Group Project for Compilers 2013/14

Sample report by teachers and teaching assistants

January 10, 2014

Contents

Introduction

This report covers extensions for the Paladim compiler as covered in the G-assignment. We describe the most significant challenges we encountered and our solutions to them. We briefly discuss the tests we performed.

1 Task 1: Parser.grm

We wrote a parser specification for `mosmlyac` from scratch based on the handed out grammar. The grammar was changed in certain places to avoid shift/reduce conflicts. Tokens were named differently than in the handed out *LL1Parser.sml*, so *Lexer.lex* was adjusted to generate these new tokens:

Parser.grm (tokens)

```
%token <(int*int)> IF THEN ELSE INT BOOL CHAR WHILE DO RETURN ASSIGN PROG VAR
%token <(int*int)> ARR OF BEG END FUN PROC EOF PLUS MINUS TIMES DIV EQ LTH
%token <(int*int)> LPAR RPAR LBRACKET RBRACKET LCURLY RCURLY COMMA SEMICOL COLON
%token <(int*int)> TRUE FALSE NOT AND OR OP
%token <string*(int*int)> ID STRINGLIT
%token <int*(int*int)> NUM
%token <char*(int*int)> CHARLIT
```

The new parser was added in *Driver.sml* as suggested. The changes in *Lexer.lex* were throughout the file, but were only related to changing the token names and module references, e.g. from `LL1Parser.TProg` to `Parser.PROG` and so on.

Shift/Reduce Conflicts

We encountered a number of shift/reduce conflicts that were resolved mostly by precedence/associativity and in a few places by rewriting the grammar.

Associativity and Precedence of Binary Expression Operators

To resolve precedence and associativity for logical and arithmetic expression operators as specified in the assignment text, precedence rules are added as follows:

Parser.grm (precedences)

```
%nonassoc ifprec
%nonassoc ELSE
%left OR
%left AND
%nonassoc NOT
%left EQ LTH
%left PLUS MINUS
%left TIMES DIV
...
```

Note that parentheses must be used for comparing composed boolean expressions: `1=1 = true` type checks, but `1=1 = 0=0` does not. To achieve this, one must use parentheses: `1=1 = (0=0)`.

Dangling Else and Declaration Blocks

The dummy token `ifprec` is introduced and is given a lower precedence than the `ELSE` token. The `%prec` precedence predicate subsequently gives the *if-then* production a precedence lower than `ELSE` as well. When a shift/reduce-conflict occurs after `Blk` in the productions below, this will *shift* `ELSE` and thus give precedence to the first of these productions.

Parser.grm (dangling else)

```
Stmt : RETURN Exp                    { AbSyn.Return  (SOME $2, $1) }
       ...
```

```
     | IF Exp THEN  Blk ELSE  Blk     { AbSyn.IfThEl  ($2, $4, $6, $1) }
     | IF Exp THEN  Blk %prec ifprec  { AbSyn.IfThEl  ($2, $4, AbSyn.Block([], []), $1) }
```

This way, we force the parser to choose *if-then-else* whenever this is possible, making *else* bind to the nearest *if-then-* as intended. Correct behaviour can be verified by the following test program:

Test program *dangling.pal*

```
program dangling;
procedure main()
var i : int;
begin
   i := 0;
   if false then if true then i := 1 else i := 2;
   if i = 0 then write("passed.\n") else write("failed.\n");
end;
```

Productions of `OP` and Declaration Blocks

Most grammar rules were based directly on the given **Paladim** grammar, but some changes were made to avoid conflicts due to lookahead. None of these grammar transformations changes the language.

