Implementing array iterators

Mike Spivey, December 2019

The first part of this year's assignment asks for an implementation of a new kind of `for` loop that iterates over an array. The statement

> `for` *var* `in` *array* `do` *body* `end`

should execute *body* repeatedly, making *var* denote successive elements of the array. For example, a subroutine *sum* that adds up the elements of an array can be defined by

```
type vec = array 10 of integer;

proc sum(var a: vec): integer;
  var s: integer;
begin
  s := 0;
  for x in a do s := s + x end;
  return s
end;
```

In the body of the loop, the controlled variable actually denotes an element of the array, so that we could set all elements to zero with the loop,

> `for x in a do x := 0 end.`

The problem statement gives a hint that the name behaves like a parameter passed by reference: and indeed we could rewrite the *sum* example using an ordinary `for` loop whose body is a call to a local procedure, as shown in Figure 1. Notice that the translation depends on an additional, implicit local variable `i` to control the loop. The use of a `var` parameter `x` for `body` allows for assignments to `x` within the body, as in the example that zeros the array.

Although it allows us to explore the meaning of the construct, this translation has disadvantages. Apart from the overhead of the procedure call, there is the problem that local variables like `s` referred to in the loop body must be addressed via the static link, and so cannot reside in registers. Additionally, we must be careful about the possibility, in the loop

> `for` *var* `in` *array* `do` *body* `end`,

that the array denoted by the expression *array* may change as *body* is executed. In the translation shown below, we use local variables to ensure that


```
proc sum(var a: vec): integer;
  var s: integer;

  proc body(var x: integer);
  begin
    s := s + x
  end;

  var i: integer;
begin
  s := 0;
  for i := 0 to 9 do
    body(a[i])
  end;
  return s
end;
```

Figure 1: *Translation with local procedure*

the action of the loop is unaffected by any such changes that happen after the loop has started.

1 Abstract and concrete syntax

We can add the new form of statement – let's call it an *IterStmt* – to the type *stmt* of abstract syntax trees like this:

type *stmt* = . . .
  | *IterStmt* **of** *name* ∗ *expr* ∗ *stmt* ∗ *def option ref*

The controlled variable is represented by a *name* tree that can be annotated with an appropriate definition; there is also an expression (*expr*) for the array to iterate over, and a statement (*stmt*) for the body. In a similar manner to the existing `for` loop, where a hidden variable keeps the upper bound, we allow for an additional hidden variable associated with the loop: it will hold an address just beyond the end of the array being iterated over.

We must add the new keyword `in` to the lexer and provide a terminal symbol *IN* as part of the parser specification. Then it's a straightforward matter to add a production to the grammar giving the concrete syntax of the new construct.

stmt : . . .
  | *FOR name IN expr DO stmts END*
    { *IterStmt* (\$2, \$4, \$6, *ref None*) }

In the same way as other control structures in the language, we allow a sequence of statements as the loop body, though these are represented by a single compound statement in the abstract syntax tree.

2 Semantic analysis

The name introduced as part of the new statement will behave rather like a reference parameter, but let's make it a new kind of entity *IterDef* in case there are differences in the way it should be treated. In file dict.mli:

```
type def_kind = ...
  | VarDef                    (* Variable *)
  | IterDef                   (* Iterator variable *)
  | ...
```

Analysing the new construct is part of the job of *check_stmt*. Our first task is to check that the given expression does indeed have an array type, and that it denotes a variable (lest we attempt to iterate over a string constant in a way that makes the characters of the string assignable).

```
let rec check_stmt s env alloc =
  match s.s_guts with ...
    | IterStmt (var, gen, body, limit) →
        let gt = check_expr gen env in
        if not (is_array gt) then
          sem_error "Iteration needs an array" [ ];
        check_var gen true;
```

It's convenient for this purpose to add a new function *is_array* to the *Dict* module; details are in the listings at the end of this document.

