elements within the array.
Arrays and Pointers

int a[10];
int *pa = &a[0];
Arrays and Pointers

```c
int a[10];
int *pa = &a[0];
pa = pa + 1;
```
Arrays and Pointers

```c
int a[10];
int *pa = &a[0];
pa = pa + 1;
pa = &a[1];
```
Arrays and Pointers

int a[10];
int *pa = &a[0];
pa = pa + 1;
pa = &a[1];
pa = a + 5;

a[i] is equivalent to *(a + i)
Multi-Dimensional Arrays

```c
int monthdays[2][12] = {
    { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 },
    { 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 }
};
```

*monthdays[i][j]* is at address *monthdays + 12 * i + j*
Structures

Structures: each field has own storage

```c
struct box {
    int x, y, h, w;
    char *name;
};
```

Unions: fields share same memory

```c
union token {
    int i;
    double d;
    char *s;
};
```
Structs can be used like the objects of C++, Java, et al.
Group and restrict what can be stored in an object, but not what operations they permit.

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

```c
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:

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

void draw(struct poly *shape) {
    switch (shape->type) {
    case CIRCLE: /* use shape->d.radius */
    case SQUARE: /* use shape->d.size */
    case LINE: /* use shape->d.angle */
    }
}
```
Name vs. Structural Equivalence

```c
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’s Declarations and Declarators

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

\[
\text{basic type}\]
\[
\text{static unsigned int} (*f[10])(\text{int, char*});
\]

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).
Types of Type Systems
Strongly-typed Languages

Strongly-typed: no run-time type clashes (detected or not).

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?
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;
            }
}
```
Polymorphism

To sort doubles, only need to change two 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;
            }
}
```
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 are essentially language-aware macros. Each instance generates a different refinement of the same code.

```c++
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;
            }
}
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.
Static Semantic Analysis
Static Semantic Analysis

Lexical analysis: Make sure tokens are valid

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

Syntactic analysis: Makes sure tokens appear in correct order

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

Semantic analysis: Makes sure program is consistent

```java
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.

```java
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.

```java
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

1 - 5
   \  
   1  5

1 + "Hello"
   \  
   +
   1   "Hello"

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

1 + a

+  
1   a

determine the type of a symbol when

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.”

```c
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.”

```c
void foo()
{
    int x;
    while ( a < 10 ) {
        int x;
    }
}
```
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.

```ocaml
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 lets you bind multiple names at once. Definitions are not mutually visible unless marked “rec.”

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

let rec fac n =
  if n < 2 then
    1
  else
    n * fac (n - 1)

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"]

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"]
```

Produces “a: 1, the: 2″
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 type expression =
    expr_detail * Type.t

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

```haskell
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 *)
}
```
Basic operation is string → type. Map or hash could do this, but a list is fine.

```ocaml
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 *)

```ocaml
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 ^ "\"")

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