The `OP` productions under `Exp` → `Exp OP Exp` were inlined:

Parser.grm (expression operators)

```
Exp :     NUM            { AbSyn.Literal( AbSyn.BVal(AbSyn.Num(#1 $1)), #2 $1) }
          ...
        | Exp PLUS  Exp  { AbSyn.Plus ($1, $3, $2) }
        | Exp MINUS Exp  { AbSyn.Minus($1, $3, $2) }
          ...
```

The `DBlock` productions were inlined so that `Blk` contains variable declarations directly:

Parser.grm (conflict in Decs rule)

```
/* This variant follows the published grammar, produces a conflict */
/*
 *   Decs    : Decs Dec SEMICOL { $1 @ [$2] }
 *           | Dec SEMICOL      { $1::[] }
 *   Blk     : DBlk SBlk        { AbSyn.Block($1, $2) }
 *   DBlk    : VAR Decs         { $2 }
 *           |                  { [] }
 */

/* this variant fixes the conflict */
Decs : Decs Dec SEMICOL { $1 @ [$2] }
     | Dec SEMICOL      { [$1] }
Blk  : VAR Decs SBlk    { AbSyn.Block($2, $3) }
     |          SBlk    { AbSyn.Block([], $1) }
```

Task 1 was tested by parsing all test programs (delivered and ours) correctly.

2 Task 2: More operators

We uncommented the constructors for multiplication, division, logical disjunction (`or`) and logical negation (`not`) in *TpAbSyn.sml*. We also added cases for them to the helper functions `pp_exp`, `typeOfExp` and `posOfExp`.

Type-checking for the added operations is quite straightforward: code for all binary operators is mostly identical to the one for addition and subtraction. This code is explained together with task 3 and the version for multiplication shown on page 6. Type-checking expressions with `not` is particularly brief:

Type.sml (type-checking not)

```
    | typeCheckExp ( vtab, AbSyn.Not (e, pos), _) =
        let val e_new = typeCheckExp(vtab, e, KnownType (BType Bool))
        in Not(e_new, pos) end
```

In the interpreter, we defined helper functions similar to `evalAnd`:

TpInterpret.sml (binary operation helper functions)

```
fun evalOr (BVal (Log b1), BVal (Log b2), _) = BVal (Log (b1 orelse b2))
  | evalOr (v1, v2, pos) =
    raise Error( "Or: argument types do not match. Arg1: " ^
                 pp_val v1 ^ ", arg2: " ^ pp_val v2, pos )

fun evalNot (BVal (Log b), pos) = BVal (Log (not b))
  | evalNot (v, pos) =
    raise Error( "Not: argument type does not match. Arg1: " ^ pp_val v, pos )
```

Expressions with the added operators are evaluated similar to other expressions:

TpInterpret.sml

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

   | evalExp ( Or(e1, e2, pos), vtab, ftab ) =
        let val r1 = evalExp(e1, vtab, ftab)
            val r2 = evalExp(e2, vtab, ftab)
        in  evalOr(r1, r2, pos)
        end
```

Multiplication and division in the compiler are very similar to addition and subtraction, as the MIPS module provides simplistic machine instructions MUL (ignoring overflows) and DIV:

Compiler.sml

```
    | compileExp( vtable, Times(e1, e2, _), place ) =
        let val t1 = "times1_" ^ newName()
            val c1 = compileExp(vtable, e1, t1)
            val t2 = "times2_" ^ newName()
            val c2 = compileExp(vtable, e2, t2)
        in c1 @ c2 @ [Mips.MUL (place,t1,t2)]
        end

    | compileExp( vtable, Div(e1, e2, _), place ) =
        let val t1 = "div1_" ^ newName()
            val c1 = compileExp(vtable, e1, t1)
            val t2 = "div2_" ^ newName()
            val c2 = compileExp(vtable, e2, t2)
        in c1 @ c2 @ [Mips.DIV (place,t1,t2)]
        end
```

The `or` operation is implemented lazily in the same way as the `and` operation (with the bugfix). The `not` operation is even simpler, just negating the value, here done using `XORI`.

Compiler.sml

