# Implementing multi-level break and compact storage

Mike Spivey, December 2020

This year's Christmas Assignment asks for implementations of one feature no sane programmer would use (multi-level `break` statements) and another that every programmer can benefit from (compact storage for records and local variables).

## 1 Break statements

Implementing `break` statements is a straight-forward exercise that follows through the stages of the compiler. The idea is that every kind of loop – beginning `while`, `repeat`, or `for` – should support embedded `break` statements, with `break 3` (for example) breaking out of three levels of loop, and `break` on its own equivalent of `break 1`.

We may as well make `break` and `break 1` equivalent in the parser, so that the abstract syntax is simply (from tree.mli)

```
type stmt_guts =
    ...
  | Break of int
  | ...
```

We first augment the lexer with a new keyword – in lexer.mll:

```
let symtable =
  Util.make_hash 100 [ ...; ("break", BREAK); ... ]
```

That keyword is used for the concrete syntax – in parser.mly we add a new token type *BREAK*, and productions for the nonterminal $stmt_1$ that denotes different kinds of statement:

```
stmt1 : ...
  | BREAK                 { Break 1 }
  | BREAK NUMBER          { Break $2 }
  | ...
```

There's no particular advantage for clarity or efficiency in introducing a new nonterminal *level_opt* for the optional level appearing after `break`.

Following through the compiler pass by pass, we can make a minimal alteration to the semantic analyser by adding an argument *nest* that gives the nest-

 

ing depth to the function *check_stmt* for analysing statements. The changes are straightforward, and to illustrate the approach I give here the checking code for `while` loops and `break` statements. Other kinds of loop follow the pattern of incrementing *nest* in the recursive call for the loop body, while constructs that are not loops simply pass on *nest* unchanged.

```
(* check_stmt - check and annotate a statement *)
let rec check_stmt s env alloc nest =
  err_line := s.s_line;
  match s.s_guts with
      ...

  | WhileStmt (cond, body) →
      let ct = check_expr cond env in
      if not (same_type ct boolean) then
        sem_error "type mismatch in while statement" [ ];
      check_stmt body env alloc (nest + 1)

  | Break n →
      if nest = 0 then
        sem_error "break outside any loop" [ ]
      else if n < 1 || n > nest then
        sem_error "break depth not in range" [ ]
```

It's best here to generate a special error message for the case *nest* = 0, because that covers the case where `break` is implicitly equivalent to `break 1` without mentioning the depth argument inserted by the parser. There's no great advantage in allocating code labels at this stage, since there is no interaction between the environment and loop nesting. The places where procedure bodies and the main program are analysed need a small change to initialise the *nest* argument to zero when calling *check_stmt*.

Next we can look at the intermediate code generator tgen.ml, and our job is then done, because the new statements can be implemented in terms of existing back-end mechanisms, and the jump optimiser works as well for them as for other patterns of branching. This time, we add to *gen_stmt* an extra argument that is a *stack* of break labels.

```
let rec gen_stmt s brkstack =
  let code =
    match s.s_guts with
        ...

    | WhileStmt (test, body) →
        let l1 = label ( ) and l2 = label ( ) and l3 = label( ) in
        ⟨SEQ,
          ⟨LABEL l1⟩,
          gen_cond test l2 l3,
          ⟨LABEL l2⟩,
          gen_stmt body (l3 :: brkstack),
          ⟨JUMP l1⟩,
          ⟨LABEL l3⟩⟩

    | Break n →
        ⟨JUMP (List.nth brkstack (n−1))⟩
```

As with the nesting count in the semantic analyser, we can pass the break stack to recursive calls unchanged when the construct is not a loop. In the case of loops, each construct already has a label at the bottom that can be pushed on the break stack for the recursive call. There's just one place – in *do_proc* – where the stack is initialised to empty.

