A Compiler for the FASTO Language

Allan Nielsen, Christian Nielsen, Troels Kamp Leskes

December 22, 2014

CONTENTS

1 MULTIPLICATION, DIVISION, BOOLEAN OPERATORS AND LITERALS

Implementing multiplication and division was a simple matter, when having the already implemented code for addition and subtraction to look at. They served as a great way of getting to know the fasto compiler, and how things operate.

Negation is implemeted using the Mips.SUB instruction, where we pass the original argument to the operator, and subtracts this from zero.

So the instruction looks like:

```
Mips.SUB(place, "0", t1),
```

where t1 register that holds the argument x for $\sim x$, and place is the register in which we place the result.

Not was more complicated than the previous ones, given that this requires more than one instruction to execute. However, the pattern learned here, proved to be useful for implementing and and or as well.

```
[ Mips.LI (place,"0")
, Mips.BNE (b,"0",falseLabel)
, Mips.LI (place,"1")
, Mips.LABEL falseLabel ]
```

Place is the register in which we want to store our result. We start by putting 0 into place, we then check if our argument b, actually is 0. Since Mips.BNE branches if its arguments are not equal, we will jump to falseLabel if and only if our argument b is 1, thus ending with a 0 in the place register, given we never execute Mips.LI(place, "1").

HAR INGEN IDE OM BOOL LITS

2 FILTER AND SCAN

3 λ-EXPRESSIONS IN SOACS

4 COPY PROPAGATION AND CONSTANT FOLDING

We have implemented *copy propagation and constant foldning* in our compiler, though it does not handle shadowing. Furthermore have all the features from **REF TIL TASK 1**.

Below is the test-case for Times explained. This will indicate that both the cases work and it is able to actually optimize by propagating and fold correspondingly.

When we want to fold expressions, we can do so, by predicting what the result is going to be of a given expression and return this instead.

We thought through for each expression, what base-cases there could be and what we then should return.

Below is shown our code for the *Times*-expression.

```
| Times  (e1, e2, pos) =>
  let val e1' = copyConstPropFoldExp vtable e1
      val e2' = copyConstPropFoldExp vtable e2
  in case (e1', e2') of
        (Constant (IntVal x, _), Constant (IntVal y, _)) =>
        Constant (IntVal (x*y), pos)
      | (Constant (IntVal 0, _), _) =>
        e1'
      | (_, Constant (IntVal 0, _)) =>
        e2'
      | (Constant (IntVal 1, _), _) =>
        e2'
      | (_, Constant (IntVal 1, _)) =>
        e1'
      | _ =>
        Times (e1', e2', pos)
  end
```

We start by optimizing the two subexpressions. Then we check if any of the two expressions evaluates to 0 or 1. If one of them is 0, we return 0. Also, if one of them is 1 then it will be the other expression that is returned, no matter what it states. If none of the above applies, then we want to compute the expression.

Below is the code for testing *Times*.

```
fun int main() =
  let a = 5 in
  let b = a in
  let c = b in
   write(b + c * 0)
```

If our propagation is correct, all the variables should evaluate to 5, due to *a* having that constant assigned.

In the *write*-statement we have some expressions which also can be optimized. Since we multiply by 0, that whole expression will evaluate to 0. Then we plus *b* with 0 and end up with *b* as the result, and we end up writing *b*.

In the end, our program will look like this when optimized using copy propagation and constant folding.

```
fun int main() =
    let a = 5 in
    let b = 5 in
    let c = 5 in
    write(5)
```

Lexer.lex

```
40      | "fn"            => Parser.FN pos
42      | "not"           => Parser.NOT pos

66      | "true"|"false"      { case Bool.fromString (getLexeme lexbuf) of
67                                 NONE   => lexerError lexbuf "Bad bool"
68                               | SOME b => Parser.BOOLEAN (b, getPos lexbuf) }

87      | '*'                 { Parser.TIMES (getPos lexbuf) }
88      | '/'                 { Parser.DIV (getPos lexbuf) }
89      | '~'                 { Parser.NEG (getPos lexbuf) }
91      | "=>"                { Parser.ARROW (getPos lexbuf) }
94      | "||"                { Parser.OR (getPos lexbuf) }
95      | "&&"                { Parser.AND (getPos lexbuf) }
```

