CS421 Assignment 5 Tiger Type Checker Chen Gu CG736

Notes in Implementation

Initialize the environment

First of all, we need to initialize all the builtin types and functions.

```
  val base_tenv =
    let
        val temp_tenv = S.enter(S.empty, S.symbol("int"), T.INT);
        val temp_tenv = S.enter(temp_tenv, S.symbol("string"), T.STRING)
    in temp_tenv end

  val base_env =
    let
        val temp_env = S.enter(S.empty, S.symbol("print"),
            FUNentry{level=(), label=(), formals=[T.STRING], result=T.UNIT
});
        val temp_env = S.enter(temp_env, S.symbol("flush"),
            FUNentry{level=(), label=(), formals=[], result=T.UNIT});
        val temp_env = S.enter(temp_env, S.symbol("getchar"),
            FUNentry{level=(), label=(), formals=[], result=T.STRING});
        val temp_env = S.enter(temp_env, S.symbol("ord"),
            FUNentry{level=(), label=(), formals=[T.STRING], result=T.INT}
);
        val temp_env = S.enter(temp_env, S.symbol("chr"),
            FUNentry{level=(), label=(), formals=[T.INT], result=T.STRING}
);
        val temp_env = S.enter(temp_env, S.symbol("size"),
            FUNentry{level=(), label=(), formals=[T.STRING], result=T.INT}
);
        val temp_env = S.enter(temp_env, S.symbol("substring"),
            FUNentry{level=(), label=(), formals=[T.STRING, T.INT, T.INT],
            result=T.STRING});
        val temp_env = S.enter(temp_env, S.symbol("concat"),
            FUNentry{level=(), label=(), formals=[T.STRING, T.STRING],
            result=T.STRING});
```

```
        val temp_env = S.enter(temp_env, S.symbol("not"),
            FUNentry{level=(), label=(), formals=[T.INT], result=T.INT});
        val temp_env = S.enter(temp_env, S.symbol("exit"),
            FUNentry{level=(), label=(), formals=[T.INT], result=T.UNIT})
    in temp_env end
```

Handling recursive type and function declaration

As taught in the lecture, I use two pass scan to handle recursive type declaration. The first pass scans the header information and enters special type `T.NAME` into the type environment. After first pass, we have all the header information, though they all refer to `NONE`. In the second pass, I call `transty` to get the actual type and use assignment to make the pointer point to the actual type. Below is the detailed implementation. `parseLeft` does the first pass and `parseRight` does the second.

```
fun parseLeft (env, tenv, nil) = tenv
  | parseLeft (env, tenv, ({name, ty, pos}::r)) =
      let val tenv' = S.enter(tenv, name, T.NAME(name, ref NONE))
      in parseLeft (env, tenv', r) end

fun parseRight (env, tenv, nil) = ()
  | parseRight (env, tenv, ({name, ty, pos}::r)) =
      let val (t, p) = transty(tenv, ty)
          val T.NAME (id, tyRef) = valOf(S.look(tenv, name))
      in
          tyRef := SOME t;
          parseRight (env, tenv, r)
      end
```

Handling recursive function declaration is almost the same by using two pass. First pass is to get the header information as before while the second is slightly different. Since function has its own parameters, before parsing the body, we need to enter the parameters into the variable environment. The following work should be the same as handling type declaration. Of course we need to check the return result of a function is compatible with its definition.