```
    | compileExp( vtable, Or(e1, e2, _), place ) =
        let val t1 = "or1_" ^ newName()
            val c1 = compileExp(vtable, e1, t1)
            val t2 = "or2_" ^ newName()
            val c2 = compileExp(vtable, e2, t2)
            val lA = "_or_" ^ newName()
        in c1 (* do first part, skip 2nd part if already true *)
           @ [ Mips.MOVE (place, t1), Mips.BNE (place, "0", lA) ]
           @ c2 (* when here, t1 was false, so the result is t2 *)
           @ [ Mips.MOVE (place, t2), Mips.LABEL lA ]
        end

    | compileExp( vtable, Not(e1, _), place ) =
        let val t1 = "not1_" ^ newName()
            val c1 = compileExp(vtable, e1, t1)
        in c1 @ [ Mips.XORI (place, t1, "1") ]
        end
```

Task 2 is tested using the program *prec_ assoc.pal* to verify precedence and associativity.

3 Task 3: Type-check support for special functions

Simple type inference is added throughout *Type.sml* by recursively passing *expected types* to all sub-expressions. This way, type checking polymorphic expressions `read` and `new` can rely on these types.

Expected Types

We know that a plus expects numbers (`Int`), so this type is passed to the function calls that check the operands of `Plus`. Expected types are added to all checks for arithmetic expressions (expecting `Int`) and logical expressions (expecting `Bool`) in this way, shown here only for `Plus` and `And`:

Type.sml

```
    | typeCheckExp( vtab, AbSyn.Plus (e1, e2, pos), _ ) =
        let val e1_new = typeCheckExp( vtab, e1, KnownType(BType Int) )
            val e2_new = typeCheckExp( vtab, e2, KnownType(BType Int) )
            val (tp1, tp2) = (typeOfExp e1_new, typeOfExp e2_new)
        in  if  typesEqual(BType Int, tp1) andalso typesEqual(BType Int, tp2)
            then Plus(e1_new, e2_new, pos)
            else raise Error("in␣type␣check␣plus␣exp,␣one␣argument␣is␣not␣of␣int␣type␣"^
                             pp_type tp1^"␣and␣"^pp_type tp2^"␣at␣", pos)
        end

    | typeCheckExp ( vtab, AbSyn.And (e1, e2, pos), _ ) =
      let val e1_new = typeCheckExp(vtab, e1, KnownType (BType Bool) )
          val e2_new = typeCheckExp(vtab, e2, KnownType (BType Bool) )
          val (tp1, tp2) = (typeOfExp e1_new, typeOfExp e2_new)
      in if typesEqual(BType Bool, tp1) andalso typesEqual(BType Bool, tp2)
         then And(e1_new, e2_new, pos)
         else raise Error("in␣type␣check␣and,␣one␣argument␣is␣not␣of␣bool␣type␣"^
                          pp_type tp1^"␣and␣"^pp_type tp2^"␣at␣", pos)
      end
```

Polymorphic Cases

When type-checking `Equal` and `Less`, we check each operand without expecting any type. If `read` should fail for this reason, this is caught and turned into `NONE` rather than `SOME exp`. Cases where one operand is polymorphic and the other one isn't are resolved by re-type-checking the other using the now known type.

Only combinations where both operands are polymorphic will result in `(NONE, NONE)`. If an operand should fail type-checking for other reasons, those errors will be re-thrown as the operand that failed to type-check is checked again. (If there is a type-error in both operands not related to polymorphism, the error message could be improved.)

Type.sml

