Block expressions and unsharing

Mike Spivey, December 2021

This year's Christmas Assignment asks for an implementation of *block expressions*, which allow statements to be embedded in expressions. This construct, and especially the keywords `valof` and `resultis`, date back to the language BCPL, developed by Martin Richards in the late 1960's as a simplified version of Christopher Strachey's language CPL. It also asks for a modification to the common subexpression elimination (CSE) pass of the course compiler to make it less over-enthusiastic about computing into temporaries certain operations that can be recomputed later at no cost, typically as part of an addressing mode.

1 Block expressions

This task is to be carried out on a compiler that translates PICOPASCAL programs into postfix code for the Keiko machine: conveniently so, because it allows the results of block expressions to be returned by 'leaving them on the stack' without any complications about register allocation. As usual, we can add the new feature by following through each phase of the compiler.

In the lexical analyser, we need only add the two new keywords `valof` and `resultis`. In the parser, we need two new token types, *VALOF* and *RESULTIS*, and two new productions.

```
stmt : ...
  | RESULTIS expr                    { Resultis $2 }

expr : ...
  | VALOF stmts END                  { make_expr (ValofExpr $2) }
```

Two new constructors need to be added to the abstract syntax, making the types *stmt* and *expr* mutually recursive if they were not before.

```
type stmt = ...
  | Resultis of expr

and expr = ...
  | ValofExpr of stmt
```

I omit here the changes needed in the code for dumping abstract syntax trees in readable form.

In the semantic analyser, things get more interesting, because not only are the functions for checking expressions and statements now mutually recursive, but also every statement must be checked in a context that records whether the statement is part of a `valof` expression, and if so what type that expression has. For simplicity, we require each `valof` expression to contain at least one `resultis` statement; all such statements must be consistent in their types, and their common type becomes the type of the enclosing `valof` expression.

To avoid passing an additional parameter everywhere, it's convenient to introduce a global variable *valof_cxt* with the slightly scary type *ptype option ref option ref*.

```
    let valof_cxt = ref None
```

This variable can take several kinds of value:

- *None* means that we are outside any `valof` expression: `resultis` statements are forbidden.
- *Some* (*ref None*) means that we are inside a `valof` expression whose type is not yet known. Any `resultis` statement is allowed, and the type of its argument becomes the type of the enclosing expression.
- *Some* (*ref t*) means that we are inside a `valof` expression of type *t*: a `resultis` statement is allowed, provided its argument also has type *t*.

These conventions allow us to implement checking of `resultis` statements, detecting each kind of error that may occur.

```
    [let rec] check_stmt s env =
      err_line := s.s_line;
      match s.s_guts with . . .
        | Resultis e →
            begin match !valof_cxt with
                None →
                  sem_error "resultis statement not in valof expression" [ ]
              | Some tr →
                  let t = check_expr e env in
                  begin match !tr with
                      None →
                        (∗ First occurrence of a resultis for this valof ∗)
                        if not (scalar t) then
                          sem_error "resultis statement needs a scalar" [ ];
                        tr := Some t
                    | Some t0 →
                        (∗ Second or later – check for consistency ∗)
                        if not (same_type t0 t) then
                          sem_error "type error in resultis statement" [ ]
                  end
            end
```

Because of the global variable, checking a `valof` expression requires a little dance to preserve the old value of *valof_cxt* across the recursive call of *check_stmt*.

```
let rec check_expr e env =
  let t = expr_type e env in
  e.e_type ← t; t

(* expr_type – return type of expression *)
and expr_type e env =
  match e.e_guts with ...
    | ValofExpr s →
        let tr = ref None in
        let $cxt_0$ = !valof_cxt in
        valof_cxt := Some tr;
        check_stmt s env;
        valof_cxt := $cxt_0$;
        begin match !tr with
            Some t → t
          | None →
              sem_error "valof without resultis" [ ]
        end
```

We can check after analysing the body of the expression that at least one `resultis` expression has been seen, and use its type as the type of the expression.

For code generation, we translate a `resultis` statement into code that pushes the result on the stack, then branches to a label provided as part of the context. Instead of introducing a global variable for this label, let's take the opposite approach and pass it as an argument to each invocation of *gen_stmt*. I will suppress the many changes needed in places where the parameter is passed on unchanged in a recursive call.

```
[let rec] gen_stmt s reslab =
  let code =
    match s.s_guts with
        Skip → NOP
      | Seq ss → SEQ (List.map (fun s → gen_stmt s reslab) ss)
      | ...
      | Resultis e →
          SEQ [gen_expr e; JUMP reslab]
```

`Valof` expressions are translated in a complementary way by inventing a label and using it as the context for translating the body.

```
[let rec] gen_expr e =
  match e.e_value with
      Some v → CONST v
    | None →
        begin
          match e.e_guts with ...
            | ValofExpr s →
                let reslab = label ( ) in
                SEQ [gen_stmt s reslab; ERETURN s.s_line; LABEL reslab]
        end
```

The *EReturn* instruction signals a runtime error if the body terminates without encountering a `resultis` statement.

