Group Assignment

Compilers

Oliver Mikkel Winther Cristensen (klw741)

Mathias Brndum Leroul (swn848)

Sebastian O. Jensen (gjx653)

Turn-in date: 18-12-2015

Contents

Contents of the Report
It is largely up to you to decide what you think is important to include in the report, as long as the following requirements are met: Your report should justify all your changes to the compiler modules, in particular, the lexer, parser, interpreter, type checker, machine-code generator, and the optimization modules. All major design decisions should be presented and justified. When evaluating your work, the main focus will be on verifying that your implementation of the language is correct. While we do not put particular emphasis on compiler optimizations in this course, we will also evaluate the quality of the generated code: if there are obvious inefficiencies that could have been easily solved you will be penalized, as they testify either wrong priorities or lack of understanding. You should not include the whole compiler in your report, but you must include the parts that were either added, i.e. new code, or substantially modified. Add them as code listings, and use the appendix if they get too big. Ideally, we should not need to read your source code. Your report should describe whether the compilation and execution of your input/test (FASTO) programs results in the correct/expected behavior. If it does not, try to explain why this is. In addition, (i) it must be assessed to what extent the delivered test programs cover the language features, and (ii) if the implementation deviates from the correct/expected behavior than the test program(s) should illustrate the implementation shortcomings to your best extent. Known shortcomings in type checking and machine-code generation must be described, and, whenever possible, you need to make suggestions on how these might be corrected. The report should not exceed this document in size, and should have an appropriate level of detail. You might be penalized if your report includes too many irrelevant details.

1 Introduction

We're given the grand task of

2 Implementation

Each implementation step is presented in the order, which the the code is passed through: Lexer → Parser → Interpreter → Type Checker → MIPS Code Generator

Notes

- Implemented Boolean Litterals: true and false.

  Added test files bool.fo, bool.in, bool.out

2.1 True/False

In the first task, we're given the sub-task to implement the boolean operators True and False. We start out by creating the following test files:

File name	Code

bool.fo	`fun int boolTest(int a) = if a == 1`
	`then 1 else 0`
	
	`fun int main() =`
	`  let a = read(int) in`
	`  write(boolTest(a))`

bool.in	`1`
bool.out	`1`

Then we implement the True/False token in our parser as well as add True and False to the expression list

File name	Code
Parser.grm	`%token <(int*int)> TRUE FALSE`
Parser.grm	`| TRUE  { Constant (BoolVal true, $1)}` `| FALSE  { Constant (BoolVal false, $1)}`

And finally we add the boolean constants to our MIPS code generator

CodeGen.sml

`| Constant (BoolVal b, pos) =>`
`  if b then`
`  [ Mips.LI (place, makeConst 1) ]`
`  else`
`  [ Mips.LI (place, makeConst 0) ]`

2.2 Multiplication and Division

Moving on to the implementation of multiplication and division, we again start out with creating test files.

File name	Code

Mult.fo	`fun int Mult(int a) = a * 2`
	
	`fun int main() =`
	`  let a = read(int) in`
	`  write(Mult(a))`

Mult.in	`4`
Mult.out	`8`

Div.fo	`fun int Div(int a) = a / 2`
	
	`fun int main() =`
	`  let a = read(int) in`
	`  write(Div(a))`

Div.in	`128`
Div.out	`64`

After creating the test files, we swiftly move on to the lexer

Lexer.lex
`| '*'  { Parser.TIMES  (getPos lexbuf) }` `| '/'  { Parser.DIVIDE (getPos lexbuf) }`

In the parser, we add TIMES and DIVIDE to the already created "PLUS MINUS DEQ EQ LTH" token. They're given a proper precedence order as well as added to the expression list.

File name	Code
Parser.grm	`%token <(int*int)> PLUS MINUS DEQ EQ LTH DIVIDE TIMES`
Parser.grm	`%left DIVIDE` `%left TIMES`

Parser.grm	`Exp :`
	`  | Exp DIVIDE Exp { Divide ($1, $3, $2) }`
	`  | Exp TIMES Exp  { Times ($1, $3, $2) }`

Moving on to the type checker, we make sure that we can properly check the used types, when doing multiplication and division

TypeChecker.sml

```
and checkExp ftab vtab (exp : In.Exp)
  = case exp of
     .
     .
     .
     | In.Times (e1, e2, pos)
       => let val (_, e1_dec, e2_dec) =
              checkBinOp ftab vtab (pos, Int, e1, e2)
          in (Int,
              Out.Times (e1_dec, e2_dec, pos))
          end

     | In.Divide (e1, e2, pos)
       => let val (_, e1_dec, e2_dec) =
                  checkBinOp ftab vtab (pos, Int, e1, e2)
          in (Int,
              Out.Divide (e1_dec, e2_dec, pos))
          end
```

