Group Project - FASTO

Anders Brandhof '190493' SGL135
Andreas Jørgensen '240594' SRV415
Casper Iversen '090691' JVP497
Søren Jensen '270792' PWS412

20. december 2015

Indhold

1 Opgaven

Vi skal i denne gruppe opgave færdiggøre en optimeret compiler til FASTO. FASTO er et simpelt første-ordens funktionelt sprog.

2 Design

Compileren er blevet lavet i sml. For at færdiggøre compileren har vi ændret i de følgende filer:

- Lexer.lex
- Parser.grm
- Interpreter.sml
- TypeChecker.sml
- CopyConstPropFold.sml
- CodeGen.sml

Vi vil i afsnittene gennemgå de vigtigste ændringer vi har lavet.

2.1 Lexer

I filen Lexer.lex har vi fået givet tokens og keywords. Vi har her skulle tilføje flere tokens og keywords til FASTO. Det har her været map, reduce, iota, true, false mm..
Vi har tilføjet keywords i funktionen "keyword (s, pos), dette har vi gjort på følgende måde:

```
fun keyword (s, pos) =
case s of
     "map"    => Parser.MAP pos
     "reduce" => Parser.REDUCE pos
     "true"   => Parser.BOOLVAL (true,pos)
     "false"  => Parser.BOOLVAL (false,pos)
```

Vi har kun beskrevet et par af de keywords vi har lavet, men det viser hvordan vi har gjort det.
Vi har også tilføjet tokens til Lexer.lex, dette har vi gjort på følgende måde:

```
  | '+'    { Parser.PLUS (getPos lexbuf) }
  | '-'    { Parser.MINUS (getPos lexbuf) }
  | "&&"   { Parser.AND (getPos lexbuf) }
  | "not"  { Parser.NOT (getPos lexbuf) }
  | _      { lexerError lexbuf "Illegal symbol in input" };
```

Vi har kun vist et udsnit af de tokens vi har lavet. Som det kan ses på eksemplet, bruger vi både ' ' og . Forskellen på dette er, at ' ' er en char og er en string. Grunden til vi bliver nød til at definere det som en string, er fordi det er sammensat af mere end en char, og derfor skal være en string for at blive genkendt.

2.2 Parser

I filen Parser.grm ser vi på hvordan vi bruger de tokens vi har lavet i Lexer.lex. Her er et lille udsnit af, hvordan vi har gjort dette:

```
%token <int*(int*int)> NUM
%token <bool*(int*int)> BOOLVAL
%token <char*(int*int)> CHARLIT
%token <(int*int)> AND OR
%token <(int*int)> NOT
```

Her ligger man mærke til, at vi har skrevet (int*int). Dette er for, at vi ved hvorhenne den pågældende token er blevet brugt. Dette gør det muligt for os at kunne returnere en relevant fejlbesked med lokationen i dokumentet.

Vi har i Parser.grm også skulle definere precedens af f.eks. +, -, · og /. Dette har vi gjort ud fra operatornes præcedens i matematikken.

Vi har også skulle tage stilling til, hvordan f.eks. DIVIDE skal bruge dens Exp, parameter etc.. Nedenunder har vi vist et udsnit af, hvordan vi har gjort dette:

```
| BOOLVAL          { Constant (BoolVal (#1 $1), #2 $1) }
| Exp TIMES  Exp { Times ($1, $3, $2) }
| Exp DIVIDE Exp { Divide ($1, $3, $2) }
```

2.3 TypeChecker

I filen TypeChecker sikrer vi os, at vores funktioner har de rette argumenter, og at typerne er korrekte. Hvis vi f.eks. ser på '+', skal vi sikre os, at argumentet til højre og til venstre for '+'et, er integers. Det skal derfor ikke være muligt, at kunne lægge en string sammen med en integer.
Lad os kigge på vores implementation af map:

```
| In.Map (f, arr_exp, _, _, pos)
      => let val (a_type,b) = checkExp ftab vtab arr_exp
             val (f_type,k,p) = checkFunArg(f, vtab, ftab, pos)
         in
           if a_type = k
           then (k, Out.Map(f_type, b, k, a_type, pos))
           else raise Error ("Map: Wrong argument type" ^ ppType a_type, pos)
         end
```

Vi sikrer os her, at vi får en funktion og en array. Det er derfor vigtigt, at vi returner en fejl, hvis vi ikke modtager de rigtige argument typer. Som man kan se i vores kode ovenover, tager vi netop højde for dette. For at gøre det

nemmere at se, hvor det går galt med typerne, vælger vi at sende positionen af det forkerte argument med i fejlbeskeden. Dette har vi valgt at gøre med alle type tjek i filen TypeChecker.