Decent test cases for the new construct might include the example given in the assignment instructions, a recursive definition of factorial that uses the BCPL `valof` / `resultis` style, and at least one instance of nested `valof` expressions. A more thorough test suite would also check that each of the programmed error messages could be issued in appropriate circumstances.

It's worth mentioning that this treatment of `valof` blocks works only for a language without other non-local exits: for example, if there were a loop-with-exit construct too, then occurrences of `exit` within a `valof` block would be in danger of leaving the stack in an inconsistent state.

2 Reversing excessive sharing

The second task is to reverse excessive sharing of common subexpressions in a compiler from picoPascal to ARM machine code. In several cases, most associated with addressing modes, the CSE pass is likely to compute values into a register that could later be recomputed at no cost, making the program longer for no reason.

The assignment instructions give less of a clue how this improvement can be accomplished, but a fairly good job can be done with changes confined to the CSE pass implemented in `share.ml`. The only limitation is that, unaware of the details of instruction selection, the CSE pass cannot know precisely which calculations can be folded into subsequent instructions for free.

Our approach will be to run the value numbering algorithm as usual to identify maximal common subexpressions, then have an additional pass that inspects the resulting DAG for expressions that are shared and should not be. For example, if the *Offset* node in $\langle Loadw, \langle Offset, t_1, t_2\rangle\rangle$ is shared with a later instance of $\langle Storew, t_3, \langle Offset, t_1, t_2\rangle\rangle$, then the inspection pass will duplicate it, turning one shared *Offset* node into two copies without sharing, so that it is no longer treated as a common subexpression. Duplicating the *Offset* node will lead to roots of t_1 and t_2 becoming shared, and we will need to be careful to treat them appropriately when we implement this idea.

The first change we need is to make the list of operands of a DAG node mutable, so that we can modify the DAG after it has been created. We also add a boolean flag to say whether the node has been visited during the inspection pass. Inspection will be an idempotent operation, so there is no harm in inspecting nodes twice, but no benefit either.

```
(* dagnode – node in DAG representation of an expression *)
type dagnode =
  { g_serial : int;                 (* Serial number *)
    g_op : inst;                    (* Operator *)
    mutable g_rands : dagnode list;   (* Operands *)
    mutable g_refct : int;          (* Reference count *)
    mutable g_temp : int;           (* Temp, or -1 if none *)
    mutable g_inspected : bool }
```

When DAG nodes are created, the *g_inspected* field should be initialised to **false**.

For the next change, in the function *visit*, nodes with operator *Local n* are always treated as trivial, so that they are not shared even if they have a reference count greater than one. We can treat them more sensitively, sharing them only if it looks like the address computation cannot be folded into an addressing mode. Consequently the pattern *LOCAL* _ is deleted from the trivial case in the following definition.

```
let rec visit g top =
  match g.g_op with
      TEMP _ | REGVAR _ | CONST _ →
        build g                    (* Trivial *)
    | ...
```

Now we reach the substance of the proposed change. In the main program of the CSE pass, we insert calls to a function *inspect* between the phase that builds a DAG from the input trees and the phase that turns the DAG back into trees with temps. The inspection process is run from each root of the DAG.

```
let traverse ts =
  reset ();
  (* Convert the trees to a list of roots in a DAG *)
  let gs = List.map make_dag ts in
  (* Reverse excessive sharing *)
  List.iter inspect gs;
  (* Then convert the DAG roots back into trees *)
  canon ⟨SEQ, @(List.map (fun g → visit g true) gs)⟩
```

The function *inspect* is mutually recursive with another function *unshare*, for reasons that will be discussed in a minute. If it is applied to a node that has not so far been inspected, then the fragment of DAG rooted at that node is matched with various rules that aim to reduce the amount of excessive sharing. The default, if none of these rules match, is to inspect the children of the node recursively. This function is expressed more cleanly if we use 'DAG patterns' written in the notation #⟨*OFFSET*, g_1, g_2⟩ with an intention that should be clear.

```
[let rec] inspect g =
  if not g.g_inspected then begin
    g.g_inspected ← true;
    match g with
        ...                        (* See below *)
      | #⟨op, @rands⟩ → List.iter inspect rands
  end
```

