Languages and Compilation

Based on the Jean-Christophe Filliâtre's Courses
given at École Polytechnique & École Normale Supérieure

Lecture 11 - optimizing compiler (2/3)

Léon Gondelman

AALBORG UNIVERISITY | COPENHAGEN | 2025

we took as example a fragment of C language

```
int fact(int x) {
  if (x <= 1) return 1;
  return x * fact(x-1);
}
```

phase 1 : instruction selection

- replace C arithmetic operations with MIPS operations
- explicit memory access with constant offset over signed 16 bits

```
int fact(int x) {
  if (Mle x 1) return 1;
  return Mmul x fact((Maddi -1) x);
}
```

phase 2 : RTL (Register Transfer Language)

- from code as abstract syntax tree to **control-flow graph**
- **pseudo-registers** for function parameters and intermediate computations

```
%2 fact(%1)
  entry : L10
  exit  : L1
  locals:
  L10: %7 <- %1 --> L9
  L9: %8 <- 1   --> L8
  L8: ble %7 %8 --> L7, L6
```

```
L7: %2 <- 1                --> L1
L6: %3 <- %1               --> L5
L5: %6 <- %1               --> L4
L4: %5 <- addi -1 %6       --> L3
L3: %4 <- call fact(%5)    --> L2
L2: %2 <- mul %3 %4        --> L1
```

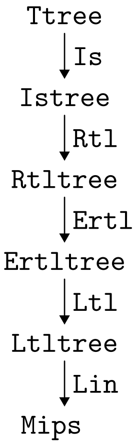
phase 3 : ERTL (Explicit Register Transfer Language)

- explicit calling conventions and instructions for handling stack frame

```
fact(1)
  entry : L19
  locals: %10, %11, %12
  L19: alloc_frame  --> L18
  L18: %10 <- $ra   --> L17
  L17: %11 <- $s0   --> L16
  L16: %12 <- $s1   --> L15
  L15: %1 <- $a0    --> L11
  L11: %8 <- %1     --> L10
  L10: %9 <- 1      --> L9
  L9: ble %8 %9  --> L8, L7
  L8: %2 <- 1       --> L1
  L7: %3 <- %1      --> L6
  L6: %6 <- %1      --> L5
```

```
L5: %7 <- 1           --> L4
L4: %5 <- sub %6 %7 --> L3
L3: goto L14
L14: $a0 <- %5        --> L13
L13: call fact        --> L12
L12: %4 <- $v0        --> L2
L2: %2 <- mul %3 %4 --> L1
L1: goto L25
L25: $v0 <- %2        --> L24
L24: $ra <- %10       --> L23
L23: $s0 <- %11       --> L22
L22: $s1 <- %12       --> L21
L21: delete_frame     --> L20
L20: return
```

1. **instruction selection**
2. **RTL (Register Transfer Language)**
3. **ERTL (Explicit Register Transfer Language)**
4. **LTL (Location Transfer Language)**
   4.1 **liveness analysis**
   4.2 **interference graph**
   4.3 **register allocation** using **graph coloring**
5. **linearization (assembly)**



the third phase is a transformation that turns RTL into **ERTL** (*Explicit Register Transfer Language*)

goal : make the **calling conventions** explicit, namely here

- the first four arguments are passed in `$a0`, `$a1`, `$a2`, `$a3`, and others on the stack
- the result is returned in `$v0`
- some registers are preserved by the callee (`$s0`, `$s1`, ...), others by the caller (`$v0`, `$a0`, ..., `$t0`, ..., `$ra`)

assume that the module `Register` describes physical registers as well

```
type t
...
val parameters: t list  (* for the first n arguments *)
val result: t           (* for the result *)
val ra: t
val callee_saved: t list

(* for syscall : *)
val a0: t
val v0: t
```

for RTL, we had

```
  | Ecall of register * ident * register list * label
```

in ERTL, we now have

```
  | Ecall of ident * int * label
```

i.e. we are only left with the name of the function to call, since new instructions will be inserted to load parameters into registers and stack, and to get the result from `$v0`
(we only keep the number k of parameters passed into registers (to be used in phase 4)

similarly, instructions `Emalloc` and `Eprintf` disappear, as we now can express them by introducing ERTL instruction

```
  | Esyscall of label
```

new instructions in the ERTL phase

other RTL instructions remain unchanged

however, we introduce some **new** instructions:

