Oversættere
Eksamensopgave 4: Void og &

Henrik Bendt, GWK553, 191191-1767

20. januar 2012

Indhold

1 Indledning

Jeg har brugt den udleverede løsning til G-opgaven som udgangspunkt. Der skulle ikke tilføjes meget til lexeren eller parseren. Typecheckeren skulle udvides med et par ydderligere typecheck. Hoveddelen af arbejdet lå i at lave Compiler.sml, her især &-operatoren, der bl.a. gør brug af delmænger af registerallokatorens kode. Til sidst blev oversætteren testet med diverse testprogrammer skrevet i 100, både udleverede og selvlavede.

2 Lexer.lex

Strengen "void" er tilføjet til `keyword` og giver `Parser.VOID` med positionen (se app. A, fig. 1). Tegnet "&" er tilføjet som et symbol og giver `Parser.ADRESSOF` med positionen (se app. A, fig. 2). "return" og ";" bliver allerede fanget som henholdsvis `keyword` og symbol.

3 Parser.grm og S100.sml

Tilføjede &-operatoren som en `ADRESSOF`-token og void-typen som en `VOID`-token (se app. B, fig.3). Parserens gramatik blev udvidet med produktionerne"*return;*" (strengen "return" efterfulgt af semikolon) (se app. B, fig.5) og "& LVAL" (se app. B, fig.6). Derudover blev typen `VOID` tilføjet som type (se app. B, fig.4).

`S100.sml` blev udvidet med typen `void` (se app. C, fig.7), udtrykket `AdressOf` (se app. C, fig.8) og erklæringen `Returnvoid` (se app. C, fig.9). Disse følger alle beskrivelserne fra opgaven og har en position. Grammatikken er entydig som før, da de tilføjede produktioner ikke skaber konflikter med andre knuder i det abstrakte syntakstræ.

4 Type.sml

`Type.sml` fik tilføjet typen void (se app. D, fig.11). Derudover blev signaturen `Type.sig` også udvidet med typen void (se app. D,fig.10).

For at kunne bruge typen void i typecheckeren, blev `convertType` også udvidet til at kunne håndtere void (se app. D,fig.11). `checkDecs` er også blevet udvidet til at kaste fejl, hvis void optræder i den dekleration af en variable (se app. D,fig.13).

`checkStat` blev tilføjet *casen* `S100.Returnvoid`, der tjekker, om returtypen af funktionen er void. Hvis den ikke er, bliver en fejlmeddelelse sendt, fordi kodestykket `return;` kun er tilladt, hvis returtypen er void. *Casen* `S100.Return` blev modificeret til at kaste fejl, hvis returtypen er void, idet der ikke må optræde kodestykker som "*return e;*" i en funktion, der returnere void (se D,fig.14). Derudover blev `checkFunDec` modificeret til at fortsætte, hvis enten `checkReturn`

er sand (hvis et returkald vil altid blive nået) eller returtypen er void, da den i så fald må returnere, når den når enden af funktionskroppen (se app. D,fig.15).

Derudover er `checkReturn` også blevet udvidet med et tjek på `S100.Returnvoid`, der returnere falsk, selvom det egentlig er underordnet, idet en void-returtype ikke kræver et returkald, og hvis returtypen er andet end void, vil returtypen være forkert og blive fundet i enten `checkFunDec` eller `checkStat` (se D,fig.15). Eftersom funktioner med returtypen void ikke behøver et returkald, er det naturligt ikke at tælle et returkald af formen `return;` med som et "gyldigt" returkald. Dette gør samtidig, at oversætteren spare kald og derved stopper tidligere, i de tilfælde hvor en funktion, der ikke returnere void, kun har returkald af formen `return;`.

`checkExp` blev tilføjet *casen* `S100.AdressOf`, der håndtere &-operatoren. Der kastes fejl, hvis &-operatoren bliver brugt med en reference og ellers returneres en referencetype til hvad end typen af variablen er, da udtrykket følgeligt vil være en reference (se app. D,fig.12).

5 Compiler.sml

Variable, der bliver tagget med &-operatoren, kan enten spildes på stakken eller gemmes på stakken som caller-saves.

