dOvs - IR-translation

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

November 4, 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 IR transformations 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 were used at relevant places in order to get the correct constructs. An example of this is seen in the `if` IR translation. Additionally, this file has support for creating a new level, which is used in function declarations. Basically, it has support for all the low-level constructs, which are close the assemble language.

TODO: Mention the extra function for headers

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

General it was hard to understand the IR code in the beginning. However, reading in the book, and looking at the code provided in `translate.sml`, help at understand it quicker. Afterwards, the simple constructs

were made, such a integer expression and binary operators, and tested. Next the more challenging constructs could be implemented. Some of the more challenging are addressed below. Additionally, a function called `prIR` was implemented inside `translate.sml`, which could print the IR tree. This IR could then be run by the online tool provided, in order to validate the expected output from the generated IR.

4.2 For Loops

In order to prevent an overflow of the control variable (here i) would lead to an infinite loop, we had to modify the test used in a for loop.

The general idea is this: i ¡= high, if and only if i - 1 ¡ high. The overflow happens if we increment i by 1 and i was maxInt(the highest integer value we could represent). If high wasn't maxInt, then the test fails once i = high + 1, since high + 1 didn't overflow. However, if high was maxInt, then high + 1 ¡= high in our code, so i would always fullfill i ¡= high.

But, once i overflow, i - 1 would be maxInt, and the test i - 1 ¡ high would fail, preventing the overflow of i to cause an infinite loop.

In pratical terms, the foor loop works like this:

First, set i = low and store the value of high in the register hiReg. Test if i ¡= high. If so, execute the body. From here on out, we store the current value of i in the register testReg. Increment i. Test if testReg ¡ hiReg. If not, jump to the done statement. If so, start over, first with executing the body. Notice we in first round aren't concerened if the values of low or high were overflowing, we are only concerend about i overflowing when we increment it in the loop.

4.3 Function Declaration

They way we generate IR code for functions is by parsing a function declaration block twice.

First, we gather functions names and the number of parameters for the functions, as well as creating a level for each. This info entered into the variable enviorment, which we use for the second pass.

In the second pass, we will generate the function bodies. Firstly, we enter the location of the arguments into the variable enviorment. We then use transExp on the bodies, where we now set the level we are working on to be the level belonging to the function.

The reason for the two phase proccess is that we have mutual recursive functions. Since we first gather info about the parameters as well as where the function will eventually live, we can use this in call expressions, without knowing what the bodies consists off.

The second pass ensures that once a function is called, it will execute correctly.

4.4 Record and Array pointers

Here we use the realization that the mememory addresses stored in the pointers are actually ints. Since we only check if the pointer adresses are equal or not equal, we are not concerened about wehter ints are signed or not signed.

5 5 tiger programs

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

TODO: Make new test

5.1 test01.tig

```
/*
Array test for equal of equal things and unequal for unequal things
*/
let
```

```
        type A = array of int
        var b := A[2] of 0
        var c := b
        var d := A[1] of 3
in
        b = c & c <> d
end
```

5.2 test02.tig

```
/*
Test for loops,
that we update control variable correctly,
and that we do not loop to much,
and that we do no enter loops where low > high
*/
let
        var a := 0
in
        for i := 1 to 0 do a := 99; for i := 1 to 3 do a := a + i; a
end
```

5.3 test03.tig

```
/*
Checks if we put the arguments in correct order
*/
let
        function sub(a:int,b:int):int = a - b
in
        sub(3,2)
end
```

5.4 test04.tig

```
let
type rectype = {name: string, address: string, id: int, age: int}
type arrtype = array of rectype
var arr := arrtype [5] of
                  rectype{name="aname", address="somewhere", id=0, age=0}
in
    arr[3].name := "kati";
    arr[1].age := 23;
    arr[1].age
end
```

5.5 test05.tig

```
/* test for recursive functions
 * current version has an infinite loop for negative cases
*/
let
        even(i:int):int = if i = 0 then 1
                                                else if i = 1 then 0
                                                else odd(i-1)
        odd(i:int):int = even(i-1)
in
        even(4) & 0 = odd(4)
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, like `translate.sml`, 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.