A good report will contain a variety of test cases, covering at least every kind of loop and various orders of nesting. I'll content myself with one, a nested `for` loop with a `break` out of the middle.

```
var i, j, x: integer;

begin
  for i := 1 to 5 do
    for j := 1 to 5 do
      x := 10*i + j;
      print_num(x); newline();
      if x = 32 then break 2 end
    end
  end
end.
```

## 2 Compact storage

Alignment requirements for different data types can lead to gaps in the storage that is allocated, both in record types and in stack frames. We can remove the gaps by the simple expedient of sorting the fields or local variables in *descending* order of alignment, so that (in our simple language) all the four-byte integers come before all the one-byte characters and booleans. More generally, if we put a variable or field with alignment $n$ after one with alignment $2n$, we can be sure that the second item will be aligned on an $n$-byte boundary, so no padding will be needed (and the same if we substitute $2^k n$ for $2n$). This will work provided alignments are all powers of two, and the size of each type is a multiple of its alignment, a constraint that we will be careful to respect when we adjust the representation of record types, padding them at the end if necessary.

Additionally, the existing compiler gives each record type the maximum alignment of any datatype, and that leads to gaps in the representation of types like

```
array 10 of record hi, lo: char end
```

because each element of the array occupies two bytes of a four-byte record, padded at the end. This can be fixed by giving each record type the maximum alignment of any of its fields, so that the record type above would have size 2 and alignment 1. The array then gets size 20 and alignment 1.

These ideas can be implemented entirely within the semantic analyser. At present, local variables can live in registers, and the compiler uses the simple but effective device of allocating registers to the first few variables declared in each procedure, omitting those that must be addressible from nested routines. It's hard to work out how this scheme will jibe with sorting the declarations, so I started by disabling register variables by nuking the appropriate part of *loc_alloc*. Support for register variables in the back-end

can remain in place: it will not be used but does no harm. I refreshed the code for the test cases using

```
$ make promote
```

so as to provide a baseline for whatever improvements we can achieve.

### 2.1 Test cases

I also added two new test cases where I hoped the improvement would be noticeable. The test pack.p contains a procedure where the sorting should give a more compact layout.

```
proc foo(x, y: integer): integer;
  var a: char; b: integer; c: boolean; d: integer;
begin
  a := chr(x);
  b := x + y;
  c := (x = y);
  d := x - y;
  print_char(a);
  if c then
    return b
  else
    return d
  end
end;

begin
  print_num(foo(65, 66)); newline();
end.
```

Test fillrec.p contains the same array of records as above: it should fit in 20 bytes, not 40 after the representation is improved.

```
var r: array 10 of record lo, hi: char end;

proc fill(s: array 20 of char);
  var i: integer;
begin
  for i := 0 to 9 do
    r[i].lo := s[2*i];
    r[i].hi := s[2*i+1]
  end
end;

proc print_low();
  var i: integer;
begin
  for i := 0 to 9 do
    print_char(r[i].lo)
  end
end;

begin
  fill("ABCDEFGHIJKLMNOPQRST");
```

```
  print_low(); newline()
end.
```

More tests would be needed in a perfect answer!

## 2.2 Implementation

Implementing the improvements is surprisingly easy, given the fact that storage for each block is allocated by first assembling a list of definitions *ds*, then calling the function *do_alloc*, defined as follows.

```
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
```

This calls a specified function *alloc* on each of the definitions in the list that actually require runtime storage, omitting things like *ConstDef* that describe 'compile-time' constants.

A typical function passed as *alloc* is the one for allocating local variables in a stack frame, defined (after removing register variables) by

```
let local_alloc size d =
  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)
```

This finds the concrete representation (size and alignment) of the type of a definition *d*, rounds up the variable *size* to a multiple of the alignment, then increases it by the size of the type. Because local variables are allocated downwards in memory, the value of *size* is subtracted from *local_base* to form the address: that constant gives the offset (if any) between the frame base and the area for local variables. On the ARM it is zero, as explained in an ASCII-art diagram in mach.ml.

Sorting local variables requires just a new variant of *do_alloc* that sorts the definitions before allocating, using an appropriate library function and a bit of higher-order programming. It's good to use the library function *List.stable_sort*, so as not to change gratuitously the order of variables that have the same alignment. That function has type

```
val stable_sort : (α → α → int) → α list → α list.
```

It takes a *comparison function* that maps two values to an indicator – positive, negative, or zero – saying how they compare. It's possible to use *List.stable_sort* directly, but I prefer to introduce a function *sort_descending* that sorts using a simpler scoring function:

```
let sort_descending score xs =
  let cf x y = score y − score x in
  List.stable_sort cf xs
```

We can then define *sorted_alloc* by using the alignment as a scoring function, then calling the original *do_alloc*.

```
let sorted_alloc alloc ds =
  do_alloc alloc (sort_descending (fun d → d.d_type.t_rep.r_align) ds)
```

It does no damage to include compile-time constants and other detritus with the list *ds*, because after sorting they are just ignored by *do_alloc*.

