Semantic Analysis

Chapter 5

Semantic Analysis

- *Determine static properties of programs*
  - association between definitions and applied occurrences of variables and other names
  - scope and visibility of names, "escaping" names
  - types of variables, functions, and expressions
    - computing types
    - detecting type errors
- *Relevant structures*
  - abstract syntax
  - symbol tables, static environments
  - types

Bindings and Environments

Binding:

- an association between an identifier and its static description:

`a → int`

Environment:

- a finite set of bindings (unique identifiers)
- a finite mapping from symbols to static information, e.g. types

`σ = {a→int, b→string}`

Declarations *extend* environments (using concatenation):

`σ + {a→string} = {b→string, a→string}`

Environments and Scope

$\sigma_0 = \{a \rightarrow int,\ b \rightarrow string,\ d \rightarrow int\}$

```
function f(a:int, b:int, c:int) =
```

$\sigma_1 = \sigma_0 + \{a \rightarrow int,\ b \rightarrow int,\ c \rightarrow int\}$

```
   (print_int(a+c);
```

σ_1

```
    let var j := a+b*d
```

$\sigma_2 = \sigma_1 + \{j \rightarrow int\}$

```
        var a = "hello"
```

$\sigma_3 = \sigma_2 + \{a \rightarrow string\}$

```
     in print(a); print_int(j)

    end;
```

σ_1

```
    print_int(b))
```

$\sigma_4 = \sigma_0 + \{f:Proc(int*int*int)\}$

Implementation of Environments

Imperative symbol tables

Destructive update: A symbol table has state, which is changed by adding and removing bindings as we enter and leave scopes

A single symbol structure value is used throughout analysis, representing different environments at different times.

Important that operations are performed in the right order at the right time.

Hash table is typical representation

Functional symbol tables

Nondestructive bind operation
No need for remove operation
Environments can be shared
Binary tree is typical representation

Hash Table

```
val SIZE = 109 (* a prime *)
type binding = ...
type bucket = (string * binding) list
type table = bucket Array.array
val t : table = Array.array(SIZE,nil)

exception NotFound

fun hash(s : string) : int =
    CharVector.foldl (fn (c,n) => (n*256+ord c) mod SIZE)
                     0 s

fun insert(s: string, b: binding) =
    let val i = hash(s)
     in Array.update(t,i,(s,b)::Array.sub(t,i))
    end
```

Hash Table (cont.)

```
fun lookup(s: string) =
    let val i = hash(s)
        fun search((s',b)::rest) =
            if s = s' then b else search rest
          | search nil = raise NotFound
     in search(Array.sub(t,i))
    end

fun pop(s: string) =
    let val i = hash(s)
        val (s',b)::rest = Array.sub(t,i)
     in assert(s = s');
        Array.update(t,i,rest)
    end
```

Imperative Environment Management

Environment represented by a global hash table.

```
fun typeExp(LetExp(decs,body,pos)) =
    let val symbols = typeDecs(decs)
          (* changes global env, returning list of
           * symbols bound in decs, in reverse order *)
     in typeExp(body);
        app HashTable.pop symbols
        (* undo bindings, restoring original state o
         * the environment *)
    end
```

Functional Symbol Tables

```
signature SYMBOL =
sig
  type symbol
  val symbol : string -> symbol
    (* create a unique symbol from a string *)
  val name : symbol -> string
    (* return the name of a symbol -- the string from which the
     * symbol was created *)
  val eq : symbol * symbol -> bool
    (* eq(s1,s2) is true if s1 and s2 are the same symbol;
     * two symbols with the same name will be equal *)
  val gt : symbol * symbol -> bool
    (* gt(s1,s2) is true if s1 > s2 in alphabetical ordering,
     * using the ascii collating sequence *)

  type 'a table
    (* functional symbol tables, parameterized by binding type *)
  val empty : 'a table
    (* empty symbol table; starting point for building tables by
     * adding bindings *)
  val enter : 'a table * symbol * 'a -> 'a table
    (* add a symbol binding to a table, producing a new table.  The
     * original table is not modified *)
  val look  : 'a table * symbol -> 'a option
    (* lookup(t,s) : returns (SOME b), where b is the binding associated
     * with symbol s in table t, if it exists; otherwise returns NONE *)
end (* signature SYMBOL *)
```

Functional Symbol Tables

Symbol tables are represented using balanced binary trees, using the symbol's number as the key for looking up the symbol in the table.

```
structure Table = IntMapTable(type key = symbol
                              fun getInt(s,n) = n)

type 'a table= 'a Table.table
val empty = Table.empty
val enter = Table.enter
val look = Table.look
```

```
(* table.sml *)
functor IntMapTable (type key
                     val getInt: key -> int) : TABLE =
struct

  type key=key
  type 'a table = 'a IntBinaryMap.map

  val empty = IntBinaryMap.empty
  fun enter(t,k,a) = IntBinaryMap.insert(t,getInt k,a)
  fun look(t,k) = IntBinaryMap.find(t,getInt k)

end (* functor IntMapTable *)
```

Tiger Environments

```
signature TC_ENV =
sig

  type ty = Types.ty

  (* bindings (table entries) for variable environments *)
  datatype enventry
    = VARentry of {ty: ty, for: bool}
    | FUNentry of {formals: ty list, result: ty}

  type tenv = ty Symbol.table
    (* type environment, giving types associated with type names *)

  type venv = enventry Symbol.table
    (* variable environments, giving types for declared
     * variables and functions *)

  type env = tenv * venv

  val base_env : env
    (* contains predefined types (tenv) and functions (venv) *)

end (* signature TC_ENV *)
```

Functional Environment Management

Environment represented by a parameter.

```
| typeExp env (A.LetExp{decs, body, pos}) =
    typeExp (typeDecs env decs) body
```

Previous environment restored automatically by leaving the scope of the environment parameter variable.

