CSCI 355, Compiler Construction (Translator for Final Project)

Hans de Nivelle

December 6, 2020

1 Introduction

The final project of this course consists of implementing a compiler for the language below, which we call $C\flat$ (pronounce 'C flat') for the time being. In order to make it digestible, we cut the final project into 3 parts:

1. Implement a parser, using **jflex** and **CUP**.
2. Implement a semantic analyzer.
3. Implement a translation function.

We will give you an implementation of the register/stack machine that was presented in the slides, so that you can test your translation function. We have now arrived at the third part. The task is to complete function `regtranslateExpr` in class `Translator`.

2 Simulator

Final goal is to translate $C\flat$ functions into a register/stack machine. Package `simulator` contains an implementation of the register/stack machine. The differences with the slides are small. In the implementation, instruction alloc has form

$$\alpha\colon \mathbf{pntr}(T) \leftarrow \text{alloc } T$$

It stores the allocated position immediately in a register. This register can be used later for referring to the local variable, instead of using

$$\alpha\colon \mathbf{pntr}(T) \leftarrow \#i.$$

In the code that we provided, this approach is not used. We always use # and skip the later variables.

We had to rename the # instruction, because # cannot be used as Java identifier. The instruction is now called 'Variable'. Instead of giving only position

i, one needs to give the types of the skipped variables. If the first n variables on the top of the variable stack have types $T_1, \ldots, T_n$, and you need a pointer to the $(n+1)$-st variable with type T_{n+1}, the variable statement takes form

$$\alpha\text{: }\mathbf{pntr}(T_{n+1}) \leftarrow \text{variable }\ T_1 \cdots T_n.$$

We changed dealloc in a similar way. The instruction

$$\text{dealloc }\ T_1 \cdots T_n.$$

removes the last n variables from the stack.

Instructions are organized in blocks. A block is a linear sequence of instructions. A block has a label, and can be entered only through this label. A block can be exited by a goto, by an iffalse, or by return. Just falling over the end is not allowed and will cause an error message.

A function is a mapping from labels to blocks. There must be one block with label 'entry'. This is where execution starts when the function is called. There are no conditions on the names of the other blocks.

The complete program is a mapping from identifiers to functions. If you want to see how programs are constructed and what instructions exist, you can have a look at `simulator.Examples`.

In this task, you don't need to create functions or blocks, because this is taken care of in `translation.Translator.translateStatement`, which is given to you. All you need to do is call the function `emit( Instruction )`, which will append the instruction to the block that is curently being constructed. If you want to understand how blocks are created, you can have a look at the translations of **if** and **while** in `translateStatement( )`.

The simulator has a `trace` instruction that turns on tracing. It remains on until the user types `q` to quit, or `n` to turn the tracing off.

We recommend that you add the `trace` command to the grammar and the tokenizer. It will be translated into a trace instruction by `translateStatement`. We added a few instructions to enable printing:

printstring "string"	prints the string
print α	prints register α
newline	prints a newline

Initially, we planned to handle newline by allowing `\n` in strings, but it causes so much headache, that we decided to add the `newline` command. Just add it to the grammar and the tokenizer, and `translateStatement( )` will translate it. Don't spend your precious time to escape characters in strings.

3 Keeping Track of Variables

It turned out that keeping track of variables is extremely tricky. The slides make it look simpler than it is in real. The `VarStack` data structure in package `semantics` keeps track of it. (Note that it was called `VarStore` before.)

Variables are addressed by the `Variable` instruction (called # in the slides.) In order to get a correct pointer to the variable, one needs to skip the variables that were declared after it. This includes implicit variables that were created for storing intermediate results in memory. For example, in the function call `n = f(n,n,n,n)`, each `n` needs to skip the `n` that were created before it.

Variables in memory are created by the function `memtranslate`. It creates code that stores the result in memory, but it also creates an anonymous variable in `localvars` of type `VarStack`. When a `Variable` instruction is generated, `regtranslateExpr` will look at `localvars` to see which variables need to be skipped. Messing up the variable stack administration causes bugs that are very hard to find, because they are discovered much later than the mistake is made.

In order to keep the task reasonable, the function `memtranslateExpr( )` is given, together with the part of `regtranslateExpr( )` that handles variables. You still have to look at it and understand it. The code follows the slides quite well.

4 Handling Fields of Structs

Accessing fields of structs is easy in principle: All one needs is a pointer to the struct, so that one can add the index of the field to it. The result is a pointer to the field. The add instruction is created by calling

```
new Instruction.Binary( "add", resultreg, fieldtype,
                        structreg, indexreg ) );
```

Variable `fieldtype` must have form `type.Pointer( type )`, where `type` is the type of the field. If semantic checking went well, it just the `trueType( )` of the `ast.Tree`. We added a field `int index` to `ast.Select`, which you should assign during semantic analysis.

We made some changes in the treatment of fields in `StructStore`, because it turned out quite unpleasant to use:

- We created a new class `FieldArray`. This class replaces `Field[]`. One can view this class as a mapping from field names to types and indices, and one should consider replacing it by `VarStack`. We didn't do it.
- If you want to find the index and the type of a field during semantic analysis, first call `StructStore.get( )`. This returns the `FieldArray` of the struct it it exists, and `null` otherwise. In the `FieldArray`, call `getIndex( )`. This returns the index of the field if it exists, and `nrFields( )` otherwise.