Several rules are written in the place marked ... in the function. First, if a *LOADC* or *LOADW* node has an *OFFSET* node as child, then we can remove any sharing of the *OFFSET*, expecting it to be computed for free as part of the *reg* + *reg* addressing mode.

```
#⟨(LOADC | LOADW), #⟨OFFSET, _, _⟩ as g1⟩ →
  inspect g1; g.g_rands ← [unshare g1]
```

The subroutine *unshare* takes a node and, if has a reference count greater than one, duplicates it, returning a copy whose reference count is exactly one. It leaves unshared nodes unchanged. We can write another, similar, rule to deal with *STOREC* and *STOREW* nodes.

```
    $\#\langle(\textit{STOREC} \mid \textit{STOREW}), g_1, \#\langle\textit{OFFSET}, \_, \_\rangle$ as $g_2, g_3\rangle \rightarrow$
      $\textit{inspect}\ g_1$; $\textit{inspect}\ g_2$; $g.\textit{g\_rands} \leftarrow [g_1; \textit{unshare}\ g_2; g_3]$
```

There's no need to inspect the speculative load g_3. Two more rules deal with loads and stores of locals, leaving them to be shared only if the address of the same local is actually needed in a register at least twice.

```
    $\#\langle(\textit{LOADC} \mid \textit{LOADW}), \#\langle\textit{LOCAL}\ \_\rangle$ as $g_1\rangle \rightarrow$
      $g.\textit{g\_rands} \leftarrow [\textit{unshare}\ g_1]$
  | $\#\langle(\textit{STOREC} \mid \textit{STOREW}), g_1, \#\langle\textit{LOCAL}\ \_\rangle$ as $g_2, g_3\rangle \rightarrow$
      $\textit{inspect}\ g_1$; $g.\textit{g\_rands} \leftarrow [g_1; \textit{unshare}\ g_2; g_3]$
```

A final rule, appropriate for the ARM, allows shifts to be folded into addressing offsets.

```
    $\#\langle\textit{OFFSET}, g_1, \#\langle\textit{BINOP Lsl}, \_, \#\langle\textit{CONST}\ \_\rangle\rangle$ as $g_2\rangle \rightarrow$
      $\textit{inspect}\ g_1$; $\textit{inspect}\ g_2$; $g.\textit{g\_rands} \leftarrow [g_1; \textit{unshare}\ g_2]$
```

Now for a word about the graph patterns written in these rules; such patterns are already used in the existing code. The idea is that, just as the tree pattern,

$\langle\textit{OFFSET}, t_1, \langle\textit{BINOP Lsl}, t_2, t_3\rangle\rangle$,

stands for the slightly cumbersome pattern,

$\textit{Node}\,(\textit{OFFSET}, [t_1; \textit{Node}(\textit{BINOP Lsl}, [t_2; t_3])])$,

so the corresponding graph pattern stands for the even more cumbersome notation,

```
  { $\textit{g\_op} = \textit{OFFSET}$;
    $\textit{g\_rands} = [g_1; \{ \textit{g\_op} = \textit{BINOP Lsl}; \textit{g\_rands} = [g_2; g_3] \}] \}$.
```

The *nodexp* tool performs this expansion for both tree and graph patterns.

What remains is to provide the function *unshare* that is applied to possibly-overshared nodes. If the argument g is shared, then an unshared copy of it is created and returned, with the reference count of g adjusted appropriately.

```
[and] unshare g =
  if $g.\textit{g\_refct} = 1$ then g else begin
    $g.\textit{g\_refct} \leftarrow g.\textit{g\_refct} - 1$;
    let $g_1$ = newnode g.g_op g.g_rands in
    $g_1.\textit{g\_refct} \leftarrow 1$; inspect $g_1$; $g_1$
  end
```

There is a recursive call of *inspect* on the new node g_1 that at first seems superfluous, but is needed in cases like `a[i] := a[i]+1`, which gives rise to a tree,

$\langle\textit{LOADW}, \langle\textit{OFFSET}, t_1, \langle\textit{LSL}, t_2, \langle\textit{CONST}\ 2\rangle\rangle\rangle$,

with the *OFFSET* node inappropriately shared. Duplicating the *OFFSET* node makes the *LSL* node shared, and the rule for folding shifts into offset calculation then applies. The call to *inspect* gives it chance to fire.

