TP3 : Introduction to Code Generation — TP NOTÉ —

Thibaut Benjamin[1] and Burkhart Wolff[2][0000−0002−9648−7663]

[1] LMF, Université Paris-Saclay, France
thibaut.benjamin@universite-paris-saclay.fr
[2] LMF, Université Paris-Saclay, France
wolff@lmf.cnrs.fr

Abstract. This TP addresses compiler backends in a broad sense and considers the example of the ARM Cortex M Instruction Set Architecture in particular.
A small interpreter for Cortex-M-Instructions is developed — just another type of interpreter, no reason to shy away from low-level instructions and resulting compilation problems.
A formal model of a fragment of the Cortex-M-Instructions is presented in Isabelle/HOL, its 'semantics', i.e. its implementation is be implemented in SML in order to increase the understanding of the Cortex-M processor.
Finally we specify the code generation for arithmetic and boolean expressions as an attributation in the Cortex-M Isabelle/HOL model. (no knowledge of proof techniques required).
The accompanying material can be found on the the following site: [1]

Keywords: Attribute Grammars, Specifying Coders, SML Programming

1 Recall: Expression Evaluation.

We have seen already a simple evaluator like this in the TP 1. Recall:

ML-file‹*tp1.sml*›

Call of a function defined in `tp1.sml`:

ML‹
```
fun eval-expr (const k) env = k
  | eval-expr (var s) env = the(Symtab.lookup env s)
  | eval-expr (mul (e1,e2)) env = (eval-expr e1 env) * (eval-expr e2 env)
  | eval-expr (add (e1,e2)) env = (eval-expr e1 env) + (eval-expr e2 env)

fun eval skip env = env
    | eval (ass(s,e)) env =  (Symtab.update (s, eval-expr e env) (env))
    | eval (seq(s1,s2)) env = eval s2 (eval s1 env)
```
›

Example:

ML‹
```
val prog = seq (ass(a, const 3), ass(b, mul((var a), (var a)))) ;
eval prog Symtab.empty
```
›

2 Part I : The ISA ARM Cortex M3

2.1 SML Basics

A "Simulator" of an "Instruction Set Architecture" (ISA) [3] is just an interpreter following the above scheme, the ARM Cortex M3 is no different from that. SML helps us that it has support for data-structures like `Word32` which are the basic building blocks of such simulators. (SML was developed for a large period at Cambridge, the site of the ARM company, you see).

The Basics: The datatype `Word32` comes with the operations:

1. `Word32.fromInt` and `Word32.toLargeInt`. So: `Word32.fromInt 13` is denoted "0wxD", and `Word32.fromInt (32767 * 2 + 1)` "0wxFFFF"
2. the operations shift-left and shift-right : `Word32.<<` and `Word32.>>`
3. the boolean word operations `Word32.notb`, `Word32.xorb` and `Word32.andb`
4. the word artihmetic operations "+","-", "*", etc.

We can also model a store of a machine by the data structure `Array` with the lookup operations `Array.array`, the (destructive, imperative) operation `Array.upd`, and the operation `Array.nth`.

ML‹*val memory = Array.array* (*10*, *Word32.fromInt 0*);
val - = Array.upd memory 3 (*Word32.fromInt 7*);
memory;
val no3 = Array.nth memory 3›

Remark: Of course, you can `open Word32;` in the SML environment; but then the usual infix operations on integers like "+" and "*" are overshadowed and have to be referenced by `Int.+` or `Int.*`, respectively. So `open Word32;` has the effect that we have to write `Word32.fromInt ((Int.+)((Int.*)(32767,2),1))` unless that we make `open Int;` again. You might want to control the effect of an "open" by "local open Word32 in ... end" statements in SML.

WARNING: The Isabelle document model is a purely functional one. Introducing destructive data-structures or imperative variables might fool Isabelle's mechanics to re-evaluate and track the consistency of the document you edit and inspect in the output-window. For this reason, The SML data-structure `ref` for imperative variables

— example: `val store = ref (35); store := !store + 2; !store` —

is hidden in the structure `Unsynchronized` and is discouraged to normal Isabelle use.

In our case, we ignore this, and have to provide operations that refresh our state variables. Or force the re-creation of imperative constructs by explicitly re-evaluating them.

[3] `https://developer.arm.com/documentation/dui0552/a/the-cortex-m3-processor/programmers-model/core-registers?lang=en`

2.2 An ARM Cortex M Model: Target ISA as AST

We use the following Isabelle/HOL model for the fragment of the Cortex M we want to work with:[4]

datatype *reg* = *R0* | *R1* | *R2* | *R3* | *R4* | *R5* | *R6* | *R7* | *R8*
| *R9* | *R10* | *R11* | *R12* | *SP* | *LR* | *PC*

datatype *opcode* = *MOV* | *MVN* | *MOVW* | *MOVT* | *LDR* | *STr*
| *ADD* | *SUB* | *MUL* | *UDIV* | *SDIV*
| *B* | *BX* | *BLE* | *BEQ* | *BNQ* | *CBZ* | *CBNZ* | *CMP*

datatype *varg* = *none* (———)
| *reg1 reg* (‹⦃-⦄› [*0*]*20*)
| *reg2 reg reg* (‹⦃(-), (-)⦄› [*0,0*]*20*)
| *direct nat* (‹# -› [*10*]*20*)
| *reg-ind reg* (‹(|-|)›[*0*]*20*)
| *reg-ind-offset reg nat* (‹(|- , -|)›[*0,0*]*20*)

datatype *com* = *label nat* (⟨*L* -⟩)
|*branch opcode nat* (⟨- *L* -⟩)
|*instruction opcode reg varg* (⟨(-) (-), (-)⟩ [*0,0,20*]*0*)