Jeg valgte at spilde dem, eftersom jeg derved kunne genbruge nogle funktioner fra registerallokatoren. Alle variable, der skal spilles som følge af &-operatoren, bliver spildt før registerallokatoren bliver kørt i `compileFun`. Når `compileStat` er blevet kørt, altså at funktionskroppen er blevet oversat til MIPS-kode, er alle variable, der skal spildes, blevet tilføjet listen `spilledVars`, der ligger i registerallokatoren.

Derpå køres `bodyspill`, der sørger for, at alle variable, der er spildt indtil nu, bliver gemt på stakken i starten af funktionen. Derpå bliver funktionen `spill` fra registerallokatorens kode i `RegAlloc.sml` brugt til at sørge for, at hver gang en spildt variable bliver brugt i funktionskroppen, så gemmes den i et register, og hvis den får tildelt en ny værdi, gemt på stakken det samme sted igen. Til sidst bliver alle spildte variable gemt til et register igen, så registerallokatoren senere kan lave en liveness analyes af variablene – gemme dem der skal gemmes og skrotte resten. For hvert spild bliver der reserveret 4 bytes (et ord) på stakken, også selvom det kun er en tegnkonstant der skal gemmes, da det gør koden for oversætteren simplere. Registerallokatoren gør det samme. Derudover bliver registerallokatoren kaldt en parameter, der angiver antallet af spildte variable – denne er derfor ikke længere 0, men antallet af spildte variable, som findes ved at tælle variable der bliver spildt i `bodyspill` (se app. E,fig.18).

For at kunne bruge funktionen `spill` fra registerallokatoren, blev denne tilføjet til registerallokatorens signatur i `regAlloc.sig`. Det samme blev `spilledVars`; listen hvor spildte variable gemmes på.

I `compileExp` blev *casen* `S100.AddressOf` tilføjet. Hvis `compileLval` returnerer en hukommelsesadresse, var erklæringen et opslag, og adressen returneres

direkte (se E,fig.16).

Hvis en variable returneres, skal den spildes, hvis den ikke allerede er. Det tjekkes via funktionen `member`, der tjekker, om en variable ligger på listen over spildte variable. Samtidig tælles hvor mange elementer den ligger nede, eftersom det bestemmer afsættet fra stakpegeren den skal lægges. Hvis den allerede er spildt, returneres adressen den vil lægges på, som er det talte afsæt fra stakpegeren. Ellers spildes den ved at blive tilføjet bagerst i listen over spildte variable, via funktionen `addset`, og derefter returneres adressen den vil lægges på, hvor afsættet til stakpegeren bestemmes af det samlede antal af elementer på listen indtil videre (eftersom variablen lægges i slutningen af listen). Det er samme procedure for heltal som tegnkonstanter. Desuden bliver returtypen en reference til den originale type (se app. E,fig.16).

Der bliver ikke håndteret referencer i `S100.AddressOf`, da disse ikke er tilladte (reference til en reference), og de bliver stoppet i typecheckeren.

`S100.Returnvoid` blev tilføjet til `compileStat` på samme måde som `S100.Return` (se app. E,fig.17).

6 Tests

Oversætteren er testet på DIKUs systemer; oversætteren giver fejl, hvor fejl skal gives, og reagerer som forventet på både testprogrammerne fra G-opgaven K-opgaven.

Alle test med "error" i navnet betyder, at den skal give fejl. Disse er de vedlagte tests fra K-opgaven samt deres betydning:

- `void-swap` tester brugen af &-operatoren variable og referencer, kald med disse som argumenter, foruden kald med variable, med og uden/efter brug af &-operatoren, som argumenter.
- `void-sort` tester for, at en funktion med returtypen void kan slutte uden returkald. Den tester også brug af opslag med &-operatoren.
- `void-error01` tester brugen af typen void på en variable.
- `void-error02` tester returkald af typen void i `main`-funktionen, der returnere heltal.
- `void-error03` tester returkald af anden type end void i en funktion med returtypen void.
- `void-error04` tester at bruge &-operatoren på en reference, i.e. lave en reference til en reference.