To test the improvement, it is enough to run the compiler on the standard set of test cases and compare the object code. The improvement is particularly striking in the case of test case `swap.p`, which contains the following procedure.

```
proc swap(i, j: integer);
```

```
  var t: integer;
begin
  t := a[i];
  a[i] := a[j];
  a[j] := t
end;
```

Before the improvement, the compiler would treat the addresses of `a[i]` and `a[j]` as common subexpressions, requiring several additional instructions and failing to exploit the ARM addressing modes. Following our improvements, the code generated for the body of `swap` is as follows.

```
@  t := a[i];
       ldr r5, =_a
       ldr r6, [fp, #40]
       ldr r4, [r5, r6, LSL #2]
@  a[i] := a[j];
       ldr r7, [fp, #44]
       ldr r0, [r5, r7, LSL #2]
       str r0, [r5, r6, LSL #2]
@  a[j] := t
       str r4, [r5, r7, LSL #2]
```

This code would be hard to improve upon.

3 Listings

Appended are *diff* listings for the two tasks, followed by three test cases for the first task.

```
--- ../../labs/ppc4/check.ml    2023-10-08 15:18:06.839584910 +0100
+++ check.ml    2023-10-08 16:07:24.500542698 +0100
@@ -10,9 +10,10 @@

 (* EXPRESSIONS *)

-(* Two global variables to save passing parameters everywhere *)
+(* Three global variables to save passing parameters everywhere *)
 let level = ref 0
 let return_type = ref voidtype
+let valof_cxt = ref None

 let err_line = ref 1

@@ -163,6 +164,22 @@
         let t = check_binop w (check_expr e1 env) (check_expr e2 env) in
         e.e_value <- try_binop w e1.e_value e2.e_value;
         t
+    | ValofExpr s ->
+        (* For simplicity, we require each valof expression to contain
+           at least one resultis statement.  The valof context is None
+           outside any valof expression, Some (ref None) when no resultis
+           has been seen yet, Some (ref (Some t)) when the valof is known
+           to have type t. *)
+        let tr = ref None in
+        let cxt0 = !valof_cxt in
+        valof_cxt := Some tr;
+        check_stmt s env;
+        valof_cxt := cxt0;
+        begin match !tr with
+            Some t -> t
+          | None ->
+              sem_error "valof without resultis" []
+        end

 (* |check_funcall| -- check a function or procedure call *)
 and check_funcall f args env v =
@@ -256,7 +273,7 @@
     | _ -> ()

 (* |check_const| -- check an expression with constant value *)
-let check_const e env =
+and check_const e env =
   let t = check_expr e env in
   match e.e_value with
       Some v -> (t, v)
@@ -266,7 +283,7 @@
 (* STATEMENTS *)

 (* check_dupcases -- check for duplicate case labels *)
-let check_dupcases vs =
+and check_dupcases vs =
   let rec chk =
     function
        [] | [_] -> ()
@@ -276,7 +293,7 @@
   chk (List.sort compare vs)

 (* |check_stmt| -- check and annotate a statement *)
-let rec check_stmt s env =
+and check_stmt s env =
   err_line := s.s_line;
   match s.s_guts with
       Skip -> ()
@@ -307,6 +324,23 @@
                  if not (same_type !return_type voidtype) then
                    sem_error "function must return a result" []
         end
+    | Resultis e ->
+        begin match !valof_cxt with
+            None -> sem_error "resultis statement not in valof expression" []
+          | Some tr ->
+              let t = check_expr e env in
+              begin match !tr with
+                  None ->
+                    (* First occurrence of a resultis for this valof *)
+                    if not (scalar t) then
+                      sem_error "resultis statement needs a scalar" [];