- to create and destroy the stack frame

```
| Ealloc_frame  of label
| Edelete_frame of label
```

  (note : we do not know yet the size of the stack frame)

- to load and store parameters using stack

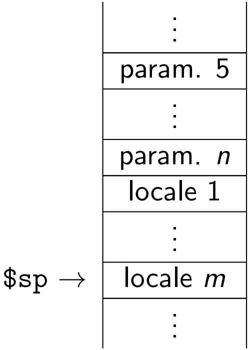
```
| Eget_stack_param of register * int * label
| Eset_stack_param of register * int * label
```

  (the integer here corresponds to the offset w.r.t. the top of the stack frame)

- explicit return instruction

```
| Ereturn
```

the stack frame is as follows:



the m local variables area will hold all the pseudo-registers that could not be allocated to physical registers; register allocation (phase 4) will determine the value of m

we do not change the structure of the control-flow graph; we simply **insert new instructions**

- at the beginning of each function, to
  - allocate the stack frame
  - save `$ra` and the *callee-saved* registers
  - copy the parameters into the corresponding pseudo-registers
- at the end of each function, to
  - copy the pseudo-register holding the result into `$v0`
  - restore `$ra` and the *callee-saved* registers
  - delete the stack frame
- around each function call, to
  - copy the pseudo-registers holding the parameters into `$a0`, ... and on the stack before the call
  - copy `$v0` into the pseudo-register holding the result after the call

we translate the RTL instructions into ERTL with a function

```
val instr: Rtltree.instr -> Ertltree.instr
```

few things change, apart from function calls that is, instructions `Ecall`, `Emalloc` and `Eprintf` recall that in RTL, the call is represented by

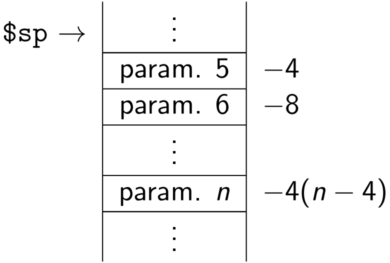
```
  | Rtltree.Ecall (r, x, rl, l) ->
```

where `r` is the pseudo-register receiving the result, `x` is the name of the function and `rl` is the list of pseudo-registers containing the arguments

we start by associating the first parameters to physical registers, i.e. to `Register.parameters`:

```
let assoc_formals formals =
  let rec assoc = function
    | [], _ -> [], []
    | rl, [] -> [], rl
    | r :: rl, p :: pl ->
        let a, rl = assoc (rl, pl) in (r, p) :: a, rl
  in
  assoc (formals, Register.parameters)
```

the parameters that are not associated with physical registers are passed on the stack; we thus store them at relative positions -4, -8, etc. w.r.t. $sp



that is, we make a choice that it's the callee that will be in charge of allocating the stack frame (parameters + locals) by a simple subtraction over $sp we provide

```
let move src dst l = generate (Emunop (dst, Mmove, src, l))
let set_stack r n l = generate (Eset_stack_param (r, n, l))
```

so that the call is realized as follows:

```
| Rtltree.Ecall (r, x, rl, l) ->
    let frl, fsl = assoc_formals rl in
    let n = List.length frl in
    let l = generate (Ecall (x, n, move Register.result r l)) in
    let ofs = ref 0 in
    let l = List.fold_left
      (fun l t -> ofs := !ofs - word_size; set_stack t !ofs l)
      l fsl
    in
    let l = List.fold_right (fun (t, r) l -> move t r l) frl l in
    Egoto l
```