Parser.grm

```
10   token <(int*int)> IF THEN ELSE LET IN INT BOOL CHAR EOF
11   token <string*(int*int)> ID STRINGLIT
12   token <int*(int*int)> NUM
13   token <char*(int*int)> CHARLIT
14   token <bool*(int*int)> BOOLEAN
15   token <(int*int)> PLUS MINUS DEQ EQ LTH NEG NOT ARROW
16   token <(int*int)> TIMES DIV AND NOT OR LPAR RPAR LBRACKET RBRACKET LCURLY RCURLY
17   token <(int*int)> COMMA
18   token <(int*int)> FUN FN IOTA REPLICATE MAP REDUCE FILTER SCAN READ WRITE
19   token <(int*int)> OP

21   nonassoc ifprec letprec
22   left DEQ LTH
23   left AND
24   left OR
25   nonassoc NOT
26   left PLUS MINUS
27   left TIMES DIV
28   nonassoc NEG
```

```
255        | In.Times (e1, e2, pos)
256        => let val (_, e1_dec, e2_dec) = checkBinOp ftab vtab (pos, Int, e1, e2)
257             in (Int,
258               Out.Times(e1_dec, e2_dec, pos))
259             end
260
261        | In.Divide (e1, e2, pos)
262        => let val (_, e1_dec, e2_dec) = checkBinOp ftab vtab (pos, Int, e1, e2)
263             in (Int,
264               Out.Divide(e1_dec, e2_dec, pos))
265             end
266
267        | In.Negate (e1, pos)
268        => let val (t1, e1') = checkExp ftab vtab e1
269             in (Int, Out.Negate(e1', pos))
270             end
271
272        | In.Not (e1, pos)
273        => let val (t1, e1') = checkExp ftab vtab e1
274             in (Bool, Out.Not(e1', pos))
275             end
276
277        | In.Or (e1, e2, pos)
278        => let val (t1, e1') = checkExp ftab vtab e1
279               val (t2, e2') = checkExp ftab vtab e2
280             in
281                 if(t1 = Bool andalso t2 = Bool) then
282                   (Bool, Out.Or(e1', e2', pos))
283                 else
284                   raise Error("Type Error: Non-boolean arguments given to ||", pos)
285             end
286
287        | In.And (e1, e2, pos)
288        => let val (t1, e1') = checkExp ftab vtab e1
289               val (t2, e2') = checkExp ftab vtab e2
290             in
291                 if(t1 = Bool andalso t2 = Bool) then
292                   (Bool, Out.And(e1', e2', pos))
293                 else
294                   raise Error("Type Error: Non-boolean arguments given to &&", pos)
295             end
296
298        | In.Scan (f, n_exp, arr_exp, _, pos) 256
299        => let val (n_type, n_dec) = checkExp ftab vtab n_exp
300               val (arr_type, arr_dec) = checkExp ftab vtab arr_exp
301               val elem_type =
302                 case arr_type of
303                   Array t => t
304                   | other => raise Error ("Scan: argument is not an array", pos)
305               val (f', f_arg_type) =
306                 case checkFunArg (f, vtab, ftab, pos) of
307                   (f', res, [a1, a2]) =>
```

```
308                    if a1 = a2 andalso a2 = res
309                      then (f', res)
310                      else raise Error
311                            ("Scan: incompatible function type of "
312                              ^ In.ppFunArg 0 f ^": " ^ showFunType ([a1, a2], res),
   pos)
313                    | (_, res, args) =>
314                      raise Error ("Scan: incompatible function type of "
315                                  ^ In.ppFunArg 0 f ^ ": " ^ showFunType (args, res),
   pos)
316              fun err (s, t) =
317                  Error ("Scan: unexpected " ^ s ^ " type " ^ ppType t ^
318                          ", expected " ^ ppType f_arg_type, pos)
319          in if elem_type = f_arg_type
320            then if elem_type = n_type
321                then (Array elem_type,
322                      Out.Scan (f', n_dec, arr_dec, elem_type, pos))
323                else raise (err ("neutral element", n_type))
324            else raise err ("array element", elem_type)
325          end


329      | In.Filter (f, arr_exp, _, pos)   260
330        => let val (arr_type, arr_exp_dec) = checkExp ftab vtab arr_exp
331
332              val elem_type =
333                case arr_type of
334                  Array t => t
335                | other => raise Error ("Filter: argument is not an array", pos)
336
337              val (f', f_res_type, f_arg_type) =
338                case checkFunArg (f, vtab, ftab, pos) of
339                    (f', Bool, [a1]) => (f', Bool, a1)
340                  | (_,  res, args) =>
341                    raise Error ("Filter: incompatible function type of "
342                              ^ In.ppFunArg 0 f ^ ":" ^ showFunType (args, res),
   pos)
343
344              in (arr_type,
345                    Out.Filter (f', arr_exp_dec, elem_type, pos))
346              end

362      | checkFunArg (In.Lambda (ret_type, params, body, funpos) 276
363                  , vtab, ftab, pos) =
364          let val (Out.FunDec ( name, _, params, body', pos)) = checkFunWithVtable (In.
   FunDec ("Lambda", ret_type, params, body, pos), vtab, ftab, funpos)
365          val arg_types = map (fn (Param (_, ty)) => ty) params
366          in
367          (Out.Lambda(ret_type, params, body', funpos), ret_type, arg_types)
368          end
```