Continuing analysis of the array iteration construct, we next make two definitions of local names: the variable *var* is one of the new iterator variables, with a type that is the base type of the array. The hidden variable is an ordinary local with the generic address type *addrtype*.

```
        let vard = make_def var.x_name IterDef (base_type gt) in
        alloc vard; var.x_def ← Some vard;
        let limd = make_def (intern "*limit*") VarDef addrtype in
        alloc limd; limit := Some limd;
```

The final task is to check the loop body, in an environment where the loop variable has been added.

```
        let env' = add_def vard env in
        check_stmt body env' alloc
```

We also need to adjust the semantic analyser to account for places where the loop variable is used. Adjustments are needed in two places, one (in *has_value*) to cover the use of the variable in expressions, and the other (in *check_var*) for uses as a `var` parameter or on the left of an assignment.

```
(* has_value – check if object is suitable for use in expressions *)
let has_value d =
  match d.d_kind with
      ConstDef _ | VarDef | IterDef | ... → true
    | _ → false

(* check_var – check that expression denotes a variable *)
let rec check_var e addressible =
  match e.e_guts with
```

```
    Variable x →
      let d = get_def x in
      begin
        match d.d_kind with
            VarDef | IterDef | ... →
              d.d_mem ← d.d_mem || addressible
          | _ →
              sem_error "$ is not a variable" [fId x.x_name]
      end
  | ...
```

The remaining changes needed in the semantic analyser concern allocation of space for variables, and specifically the function *local_alloc* that allocates space in the stack frame for locals. This must be enhanced to deal with iterator variables as well as simple locals, like this:

```
(* local_alloc – allocate locals downward in memory *)
let local_alloc size nreg d =
  match d.d_kind with
      VarDef →
        (* Existing allocation code *)
    | IterDef →
        align addr_rep.r_align size;
        size := !size + addr_rep.r_size;
        d.d_addr ← Local (local_base − !size)
    | _ → ()
```

I have written this function so that it ignores objects (such as constants and procedures) that it is not interested in. A similar change to the function *global_alloc* used for global variables allows us to eliminate the function *do_alloc* rather than adjusting it, making the semantic analyser a bit tidier: see the listing.

3 Code generation

Our plan for generating code for the statement

```
for var in gen do body end
```

is given by the following outline, using the hidden variable *limit*.

```
addr := address(gen);
limit := addr + sizeof(array);
while addr < limit do
  <code for body>;
  addr := addr + sizeof(element)
end
```

Occurrences of `var` in the loop body refer to the array element at address `addr`.

Other translation schemes are possible: like the example translation given in the introduction, we might use an integer counter that is incremented by 1 in each iteration, and compute the address of the array element each time

by multiplying and adding. We could do this by explicitly pre-computing the element address before the loop body, or (perhaps slightly better) generate code to compute the address at each place where it is used, and then rely on common sub-expression elimination to merge the repeated calculations into one. These alternative schemes result in code that is generally slightly slower than the scheme outlined here. More importantly they also use more local variables, making the very simple register allocator of the compiler less likely to succeed in keeping everything in registers, even in simple examples.

We can implement our favoured scheme by adding to the function *gen_stmt* in tgen.ml. The first task is to retrieve the definitions of *var* and *limit* for use in the code, and to invent two labels.

```
(* gen_stmt – generate code for a statement *)
let rec gen_stmt s =
  let code =
    match s.s_guts with ...
      | IterStmt (var, gen, body, limit) →
          let vard = get_def var in
          let limd = get_opt !limit in
          let l1 = label ( ) and l2 = label ( ) in
```

This gives us the resources we need to build an optree for the code, following the outline above.

```
⟨SEQ,
  ⟨STOREW, gen_addr gen, address vard⟩,
  ⟨STOREW, ⟨OFFSET, ⟨LOADW, address vard⟩,
      ⟨CONST (size_of gen.e_type)⟩⟩, address limd⟩,
  ⟨LABEL l1⟩,
  ⟨JUMPC (GeqU, l2), ⟨LOADW, address vard⟩,
      ⟨LOADW, address limd⟩⟩,
  gen_stmt body,
  ⟨STOREW, ⟨OFFSET, ⟨LOADW, address vard⟩,
      ⟨CONST (size_of vard.d_type)⟩⟩, address vard⟩,
  ⟨JUMP l1⟩,
  ⟨LABEL l2⟩⟩
```

Termination of the loop is detected by comparing the address of the current element with a limiting value. Such addresses ought to use an unsigned comparison (for a signed comparison might overflow from positive to negative). Here we use the operator *GeqU* (unsigned greater-than-or-equal), which we will add to the back end later.

