Symbol Tables and Overloading

Mooly Sagiv
Tel Aviv University
sagiv@math.tau.ac.il
and
Reinhard Wilhelm
Universität des Saarlandes
wilhelm@cs.uni-sb.de

June 6, 1997

Symbol Table

- A data structure used to store information on declared objects
- Supports insertions and deletions of declarations and opening and closing of scopes
- Supports efficient search for a declaration

Symbol Table Functionality

create_symb_table	creates an empty symbol table,
enter_block	notes the start of a new scope,
exit_block	resets the symbol table to the state before the last *enter_block*,
enter_id(*id*, *decl_ptr*)	inserts an entry for identifier *id* with a link to its defining occurrence passed in *decl_ptr*,
search_id(id)	searches the def. occ. for *id* returns a pointer to it if exists.

Symbol Table Implementation

- Data structure with constant time for $search\_id$,
- All currently valid def. occurrences of an id. are stored in a (stack like) linear list,
- New entry is inserted at the end of this list,
- The end of this list is pointed to by an array component for this id.,
- All entries for a block are chained through a linear list.

```
proc create_symb_table;
  begin create empty stack of block entries end ;
proc enter_block;
  begin push entry for the new block end ;
proc exit_block;
  begin
    foreach decl. entry of the curr. block do
      delete entry
    od;
    pop block entry from stack
  end ;
proc enter_id ( id: Idno; decl: ↑node );
  begin
    if exists entry for id in curr. block
    then error("double declaration")
    fi;
    create new entry with decl and no. of curr. block;
    insert entry at tail of linear list for id;
    insert entry at tail of linear list for curr. block
  end ;
function search_id ( id: idno ) ↑node;
  begin
```

```
    if list for id is empty
    then  error("undeclared identifier")
    else  return (value of decl-field of first elem. in id-list)
    fi
  end
```

Example Program

1

Decl. of a, b

proc p (* forward decl. *)

