Oversættere - Gruppeprojekt

Nicolai Nebel Jørgensen
Sofie Aleksandra Borup Harning
Rasmus Friis

December 22, 2014

Task 1

Lexer

First of all, we extended Lexer.lex with the required symbols. Every token carries the position along with it. Boolean values get their value along as well.

```
fun keyword s =
   ...
 | "true"         => Parser.BOOLLIT (true, pos)
 | "false"        => Parser.BOOLLIT (false, pos)
 | "not"          => Parser.NOT pos

rule token = parse
   ...
 | `~`                { Parser.NEGATE (getPos lexbuf) }

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

 | "=>"               { Parser.RARROW (getPos lexbuf) }
 | "&&"               { Parser.AND    (getPos lexbuf) }
 | "||"               { Parser.OR     (getPos lexbuf) }
```

Parser

We added two things to the mosmlyac parser specification in Parser.grm. First, we defined associativity and precedence for the new operators. Logical operators have lower precedence than comparison and integer operators.

```
\%nonassoc ifprec letprec
\%left OR
\%left AND
\%right NOT
\%left DEQ LTH
\%left PLUS MINUS
\%left TIMES DIVIDE
\%right NEGATE
```

We then added rules for parsing the different operators. This was basically copying the grammar from the assignment. Ambiguities were already solved through the associativity and precedence declarations.

Boolean values are parsed from the BOOLLIT token.

```
 | BOOLLIT            { Constant (BoolVal (#1 $1), #2 $1)  }
```

```
| Exp TIMES Exp       { Times($1, $3, $2)  }
| Exp DIVIDE Exp      { Divide($1, $3, $2) }
| Exp AND Exp         { And($1, $3, $2)    }
| Exp OR Exp          { Or($1, $3, $2)     }
| Exp DEQ   Exp       { Equal($1, $3, $2)  }
| Exp LTH   Exp       { Less ($1, $3, $2)  }
| NOT Exp             { Not($2, $1)        }
| NEGATE Exp          { Negate($2, $1)     }
```

Interpreter

Next came the interpreter changes. Times was easily implemented using almost the same code as Plus, simply changing "op +" to "op *" in the call to evalBinOpNum. As for Divide we used "int.quot" instead of "op /", as specified in the assignment.

Here is the code for Times, Divide is similar.

```
| evalExp ( Times(e1, e2, pos), vtab, ftab ) =
      let val res1   = evalExp(e1, vtab, ftab)
          val res2   = evalExp(e2, vtab, ftab)
      in  evalBinopNum(op *, res1, res2, pos)
      end
```

And and Or is done short-circuiting by evaluating the first expression and then checking if we are ready to short-circuit. If we aren't, the evaluation continues with the second expression.

This implementation doesn't use the build-in short-circuiting andalso and orelse in SML. It seems like this would be a more elegant solution, but we couldn't figure out a good way to do it.

```
| evalExp ( And(e1, e2, pos), vtab, ftab) =
      let val cond = evalExp(e1, vtab, ftab)
      in case cond of
              BoolVal false => BoolVal false
            | BoolVal true  => evalExp(e2, vtab, ftab)
            | _             => raise Error("Argument to &&
                                was not a boolean", pos)
      end
```

Not and negate are simply implemented using the SML not and "∼".

```
| evalExp ( Not(e1, pos), vtab, ftab) =
      (case evalExp(e1, vtab, ftab) of
          BoolVal b => BoolVal (not b)
        | _         => raise Error("Tried to not a
```

```
                                                    non-boolean", pos)
          )
```

Typechecker

For the typechecker, the binary operators were again much the same as the implementation for 'Plus'. In this case, it included the logical operators as well. Times and And look very much like Plus:

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

The implementation of 'times' is barely different, while And has the Int types replaced with Bool.