The other change needed to the code generator is in the treatment of places where the loop variable is used. Here the advice to treat it like a reference parameter is helpful, because in both cases we can form the address of the variable by loading a value stored in memory.

```
(* gen_addr – code for the address of a variable *)
let rec gen_addr v =
  match v.e_guts with
      Variable x →
        let d = get_def x in
        begin
```

```
            match d.d_kind with
                VarDef →
                  address d
              | VParamDef | IterDef →
                  ⟨LOADW, address d⟩
              | ...
          end
      | ...
```

4 Back end

We need to add the unsigned comparison operator *GeqU* to the intermediate language of optrees. Since the jump optimiser can replace a comparison operators by its negation, we also need to add the complementary operator *LtU*. These operators need be supported only in conditional jumps (and not as boolean values). They are implemented by adding to the function *e_stmt* in tran.ml:

```
(* e_stmt – generate code to execute a statement *)
let e_stmt t =
  match t with ...
    | ⟨JUMPC (GeqU, lab), t1, t2⟩ → condj "bhs" lab t1 t2
    | ⟨JUMPC (LtU, lab), t1, t2⟩ → condj "blo" lab t1 t2
    | ...
```

The ARM instructions we need are bhs – 'branch-if-higher-or-same' – and blo – 'branch-if-lower'.

We also need to add *GeqU* and *LtU* to the type *op*, and to the function *negate* in optree.ml that maps each comparison to its opposite.

```
let negate =
  function Eq → Neq | Neq → Eq | Lt → Geq
    | Leq → Gt | Gt → Leq | Geq → Lt
    | GeqU → LtU | LtU → GeqU
    | _ → failwith "negate"
```

5 Optimisation

Looking at the object code produced by the compiler, it seems quite reasonable, except that neither the *limit* variable, nor the address of the loop variable itself, are treated as register variables even in simple cases. For the *limit* variable, the fix is simple: its type *addrtype* is not among those regarded as scalar according to the test in the *Dict* module, and we can easily change that.

```
let scalar t =
  match t.t_guts with
      BasicType (IntType | ... | AddrType) → true
    | PointerType _ → true
    | _ → false
```

For the loop variable itself, we need to change what happens in *local_alloc* to permit it (or its address) to live in a register, replacing the code shown above with this improved version:

```
(* local_alloc – allocate locals downward in memory *)
let local_alloc size nreg d =
  match d.d_kind with
      VarDef → ...
    | IterDef →
        if !regvars && !nreg < Mach.nregvars then begin
          d.d_addr ← Register !nreg; incr nreg
        end
        else begin
          align addr_rep.r_align size;
          size := !size + addr_rep.r_size;
          d.d_addr ← Local (local_base − !size)
        end
    | _ → ()
```

Closer examination of the object code reveals some additional weaknesses in the compiler's implementation of common sub-expression elimination that I shall not pursue here.

6 Test cases

Several test cases are attached in the form of compiler test cases, with the object code embedded as a comment.

- iter.p is the example given in the problem statement.
- itersum.p shows a function that computes the sum of an array given as a reference parameter, and an equivalent program that uses a local subroutine.
- matrix.p illustrates the use of nested `for` statements to traverse a two-dimensional array.