the RTL code

```
L3: %4 <- call fact(%5)   --> L2
```

is translated into the ERTL code

```
L3: goto L14
L14: $a0 <- %5      --> L13
L13: call fact      --> L12
L12: %4 <- $v0      --> L2
```

recall that for `malloc`, we use the system call 9

```
  | Rtltree.Emalloc (r, n, l) ->
     Econst (Register.a0, n, generate (
     Econst (Register.v0, 9, generate (
     Esyscall (
     move Register.v0 r l)))))
```

similarly for `printf`,
we use system calls 1 (`print_int`) and 11 (`print_char`)

```
  | Rtltree.Eprintf (r, l) ->
      Econst (Register.v0, 1, (
      move r Register.a0 (generate (
      Esyscall (generate (
      Econst (Register.a0, 10, generate (
      Econst (Register.v0, 11, generate (
      Esyscall l)))))))))
```

it remains to translate each function

RTL

```
type deffun = {
  fun_name    : ident;
  fun_formals: register list;
  fun_result : register;
  fun_locals : Register.set;
  fun_entry   : label;
  fun_exit    : label;
  fun_body    : instr Label.map;
}
```

ERTL

```
type deffun = {
  fun_name    : ident;
  fun_formals: int; (* num *)

  fun_locals : Register.set;
  fun_entry   : label;

  fun_body    : instr Label.map;
}
```

we start by reconstructing the graph for the function body (which is a graph computed from a fresh exit label to some entry label) :

```
let deffun f =
  let () = graph := Label.M.empty in
  let () = Label.M.iter
              (fun l i -> let i = instr i in
                          graph := Label.M.add l i !graph)
              f.Rtltree.fun_body in
  ...
```

we associate a fresh pseudo-register to each physical register that has to be saved *i.e.* `$ra` and the `callee_saved` registers

```
let deffun f =
  let () = graph := Label.M.empty in
  let () = Label.M.iter
              (fun l i -> let i = instr i in
                          graph := Label.M.add l i !graph)
              f.Rtltree.fun_body in
  let svl = List.map (fun r -> Register.fresh(), r)
             (Register.ra :: Register.callee_saved) in
  ...
```

at the entry of the function (i.e. before the entry to the call), we must

- allocate its stack frame with `Ealloc_frame`
- save the callee-saved registers (list `svl`)
- copy the arguments into their pseudo-registers (`formals`)

```
let fun_entry svl formals entry =
  let frl, fsl = assoc_formals formals in
  let ofs = ref 0 in
  let l = List.fold_left
    (fun l t -> ofs := !ofs - word_size; get_stack t !ofs l)
    entry fsl
  in
  let l = List.fold_right (fun (t, r) l -> move r t l) frl l in
  let l = List.fold_right (fun (t, r) l -> move r t l) svl l in
  generate (Ealloc_frame l)
```

(note : the offset of `get_stack` is computed as for `set_stack` here) at function exit, we

- delete the stack frame
- restore the saved registers
- copy the pseudo-register holding the result in `$v0`

```
let fun_exit svl retr exitl =
  let l = generate (Edelete_frame (generate Ereturn)) in
  let l = List.fold_right (fun (t, r) l -> move t r l) svl l in
  let l = move retr Register.result l in
  graph := Label.M.add exitl (Egoto l) !graph
```

(now, the fresh dummy `exitl` label finally makes sense)