+                    tr := Some t
+                | Some t0 ->
+                    (* Second or later -- check for consistency *)
+                    if not (same_type t0 t) then
+                      sem_error "type mismatch in resultis statement" []
+              end
+        end
     | IfStmt (cond, thenpt, elsept) ->
         let ct = check_expr cond env in
         if not (same_type ct boolean) then
--- ../../labs/ppc4/kgen.ml     2023-10-08 15:18:06.827584992 +0100
+++ kgen.ml     2023-10-08 16:07:24.500542698 +0100
@@ -78,6 +78,18 @@
           address d; load_addr]
     | _ -> failwith "missing closure"

+let gen_jtable tab0 deflab =
+  if tab0 = [] then JUMP deflab else begin
+    let table = List.sort (fun (v1, l1) (v2, l2) -> compare v1 v2) tab0 in
+    let lob = fst (List.hd table) in
+    let rec tab u qs =
+      match qs with
+          [] -> []
+        | (v, l) :: rs ->
+            if u = v then l :: tab (v+1) rs else deflab :: tab (u+1) qs in
+    SEQ [CONST lob; BINOP Minus; JCASE (tab lob table); JUMP deflab]
+  end
+
 (* |gen_addr| -- code for the address of a variable *)
 let rec gen_addr v =
   match v.e_guts with
@@ -134,6 +146,9 @@
                  SEQ [gen_expr e1; gen_expr e2; BINOP w]
             | FuncCall (p, args) ->
                  gen_call p args
+             | ValofExpr s ->
+                 let reslab = label () in
+                 SEQ [gen_stmt s reslab; ERETURN s.s_line; LABEL reslab]
             | _ -> failwith "gen_expr"
         end

@@ -229,24 +244,12 @@
   SEQ [gen_cond l1 l2 test;
     LABEL l1; CONST 1; JUMP l3; LABEL l2; CONST 0; LABEL l3]

-let gen_jtable tab0 deflab =
-  if tab0 = [] then JUMP deflab else begin
-    let table = List.sort (fun (v1, l1) (v2, l2) -> compare v1 v2) tab0 in
-    let lob = fst (List.hd table) in
-    let rec tab u qs =
-      match qs with
-          [] -> []
-        | (v, l) :: rs ->
-            if u = v then l :: tab (v+1) rs else deflab :: tab (u+1) qs in
-    SEQ [CONST lob; BINOP Minus; JCASE (tab lob table); JUMP deflab]
-  end
-
 (* |gen_stmt| -- generate code for a statement *)
-let rec gen_stmt s =
+and gen_stmt s reslab =
   let code =
     match s.s_guts with
        Skip -> NOP
-      | Seq ss -> SEQ (List.map gen_stmt ss)
+      | Seq ss -> SEQ (List.map (fun s -> gen_stmt s reslab) ss)
      | Assign (v, e) ->
          if scalar v.e_type || is_pointer v.e_type then
            SEQ [gen_expr e; gen_addr v; STORE (size_of v.e_type)]
@@ -261,25 +264,27 @@
                Some e -> SEQ [gen_expr e; RETURN 1]
              | None -> SEQ [RETURN 0]
          end
+      | Resultis e ->
+          SEQ [gen_expr e; JUMP reslab]
      | IfStmt (test, thenpt, elsept) ->
          let lab1 = label () and lab2 = label () and lab3 = label () in
          SEQ [gen_cond lab1 lab2 test;
-            LABEL lab1; gen_stmt thenpt; JUMP lab3;
-            LABEL lab2; gen_stmt elsept; LABEL lab3]
+            LABEL lab1; gen_stmt thenpt reslab; JUMP lab3;
+            LABEL lab2; gen_stmt elsept reslab; LABEL lab3]
      | WhileStmt (test, body) ->
        let lab1 = label () and lab2 = label () and lab3 = label () in
-          SEQ [JUMP lab2; LABEL lab1; gen_stmt body;
+          SEQ [JUMP lab2; LABEL lab1; gen_stmt body reslab;
            LABEL lab2; gen_cond lab1 lab3 test; LABEL lab3]
      | RepeatStmt (body, test) ->
          let lab1 = label () and lab2 = label () in
-          SEQ [LABEL lab1; gen_stmt body;
+          SEQ [LABEL lab1; gen_stmt body reslab;
            gen_cond lab2 lab1 test; LABEL lab2]
      | ForStmt (var, lo, hi, body) ->