Many other test cases could be added.

```
--- ../../labs/lab4/check.ml    2023-10-05 17:08:52.117515957 +0100
+++ check.ml    2023-10-20 16:27:32.657790912 +0100
@@ -90,7 +90,7 @@
 (* |has_value| -- check if object is suitable for use in expressions *)
 let has_value d =
   match d.d_kind with
-      ConstDef _ | VarDef | CParamDef | VParamDef | StringDef -> true
+      ConstDef _ | VarDef | IterDef | CParamDef | VParamDef | StringDef -> true
     | _ -> false

 (* |check_var| -- check that expression denotes a variable *)
@@ -100,7 +100,7 @@
         let d = get_def x in
         begin
           match d.d_kind with
-              VarDef | VParamDef | CParamDef ->
+             VarDef | VParamDef | CParamDef | IterDef ->
                d.d_mem <- d.d_mem || addressible
             | _ ->
                sem_error "$ is not a variable" [fId x.x_name]
@@ -361,6 +361,20 @@
         let d = make_def (intern "*upb*") VarDef integer in
         alloc d; upb := Some d

+    | IterStmt (var, gen, body, limit) ->
+        let gt = check_expr gen env in
+        if not (is_array gt) then
+          sem_error "Iteration needs an array" [];
+        check_var gen true;
+
+        let vard = make_def var.x_name IterDef (base_type gt) in
+        alloc vard; var.x_def <- Some vard;
+        let limd = make_def (intern "*limit*") VarDef addrtype in
+        alloc limd; limit := Some limd;
+
+        let env' = add_def vard env in
+        check_stmt body env' alloc
+
     | CaseStmt (sel, arms, deflt) ->
         let st = check_expr sel env in
         if not (discrete st) then
@@ -401,17 +415,29 @@

 (* local_alloc -- allocate locals downward in memory *)
 let local_alloc size nreg d =
-  if !regvars && not d.d_mem && scalar (d.d_type)
-      && !nreg < Mach.nregvars then begin
-    d.d_addr <- Register !nreg; incr nreg
-  end
-  else begin
-    let r = d.d_type.t_rep in
-    align r.r_align size;
-    size := !size + r.r_size;
-    d.d_addr <- Local (local_base - !size)
-  end
-
+  match d.d_kind with
+      VarDef ->
+        if !regvars && not d.d_mem && scalar (d.d_type)
+            && !nreg < Mach.nregvars then begin
+          d.d_addr <- Register !nreg; incr nreg
+        end
```

```
+        else begin
+          let r = d.d_type.t_rep in
+          align r.r_align size;
+          size := !size + r.r_size;
+          d.d_addr <- Local (local_base - !size)
+        end
+    | IterDef ->
+        if !regvars && !nreg < Mach.nregvars then begin
+          d.d_addr <- Register !nreg; incr nreg
+        end
+        else begin
+          align addr_rep.r_align size;
+          size := !size + addr_rep.r_size;
+          d.d_addr <- Local (local_base - !size)
+        end
+    | _ -> ()
+
 (* |param_size| -- size of a formal parameter *)
 let param_size d =
   match d.d_kind with
@@ -427,17 +453,11 @@

 (* |global_alloc| -- allocate label for global variable *)
 let global_alloc d =
-  let sym = sprintf "_$" [fId d.d_tag] in
-  d.d_addr <- Global sym
-
-(* |do_alloc| -- call allocation function for definitions that need it *)
-let do_alloc alloc ds =
-  let h d =
-    match d.d_kind with
-        VarDef | CParamDef | VParamDef | FieldDef | PParamDef ->
-          alloc d
-      | _ -> () in
-  List.iter h ds
+  match d.d_kind with
+      VarDef ->
+        let sym = sprintf "_$" [fId d.d_tag] in
+        d.d_addr <- Global sym
+    | _ -> ()

 (* |name_stack| -- stack of nested procedure names *)
 let name_stack = ref []
@@ -463,7 +483,7 @@
         let env' = check_decls fields (new_block env) in
         let defs = top_block env' in
         let size = ref 0 in
-        do_alloc (upward_alloc size) defs;
+        List.iter (upward_alloc size) defs;
         align max_align size;
         let r = { r_size = !size; r_align = max_align } in
         mk_type (RecordType defs) r
@@ -530,7 +550,7 @@
   let env' = check_decls fparams (new_block env) in
   let defs = top_block env' in
   let pcount = ref 0 in
-  do_alloc (param_alloc pcount) defs;