Jeg fremstillede ydderligere testprogrammerne

- `void-address01` tester brugen af mange &-operatore og pegere til disse på heltal.
- `void-address02` tester brugen af mange &-operatore og pegere til disse på tegnkonstanter.

- `void-address03` tester funktion med returtypen void og et returkald til void der aldrig kan nåes.
- `void-error05` tester `main`-funktion med returtypen void.
- `void-error06` tester funktion med returtype void og et returkald af forkert type, men som aldrig kan nåes.
- `void-error07` tester `main`-funktion uden returkald.
- `void-error08` tester om en reference til en tegnkonstant kan referere til et heltal. Dette kaster en fejl. Det er dog ikke klart fra hverken G-opgaven eller K-opgaven, om en tegnreference må blive en heltalsreference, og omvendt, så det antages at den ikke må.

Alle tests reagerer som forventet.

Det kan antages ud fra testene, at oversætteren kan behandle alt fra G-opgaven. Derudover kan den håndtere alle tilfælde beskrevet ovenfor, samt programmer der består af delmænger fra testprogrammerne. Det kan dog ikke garanteres, at oversætteren ikke fejler på et bestemt program, der indeholder dele, som ikke er testet ved ovenstående testprogrammer. Derudover er meget store programmer ikke blevet testkørt, men det må antages, at eftersom et stort program symbolsk kan opdeles til flere små, at oversætteren kan oversætte uden fejl.

Liveness analysen af spildte variable er ikke testet og jeg har heller ikke undersøgt, om det er nødvendigt at hente spildte variable ind til sidst i funktionen, før *callee-saves* bliver gemt. Jeg valgte dog at hente spildte variable efter funktionskroppen, da registerallokatoren efterfølgende laver en *liveness*-analyse og derpå bliver alle *callee-saves* gemt, så jeg antog, at de skulle ligge i et register, for at det kunne lade sig gøre.

Der er ikke testet for at løbe tør for stakplads, men det antages at der kastes en fejl som "out of memory".

7 Evaluering

Der kan spares plads på stakken, ved at lade spildte tegnkonstanter fylde kun en byte, i stedet for fire, som bliver reserveret nu. Dette vil spare 3/4-dele plads, opfyldt af tegnkonstanter, på stakken, hvilket kan være betydeligt, hvis det er f.eks. et tekstbehandlingsprogram. Det vil dog gøre behandling af stakken det mere besværlig, fordi der nu skal tages højde for, hvilken type der gemmes på stakken – et register kan ikke bare gemmes mere (da et register fylder et maskinord). Derfor har jeg ikke taget højde for det, men hvis oversætteren skal behandle mange tegnkonstanter eller blot har meget begrænset plads, bør det revurderes.

Man kunne også have omskrevet registerallokatoren til at *caller-save* alle variable der bliver tagget med &-operatoren. Dette vil dog ikke give en hurtigere oversætter, idet den spild-funktion jeg har indført laver lige så mange kald til `spill`-funktionen, som hvis det havde været registerallokatoren, og hele instruktionslisten for funktionskroppen skal for hver spildt variable løbes igennem, da

en spildt variable skal hentes og/eller gemmes hver gang den optræder i en instruktion.

Alle spild bliver allokeret før funktionskroppen bliver kørt, hvilket vil sige, at de er allokeret hele funktionen igennem. Man kunne have spildt dem, som de bliver brugt i koden, og samtidig have lavet en *liveness*-analyse, så variable bliver deallokeret løbende, som de ikke skal bruges længere. Dog er der et problem:Antag at tre variable spildes, og den første bliver spildt/allokeret først på stakken, dernæst den anden og til sidst den sidste variable. Den første variable bliver dør (skal ikke efterfølgende bruges) dog først, hvilket vil sige at den deallokeres. Dette kan dog ikke lade sig gøre, idet der er to elementer allokeret efter denne; de to andre variable. Dernæst dør den anden variable og problemet er det samme. Først når den sidste variable dør, kan hele pladsen deallokeres.

