Chapter 9 — Semantic Analysis

1 Identifying issues

Semantic issue examples : variable read before initialization, duplicate labels of a switch, reassignment of constants, exhaustiveness of pattern matching, visibility of an invoked method, type mismatches, incorrect variable usage, incorrect function usage, logic mistakes.
Semantic warning examples : implicit type conversions, unused variables or functions, uninitialized variables, possible null or uninitialized pointer dereference, unused function parameters, unreachable code.

1.1 Name analysis

Names **identify** declarations of entities and allow for **references** to its declaration.s according to semantics, also implementing a **symbol table** for storing pointers to declarations, the scope, the types, and bindings.
Name analysis typically involves name resolution, scope determination, conflict resolution.

1.1.1 Identifying undeclared variables

```
check :: Expr -> [Expr]
check expr = c expr []
  where
    c :: Expr -> [String] -> [Expr]
    c (Cst _) _ = []
    c (Var v) env = [Var v | v `notElem` env]
    c (Bin e1 _ e2) env = c e1 env ++ c e2 env
    c (Let x e1 e2) env = c e1 env ++ c e2 (x:env)
```

1.1.2 Scoping

A scope defined where something can be referenced and manipulated. Types includes : global scope, local scope, class scope, function scope, block scope.
Nested scopes are enclosed scopes with access to upper levels, but inner names **shadow** upper names.

2 Type checking systems

Type systems provide type checking, type inference, abstractions, documentation, and tooling. It classifies values into types that can interact with each other and how its misuses can be reported.

2.1 Typing rules

To show that an expression of the form $e_1 + e_2$ has type int, we need to show that e_1 and e_2 have type int.

$$\textsc{BinOp}\ \frac{\vdash e_1 : \text{int} \quad \vdash e_2 : \text{int}}{\vdash e_1 + e_2 : \text{int}} \qquad \textsc{LetIn}\ \frac{\Gamma \vdash e_1 : T_1 \quad \Gamma[x \mapsto T_1] \vdash e_2 : T_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : T_2}$$

2.1.1 Environment

$\Gamma(x) = T$ is a list of key–Type pairs, asserting a variable has a type and allowing to type check with variables.
A binding $[x \mapsto T]$

2.1.2 Derivation tree

Show that $2 * (z + 1)$ is well-typed using :

$$\textsc{Lit}\ \frac{}{\Gamma \vdash i : \text{int}} \quad \textsc{Ident}\ \frac{\Gamma(x) = T}{\Gamma \vdash y : T} \quad \textsc{Add}\ \frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}} \quad \textsc{Prod}\ \frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 * e_2 : \text{int}}$$

$$\textsc{Prod}\ \dfrac{\textsc{Lit}\ \dfrac{}{\Gamma \vdash 2 : \text{int}} \qquad \textsc{Add}\ \dfrac{\textsc{Ident}\ \dfrac{\Gamma(z) = \text{int}}{\Gamma \vdash z : \text{int}} \qquad \textsc{Lit}\ \dfrac{}{\Gamma \vdash 1 : \text{int}}}{\Gamma \vdash z + 1 : \text{int}}}{\Gamma \vdash 2 * (z + 1) : \text{int}}$$

2.2 Type checking

```
data Type = ...
data Expr = ... | Let String Expr Expr
type Env = [(String, Type)]
typecheck :: Expr -> Env -> Type
typecheck (Let x e1 e2) env = t2
    where t1 = typecheck e1 env
          t2 = typecheck e2 ((x,t1):env)
typecheck _ _ = ...
```

Haskell implementation for the LetIn type rule.

2.3 Type inference – Algorithm W

```
Algo. of Hindley-Milner (f x = x+1)
Step 1 : Assign type variable
Type(f) = a | Type(x) = b
Step 2 : Generate type constraints
a = c -> d | d = type(x + 1)
b = c <=> b = type(1)
Step 3 : Unify type constraints
b = Int = c | d = Int | a = Int -> Int
Step 4 : Generalize return type
f :: Int -> Int <=> f :: Num a => a -> a
```

Chapter 10 — Interpreters

1 AST evaluation

Expressions are **evaluated** (give a value). Statements are **executed** (update the key–Value environment).

