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

I filen CopyConstPropFold har vi optimeret compileren. Dette har vi gjort ved brug af simple matematiske operationer. Hvis vi ser på vores gange operation, har vi valgt at optimere den ved, at definere $0 \cdot x$ som værende lig med 0, og $1 \cdot x$ som værende lig med x. Det ser måske ikke ud af meget, men vil få det til at køre hurtigere. Vores implementation af dette kan ses i nedenstående kode:

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

Denne form for optimering har vi lavet for flere af funktionerne. I forhold til opgaven stilt i Task 3, har vi også lavet følgende ændring i koden til Copy Propagation:

```
fun copyConstPropFoldExp vtable e =
    case e of
      (* Copy propagation is handled entirely in the following three
      cases for variables, array indexing, and let-bindings. *)
        Var (name, pos) =>
        (case SymTab.lookup name vtable of
             SOME (VarProp newname) => Var (newname, pos)
           | SOME (ConstProp value) => Constant (value, pos)
           | _                      => Var (name, pos))
      | Index (name, e, t, pos) =>
        (case SymTab.lookup name vtable of
             SOME (VarProp newname) =>
               Index (newname, copyConstPropFoldExp vtable e, t, pos)
           | _ =>
               Index (name, copyConstPropFoldExp vtable e, t, pos))
      | Let (Dec (name, e, decpos), body, pos) =>
        let val e' = copyConstPropFoldExp vtable e
```

```
in case e' of
        Var (varname, p) =>
        (case Symtab.lookup varname vtable of
             SOME (VarProp newname) => Var (newname, p)
           | _                      => Var (varname, p))
      | Constant (value, p) =>
        (case Symtab.lookup value vtable of
             SOME (ConstProp newvalue) => Constant (newvalue, p)
           | _                         => Constant (valuename, p))
      | Let (Dec bindee, inner_body, inner_pos) =>
        let val new_vtab = vtable
            val new_vtab = SymTab.bind name (VarProp varname) new_vtab
        in copyConstPropFoldExp new_vtab e'
        end
        | Constant (value, p) =>
          let val new_vtab = vtable
              val new_vtab = SymTab.bind name (ConstProp value) new_vtab
          in copyConstPropFoldExp new_vtab e'
          end
        | Let (Dec bindee, inner_body, inner_pos) =>
        raise Fail "Cannot copy-propagate Let yet"
      | _ => (* Fallthrough - for everything else, do nothing *)
        let val body' = copyConstPropFoldExp vtable body
        in Let (Dec (name, e', decpos), body', pos)
        end
end
```

2.6 CodeGen

3 Tests