FASTO - Technical Report Milestone

Implementation of Programming Languages 2022
Department of Computer Science
University of Copenhagen

Kristoffer Colding-Poulsen <bwk298>
Wadah Ahmed <rds343>
Oscar Kofod <qhm995>

Friday, May 27, 12:00

Contents

0 Introduction

This report describes the implementation of an optimizing compiler for the FASTO language. The compiler has been developed in F# using the .NET 6.0 SDK as part of the course *Implementation of Programming Languages* at UCPH. Most of the compiler has already been completed and this report only describes implementation of the missing parts.

We were tasked to extend the FASTO compiler by getting legal programs to be interpreted and translated into MIPS code correctly. Programs that are illegal should be rejected with an appropriate feedback given to the user. This is done by adhering the grammar of the language and by implementing the missing functionalities in all the compiler parts; lexer, parser, interpreter, type checker and the MIPS code generator.

The missing parts were split into three parts, the implementation and evaluation of which will be described in detail in the following sections.

1 Task 1: Simple Expressions

The first task was to implement the following productions of `FASTO`s syntax:

`integer-multiplication operator`	:	$Exp \to Exp * Exp$
`integer-division operator`	:	$Exp \to Exp / Exp$
`boolean-and operator`	:	$Exp \to Exp$ `&&` Exp
`boolean-or operator`	:	$Exp \to Exp$ `||` Exp
`boolean-true value`	:	$Exp \to$ `true`
`boolean-false value`	:	$Exp \to$ `false`
`boolean-negation unary operator`	:	$Exp \to$ `not` Exp
`integer-negation unary operator`	:	$Exp \to$ `~`Exp

We were further to generalize the syntax to allow multiple variable declarations in a single `let-in` statement. This required an extension of the syntax with nonterminals and productions for variable declarations. Then `let-in` was modified:

`variable-declaration`	:	$Dec \to ID = Exp$
`variable-declaration list`	:	$Decs \to Dec$ `;` $Decs$
`multiple-declaration lets`	:	$Exp \to$ `let` $Decs$ `in` Exp

1.1 Lexing and parsing

The lexer required only simple extensions to include the missing symbols above. The arithmetic and boolean parsing was similarly simple, as it followed the style of the existing implementation. The `and` and `or` operations had the lowest precedence of the boolean operations. The boolean and arithmetic negations were to be non associative. These modifications can be seen at lines 1-8 of Listing 1.

The functionality for the multiple-declaration `let-ins` required nontrivial extensions to the parser. Here we added a nonterminal for `Decs` which is a list of at least one variable declaration. Which when parsed were folded into a chain of `let-in` expressions.

```
%token <Position> TRUE FALSE NOT AND OR
%left OR
%left AND
%left DEQ LTH
%nonassoc NOT
%left PLUS MINUS
%left TIMES DIVIDE
%nonassoc NEGATE
Exp : LET Decs IN Exp %prec letprec
        { List.foldBack (fun dec exp -> Let (dec, exp, $1)) $2 $4 }
;
Decs : ID EQ Exp SEMICOLON Decs { Dec (fst $1, $3, $2) :: $5 }
     | ID EQ Exp                { Dec (fst $1, $3, $2) :: [] }
```

Listing 1: Excerpt of changes made to `Parser.fsp` for Task 1

Lastly we made an extension of the anonymous binary integer operations.

1.2 Type checking

This extension was likewise very simple for the binary operators added, these were basically copied from the existing implementation. The unary operators were different but as simple, their form can be seen in Listing 2. As the multi-variable `let-ins` were simply parsed as nested expression, there was no need for altering either type checking or other functionality.

```
and checkExp (ftab : FunTable) (vtab : VarTable) (exp : UntypedExp)
           : (Type * TypedExp)  =
    match exp with
    | Not (e, pos) ->
        let (t, e_dec) = checkExp ftab vtab e
        if t <> Bool then
            reportTypeWrong "argument of unary operator" Bool t pos
        (Bool, Not (e_dec, pos))
```

Listing 2: Excerpt of changes made to `TypeChecker.fs` for Task 1

1.3 Interpretation

The evaluation of the new expressions was easily added to the interpreter, written in F#. They mostly followed the simple form of the existing code, except in the nontrivial cases of `division`, `or`, and `and`. The first of which were to check for division by 0. The last two were to be *short-circuiting*. Handling these cases involved matching on the result of one expression and only advancing if it makes sense as seen in the case for `dnd` in Listing 3.

```
let rec evalExp (e : UntypedExp, vtab : VarTable, ftab : FunTable) : Value =
  match e with
  | And(e1, e2, pos) ->
    match evalExp(e1, vtab, ftab) with
    | BoolVal b1 when not b1 -> BoolVal (b1)
    | BoolVal b1 ->
      match evalExp(e2, vtab, ftab)  with
      | BoolVal b2 -> BoolVal (b2)
      | res2 -> reportWrongType "right operand of &&" Bool res2 (expPos e2)
    | res1 -> reportWrongType  "left operand of &&" Bool res1 (expPos e1)
```