CodeGen.sml

```
614        | Constant (BoolVal b, pos) =>
615            if(b) then
616              [Mips.LI(place, "1")]
617            else
618              [Mips.LI(place, "0")]


624        | Times (e1, e2, pos) =>
625            let val t1 = newName "times_L"
626                val t2 = newName "times_R"
627                val code1 = compileExp e1 vtable t1
628                val code2 = compileExp e2 vtable t2
629            in code1 @ code2 @ [Mips.MUL (place,t1,t2)]
630            end
631        | Divide (e1, e2, pos) =>
632            let val t1 = newName "div_L"
633                val t2 = newName "div_R"
634                val code1 = compileExp e1 vtable t1
635                val code2 = compileExp e2 vtable t2
636            in code1 @ code2 @ [Mips.DIV (place,t1,t2)]
637            end
638
639        | Negate (e1, pos) =>
640            let val t1 = newName "negateVal"
641                val code1 = compileExp e1 vtable t1
642            in code1 @ [ Mips.SUB(place, "0", t1) ]
643            end

645        | Not (b_exp, pos) =>
646            let val b = "boolean"
647                val code1 = compileExp b_exp vtable b
648                val falseLabel = newName "false"
649            in code1 @
650                [ Mips.LI (place,"0")
651                , Mips.BNE (b,"0",falseLabel)
652                , Mips.LI (place,"1")
653                , Mips.LABEL falseLabel ]
654            end
655
656        | Or (e1, e2, pos) =>
657            let val trueLabel = newName "trueLabel"
658                val falseLabel = newName "falseLabel"
659                val endLabel = newName "endLabel"
660                val code1 = compileCond e1 vtable trueLabel falseLabel
661                val code2 = compileCond e2 vtable trueLabel endLabel
662            in  [Mips.LI(place, "0")] @
663                code1 @
664                [Mips.LABEL falseLabel] @
665                code2 @
666                [Mips.LABEL trueLabel, Mips.LI(place, "1"), Mips.LABEL endLabel]
667            end
668
669        | And (e1, e2, pos) =>
```

```
670            let val trueLabel = newName "trueLabel"
671                val falseLabel = newName "falseLabel"
672                val endLabel = newName "endLabel"
673                val code1 = compileCond e1 vtable trueLabel falseLabel
674                val code2 = compileCond e2 vtable endLabel falseLabel
675            in  [Mips.ADD(place, "0", "1")] @
676                  code1 @
677                 [Mips.LABEL trueLabel] @
678                  code2 @
679                 [Mips.LABEL falseLabel, Mips.ADD(place, "0", "0"), Mips.LABEL endLabel]
680            end


688        (* Scan(f, e, [a1, a2, ..., an]) = [e, f(e, a1), f(f(e, a1), a2), ...] *)
689      | Scan (farg, acc_exp, arr_exp, elem_type, pos) =>
690            let val in_size_reg = newName "in_size_reg" (* size of input array *)
691                val out_size_reg = newName "out_size_reg" (* size of ouput array *)
692                val acc_reg = newName "acc_reg" (* last computed value for output *)
693                val i_reg = newName "i_reg" (* Iterator register *)
694                val addr_reg = newName "addr_reg" (* address of element in output array
     *)
695                val arr_reg = newName "arr_reg"
696                val elem_reg = newName "elem_reg"
697
698                val arr_code = compileExp arr_exp vtable arr_reg
699                val acc_code = compileExp acc_exp vtable acc_reg
700
701                val get_size = [ Mips.LW (in_size_reg, arr_reg, "0") (* Loads array-size
      into in_size_reg *)
702                               , Mips.ADDI(out_size_reg, in_size_reg, "1") (* Puts size
     into out_size_reg *)
703                               ]
704
705                (* Initiate registers.
706                     Put address of place into addr_reg, so we return proper addresses.
