dOvs - Semantic Analysis

Group 9
Miran Hasanagić - 20084902
Jakob Graugaard Laursen - 20093220
Steven Astrup Sørensen - 201206081

October 7, 2015

1 Introduction

This report describes how the semantic analysis module was developed. The semantic analysis is the third component of the compiler. This component takes the Abstract Syntax Tree (AST) from the parser and performs a semantic analysis, also known as type-checking, to produce a typed syntax tree.

First we described how the missing cases where implemented in the provided file `semant.sml` and which other files had to be extended in order to developed the semantic analysis component. Afterwards, the problems encountered and experience gained are described. Then 5 tiger programs are presented, which tests different cases of the semantics analysis.

2 The semantic analysis

The main task of the semantics analysis is that it takes the AST provided by the parser and applies the function `transExp` to it. This function is able to traverse a AST, and create a Typed AST (TAST), based on the file `tabsyn.sml`. This new TAST, is a AST which has be decorated with type information. We will provide furher details on a few parts of the semantic analysis.

The main development in order to create the semantics analysis module, occured inside the file `semant.sml`. We used the approach from the book as well as what was described by the material provided. For example, the nested function `trexp` inside the function `transExp`, is used on all expressions which do not change the environment. Then `transExp` is only used when an expression has to be evaluated with an new, and altered, environment. As an example of this is the case when the body of let expression has to be evaluated with possibly new declarations, or a for loop with a newly defined variabe. Also the `extra` record is used in order to send additional needed information with the relevant functions.

Also a key part of this semantic analysis is the handling of the declarations, especially the recursive declarations of types and functions. We first developed a non recursive version, in order to test other parts of the semantic analysis. Afterwards, it was extended to handle recursive declarations. The recursive declarations are handled by parsing the defintions twice. While we add stuff to their respective enviorments in the first pass, they are far from equal in what is occuring.

2.1 Types and Mutual Recursive Types

They way we handle recursive types is by processing type declarations twice. First time through, we put the types into the type enviorment. More concretly we add the entry `(name, TY.NAME(name,ref(NONE)))` to the type enviorment.

While here, we check if the same name occurs twice in the block. If it does, we report it as an error

Second time through, we use the name to lookup our type in the type enviorment. Then we call transTy on the returned type with our updated type enviorment and the abstract syntax trees type declaration. Trans ty then creates the relevant type if the user wanted to create a record or array type, or fetches a

named type if the user wanted his or her type to refer to an already named type. In all cases, it updates the named types reference to point to the a newly created or fetched type.

While we are in the proccess of creating the types we maintain a list of the names of the new type declarations. After we are done creating, we use this list to check for cyclic type declarations.

The way we detect cycles is simple: We use Floyds cycle detection algorithm. If A has a reference to named type B, we start the algorithm with these as inputs. We then check if B has a reference to a named type C and if so, if it has a reference to a named type D. Then, we allow the pointer to A to jump to B. If there is a cycle, we will find it once the pointers point to the same names.

The only time there won't be a cycle is if we in a long sequence of named types end up at a record or array type. If so, we do stop, since we then can't have a jump step of size 2 at the end.

2.2 Functions and Mutual Recursive Functions

The way we handle functions is a bit simpler, conceptually at least.

Once again, a two phase proccess is used. Firstly, function headers are gathered up and entered in the variable enviorment. Then in the second phase, we go through each function body separately. Firstly, the arguments are added as simple variables of their declared type to the variable enviorment, then we proccess the bodies with the newly updated variable enviorment, containing functions headers as well as the arguments.

After passing, the arguments will drop from the variable enviorment, so only the function headers remain and we continue on.

3 The support modules

The support module `ENV` was extended with the information about the standard library functions as described in Appendix A in the course book. This `ENV` module also was used as part of creating the environment for typs and values during the semantical analysis.

4 Problems encountered & experience gained

One key experience is how to create the semantic analysis in order to detect which parts are not working correctly during development. For this reason Test Driven Development was used, like in our work in the warm up assignment. During the development small unit test where created in order to test single aspects of the module in isolation. This helped in order to identify the source of errors encountered.

Additionally, it led to discussions about how to handle type errors in order not to let an error propagate. The main principle in this approach is to ensure that the semantic analysis can continue even though an error was encountered. Such things can be handled in different ways, and here we had to make a compromise and decision, which are mainly based on the severity of the error. Mostly, we want to report as many errors as possible to the user.

5 5 tiger programs

The five tiger programs presented here are simple, created in order to focus on different aspects of the implemented semantic analysis. Some of our tests cover aspects which provided testcases does not. These tests contain different syntax errors as required by the project description. In order to ensure that this semantic analysis works it is run on the provided testcases by the course.

5.1 break_exp_tests.tig

```
/* Check the three possible states for break.
       in for loop, in while loop, outside loop
       EXPECTION: an error at last break statement
```

```
 */
(for i := 1 to 3 do break; while 2 do break;break)
```

5.2 cycle_in_typedecl.tig

```
/* test recursive typedefintion.
        ERROR: Cycle will be detected with A,B,D */
let
        type A = B
        type B = D
        type D = A
        type t1 = t2
        type t2 = t3
        type t4 = t2
        type t3 = int
        type C = int
        type tree = {value: int, children:treelist}
        type treelist = {hd:tree, tl:treelist}
in
end
```

5.3 unassignable.tig

```
/* test for immutable for ID.
        ERROR: i is immutable, and can not be assigned to.*/
(for i := 1 to 3 do i:=2)
```

5.4 recursive_type_usage.tig

```
 /* test recursive type usage.
        PASS: No Errors should be produced */
let
  type tree = {value:int, children:treelist}
  type treelist = {hd: tree, tl:treelist}
  var c := tree{value = 0, children = nil}
  var children := treelist{hd = c, tl = nil}
  var tree := tree{value = 1, children = children}
in tree
end
```

5.5 using_a_record_create_in_a_record.tig

```
/* Testing to allow to create a
* inside a record,
* if the field name is a record */
let
        type A = {c:int}
        type C = A
        type B = {d:A}
```

```
        var a := B{d = C{c = 2}}
in
        a
end
```

6 Conclusion

This report presented the development of the semantics analysis, which creates a typed AST. Additionally the gained experience was described. Finally, additional tests provided confidence that this semantic analysis is working correctly and also reports correct error messages.

This report showed the implementation of the semantic analysis. Furthermore some the problems encountered where described.