```
(* The following allows either left or right to be polymorphic, but not both *)
| typeCheckExp ( vtab, AbSyn.Equal(e1, e2, pos), _ ) =
  let val e1_try = SOME ( typeCheckExp (vtab, e1, UnknownType) ) handle _ => NONE
      val e2_try = SOME ( typeCheckExp (vtab, e2, UnknownType) ) handle _ => NONE
      val (e1_new, e2_new) =
          case (e1_try, e2_try) of
          (SOME a, SOME b) => (a, b)
        | (SOME a, NONE  ) => (a, typeCheckExp (vtab, e2, KnownType(typeOfExp a)))
        | (NONE  , SOME b) => (typeCheckExp (vtab, e1, KnownType(typeOfExp b)), b)
        | (NONE  , NONE  ) => raise Error("in␣type␣check␣equal,␣neither␣operand" ^
                                          "type-checks␣(possibly␣polymorphism)," ^
                                          "␣at␣", pos)

      val (tp1, tp2) = (typeOfExp e1_new, typeOfExp e2_new)

      val _ = typesEqual(tp1, tp2) orelse
              raise Error("in␣type␣check␣equal,␣argument␣types␣do␣not␣match,␣"^
                          pp_type tp1^"␣isn't␣"^pp_type tp2^"␣at␣", pos)

      val _ = case tp1 of
                  Array _ => raise Error("in␣type␣check␣equal,␣first␣expression␣"^
                                         pp_exp e1_new^ "␣is␣an␣array␣(of␣type)␣"^
                                         pp_type tp1^"␣at␣", pos)
                | _ => ()

  in Equal(e1_new, e2_new, pos) end

| typeCheckExp ( vtab, AbSyn.Less (e1, e2, pos), _ ) =
   let (* ...same... *)
   in Less(e1_new, e2_new, pos) end
```

This code was tested by the program *readtest.pal*, which makes different calls to `read` to test that the correct types are inferred. Program *testTypeInference.pal* was another test, we used it after removing the calls to `map`.

Type-checking `new`

The type of `new` is $int_1 * \ldots * int_m \rightarrow Array(m, \alpha)$, where m is the number of actual arguments of `new`, and α is a basic type determined from the context via the *expected type* mechanism. The typing rule for `new` is:

- *if* m, the number of arguments of `new` is greater than 0,
- and all arguments of `new` are integers,
- and the basic type, α of the result array can be extracted from the expected type,

- *then* the result type is $Array(m, \alpha)$,
- *else* type checking fails.

Type.sml

```
   | typeCheckExp ( vtab, AbSyn.FunApp ("new", args, pos), etp ) =
1    ( case expectedBasicType etp of
2        SOME btp =>
3          let val expint  = KnownType (BType Int)
4              val new_args= map   ( fn x => typeCheckExp(vtab, x, expint) ) args
5              val tps     = map   ( fn x => typeOfExp x ) new_args
6              val okargs  = foldl ( fn (x,b)=>typesEqual(x,BType Int) andalso b)
7                                  true tps
8              val rtp      = Array ( length args, btp )
9          in  if okargs andalso (length args) > 0
10             then FunApp( ("new", (tps, SOME rtp)),  new_args, pos)
11             else raise Error("in call to new, arg not of type int", pos)
12         end
13     | NONE     => raise Error("type inference fails because "^
14                               "of unknwon expected basic type, at ", pos) )
```

Type-checking `new` is fairly straightforward:

1,13-14 If the expected type is unknown then type inference fails because the basic-element type of the result array cannot be determined.

1-2 Otherwise, the expected basic-element type of the result array is extracted in `btp`.

3-4 Each actual argument of `new` is type checked by passing the `int` expected type, because the arguments of `new` represent the sizes of the result-array dimensions.

5-7 The (computed) types of the arguments are checked to be all `int`.

11 Otherwise type checking fails. Type checking also fails if `new` receives no arguments because **Paladim** does not allow empty arrays.

8 The result is an array of rank equal to the number of arguments of `new`, i.e., each argument denotes a dimension's size, and of basic-element type `btp`.

10 A typed representation, i.e., `TpAbSyn.sml`, is constructed and returned for the `new` function call, e.g., `(tps,SOME rtp)` denotes the signature of `new`: `tps` is the list of argument types and `rtp` is the result-array type, i.e., `Array(length args, btp)`.

This code was tested by our modified version of *testTypeInference.pal*, which makes several different calls to `new`. Error cases were tested by the small trivial program *arrFail.pal*:

Test program *arrFail.pal* for array errors