Representing Types

```
(* types.sml *)

structure Types =
struct

  type unique = unit ref
    (* values used to provide unique type identities for declared
     * record and array types *)

  datatype ty
    = NIL       (* null record, matches any record type *)
    | UNIT      (* represents no value *)
    | INT       (* primitive integer type *)
    | STRING    (* primitive string type *)
    | RECORD of (Symbol.symbol * ty) list * unique
    | ARRAY of ty * unique
    | NAME of Symbol.symbol * ty option ref
                (* for forward references *)
    | ERROR     (* for error recovery *)

end (* structure Types *)
```

Types

1. *unique* is used to provide unique identities for declared record and array types. These identities will be tested to determine type equality.

2. *NIL* and *UNIT* are used internally, and are not directly expressible in the source code

3. All type declarations cause type ids to be bound to *NAME* types, initially of form NAME(id, ref NONE), with the ref assigned the appropriate (SOME ty) value in a second pass.

4. The *ERROR* type is used to represent the type of an erroneous expression, and plays a role in error recovery

Type Checking Expressions

Translating abstract syntax of types into Types.ty:

```
transTy : (E.env * A.ty) -> (T.ty * A.pos)
```

Computing types of expressions in an environment:

```
typeExp : E.env -> A.exp -> ty
   tych: A.exp -> ty
   tyvar: A.var -> ty * bool  (* for counter var? *)

    fun typeExp (tenv,venv) =
        let fun ...  (* auxiliary fns *)
            and tych((A.IntExp _)) = T.INT
              | tych(...) = ...
            and tyvar((A.SimpleVar(name,pos)) =
                case S.look(venv,name) of ...
         in tych
        end
```

Type Checking Declarations

The effect of typing a declaration is to produce an extended environment:

```
typeDec : (A.dec * E.env) -> E.env

and typeDec (A.VarDec(vardec), env as (tenv, venv)) =
```
(check that declared type (if any) and initialization expression are consistent)

```
  | typeDec (A.FunctionDec(fundecs), (tenv, venv)) =
```
(first pass: collect types of functions from headers and add bindings to env;
second pass: type bodies of functions in environments extended with formal parameters, check consistency with return type.)

```
  | typeDec (A.TypeDec(tydecs), (tenv, venv)) =
```
(first pass: bind type ids to NAME(type_id, ref(NONE));
second pass: translate definition and assign SOME(result) to ref;
check for illegal cycles)

Recursive Functions

```
let function f(x: int): int = g(x+1)
    function g(y: int): int = if y=0 then 1 else f(x-1)
 in f(3)
end
```

starting environment: ρ_0

env for body of let

$\rho_1 = \rho_0 + \{f \rightarrow \text{Fun(int; int)}, g \rightarrow \text{Fun(int; int)}\}$

env for body of function f (add binding for parameter x)

$\rho_f = \rho_1 + \{x \rightarrow \text{int}\}$

env for body of function g (add binding for parameter y)

$\rho_g = \rho_1 + \{y \rightarrow \text{int}\}$

Type Declarations

```
let type a = b
    type b = {x : b, y: c, z: int}
    type c = array of a
    var v : a = b{x = nil, y = c[3] of nil, z = 2}
 in v.y
end
```

t_a = NAME(sym_a, ref NONE)

t_b = NAME(sym_b, ref NONE)

t_c = NAME(sym_c, ref NONE)

$\rho_1 = \rho_0 + \{a \rightarrow t_a, b \rightarrow t_b, c \rightarrow t_c\}$

ref_a := SOME t_b

ref_b :=

SOME(RECORD([(sym_x,t_b),(sym_y,t_c),(sym_z,INT)],ref()))

ref_c := SOME(ARRAY(t_a,ref()))

Type Circularities

```
let type a = c
    type b = {x : b, y: c, z: int}
    type c = a
    var v : a = b{x = nil, y = c[3] of nil, z = 2}
 in v.y
end
```

t_a = NAME(sym_a, ref NONE)

t_b = NAME(sym_b, ref NONE)

t_c = NAME(sym_c, ref NONE)

$\rho_1 = \rho_0 + \{a \rightarrow t_a, b \rightarrow t_b, c \rightarrow t_c\}$

ref_a := SOME t_c

ref_b :=

SOME(RECORD([(sym_x,t_b),(sym_y,t_c),(sym_z,INT)],ref()))

ref_c := SOME t_a

Following Type Alias Chains

```
let type a = c
    type b = {x : b, y: c, z: int}
    type c = a
    var v : a = b{x = nil, y = c[3] of nil, z = 2}
 in v.y
end
```

t_a = NAME(sym_a, ref(SOME(NAME(sym_b, ref(SOME(RECORD(...))))))))

```
(* a function to strip indirections *)
 * actualTy : Types.ty -> Types.ty *)
fun actualTy (NAME(_,ref(SOME ty))) = actualTy ty
  | actualTy ty = ty
```

actualTy will return a type of one of the forms

NIL, INT, STRING, UNIT, RECORD, ARRAY, (ERROR?)

Problem: where is the name of a bare RECORD or ARRAY?