+  List.iter (param_alloc pcount) defs;
   decr level;
   let rt = (match result with
               Some te -> check_typexpr te env | None -> voidtype) in
@@ -557,7 +577,7 @@
```

```
   check_bodies env' ds;
   return_type := rt;
   check_stmt ss env' pre_alloc;
-  do_alloc (local_alloc fsize nregv) !defs;
+  List.iter (local_alloc fsize nregv) !defs;
   align max_align fsize

 (* |check_bodies| -- check bodies of procedure declarations *)
@@ -618,11 +638,10 @@
 let annotate (Prog (Block (globals, ss, fsize, nregv), glodefs)) =
   level := 0;
   let env = check_decls globals (new_block init_env) in
-  do_alloc global_alloc (top_block env);
+  List.iter global_alloc (top_block env);
   check_bodies env globals;
   return_type := voidtype;
   level := 1;
-  let alloc = local_alloc fsize nregv in
-  check_stmt ss env alloc;
+  check_stmt ss env (local_alloc fsize nregv);
   align max_align fsize;
   glodefs := top_block env
--- ../../labs/lab4/dict.ml     2023-10-05 17:08:52.129516553 +0100
+++ dict.ml     2023-10-20 16:27:32.657790912 +0100
@@ -78,6 +78,7 @@
   | StringDef                   (* String *)
   | TypeDef                     (* Type *)
   | VarDef                      (* Variable *)
+  | IterDef                     (* Iterator variable *)
   | CParamDef                   (* Value parameter *)
   | VParamDef                   (* Var parameter *)
   | FieldDef                    (* Field of record *)
@@ -209,7 +210,7 @@

 let scalar t =
   match t.t_guts with
-      BasicType (IntType | CharType | BoolType) -> true
+      BasicType (IntType | CharType | BoolType | AddrType) -> true
     | PointerType _ -> true
     | _ -> false

@@ -218,6 +219,9 @@
       PointerType t1 -> true
     | _ -> false

+let is_array t =
+  match t.t_guts with ArrayType (_, _) -> true | _ -> false
+
 let bound t =
   match t.t_guts with
       ArrayType (n, t1) -> n
--- ../../labs/lab4/dict.mli    2023-10-05 17:08:52.125516354 +0100
+++ dict.mli    2023-10-20 16:27:32.661790945 +0100
@@ -42,6 +42,7 @@
   | StringDef                   (* String constant *)
   | TypeDef                     (* Type *)
   | VarDef                      (* Variable *)
+  | IterDef                     (* Iterator variable *)
   | CParamDef                   (* Value parameter *)
   | VParamDef                   (* Var parameter *)
   | FieldDef                    (* Field of record *)
@@ -150,6 +151,9 @@
```

```
 (* |bound| -- get bound of array type *)
 val bound : ptype -> int

+(* |is_array| -- test for array type *)
+val is_array : ptype -> bool
+
 (* |is_pointer| -- test if a type is 'pointer to T' *)
 val is_pointer : ptype -> bool

--- ../../labs/lab4/optree.mli  2023-10-05 17:08:52.117515957 +0100
+++ optree.mli  2023-10-20 16:27:32.661790945 +0100
@@ -23,7 +23,7 @@

 (* |op| -- type of picoPascal operators *)
 type op = Plus | Minus | Times | Div | Mod | Eq
-  | Uminus | Lt | Gt | Leq | Geq | Neq
+  | Uminus | Lt | Gt | Leq | Geq | Neq | GeqU | LtU
   | And | Or | Not | Lsl | Lsr | Asr | BitAnd | BitOr | BitNot

 (* |fOp| -- format operator for printing *)
--- ../../labs/lab4/optree.ml   2023-10-05 17:08:52.125516354 +0100
+++ optree.ml   2023-10-20 16:27:32.661790945 +0100
@@ -25,16 +25,16 @@

 (* |op| -- type of picoPascal operators *)
 type op = Plus | Minus | Times | Div | Mod | Eq
-  | Uminus | Lt | Gt | Leq | Geq | Neq | And | Or | Not | Lsl
-  | Lsr | Asr | BitAnd | BitOr | BitNot
+  | Uminus | Lt | Gt | Leq | Geq | Neq | GeqU | LtU
+  | And | Or | Not | Lsl | Lsr | Asr | BitAnd | BitOr | BitNot

 let op_name =
   function
       Plus -> "Plus" | Minus -> "Minus" | Times -> "Times"
     | Div -> "Div" | Mod -> "Mod" | Eq -> "Eq"
     | Uminus -> "Uminus" | Lt -> "Lt" | Gt -> "Gt"
-    | Leq -> "Leq" | Geq -> "Geq" | Neq -> "Neq"
-    | And -> "And" | Or -> "Or" | Not -> "Not"
+    | Leq -> "Leq" | Geq -> "Geq" | Neq -> "Neq" | GeqU -> "GEQU"
+    | LtU -> "LTU" | And -> "And" | Or -> "Or" | Not -> "Not"
     | Lsl -> "Lsl" | Lsr -> "Lsr" | Asr -> "Asr"