  Now you know the index of the field. You can store in the tree, which must have type `ast.Select`. In order to get the type, use `getType`.

We are somewhat sorry for the fact that you have to rewrite parts of the semantic checker that you wrote already, but unfortunately that is the reality of programming. It is better to correct than to keep working with bad designs.

5 Code Creation

In the slides, most of the code for built-in operations is created by means of definitions of form

$$\beta\colon U\ \mathbf{op}(\ \beta_1\colon U_1, \ldots, \beta_n\colon U_n\ ) \Rightarrow \text{ implementation of } \mathbf{op}.$$

It seems very easy. All one has to do is look up the definition of **op**, replace $\beta_1, \ldots, \beta_n$ by the concrete registers, and emit the implementation with the registers replaced. This looked like a very smart idea when I wrote the slides, but it completely failed in practice. It turned out unpleasant to implement register replacement in existing instructions, and one needs a dedicated data structure for the definitions, together with a mapping from built-in operators to definitions.

But this is not the real problem. The real problem is that in reality, many built-in functions have very much in common, and one looses the possibility of sharing when one has to create separate definitions for each of them. For example, the comparison operators `eq,ne,lt,gt,le,gt` can be translated in the same way for every type, and there are 5 of them. So instead of creating 30 definitions in a table, one can write 4 lines of code that emit the proper instruction.

Replacing registers in existing instructions is not necessary, because one can call the constructors with variables containing the proper registers, for example:

```
java.lang.String resultreg = registers. create( );
emit( new Instruction.Binary( binop, resultreg, new type.Bool( ),
                              reg1, reg2 ));
return result;
```

The same applies to other groups of built-in operators that have much in common, like `add,sub` and `mul,truediv,mod`.

With a little bit of creativity, `[ppx]`, `[mmx]`, `[xpp]`, `[xmm]`, can be treated by a single block of code for all types.

6 Translating Statements

Statements are translated by recursively walking through them and translating the statements on the way. It is not difficult to implement, but you don't have to write it, because we want to keep the task reasonable. You should still look at the code of `translateStatement( )` and make sure that you understand it. You should understand how **while**, **if** and **return** are handled. Whenever a compound block is entered, the size of `localvars` is remembered, and it is restored when the block is exited.
Note that there is a bug in the compiler. Statements of form

```
if x == 3 then
   y : integer
```

```
else
   z : double;
```

are possible, but produce meaningless code. Probably this should be handled by forbidding it in `semantics.SanityChecks`, because it really is insane.

7 Translation into Register

Function `regtranslateExpr( expr )` emits code that evaluates the value of `expr` into a register. It returns the register. This function is partially given, and you have to complete the rest of it, so that you can translate and execute most of `test.cfl`. In order to create a new register, use `registers.create( )`. It returns strings of form `R0,R1,R2,...` In order to create a statement, use `emit( )`. There are some examples in the given code, how it is used.

The harder cases, like calls of defined functions and variable look up, are given.

8 Warnings and Remarks

- You will need to replace a lot of code that was given to you for semantic analysis:
  - all of package `ast`,
  - all of package `type`,
  - function `semantics.FunctionChecker.checkStatement( )`,
  - `Main.java`,
  - `cflat/Program.java`,
  - all of package `simulator`.

  Your will have to fix your solution to semantic analysis accordingly.

- There are some changes in the `VarStore` datastructure:
  1. It is renamed into `VarStack`.
  2. I didn't find it a good idea anymore that `VarStore` computes the size of a variable when it is pushed. Because of that, I replaced `push( String, StructStore, Type )` by `push( String, int, Type )`. The caller now has to compute the size.
  3. After that, I realized that during the semantic checks, sizes of variables don't matter, so I put a 1 in the place of every size.
  4. Finally, I realized that also during translation, sizes are also never used, because `allocate, deallocate` and `variable` compute sizes by themselves. This is how it was done in the slides. So, we remain with an unused field and unused functionality.

- `main` tries to translate all functions in `test.cfl`. If it succeeds, it calls function `main` in the resulting translation. You can replace it by any other function, but it should be a function without arguments and returning `type.Void( )`. Otherwise, you would have to prepare the parameters in memory, which is complicated.

- If you feel like a true hero, you can try to translate `??(c,s1,s2)`. You will need two versions, one in memory and one in register. You can look at the translation of `[if]` in `translateStatement`. It is not required to implement `??` in order to get full credit, but it may be fun.

- The handling of constants turns out problematic. One would like to be able to write `constant minus1 = -1;` In order to make this possible, the tokenizer has to recognize `-1` as a single token. Unfortunately, this makes it impossible to parse the expression `a-1` correctly. The only possible solution is to allow some arithmetic expresssions in constants, and to evaluate them. This is not difficult, but it is too much for this course.

- There is no 'over the end' return. In most programming languages, **void** functions can return by simply falling over the end. In our implementation this is not possible. The stack machine throws an exception when it reaches the end of a block. Making 'over the end' return possible is harder than it seems: You need to keep track of depth during translation, and insert a return when you reach end of a statement at depth 0 in a function returning **void**.