so we compute label for the function entry and exit
given the code on the previous slides, and we are done

```
let deffun f =
  let () = graph := Label.M.empty in
  let () = Label.M.iter (fun l i -> let i = instr i in
                          graph := Label.M.add l i !graph)
             f.Rtltree.fun_body in
  let svl = List.map (fun r -> Register.fresh(), r)
             (Register.ra :: Register.callee_saved) in
  let entry =
     fun_entry svl f.Rtltree.fun_formals f.Rtltree.fun_entry in
  let () = fun_exit svl f.Rtltree.fun_result f.Rtltree.fun_exit in
  { fun_name = f.Rtltree.fun_name;
    fun_formals = List.length f.Rtltree.fun_formals;
    fun_locals = locals;
    fun_entry = entry;
    fun_body = !graph; }
```

```
fact(1)
  entry : L19
  locals: %10, %11, %12
  L19: alloc_frame  --> L18
  L18: %10 <- $ra   --> L17
  L17: %11 <- $s0   --> L16
  L16: %12 <- $s1   --> L15
  L15: %1 <- $a0    --> L11
  L11: %8 <- %1     --> L10
  L10: %9 <- 1      --> L9
  L9: ble %8 %9  --> L8, L7
  L8: %2 <- 1       --> L1
  L7: %3 <- %1      --> L6
  L6: %6 <- %1      --> L5
```

```
L5: %7 <- 1          --> L4
L4: %5 <- sub %6 %7 --> L3
L3: goto L14
L14: $a0 <- %5       --> L13
L13: call fact       --> L12
L12: %4 <- $v0       --> L2
L2: %2 <- mul %3 %4 --> L1
L1: goto L25
L25: $v0 <- %2       --> L24
L24: $ra <- %10      --> L23
L23: $s0 <- %11      --> L22
L22: $s1 <- %12      --> L21
L21: delete_frame    --> L20
L20: return
```

(here we assume that the only *callee-saved* registers are $s0 and $s1) this is far from being what we think is a good MIPS code for the factorial

at this point, we have to understand that

- register allocation (phase 4) will try to match physical registers to pseudo-registers to minimize the use of the stack and certain instructions; e.g., if we map `%11` to `$s0`, we can get rid of instructions at `L17` et `L23`
- the code is not linearized yet (the graph is simply printed in some arbitrary order)

another example, a function with more than 4 arguments

```
int many(int a, int b, int c, int d, int e, int f) {
  if (a == 0) return b; else return many(b, c, d, e, f, a);
}
```

```
many(6)
entry : L31
locals: %17, %18, %19
L31: alloc_frame->L30
L30: %17<-$ra    ->L29
L29: %18<-$s0    ->L28
L28: %19<-$s1    ->L27
L27: %1<-$a0     ->L26
L26: %2<-$a1     ->L25
L25: %3<-$a2     ->L24
L24: %4<-$a3     ->L23
L23: %6<-st(-8)  ->L22
L22: %5<-st(-4)  ->L13
```

```
L13: %15<-%1  ->L12
L12: %16<-0   ->L11
L11: %14<-seq %15 %16
              ->L10
L10: beqz %14->L8,L9
L8: %8<-%2    ->L7
L7: %9<-%3    ->L6
L6: %10<-%4   ->L5
L5: %11<-%5   ->L4
L4: %12<-%6   ->L3
L3: %13<-%1   ->L2
L2: goto L21
L21: $a0<-%8 ->L20
L20: $a1<-%9 ->L19
```

```
L19: $a2<-%10    ->L18
L18: $a3<-%11    ->L17
L17:st(-8)<-%13 ->L16
L16:st(-4)<-%12 ->L15
L15: call many   ->L14
L14: %7<-$v0     ->L1
L1: goto L37
L37: $v0<-%7     ->L36
L36: $ra<-%17    ->L35
L35: $s0<-%18    ->L34
L34: $s1<-%19    ->L33
L33:delete_frame->L32
L32: return
L9: %7<-%2       ->L1
```

the next phase translates ERTL to **LTL** (Location Transfer Language)

the goal is to get rid of pseudo-registers, replacing them with

- physical registers preferably
- stack locations otherwise

this is called **register allocation**

register allocation is complex, and decomposed into several steps

1. we perform a **liveness analysis**
   - it tells when the value contained in a pseudo-register is needed for the remaining of the computation
2. we build an **interference graph**
   - it tells what are the pseudo-registers that cannot be mapped to the same location
3. we allocate registers using a **graph coloring**
   - it maps pseudo-registers to physical registers or stack locations