Compilers (Sheet #2)

Marius Gavrilescu

2.1 Program:

```
begin
  s := 1;
  e := 20000;

  while s < e do
    m := (s + e + 1) div 2;
    if m * m < 200000000 then s := m else e := m - 1 end;
  end;

  print s;
end.
```

compiles to:

```
MODULE Main 0 0
IMPORT Lib 0
ENDHDR

PROC MAIN 0 0 0
!   s := 1;
CONST 1
STGW _s
!   e := 20000;
CONST 20000
STGW _e
JUMP L3
LABEL L2
!     m := (s + e + 1) div 2;
LDGW _s
LDGW _e
PLUS
CONST 1
PLUS
CONST 2
DIV
STGW _m
!     if m * m < 200000000 then s := m else e := m - 1 end;
LDGW _m
LDGW _m
TIMES
```

```
CONST 200000000
JLT L5
JUMP L6
LABEL L5
LDGW _m
STGW _s
JUMP L7
LABEL L6
LDGW _m
CONST 1
MINUS
STGW _e
LABEL L7
LABEL L3
!   while s < e do
LDGW _s
LDGW _e
JLT L2
JUMP L4
LABEL L4
!   print s;
LDGW _s
CONST 0
GLOBAL lib.print
PCALL 1
RETURN
END

GLOVAR _m 4
GLOVAR _s 4
GLOVAR _e 4
```

The comments in the assembly help us identify where each block of Keiko instructions comes from. First we initialize s and e with CONST/STGW instructions, then we enter the loop (which loads _s and _e and compares them to decide whether to run the loop body or to exit the loop).

The loop body computes $s + e + 1 \operatorname{div} 2$ and stores it as _m, then squares it, and compares it against 20000000 to decide whether to update _s or _e. At the end of the loop we load _s and print it.

It should be noted that this is very inefficient code: we have multiple labels at the same position, jumps that jump to the next instruction (therefore achieving nothing), places where we store a number to memory just to immediately reload it.

2.2
```
LDGW _x
CONST 1
PLUS
CONST 1
SWAP
DIV
```

To evaluate Binop(w, e_1, e_2), we have two cases:

If $d_1 > d_2$, we evaluate e_1 (takes d_1 space) which results in one element on the stack, then we evaluate e_2 (takes d_2 space plus the 1 element already on the stack) and finally we apply w. Uses d_1 space.

If $d_1 = d_2$ we do the same, except now this uses $d_1 + 1 = d_2 + 1$ space.

If $d_2 > d_1$, we evaluate e_2 (takes d_2 space) which results in one element on the stack, then we evaluate e_1 (takes d_1 space plus the 1 element already on the stack), then we swap the top two elements, and finally we apply w. Uses d_2 space.

```
let rec cost =
  function
  | Constant _ -> 1
  | Variable _ -> 1
  | Monop _ e  -> cost e
  | Binop _ e1 e2 ->
      let d1 = cost e1 in
      let d2 = cost e2 in
      min (max d1 (d2 + 1)) (max (d1 + 1) d2)
```

We can see Monops do not affect the cost, so we can ignore them and consider the expression as a binary tree where every leaf node is a Constant or Variable and every other node is a Binop with two children.

The cost of a non-leaf node can be computed by only looking at its children: if they have different costs, the cost of this node will be the maximum of the children's costs; if they have the same cost, the cost of this node will be one more than the cost of one of its children.

Now take the smallest tree with cost N+1. If any node in the tree has children with different costs, we can remove the child with smaller cost and replace the node with its other child. This operation would maintain the cost of the tree but reduce its size, thus contradicting our assumption. We can therefore deduce that the tree is in fact a perfect binary tree.