Listing 3: Excerpt of changes made to `Interpreter.fs` for Task 1

1.4 Compiler

Lastly the the operators were added to the compiler. The nontrivial cases of `divide` and `and`, which was similar to that of `or`, can be seen in subsection A.1.

To exit properly when dividing by zero, we made use of the predefined error messages at the end of the MIPS code. This code is only reached if the check at line 13 fails. For `and`, we first set the the return value `place` to 0. Then if the first operand is false, we immediately return. But if both operands are true, we change this value to 1 before returning.

1.5 Evaluation

A satisfactory compiler, needs not only produce the expected outputs, but also explanatory error messages, when various errors occur. The latter was tested by continuously updating and running a test script.

The former was tested with a test suite following the style shown in subsection A.2. These tests tested for correct precedence and calculations.

2 Task 2: Array Combinators

2.1 Lexing and parsing

FASTO includes function that perform computation on arrays. The implemention of `map` and `reduce` had already been completed. We were tasked with extending the compiler with implementation of:

- `replicate(n, a)` which takes an integer *n* and *a* element as input and returns a list of length *n* of *a* elements.
- `filter(f, [a])` which takes a predicate function and an array as input. The function is then applied on the array and the elements satisfying the predicate is returned.
- `scan(f, e, [a])` Finally, scan receives the same arguments as reduce but produces an array of the same length as the input array, by computing all prefix-sums under the given operator.

Like the arithmetic and boolean operators by identifying the keywords in the lexer and adding them as tokens to the parser. We have derived the productions of the function from the abstract syntax tree that has already been defined for us.

```
  | "replicate" -> Parser.REPLICATE  pos
  | "scan"      -> Parser.SCAN       pos
  | "filter"    -> Parser.FILTER     pos
```

lexer.fs

```
   | REPLICATE LPAR Exp COMMA Exp RPAR
                     { Replicate ($3, $5, (), $1) }
   | FILTER LPAR FunArg COMMA Exp RPAR
                     { Filter ($3, $5, (), $1) }
   | SCAN LPAR   FunArg COMMA Exp COMMA Exp RPAR
                     { Scan ($3, $5, $7, (), $1) }
   | SCAN LPAR OP BinOp COMMA Exp COMMA Exp RPAR
                     { Scan ($4, $6, $8, (), $1) }
```

parser.fs

A scan operation can also be parsed given a binary operation over an array. An extra detail that could help write more compact and possibly faster computations.

2.2 Type checking

To verify that the type system's rules are respected, we have used the abstract syntax tree to for the expression constructors and derived the following type constraints.

`replicate`	:	$int * a \rightarrow [a]$
`filter`	:	$(a \rightarrow bool) * [a] \rightarrow [a]$
`scan`	:	$(a * a \rightarrow a) * a * [a] \rightarrow [a]$

2.2.1 Replicate

The idea behind type checking `replicate(n, a)` is to recursively type check n and check that it has integer type. The resulting array after calling `replicate` should be an array of type `a_type` (`[a_type]`).

```
| Replicate (e1, e2, _, pos) ->
    let (t1, e1') = checkExp ftab vtab e1
    let (t2, e2') = checkExp ftab vtab e2
    match t1 with
    | Int -> (Array t2, Replicate (e1', e2', t2, pos))
    | _ -> reportTypeWrongKind "arguments of replicate" "int" t1 pos
```

TypeChecker.fs

2.2.2 Filter

Type checking `filter(f, a)` is implemented by checking if the array expression has the type `[type_elm]` for some `type_elm`. Thereafter we check that `type_elm` has the same type as the arguments for the predicate function. Lastly, we determine if the predicate function return a boolean, if so return `[type_elm]`. This functionality is shown in the code snippet below.

```
if elem_type <> f_arg_type then
      reportTypesDifferent "function-argument and
          array-element types in filter" f_arg_type elem_type pos
    if f_res_type <> Bool then
      reportTypesDifferent "return type of operation in filter"
            f_res_type Bool pos
    (Array elem_type, Filter (f', arr_dec, elem_type, pos))
```

TypeChecker.fs

2.2.3 Scan

The implementation of `scan(f, ne, arr)` was greatly inspired by the implementation of `reduce` as they share most of the functionality. The only difference being `scan` scan's return type is the same as the type of `arr`, while `reduce` return type is that of an element of `arr`.