Problemet kan løses ved, at alle variable, der skal spildes, spildes i starten af funktionen, som jeg har gjort det. Derved lader man en *liveness*-analyse sortere dem efter hvornår de dør, så den første, der dør, allokeres sidst på stakken osv. Derved deallokeres pladsen løbende, som variable dør. Dette vil dog reelt set ikke gøre den store forskel, idet stakken stadig vil have reserveret pladser for alle spildte variable fra start af. Derfor er fordelen ikke særlig stor ift. min implementation, idet det kun vil gøre en forskel, hvis stakken kan blive påvirket under køretid af programmet; hvis input kan diktere, hvor meget der skal lægges på stakken. Dette plejer hoben dog at tage sig af.

Man kunne have spildt til hoben i stedet for stakken. Det havde dog givet problemer ift. deallokering (frigivelse) af den plads, der var blevet allokeret til spildet, da der dels ikke er implementeret en sådan funktion og da der heller ikke er en `garbage collector` (spildopsamler). Derved kunne hoben meget hurtigt blive fyldt, hvis der blev spildt mange variable (f.eks. ved store rekursive kald, hvor der bliver spildt hver iteration), og dette ville aldrig kunne blive brugt igen. Stakken kan selvfølgelig også blive fyldt, men hvis spildningen er spredt over flere funktioner, vil pladsen blive frigivet hver gang – antaget at spildet ikke bliver givet videre mellem funktionerne. Derfor var det en fordel at spilde til stakken i denne opgave, men med en *garbage collector* ville dette kunne revurderes. Samtidig ville ovenstående forbedringsforslag have god effekt, fordi hoben ofte bliver påvirkert under køretid.

8 Bilag

A Lexer

```
fun keyword (s, pos) =
    case s of
        "if"          => Parser.IF pos
      | "else"        => Parser.ELSE pos
      | "int"         => Parser.INT pos
      | "char"        => Parser.CHAR pos
      | "void"        => Parser.VOID pos
      | "while"       => Parser.WHILE pos
      | "return"      => Parser.RETURN pos
      | _             => Parser.ID (s, pos)
```

Figur 1: Tilføjede til "void" Keywords

```
| `,`                 { Parser.COMMA (getPos lexbuf) }
| `;`                 { Parser.SEMICOLON (getPos lexbuf) }
| `*`                 { Parser.DEREF (getPos lexbuf) }
| `&`                 { Parser.ADDRESSOF (getPos lexbuf) }
| `[`                 { Parser.LBRACK (getPos lexbuf) }
| `]`                 { Parser.RBRACK (getPos lexbuf) }
| `{`                 { Parser.LBRACE (getPos lexbuf) }
| `}`                 { Parser.RBRACE (getPos lexbuf) }
```

Figur 2: Tilføjede symbol '&'

B Parser

```
%token <int*(int*int)> NUM
%token <string*(int*int)> ID STRINGCONST
%token <char*(int*int)> CHARCONST
%token <(int*int)> IF ELSE INT CHAR VOID WHILE RETURN
%token <(int*int)> PLUS MINUS LESS EQUAL ASSIGN DEREF ADDRESSOF
%token <(int*int)> LPAR RPAR LBRACK RBRACK LBRACE RBRACE
%token <(int*int)> COMMA SEMICOLON EOF
```

Figur 3: Tilføjede tokens VOID og ADDRESSOF

```
Type :    INT       { S100.Int $1 }
    | CHAR      { S100.Char $1 }
    | VOID      { S100.Void $1 }
;
```

Figur 4: Tilføjede typen VOID

```
Stat :    LBRACE Decs1 Stats RBRACE
            { S100.Block ($2,$3,$1) }
    | Exp SEMICOLON { S100.EX $1 }
    | IF LPAR Exp RPAR %prec ELSE Stat
            { S100.If ($3,$5,$1) }
    | IF LPAR Exp RPAR Stat ELSE Stat
            { S100.IfElse ($3,$5,$7,$1) }
    | WHILE LPAR Exp RPAR Stat
            { S100.While ($3,$5,$1) }
    | RETURN Exp SEMICOLON
            { S100.Return ($2,$1) }
    | RETURN SEMICOLON
            { S100.Returnvoid $1 }
;
```