We go ahead and implement them in our interpreter

Interpreter.sml

```
| evalExp ( Times(e1, e2, pos), vtab, ftab ) =
      let val res1   = evalExp(e1, vtab, ftab)
          val res2   = evalExp(e2, vtab, ftab)
      in  case (res1, res2) of
              (IntVal n1, IntVal n2) => IntVal (n1*n2)
            | _ => invalidOperands
                   "Multiplication on non-integral args: "
                   [(Int, Int)] res1 res2 pos
      end

| evalExp ( Divide(e1, e2, pos), vtab, ftab ) =
      let val res1   = evalExp(e1, vtab, ftab)
          val res2   = evalExp(e2, vtab, ftab)
      in  case (res1, res2) of
              (IntVal n1, IntVal n2) => IntVal (n1 div n2)
            | _ => invalidOperands
                   "Division on non-integral args: "
                   [(Int, Int)] res1 res2 pos
      end
```

Finally they're added to the MIPS code generator

CodeGen.sml

```
fun compileExp e vtable place =
    case e of
    .
    .
    .
    | Divide (e1, e2, pos) =>
        let val t1 = newName "divide_L"
            val t2 = newName "divide_R"
            val code1 = compileExp e1 vtable t1
            val code2 = compileExp e2 vtable t2
        in code1 @ code2 @ [Mips.DIV (place,t1,t2)]
        end
    | Times (e1, e2, pos) =>
        let val t1 = newName "times_L"
            val t2 = newName "times_R"
            val code1 = compileExp e1 vtable t1
            val code2 = compileExp e2 vtable t2
        in code1 @ code2 @ [Mips.MUL (place,t1,t2)]
        end
```

2.3 AND and OR

Moving on to the implementation of the boolean literals, AND and OR, we again start out with creating test files.

File name	Code

and.fo	`fun bool FAnd(bool a, bool b) = a && b`
	
	`fun bool main() =`
	`  let a = read(bool) in`
	`  let b = read(bool) in`
	`  let c = write(FAnd(a,a)) in`
	`  let d = write(FAnd(b,b)) in`
	`  let e = write(FAnd(a,b)) in`
	`  write(FAnd(b,a))`

and.in	`1` `0`
and.out	`truefalsefalsefalse`

or.fo	`fun bool orTest(bool a, bool b) = a || b`
	
	`fun bool main() =`
	`  let a = read(bool) in`
	`  let b = read(bool) in`
	`  let c = write(orTest(b,b)) in`
	`  let d = write(orTest(a,a)) in`
	`  let e = write(orTest(a,b)) in`
	`  write(orTest(b,a))`

or.in	`1` `0`
or.out	`falsetruetruetrue`

After creating the test files, we swiftly move on to the lexer

Lexer.lex
`| "&&"  { Parser.AND  (getPos lexbuf) }` `| "||"  { Parser.OR  (getPos lexbuf) }`

In the parser, we add AND and OR to the "TRUE FALSE" token as well as added to the expression list.

File name	Code
Parser.grm	`%token <(int*int)> TRUE FALSE AND OR`

Parser.grm	`Exp :`
	`  | Exp AND Exp { And ($1, $3, $2) }`
	`  | Exp OR  Exp { Or  ($1, $3, $2) }`

Moving on to the type checker, we make sure that we can properly check the used types, when using the boolean literals AND and OR

TypeChecker.sml

```
| In.And (e1, e2, pos)
  => let val (t1, e1') = checkExp ftab vtab e1
         val (t2, e2') = checkExp ftab vtab e2
     in case (t1 = t2, t1) of
             (false, _) => raise Error
                           ("And cannot take "^ ppType t1 ^
                            "and "^ppType t2, pos)
           | (true, Array _) =>
             raise Error ("And cannot oporate on arrays", pos)
           | _ => (Bool, Out.And (e1', e2', pos))
    end
| In.Or (e1, e2, pos)
  => let val (t1, e1') = checkExp ftab vtab e1
         val (t2, e2') = checkExp ftab vtab e2
     in case (t1 = t2, t1) of
             (false, _) => raise Error
                           ("Or cannot take "^ ppType t1 ^
                            "and "^ppType t2, pos)
           | (true, Array _) =>
             raise Error ("Or cannot operate on arrays", pos)
           | _ => (Bool, Out.Or (e1', e2', pos))
     end
```

We go ahead and implement them in our interpreter

Interpreter.sml

```
| evalExp (And (e1, e2, pos), vtab, ftab) =
      let val res1 = evalExp(e1, vtab, ftab)
          val res2 = evalExp(e2, vtab, ftab)
      in  case (res1, res2) of
            (BoolVal n1, BoolVal n2) => BoolVal (n1 andalso n2)
          | _ => invalidOperands "OR on non-boolean args: "
                 [(Int, Int)] res1 res2 pos
      end

| evalExp (Or (e1, e2, pos), vtab, ftab) =
      let val res1 = evalExp(e1, vtab, ftab)
          val res2 = evalExp(e2, vtab, ftab)
      in  case (res1, res2) of
            (BoolVal n1, BoolVal n2) => BoolVal (n1 orelse n2)
          | _ => invalidOperands "AND on non-boolean args: "
                 [(Int, Int)] res1 res2 pos
      end
```

