Languages and Compilation

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

Lecture 10 - optimizing compiler (1/3)

Léon Gondelman

AALBORG UNIVERISITY | COPENHAGEN | 2025

compilation phases

$$\begin{tikzcd}
\text{source} \arrow[d] \\
\boxed{\text{lexical analysis}} \arrow[d] \\
\text{stream of tokens} \arrow[d] \\
\boxed{\text{parsing}} \arrow[d] \\
\text{abstract syntax tree} \arrow[d] \\
\boxed{\text{semantic analysis}} \arrow[d] \\
\text{abstract syntax + symbol table}
\end{tikzcd}$$

frontend

$$\begin{tikzcd}
{} \arrow[d] \\
\boxed{\textbf{code production}} \arrow[d] \\
\text{assembly code} \arrow[d] \\
\boxed{\text{assembler (}\texttt{as}\text{)}} \arrow[d] \\
\text{machine language} \arrow[d] \\
\boxed{\text{linking (}\texttt{ld}\text{)}} \arrow[d] \\
\text{executable}
\end{tikzcd}$$

backend

today

goal for the next lectures:

- how the backend of a compiler works (from AST to assembly)
- writing an optimizing compiler

lab sessions:

- today – help with the projects
- next week – handwritten exercises related to the backend

remark

until now, we did not use MIPS assembly efficiently:

- we didn't use much registers, while there are 32
  - function arguments and local variables were put on the stack
  - intermediate computations were on the stack as well

- we didn't exploit the capacities of MIPS instructions
  - example : we didn't use addi which allows to add a constant

```
        addi  $t0, $t1, 12
```

(we used instead li and a temporary value often stored on a stack)

compiler phases

emitting optimized code in a single pass is doomed to failure

we decompose code production into **several phases**

1. instruction selection
2. RTL (*Register Transfer Language*)
3. ERTL (*Explicit Register Transfer Language*)
4. LTL (*Location Transfer Language*)
5. code linearization

compiler phases = many ASTs

the starting point: the abstract syntax tree output by the type checker

$$\begin{tikzcd}
\texttt{Ttree} \arrow[d, "\texttt{Is}"] \\
\texttt{Istree} \arrow[d, "\texttt{Rtl}"] \\
\texttt{Rtltree} \arrow[d, "\texttt{Ertl}"] \\
\texttt{Ertltree} \arrow[d, "\texttt{Ltl}"] \\
\texttt{Ltltree} \arrow[d, "\texttt{Lin}"] \\
\texttt{Mips}
\end{tikzcd}$$

Mini-C

this compiler architecture is independent of the programming paradigm (imperative, functional, object oriented, etc.)

we illustrate it on a small fragment of **C** with

- integers (type `int`)
- heap-allocated structures (using `malloc`), only pointers to structures, no pointer arithmetic
- functions
- a "primitive" to print an integer `printf("%d\n",`*e*`)`

Mini-C

$$
\begin{array}{rcl}
E & \rightarrow & n \\
  & \mid & L \\
  & \mid & L \texttt{ = } E \\
  & \mid & E \ \mathit{op}\ E \mid \texttt{-}\ E \mid \texttt{!}\ E \\
  & \mid & x\texttt{(}E,\ldots,E\texttt{)} \\
  & \mid & \texttt{malloc(sizeof(struct } x\texttt{))} \\
L & \rightarrow & x \\
  & \mid & E\texttt{->}x \\
\mathit{op} & \rightarrow & \texttt{==} \mid \texttt{!=} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \\
  & \mid & \texttt{\&\&} \mid \texttt{||} \mid \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \\
D & \rightarrow & V \\
  & \mid & T\ x(T\ x,\ldots,T\ x)\ B \\
  & \mid & \texttt{struct } x\ \{V \ldots V\}\texttt{;}
\end{array}
$$