```
program arrFail;
procedure main()
var a : array of array of int;
begin
   a := new(2,2);
   // use any of these to test type errors of new
   //a := new(1); a := new(1,2,3); a := new(1=0,2);

   // use any of these to test runtime failures for indexing
   write(a[2,0]); write(a[0-1,0]); write(a[0,2]); write(a[0,0-1]);
end;
```

4 Task 4: Indexing

Type checking

The type rule in **Paladim** for an index expression containing m indices is very similar to the one for `new`:

- *if* $m > 0$ and all index expressions have integer type,
- and the array variable has type $Array(m, \alpha)$, where α is a basic type
- *then* the result is of type α
- *else* type checking fails.

Type.sml

```
|  typeCheckExp(vtab, AbSyn.LValue( AbSyn.Index(id, inds), pos ), _) =
1   let
2     val new_inds = map (fn x => typeCheckExp(vtab, x, KnownType (BType Int))) inds
3
4     (* check all indices evaluate to int scalars *)
5     val tp_inds  = map (fn e => typeOfExp e) new_inds
6     val ok_inds  = List.all (fn x => typesEqual (x, BType Int)) tp_inds
7     val _ = if ok_inds then true
8             else raise Error("␣in␣indexed␣variable␣"^id^"[␣"^pp_exps new_inds^
9                              "␣]␣not␣all␣indices␣are␣ints,␣at␣", pos);
10
11   (* check that array is defined and has correct number of dimensions *)
12   val id_tp =
13       case SymTab.lookup id vtab of
14         SOME (Array(r,btp)) =>
15           if(r = length inds andalso r > 0) then Array(r,btp)
16           else raise Error("␣in␣indexed␣variable␣"^id^"[␣"^pp_exps new_inds^
17                            "␣]␣num␣of␣indexes␣does␣not␣match␣the␣array␣rank,"^
18                            "␣at␣", pos)
19       | SOME tp =>
20           raise Error("␣in␣indexed␣variable␣"^id^"[␣"^pp_exps new_inds^
21                       "␣],␣var␣"^id^"␣is␣of␣non-array␣type␣"^pp_type tp^
22                       ",␣at␣", pos)
23       | NONE    =>
24           raise Error("in␣type␣check␣indexed␣exp," ^
25                       "␣var␣"^id^"␣not␣in␣VTab,␣at␣", pos)
26
27  in LValue(Index ((id, id_tp), new_inds), pos)
28  end
```

The implementation is also similar to the one of `new`:

2-5 the arguments types are computed, by passing an integer expected type,

6-9 arguments types are checked to be all integers,

12-26 the type of the associated array variable is taken from `vtable` and it is checked that the number of indices matches the array rank and is greater than 0.

27 Finally, the typed representation of an index expression is returned.

Compilation of index expressions

How is an offset into an array for an index offset computed?
Let a be an array of int of rank n, with dimensions $[d_1, \ldots d_n]$, and let $[s_1, s_{n-1}]$ be the

stored strides. When indexing into a in an expression $a[e_1, \ldots, e_n]$, and assuming a function v to evaluate an expression to an int, the computed offset o inside the array data space is:

$$o = \sum_{j=1}^{n-1} \mathsf{v}(e_j) \cdot s_j + \mathsf{v}(e_n) \tag{1}$$

$$= \mathsf{v}(e_1) \cdot s_1 + \mathsf{v}(e_2) \cdot s_2 + \ldots + \mathsf{v}(e_{n-1}) \cdot s_{n-1} + \mathsf{v}(e_n) \tag{2}$$

$$= \mathsf{v}(e_1) \cdot \underbrace{d_2 \cdot \ldots d_n}_{s_1} + \mathsf{v}(e_2) \cdot \underbrace{d_3 \cdot \ldots d_n}_{s_2} + \ldots + \mathsf{v}(e_{n-1}) \cdot d_n + \mathsf{v}(e_n) \tag{3}$$

$$= (\ldots(\mathsf{v}(e_1)) \cdot d_2 + \mathsf{v}(e_2)) \cdot d_3 + \ldots + \mathsf{v}(e_{n-1})) \cdot d_n + \mathsf{v}(e_n) \tag{4}$$

Equation 2 indicates how to compute using the strides. When using $s_j = d_{j+1} \cdot \ldots \cdot d_n$, we can derive 4, and could devise a loop that computes only with the dimensions (but which was not done here).

Compiler.sml: bounds check helper