We cannot use *sorted_alloc* everywhere *do_alloc* was used before, because some declarations, for example, formal parameters, must maintain their order. But we *can* use it for the local variables of a procedure:

```
let rec check_block level rt env (Block (ds, ss, fsize, nregv)) =
  let env' = check_decls ds (new_block env) in
  let defs = ref (top_block env') in
  let pre_alloc d = defs := !defs @ [d] in
  check_bodies env' ds;
  return_type := rt;
  check_stmt ss env' pre_alloc 0;
  sorted_alloc (local_alloc fsize) !defs;
  align max_align fsize
```

This code shows the mechanism (originally introduced to support register variables) that uses a dummy function *pre_alloc* to gather definitions first, then sorts and allocates them later. The only changes here are to use *sorted_alloc* in place of *do_alloc*, and to use the *local_alloc* shown above that omits register variables.

The *sorted_alloc* function can also be used for record fields, and we can additionally compute the necessary alignment for the whole record a bit more carefully. For tidiness, it's wise to ensure that the empty record type works properly, even if the source language forbids it.

```
let max_alignment defs =
  List.fold_left (fun m d → max m d.d_type.t_rep.r_align) 1 defs

let rec check_typexpr te env =
  match te with
      ...
    | Record fields →
        let env' = check_decls fields (new_block env) in
        let defs = top_block env' in
        let size = ref 0 in
        sorted_alloc (upward_alloc size) defs;
        let almt = max_alignment defs in
        align almt size;
        let r = { r_size = !size; r_align = almt } in
        mk_type (RecordType defs) r
```

The changes here are to use *sorted_alloc* in place of *do_alloc*, and to compute the alignment *almt* as the maximum alignment of any field (or 1 for the empty record), using the helper function *max_alignment* defined in terms of *List.fold_left*. We use the alignment for two purposes: first, to round up the size to a multiple of the alignment, and second, to be the alignment of the

record type itself. Padding at the end of a record is still needed for a type such as

```
record a: char; b: integer; c: char end,
```

which after sorting becomes the same as `record b: integer; a, c: char end`. It will be represented by an eight-byte object containing the four-byte integer `b`, two one-byte characters `a` and `c`, and two bytes of padding. The padding is necessary so that in an array of records placed next to each other, each individual record is properly aligned.

### 2.3 Results

Let's see – relative to the baseline where register variables have been eliminated from the Lab 4 compiler – how the claimed improvements stack up. First, the test case `pack.p`. Compiling with declaration sorting changes the addresses that are used for local variables: for example, the first two lines of the procedure body are translated[1]

```
@ a := chr(x);                  @ a := chr(x);
   ldr r0, [fp, #40]               ldr r0, [fp, #40]
   strb r0, [fp, #-1]              strb r0, [fp, #-9]
@ b := x + y;                   @ b := x + y;
   ldr r0, [fp, #40]               ldr r0, [fp, #40]
   ldr r1, [fp, #44]               ldr r1, [fp, #44]
   add r0, r0, r1                  add r0, r0, r1
   str r0, [fp, #-8]               str r0, [fp, #-4]
```

before (left) and after (right) the change. You can see that the addresses of the parameters `x` and `y` are unchanged as +40 and +44 from the frame pointer, but the locals `a` and `b` change their addresses from $-1$ and $-8$ to $-9$ and $-4$ because of the sorting by alignment. Sadly, the frame size of the procedure is not reduced, because whereas before it was $(1 + 3) + 4 + (1 + 3) + 4 = 16$ bytes, after the change it becomes $4 + 4 + 1 + 1 = 10$ bytes, but the size is rounded up to a multiple of 8 to satisfy alignment requirements of the ARM ABI, so the end result is unchanged.