$$
\begin{array}{rcl}
S & \rightarrow & E\texttt{;} \\
  & \mid & \texttt{if } (E)\ S \\
  & \mid & \texttt{if } (E)\ S \texttt{ else } S \\
  & \mid & \texttt{while } (E)\ S \\
  & \mid & \texttt{return } E\texttt{;} \\
  & \mid & \texttt{printf("\%d\textbackslash n",}E\texttt{);} \\
  & \mid & B \\
B & \rightarrow & \{\ V \ldots V\ S \ldots S\ \} \\
V & \rightarrow & \texttt{int } x,\ldots,x\texttt{;} \\
  & \mid & \texttt{struct } x\ \texttt{*}x,\ldots,\texttt{*}x\texttt{;} \\
T & \rightarrow & \texttt{int} \mid \texttt{struct } x\ \texttt{*} \\
P & \rightarrow & D \ldots D
\end{array}
$$

examples

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

```
struct list { int val; struct list *next; };

int print(struct list *l) {
  while (l != 0) {
    printf("%d\n", l->val);
    l = l->next;
  }
  return 0;
}
```

starting point

we assume that type checking is done, so the resulting tree is as follows:

```
type file = { gvars: decl_var list; funs: decl_fun list; }
and decl_var = typ * ident
and decl_fun = {
  fun_typ:     typ;            fun_name: ident;
  fun_formals: decl_var list; fun_body: block; }
and block = decl_var list * stmt list
and stmt =
  | Sskip
  | Sexpr   of expr
  | Sif     of expr * stmt * stmt
  | Swhile  of expr * stmt
  | Sblock  of block
  | Sreturn of expr
  | Sprintf of expr
```

starting point

in the expressions, we distinguished between local and global variables, and their names are unique; field access is done now using indices.

```
and expr = { expr_node: expr_node; expr_typ: typ }
and expr_node =
  | Econst         of int
  | Eaccess_local  of ident
  | Eassign_local  of ident * expr
  | Eaccess_global of ident
  | Eassign_global of ident * expr
  | Eaccess_field  of expr * int    (* index of the field *)
  | Eassign_field  of expr * int * expr
  | Eunop          of unop * expr                  (* - ! *)
  | Ebinop         of binop * expr * expr  (* + - == etc. *)
  | Ecall          of ident * expr list
  | Emalloc        of structure
```

Phase 1: instruction selection

the first phase is **instruction selection**
this is where architecture-specific instructions enter the picture

goal :

- replace C arithmetic operations with MIPS operations
- replace structure field access with explicit memory access with MIPS `lw` and `sw` operations

arithmetic operations

naively, we can simply translate each C arithmetic operation with the corresponding MIPS operation

however, MIPS provides us with better instructions in some cases, notably

- addition of a register and a 16-bit signed constant
- bit shifting to the left or to the right, corresponding to a multiplication or a division by a power of 2
- comparison of a register and a 16-bit signed constant

partial evaluation

beside, it is advisable to perform as much evaluation as possible during compilation (partial evaluation)

examples: we can simplify

- $(1 + e_1) + (2 + e_2)$ into $e_1 + e_2 + 3$
- $e + 1 < 10$ into $e < 9$
- $!(e_1 < e_2)$ into $e_1 \geq e_2$
- $0 \times e$ into 0, but only if e is **pure** *i.e.* without side effects

new abstract syntax

we define new AST for the result of the instruction selection phase

`Ttree.mli` (before)

```
type expr =
  ...
type stmt =
  ...
type file =
  ...
```

`Istree.mli` (after)

```
type expr =
  ...
type stmt =
  ...
type file =
  ...
```

the goal is to implement functions

```
    val expr   : Ttree.expr -> Istree.expr
    val stmt   : Ttree.stmt -> Istree.stmt
    val program: Ttree.file -> Istree.file
```

new abstract syntax

operations are now those of MIPS

```
type munop = Mmove | Maddi of int | Mseqi of int | ...
type mbinop = Madd | Msub | Mmul | Mdiv | Meq | Mneq | Mlt ..
```

and they replace the operations of C

```
type expr =
  | Emunop  of munop * expr                (* replaces Eunop *)
  | Embinop of mbinop * expr * expr       (* replaces Ebinop *)
  | ...
```

