SOM2IF15 – Compilation

Semantic Analysis

Frédéric Loulergue

2022

Semantic Analysis

Goal

1. Build the symbol table(s)
2. Perform some semantic analysis, in particular:
   - Use of undeclared names
   - Type checking (or type inference for some languages)

Symbol Table

- Information about declared identifiers
- Depending on the features of the language:
  - A table at the “global” level
  - A table for each scoping block

Outline

Symbol Tables

LPL

- A block is a statement (like C)
- Declarations and instructions are ordered. Do we need a new symbol table for each declaration?

Symbol Tables

LPL

- A block is a statement (like C)
- Declarations and instructions are ordered. Do we need a new symbol table for each declaration?
  No if we check for the use of undeclared variables
  while we build the symbol tables

Symbol Tables

Symbol Tables for LPL

- One symbol table at the global level: as LPL does support global variables and type definitions, we need several tables (one for function signatures, *etc.*)
- One symbol table per:
  - function?
  - block?

Symbol Tables

Symbol Tables for LPL

- One symbol table at the global level: as LPL does support global variables and type definitions, we need several tables (one for function signatures, *etc.*)
- One symbol table per:
  - function?
  - block?

A LPL Function

```
int abs(int x){
  if(x < 0) {
    int result = - x;
    return result;
  } else {
    char result = '!'
    int resultat = x;
    return resultat;
  }
}
```

Symbol Tables

Symbol Tables for LPL

- One symbol table at the global level: as LPL does support global variables and type definitions, we need several tables (one for function signatures, *etc.*)
- One symbol table per:
  - function?
  - block?

A LPL Function

```
int abs(int x){
  if(x < 0) {
    int result = - x;
    return result;
  } else {
    char result = '!'
    int resultat = x;
    return resultat;
  }
}
```

$\Rightarrow$ A symbol table per function is not enough

Symbol Tables for LPL

Semantic Analysis for LPL

1. AST Visitor:
   - to build the symbol tables
   - to check if undeclared names are used
   - to check that LHS of assignments are assignable
2. AST Visitor:
   - to type check LPL programs

Data Structures

For tables:

- interface `Map<K, V>` with `K = String`
- class `HashMap` for objects

Design Choices

- How to represent types?
- How to represent passed type checking?
- How to deal with errors?
  (`visit` methods don't throw any exceptions)
- How to associate symbol tables with blocks?

Outline

Reminder

```
program : ( declaration ';' ) * block;

block : (instruction ';')* ;

declaration : type Identifier ( ',' Identifier )*;

type : 'integer'  #TINT
     | 'boolean'  #TBOOL
     ;
instruction :
        Identifier ':=' expr                          #IAssign
    | 'if' expr 'then' block 'else' block 'end'       #IIf
    | 'while' expr 'do' block 'end'                   #IWhile
    | 'print' '(' expr ')'                            #IPrint
    | 'input' '(' Identifier ')'                      #IInput
    ;
expr: Identifier                                      #EId
    | Number                                          #EInt
    | Boolean                                         #EBool
    // ...
```

Type Representation

W2

- Only two types: `int` and `bool`
- But the visitor pattern visits all the nodes of the AST including instructions, blocks, *etc.*
- We need a type to return in these cases

Examples of Design Choices

- `Optional<TypBasic>` where `TypBasic` in the type enumeration in the package `ast`
- A new type as a new enumeration
- A new type with only three different objects
  (Singleton Pattern, one benefit: == to compare values)

Type Representation: New Type with 3 Values

```
package compiler;

import ast.TypBasic;

import java.util.Optional;

public class Type {
  public static final Type integer = new Type(TypBasic.INTEGER);
  public static final Type bool = new Type(TypBasic.BOOLEAN);
  public static final Type none = new Type();
  private Optional<TypBasic> type;

  private Type() { this.type = Optional.empty(); }

  private Type(TypBasic type) { this.type = Optional.of(type); }

  public boolean isType() { return type.isPresent(); }

  public TypBasic get() { return type.get(); }

  @Override
  public String toString() {
    if (type.isPresent())
      return type.get().toString();
    else
      return "none";
  }
}
```