A syntactic test for the Isabelle model of ISA:

term‹[⟨ *MOV R0*, *#11*⟩,
⟨ *MOV R1*, *#2560*⟩,
⟨ *MVN R2*, *#4*⟩,
⟨ *MOVW R3*, *#CODE*⟩,
⟨ *MOVT R3*, *#FEED*⟩,
⟨ *MOV R0*, ⦃*R1*⦄⟩]›

term‹[⟨ *LDR R3*, (|*R1*|)⟩,
⟨ *STr R3*, (|*R1*|)⟩,
⟨ *LDR R3*, (|*R1*, *4*|)⟩,
⟨*L 2*⟩,
⟨ *ADD R5*, ⦃*R3*, *R4*⦄⟩,
⟨ *BLE R5*, (|*R3*, *4*|)⟩]›

2.3 A partial implementation of an ARM Cortex M3 ISA evaluator in SML

A fragment of this machine is given in the accompanying file `tp3.sml`. it separates the assembler program from the actual memory (which is a simplification that will do for our purposes). You may load it by:

ML-file‹*tp3.sml*›

A very basic test for this SML implementation is

[4] Note the slight differences to notation used on the slides in order to avoid ambiguities with HOL-library notation for strings, lists and products.

```
ML‹
reboot-processor();
run-processor[ins(LDR, r0, direct 5),
              ins(ADD, r1, reg2(r0,r0))];

(*consider the output window: *) !R0; !R1;›
```

Task: Extend the given ARM ISA interpreter in SML.

1. Extend the implementation to the actual number of registers
2. Add the interpretation of arithmetic operators
3. Add the boolean operators
4. Add the basic commands for branching and jumps. (you may add a function that retrieves from the *program* the positions of labels used for direct or conditional jumps)
5. Provide a reasonable test-set of your extended implementation.

3 Part II: Modeling the Code-Generation by Attributation of Expressions

3.1 Recap

Recall the simple code-generation model for some boolean expressions discussed in the class:

```
datatype expr = LVAR string
              | And  expr expr
              | Or   expr expr
              | Not  expr
```

with the attributes:

```
consts  Env_in   :: expr ⇒ (string⇒reg) (env↓⟦(-)⟧ [70]70)
consts  Lab_in   :: expr ⇒ nat          (lab↓⟦(-)⟧ [70]70)
consts  Lab_out  :: expr ⇒ nat          (lab↑⟦(-)⟧ [70]70)
consts  Code_out :: expr ⇒ com list     (code↑⟦(-)⟧ [70]70)
```

allowing the attributation rule:

```
prop ‹e = And a b ⟹
        lab↓⟦a⟧ = lab↓⟦e⟧ + 1 ∧ lab↓⟦b⟧ = lab↑⟦a⟧ ∧ lab↑⟦e⟧ = lab↑⟦b⟧ ∧

          env↓⟦a⟧ = env↓⟦e⟧ ∧  env↓⟦b⟧ = env↓⟦e⟧ ∧

          code↑⟦e⟧ = code↑⟦a⟧ @ [⟨CMP,r7 #0⟩]
```

@ [⟨*BEQ L lab*↓⟦*e*⟧ + *1*⟩]
@ *code*↑⟦*b*⟧
@ [⟨*L lab*↓⟦*e*⟧ + *1*⟩]›

The complete source of this model can be found under `tp3_model.thy`[1].

3.2 Towards a Coder for a larger Expression Language

In the following, we generalize the expression language as follows:

type-synonym *Ident* = *string*
datatype *Type* = *int* | *bool*

datatype *Expr* = *true* | *false* | *const* ‹*int*› | *var* ‹*Ident*›
| not_E ‹*Expr*› | and_E ‹*Expr*› ‹*Expr*› | or_E ‹*Expr*› ‹*Expr*›
| eq_E ‹*Expr*› ‹*Expr*› | leq_E ‹*Expr*› ‹*Expr*›
| *add* ‹*Expr*› ‹*Expr*› | *mul* ‹*Expr*› ‹*Expr*›

We will assume that the terms are context correct (i.e. the attribute *CC*↑⟦*e*⟧ from the context analysis discussed in the class is true anywhere), and the attribute *tenv*↓⟦*e*⟧ is also provided from the context analysis.

Task:

1. Specify the code-generation by an attributation of the expression language.
2. Provide some test examples.
3. You may do this by paper and pencil or inside Isabelle, modifying the given model. However, you might be taken responsible for typing errors in a paper-and-pencil version.

4 Rendu : Report

Write a report on your results from Part I and Part II, providing explanations and some documentation. You may use screenshots whenever you might find this appropriate.

The work can be done by binome and has to be delivered the 26th of february per mail at your responsable TP.

References

1. Wolff, B.: Teaching Website: PolyTech Compiler Course. `https://usr.lmf.cnrs.fr/~wolff/teach-material/2025-2026/ET4-Compil/index.html` (2025), [Online; accessed 8-Dec-2025]