Now observe that perfect binary trees of depth d have 2^{d-1} leaves, and their cost is d. This means that the smallest tree with cost $N + 1$ has 2^N leaves. Thus if we have fewer than 2^N leaves (operands), the cost cannot be higher than N.

```
let rec gen_expr expr =
  function
  | Constant x        -> CONST x
  | Variable l        -> LDGW  l
  | Monop (m, e)      -> SEQ [gen_expr e; MONOP m]
  | Binop (b, e1, e2) ->
    if cost e1 >= cost e2 then
      SEQ [gen_expr e1; gen_expr e2; BINOP b]
    else
      SEQ [gen_expr e2; gen_expr e1; SWAP; BINOP b]
```

2.4 The code:

```
begin
  a := false;
  b := false;
  if a then
    x := 1;
  else
    if b then
      x := 2;
    else
      x := 3;
    end
  end
end.
```

compiles to:

```
MODULE Main 0 0
IMPORT Lib 0
ENDHDR

PROC MAIN 0 0 0
!   a := false;
CONST 0
STGW _a
!   b := false;
CONST 0
STGW _b
!   if a then
LDGW _a
CONST 0
JNEQ L2
JUMP L3
LABEL L2
!     x := 1;
CONST 1
STGW _x
JUMP L4
LABEL L3
!     if b then
LDGW _b
CONST 0
JNEQ L5
JUMP L6
LABEL L5
!       x := 2;
CONST 2
STGW _x
JUMP L7
```

```
LABEL L6
!       x := 3;
CONST 3
STGW _x
LABEL L7
LABEL L4
RETURN
END

GLOVAR _x 4
GLOVAR _b 4
GLOVAR _a 4
```

One inefficiency is that in the "else" case of the first "if" we unconditionally jump to the second "if" and from there we evaluate the second "if"'s condition. We can instead generate all condition checks at the beginning:

```
LDGW _a
CONST 0
JNEQ L2
LDGW _b
CONST 0
JNEQ L5
JUMP L6
```

There are other inefficiencies not specific to nested ifs. For example a `CJUMP C, X` followed by `JUMP Y; LABEL X` should be replaced by `CJUMP NOT(C), Y`. We can also replace any two labels pointing to the same place by a single label. Finally we can replace `LABEL X; JUMP Y` with `JUMP Y` and replacing all jumps to X with jumps to Y.

2.7 Abstract syntax could be `WhileCases of (expr * stmt) list`, `Loop of stmt` and `Exit`.

For the loop and exit statements, add the following to the production rules for stmt:

```
| LOOP stmts END                    { LoopStmt ($2) }
| EXIT                              { Exit }
```

Add to the `gen_stmt` function which now takes an extra `exit_lab` argument:

```
  | WhileCases cases ->
     let (preamble, rest) =
       List.fold_left
         (fun (preamble, rest) (cond, body) ->
           let tlab = label () in
           let flab = label () in
           let preamble =
             SEQ [gen_cond cond tlab flab; LABEL flab] :: preamble
           in
           let rest = SEQ [LABEL tlab; gen_stmt body exit_lab] :: rest in
           (preamble, rest))
```

```
            ([], [])
            cases
        in
        let lab1 = label () and lab2 = label () in
        SEQ
          [ LABEL lab1
          ; SEQ (reverse preamble)
          ; JUMP lab2
          ; SEQ (reverse rest)
          ; LABEL lab2
          ]
    | LoopStmt (body) ->
        let lab1 = label () and lab2 = label () in
        SEQ [LABEL lab1; gen_stmt body lab2; JUMP lab1; LABEL lab2]
    | Exit ->
        if dummy_exit_lab = exit_lab then
          failwith "Exit not allowed outside Loop statements";
        SEQ [JUMP exit_lab]
```

After peephole optimization to fix inefficiencies such as `JUMP a` followed immediately by `LABEL a` we get:

```
LABEL L1
LDLW 4
LDLW 8
JGT L2
LDLW 4
LDLW 8
JLT L3
JUMP L4
LABEL L2
LDLW 4
LDLW 8
MINUS
STLW 4
LABEL L3
LDLW 8
LDLW 4
MINUS
STLW 8
JUMP L1
LABEL L4
```

The loop statement compiles to:

```
LABEL L1
LDLW 4
LDLW 8
JGT L2
LDLW 4
```

```
LDLW 8
JLT L3
JUMP L4
LABEL L4
JUMP L5
LABEL L2
LDLW 4
LDLW 8
MINUS
STLW 4
LABEL L3
LDLW 8
LDLW 4
MINUS
STLW 8
JUMP L1
LABEL L5
```

The difference being the extra `JUMP L4; LABEL L4` added by the else: but this is also removed by the peephole optimiser. So in the end both programs compile to the same Keiko code without any extra peepopt rules.

