dOvs - IR-translation

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

November 3, 2015

1 Introduction

This report described the IR-translation phase of the compiler, which is the fourth phase of the whole compilation process. This component takes the type-check output from the `semant.sml`, and creates an Intermediate Representation (IR) as output. How this functionality was achieved is described below.

The first section described how the IR-translation is done, and afterwards the following section describes the additions made to the support modules. Then problems encountered during the development are highlighted. Finally, five tiger test programs are shown, and concluding remarks are provided.

2 The IR-translation

The goal of the IR-translation is to create the IR of the typed abstract syntax tree from the previous phase. The `irgen.sml` takes the output from the `semant.sml`, and creates the IR tree as output. Its main structure is similar to the `semant.sml`, in that recursive function are used to transverse the typed abstract syntax tree. However, for the IR we do not need to type-check, but only focus on creating the IR. This IR is created by using the support module `translate.sml`, which is able to create the IR constructs.

3 The support modules

One support module is `translate.sml`. The transformation themselves of the different tiger language constructs, such as `for` and `if-then`, are inside this file. The main functionality of this file is described in chapter 7 in the book. This IR-translation is done for each construct of the tiger language, as mentioned above. In order to support the different constructs, the suggestions in the book where followed. An example is to translate a for-loop to a let expression in the IR tree. Also the unCx, unEx and unNx functions where used at relevant places in order to get the correct constructs. As an example in the if IR translation. Additionally, this file has support for creating a new level, which is used in the function declaration. Basically, it has support for all the low-level constructs, which are close the assemble language.

Another support module is `irgenenv.sml`, which holds the environment. This environment is extended with the functionality described in chapter 6. This contains both the information needed, and the functionality described in chapter 6, such as storing the level of a function declaration.

4 Problems encountered & experience gained

4.1 General Experience

4.2 Function Declaration

4.3 Record and Array pointers

5 5 tiger programs

The five tiger programs presented here are simple, created in order to focus on different aspects of the implemented IR generator. Some of our tests cover aspects which provided test-cases do not.

TODO: Make new test

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 IR generator in `irgen.sml`, which creates an IR tree from the typed abstract syntax tree produced by the `semant.sml`. The main functionality was described, and how some modules are used in order to support the IR generation. Additionally the gained experience was described. Finally, additional tests provided confidence that this IR generator is working correctly.