proc q

proc p **is**

2

Decl. of a, c

Decl. of c, d

proc q **is**

3

Decl. of a, d

proc r

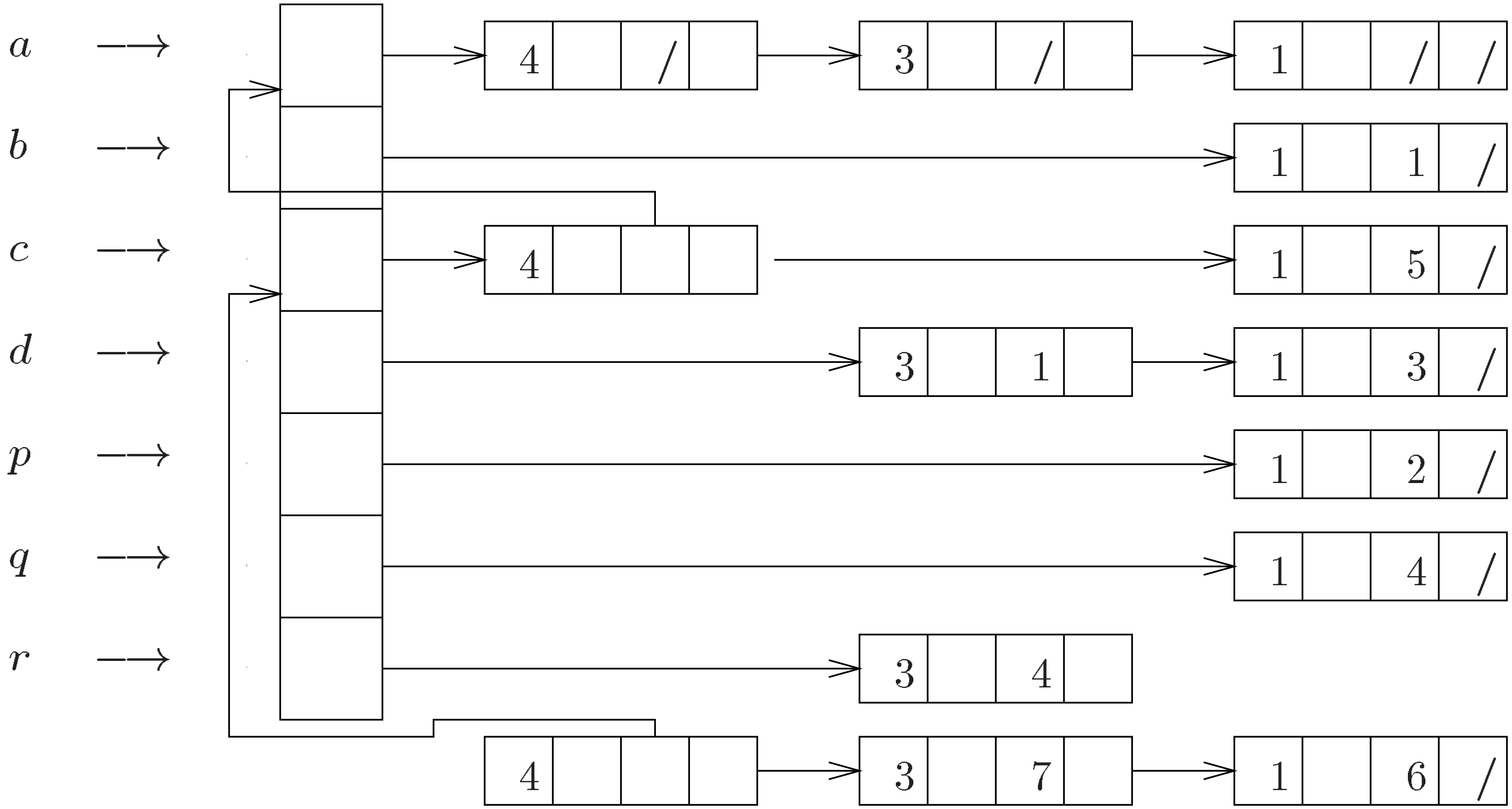
proc r **is**

4

Decl. of a, c

*

Symbol Table Implementation



Declaration Analysis

```
proc  analyze_decl (k : node);
  proc  analyze_subtrees (root: node);
  begin
      for  i := 1 to  #descs(root) do (* # children *)
              analyze_decl(root.i)          (* i-th child of root *)
      od
  end ;
begin
  case  symb(k) of      (* label of k *)
  block:   begin
                enter_block;
                analyze_subtrees(k);
                exit_block
           end ;
  decl:    begin
                analyze_subtrees(k);
                foreach  identifier declared here id do
                    enter_id(id, ↑ k)
                od
           end ;
```

```
  appl_id: (* appl. occ. of identifier id *)
           store  search_id(id) at k;
  otherwise: if  k no leaf then  analyze_subtrees(k) fi
  od
end
```

Example Data Structure — Chained Hashing

```
typedef struct nlist {
        char * name ;
        int nstLevel ;
        P_Decl decl ;
        struct nlist *next ; } Element, *Chain ;
#define NULLC (Chain) NULL
#define HASHSIZE 101

Chain sym_tab[HASHSIZE] ;

int hash(char *name)
{
int hashval ;

for (hashval = 0 ; *name != '\0';)
   hashval += *name++ ;
return(hashval % HASHSIZE);
}
```

Example Clax Program

```
PROGRAM test ;
DECLARE xy: INTEGER ;
        z: INTEGER ;
PROCEDURE p ;
DECLARE
                xy : REAL;
                yx : REAL;
BEGIN
  xy := yx ;
END ;
BEGIN
xy := 1 ;
p ;
END.
```

Resolving Declarations in Clax

- procedure is encountered — open a new scope
- declaration is encountered — insert into symbol table
- use is encountered — search for the innermost entry in the symbol table
- procedure ends — close a scope: remove all the local declarations