The other test case fares better. Here, the object code (with `-O`) for an assignment to `r` looks like this:

```
@ r[i].lo := s[2*i];            @ r[i].lo := s[2*i];
   ldr r0, [fp, #40]               ldr r0, [fp, #40]
   ldr r1, [fp, #-4]               ldr r1, [fp, #-4]
   lsl r1, r1, #1                  lsl r1, r1, #1
   add r0, r0, r1                  add r0, r0, r1
   ldrb r0, [r0]                   ldrb r0, [r0]
   set r1, _r                      set r1, _r
   ldr r2, [fp, #-4]               ldr r2, [fp, #-4]
   lsl r2, r2, #2                  lsl r2, r2, #1
   add r1, r1, r2                  add r1, r1, r2
   strb r0, [r1]                   strb r0, [r1]
```

Note that the compiler, based on the original code for Lab 4, does not have access to the *reg* + *reg* addressing mode; basing our work on the solution

[1] I've shown the unoptimised code for simplicity.

to Lab 4 would have given shorter but equivalent code. The only difference between the two samples is in the addressing calculation for array `r`, which multiplies `i` by 4 in one case, and by 2 in the other – because each element of `r` has reduced from 4 bytes to 2. Such differences in the addressing calculations are the only differences between the code output by the two compilers for the whole test case, apart from the assembly language directive that reserves space for `r`:

```
.comm _r, 40, 4                    .comm _r, 20, 4
```

As each element has reduced from 4 bytes to 2, so has the whole array reduced from 40 bytes to 20.

A few of the existing test cases show minor improvements, though rarely enough to decrease any actual frame sizes, once the 8-byte stack alignment is taken into account. The test case `sudoku.p`, a solver for Sudoku problems based on Knuth's method of Dancing Links, shows a particularly satisfying improvement, because there is a type *ColRec* of dynamically allocated records that has a couple of single-byte fields, and shrinks from 32 to 28 bytes when they are reordered. Even here the improvement may be illusory, because the storage allocator may gain speed by providing only a limited number of sizes for small blocks, and trading a little internal fragmentation for the increase in speed and a decrease in external fragmentation. In this case, a size of 28 may well be rounded up to 32, a power of two and also a multiple of the typical 16-byte cache line size.