Handling duplicated declaration in type and function declaration

The appendix says that *no two functions in a sequence of mutually recursive functions may have the same name; and no two types in a sequence of mutually recursive types may have the* *same name*. I write a helper function to determine if there are any duplicated declaration. If yes, and I should print out error message. I tried to find a set data structure in SML, but I didn't find one that is appropriate, so I just a list to store all symbols that I have seen so far. I call it `visited`. The idea is pretty straight forward, I compare every element with all the elements in `visited`. If I find a match, which means this symbol has already been declared before, and I will stop search and print out error message. Otherwise, I will continue search till every element is in the `visited` list. Detailed implementation is as follow:

```
fun duplicateDection (visited, nil) =()
| duplicateDection (visited, {name, params, result, body, pos}::r) =
 if List.exists (fn y => y = name) visited then
     error pos ("Duplicate definition of " ^
            S.name(name))
 else (duplicateDection(name::visited, r))
```

Cycle Detection

One problem with recursive type declaration is that it may result in a cycle declaration, which should never happen. The method I adopt is basically the same as how I handle duplicated declaration. I maintain a list called `visited` and I traverse all the way followed by the reference in the `T.NAME` type, I put all references that have seen so far in `visited` and use `List.exists` to see if current element has already been visited. If yes, we find a cycle and should report an error. Besides, we should force one of type in the circle to be a basic type (e.g. T.INT) to break the cycle. Otherwise we may enter a deadlock when trying to get the actual type.

```
fun cycleDection (env, tenv, nil) = ()
  | cycleDection (env, tenv, ({name, ty, pos}::r)) =
      let val ty = valOf(S.look(tenv, name))
          fun helper (T.NAME(sym, tyRef), visited) =
                 if List.exists (fn y => y = sym) visited then
                     (tyRef := SOME(T.INT);
                     // report error
                 else helper(valOf(!tyRef), sym::visited)
            | helper (_, _) = ()
      in helper (ty, []) end
```

Getting the actual type

Though the implementation is trivial, but it is important throughout the whole assignment. We have to keep in mind that in order to handle recursive declaration, we introduce a special type `T.NAME`. But this type is not the actual type. However we need the actual type to check type. So it is quite important to do such transformation. Below is the code.

```
fun actualTy {exp=_, ty=ty} =
   case ty of T.NAME(name, tyRef) =>
       (case !tyRef of NONE => T.INT
           | SOME(t) => (actualTy {exp=(), ty=t}))
   | t => t
```

Since we already break the cycle in cycle detection, we don't need to worry about deadlock issues here.

Force casting

If any expression is erroneous, for example, type mismatch, what should be the type of that erroneous expression? It is actually implementation dependent. But I've encountered problem when facing array creation.

```
type a1 = array of int
type a2 = array of a1
type a3 = array of a2
x3 := a3[9] of a2 [4] of 67       /* err */
test06.tig:41.20:array element type mismatch
test06.tig:41.11:array element type mismatch
test06.tig:41.8:incompatible type in assignment
```

In the example above, we can see that there is one obvious error, which is `a2 [4] of 67`. An error should be definitely reported here. But should we report the next 2 errors? It depends. My original implementation will not report those two chain-errors. Because originally, I still maintain the array type for `a2 [4] of 67`. The sample solution clearly cast it into a basic type like `T.INT`. I modified my implementation to follow the standard error messages. But which one is better, I still cannot figure out right now.

Test

I use the 10 provided test cases to test my program, I add a function in `main.sml` to output the result to a file. Then I use `diff` command to compare my results with standard results.

```
fun writeFile (filename, content) =
    let val fd = TextIO.openOut filename
        val _ = TextIO.output (fd, content ^ "\n") handle e => (TextIO.closeOut fd; raise e)
        val _ = TextIO.closeOut fd
    in () end
```

The result of `diff` is

```
my result
< test10.tig:11.4:Type mismatch in parameter 1
< test10.tig:12.6:Type mismatch in parameter 1
< test10.tig:13.8:Type mismatch in parameter 1
< test10.tig:13.8:Type mismatch in parameter 2
---
std result
> test10.tig:13.8:Type mismatch in parameter 2.
> test10.tig:13.8:Type mismatch in parameter 1.
> test10.tig:12.6:Type mismatch in parameter 1.
> test10.tig:11.4:Type mismatch in parameter 1.
```

The result of first 9 test cases is the same as standard files. The only difference is the sequence of the error message. I think it's because the way we traverse the formals list is different. I think mine is more readable since it follows the increasing line number.