          (* For simplicity, this code re-evaluates hi on each iteration *)
          let l1 = label () and l2 = label () in
          let s = int_rep.r_size in
          SEQ [gen_expr lo; gen_addr var; STORE s; JUMP l2;
-            LABEL l1; gen_stmt body;
+            LABEL l1; gen_stmt body reslab;
            gen_expr var; CONST 1; BINOP Plus; gen_addr var; STORE s;
            LABEL l2; gen_expr var; gen_expr hi; JUMPC (Leq, l1)]
      | CaseStmt (sel, arms, deflt) ->
@@ -288,10 +293,10 @@
          let get_val (v, body) = get_value v in
          let table = List.combine (List.map get_val arms) labs in
          let gen_case lab (v, body) =
-            SEQ [LABEL lab; gen_stmt body; JUMP donelab] in
+            SEQ [LABEL lab; gen_stmt body reslab; JUMP donelab] in
          SEQ [gen_expr sel; gen_jtable table deflab;
```

```
             SEQ (List.map2 gen_case labs arms);
-            LABEL deflab; gen_stmt deflt;
+            LABEL deflab; gen_stmt deflt reslab;
             LABEL donelab] in
   SEQ [if s.s_line <> 0 then LINE s.s_line else NOP; code]

@@ -300,7 +305,7 @@
   printf "FUNC $ $\n" [fStr lab; fNum fsize];
   level := lev+1;
   let code =
-    SEQ [gen_stmt body;
+    SEQ [gen_stmt body nolab;
       (if rtype.t_rep.r_size = 0 then RETURN 0 else ERETURN 0)] in
   Keiko.output (if !optflag then Peepopt.optimise code else code);
   printf "END\n\n" []
--- ../../labs/ppc4/lexer.mll   2023-10-08 15:18:06.839584910 +0100
+++ lexer.mll   2023-10-08 16:07:24.500542698 +0100
@@ -19,6 +19,7 @@
       ("pointer", POINTER); ("nil", NIL);
       ("repeat", REPEAT); ("until", UNTIL); ("for", FOR);
       ("elsif", ELSIF); ("case", CASE);
+      ("valof", VALOF); ("resultis", RESULTIS);
       ("and", MULOP And); ("div", MULOP Div); ("or", ADDOP Or);
       ("not", NOT); ("mod", MULOP Mod) ]

--- ../../labs/ppc4/parser.mly  2023-10-08 15:18:06.827584992 +0100
+++ parser.mly  2023-10-08 16:07:24.500542698 +0100
@@ -23,6 +23,7 @@
 %token                 PROC RECORD RETURN THEN TO TYPE
 %token                 VAR WHILE NOT POINTER NIL
 %token                 REPEAT UNTIL FOR ELSIF CASE
+%token                 VALOF RESULTIS

 /* operator priorities */
 %left                  RELOP EQUAL
@@ -119,6 +120,7 @@
   | variable ASSIGN expr              { Assign ($1, $3) }
   | name actuals                      { ProcCall ($1, $2) }
   | RETURN expr_opt                   { Return $2 }
+  | RESULTIS expr                     { Resultis $2 }
   | IF expr THEN stmts elses END      { IfStmt ($2, $4, $5) }
   | WHILE expr DO stmts END           { WhileStmt ($2, $4) }
   | REPEAT stmts UNTIL expr           { RepeatStmt ($2, $4) }
@@ -166,6 +168,7 @@
   | expr MINUS expr                   { make_expr (Binop (Minus, $1, $3)) }
   | expr RELOP expr                   { make_expr (Binop ($2, $1, $3)) }
   | expr EQUAL expr                   { make_expr (Binop (Eq, $1, $3)) }
+  | VALOF stmts END                   { make_expr (ValofExpr $2) }
   | LPAR expr RPAR                    { $2 } ;

 actuals :
--- ../../labs/ppc4/peepopt.ml  2023-10-08 15:18:06.843584883 +0100
+++ peepopt.ml  2023-10-08 16:07:24.500542698 +0100
@@ -107,6 +107,9 @@

     | CONST 0 :: JUMPC (w, lab) :: _ ->
         replace 2 [JUMPCZ (w, lab)]
+    | BINOP ((Lt|Gt|Leq|Geq|Eq|Neq) as w)
+          :: JUMPCZ ((Eq|Neq) as w', lab) :: _ ->
+        replace 2 [JUMPC ((if w' = Neq then w else opposite w), lab)]

     | LINE n :: LABEL a :: _ ->
```

```
         replace 2 [LABEL a; LINE n]
@@ -118,6 +121,8 @@
         equate a b; replace 2 [JUMP b]
     | JUMPC (w, a) :: JUMP b :: LABEL c :: _ when same_lab a c ->