Figur 5: Tilføjede check på RETURN SEMICOLON

```
Exp :     NUM       { S100.NumConst $1 }
    | CHARCONST { S100.CharConst $1 }
    | STRINGCONST   { S100.StringConst $1 }
    | Lval      { S100.LV $1 }
    | Lval ASSIGN Exp
           { S100.Assign ($1,$3,$2) }
    | ADDRESSOF Lval {S100.AddressOf ($2,$1) }
    | Exp PLUS Exp  { S100.Plus ($1, $3, $2) }
    | Exp MINUS Exp { S100.Minus ($1, $3, $2) }
    | Exp LESS Exp  { S100.Less ($1, $3, $2) }
    | Exp EQUAL Exp { S100.Equal ($1, $3, $2) }
    | ID LPAR Exps RPAR
           { S100.Call (#1 $1, $3, $2) }
    | LPAR Exp RPAR
           { $2 }
```

Figur 6: Tilføjede check på ADRESSOF Lval

C S100

```
datatype Type
  = Int of pos
  | Char of pos
  | Void of pos
```

Figur 7: Tilføjede typen Void

```
datatype Exp
  = NumConst of int * pos
  | CharConst of char * pos
  | StringConst of string * pos
  | LV of Lval
  | Assign of Lval * Exp * pos
  | AddressOf of Lval * pos
  | Plus of Exp * Exp * pos
  | Minus of Exp * Exp * pos
  | Equal of Exp * Exp * pos
  | Less of Exp * Exp * pos
  | Call of string * Exp list * pos
```

Figur 8: Tilføjede AddressOf til udtryk

```
datatype Stat
  = EX of Exp
  | If of Exp * Stat * pos
  | IfElse of Exp * Stat * Stat * pos
  | While of Exp * Stat * pos
  | Return of Exp * pos
  | Returnvoid of pos
  | Block of Dec list * Stat list * pos
```

Figur 9: Tilføjede Returnvoid til erklæringer

D Typechecker

```
datatype Type = Int | Char | Void | Ref of Type
```

Figur 10: Tilføjede typen void til Type.sig, typecheckerens signatur

```
datatype Type = Int | Char | Void | Ref of Type

fun convertType (S100.Int _) = Int
  | convertType (S100.Char _) = Char
  | convertType (S100.Void _) = Void
```

Figur 11: Tilføjede typen void og udvidede `convertType` til at behandle void

```
| S100.AddressOf (lv,p) =>
    let
      val t = checkLval lv vtable ftable
    in
        case t of
             Ref _ => raise Error("Can't have a reference to a reference",p)
           | _     => (Ref t) (*Is now a reference*)
    end
```

Figur 12: Tilføjede AddressOf som et udtryk, der kalder fejl hvis den modtager en reference

```
fun checkDecs [] = []
  | checkDecs ((t,sids)::ds) =
  let
    val ty = convertType t
    val pos = case t of
              (S100.Void p) => p
             |(S100.Int p)  => p
             |(S100.Char p) => p
  in
    if ty = Void
      then raise Error ("Cannot have type void",pos)
    else
      extend (List.rev sids) (convertType t) (checkDecs ds)
  end
```

Figur 13: Udvidede checkDecs, der tjekker om en erklæring har fået typen void, og i så fald kaldes fejl

```
| S100.Return (e,p) =>
    if resultT <> Void andalso checkExp e vtable ftable = promote resultT
    then ()
    else raise Error ("Wrong return type",p)
| S100.Returnvoid p =>
    if resultT = Void
    then ()
    else raise Error ("Wrong return type",p)
```

