Scope

Types

Types in C

Types of Type Systems

Overloading

Binding Time

Static Semantic Analysis
What’s Wrong With This?

\[ a + f(b, c) \]
What’s Wrong With This?

\[ a + f(b, c) \]

Is \( a \) defined?

Is \( f \) defined?

Are \( b \) and \( c \) defined?

Is \( f \) a function of two arguments?

Can you add whatever \( a \) is to whatever \( f \) returns?

Does \( f \) accept whatever \( b \) and \( c \) are?

Scope questions      Type questions
Scope

What names are visible?
Scope: where/when a name is bound to an object
Useful for modularity: want to keep most things hidden

<table>
<thead>
<tr>
<th>Scoping Policy</th>
<th>Visible Names Depend On</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static</td>
<td>Textual structure of program</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Run-time behavior of program</td>
</tr>
</tbody>
</table>
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; // This is a local variable.
    while ( a < 10 ) {
        int x; // This is a local variable.
    }
}
```
Static vs. Dynamic Scope

C

```c
int a = 0;

int foo() {
    return a + 1;
}

int bar() {
    int a = 10;
    return foo();
}
```

OCaml

```ocaml
let a = 0 in
let foo x = a + 1 in
let bar =
    let a = 10 in
    foo 0
```

Bash

```bash
a=0

foo ()
{
    a='expr $a + 1'
}

bar ()
{
    local a=10
    foo
    echo $a
}

bar
```
A name is bound after the “in” clause of a “let.” If the name is re-bound, the binding takes effect after the “in.”

Returns the pair (12, 8):

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

```
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 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
```
Languages such as C, C++, and Pascal require *forward declarations* for mutually-recursive references.

```c
int foo(void);
int bar() { ... foo(); ... }
int foo() { ... bar(); ... }
```

Nesting Function Definitions

```ocaml
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 String.concat "", "
    (List.map report words)
  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)
  in articles
    ["the"; "plt"; "class"; "is"; "a"; "pain"; "in"; "the"; "butt"]

Produces “a: 1, the: 2”
Dynamic Definitions in \TeX

\% \x, \y undefined
{
\% \x, \y undefined
\texttt{def} \x 1
\% \x defined, \y undefined

\texttt{ifnum} \a < 5
  \texttt{def} \y 2
\texttt{fi}

\% \x defined, \y may be undefined
}
\% \x, \y undefined
Most modern languages use static scoping.
Easier to understand, harder to break programs.
Advantage of dynamic scoping: ability to change environment.
A way to surreptitiously pass additional parameters.
program messages;
var message : string;

procedure complain;
begin
  writeln(message);
end

procedure problem1;
var message : string;
begin
  message := 'Out of memory';
  complain
end

procedure problem2;
var message : string;
begin
  message := 'Out of time';
  complain
end
Open vs. Closed Scopes

An open scope begins life including the symbols in its outer scope.

Example: blocks in Java

```java
{
    int x;
    for (;;) {
        /* x visible here */
    }
}
```

A closed scope begins life devoid of symbols.

Example: structures in C.

```c
struct foo {
    int x;
    float y;
}
```
Types

What operations are allowed?
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

What types are processors best at?
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 */
```
Number Behavior

Basic number axioms:

\[ a + x = a \text{ if and only if } x = 0 \]  
Additive identity

\[ (a + b) + c = a + (b + c) \]  
Associative

\[ a(b + c) = ab + ac \]  
Distributive
Misbehaving Floating-Point Numbers

\[ 1 \times 10^{20} + 1 \times 10^{-20} = 1 \times 10^{20} \]

\[ 1 \times 10^{-20} \ll 1 \times 10^{20} \]

\[ (1 + 9 \times 10^{-7}) + 9 \times 10^{-7} \neq 1 + (9 \times 10^{-7} + 9 \times 10^{-7}) \]

\[ 9 \times 10^{-7} \ll 1, \text{ so it is discarded, however, } 1.8 \times 10^{-6} \text{ is large enough} \]

\[ 1.00001(1.000001 − 1) \neq 1.00001 \cdot 1.000001 − 1.00001 \cdot 1 \]

\[ 1.00001 \cdot 1.000001 = 1.00001100001 \text{ requires too much intermediate precision.} \]
What’s Going On?

Floating-point numbers are represented using an exponent/significand format:

\[
\begin{align*}
\text{1} & \quad 10000001 & \quad 01100000000000000000000 \\
\text{8-bit exponent} & \quad \text{23-bit significand} \\
\text{\phantom{1}} = & \quad -1.011_2 \times 2^{129-127} = -1.375 \times 4 = -5.5.
\end{align*}
\]

What to remember:

\[1363.456846353963456293\]

represented \quad rounded
Results are often rounded:

$$
\begin{array}{c}
1.0000100000 \\
\times 1.0000100000 \\
\hline
1.00001100001 \\
\end{array}
$$

rounded

When $b \approx -c$, $b + c$ is small, so $ab + ac \neq a(b + c)$ because precision is lost when $ab$ is calculated.

Moral: Be aware of floating-point number properties when writing complex expressions.
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

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

Does this work?

Nope.

```c
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

int a[10];

Arrays and Pointers

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

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

```
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: 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.

```
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).

```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 */
```

Kind of like a bathroom on an airplane
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.