Not and Negate only needs to check one parameter, so the implementation just calls CheckExp. Negate needs an Int and Not needs a Bool.

```
| In.Not (e1, pos)
   => let val (t, e_dec) = checkExp ftab vtab e1
      in
        if t = Bool
        then
          (t, Out.Not (e_dec, pos))
        else raise Error ("Not: Argument is not a bool", pos)
      end
```

Code generation

The next step is code generation. The first thing to do was to add support for boolean values. A reasonable representation seemed to be 0 for false and 1 for true. The following 'Constant' patters were added in compileExp:

```
    | Constant (BoolVal false, pos) =>
              [ Mips.LI (place, makeConst 0) ]
    | Constant (BoolVal true, pos) =>
              [Mips.LI (place, makeConst 1) ]
```

The code for the different binary operators is pretty similar. We compile the individual expressions and add a few lines of code that operate on the results. Times, for example just uses the Mips instruction mul to compute the result:

```
      | 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
```

Divide looks the same, but uses div instead of mul to compute. Negate only has one expression to compile beforehand, but otherwise is the same. It is implemented by subtracting the result from 0.

Not is a bit more interesting. The result is found by xor-ing the result by 1. If the expression compils to true, it will be 1 xor 1 = 0 and if it is false it will be 0 xor 1 = 1.

```
       | Not (e1, pos) =>
           let
             val t1 = newName "not"
             val code = compileExp e1 vtable t1
           in
             code @ [ Mips.XORI (place, t1, "1") ]
           end
```

The logical operators And and Or needed to be short-circuiting. For And, the code looks like this:

```
       | And (e1, e2, pos) =>
           let
             (...)
           in
             code1
             @ [Mips.BEQ(place, "0", end_loop)]
             @ code2
             @ [Mips.LABEL(end_loop)]
           end
```

First, code1 is evaluated. If it is false, we already have the answer and branch out. Note that the result is already in place, so we don't need to move it. code2 is then evaluated, and it's result is put in place. Whatever it evaluates to will be the result. Or is implemented the same way, but it branches if code1 returns true.

Task 2

The first thing we needed to do, was modify the lexer and parser. To the lexer we simply added two lines in the keyword function to match "scan" and "filter". To the parser we added the following two grammar rules:

```
| SCAN LPAR FunArg COMMA Exp COMMA Exp RPAR
                  { Scan ($3, $5, $7, (), $1) }
| FILTER LPAR FunArg COMMA Exp RPAR
                  { Filter ($3, $5, (), $1) }
```

Scan parses like reduce, since they take the same type of arguments. Filter looks a lot like Map, for the same reason.

To implement Scan in the typechecker, we started with a copy of Reduce, and modified it to the point where it would work for Scan. The difference between the functions is tiny. Scan is, if you only look at types, precisely like Reduce, except it returns an array of things, as seen in the following code, which shows the only significant change between the two.

```
 then (arr_type,
               Out.Scan (f', n_dec, arr_dec, elem_type, pos))
```

Then we implemented Filter into the typechecker and based the structure of it on Map. There are two changes from the case of Map. Firstly, Filter takes a predicate, and therefore the given function must have one argument and return a boolean. Secondly, Filter returns an array of the same type as array given as argument, where Map returns an array with elements of the return type of its passed function.

The first change is where the function argument types are computed. Notice that we no longer care about the return type of the predicate, since we check it directly with the call to checkFunArg.

```
      val (f', _, f_arg_type) =
      case checkFunArg (f, vtab, ftab, pos) of
           (f', Bool, [a1]) => (f', Bool, a1)
         | (_,  res, args)  =>
                raise Error ("Filter: incompatible
                 function type of " ^ In.ppFunArg 0 f ^ ":"
                 ^ showFunType (args, res), pos)
```

The second change happens in the return type.

```
      if elem_type = f_arg_type
        then (Array elem_type,
```

```
                Out.Filter (f', arr_exp_dec, elem_type, pos))
          else raise Error (...)
```

That concludes the typechecking changes for both Filter and Scan.

Filter is, as we've seen before, very similar to Map. That is also the case in code-generation. The implementation, like map, loops through the given array and places results in the newly allocated location.

The problem is, however, that we don't know the final size of the result array. Therefore, we allocate for the worst case, being that every element is copied over. This means we will in most cases have some wasted space.