we keep however && et || for now

```
  | Eand of expr * expr
  | Eor  of expr * expr
```

smar constructors

to implement partial evaluation, we can use **smart constructors**

rather than write `Embinop (Madd, e1, e2)` directly, we define a function

```
    val mk_add: expr -> expr -> expr
```

that performs possible simplifications and behaves as the corresponding constructor otherwise

smart constructor for addition

```
let rec mk_add e1 e2 = match e1, e2 with
  | Econst n1, Econst n2 ->
      Econst (n1 + n2)
  | e, Econst 0 | Econst 0, e ->
      e
  | Emunop (Maddi n1, e), Econst n2
  | Econst n2, Emunop (Maddi n1, e) ->
      mk_add (Econst (n1 + n2)) e
  | e, Econst n | Econst n, e when is16op n ->
      Emunop (Maddi n, e)
  | _ ->
      Embinop (Madd, e1, e2)
```

with

```
let is16op n = -32768 <= n && n <= 32767
```

termination and correctness

two aspects are crucial in regard of those simplifications

- the semantics of the programs must be preserved
  - example : if an order of evaluation left/right is specified, we cannot simplify $(0 - e_1) + e_2$ into $e_2 - e_1$ when e_1 or e_2 is not pure
- simplification function must terminate
  - we need to find a positive measure over expressions that strictly decreases at each recursive call of **smart constructor**

translation

```
let rec expr e = match e.Ttree.expr_node with
  | Ttree.Ebinop (Badd, e1, e2) ->
      mk_add (expr e1) (expr e2)
  | Ttree.Ebinop (Bsub, e1, e2) ->
      mk_sub (expr e1) (expr e2)
  | Ttree.Eunop (Ptree.Unot, e) ->
      mk_not (expr e)
  | Ttree.Eunop (Ptree.Uminus, e) ->
      mk_sub (Econst 0) (expr e)
  | ...
```

and its a morphism for other constructions (`Eaccess_local`, `Eaccess_global`, `Ecall`, etc.)

memory access

instruction selection also introduces explicit memory access

a memory address is given by an expression together with a constant offset over signed 16 bits

```
type expr =
  | ...
  | Eload  of expr * int
  | Estore of expr * int * expr
```

memory access

in our case, it is the access to the structure fields that are turned into memory accesses

and here we are in a simple case where each field is exactly one word long

therefore we have

```
let rec expr e = match e.Ttree.expr_node with
  | ...
  | Ttree.Eaccess_field (e, n) ->
      Eload (expr e, n * word_size)
  | Ttree.Eassign_field (e1, n, e2) ->
      Estore (expr e1, n * word_size, expr e2)
```

with

```
let word_size = 4    (* 32 bits architecture *)
```

statements and functions

instruction selection is a morphism over statements (`if`, `while`, etc.)

for functions, we erase types (not needed anymore) and we gather all variables at the function level (type checking made all variables distinct)

```
type deffun = {
  fun_name   : ident;
  fun_formals: ident list;
  fun_locals : ident list;
  fun_body   : stmt list;
}

type file = {
  gvars: ident list;
  funs : deffun list;
}
```

phase 2: RTL

the next phase transforms the code to the language **RTL** (*Register Transfer Language*)

goal:

- get rid of the tree structure of expressions and statements, in favor of a **control-flow graph** (CFG), to ease further phases; in particular, we make no distinction between expressions and statements anymore

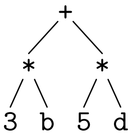
- introduce **pseudo-registers** to hold function parameters and intermediate computations; there are infinitely many pseudo-registers, that will later be either MIPS registers or stack locations

example

let us consider the C expression

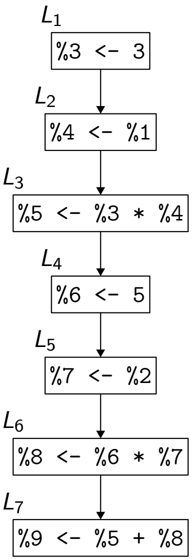
```
     3*b + 5*d
```

that is the syntax tree



let us assume that b is in the pseudo-register %1 and d in the pseudo-register %2

then we build the following graph:



pseudo-registers

we define a module Register for the pseudo-registers

```
type t

val fresh: unit -> t

module S: Set.S with type elt = t
```