         replace 2 [JUMPC (opposite w, b)]
+    | JUMPCZ (w, a) :: JUMP b :: LABEL c :: _ when same_lab a c ->
+        replace 2 [JUMPCZ (opposite w, b)]
     | JUMP a :: LABEL b :: _ when same_lab a b ->
        replace 1 []
     | JUMP a :: LABEL b :: _ ->
--- ../../labs/ppc4/tree.ml     2023-10-08 15:18:06.855584802 +0100
+++ tree.ml     2023-10-08 16:07:24.500542698 +0100
@@ -35,6 +35,7 @@
   | Assign of expr * expr
   | ProcCall of name * expr list
   | Return of expr option
+  | Resultis of expr
   | IfStmt of expr * stmt * stmt
   | WhileStmt of expr * stmt
   | RepeatStmt of stmt * expr
@@ -57,6 +58,7 @@
   | FuncCall of name * expr list
   | Monop of op * expr
   | Binop of op * expr * expr
+  | ValofExpr of stmt

 and typexpr =
     TypeName of name
@@ -140,6 +142,7 @@
     | ProcCall (p, aps) -> fMeta "(CALL $$)" [fName p; fTail(fExpr) aps]
     | Return (Some e) -> fMeta "(RETURN $)" [fExpr e]
     | Return None -> fStr "(RETURN)"
+    | Resultis e -> fMeta "(RESULTIS $)" [fExpr e]
     | IfStmt (test, thenpt, elsept) ->
        fMeta "(IF $ $ $)" [fExpr test; fStmt thenpt; fStmt elsept]
     | WhileStmt (test, body) ->
@@ -167,6 +170,8 @@
        fMeta "($ $)" [fOp w; fExpr e1]
     | Binop (w, e1, e2) ->
        fMeta "($ $ $)" [fOp w; fExpr e1; fExpr e2]
+    | ValofExpr s ->
+        fMeta "(VALOF $)" [fStmt s]

 and fType =
   function
--- ../../labs/ppc4/tree.mli    2023-10-08 15:18:06.839584910 +0100
+++ tree.mli    2023-10-08 16:07:24.500542698 +0100
@@ -50,6 +50,7 @@
   | Assign of expr * expr
   | ProcCall of name * expr list
   | Return of expr option
+  | Resultis of expr
   | IfStmt of expr * stmt * stmt
   | WhileStmt of expr * stmt
   | RepeatStmt of stmt * expr
@@ -72,6 +73,7 @@
   | FuncCall of name * expr list
   | Monop of op * expr
   | Binop of op * expr * expr
+  | ValofExpr of stmt

 and typexpr =
      TypeName of name
```

```
--- ../lab4/share.ml    2021-07-30 11:47:56.995688581 +0100
+++ share.ml    2022-04-29 08:38:34.895180219 +0100
@@ -11,9 +11,10 @@
 type dagnode =
   { g_serial: int;                      (* Serial number *)
     g_op: inst;                                 (* Operator *)
-    g_rands: dagnode list;               (* Operands *)
+    mutable g_rands: dagnode list;       (* Operands *)
     mutable g_refct: int;               (* Reference count *)
-    mutable g_temp: int }               (* Temp, or -1 if none *)
+    mutable g_temp: int;                (* Temp, or -1 if none *)
+    mutable g_inspected: bool }

 let fNode g = fMeta "@$($)" [fNum g.g_serial; fNum g.g_refct]

@@ -34,7 +35,7 @@
     printf "@ Node @$ = $ [$]\n"
       [fNum !node_count; fInst op; fList(fNode) rands];
   { g_serial = !node_count; g_op = op; g_rands = rands;
-    g_refct = 0; g_temp = -1 }
+    g_refct = 0; g_temp = -1; g_inspected = false }

 (* |node| -- create a new node or share an existing one *)
 let node op rands =
@@ -137,7 +138,7 @@
 (* |visit| -- convert dag to tree, sharing the root if worthwhile *)
 let rec visit g top =
   match g.g_op with
-      TEMP _ | LOCAL _ | REGVAR _ | CONST _ ->
+      TEMP _ | REGVAR _ | CONST _ ->
         build g (* Trivial *)
     | GLOBAL _  when not Mach.share_globals ->
         build g
@@ -184,9 +185,42 @@
           <AFTER, <DEFTEMP n, d'>, <TEMP n>>
   end

+(* unshare -- duplicate a node if it is shared but should be recomputed *)
+let rec unshare g =
+  if g.g_refct = 1 then g else begin
+    if debug then printf "@ Unsharing $\n" [fNode g];
+    g.g_refct <- g.g_refct-1;
+    let g1 = newnode g.g_op g.g_rands in
+    g1.g_refct <- 1; inspect g1; g1
+  end
+
