Compiler Principle

Prof. Dongming LU

Mar. 25th, 2024

Content

1. INTRODUCTION

2. LEXICAL ANALYSIS
3. PARSING
4. ABSTRACT SYNTAX
5. **SEMANTIC ANALYSIS**

6. ACTIVATION RECORD
7. TRANSLATING INTO INTERMEDIATE CODE

8. OTHERS

5 SEMANTIC ANALYSIS

5.2 BINDINGS FOR THE Tiger COMPILER

What is a binding? What should a symbol table be filled?

BINDINGS FOR THE Tiger COMPILER

Tiger has two separate name spaces:

- **One for types**
- **The other for functions and variables**

All Types:

- **The primitive type: int, string**
- **Other types : constructed using records and arrays from other types.**

The Ty_array or Ty_record object :

- **Carry the implicit piece of information: the address of the object itself**

BINDINGS FOR THE Tiger COMPILER

- Every "record type expression" creates a new (and different) record type.

```
Let type a= {x:int; y:int }
     type b= {x:int; y:int }
     var i:  a:=......
     var j:  b:=.......
Int  i:= j
end
```

It is illegal in Tiger.

```
Let type a= {x:= int; y:int }
     type c = a;
     var i:  a:=......
     var j:  c:=.......
Int  i:= j
end
```

It is legal in Tiger.

Classification of type equivalence

- **Structural equivalence**

  Two types are the same if and only if they have the same structure.

- **Name equivalence**

  Two type expressions are equivalence if and only if they are either the same simple type or are the same type name.

- **Declaration equivalence**

  t2 = t1; are interpreted as establishing type aliases, rather than new types.

Classification of type equivalence

t1 = int;

t2 = int;

- **The types t1 and t2 are not equivalence and are also not equivalent to int.**
- **This is very strong sort of type equivalence.**

ENVIRONMENT

The table **type** of Symbol module provides **mappings** from **symbols** to **bindings**.

```
Let type a =
 int
    var a:
 a := 5
    var b:
 a := a
  in b+a
end
```

- **Type** environment
- **Value** environment

ENVIRONMENT

```
typedef struct E_enventry_ *E_enventry
Struct E_enventry_ {enum { E_varEntry, E_funEntry} kind;
                    union { struct { Ty_ty ty;} var;
                            struct { Ty_tyList formals; Ty_ty result;} fun;
                          } u;
                    }

E_enventry E_VarEntry(Ty_ty ty);
E_enventry E_FunEntry(Ty_tyList formals, Ty_ty result);

S_table E_base_tenv(void); /*Ty_ty environment*/
S_table E_base_venv(void); /*E-enventry environment*/
```

5.3 TYPE-CHECKING EXPRESSIONS

Performing semantic analysis of abstract syntax

Four functions over syntax trees

```
Struct expty transVar(S_table venv, S_table tenv,
  A_var  v);

Struct expty transExp(S_table venv, S_table tenv,
  A_exp a);

Void         transDec(S_table venv, S_table tenv,
  A_dec d);

Ty_ty        transTY( S_table tenv, A_ty a);
```

- The type checker is a recursive function of the abstract syntax tree.
- The result is an expty.

```
struct expty transExp(S_table venv, S_table tenv, A_exp a){
switch(a->kind){
  …
   case A_opExp:{
     A_oper oper=a->u.op.oper;
     struct expty left=transExp(venv,tenv,a->u.op.left);
     struct expty right=transExp(venv,tenv,a->u.op.right);
     if (oper==A_plusOp){
        if (left.ty->kind!=Ty_int)
           EM_error(a->u.op.left->pos,"integer required");
        if (right.ty->kind!=Ty_int)
           EM_error(a->u.op.right->pos,"integer required");
        return expTy(NULL, Ty_int());
        }
    …
     }
  }
  assert(0)
}
```

TYPE-CHECKING VARIABLES,SUBSCRIPTS, AND FIELDS

```
struct expty transVar(S_table venv, S_table tenv, A_var v){
switch(v->kind){
   case A_simpleVar:{
       E_enventy x= S_look(venv, v->u.simple)
        if (x && x->kind==E_varEntry)
           return expTy(NULL  ,   actual_ty(x-u.var.ty));
        else { EM_error(v->pos, “undefined variable %s”,
                          S_name(v->u.simple));
               return expTy(NULL, Ty_int());}
     }
    case A_fieldVar:
     …
  }
```

5.4 TYPE-CHECKING DECLARATION

Environments are constructed and augmented

Translate declarations

```
struct expty transExp(S_table venv, S_table tenv, A_exp a){
switch(a->kind){
…
  case A_letExp:{
        struct expty exp;
         A_declist d;
        S_beginScope(venv);
        S_beginScope(tenv);

        for (d=a->u.let.decs; d; d=d->tail)
              transDec(venv, tenv, d->head);
        exp=transexp(venv,tenv,a->u.let.body);

        S-endScope(tenv);
        S-endScope(venv);
        return exp;
}
…
```

VARIABLE DECLARATION

```
Void transDec(S_table venv, S_table tenv, A_dec d)  {
    switch (d->kind) {
       case A_varDec: {
           struct expty e =transExp(venv, tenv, d->u.var.init);
           S_enter(venv, d->u.var.var, E_VarEntry(e.ty));
       }
     …
}
```

Example:

var x: type-id := exp

TYPE DECLARATION

```
Void transDec(S_table venv, S_table tenv, A_dec d)  {
    ….
     switch (d->kind) {
       case A_typeDec: {
           S_enter(tenv, d->u.type->head->name,
                         transTy(tenv, d->u.type->head->ty));
       }
     …
}
```

Example:

type type-id := typeExp

FUNCTION DECLARATION

```
Void transDec(S_table venv, S_table tenv, A_dec d)  {
    switch (d->kind) {
    …
       case A_functionDec: {
         A_fundec f=d->u.function->head;
         Ty_ty resultTy=S_look(tenv,f->result);
         Ty_tyList formalTys=makeformalTyList(tenv,f->params);
         S_enter(venv,f->name,E_FunEntry(formalTys,resultTy));
         S_beginScope(venv);
         { A_filedList l; Ty_tyList t;
           for (l=f->params, t=formalTys; l; l=l->tail, t=t->tail)
                 S_enter(venv, l->head->name, E_VarEntry(t->head));
         }
         transExp(venv, tenv, d->u.function->body);
         S_endScope(venv);
         break;
         }
     …
```

RECURSIVE DECLARATIONS

- Encounter undefined type or function identifiers
  - transTy for recursive record types
  - trandExp(body) for recursive functions

Example:

type list = {first: int, rest :list}

S_enter(tenv, name, Ty_Name(name, NULL))

Put all the "headers" in the environment first.

The end of Chapter 5(2)