When choosing which elements to add to the result, we simply run the compiled predicate on the element. If it returns false, we jump to the end of the loop.

The last problem is the size of the result array. In Fasto, the first word contains an integer that stores the size of the array. Since we at the start don't know the size of the result, we must add it last. Therefore, in a separate register, we store the size of the result array, starting at 0. This register is incremented every time an element is added to the result. At the end, the 'place' register points to the beginning of the array. We store the computed size in the address of 'place' and then we are done.

```
| Filter (farg, arr_exp, tp, pos) =>
  let
    (... Name definitions ...)
    val arr_code = compileExp arr_exp vtable arr_reg
    val get_size = [ Mips.LW (size_reg, arr_reg, "0") ]
    val init_regs = (... Initialize size and
                    address for registers ...)
    val loop_header = [ Mips.LABEL (loop_beg)
                      , Mips.SUB (tmp_reg, i_reg, size_reg)
                      , Mips.BGEZ (tmp_reg, loop_end) ]

    val load_code = (...)

    val if_code =
      [ Mips.BEQ (tmp_reg, "0", if_end)
      , Mips.ADDI (res_size_reg, res_size_reg, "1") ]

    val store_code = (..)

    val if_end_code =
      [ Mips.LABEL(if_end)
      , Mips.ADDI (i_reg, i_reg, "1")
      , Mips.J (loop_beg)
      , Mips.LABEL (loop_end) ]
```

```
  in
    arr_code
    @ get_size @ dynalloc (size_reg, place, tp) @ init_regs
    @ loop_header @ load_code @ if_code @ store_code
    @ if_end_code @ [ Mips.SW (res_size_reg, place, "0") ]
  end
```

We implemented scan in a way that was very similar to the existing implementation of reduce, but with some added elements from map. From reduce we took the code that accumulated a value in a register and from map we took the code that saved the results in a new array.

The size of the return array is the size of the given array plus one, for the original accumulator value.

The loop is pretty straightforward. Load the next value from the array, apply the binary function, save the result in the accumulator and save that accumulator in the return array. Move all appropriate pointers forward and repeat. Like map

```
| Scan (binop, acc_exp, arr_exp, tp, pos) =>
    let
      (... Name definitions ...)
      (... Compile expressions, save array registers,
      save target address ...)

      (* loop code *)
      val loop_header =
        [ Mips.LABEL(loop_beg)
        , Mips.SUB(tmp_reg, i_reg, size_reg)
        , Mips.BGEZ(tmp_reg, loop_end) ]

      (* Load arr[i] into tmp_reg *)
      val load_code = (...)

      val apply_code =
            applyFunArg(binop, [acc_reg, tmp_reg],
                        vtable, acc_reg, pos)

      (* Store acc in addr_reg, increment addr_reg *)
      val store_code = (...)

      val loop_footer =
        [ Mips.ADDI (i_reg, i_reg, "1")
        , Mips.J (loop_beg)
        , Mips.LABEL (loop_end)  ]
    in
      arr_code @ header1 @ acc_code
```

```
      @ dynalloc(size_reg, place, tp)
      @ init_regs
      @ store_code (* Save first elem *)
      @ loop_header @ load_code @ apply_code @ store_code
      @ loop_footer
    end
```

Task 3

The first part to implementing Lambda-expressions was to modify the lexer specification. "fn" was added to the 'keyword' helper function, and the pattern "=¿" was added as a pattern.

We modified the parser specification to parse Lambdas. In Fasto, they are only ever used inside the pre-defined higher-order functions, parsed with the FunArg nonterminal. Therefore we add the rule for parsing Lambdas inside that nonterminal.

```
FunArg : ID { FunName (#1 $1) }
       | FN Type LPAR Params RPAR
                    RARROW Exp { Lambda ($2, $4, $7, $1) }
;
```

When figuring out how to implement Lambdas in the interpreter, we looked at the already defined case of evalFunArg. We could see that the return type had to be a function that matched actual arguments into a call to callFunWithVtable. The arguments to that function had to be some FunDec, along with a vtab and ftab.