Signatures

The W2 Language

- does not support function definition
- but it has predefined operations
- let's encode the signatures of these operations

Overview

- `Signature`: a class to represent signatures including type-checking features
- `Signatures`: maps from unary and binary operations to their signatures

Signature (1/3)

```
public class Signature {
  public final static Signature binaryArithmetic =
    buildBinary(Type.integer, Type.integer, Type.integer);
  public final static Signature binaryBoolean =
    buildBinary(Type.bool, Type.bool, Type.bool);
  public final static Signature unaryArithmetic =
    buildUnary(Type.integer, Type.integer);
  public final static Signature unaryBoolean =
    buildUnary(Type.bool, Type.bool);
  public final static Signature comparison =
    buildBinary(Type.integer, Type.integer, Type.bool);
  public List<Type> argTypes;
  public Type returnType;
```

Signature (2/3)

```
// no need to build new signatures in W2
private Signature() {
}
private Signature(List<Type> argTypes,
                  Type returnType) {
  this.argTypes = argTypes;
  this.returnType = returnType;
}

private static Signature buildBinary(Type t1, Type t2, Type rt) {
  List<Type> argTypes = new ArrayList<>();
  argTypes.add(t1);
  argTypes.add(t2);
  return new Signature(argTypes, rt);
}

private static Signature buildUnary(Type type,
                                    Type rt) {
  List<Type> argTypes = new ArrayList<>();
  argTypes.add(type);
  return new Signature(argTypes, rt);
}
```

Signature (3/3)

```
  // The general check does not need to be public for W2
  private boolean check(List<Type> types) {
    if (types.size() == argTypes.size()) {
      for (int counter = 0; counter < types.size(); counter++)
        if (!types.get(counter).equals(argTypes.get(counter)))
          return false;
      return true;
    }
    return false;
  }

  public boolean check(Type type) {
    List<Type> types = new ArrayList<>();
    types.add(type);
    return check(types);
  }

  public boolean check(Type t1, Type t2) {
    List<Type> types = new ArrayList<>();
    types.add(t1);
    types.add(t2);
    return check(types);
  }
}
```

Signatures

```
public class Signatures {
  public static final Map<OpBinary, Signature> binary = buildBinary();
  public static final Map<OpUnary, Signature> unary = buildUnary();

  private static Map<OpBinary, Signature> buildBinary() {
    Map<OpBinary, Signature> mapping = new HashMap<>();
    mapping.put(OpBinary.ADD, Signature.binaryArithmetic);
    mapping.put(OpBinary.SUB, Signature.binaryArithmetic);
    mapping.put(OpBinary.MUL, Signature.binaryArithmetic);
    mapping.put(OpBinary.DIV, Signature.binaryArithmetic);
    mapping.put(OpBinary.MOD, Signature.binaryArithmetic);
    mapping.put(OpBinary.AND, Signature.binaryBoolean);
    mapping.put(OpBinary.OR, Signature.binaryBoolean);
    mapping.put(OpBinary.LT, Signature.comparison);
    mapping.put(OpBinary.LE, Signature.comparison);
    mapping.put(OpBinary.GT, Signature.comparison);
    mapping.put(OpBinary.GE, Signature.comparison);
    // EQ and NEQ not there: they'd need a polymorphic signature
    return mapping;
  }

  private static Map<OpUnary, Signature> buildUnary() {
    Map<OpUnary, Signature> mapping = new HashMap<>();
    mapping.put(OpUnary.MINUS, Signature.unaryArithmetic);
    mapping.put(OpUnary.NOT, Signature.unaryBoolean);
    return mapping;
  }
}
```

Types and Signatures for LPL

Constraints and Design Choices

Types:

- Unlike W2, LPL has an infinite number of possible types
- It is possible to reuse your type class in the AST package
- Warning: types should be equal if they have the same structure. You need to override `equals`

Signatures:

- Unlike W2, LPL has user-defined functions hence signatures
- The general constructor should be public
- Pre-defined signatures can be implemented as in W2