1.1 REPL

```
data Token = Digit Int | Plus
data Expr = Number Int | Sum Expr Expr
tokenize :: String -> [Token]
tokenize ('+':xs) = Plus : tokenize xs
tokenize (x:xs)
  | isDigit x = Digit (read [x]) : tokenize xs
  | isSpace x = tokenize xs
  | otherwise = error "Unexpected token !"

parse :: [Token] -> Expr
parse tokens = parseSum tokens
  where
    parseSum xs = case parseNum xs of
      (n, []) -> n
      (n, Plus : rest) -> Sum n (parseSum rest)
      _ -> error "Parsing error !"
    parseNum (Digit n : rest) = (Number n, rest)
    parseNum _ = error "Parsing error !"

eval :: Expr -> Int
eval (Number n) = n
eval (Sum x y) = eval x + eval y

repl :: IO ()
repl = do
  putStr "repl> "; userInput <- getLine
  putStrLn $ show $ eval $ parse $ tokenize userInput; repl
```

1.2 Expression evaluations

Evaluate arithmetic and logical expressions.

```
data Expr = Const Int | Binary Expr Char Expr
data Value = Number Int | Boolean Bool
eval :: Expr -> Value
eval (Const val) = Number val
eval (Binary op e1 op e2) =
  case (lhs, op, rhs) of
    (Number x, '+', Number y) -> Number (x + y)
    (Number x, '>', Number y) -> Boolean (x > y)
    _ -> error "unsupported operation"
  where (lhs, rhs) = (eval e1, eval e2)
```

1.3 Statements and state

```
type Env = Map String Value
data State = State { globals :: Env, locals :: Env }
data Stmt = ... | Block [Stmt]
exec :: Stmt -> State -> State
exec (Block ss) state = foldl (\st s -> exec s st) state ss
```

1.4 Control flow

```
data Expr  = ... | Const Bool | And Expr Expr | Or Expr Expr
data Stmt  = ... | DoWhile Expr Stmt
data Value = ... | Bool Boolean
eval :: Expr -> State -> Value
eval (Const val) = val                - \/ Shortcircuiting
eval (And e1 e2) = if eval e1 then eval e2 else (Bool False)
eval (Or e1 e2) = if eval e1 then (Bool True) else eval e2

exec :: Stmt -> State -> State
exec (DoWhile expr stmt) state =
    let state' = exec stmt state
    in if (eval expr state') == Bool True
       then exec (DoWhile expr stmt) state' else state'
```

1.5 Function calls

```
data Expr  = ... | Call String [Expr]
data Decl  = ... | Fun String [String] Expr
data Stmt  = ... | Return Expr
data Value = ... | Closure [String] Expr Env
eval :: Expr -> Value
eval (Call "max" args)
  | length args == 2 = let [n,m] = args in max (eval n) (eval m)
  | otherwise = error "expected 2 arguments but got n"
```

1.6 Structures

Data structure for representing expressions in C.

```
struct Expr {
  enum { CONST,BINARY} type;
  union {
    double constant_value;
    struct { enum { ADD, SUB, MUL, DIV } op;
             struct Expr* left; struct Expr* right; } binary;
  };
};
```

Data structure for representing expressions in Java.

```
interface Expr { Value eval(Map<String, Value> env); }
interface Stmt { void exec(Map<String, Value> env); }
interface Value {}
class Assignment implements Stmt {
    String lvalue; Expr expr;

    @Override
    void exec(Map<String, Value> env) {
        env.put(lvalue, expr.eval(env));
    }
}
class Call implements Expr { String name; List<Expr> args; }
class Function implements Value {
    String name; List<String> params; List<Stmt> body;
}
```

Chapter 11 — Compilers

1 Generating runtimes

1.1 Stack machines

An abstract machine includes instruction sets, memory model, execution model and registers. They function in stages such as fetching, decoding, choosing, execution and storing. Stack machines use a minimalistic instruction set, on a LIFO stack memory, in sequential execution, with few registers.

```
data Instr = Push Int | Pop | Add | Sub | Mul | Div
type Stack = [Int]
exec :: [Instr] -> Int
exec instrs = head $ foldl exec' [] instrs
  where
    exec' :: Stack -> Instr -> Stack
    exec' stack instr = case instr of
      Push val -> val : stack
      Pop -> tail stack
      Add -> apply (+) stack
      Sub -> apply (-) stack
      Mul -> apply (*) stack
      Div -> apply div stack - \/ Note reverse operand order
    apply op (x:y:rest) = (y `op` x) : rest
    apply _ _ = error "Not enough operands"
```

In Java, a `Stack<Integer>` can be used for the LIFO stack.