2.3 Interpretation

Extending the interpreter to be able to interpret the array combinators was a trivial task. We had the liberty to use the already existing functions in F#'s `List` module. Futher exception handling was however still required. A quick rundown is given below:

- `replicate(n, a)`: The arguments of the function should evaluate as follow; `n` must evaluate to `a` and `n` must be a positive integer.
- `filter(p, arr)`: The interpreter checks that the predicate `p` has a boolean return type. Then a check is made to determine that the input array is indeed valid. If so, the expression is evaluated with the `List.filter` function.
- `scan(f, ne, arr)`: The implementation of scan is again a copy of `reduce` but the expression is evaluated to an array of the same length as the input.

2.4 Compiler

The strategy we used to implement the code generation is to try to write the code in C code first and then try to translate it to MIPS code.

2.4.1 Replicate

The code generation of the `replicate(n, a)` function is done by allocating an appropriate size of memory for the array by using the built-in `dynalloc` function. a while loop is then ran, were `n` elements of `a` is then copied to the newly allocated array. Before the loop we check that both the arguments are positive and keep looping until we have an array of length `n`.

2.4.2 Filter

The code generation for `filter(p, arr)` is similar to `map`. Below is the `C`-like code generation.

The predicate function is called on every element in the array. If the result is true then we put the element in then new array, otherwise we continue and check the next element. This implementation, even though it works, is quite inefficient. For example when creating the output we allocate the space for it as the input array, but not all elements may satisfy the predicate, thus ending up wasting space. We also add padding by multiplying by four, which is a further waste of space.

```
len = length(arr);
b   = malloc(len*4);
i   = 0;
j   = 0;
while(i < len) {
    tmp  = p(a[i]);
    if (tmp == true) {
        b[j] = arr[i];
        j = j + 1;
        i = i + 1;
    } else {
          i = i+1;
    }
};
```

C code for `filter`

2.4.3 Scan

Code generation of `scan(f, ne, arr)` was also inspired by code generation of `reduce`. However, created an accumulator of the result in every iteration of the loop. This required us to use a couple of extra registers to keep temporal values.

After creating the labels for the control flow and initializing the register, the elements are loaded into the temporary register. The function is then called with the elements and the result is stored in the output array. The code snippet below demonstrates the functionality.

2.5 Evaluation

Our development of the compiler has mostly been test-driven. We have made use of the test suite that has been delivered to us and used it as a bench mark for the correction of the implementation. All the tests have passed successfully without errors nor warnings.

```
(* Load arr[i] into tmp_reg *)
  let elem_size = getElemSize elm_type
  let load_code =
    [ mipsLoad elem_size (tmp_reg, arr_reg, 0)
    ; Mips.ADDI (arr_reg, arr_reg, elemSizeToInt elem_size)
    ]
  (* place := binop(acc_reg, tmp_reg) *)
  let apply_code =
        applyFunArg(binop, [acc_reg; tmp_reg], vtable, acc_reg, pos)

  (* Store accumulator in output array and icnrement output array *)
  let store_code = [ mipsStore elem_size (acc_reg, addr_reg, 0)
        ;  Mips.ADDI(addr_reg, addr_reg, elemSizeToInt elem_size)
        ]
```

3 Task 3: Optimisations

The last task was to finish the implementation of three optimisations; copy propagation, constant folding, and dead-binding removal. All cases had existing implementation for all but variable, array index, and `let` binding expressions. Constant folding for integer multiplication and logical and expressions, were also missing.

3.1 `CopyConstPropFold.fs`

The extensions made for copy/constant propagation of `let` can be seen in subsection B.1. We start by matching on the variable declaration expression to see if it can be easily propagated to the body expression. In the case of a nested `let` expression, we first check the body of the nested `let` for a constant to propagate. If not we check if the variable declaration can be optimised. Otherwise we return the fall-through case with no optimisations done.

The extensions made for copy/constant propagation of variable and index expressions as well as constant folding, can be seen in subsection B.2. For the former, we simply looked up the given variable or array name in the propagated symbol table, to see if it contains a constant or reference to another variable which can be returned, otherwise we do nothing.

For constant folding, return something similar to the cases of the `plus` and `or` expressions. For example, we check whether or not any operand of a multiplication is 1 or 0, in which case we can return the other operand expression, without regard for its contents, or return 0, respectively.

3.2 `DeadBindingRemoval.fs`

The extensions for dead-binding removals can be seen in subsection B.3.