     | BitAnd -> "BitAnd" | BitOr -> "BitOr" | BitNot -> "BitNot"

@@ -142,6 +142,7 @@
 let negate =
   function Eq -> Neq | Neq -> Eq | Lt  -> Geq
     | Leq -> Gt | Gt  -> Leq | Geq -> Lt
+    | GeqU -> LtU | LtU -> GeqU
     | _ -> failwith "negate"


--- ../../labs/lab4/tgen.ml     2023-10-20 16:13:52.955228646 +0100
+++ tgen.ml     2023-10-20 16:27:32.661790945 +0100
@@ -12,6 +12,9 @@
 let optlevel = ref 0
 let debug = ref 0

+let get_opt =
+  function Some x -> x | None -> failwith "get_opt"
+
 (* |level| -- nesting level of current procedure *)
 let level = ref 0
```

```
@@ -80,7 +83,7 @@
           match d.d_kind with
               VarDef ->
                 address d
-            | VParamDef ->
+            | VParamDef | IterDef ->
                 <LOADW, address d>
             | CParamDef ->
                 if scalar d.d_type || is_pointer d.d_type then
@@ -300,7 +303,7 @@
       | ForStmt (var, lo, hi, body, upb) ->
           (* Use previously allocated temp variable to store upper bound.
              We could avoid this if the upper bound is constant. *)
-          let tmp = match !upb with Some d -> d | _ -> failwith "for" in
+          let tmp = get_opt !upb in
           let l1 = label () and l2 = label () in
           <SEQ,
             <STOREW, gen_expr lo, gen_addr var>,
@@ -312,6 +315,22 @@
             <JUMP l1>,
             <LABEL l2>>

+      | IterStmt (var, gen, body, limit) ->
+          let vard = get_def var in
+          let limd = get_opt !limit in
+          let l1 = label () and l2 = label () in
+          <SEQ,
+            <STOREW, gen_addr gen, address vard>,
+            <STOREW, <OFFSET, <LOADW, address vard>,
+                <CONST (size_of gen.e_type)>>, address limd>,
+            <LABEL l1>,
+            <JUMPC (GeqU, l2), <LOADW, address vard>, <LOADW, address limd>>,
+            gen_stmt body,
+            <STOREW, <OFFSET, <LOADW, address vard>,
+                <CONST (size_of vard.d_type)>>, address vard>,
+            <JUMP l1>,
+            <LABEL l2>>
+
       | CaseStmt (sel, arms, deflt) ->
           (* Use one jump table, and hope it is reasonably compact *)
           let deflab = label () and donelab = label () in
--- ../../labs/lab4/lexer.mll   2023-10-05 23:36:27.294645258 +0100
+++ lexer.mll   2023-10-20 16:27:32.661790945 +0100
@@ -18,7 +18,7 @@
       ("type", TYPE); ("var", VAR); ("while", WHILE);
       ("pointer", POINTER); ("nil", NIL);
       ("repeat", REPEAT); ("until", UNTIL); ("for", FOR);
-      ("elsif", ELSIF); ("case", CASE);
+      ("elsif", ELSIF); ("case", CASE); ("in", IN);
       ("and", MULOP And); ("div", MULOP Div); ("or", ADDOP Or);
       ("not", NOT); ("mod", MULOP Mod) ]

--- ../../labs/lab4/parser.mly  2023-10-05 17:08:52.133516753 +0100
+++ parser.mly  2023-10-20 16:27:32.661790945 +0100
@@ -22,7 +22,7 @@
 %token                  ARRAY BEGIN CONST DO ELSE END IF OF
 %token                  PROC RECORD RETURN THEN TO TYPE
 %token                  VAR WHILE NOT POINTER NIL
-%token                  REPEAT UNTIL FOR ELSIF CASE
+%token                 REPEAT UNTIL FOR ELSIF CASE IN
```

```
 %type <Tree.program>    program
 %start                  program
@@ -119,6 +119,7 @@
   | FOR name ASSIGN expr TO expr DO stmts END
                                         { let v = make_expr (Variable $2) in
                                           ForStmt (v, $4, $6, $8, ref None) }
+  | FOR name IN expr DO stmts END        { IterStmt ($2, $4, $6, ref None) }
   | CASE expr OF arms else_part END     { CaseStmt ($2, $4, $5) } ;

 elses :
--- ../../labs/lab4/tran.ml     2023-10-05 17:08:52.137516951 +0100
+++ tran.ml     2023-10-20 16:27:32.661790945 +0100
@@ -247,6 +247,8 @@
     | <JUMPC (Leq, lab), t1, t2> -> condj "le" lab t1 t2
     | <JUMPC (Geq, lab), t1, t2> -> condj "ge" lab t1 t2
     | <JUMPC (Neq, lab), t1, t2> -> condj "ne" lab t1 t2
+    | <JUMPC (GeqU, lab), t1, t2> -> condj "hs" lab t1 t2
+    | <JUMPC (LtU, lab), t1, t2> -> condj "lo" lab t1 t2

     | <JCASE (table, deflab), t1> ->