707                     Increment in_size_reg by 1 to determine out_size_reg, since output
     array is
708                     1 element longer. *)
709                val init_regs = [ Mips.ADDI (addr_reg, place, "4") (* point to the next
     word *)
710                                , Mips.SW (acc_reg, addr_reg, "0")
711                                , Mips.ADDI (addr_reg, addr_reg, "4") (* point to next
     word*)
712                                , Mips.MOVE(i_reg, "0") (* initialize iterator*)
713                                , Mips.ADDI (arr_reg, arr_reg, "4") (* Look at first
     element in input array*)
714                                ]
715
716
717                val loop_beg = newName "loop_beg"
718                val loop_end = newName "loop_end"
719                val tmp_reg = newName "tmp_reg"
720
721                (* while i_reg < in_size_reg*)
722                val loop_head =
```

```
723                    [ Mips.LABEL(loop_beg)
724                    , Mips.SUB(tmp_reg, i_reg, out_size_reg) (* make statement*)
725                    , Mips.BGEZ(tmp_reg, loop_end) ] (* if statement is equal to zero,
    jump to end *)
726                (* loads next value in input array into tmp_reg *)
727                val load_value =
728                    case getElemSize elem_type of
729                        One => [ Mips.LB   (elem_reg, arr_reg, "0")
730                               , Mips.ADDI (arr_reg, arr_reg, "1") ]
731                      | Four => [ Mips.LW   (elem_reg, arr_reg, "0")
732                               , Mips.ADDI (arr_reg, arr_reg, "4") ]

734                val apply_code =
735                    applyFunArg(farg, [acc_reg, elem_reg], vtable, acc_reg, pos)

737                (* save the current accumulated value to a register for later
    computations,
738                   and to memory for later output *)
739                val save_value =
740                    case getElemSize elem_type of
741                        One => [ Mips.SB (acc_reg, addr_reg, "0")
742                               , Mips.ADDI (addr_reg, addr_reg, "1")]
743                      | Four => [ Mips.SW (acc_reg, addr_reg, "0")
744                               , Mips.ADDI (addr_reg, addr_reg, "4")]




748                (* increments i_reg *)
749                val loop_foot =
750                    [ Mips.ADDI(i_reg, i_reg, "1")
751                    , Mips.J loop_beg
752                    , Mips.LABEL loop_end ]

754          in [Mips.LABEL "Det_her_er_starten"]
755             @ arr_code
756             @[Mips.LABEL "array_kode"]
757             @ acc_code
758             @[Mips.LABEL "akku_kode"]
759             @ get_size
760             @[Mips.LABEL "Stoerrelsen_på_dyret"]
761             @ dynalloc (out_size_reg, place, elem_type)
762             @[Mips.LABEL "init_regz"]
763             @ init_regs
764             @[Mips.LABEL "starten_af_loopet"]
765             @ loop_head
766             @[Mips.LABEL "midten_af_loopet"]
767             @ load_value
768             @[Mips.LABEL "mere_midte"]
769             @ apply_code
770             @[Mips.LABEL "gem_den_akku"]
771             @ save_value
772             @[Mips.LABEL "foden_af_loopet"]
773             @ loop_foot
774             @ [Mips.LABEL "Det_her_er_slutningen_SCAN"]
```

```
775            end


783        (* filter(f(), acc, [a1,a2]) = [f(acc, a1), f(acc, a2)] *)
784        | Filter (farg, arr_exp, elem_type, pos) =>
785           let val size_reg = newName "size_reg" (* size of input/output array *)
786               val arr_reg  = newName "arr_reg" (* address of input array *)
787               val elem_reg = newName "elem_reg" (* address of single element *)
788               val res_reg = newName "res_reg"
789               val val_reg = newName "val_reg"