For the case of a variable name, we mark the variable name as used, and indicate that the expression does not contain I/O.

In the case of array indexes, we again mark the variable name as used, and let the expression indexing the array decide whether I/O is used.

For the `let`-bindings, we make recursive checks on both the variable declaration and body. We remove the name declared in the let expression from the `vtable` of the body. Lastly, if the variable name is not used in the body expression or if the variable declaration does not contain I/O we simply return the expression of the body.

3.3 Evaluation

When running the test suite with optimisations, all tests passed, but the file `filter-on-2darr.fo` gave the warning; `_fun_arg_res_85_ spilled`. Meaning not enough memory was allocated. As this was not a fatal error, and also not an obvious result of our extensions, we elected to ignore this.

Beyond that we wrote a small test script that was substantially optimised with `CopyConstPropFold.fs`, with one statement that would obviously increase runtime if not ignored. The results are shown in subsection B.4.

4 Conclusion

In conclusion, we now have a working compiler for FASTO language. Through testing we are confident in its ability produce correct programs.

A Task 1

A.1 Code Generation

```
let rec compileExp (e : TypedExp) (vtable : VarTable) (place : Mips.reg)
                      : Mips.Instruction list =
    match e with
    | Divide (e1, e2, pos) ->
        let t1 = newReg "divide_L"
        let t2 = newReg "divide_R"
        let code1 = compileExp e1 vtable t1
        let code2 = compileExp e2 vtable t2
        let nz = newLab "nonzero"
        let z  = newReg "zero"
        code1 @ code2 @
        [ Mips.LI (z,  0)
        ; Mips.BNE (t2, z, nz)
        ; Mips.LI(RN5, fst pos)
        ; Mips.LA(RN6, "_Msg_DivZero_")
        ; Mips.J "_RuntimeError_"
        ; Mips.LABEL nz
        ; Mips.DIV (place, t1, t2)
        ]
    | And (e1, e2, pos) ->
        let t1 = newReg "and_L"
        let t2 = newReg "and_R"
        let code1 = compileExp e1 vtable t1
        let code2 = compileExp e2 vtable t2
        let ret = newLab "return"
        let t   = newReg "true"
        code1 @ [ Mips.LI  (place,  0)
                ; Mips.ORI (t, t1,  1)
                ; Mips.BNE (t1, t, ret)
                ] @
        code2 @ [ Mips.BNE (t2, t, ret)
                ; Mips.LI  (place,  1)
                ; Mips.LABEL ret
                ]
```

Listing 9: Changes made to `CodeGen.fs` for task 1

A.2 Evaluation

```
fun int negation(int a, int b, int c) =
  let x = ~a + b * c == b * c - a
    ; y = ( ( (~a) + (b*c) ) == ( (b * c) - a) ) in
  let z = if x == y then 1 else 0 in
  write(z)

fun int precedence(int a, int b, int c) =
  let x = a / b + 4 * ~a / a * c - b / a
    ; y = (( (a / b) + (((4 * (0 - a)) / a) * c)) - (b / a)) in
  let z = if x == y then 1 else 0 in
  write(z)

fun int main() =
  let n0 =   negation(2, 6, 4 - 5) in
  let p0 = precedence(2, 6, 4 - 5) in
  1
```

Listing 10: arithmetic.fo

B Task 3

B.1 Copy / Constant Propagation

```
let rec copyConstPropFoldExp (vtable : VarTable) (e : TypedExp) =
    match e with
    | Let (Dec (name, e, decpos), body, pos) ->
        let e' = copyConstPropFoldExp vtable e
        match e' with
        | Var (name', pos') ->
            let ntable = SymTab.bind name (VarProp name') vtable
            copyConstPropFoldExp ntable body
        | Constant (x', pos') ->
            let ntable = SymTab.bind name (ConstProp x') vtable
            copyConstPropFoldExp ntable body
        | Let (Dec (name', e'', decpos'), body', pos') ->
            match body' with
            | Constant (c, p) ->
                let vtab = SymTab.bind name (ConstProp c) vtable
                copyConstPropFoldExp vtab body
            | _ ->
                let ie = copyConstPropFoldExp vtable e'
                match ie with
                | Constant (c, p) ->
                    let vtab = SymTab.bind name (ConstProp c) vtable
                    copyConstPropFoldExp vtab body
                | Var (c, p) ->
                    let vtab = SymTab.bind name (VarProp c) vtable
                    copyConstPropFoldExp vtab body
                | _ ->
                    let body' = copyConstPropFoldExp vtable body
                    Let (Dec (name, e', decpos), body', pos)
            | _ -> (* Fallthrough  - for everything else, do nothing *)
                let body' = copyConstPropFoldExp vtable body
                Let (Dec (name, e', decpos), body', pos)
```