## 3 Listings

Appended are two *diff* listings, one showing the changes needed to many files to implement the multi-level `break` statement, and the other showing the changes to `check.ml` needed to reduce the amount of padding between variables and record fields.

```
--- d1/check.ml 2023-10-08 16:05:22.255126557 +0100
+++ d2/check.ml 2023-10-08 16:05:22.263126218 +0100
@@ -288,13 +288,13 @@
   chk (List.sort compare vs)

 (* |check_stmt| -- check and annotate a statement *)
-let rec check_stmt s env alloc =
+let rec check_stmt s env alloc nest =
   err_line := s.s_line;
   match s.s_guts with
       Skip -> ()

     | Seq ss ->
-        List.iter (fun s1 -> check_stmt s1 env alloc) ss
+       List.iter (fun s1 -> check_stmt s1 env alloc nest) ss

     | Assign (lhs, rhs) ->
         let lt = check_expr lhs env
@@ -328,17 +328,17 @@
         let ct = check_expr cond env in
         if not (same_type ct boolean) then
           sem_error "test in if statement must be a boolean" [];
-        check_stmt thenpt env alloc;
-        check_stmt elsept env alloc
+       check_stmt thenpt env alloc nest;
+       check_stmt elsept env alloc nest

     | WhileStmt (cond, body) ->
         let ct = check_expr cond env in
         if not (same_type ct boolean) then
           sem_error "type mismatch in while statement" [];
-        check_stmt body env alloc
+       check_stmt body env alloc (nest+1)

     | RepeatStmt (body, test) ->
-        check_stmt body env alloc;
+        check_stmt body env alloc (nest+1);
         let ct = check_expr test env in
         if not (same_type ct boolean) then
           sem_error "type mismatch in repeat statement" []
@@ -351,13 +351,19 @@
             || not (same_type hit integer) then
           sem_error "type mismatch in for statement" [];
         check_var var false;
-        check_stmt body env alloc;
+        check_stmt body env alloc (nest+1);

         (* Allocate space for hidden variable.  In the code, this will
            be used to save the upper bound.  *)
         let d = make_def (intern "*upb*") VarDef integer in
         alloc d; upb := Some d

+    | Break n ->
+        if nest = 0 then
+          sem_error "break outside any loop" []
+        else if n < 1 || n > nest then
+          sem_error "break depth not in range" []
+
     | CaseStmt (sel, arms, deflt) ->
         let st = check_expr sel env in
         if not (discrete st) then
@@ -367,11 +373,11 @@
```

```
           let (t1, v) = check_const lab env in
           if not (same_type t1 st) then
             sem_error "case label has wrong type" [];
-          check_stmt body env alloc; v in
+          check_stmt body env alloc nest; v in

         let vs = List.map check_arm arms in
         check_dupcases vs;
-        check_stmt deflt env alloc
+       check_stmt deflt env alloc nest


 (* TYPES AND DECLARATIONS *)
@@ -547,7 +553,7 @@
   let pre_alloc d = defs := !defs @ [d] in
   check_bodies env' ds;
   return_type := rt;
-  check_stmt ss env' pre_alloc;
+  check_stmt ss env' pre_alloc 0;
   do_alloc (local_alloc fsize) !defs;
   align max_align fsize

@@ -614,6 +620,6 @@
   return_type := voidtype;
   level := 1;
   let alloc = local_alloc fsize in
-  check_stmt ss env alloc;
+  check_stmt ss env alloc 0;
   align max_align fsize;
   glodefs := top_block env
--- d1/tgen.ml  2023-10-08 15:18:06.787585264 +0100
+++ d2/tgen.ml  2023-10-08 16:05:22.263126218 +0100
@@ -250,12 +250,12 @@
   end

 (* |gen_stmt| -- generate code for a statement *)
-let rec gen_stmt s =
+let rec gen_stmt s brkstack =
   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 brkstack) ss)>

       | Assign (v, e) ->
           if scalar v.e_type || is_pointer v.e_type then begin
@@ -280,10 +280,10 @@
           <SEQ,
             gen_cond test l1 l2,
             <LABEL l1>,
-            gen_stmt thenpt,
+            gen_stmt thenpt brkstack,
             <JUMP l3>,
             <LABEL l2>,
-            gen_stmt elsept,
+            gen_stmt elsept brkstack,
             <LABEL l3>>

       | WhileStmt (test, body) ->
@@ -294,7 +294,7 @@
             <LABEL l1>,
```

```
              gen_cond test l2 l3,
              <LABEL l2>,
-             gen_stmt body,
+             gen_stmt body (l3::brkstack),
              <JUMP l1>,
              <LABEL l3>>

@@ -302,7 +302,7 @@
            let l1 = label () and l2 = label () in
            <SEQ,
              <LABEL l1>,
-             gen_stmt body,
+             gen_stmt body (l2::brkstack),
              gen_cond test l2 l1,
              <LABEL l2>>

@@ -316,11 +316,14 @@
              <STOREW, gen_expr hi, address tmp>,
              <LABEL l1>,
              <JUMPC (Gt, l2), gen_expr var, <LOADW, address tmp>>,