Symbol Table Lookup Operations

```
Chain lookup(char *name, int nstLevel)
{
Chain np ;

for (np = sym_tab[hash(name)] ; np != NULLC ;
     np = np -> next)
 if (strcmp(name, np ->name) == 0 &&
           nstLevel == np -> nstLevel)
           return(np);
return(NULLC);
}

Chain most_local(char *name)
{
Chain np ;

for (np = sym_tab[hash(name)] ; np != NULLC ;
     np = np -> next)
 if (strcmp(name, np ->name) == 0)
           return(np);
return(NULLC);
}
```

Symbol Table Insertion Operation

```
Chain install(char * name, int nstLevel, P_Decl decl)
{
Chain np ;
char * malloc(int);
int hashval ;

if ((np = lookup(name, nstLevel)) == NULLC) {
      np = (Chain) malloc(sizeof(Element)) ;
      if (np == NULLC)
         return(NULLC) ;
      if ((np -> name = strsave(name)) == (char *) NULL)
         return(NULLC);
      np -> nstLevel = nstLevel ;
      hashval = hash(np -> name);
      np -> next = sym_tab[hashval];
      sym_tab[hashval] = np ; }
      else yyerror("multiple declaration of identifier");
 np -> decl = decl ;
 return(np);
}
```

Symbol Table Deletion

```
void delete(char * name, int nstLevel)
{
Chain np, prev ;
for (prev = np = sym_tab[hash(name)] ;
     np != NULLC && (strcmp(name, np ->name) != 0 ||
                    nstLevel != np -> nstLevel) ;
     prev = np, np = np -> next)  ;
if (np == NULLC) {
  yyerror("internal error: No declaration exists");
  fprintf(stderr, "(%s, %d)\n", name, nstLevel);
}
else if (np = prev) {
         sym_tab[hash(name)] = np -> next ;
         free(np); }
else {
      prev -> next = np -> next ;
      free(np);  }
}
```

Handling Scopes

```
void open_scope() { nst_level++;}

void close_scope(P_FormalList formalList,
                 P_DeclList declList)
  {
for (; formalList != (P_FormalList) NULL ;
       formalList = formalList ->  formalList) {
 delete(formalList->formal->varDecl->id->value,nst_level);}
for (; declList != (P_DeclList) NULL ;
       declList = declList ->  declList) {
  delete(decl_id(declList->decl) -> value, nst_level); }
nst_level-- ;
  }
```

Bison Code for Procedure Declarations

```
procdecl:       prochead ';' block
        { $$ = create_ProcDecl($1, $3);
          close_scope($1 ->formalList, $3 ->declList);  }
 ;
prochead:       PROCEDURE pid parameters
{$$ = create_ProcHead($2, $3) ;
 install($2 -> value,
         nst_level - 1,
         create_Decl(ProcD,
                     create_NstLevel(nst_level),
                     create_LineNumber(line_number),
                     (P_VarDecl) NULL,
                     create_ProcDecl($$, (P_Block) NULL),
                     (P_LabDecl) NULL)) ; }
       ;
pid:   ID
{if (nst_level > 0)
 yyerror("Only two nesting levels are currently supported");
 $$ = $1 ;
 open_scope() ; }
```

Bison Code for Variable Declarations

```
decl:           set_line vardecl
 { $$ = create_Decl(VarD,
                    create_NstLevel(nst_level),
                    $1,
                    $2,
                    (P_ProcDecl) NULL,
                    (P_LabDecl) NULL) ;
   install($2 ->id -> value, nst_level, $$) ;}
     ;
set_line:  /* Empty */
    { $$ = create_LineNumber(line_number) ;}
        ;
vardecl:     ID ':' type
   { $$ = create_VarDecl($1, $3); }
      ;
```