Listing 11: Changes made to CopyConstPropFold.fs for task 3

B.2 Copy / Constant Propagation and Constant Folding

```
let rec copyConstPropFoldExp (vtable : VarTable) (e : TypedExp) =
    match e with
    | Var (name, pos) ->
        match SymTab.lookup name vtable with
        | Some (ConstProp  x') -> Constant (x', pos)
        | Some (VarProp name') -> Var   (name', pos)
        | None -> Var (name, pos)
    | Index (name, e, t, pos) ->
        let e' = copyConstPropFoldExp vtable e
        match SymTab.lookup name vtable with
        | Some (VarProp name') -> Index (name', e', t, pos)
        | _ -> Index (name, e', t, pos)
    | Times (e1, e2, pos) ->
        let e1' = copyConstPropFoldExp vtable e1
        let e2' = copyConstPropFoldExp vtable e2
        match (e1', e2') with
        | (Constant (IntVal x, _), Constant (IntVal y, _)) ->
            Constant (IntVal (x * y), pos)
        | (Constant (IntVal 1, _), _) -> e2'
        | (Constant (IntVal 0, _), _) -> Constant (IntVal 0, pos)
        | (_, Constant (IntVal 1, _)) -> e1'
        | (_, Constant (IntVal 0, _)) -> Constant (IntVal 0, pos)
        | _ -> Times (e1', e2', pos)
    | And (e1, e2, pos) ->
        let e1' = copyConstPropFoldExp vtable e1
        let e2' = copyConstPropFoldExp vtable e2
        match (e1', e2') with
        | (Constant (BoolVal a, _), Constant (BoolVal b, _)) ->
            Constant (BoolVal (a && b), pos)
        | (Constant (BoolVal a, _), _) when not a -> Constant (BoolVal false, pos)
        | (_, Constant (BoolVal b, _)) when not b -> Constant (BoolVal false, pos)
        | _ -> And (e1', e2', pos)
```

Listing 12: Changes made to `CopyConstPropFold.fs` for task 3

B.3 Dead Binding Removal

```
let rec removeDeadBindingsInExp (e : TypedExp)
                    : (bool * DBRtab * TypedExp) =
    match e with
    | Var (name, pos) ->
        (true, recordUse name (SymTab.empty()), e)
    | Index (name, e, t, pos) ->
        let (eio, euses, e') = removeDeadBindingsInExp e
        (eio, recordUse name euses, Index (name, e', t, pos))

    | Let (Dec (name, e, decpos), body, pos) ->
        let (eio, euses,    e') = removeDeadBindingsInExp e
        let (bio, buses, body') = removeDeadBindingsInExp body
        let dbr = SymTab.combine euses buses
        if not (isUsed name buses || eio) then
            (bio, recordUse name dbr, body')
        else
            (eio || bio, recordUse name dbr,
             Let (Dec (name, e', decpos), body', pos))
```

Listing 13: Changes made to DeadBindingRemoval.fs for task 3

B.4 Evaluation

```
fun int copyconstpropfold([int] a, int b) =
  let x = 5 in
  let y = x in
  let i = (let i = reduce(op +, 2, iota(1000)) in  2) in
  let j = a in
  let k = (let a = (let a = b in 3) in j[a + 2] * a) in
  b

fun int main() =
  let a = iota(1000) in
  let b = map(fn int (int x) => copyconstpropfold(a, 100), a) in
  let d = reduce(op +, 0, b) in
  d
```

Listing 14: optimisation.fo

```
fun int copyconstpropfold([int] a, int b) =
    let k = (a[5] * 3) in
    b

fun int main() =
    let a = iota(1000) in
    let b = map(fn int (int x) => copyconstpropfold(a, 100), a) in
    let d = reduce(fn int (int x, int y) => (x + y), 0, b) in
    d
```

Listing 15: bin/fasto.sh -p c task3tests/optimisation.fo

```
fasto % time bin/fasto.sh -i task3tests/optimisation.fo
...  1.13s user 0.04s system 99% cpu 1.179 total

fasto % bin/fasto.sh -o c task3tests/optimisation.fo
fasto % time bin/mars.sh task3tests/optimisation.asm
...  0.66s user 0.08s system 188% cpu 0.394 total

fasto % bin/fasto.sh -c task3tests/optimisation.fo
fasto % time bin/mars.sh task3tests/optimisation.asm
...  1.85s user 0.09s system 102% cpu 1.891 total
```

Listing 16: Runtimes of optimisation.fo