```
(* 2 *)      fun checkBounds(res_reg,ind_reg,arr_reg,pos:int) =
               let val cond_reg = "_cond_"^newName()
               in [Mips.LI (cond_reg, "-1"),
                   Mips.SLT(cond_reg, cond_reg, ind_reg),
                   Mips.AND(res_reg, res_reg, cond_reg),
                   Mips.LW (cond_reg, arr_reg, makeConst (4*pos)),
                   Mips.SLT(cond_reg, ind_reg, cond_reg),
                   Mips.AND(res_reg, res_reg, cond_reg)]
               end
```

The code snippet above shows helper function `checkBounds`. `ind_reg` is a (symbolic) register that stores the value of an array index, `arr_reg` is the array's place, `pos` is the index of `ind_reg` w.r.t. the full index, e.g., in `a[1,2,3]`, index 1 appears in position 0. The function aggregates in the `res_reg` symbolic (boolean) register the condition for the current index to be within the array bounds.

The following code realises the computation (2) in the compiler. Since the code is relatively long, we refer to marks such as `(* N *)` when explaining its corresponding code. The code below computes first the values of each index (function `v` `(* 1 *)`) and checks that it does not exceed the range ($\mathsf{v}(e_i) < d_i$). The latter uses function `checkBounds`, marked `(* 2 *)`. Each value is then multiplied with the respective stride, read from memory `(* 3 *)`, except the last value (computed separately, marked `(* 1 *)` as well). In `(* 4 *)` we see that all values are added together in `tmp_reg` (the sum in the formula).
When this is done, we consider the element size in `(* 5 *)`. Int values take up 4 bytes, bool and char only one per element. Finally, we read and add the base address for the array data space in `(* 6 *)` ; resulting in the array element address referred to by the indexed expression.

Compiler.sml: compileLVal function

```
| compileLVal( vtab : VTab, Index ((n,t),inds) : LVAL, pos : Pos ) =
    ( case SymTab.lookup n vtab of
        NONE => raise Error ("unknown␣variable␣"^n, pos) )
      | SOME arr_reg =>
          let val tmp_reg   = "_tmp_" ^newName()
              val e_reg     = "_ereg_"^newName()
              val strd_reg  = "_tmp_" ^newName()
              val ind_check = "_safe_"^newName()
              val (line, _) = pos
              val el_type   = typeOfExp (LValue(Index((n,t),inds),pos))
              val arr_rank  = length inds
```

```
                (* code that computes the last index *)
(* 1 *)         val last_ind = compileExp(vtab, List.last inds, tmp_reg)
                val code_last_ind =
                     [ Mips.LI(ind_check, "1")] @ last_ind @
(* 2 *)               checkBounds(ind_check, tmp_reg, arr_reg, arr_rank-1)

                (* the flat index of [i_1,..,i_r], held in tmp_reg, is
                    i_r + i_{r-1}*stride_{r-1} + .. + i_1 * stride_1    *)
                val ind_int_pairs =
                  ListPair.zip (inds, List.tabulate(arr_rank-1, fn x => x))
                val code_flat_ind = code_last_ind @
                    foldl( fn ((e,i),code) =>
                        let val offset = 4*(i + arr_rank)
(* 1 *)                      val code_e = compileExp(vtab, e, e_reg)
                        in    code_e
(* 2 *)                   @ checkBounds(ind_check, e_reg, arr_reg, i)
                           @ [ Mips.LW (strd_reg, arr_reg, makeConst offset),
(* 3 *)                       Mips.MUL(e_reg, e_reg, strd_reg),
(* 4 *)                       Mips.ADD(tmp_reg, tmp_reg, e_reg) ]
                           @ code
                        end
                    ) [] ind_int_pairs