2.4 Interpreter

2.5 CopyConstPropFold

2.6 CodeGen

CodeGen er der hvor vores reale handlinger for vores sprog sker. Det er her vi skriver vores Mips kode og sørger for at de rigtige funktioner bliver udført af funktionerne når de bliver brugt. for at forstå dette bedre lad os kigge på et stykke af vores funktion Map:

```
val loop_map = case getElemSize elem_type of
         One => Mips.LB (res_reg, elem_reg, "0")
                :: applyFunArg (farg, vtable, [res_reg], res_reg, pos)
                @ [Mips.ADDI (elem_reg, elem_reg, "1")]
                @ [Mips.ADDI (addr_reg, addr_reg, "1")]
       |Four => Mips.LW(res_reg, elem_reg, "0")
                :: applyFunArg (farg, vtable, [res_reg], res_reg, pos)
                @ [Mips.ADDI (elem_reg, elem_reg, "4")]
                @ [Mips.ADDI (addr_reg, addr_reg, "4")]
```

Hvad vi gør i dette stykke er vi tjekker for at se om vores element i listen er 1 eller 4 for at beslutter hvor langt fremme det næste element i listen så er. Og hvad dette betyder er blot om elementet kun er 1 bit eller om det er et word vi skal loade
Når vi så har fundet ud af dette loader vi vores word/bit til res_ reg og smider den ind i vores applyfunarg funktion som håndtere vores funktion alt efter om den er en anonym funktion eller blot en string. Derefter hopper vi frem i vores array så vi er klar til at loade det næste element i næste iteration. Og alt dette samles da til en liste af Mips kommandoer vi kalder til sidst.

Hvad der dog har meget stor betydning for netop denne funktion er håndteringen af vores funktion, derfor lad os kigge på vores applyFunArg og se hvad vi gør anderledes når vi møder en anonym funktion:

```
and applyFunArg (FunName fname, vtable, aargs, place, pos) =
    let val tmp = newName "tmp_reg"
    in
      applyRegs(fname,aargs, tmp, pos) @ [Mips.MOVE(place, tmp)]
    end
  | applyFunArg ((Lambda(tp, pars, body, _)), vtable, aargs, place, pos) =
    let fun bindArgToVtab ((Param(fa,_)::fargs), (aa::aargs), vtab) =
            SymTab.bind fa aa (bindArgToVtab (fargs, aargs, vtab))
          | bindArgToVtab (_, _, vtab) = vtab
        val tmp = newName "tmp_reg"
        val new_vtab = bindArgToVtab (pars, aargs, vtable)
    in
      (compileExp body new_vtab place) @ [Mips.MOVE(place, tmp)]
    end
```

Hvad vi har gjort her er at vi tjekker om det er en anonym funktion og hvis det er bruger vi den opgivede funktion applyregs til at håndtere dem og ellers går vi ned til vores lambda del.
Det vi så har gjort i denne del er blot at lave en funktion som modtager en Param Liste, end arg liste og en vtab. Dette kalder vi så blot rekusivt til vi er igennem enten alle elementerne i en liste eller begge lister og får så vtable. Hvad der sker er blot at vi binder elementerne i listen til vtable. Så bruger vi da funktionen til at lave en ny vtab som vi bruger i compileExp og sætter så værdien til placeringen som vi får fra applyFunargs.
Alt det her gør blot at vi fortolker vores funktion til et Mips kald vi kan bruge. Funktinoen her vil både kunne bruges i Map men også i reduce da begge funktionerne er meget simulere, med den forskel at reduces funktion har 2 input, mens Map kun har 1.

Måde vi ville have implementeret Reduce på ville have været meget simuler med Map dog med forskellen at der skulle have været en værdi er blev gemt for hver iteration som kunne bruges til næste som blot er forskellen på reduce og map (og at reduce har 1 element mere, men det ville vores applyFunArg stadig godt kunne håndtere med lidt kreativ tænkning.)

3 Tests