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

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

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).
Types of Type Systems

What kinds of type systems do languages have?
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?

class Foo {
    public void x() { ... }
}
class Bar extends Foo {
    public void x() { ... }
}

void baz(Foo f) {
    f.x();
}
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;
            }
}
```
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 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.
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
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.
Parametric Polymorphism

In C++,

```cpp
template<typename T>
T max(T x, T y)
{
    return x > y ? x : y;
}

struct foo {int a;} f1, f2, f3;

int main()
{
    int a = max<int>(3, 4); /* OK */
    f3 = max<struct foo>(f1, f2); /* No match for operator> */
}
```

The `max` function only operates with types for which the `>` operator is defined.
**Parametric Polymorphism**

In OCaml,

```ocaml
let max x y = if x - y > 0 then x else y
max : int -> int -> int
```

Only `int` arguments are allowed because in OCaml, only operates on integers.

However,

```ocaml
let rec map f = function [] -> [] | x::xs -> f x :: map f xs
map : ('a -> 'b) -> 'a list -> 'b list
```

Here, ’a and ’b may each be any type.

OCaml uses parametric polymorphism: type variables may be of any type.

C++‘s template-based polymorphism is ad hoc: there are implicit constraints on type parameters.
Overloading

What if there is more than one object for a name?
Overloading versus Aliases

Overloading: two objects, one name

Alias: one object, two names

In C++,

```c++
int foo(int x) { ... }
int foo(float x) { ... }  // foo overloaded

void bar()
{
    int x, *y;
    y = &x;   // Two names for x: x and *y
}
```
Examples of Overloading

Most languages overload arithmetic operators:

```
1 + 2       // Integer operation
3.1415 + 3e-4 // Floating-point operation
```

Resolved by checking the *type* of the operands.
Context must provide enough hints to resolve the ambiguity.
C++ and Java allow functions/methods to be overloaded.

```
int foo();
int foo(int a); // OK: different # of args
float foo();   // Error: only return type
int foo(float a); // OK: different arg types
```

Useful when doing the same thing many different ways:

```
int add(int a, int b);
float add(float a, float b);

void print(int a);
void print(float a);
void print(char *s);
```
Function Overloading in C++

Complex rules because of *promotions*:

```
int i;
long int l;
l + i
```

Integer promoted to long integer to do addition.

```
3.14159 + 2
```

Integer is promoted to double; addition is done as double.
Function Overloading in C++

1. Match trying trivial conversions
   int a[] to int *a, T to const T, etc.

2. Match trying promotions
   bool to int, float to double, etc.

3. Match using standard conversions
   int to double, double to int

4. Match using user-defined conversions
   operator int() const { return v; }

5. Match using the elipsis ...

Two matches at the same (lowest) level is ambiguous.
Binding Time

When are bindings created and destroyed?
### Binding Time

When a name is connected to an object.

<table>
<thead>
<tr>
<th>Bound when</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>language designed</td>
<td>if else</td>
</tr>
<tr>
<td>language implemented</td>
<td>data widths</td>
</tr>
<tr>
<td>Program written</td>
<td>foo bar</td>
</tr>
<tr>
<td>compiled</td>
<td>static addresses, code</td>
</tr>
<tr>
<td>linked</td>
<td>relative addresses</td>
</tr>
<tr>
<td>loaded</td>
<td>shared objects</td>
</tr>
<tr>
<td>run</td>
<td>heap-allocated objects</td>
</tr>
</tbody>
</table>
Binding Time and Efficiency

Earlier binding time ⇒ more efficiency, less flexibility

Compiled code more efficient than interpreted because most decisions about what to execute made beforehand.

```c
switch (statement) {
  case add:
    r = a + b;
    break;
  case sub:
    r = a - b;
    break;
  /* ... */
}
```

```
ad %o1, %o2, %o3
```
Dynamic method dispatch in OO languages:

class Box : Shape {
    public void draw() { ... }
}

class Circle : Shape {
    public void draw() { ... }
}

Shape s;
s.draw(); /* Bound at run time */
Interpreters better if language has the ability to create new programs on-the-fly.

Example: Ousterhout’s Tcl language.

Scripting language originally interpreted, later byte-compiled.

Everything’s a string.

```
set a 1
set b 2
puts "$a + $b = [expr $a + $b]"
```
Tcl’s eval runs its argument as a command.

Can be used to build new control structures.

```tcl
proc iff forall {list pred ifstmt} {
    foreach i $list {
        if [expr $pred] {
            eval $ifstmt
        }
    }
}

iff forall {0 1 2} { $i % 2 == 0 } {
    puts "$i even"
}

0 even
2 even
```
Static Semantic Analysis

How do we validate names, scope, and types?
Static Semantic Analysis

Lexical analysis: Each token is valid?

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

Syntactic analysis: Tokens appear in the correct order?

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

Semantic analysis: Names used correctly? Types 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 Expressions: 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

\[
\begin{array}{c}
1 + a \\
+ \\
1 & a
\end{array}
\]

\[
\begin{align*}
\text{check}(+) \\
\text{check}(1) &= \text{int} \\
\text{check}(a) &= \text{int} \\
\text{Success: } \text{int} + \text{int} &= \text{int}
\end{align*}
\]

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.
A Static Semantic Checking Function

A big function: “check: ast → sast”

Converts a raw AST to a “semantically checked AST”

Names and types resolved

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

AST:

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

SAST:

```
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:

```ocaml
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:

```plaintext
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.

```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 ^ "\""
               ) loc;

  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 *)