Bison Code for Variable References

```
var:            ID
 { Chain temp = most_local($1 ->value) ;
   P_VarDecl t = (P_VarDecl) NULL ;
   P_Type type = (P_Type) NULL ;
   if (temp == NULL)
        yyerror("Undeclared identifier") ;
   else if (temp-> decl->ind != VarD)
           yyerror("Must be a simple declaration") ;
        else  {t = temp -> decl ->value.varDecl ;
               type = t -> type ;}
   $$ = create_Var(type,
                   SimpleVar,
                   $1,
                    create_NstLevel(nst_level),
                    t,
                    (struct VarRef *) NULL,
                    (struct Expr *) NULL) ; }
   ;
```

Overloading

- Several defining occurrences of an identifier may be visible at an applied occurrence,
- Have to have different parameter profile, i.e.
  - different no. of parameters,
  - different types of parameters,
  - different result type
- Compiler has to resolve the overloading, i.e., determine the one corresponding def. occ. to an appl. occ.

An Example ADA Program

```
procedure BACH is
 procedure put (x: boolean) is begin null; end;
 procedure put (x: float)   is begin null; end;
 procedure put (x: integer) is begin null; end;
 package x is
    type boolean is (false, true);
    function f return boolean;                             -- (D1)
 end x;
 package body x is
    function f return boolean is begin null; end;
 end x;
 function f return float is begin null; end;               -- (D2)
 use x;
begin
 put (f);                                                  -- (A1)
 A: declare
    f: integer;                                            -- (D3)
 begin
    put (f);                                               -- (A2)
    B: declare
      function f return integer is begin null; end; -- (D4)
      begin
         put (f);                                          -- (A3)
      end B;
 end A;
end BACH;
```

- `put`, `f` are overloaded,

- `use x` at (D3) makes f of (D1) visible,

- `f : integer` at (D3) hides decl. of f at (D1), (D2),

- `function f` at (D4) hides decl. of f at (D3), (D2).

Overload Resolution, Notation

At each node k of the abstract parse tree

$\#descs(k)$	# child nodes of k,
$symb(k)$	the symbol labelling k,
$vis(k)$	set of definitions of $symb(k)$ visible at k
$ops(k)$	set of actual candidates for the overloaded symbol $symb(k)$
$k.i$	the i-th child of k.

For each def. occ. of an overloaded symbol op with type
$t_1 \times \cdots \times t_m \rightarrow t$ let

$$\begin{array}{lcl} rank(op) & = & m \\ res\_typ(op) & = & t \\ par\_typ(op, i) & = & t_i \;\; (1 \leq i \leq m). \end{array}$$

Overload Resolution, Algorithm

```
proc  resolve_overloading (root: node, a_priori_type: type);
func pot_res_types (k: node): set of type;
        (* potential types of the result *)
  return {res_typ(op) | op ∈ ops(k)}
func act_par_types (k: node, i: integer): set of type;
  return {par_typ(op, i) | op ∈ ops(k)}
proc  init_ops
begin
  foreach   k
    ops(k) := {op | op ∈ vis(k) and rank(op) = #descs(k)}
  od;
  ops(root) := {op ∈ ops(k) |  res_typ(op) = a_priori_typ}
end ;
proc  bottom_up_elim (k: node);
begin
  for  i := 1 to  #descs(k) do
    bottom_up_elim (k.i);
    ops(k) := ops(k) − {op ∈ ops(k) |
         par_typ(op, i) ∉ pot_res_types(k.i)}
         (* remove the operators,
```

```
        whose i-th param. type doesn't match
        any pot. result type of i-th operand *)
  od;
end ;

proc top_down_elim (k: node);
begin
  for i := 1 to #descs(k) do
    ops(k.i) := ops(k.i) − {op ∈ ops(k.i) |
        res_typ(op) ∉ act_par_types(k, i)};
        (* remove the operators,
        whose res. type does not match
        any type of the corresp. param. *)
    top_down_elim(k.i)
  od;
end ;

begin
  init_ops;
  bottom_up_elim(root);
  top_down_elim(root);
  check if all ops-sets are singletons; otherwise error
end
```