                (* multiply by 4 if element type is Int, stored in tmp_reg *)
                val code_byte_ind = code_flat_ind
                    @ [ Mips.LI("5", makeConst line),(* if index check fails, error *)
                        Mips.BEQ(ind_check, "0", "_IllegalArrIndexError_") ]
                    @ ( case el_type of
(* 5 *)                   BType Int => [ Mips.SLL (tmp_reg, tmp_reg, "2") ]
                        | _          => [] )

                (* finally get the data pointer and add the byte index to it *)
                val content = 4 * (2 * arr_rank - 1)
                val code = code_byte_ind @
                             [ Mips.LW (e_reg, arr_reg, makeConst content),
(* 6 *)                       Mips.ADD(e_reg, e_reg, tmp_reg) ]
                (* val () = print (concat (map Mips.pp_mips code)) *)
            in ( code, Mem e_reg )
            end
```

Task 4 has been tested by the array-intensive test programs *sort.pal* and *gol.pal*. Type error cases were tested using the TestSyntax script, typing in erroneous index expressions and calls to `new`. The possible runtime error (`IllegalArrIndexError`) was tested using *arrFail* given on page 7

Test program *arrFail.pal* (array indexing runtime errors)

```
program arrFail;
...
   // use any of these to test runtime failures for indexing
   write(a[2,0]); write(a[0-1,0]); write(a[0,2]); write(a[0,0-1]);
end;
```

5 Task 5: Call-by-value-result for procedure calls

Implementing call-by-value-result requires some modifications to the machine code generator, i.e., `Compiler.sml`, and the interpreter, i.e., `TpInterpreter.sml`. We discuss the compiler first.

After returning from a procedure call, we must copy back the final values of their arguments from registers 2-N (where N is the number of arguments). This helper function is used:

Compiler.sml (helper function)

```
  (* move args from callee registers *)
  and moveBackArgs [] vtable reg p = []
    | moveBackArgs (e::es) vtable reg p =
      let
          val uncode2 = moveBackArgs es vtable (reg+1) p
          val moveBack = case e of
                           LValue (Var (name, _), _) =>
                           let val x = SymTab.tryLookup name vtable p
                           in [Mips.MOVE (x, makeConst reg)]
                           end
                         | _ => []
      in
          uncode2 @ moveBack
      end
```

The procedure call uses this helper function to copy back arguments from registers 2-N to their original places after returning.

Compiler.sml (compileStmt function)

```
     | ProcCall ((n,_), es, p) =>
       let
           val (mvcode, maxreg) = putArgs es vtable minReg
           val unmvcode = moveBackArgs es vtable minReg p
       in mvcode @ [Mips.JAL (n, List.tabulate (maxreg, makeConst))] @ unmvcode end
(*BEFORE: mvcode @ [Mips.JAL (n, List.tabulate (maxreg, fn reg => makeConst reg))]*)
```

In the function which compiles an entire procedure, the procedure body must copy back values to the numeric registers. Also, the function/procedure exit label (`fname_exit`) must be placed before the copy-out code, inside the part passed through the register allocator.

Compiler.sml (compileF function)