         (* This jump table code exploits the fact that on ARM,
--- ../../labs/lab4/tree.ml     2023-10-05 17:08:52.125516354 +0100
+++ tree.ml     2023-10-20 16:27:32.661790945 +0100
@@ -39,6 +39,7 @@
   | WhileStmt of expr * stmt
   | RepeatStmt of stmt * expr
   | ForStmt of expr * expr * expr * stmt * def option ref
+  | IterStmt of name * expr * stmt * def option ref
   | CaseStmt of expr * (expr * stmt) list * stmt

 and expr =
@@ -148,6 +149,8 @@
         fMeta "(REPEAT $ $)" [fStmt body; fExpr test]
     | ForStmt (var, lo, hi, body, _) ->
         fMeta "(FOR $ $ $ $)" [fExpr var; fExpr lo; fExpr hi; fStmt body]
+    | IterStmt (var, gen, body, _) ->
+        fMeta "(ITER $ $ $)" [fName var; fExpr gen; fStmt body]
     | CaseStmt (sel, arms, deflt) ->
         let fArm (lab, body) = fMeta "($ $)" [fExpr lab; fStmt body] in
         fMeta "(CASE $ $ $)" [fExpr sel; fList(fArm) arms; fStmt deflt]
--- ../../labs/lab4/tree.mli    2023-10-05 17:08:52.129516553 +0100
+++ tree.mli    2023-10-20 16:27:32.661790945 +0100
@@ -54,6 +54,7 @@
   | WhileStmt of expr * stmt
   | RepeatStmt of stmt * expr
   | ForStmt of expr * expr * expr * stmt * def option ref
+  | IterStmt of name * expr * stmt * def option ref
   | CaseStmt of expr * (expr * stmt) list * stmt

 and expr =
```

```
type point = record x, y: integer end;

var a: array 10 of point; i, sum: integer;

begin
  i := 0;
  for r in a do
    r.x := i mod 3;
    r.y := i mod 5;
    i := i+1
  end;

  sum := 0;
  for r in a do
    sum := sum + r.x * r.y
  end;
  print_num(sum); newline()
end.

(*<<
17
>>*)

(*[[
@ picoPascal compiler output
        .include "fixup.s"
        .global pmain

        .text
pmain:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
        sub sp, sp, #8
@   i := 0;
        mov r0, #0
        set r1, _i
        str r0, [r1]
@   for r in a do
        set r4, _a
        add r5, r4, #80
.L2:
        cmp r4, r5
        bhs .L3
@     r.x := i mod 3;
        set r7, _i
        mov r1, #3
        ldr r0, [r7]
        bl int_mod
        str r0, [r4]
@     r.y := i mod 5;
        mov r1, #5
        ldr r0, [r7]
        bl int_mod
        str r0, [r4, #4]
@     i := i+1
        ldr r0, [r7]
        add r0, r0, #1
        str r0, [r7]
        add r4, r4, #8
        b .L2
.L3:
```

```
@   sum := 0;
        mov r0, #0
        set r1, _sum
        str r0, [r1]
@   for r in a do
        set r6, _a
        add r0, r6, #80
        str r0, [fp, #-4]
.L4:
        ldr r0, [fp, #-4]
        cmp r6, r0
        bhs .L5
@     sum := sum + r.x * r.y
        set r7, _sum
        ldr r0, [r7]
        ldr r1, [r6]
        ldr r2, [r6, #4]
        mul r1, r1, r2
        add r0, r0, r1
        str r0, [r7]
        add r6, r6, #8
        b .L4
.L5:
@   print_num(sum); newline()
        set r0, _sum
        ldr r0, [r0]
        bl print_num
        bl newline
        ldmfd fp, {r4-r10, fp, sp, pc}
        .ltorg

        .comm _a, 80, 4
        .comm _i, 4, 4
        .comm _sum, 4, 4
@ End
]]*)
```

```
type vec = array 10 of integer;

proc sum1(var a: vec): integer;
  var s: integer;
begin
  s := 0;
  for x in a do s := s + x end;
  return s
end;

proc sum2(var a: vec): integer;
  var s: integer;

  proc body(var x: integer);
  begin
    s := s + x
  end;

  var i: integer;
begin
  s := 0;
  for i := 0 to 9 do
    body(a[i])
  end;
  return s
end;

var b: vec; j: integer;

begin
  for j := 0 to 9 do b[j] := (j+1)*(j+1) end;

  print_num(sum1(b)); newline();
  print_num(sum2(b)); newline()
end.

(*<<
385
385
>>*)

(*[[
@ picoPascal compiler output
        .include "fixup.s"
        .global pmain

@ proc sum1(var a: vec): integer;
        .text
_sum1:
        mov ip, sp
        stmfd sp!, {r0-r1}
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@   s := 0;
        mov r4, #0
@   for x in a do s := s + x end;
        ldr r5, [fp, #40]
        add r6, r5, #40
.L3:
        cmp r5, r6
        bhs .L4
        ldr r0, [r5]
```

```
        add r4, r4, r0
        add r5, r5, #4