Type Checking in W2

Constraints

- No recursion in W2
- No block with declaration in W2

$\Rightarrow$ One symbol table

$\Rightarrow$ Analysis in one phase

Design Choices

- Type environment (symbol table) as a private field
- Errors accumulated in a data structure (here a `List`)
- Accessor to the type environment throws an error if the error list is not empty

Type Checker (1/7)

```
public class TypeChecker implements Visitor<Type> {
  private Map<String, Type> typeEnvironment;
  private List<String> errors;

  public TypeChecker() {
    typeEnvironment = new HashMap<>();
    errors = new ArrayList<>();
  }
  public boolean hasErrors() {
    return !errors.isEmpty();
  }
  public List<String> errors() {
    return errors;
  }
  // return the data structure needed for the interpreter
  public Map<String, TypBasic> typeEnvironment() {
    if (hasErrors())
      throw new Error("Type checked failed");
    else {
      Map<String, TypBasic> te = new HashMap<>();
      for (Map.Entry<String, Type> kv : typeEnvironment.entrySet())
        te.put(kv.getKey(), kv.getValue().get());
      return te;
    }
  }
```

Type Checker (2/7)

```
@Override
public Type visit(ExpBinop exp) {
  Type left = exp.left.accept(this);
  Type right = exp.right.accept(this);
  // Special case for EQ and NEQ because of polymorphism
  if (exp.op == OpBinary.EQ || exp.op == OpBinary.NEQ) {
    if (left != right) // (left.equals(right))
      errors.add("Expression at " + exp.left.pos
        + " should have the same type "
        + "than expression at "
        + exp.right.pos + ".");
    return Type.bool;
  }
  Signature signature = Signatures.binary.get(exp.op);
  if (!signature.check(left, right))
    errors.add("At " + exp.pos + ": arguments have types "
      + left + " and " + right + " but types "
      + signature.argTypes.get(0) + " and "
      + signature.argTypes.get(1)
      + " are expected.");
  return signature.returnType;
}
```

Type Checker (3/7)

```
@Override
public Type visit(ExpInt num) {
  return Type.integer;
}

@Override
public Type visit(ExpBool bool) {
  return Type.bool;
}

private Type checkDeclared(String var, Position pos) {
  Type type = typeEnvironment.get(var);
  if (type == null) {
    errors.add("At " + pos + " variable "
      + var + " is undeclared.");
    return Type.none;
  }
  return type;
}

@Override
public Type visit(ExpVar var) {
  return checkDeclared(var.name, var.pos);
}
```

Type Checker (4/7)

```
@Override
public Type visit(InsInput ins) {
  checkDeclared(ins.var, ins.pos);
  return Type.none;
}

@Override
public Type visit(InsPrint ins) {
  ins.exp.accept(this);
  return Type.none;
}

@Override
public Type visit(InsIf ins) {
  if (ins.exp.accept(this) != Type.bool)
    errors.add("At " + ins.exp.pos + " an expression "
      + "of type boolean is expected.");
  ins.then_branch.accept(this);
  ins.else_branch.accept(this);
  return Type.none;
}
```

Type Checker (5/7)

```
@Override
public Type visit(InsWhile ins) {
  if (ins.exp.accept(this) != Type.bool)
    errors.add("At " + ins.exp.pos +
      " an expression of type boolean is expected.");
  ins.body.accept(this);
  return Type.none;
}

@Override
public Type visit(InsAssign ins) {
  Type e_type = ins.exp.accept(this);
  Type v_type = checkDeclared(ins.var, ins.pos);
  if (v_type != e_type)
    errors.add("At " + ins.pos + " variable "
      + ins.var + " has type " + v_type
      + " but is assigned a value of type "
      + e_type + ".");
  return Type.none;
}
```

Type Checker (6/7)

```
@Override
public Type visit(Typ type) {
  switch (type.type) {
    case INTEGER:
      return Type.integer;
    case BOOLEAN:
      return Type.bool;
  }
  return Type.none;
}

@Override
public Type visit(Block block) {
  for (Ins instruction : block.instructions)
    instruction.accept(this);
  return Type.none;
}
```

