City University London

IN2009

Language Processors

Session 9

Semantic Analysis Typechecking

Igor Siveroni

Session Plan

Session 9:

- Extending SPL: Syntax
- Semantics of extension
- Semantic Analysis: Typechecking

Test3: Monday, April 26. Room/Time TBA

Topic: Semantic Analysis and Frames.

Extending SPL

We extend SPL with new basic types and include global variables and function declarations:

Program$_S$ → *VarDecl** *MainDecl* *FuncDecl**

MainDecl → **`void main(){`** *VarDecl** *Statement*$^+$ **`}`**

FuncDecl → S*Type* id **`(`** `ParamDecls` **`)`** **`{`**
VarDecl* *Statement** **`return`** *Exp;* **`}`**

VarDecl → *SType* *id* **`;`**

ParamDecls → *ParamDecl* (**`,`** *ParamDecl*)* | ε

ParamDecl → S*Type id*

Extending SPL: Syntax

SType → **bool** | **int** | **float**

Exp → **…** | *CallExp* | *FloatLiteral*

CastExp → **(** *SType* **)** *Exp*

`CallExp` → *id* **(** *Params* **)**

`Params` → *Exp* (**,** *Exp*)* | ε

SPL: Example

```
float pi;    // global var
void main() {
  int x; bool t;
  pi = 3.1416;  // assign global var
  x := 1;
  b := 5 > x;   // boolean value
  while (b)
   { print((float) x); // cast
     x := x+1;
     b := 5 > y; }
}

float area(float r) { return pi * r * r; }
```

SPL: Updated Abstract Syntax

We only show the new/updated abstract syntax:

Program(List<VarDecl> globals, MainDecl m, List<FuncDecl> fs)
MainDecl(List<VarDecl> locals, List<Statement> body)
FuncDecl(SType rtype, id fname, List<ParamDecl> pdecls,
List<VarDecl> locals, List<Statement> body, Exp e)

VarDecl(SType t, id x)
ParamDecl(Stype t, id x)

CallExp(id fname, List<Exp> ps)

CastExp(Stype t, Exp e)
FloatLiteralExp(float n)

Recall SPL: Semantics

- A lookuptable **table** of class **Table**:
  - table.update(x,v): Variable x has now value v.
  - table.lookup(x): returns the value associated to x.
- Statement execution: **execStm(s,table,stdout)**
- Expression evaluation: **evalExp(e,table)=v**
- New:
  - **Value v** can be an integer, boolean or float.
  - Function calls:

execFunction(FuncDecl f, List<Value> ps, table, stdout)=v

Modifying Table

We need to adapt Table to store multiple instances of the same variable due to (recursive) function calls:

- **table.pushFrame(fname):** Creates a separate space for all variables/params declared in function fname.
- **table.popFrame():** Removes last frame.

Example:

```
int x;
void main() { x := 5; print (f(x)); }
int f(int x) {
  int r;
  if (1 >= x) then { r := 1; } else { r := x*f(x-1); }
  return r;   }
```

SPL updated semantics

execProgram(P,table,stdout):

P is Program(List<VarDecl(t,x)> globals, MainDecl m,
List<FuncDecl> fs)
m is MainDecl(List<VarDecl> vars, Statement* ls)

For each x in globals: **table.update(x,0)**
(initialise each global variable to 0)

execFunction(m, -,table,stdout)
(execute main function m)

SPL updated semantics

execFunction(m, -, table,stdout): // no parameters

m is MainDecl(List<VarDecl> locals,
List<Statement> body)

table.pushFrame(fname)
For each x in locals: **table.update(x,0)**
(initialise each local variable to 0)
For each s in lbody: **execStm(s,table,stdout)**
(execute each statement in main body)
table.popFrame()

SPL updated semantics

execFunction(f, List<Exp> params, table,stdout):

f is FunDecl(rtype, fname, pdecls, locals, body,e)

table.pushFrame(fname)
For each (pi,ti) in pdecls, vi = params[i]:
ReportError if ti != type of vi
  table.update(pi,vi)
For each x in locals: **table.update(x,0)**
For each s in lbody: **execStm(s,table,stdout)**
v = evalExp(e,table)
ReportError if rtype != type of v
table.popFrame()
return v of type rtype

SPL updated semantics

evalExp(CallExp(fname, List<Exp> ps), table):

f is FunDecl(rtype, fname, pdecls, locals, body,e)

Create List<Value> params where:
    params[i] = evalExp(ps[i], table)
v = execFunction(f, params, table)
return v

Example: call g(x+2,y) with **fname** g, and parameters **ps** = list of expressions x+2 and y.

Typechecking

- We would like to execute an SPL program without performing dynamic runtime checks.
- We can get rid of runtime checks by making sure that programs are correctly typed.
- **Typechecking** guarantees that all programs that pass the check are correctly typed.
- **Typechecking** is part of the semantic analysis phase.
- It also makes sure that generated (compiled) code does not incur in runtime type errors.