        b .L3
.L4:
@   return s
        mov r0, r4
        ldmfd fp, {r4-r10, fp, sp, pc}
        .ltorg

@ proc sum2(var a: vec): integer;
_sum2:
        mov ip, sp
        stmfd sp!, {r0-r1}
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
        sub sp, sp, #8
@   s := 0;
        mov r0, #0
        str r0, [fp, #-4]
@   for i := 0 to 9 do
        mov r4, #0
        mov r5, #9
.L6:
        cmp r4, r5
        bgt .L7
@     body(a[i])
        ldr r0, [fp, #40]
        lsl r1, r4, #2
        add r0, r0, r1
        mov r10, fp
        bl _body_g1
        add r4, r4, #1
        b .L6
.L7:
@   return s
        ldr r0, [fp, #-4]
        ldmfd fp, {r4-r10, fp, sp, pc}
        .ltorg

@   proc body(var x: integer);
_body_g1:
        mov ip, sp
        stmfd sp!, {r0-r1}
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@     s := s + x
        ldr r0, [fp, #24]
        add r4, r0, #-4
        ldr r0, [r4]
        ldr r1, [fp, #40]
        ldr r1, [r1]
        add r0, r0, r1
        str r0, [r4]
        ldmfd fp, {r4-r10, fp, sp, pc}
        .ltorg

pmain:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@   for j := 0 to 9 do b[j] := (j+1)*(j+1) end;
        mov r0, #0
```

```
        set r1, _j
        str r0, [r1]
        mov r4, #9
.L10:
        set r5, _j
        ldr r6, [r5]
        cmp r6, r4
        bgt .L11
        add r7, r6, #1
        mul r0, r7, r7
        set r1, _b
        lsl r2, r6, #2
        add r1, r1, r2
        str r0, [r1]
        ldr r0, [r5]
        add r0, r0, #1
        str r0, [r5]
        b .L10
.L11:
@   print_num(sum1(b)); newline();
        set r5, _b
        mov r0, r5
        bl _sum1
        bl print_num
        bl newline
@   print_num(sum2(b)); newline()
        mov r0, r5
        bl _sum2
        bl print_num
        bl newline
        ldmfd fp, {r4-r10, fp, sp, pc}
        .ltorg

        .comm _b, 40, 4
        .comm _j, 4, 4
@ End
]]*)
```

```
type matrix = array 3 of array 3 of integer;

proc set(var m: matrix);
  var x: integer;
begin
  x := 0;
  for row in m do
    for y in row do
      y := x; x := x+1
    end
  end
end;

proc test();
  var m: matrix;
begin
  set(m);

  for row in m do
    for y in row do
      print_num(y)
    end;
    newline()
  end
end;

begin
  test()
end.

(*<<
012
345
678
>>*)

(*[[
@ picoPascal compiler output
        .include "fixup.s"
        .global pmain

@ proc set(var m: matrix);
        .text
_set:
        mov ip, sp
        stmfd sp!, {r0-r1}
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
        sub sp, sp, #8
@   x := 0;
        mov r4, #0
@   for row in m do
        ldr r5, [fp, #40]
        add r6, r5, #36
.L2:
        cmp r5, r6
        bhs .L1
@     for y in row do
        str r5, [fp, #-4]
        ldr r7, [fp, #-4]
        add r0, r7, #12
        str r0, [fp, #-8]
```

```
.L4:
        ldr r7, [fp, #-4]
        ldr r0, [fp, #-8]
        cmp r7, r0
        bhs .L5
@       y := x; x := x+1
        str r4, [r7]
        add r4, r4, #1
        ldr r0, [fp, #-4]
        add r0, r0, #4
        str r0, [fp, #-4]
        b .L4
.L5:
        add r5, r5, #12
        b .L2
.L1:
        ldmfd fp, {r4-r10, fp, sp, pc}
        .ltorg

@ proc test();
_test:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
        sub sp, sp, #40
@   set(m);
        add r0, fp, #-36
        bl _set
@   for row in m do
        add r4, fp, #-36
        add r5, r4, #36
.L7:
        cmp r4, r5
        bhs .L6
@     for y in row do
        mov r6, r4
        add r0, r6, #12
        str r0, [fp, #-40]
.L9:
        ldr r0, [fp, #-40]
        cmp r6, r0
        bhs .L10
@       print_num(y)
        ldr r0, [r6]
        bl print_num
        add r6, r6, #4
        b .L9
.L10:
@     newline()
        bl newline
        add r4, r4, #12
        b .L7
.L6:
        ldmfd fp, {r4-r10, fp, sp, pc}
        .ltorg

pmain:
        mov ip, sp
        stmfd sp!, {r4-r10, fp, ip, lr}
        mov fp, sp
@   test()
        bl _test
```

```
        ldmfd fp, {r4-r10, fp, sp, pc}
        .ltorg

@ End
]]*)
```