Our strategy then simply is to use the components of the Lambda to build a dummy FunDec and passing it into callFunWithVtable in a similar way.

```
  | evalFunArg (Lambda (tp, params, body, fpos),
                                    vtab, ftab, callpos) =
        (
         (fn aargs =>
              callFunWithVtable (
                 FunDec("", tp, params, body, fpos), aargs,
                                    vtab, ftab, callpos)),
          tp)
```

In the typechecker, the strategy is actually very similar. Here instead we use checkFunWithVtable, but we still construct the dummy FunDec. We compute the argument types by extracting it from Params.

```
  | checkFunArg (In.Lambda(tp ,params, body, fpos),
                                    vtab, ftab, pos) =
```

```
        let
          val (Out.FunDec(_, tp', params', body', fpos)) =
              checkFunWithVtable (In.FunDec ("", tp, params,
                                  body, fpos),vtab, ftab, pos)
          val arg_types = map (fn Param(_,t) => t) params
          in
              (Out.Lambda(tp', params', body', fpos), tp,
                                                  arg_types)
         end
```

Code generation for Lambdas use a different idea. Since an anonymous function is only ever used where it is defined, we can just compile the body in place and use it as is.

```
      | applyFunArg (Lambda (tp, params, body, fpos), args,
                                      vtable, place, pos) =
          let
            val tmp_reg = newName "tmp_reg"
            val param_names = map (fn Param(s,_) => s) params
            val bindings = SymTab.fromList (ListPair.zip
                                        (param_names, args))
            val vtab_local = SymTab.combine bindings vtable
            val lambdaBody = compileExp body vtab_local tmp_reg
          in
            lambdaBody @ [ Mips.MOVE(place, tmp_reg) ]
          end
```

In the code above, we first extract parameter names from the definition and bind them to the given arguments. Then we make a new, local value table with the bindings added. Using this table we compile the body of the expression. Lastly, we move the result into 'place'.

Task 4

The first thing we did was to define what the copyConstPropFoldExp function should do, if it finds a variable, which is look up the vtable to see if it is already defined, and if it is, replacing it with the known value.

```
      | Var (name, pos) => (case SymTab.lookup name vtable of
                               NONE => Var(name, pos)
                             | SOME (ConstProp x) =>
                                          Constant (x, pos)
                             | SOME (VarProp n) => Var (n,pos)
                             )
```

Then we changed Let, to actually handle variables. If a variable or a constant with a defined value is found, that informaton will be place that the vtable. Otherwise it will try remove the variable from the vtable, since it is no longer known.

```
    val vtable' = (case e' of
            Constant (v, _) =>
                          SymTab.bind name (ConstProp v) vtable
            | Var (n, _) => SymTab.bind name (VarProp n) vtable
            | _          => SymTab.remove name vtable
            )
```

Lastly all that remained was to write the actual optimizations for Times, Divide, Negate, Not, And and Or.

Times returns 0 if either of the constants multiplied is 0. If one of the constants is 1, the other constant is returned.

```
   | 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, _), _) =>
              Constant (IntVal 0, pos)
          | (_, Constant (IntVal 0, _)) =>
              Constant (IntVal 0, pos)
          | (Constant (IntVal 1, _), _) =>
              e2'
          | (_, Constant (IntVal 1, _)) =>
              e1'
          | _ =>
              Times (e1', e2', pos)
     end
```

Divide returns only the first constant, if the second is 1.

```
       case (e1', e2') of
            (Constant (IntVal x, _), Constant (IntVal y, _)) =>
              Constant (IntVal (x*y), pos)
          | (_, Constant (IntVal 1, _)) =>
              e1'
          | _ =>
              Divide (e1', e2', pos)
```

Negate and Not both returns the variable, if they are called twice.

```
          case e' of
               Negate(Negate(x, _),_) => x
             | _                      => e
...
          case e' of
               Not(Not(x, _), _) => x
             | _                 => e
```

If any of the constants in And or Or is known, they can be optimized. For And, it one of the variables is true, true is returned. If the first va