Typechecking specification

- We will define the following typechecking functions:
  typecheck(Program, STable)
  typecheck(FunDecl, STable)
  typecheck(FunDecl, STable)
  typecheck(Stm, FunDecl, STable)
  typecheck(Exp, FunDecl, Stable) = type
- We will use a Symbol Table (Stable) with the following interface:
  stable.getVarType(id,FunDecl f): Returns the type of variable id inside **f**.
  stable.getFunctionDecl(fname): Returns the function declaration associated to function name **fname**.

Semantic Analysis

- The symbol table connects variable and function definitions to their uses (identifiers)
- Semantic analysis:
  - Checks that each use matches an appropriate declaration.
  - Checks that each expression/statement of the program is of correct type
- The symbol table is generated by traversing all declaration ASTs and collecting all identifiers together with their types (return and parameter types in the case of functions).
- We will assume that the symbol table has been generated.
- We start the typechecking specification with **Stable stable,** created given program P.

Typechecking SPL

typecheck(Program(globals, MainDecl m,
List<FuncDecl> fs)**, stable):**
typecheck(m,stable)
For each f in fs: **typecheck(f, stable)**

typecheck(m,stable)
m = MainDecl(locals, List<Statement> body)
For each s in body: **typecheck(s, m, stable)**

typecheck(f, stable):
f = FunDecl(rtype, fname, pdecls, locals, body,e)
for each s in body: **typecheck(s,f,stable)**
t = typecheck(e, f, stable) // returns type of e
ReportError if t != rtype

Typechecking SPL

typecheck(AssignStm(Id x, Exp e),f,stable):

t1 = typecheck(e,stable) // returns type of e, if e is correctly typed

t2 = stable.getVarType(f,x) // retrieves type of variable x in function f.

ReportError if t1 != t2

typecheck(PrintStm(Exp e), stable)

typecheck(e,f,stable) // makes sure that e is correctly typed

Typechecking SPL

typecheck(IfStm(Exp e,Stm+ ls1,Stm+ ls2), f, stable):

t = typecheck(e,f,stable) // typechecks condition e and returns type, if successful

ReportError if t != boolean

// typechecks each statement in ls1 and ls2

For each s in ls1: **typecheck(s, f, stable)**

For each s in ls2: **typecheck(s, f, stable)**

typecheck(WhileStm(Exp e, List<Stm> body), f, stable):

t = evalExp(e,f, stable) // typecheck condition e

ReportError if t != boolean

For each s in ls1: **typecheck(s, f, stable)**

Typechecking SPL

typecheck(OpExp(Exp e1,Aop op, Exp e2), f, stable):

t1 = typecheck(e1, f, stable) // first typecheck both operators
t2 = typecheck(e2, f, stable)
// both types need to be equal – **int** or **float**
ReportError if not ((t1=t2=int) or (t1=t2=float))
return t1

typecheck(BoolExp(Exp e1,BOp op, Exp e2), f, stable):

t1 = typecheck(e1, f, stable)
t2 = typecheck(e2, f, stable)
ReportError if t1 != boolean or t2 !=boolean // not (t1=t2=boolean)
return boolean

Typechecking SPL

typecheck(CmpExp(Exp e1, COp op, Exp e2), f,stable):

t1 = typecheck(e1,f,stable) // first typechecks both operands
t2 = typecheck(e2,f,stable)
// both types need to be equal, - int of float.
ReportError if not ((t1=t2=int) or (t1=t2=float))
return boolean

typecheck(IdExp(Id x), f, stable) :

t = stable.getVarType(x,f)
return t // returns type of x in function declaration f

evalExp(NumLiteralExp(int n), f, stable) :

return int // the type of a NumLiteralExp e.g. 5, is always **int**

SPL: Typechecking

typecheck(BoolLiteralExp(bool v), f, stable) :

return boolean

typecheck(FloatLiteralExp(float v), f, stable) :

return float

typecheck(CallExp(fname, List<Exp> ps), f, stable):

```
FunDecl(rtype, fname, pdecls, -, -,-) =
                        stable.getFunctionDecl(fname)
// number of paramenters and param. decl. must be the same
ps.size() == pdecls.size()
// type of each paramenter must match its declaration
For each i: 0 .. ps.size()-1
    pdecls[i].type = typecheck(ps[i], f, stable)
return rtype   // the type of the call is the return type of function
```

SPL: Runtime errors

```
void main() {
int x; int y, float z;
  x := (y < 5);     // type error
  y := 10*z;        // wrong types for *
  x := f(y);    // error: arg number
  z := f(y,5.5);  // return type}

int  f(int p1, float p2) {
  p2 := p1 * 10.5;
  return p2;  // error: return value must be int
}
```

Next…

• Typechecking verifies if a program is correctly typed.

• Correctly typed program will not generate runtime type errors. Runtime checks can be removed all together.

New issues:

• The presence of functions requires that the lookup table keeps track of different sets of variables and values, one per active function call.

•The lookup table needs to handle the notion of frames.

•The generated program in TPL needs to do the same. How? Answer: Stack frames/Activation records.