1.2 Code generation

Types of instructions include : arithmetic, logical, data transfer, control transfer.

```
x := 10;
while x > 0 do
  x := x - 1;
--
LOAD 10               ; Load 10 into the acc
STORE x               ; Store the acc into x
start_loop:
LOAD x                ; Load x into the acc
JUMPIFZERO x end_loop ; Jump to end_loop if x is zero
SUB 1                 ; Subtract 1 from acc
STORE x               ; Store the result back into variable x
JUMP start_loop       ; Jump back to the start of the loop
end_loop:
HALT                  ; Halt the program
```

1.3 Runtime environment

Responsibilities are : de.allocating resources, handling memory, i/o, os interaction and function calls. Examples : JVM JRE (Java runtime environment), CLR (C#).
Function prolog allocates space for variables, saves callee-saved registers and configures the stack frame.
Function epilog cleans up the stack frame, restores callee-saved registers, returns control to caller.

```
+----------------------------------+ void foo(int n) {
| Variables locales de main        |   printf("foo(%d)\n", n);
| Paramètres de main               |   if (n > 0)
| Adresse de retour vers OS        |     foo(n - 1);
+----------------------------------+   printf("foo(%d)\n", n);
| "Main program starts\n"          | }
+----------------------------------+ int main(void) {
| Variables locales de foo         |   printf("Main starts\n");
| Param`etres de foo (n=2)         |   foo(2);
| Adresse de retour vers main      |   printf("Main ends\n");
| Pointeur vers la frame de main   |   return 0;
+----------------------------------+ }
| "Entering foo(2)\n"              |
+----------------------------------+
| Variables locales de foo         |
| Paramètres de foo (n=1)          |
| Adresse de retour vers foo(2)    |
| Pointeur vers la frame de foo(2) |
+----------------------------------+
|                                  | <-- SP
+----------------------------------+
```

2 Optimization

2.1 Methods

2.1.1 Constant folding

Evaluate constants : `x = 60 * 60 * 24` ⇔ `x = 86400`

2.1.2 Common subexpression elimination

Rewrite `int res = a * (b-c) + d * (b-c)`
as `int common = b-c; int res = a * common + d * common`

2.1.3 Dead code elimination

Simply removes code that is never used nor accessed.

2.1.4 Constant propagation

Replace constant values : `x = 5; y = x + 3` ⇔ `y = 5 + 3`

2.1.5 Function inlining

Replaces a function call with the code of the function.
`int add(int a, int b); int res = add(3,5);` ⇔ `int res = 3 + 5;`

2.2 Optimization levels

Compilers use different levels that toggle algorithms that take more time to resolve or are very aggressive.

Chapter 12 — Garbage collection

1 Data structures

1.1 Reachable objects

Objects that are immediately accessible from the global variables, the stack, or the registers, or objects reachable from other reachable objects by following pointers.

1.2 Other structures

Free list : The list of heap blocks that are free
Block header : Properties (like size) given to a block
BiBOP : Regroups objects of identical size into contiguous pages, with block size for a page being stored in the first x bytes
Fragmentation : **external fragmentation** is free memory split in many small blocks, while **internal fragmentation** is wasted memory inside an allocated area.

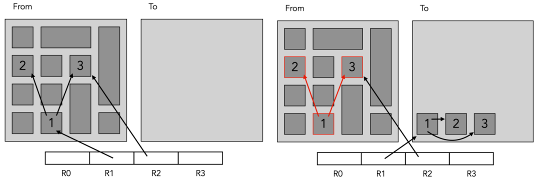
2 Management techniques

2.1 Reference counting

```
struct SmartPtr {
    SmartPtr(int* ptr) : data(ptr), refCount(new size_t(1)) {}
    ~SmartPtr() { release(); }
    SmartPtr(const SmartPtr& other)
        : data(other.data),
          refCount(other.refCount) { (*refCount)++; }
    SmartPtr& operator=(const SmartPtr& other) {
        if (this != &other) {
            release();
            data = other.data; refCount = other.refCount;
            (*refCount)++;
        }
        return *this;
    }
private:
    int* data; size_t* refCount;
    void release() {
        if (refCount != nullptr && -(*refCount) == 0) {
            delete data;
            delete refCount;
        }
    }
};
```

2.2 Copying GC

Split the heap in two, copy reachable blocks when full. Keeps track of copied objects by marking them along the DFS exploration.



2.3 Mark & sweep GC

A first DFS exploration is made from root objects, marking reachable objects in the process. Then the system sweeps the data by removing unallocating unused objects and updating the free list. Allocating data follows the first-fit or best-fit principle, always giving back the remaining free data block to the free list.