Figur 14: Udvidede erklæringen `Return` til at kalde fejl, hvis funktionens returtype er void. Tilføjede erklæringen ReturnVoid: et returkald med typen void

```
fun checkReturn s =
  case s of
    S100.EX e => false
  | S100.If (e,s1,p) => false
  | S100.IfElse (e,s1,s2,p) => checkReturn s1 andalso checkReturn s2
  | S100.While (e,s1,p) => false
  | S100.Return (e,p) => true
  | S100.Returnvoid (e,p) => false
  | S100.Block (decs,stats,p) => List.exists checkReturn stats

fun checkFunDec (t,sf,decs,body,p) ftable =
      if checkReturn body orelse convertType t = Void then
        checkStat body (checkDecs decs) ftable (getType t sf)
      else
        raise Error (getName sf ^" can end without a reachable return statement",p)
```

Figur 15: Udvidede `checkReturn` med Returnvoid og `checkFunDec` med muligheden for at en funktion med returtype void ikke har et returkald

E Compiler

```
| S100.AddressOf (lval,p) =>
    let
      val t = "_AddressOf_"^newName()
      val (code0,ty,loc) = compileLval lval vtable ftable
      val off = ref 0
      fun member [] _ = false
        | member (v::table) x = if v=x then true
                                else (off := !off+1; member table x)
      fun addset x = (RegAlloc.spilledVars := (!RegAlloc.spilledVars) @ [x];())
    in
      case (ty,loc) of
        (Type.Int, Reg x) =>
        (Type.Ref Type.Int,
        (if member(!RegAlloc.spilledVars) x then
             [Mips.ADDI(t,SP,makeConst (!off*4)),Mips.MOVE (place, t)]
        else
           (addset x;
           [Mips.ADDI(t,SP,makeConst (!off*4)),Mips.MOVE (place, t)])))
      | (Type.Char, Reg x) =>
          (Type.Ref Type.Char,
          (if member(!RegAlloc.spilledVars) x  then
             [Mips.ADDI(t,SP,makeConst (!off*4)),Mips.MOVE (place, t)]
          else
           (addset x;
           [Mips.ADDI(t,SP,makeConst (!off*4)),Mips.MOVE (place, t)])))
      | (Type.Int, Mem x) =>
          (Type.Ref Type.Int, code0 @ [Mips.MOVE
          (t,x), Mips.MOVE(place,t)])
      | (Type.Char, Mem x) =>
          (Type.Ref Type.Char, code0 @ [Mips.MOVE
          (t,x), Mips.MOVE(place,t)])
      | _ => raise Error ("Bad Lval",(0,0))
    end
```

Figur 16: Tilføjede udtrykket AddressOf til `compileExp`

```
| S100.Returnvoid p =>
    let
      val t = "_return"^newName()
    in
      [Mips.MOVE("2",t), Mips.J exitLabel]
    end
```

Figur 17: Tilføjede ReturnVoid til `compileStat`

```
val body = compileStat body vtable ftable (fname ^ "_exit")

(*Counts total number of spills by AdressOf*)
val spills = ref 0

(*Spills all AdressOf variables at the start of the
function body and adds load and save of the variables before and
after they are used.*)
val bodyspill =
  let
    fun doSpill [] ilist num = ilist
      | doSpill (v::tbl) ilist num =
      let
        val offset = makeConst (num*4)
        val _ = spills := (!spills)+1
      in
          doSpill tbl ([Mips.SW (v,SP,offset)]@(RegAlloc.spill ilist v
          offset)@[Mips.LW(v,SP,offset)]) (num+1)
      end
  in
      doSpill (!RegAlloc.spilledVars) body 0
  end


val (body1, _, maxr,spilled)  (* call register allocator *)
    = RegAlloc.registerAlloc
      (parcode @ bodyspill) [] 2 maxCaller maxReg (!spills)
```

Figur 18: Tilføjede variablen `spills` og funktionen `bodyspill` til `compileFun`. Disse står for spildning af variable

F Register allocator

```
val spill : Mips.mips list -> string -> string -> Mips.mips list
(* Runs a list of instructions through and if the variable is in a used or
 * destination register, it is loaded and stored accordingly by the given
 * offset.
 * Returns the instruction list with the new loads and store instructions*)

val spilledVars : string list ref
(* The list of spilled variables *)
```

Figur 19: Uvidede signaturen til `RegAlloc`, så listen `spilledVars` og funktionen `spill` kan tilgåes