Type Checker (7/7)

```
@Override
public Type visit(Declaration declaration) {
  Type type = declaration.type.accept(this);
  for (String var : declaration.vars) {
    Type previousType =
      typeEnvironment.put(var, type);
    if (previousType != null)
      errors.add("At " + declaration.pos + ":"
        + var + " already declared.");
  }
  return Type.none;
}

@Override
public Type visit(Program program) {
  for (Declaration d : program.declarations)
    d.accept(this);
  program.main.accept(this);
  return Type.none;
}
```

Semantic Analysis for LPL

Two Phases

- `SymbolTable`: a visitor to build the symbol tables and check for usage of undeclared identifiers
- `TypeChecker`: a visitor to type check LPL programs and to check expressions are assignable for the assignment and read instructions

Semantic Analysis for LPL

Symbol Tables

- Formally, one table per block
- More complicated than W2 to get the information related to an identifier:
  - If the identifier has been declared locally, it is in the local symbol table
  - If not, it is needed to lookup in the enclosing block, possibly up to the function's block

Design choices

- association block – type environment
- reference to the enclosing block of a block
- decoupling AST/these structures possible, for e.g.:
  - a `Map<Block, Map<String, Type> >` association,
  - for visitor: stack of the current block nesting (private field)
- also possible to have new fields in the AST

Semantic Analysis for LPL

Assignable

The syntax rules for assigment and read are:

```
instruction:
  expression op=(MinusEqual|...|EqualSymbol) expression
```

Not all syntactically correct instructions are semantically correct:

```
int abs(int x) { if (x <0) { x = - x} return x }
void main(){
   int x = -42
   1 = 2          // 1 is not assignable
   abs(x) = 0     // abs(x) is not assignable
}
```

Semantic Analysis for LPL

Assignable

The syntax rules for assigment and read are:

```
instruction:
  expression op=(MinusEqual|...|EqualSymbol) expression
```

Not all syntactically correct instructions are semantically correct:

```
int abs(int x) { if (x <0) { x = - x} return x }
void main(){
   int x = -42
   1 = 2          // 1 is not assignable
   abs(x) = 0     // abs(x) is not assignable
}
```

When is an expression assignable?

Assignable

When is an expression assignable?

Assignable

When is an expression assignable?

- If an expression is a variable, it is assignable

Assignable

When is an expression assignable?

- If an expression is a variable, it is assignable
- If an expression is an indexed access, it is assignable
  Do we need to worry what's `e` in `e[index]`?

Assignable

When is an expression assignable?

- If an expression is a variable, it is assignable
- If an expression is an indexed access, it is assignable
  Do we need to worry what's `e` in `e[index]`?
  No: just that it has type `Array<`τ`>` or `Tuple<`$\tau_1, \ldots, \tau_n$`>` or ... for some types τ and τ_i
  This is done during type-checking

Assignable

When is an expression assignable?

- If an expression is a variable, it is assignable
- If an expression is an indexed access, it is assignable
  Do we need to worry what's `e` in `e[index]`?
  No: just that it has type `Array<`τ`>` or `Tuple<`$\tau_1, \ldots, \tau_n$`>` or ... for some types τ and τ_i
  This is done during type-checking
- If an expression is a field access, it is assignable

Assignable

When is an expression assignable?

- If an expression is a variable, it is assignable
- If an expression is an indexed access, it is assignable
  Do we need to worry what's `e` in `e[index]`?
  No: just that it has type `Array<`τ`>` or `Tuple<`$\tau_1, \ldots, \tau_n$`>` or ... for some types τ and τ_i
  This is done during type-checking
- If an expression is a field access, it is assignable
- The assignable check can be performed during phase one or two

Semantic Analysis for LPL

Type Checking

- Not fundamentally different from W2 type checking
- More cases including:
  - user-defined function, thus user-defined signatures
  - the type-checking of function calls

Some difficulties

- Typing of record enumerations `{ key = '1', value = 1}`
- Type checking of dictionaries (the argument of `map` should be a record with two fields, `key` and `value`)
- Overloading
- Genericity