Finally we add thr short-circuit AND and OR to the MIPS code generator

CodeGen.sml

```
| And (e1, e2, pos) =>
  let val t1 = newName "and_L"
      val t2 = newName "and_R"
      val code1 = compileExp e1 vtable t1
      val code2 = compileExp e2 vtable t2
      val finish = newName "finish"
  in  code1 @
      [ Mips.LI (place,"0")
      , Mips.BEQ (t1, "0", finish) ] @
      code2 @
      [ Mips.BEQ (t2, "0", finish)
      , Mips.LI (place, "1")
      , Mips.LABEL finish ]
  end

| Or (e1, e2, pos) =>
  let val t1 = newName "or_L"
      val t2 = newName "or_R"
      val code1 = compileExp e1 vtable t1
      val code2 = compileExp e2 vtable t2
      val finish = newName "finish"
  in  code1 @
      [ Mips.LI (place,"1")
      , Mips.BNE (t1, "0", finish) ] @
      code2 @
      [ Mips.BNE (t2, "0", finish)
      , Mips.LI (place, "0")
      , Mips.LABEL finish ]
  end
```

2.4 Not and Negation

Lastly for this task, we will implement Not and Negation. We will, once again, start out with creating test files.

File name	Code

not.fo	`fun bool notFun(bool t) = not t`
	
	`fun bool main() =`
	`  let a = read(bool) in`
	`  let b = read(bool) in`
	`  let c = write(notFun(a)) in`
	`  write(notFun(b))`

not.in	`1` `0`
not.out	`falsetrue`

negate.fo	`fun bool write_nl(bool b) =`
	`  let res = write(b) in`
	`  let tmp = write("\n") in`
	`  res`
	
	`fun bool main() =`
	`  let x0 = write_nl(3 / 2 == 1) in`
	`  let x1 = write_nl(~3 / 2 == ~2) in`
	`  let x2 = write_nl(3 /~2 == ~2) in`
	`  let x3 = write_nl(~3 /~2 == 1) in`
	`  write_nl(x0 && x1 && x2 && x3)`

negate.in	

negate.out	`true`
	`false`
	`false`
	`true`
	`false`

After creating the test files, we move on to the lexer

File name	Code
Lexer.lex	`| "not" => Parser.NOT pos`
Lexer.lex	`| '~' { Parser.NEGATE (getPos lexbuf) }`

In the parser, we add Not and Negate to the "TRUE FALSE AND OR" token.

We also include them in `%nonassoc ifprec letprec` making sure, they are nonassiciative. And lastly, we add the expressions to the expressionlist.

File name	Code
Parser.grm	`%token <(int*int)> TRUE FALSE AND OR NOT NEGATE`
Parser.grm	`%nonassoc NOT` `%nonassoc NEGATE`

Parser.grm	`Exp :`
	`  | NOT Exp  { NOT  ($2, $1) }`
	`  | NEGATE Exp { NEGATE ($2, $1) }`

Moving on to the type checker, we make sure that we can properly check the used types, when using Not and Negate.

TypeChecker.sml

```
| In.Not (e, pos)
  => let val (t1, e') = checkExp ftab vtab e
     in  case (t1) of
           (Bool) => (Bool, Out.Not (e', pos))
         | _ => raise Error ("Invalid Argument type", pos)
     end

| In.Negate (e, pos)
  => let val (t1, e') = checkExp ftab vtab e
     in  case (t1) of
           (Int) => (Int, Out.Negate (e', pos))
         | _ => raise Error ("Invalid Argument type", pos)
     end
```

We go ahead and implement them in our interpreter.

Interpreter.sml

```
| evalExp ( Not(e, pos), vtab, ftab ) =
      let val res = evalExp(e, vtab, ftab)
      in  case (res) of
            (BoolVal n) => BoolVal (if n then false else true)
          | _ => invalidOperand "NOT on non-boolean args: "
                Bool res pos
      end

| evalExp ( Negate(e, pos), vtab, ftab ) =
      let val res = evalExp(e, vtab, ftab)
      in  case (res) of
            (IntVal n) => IntVal (n * (~1))
          | _ => invalidOperand "Negate on non-integer args: "
                Int res pos
      end
```

Finally we add the expressions to the MIPS code generator

CodeGen.sml

```
| Not (e', pos) =>
      let val t1 = newName "bool"
          val code = compileExp e' vtable t1
      in  code @
          [Mips.XORI (place, t1, "1")]
      end


| Negate (e', pos) =>
         let val t1 = newName "negate"
             val code = compileExp e' vtable t1
         in  code @ [Mips.SUB (place, "0", t1)]
         end
```

3 Results

4 Conclusion