Compilation Project

User Documentation
Les Quatre MousquetEngineers
Master M1 MOSIG, Grenoble Universities

Cristian HARJA Yassine JAZOUANI Lina MARSS0
Clément MOMMESSIN

29/01/2015

1 Introduction

This is a group project (part of the MoSIG graduate program) in which we are required to implement a compiler for the min-caml language, targeting the ARM architecture. We chose to implement our compiler in Java.

2 Compilation and execution

2.1 Building

In order to build and run this project you need to have installed Java and Apache Maven on your machine. Make sure the `java` and `mvn` commands are in your PATH. You also need an Internet connection the first time you build the project and approximately 50 MB of disk space (for Maven to do its job).

To compile our program, just execute `mvn package`. This will download all dependencies, generate the parser, run the unit tests and finally, compile the project. If successful, a `target/mini-camel.jar` file will be created.

2.2 Running

To run our program, execute `java -jar target/mini-camel.jar` (after compiling the project) followed by any command line options that are specific to our application. By running the application with no command line arguments, you can see a brief description of the available commands. For example: `java -jar target/mini-camel.jar -v` will print the current version number of our application.

2.3 Test suite

To run the unit tests bundled with our source code, execute `mvn test`. This is also done automatically when you compile the project with `mvn package`.

3 Command-line options

```
java -jar mini-camel.jar <action> [-o <output>] [<input>]
```

In case of an error in the source code, or a typing error, or simply an exception in our compiler, our program prints the list of errors (and warnings) on stderr. The process will terminate with exit code 0 in case of success (the action was carried out successfully and an output was printed to stdout), or exit code 1 in case of an error.

3.1 Actions

Options	Descriptions
-h	Displays usage instructions
-v	Displays the version number
-p	Only parse the input (no further processing)
-t	Only perform type checking (after parsing)
-a	Print the Abstract Syntax Tree
-A	Print the pre-processed AST
-I	Print compiled code in Intermediate Representation (IR)
(nothing)	Perform all steps and generate ARM assembly

3.2 Other options

Options	Descriptions
-o <output>	Optional. Specifies an output file; default is stdout.
<input>	Optional. Specifies the input file; default is stdin.

4 Example usage

We will abbreviate `java -jar target/mini-camel.jar` by `mini-camel.jar`. The following sample is found in `src/test/resources/mini_camel/tests/fib.ml`, which we will abbreviate by `fib.ml`:

```
let rec fib n =
  if n <= 1 then n else
  fib (n - 1) + fib (n - 2) in
print_int (fib 30)
```

Output of executing `mini-camel.jar -A fib.ml` (pre-processed source):

```
let rec fib_1 n_2 = (
  if (n_2 <= 1) then n_2
  else (
      (fib_1 (n_2 - 1)) + (fib_1 (n_2 - 2)))
  )
in
  print_int (fib_1 30)
```

```

```

Output of executing `mini-camel.jar -I fib.ml` (intermediate code):

```
# Function: fib_1
# Closure: []
# Arguments: [n_2]
# Locals: [ret, tmp7, tmp6, tmp4, tmp3, tmp5, tmp2, tmp1]
  tmp7 := 1
  IF (n_2 <= tmp7) THEN l0 ELSE l1
  l0:
  ret := n_2
  JMP l2
  l1:
  tmp3 := 2
  tmp4 := n_2 - tmp3
  tmp6 := CALL fib_1[tmp4]
  tmp1 := 1
  tmp2 := n_2 - tmp1
  tmp5 := CALL fib_1[tmp2]
  ret := tmp5 + tmp6
  l2:
  RET ret

# Function: _main
# Closure: []
# Arguments: []
# Locals: [tmp9, tmp8]
  tmp8 := 30
  tmp9 := CALL fib_1[tmp8]
  CALL print_int[tmp9]
  RET
```

Final ARM code (summary), obtained by executing `mini-camel.jar fib.ml`:

```
fib_1:
  @ arguments
  @    n_2 -> r12+40

  @ prologue
  SUB sp, #4
  STR lr, [sp]
  stmfd sp!, {r3 - r10}
  SUB sp, #4
  STR r12, [sp]
  MOV r12, sp

  @ tmp7 := 1
  LDR r3, =1

  @ IF (n_2 <= tmp7) THEN l0 ELSE l1
  MOV r2, r3
  CMP r1, r2
  BLE l0
  BAL l1
l0:

  @ ret := n_2

```

```
  @ JMP l2
  BAL l2
l1:

  @ tmp3 := 2
  LDR r5, =2

  @ tmp4 := n_2 - tmp3
  MOV r2, r5
  SUB r0, r1, r2
  MOV r6, r0

  @ tmp6 := CALL fib_1[tmp4]
  @ push argument 0
  SUB sp, #4
  MOV r0, r6
  STR r0, [sp]
  @ call, free stack and set the return value
  BL fib_1
  ADD sp, #4
  MOV r7, r11
  @ end call

  @ tmp1 := 1
  LDR r8, =1

  @ tmp2 := n_2 - tmp1
  MOV r2, r8
  SUB r0, r1, r2
  MOV r9, r0

  @ tmp5 := CALL fib_1[tmp2]
  @ push argument 0
  SUB sp, #4
  MOV r0, r9
  STR r0, [sp]
  @ call, free stack and set the return value
  BL fib_1
  ADD sp, #4
  MOV r10, r11
  @ end call

  @ ret := tmp5 + tmp6
  MOV r1, r10
  MOV r2, r7
  ADD r0, r1, r2
  MOV r11, r0
l2:

  @ RET ret
  @epilogue
  MOV sp, r12
  LDR r12, [sp]
  ADD sp, #4
  ldmfd sp!, {r3 - r10}
  LDR lr, [sp]
  ADD sp, #4
  MOV pc, lr
```

5 Features of our compiler

5.1 Part of min-caml supported

As of writing this report, the following features of min-caml are fully supported by our compiler:

- Type checking & inference
- Integer computations
- `let`-binding
- Basics and recursive functions
- Array operations

Supported, but insufficiently tested (might break):

- Tuples (and `let (x, y, ...) = ... in ...`)
- Closures
- ARM code generation involving closures

5.2 Functionality implemented

The following algorithms are implemented as part of our compiler:

1. Detection of free variables (+ unit tests)
2. Type checking (+ unit tests)
3. Alpha Conversion (+ unit tests)
4. Beta Reduction (+ unit tests)
5. Constant Folding (tested manually)
6. Function Inlining (tested manually)
7. K-normalization
8. Closure conversion
9. Intermediate Code Generation
10. Assembly Code generation (integer computation, print_newline and print_int)

5.3 Limitation of our compiler

- Inlining for recursive function: It's surely possible to transform any recursive function into an iterative one, and then apply inlining. Because of a lack of time, we didn't implemente inlining for all functions.
- We don't compile float operation ARM assembly for floating point operations, but they are supported in the other stages of the compiler (up to and intermediate code generation with the `-I` switch)
- Function currying is not supported; however, because it exists in the type system, the user may be able to partially apply functions (though this would lead to erroneous assembly code). We sometimes detect this while doing code generation and throw an exception. We can't always detect this (when calling a closure function) so this is a problem.

5.4 Optimisation of our compiler

1. Detection of free variables
2. Type Checking
3. Alpha Conversion
4. Beta Reduction
5. Constant Folding
6. Inlining
7. Elimination of unused let-expression
8. K-normalization
9. Intermediate Code Generation
10. Assembly Code generation (without float)