+(* inspect -- find shared nodes that can be recomputed at no cost *)
+and inspect g =
+  if not g.g_inspected then begin
+    g.g_inspected <- true;
+    if debug then printf "@ Inspecting $ : $\n" [fNode g; fInst g.g_op];
+    match g with
+        #<(LOADC|LOADW), #<LOCAL _> as g1> ->
+          g.g_rands <- [unshare g1]
+      | #<(STOREC|STOREW), g1, #<LOCAL _> as g2, g3> ->
+          inspect g1; g.g_rands <- [g1; unshare g2; g3]
+
+      | #<(LOADC|LOADW), #<OFFSET, _, _> as g1> ->
+          inspect g1; g.g_rands <- [unshare g1]
+      | #<(STOREC|STOREW), g1, #<OFFSET, _, _> as g2, g3> ->
+          inspect g1; inspect g2; g.g_rands <- [g1; unshare g2; g3]
+
+       | #<OFFSET, g1, #<BINOP Lsl, _, #<CONST _>> as g2> ->
+           inspect g1; inspect g2; g.g_rands <- [g1; unshare g2]
+
+       | #<op, @rands> -> List.iter inspect rands
+   end
+
 let traverse ts =
   reset ();
   (* Convert the trees to a list of roots in a DAG *)
   let gs = List.map make_dag ts in
+  (* Reverse excessive sharing *)
+  List.iter inspect gs;
   (* Then convert the DAG roots back into trees *)
   canon <SEQ, @(List.map (fun g -> visit g true) gs)>
```

```
var b: boolean;
var i: integer;

begin
  i:=0;
  b:= valof while true do
    if i > 2021 then resultis (i <= 2021) else i:=i+1 end end
  end;
  (* At this point i=2022 and b=false. *)
  print_num(valof i:=i-1; if (i=2021) and not b then resultis i+1 end end);
  newline();
  print_num(i); newline()
  (* The program should print 2022 and 2021. *)
end.

(*<<
2022
2021
>>*)

(*[[
MODULE Main 0 0
IMPORT Lib 0
ENDHDR

FUNC MAIN 0
!   i:=0;
CONST 0
STGW _i
LABEL L2
!     if i > 2021 then resultis (i <= 2021) else i:=i+1 end end
LDGW _i
CONST 2021
JLEQ L6
LDGW _i
CONST 2021
LEQ
JUMP L1
LABEL L6
LDGW _i
CONST 1
PLUS
STGW _i
JUMP L2
LABEL L1
STGC _b
!   print_num(valof i:=i-1; if (i=2021) and not b then resultis i+1 end end);
LDGW _i
CONST 1
MINUS
STGW _i
LDGW _i
CONST 2021
JNEQ L11
LDGC _b
JNEQZ L11
LDGW _i
CONST 1
PLUS
JUMP L8
LABEL L11
ERROR E_RETURN 0
LABEL L8
CONST 0
GLOBAL lib.print_num
PCALL 1
!   newline();
CONST 0
GLOBAL lib.newline
PCALL 0
!   print_num(i); newline()
LDGW _i
CONST 0
GLOBAL lib.print_num
PCALL 1
CONST 0
GLOBAL lib.newline
PCALL 0
RETURN
END

GLOVAR _b 1
GLOVAR _i 4
! End
]]*)
```

```
(* fac.p *)

proc fac(n: integer): integer;
begin return valof
  if n = 0 then
    resultis 1
  else
    resultis n * fac(n-1)
  end
end end;

var f: integer;

begin
  f := fac(10);
  print_num(f);
  newline()
end.

(*<<
3628800
>>*)

(*[[
MODULE Main 0 0
IMPORT Lib 0
ENDHDR

FUNC _fac 0
!   if n = 0 then
LDLW 16
JNEQZ L3
!     resultis 1
CONST 1
JUMP L1
LABEL L3
!     resultis n * fac(n-1)
LDLW 16
LDLW 16
CONST 1
MINUS
CONST 0
GLOBAL _fac
PCALLW 1
TIMES
LABEL L1
RETURN
END

FUNC MAIN 0
!   f := fac(10);
CONST 10
CONST 0
GLOBAL _fac
PCALLW 1
STGW _f
!   print_num(f);
LDGW _f
CONST 0
GLOBAL lib.print_num
PCALL 1
!   newline()
CONST 0
GLOBAL lib.newline
PCALL 0
RETURN
END

GLOVAR _f 4
! End
]]*)
```

```
var x, y: integer;

begin
  print_num(valof while true do x := x+3;
    if valof repeat y := y+1 until y > 9; resultis x > y end then
      resultis x
    end end end);
  newline()
end.

(*<<
15
>>*)

(*[[
MODULE Main 0 0
IMPORT Lib 0
ENDHDR

FUNC MAIN 0
LABEL L2
!   print_num(valof while true do x := x+3;
LDGW _x
CONST 3
PLUS
STGW _x
LABEL L9
!     if valof repeat y := y+1 until y > 9; resultis x > y end then
LDGW _y
CONST 1
PLUS
STGW _y
LDGW _y
CONST 9
JLEQ L9
LDGW _x
LDGW _y
JLEQ L2
!       resultis x
LDGW _x
CONST 0
GLOBAL lib.print_num
PCALL 1
!   newline()
CONST 0
GLOBAL lib.newline
PCALL 0
RETURN
END

GLOVAR _x 4
GLOVAR _y 4
! End
]]*)
```