```
  and compileF (isProc, fname, args, block, pos) =
          ...
          val (movePairs, vtable) = getMovePairs args [] minReg
(* BEFORE val argcode = map (fn (vname, reg) => Mips.MOVE (vname, reg)) movePairs *)
          val copy_in  = map (fn (v, reg) => Mips.MOVE (v, reg)) movePairs
          val copy_out = if isProc
                         then map (fn (v, reg) => Mips.MOVE (reg, v)) movePairs
                         else []
          val body = compileStmts block vtable (fname ^ "_exit")
          val (body1, _, maxr, spilled) =  (* call register allocator *)
(* BEFORE:     RegAlloc.registerAlloc ( argcode @ body ) *)
              RegAlloc.registerAlloc
                  (copy_in @ body @ (* copy-in, body, exit label, copy-out *)
                   [(Mips.LABEL (fname^"_exit"))] @ copy_out )
                                  ["2"] minReg maxCaller maxReg 0
                                  (* 2 contains return val*)
          val (savecode, restorecode, offset) = (* save/restore callee-saves *)
              stackSave (maxCaller+1) maxr [] [] (4*spilled)
        in  [Mips.COMMENT ("Function " ^ fname),
             Mips.LABEL fname,        (* function label *)
             Mips.SW (RA, SP, "-4"), (* save return address *)
             Mips.ADDI (SP,SP,makeConst (~4-offset))] (* move SP "up" *)
          @ savecode                  (* save callee-saves registers *)
          @ body1                     (* code for function body *)
(* BEFORE  @ [Mips.LABEL (fname^"_exit")] (* exit label *)
               previously, the _exit label (jump target for the return
               statement) was here. With call-by-value-result, we must
               not jump over the copy-out code we created, so it is now
               inside the "body1" code block, see above *)
          @ restorecode               (* restore callee-saves registers *)
          @ [Mips.ADDI (SP,SP,makeConst (4+offset))] (* move SP "down" *)
          @ [Mips.LW (RA, SP, "-4"),  (* restore return addr *)
             Mips.JR (RA, [])]        (* return *)
```

```
        end
```

The interpreter holds values for variables in a table during execution. Each function and procedure has its own symbol table. We need to copy the new values back from the inner vtable to the outer vtable (with the helper function `updateOuterVtable`).

TpInterpret.sml

```
and callFun ((rtp : Type option, fid : string, fargs : Dec list,
              body : StmtBlock, pdcl : Pos ),
             aargs : Value list, aexps : Exp list,
             vtab, ftab, pcall : Pos ) : Value option =
    case fid of
          ...
        | other    =>
          let val new_vtab = bindTypeIds(fargs, aargs, fid, pdcl, pcall)
              val res  = execBlock( body, new_vtab, ftab )
          in case (rtp, res) of
                (NONE  , _     ) => (map (updateOuterVtable vtab new_vtab)
                                         (ListPair.zip (aexps, fargs)); NONE)
              | ...
          end
(* ... *)
and updateOuterVtable vtabOuter vtabInner (out_exp, in_arg) =
    case (out_exp, in_arg) of
        (LValue (Var (name, _), _), Dec ((faid, _), _)) =>
        (case (SymTab.lookup name vtabOuter, SymTab.lookup faid vtabInner) of
             (SOME v_out, SOME v_in) => v_out := !v_in
           | _ => ())
      | _ => ()
```

This functionality has been tested with the delivered test programs, and produced the expected results (swapping parameters, computing correct results).

Even with the exit label in the wrong place, all provided tests succeeded. To trigger the potential bug related to the exit label, an additional program was written, which calls another (register-heavy) function `eatregs`[1] inside the swap procedure, and uses a return statement at the end of `swap`.

Test program *procSwapEatRegs.pal*

```
procedure swap(x : int; y : int)
var
   t : int;
begin
   t := x + eatregs(1,2,3,4,5,6,7,8,9,10);
   x := y;
   y := t - eatregs(1,2,3,4,5,6,7,8,9,10);
   return;
end;
```

`swap` should still swap the arguments, but the additional function call will overwrite values in registers so they are not there accidentally. With this swap function, the test only succeeds when the exit label is in the correct place.

That's all, folks!

[1] Taken from the test program `regs.pal` – which btw contained a mistake.