790               val arr_code = compileExp arr_exp vtable arr_reg
791
792               val get_size = [ Mips.LW (size_reg, arr_reg, "0") ] (* *)
793
794               val addr_reg = newName "addr_reg" (* address of element in new array *)
795               val i_reg = newName "i_reg"
796               val j_reg = newName "j_reg"
797               val init_regs = [ Mips.ADDI (addr_reg, place, "4") (*point to the next
    word, so we don't overwrite the size*)
798                               , Mips.MOVE (i_reg, "0")
799                               , Mips.LI (j_reg, "0")
800                               , Mips.ADDI (elem_reg, arr_reg, "4") ]
801
802               val loop_beg = newName "loop_beg"
803               val loop_end = newName "loop_end"
804               val tmp_reg = newName "tmp_reg"
805               val loop_header = [ Mips.LABEL (loop_beg)
806                                 , Mips.SUB (tmp_reg, i_reg, size_reg)
807                                 , Mips.BGEZ (tmp_reg, loop_end)
808                                 , Mips.ADDI (i_reg, i_reg, "1") ]
809
810               (* map is 'arr[i] = f(old_arr[i])'. *)
811               val loop_load =
812                   case getElemSize elem_type of
813                       One  => Mips.LB(val_reg, elem_reg, "0")
814                                :: applyFunArg(farg, [val_reg], vtable, res_reg, pos)
815                               @ [ Mips.ADDI(elem_reg, elem_reg, "1") ]
816                     | Four => Mips.LW(val_reg, elem_reg, "0")
817                                :: applyFunArg(farg, [val_reg], vtable, res_reg, pos)
818                               @ [ Mips.ADDI(elem_reg, elem_reg, "4") ]
819
820               val check_fun = [ Mips.BEQ (res_reg, "0", loop_beg)
821                               , Mips.ADDI (j_reg, j_reg, "1") ]
822
823               val loop_save =
824                   case getElemSize elem_type of
825                       One => [ Mips.SB (val_reg, addr_reg, "0")]
826                     | Four => [ Mips.SW (val_reg, addr_reg, "0") ]
827
828               val loop_footer =
829                   [ Mips.ADDI (addr_reg, addr_reg,
830                                makeConst (elemSizeToInt (getElemSize elem_type)))
831                   , Mips.J loop_beg
832                   , Mips.LABEL loop_end
833                   , Mips.SW (j_reg, place, "0")
```

```
834                    ]
835          in [Mips.LABEL "array_kode"]
836             @ arr_code (* make arr_reg point to input array *)
837             @[Mips.LABEL "measuring"]
838             @ get_size (* gets the size of the input array *)
839             @[Mips.LABEL "allocating_da_mem"]
840             @ dynalloc (size_reg, place, elem_type) (* return place, which is an
     address where the ouput array is going to be *)
841             @[Mips.LABEL "initializing"]
842             @ init_regs
843             @[Mips.LABEL "loopin"]
844             @ loop_header
845             @ loop_load
846             @ check_fun
847             @ [Mips.LABEL "saving"]
848             @ loop_save
849             @ loop_footer
850          end


868       | applyFunArg (Lambda(ret_type, params, body', funpos), args, vtable, place,
     pos) : Mips.Prog =
868          let
869            fun bindVars ([], [], vtable) = vtable
870              | bindVars([], args, vtable) = raise Error("stop det pjat", pos)
871              | bindVars(params, [], vtable) = raise Error("stop det pjat stadigvæk",
     pos)
872              | bindVars(Param (name, paramtype)::params, arg::args, vtable) = SymTab.
     bind name arg (bindVars(params, args, vtable))
873          val newVtable = bindVars(params, args, vtable)
874          val code1 = compileExp body' newVtable place
875          in
876            code1
877          end
```