2.8 Abstract syntax could be `Cond of expr * expr * expr`.

```
gen_expr:
  ...
  | Cond (cond, tpart, fpart) ->
    let lab1 = label () in
    let lab2 = label () in
    SEQ
      [ gen_cond cond lab1 lab2
      ; LABEL lab1
      ; gen_expr tpart
      ; LABEL lab2
      ; gen_expr fpart
      ]
```

No changes to gen_cond are needed.

For `(i >= 0) and (a[i] > x)` we will translate it to `if i > 0 then a[i] > x else false` and generate:

```
LDGW _i
CONST 0
JGEQ L1
JUMP L2
LABEL L1
! code for a[i] > x
LABEL L2
CONST 0
```

After peephole optimisation with the already-defined rules we get:

```
LDGW _i
CONST 0
JLT L2
! code for a[i] > x
LABEL L2
CONST 0
```

which is optimal code for short-circuiting and. So there is no need for new peepopt rules.

3.1
```
LDGW _q
LOADW
LDGW _s
PLUS
STGW _s
LDGW _q
CONST 4
OFFSET
LOADW
STGW _q
```

3.2 We add a new statement constructor in the Tree module, which is `Local of decl list * stmt`. We change the lexer and parser to parse the new syntax.

We modify the type checker (`check.ml`) such that when a Local statement is encountered we locally define the variables and recurse in the body. So we add to the match in check_stmt code like:

```
| Local (decls, body) ->
  let env' = check_decls decls env in
  check_stmt body env'
```

And we move the `check_decls` function (and its callees) above `check_stmt` to ensure everything still compiles.

To implement shadowing, we want to change `Dict.add_def` to simply overwrite definitions instead of raising Exit. This means our compiler will no longer fail on duplicate global variables (which seems reasonable behaviour), but if we specifically want to avoid this we can do so by modifying the entire chain of functions `check_decls, check_decl, add_def, Dict.define` to take an extra parameter saying whether we can overwrite definitions (this is fine in Local statements, not in global Decls).

So now the tree coming out of the type checker has all local variables pointing to all the right (global) places in memory. All we need to do is to modify `Kgen.gen_stmt` to treat `gen_stmt (Local (_decls, body))` as `gen_stmt body` and modify `translate` so it iterates over the tree and for each Local variable it finds it calls `gen_decl` on it (so a GLOVAR statement is generated for it).

3.4
```
let missing x = function
 | [] -> true
 | (y:ys) -> if x = y then false else missing x ys
```

```
;;

let check_vars_in_expr expr used_vars =
  match expr with
  | Constant _ -> ()
  | Variable x ->
    if missing x used_vars then
      failwithf "Unused variable %s" x
  | Monop (_, e) -> check_vars_in_expr e used_vars
  | Binop (_, e1, e2) ->
    check_vars_in_expr e1 used_vars;
    check-vars_in_expr e2 used_vars
;;

let rec vars_used_in stmt used_vars =
  match stmt with
  | Skip | Newline | Exit -> used_vars
  | Seq stmts ->
    List.fold_left
      (fun acc s -> vars_used_in s acc) used_vars
  | Assign (v, e) ->
    check_vars_in_expr (Variable v) used_vars;
    check_vars_in_expr e used_vars;
    v :: used_vars
  | Print e ->
    check_vars_in_expr e used_vars
  | IfStmt (test, thenpt, elsept) ->
    check_vars_in_expr test used_vars;
    (vars_used_in thenpt used_vars)
    @ (vars_used_in elsept used_vars)
  | WhileStmt (test, body) ->
    check_vars_in_expr test used_vars;
    vars_used_in body used_vars
;;

let translate (Program ss) =
  check_vars ss [];
  ...
;;
```

The logic is that `vars_used_in stmt used_vars` returns the list of variables used in stmt assuming used_vars are already used above. This function raises if any unused variable is encountered. The only time we add a variable to the list is when an Assign statement is encountered.

We cannot do a perfect job with static analysis because we cannot predict (for example) which branch of an if statement is taken. So if one branch assigns to `x` and the other does not, we cannot know for sure whether `x` has been assigned to or not. Best we can do here is to assume it has been assigned to. This means we will not catch some mistakes (false negative), while the opposite assumption means some otherwise valid programs will fail to compile (false positive), and false positives are clearly unacceptable here.