-             gen_stmt body,
+             gen_stmt body (l2::brkstack),
              <STOREW, <BINOP Plus, gen_expr var, <CONST 1>>, gen_addr var>,
              <JUMP l1>,
              <LABEL l2>>

+       | Break n ->
+           <JUMP (List.nth brkstack (n-1))>
+
        | CaseStmt (sel, arms, deflt) ->
            (* Use one jump table, and hope it is reasonably compact *)
            let deflab = label () and donelab = label () in
@@ -330,13 +333,13 @@
            let gen_case lab (v, body) =
              <SEQ,
                <LABEL lab>,
-               gen_stmt body,
+               gen_stmt body brkstack,
                <JUMP donelab>> in
            <SEQ,
              gen_jtable (gen_expr sel) table deflab,
              <SEQ, @(List.map2 gen_case labs arms)>,
              <LABEL deflab>,
-             gen_stmt deflt,
+             gen_stmt deflt brkstack,
              <LABEL donelab>> in

    (* Label the code with a line number *)
@@ -373,7 +376,7 @@
     Regs.init ();
     let code0 =
       show "Initial code"
-        (Optree.canonicalise <SEQ, gen_stmt body, <LABEL !retlab>>) in
+      (Optree.canonicalise <SEQ, gen_stmt body [], <LABEL !retlab>>) in
     let code1 =
       if !optlevel < 1 then code0 else
         show "After simplification" (Jumpopt.optimise (Simp.optimise code0)) in
--- d1/lexer.mll        2023-10-08 15:18:06.767585400 +0100
+++ d2/lexer.mll        2023-10-08 16:05:22.263126218 +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); ("break", BREAK);
       ("and", MULOP And); ("div", MULOP Div); ("or", ADDOP Or);
       ("not", NOT); ("mod", MULOP Mod) ]

--- d1/parser.mly       2023-10-08 16:05:22.255126557 +0100
+++ d2/parser.mly       2023-10-08 16:05:22.263126218 +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                  BREAK

 %type <Tree.program>    program
 %start                  program
@@ -119,6 +120,8 @@
   | FOR name ASSIGN expr TO expr DO stmts END
                                         { let v = make_expr (Variable $2) in
                                           ForStmt (v, $4, $6, $8, ref None) }
+  | BREAK                               { Break 1 }
+  | BREAK NUMBER                        { Break $2 }
   | CASE expr OF arms else_part END     { CaseStmt ($2, $4, $5) } ;

 elses :
--- d1/tree.ml  2023-10-08 15:18:06.791585236 +0100
+++ d2/tree.ml  2023-10-08 16:05:22.263126218 +0100
@@ -39,6 +39,7 @@
   | WhileStmt of expr * stmt
   | RepeatStmt of stmt * expr
   | ForStmt of expr * expr * expr * stmt * def option ref
+  | Break of int
   | 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]
+    | Break n ->
+        fMeta "(BREAK $)" [fNum n]
     | CaseStmt (sel, arms, deflt) ->
         let fArm (lab, body) = fMeta "($ $)" [fExpr lab; fStmt body] in
         fMeta "(CASE $ $ $)" [fExpr sel; fList(fArm) arms; fStmt deflt]
--- d1/tree.mli 2023-10-08 15:18:06.803585155 +0100
+++ d2/tree.mli 2023-10-08 16:05:22.263126218 +0100
@@ -54,6 +54,7 @@
   | WhileStmt of expr * stmt
   | RepeatStmt of stmt * expr
   | ForStmt of expr * expr * expr * stmt * def option ref
+  | Break of int
   | CaseStmt of expr * (expr * stmt) list * stmt

 and expr =
```

```
--- d2/check.ml 2023-10-08 16:05:22.263126218 +0100
+++ d3/check.ml 2023-10-08 16:05:22.271125879 +0100
@@ -443,6 +443,19 @@
 let proc_symbol x =
   "_" ^ String.concat "." (List.rev_map spelling (x::!name_stack))

+(* |sort_descending| -- stable sort by descending score *)
+let sort_descending score xs =
+  let cf x y = score y - score x in
+  List.stable_sort cf xs
+
+(* |sorted_alloc| -- allocate in decreasing order of alignment *)
+let sorted_alloc alloc ds =
+  do_alloc alloc (sort_descending (fun d -> d.d_type.t_rep.r_align) ds)
+
+(* |max_alignment| -- maximum alignment for record fields *)
+let max_alignment defs =
+  List.fold_left (fun m d -> max m d.d_type.t_rep.r_align) 1 defs
+
 (* |check_typexpr| -- check a type expression, returning the ptype *)
 let rec check_typexpr te env =
   match te with
@@ -460,9 +473,10 @@
         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;
-        align max_align size;
-        let r = { r_size = !size; r_align = max_align } in
+        sorted_alloc (upward_alloc size) defs;
+        let almt = max_alignment defs in
+        align almt size;
+       let r = { r_size = !size; r_align = almt } in
         mk_type (RecordType defs) r
     | Pointer te ->
         let t =
@@ -554,7 +568,7 @@
   check_bodies env' ds;
   return_type := rt;
   check_stmt ss env' pre_alloc 0;
-  do_alloc (local_alloc fsize) !defs;
+  sorted_alloc (local_alloc fsize) !defs;
   align max_align fsize

 (* |check_bodies| -- check bodies of procedure declarations *)
```