12

Interpreter.sml

```
412    | evalExp ( Times(e1, e2, pos), vtab, ftab ) =
413          let val res1   = evalExp(e1, vtab, ftab)
414              val res2   = evalExp(e2, vtab, ftab)
415          in  evalBinopNum(op *, res1, res2, pos)
416          end
417
418    | evalExp ( Divide(e1, e2, pos), vtab, ftab ) =
419          let val res1   = evalExp(e1, vtab, ftab)
420              val res2   = evalExp(e2, vtab, ftab)
421          in  evalBinopNum(op div, res1, res2, pos)
422          end
423
424    | evalExp ( Negate(e1, pos), vtab, ftab ) =
425          let val res1 = evalExp(e1, vtab, ftab)
426              val res2 = evalExp(Constant(IntVal 0, pos), vtab, ftab)
427          in  evalBinopNum(op +, res1, res2, pos)
428          end
429
430    | evalExp ( Not(e1, pos), vtab, ftab ) =
431          let val res1 = evalExp(e1, vtab, ftab)
432          in  case res1 of
433                BoolVal true => BoolVal false
434              | BoolVal false => BoolVal true
435              | _ => raise Error("Input is not of type bool", pos)
436          end
437
438    | evalExp ( And(e1, e2, pos), vtab, ftab ) =
439          let val res1 = evalExp(e1, vtab, ftab)
440              val res2 = evalExp(e2, vtab, ftab)
441          in  case res1 of
442                BoolVal false => res1
443              | BoolVal true => (case res2 of
444                                   BoolVal _ => res2
445                                 | _ => raise Error("Second argument is not of type
    bool", pos))
446              | _ => raise Error("First argument is not of type bool", pos)
447
448          end
449
450    | evalExp ( Or(e1, e2, pos), vtab, ftab ) =
451          let val res1 = evalExp(e1, vtab, ftab)
452              val res2 = evalExp(e2, vtab, ftab)
453          in  case res1 of
454                BoolVal true => res1
455              | BoolVal false => (case res2 of
456                                   BoolVal _ => res2
457                                 | _ => raise Error("Second argument is not of type
    bool", pos))
458              | _ => raise Error("First argument is not of type bool", pos)
459
460          end
```

```
517    | evalFunArg (Lambda(ret_type, params, exp, pos), vtab, ftab, callpos) =
518       let
519          val fundec = FunDec ("Lambda", ret_type, params, exp, pos)
520       in
521        (fn aargs => callFunWithVtable(fundec, aargs, vtab, ftab, callpos), ret_type)
522       end
```

CopyConstPropFold.sml

```
16    fun copyConstPropFoldExp vtable e =
17      case e of
18          Constant x => Constant x
19        | StringLit x => StringLit x
20        | ArrayLit (es, t, pos) =>
21          ArrayLit (map (copyConstPropFoldExp vtable) es, t, pos)
22        | Var (name, pos) =>
23          (* TODO TASK 4: This case currently does nothing.
24
25           You must perform a lookup in the symbol table and if you find
26           a Propagatee, return either a new Var or Constant node. *)
27          let
28            val value = SymTab.lookup name vtable
29          in
30            case value of
31              SOME (VarProp v)   => copyConstPropFoldExp vtable (Var(v, pos))
32            | SOME (ConstProp v) => copyConstPropFoldExp vtable (Constant(v, pos))
33            | NONE               => Var(name, pos)
34          end

102     let val e' = copyConstPropFoldExp vtable e
103       in
104         let
105           val vtable' =
106             case e' of
107               (Var (vname, p))     => (SymTab.bind name (VarProp(vname)) vtable)
108             | (Constant (vval, p)) => (SymTab.bind name (ConstProp(vval)) vtable)
109             | _ => vtable
110         in
111           Let (Dec (name, e', decpos), copyConstPropFoldExp vtable' body, pos)
112         end
113       end
```