CFG representation

we also define a module `Label` for the labels representing CFG vertices

```
type t

val fresh: unit -> t

module M: Map.S with type key = t
```

and the CFG is a map from labels (program points) to RTL instructions

```
  type graph = instr Label.M.t
```

RTL instructions

conversely, each RTL instruction lists the label(s) of the next instructions

```
type instr =
  | Econst of register * int * label
  | ...
```

the instruction `Econst(`r`, `n`, `l`)` meaning “store the value n in the pseudo-register r and transfer the control to the label l”

other instructions

similarly for global variables, memory accesses, `malloc` and `printf`:

```
type instr =
  ...
  | Eaccess_global of register * ident * label
  | Eassign_global of register * ident * label

  | Eload  of register * register * int * label
  | Estore of register * register * int * label

  | Emalloc of register * int * label
  | Eprintf of register * label
```

arithmetic operations

arithmetic operations now use pseudo-registers

```
type instr =
  ...
  | Emunop  of
      register * munop * register * label
  | Embinop of
      register * mbinop * register * register * label
```

building the CFG

to build the CFG, we store it temporarily in a reference

```
let graph = ref Label.M.empty
```

and we define a function to add an edge to the graph

```
let generate i =
  let l = Label.fresh () in
  graph := Label.M.add l i !graph;
  l
```

building the CFG

we build a separate CFG for each function, with its own pseudo-registers (**intra**procedural analysis)

we build the CFG from bottom to top, which means we always know the label of the continuation (the next instructions)

translating expressions

we then translate the expressions using a function

```
val expr: register -> expr -> label -> label
```

which takes as arguments

- the destination register to receive the value of the expression
- the expression to be translated
- the output label for the continuation of the computation

and returns the label of the entry point for the evaluation of this expression

we thus build the graph starting from the end of each function

translating expressions

the translation of expressions is pretty straightforward

```
let rec expr destr e destl = match e with
 | Istree.Econst n ->
     generate (Econst (destr, n, destl))
```

when necessary, we introduce fresh pseudo-registers

```
| Istree.Embinop (op, e1, e2) ->
   let pr1 = Register.fresh () in
   let pr2 = Register.fresh () in
   let dl1 = (generate (Embinop (destr, op, pr1, pr2, destl))) in
   let dl2 = expr pr2 e2 dl1 in
   expr pr1 e1 dl2
```

translating expressions

for local variables, we set up a table where each variable is mapped to a fresh pseudo-register

```
  | Istree.Eaccess_local x ->
      let rx = Hashtbl.find locals x in
      generate (Emunop (destr, Mmove, rx, destl))
```

where `Mmove` corresponds to the MIPS operation move

etc.

to translate operations `&&`, `||`, and for `if` and `while` statements, we use RTL **branching** instructions

```
type instr =
  ...
  | Emubranch of mubranch * register * label * label
  | Embbranch of
       mbbranch * register * register * label * label
  | Egoto     of label
```

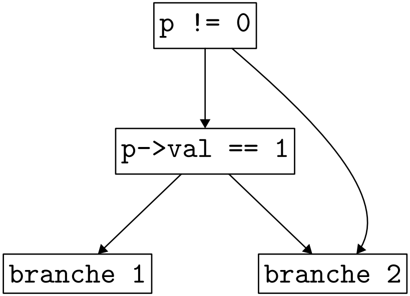
with

```
type mubranch = Mbeqz | ...

type mbbranch = Mblt | Mble | ...
```

example

```
if (p != 0 && p->val == 1)
  ...branch 1...
else
  ...branch 2...
```



(the four blocs here are actually sub-graphs)

translating a condition

to translate a condition, we define a function

```
val condition: expr -> label -> label -> label
```

the two labels that are passed to the function correspond to the continuation in the case where the condition holds or fails respectively

and we return the entry label for the evaluation of the conditional itself

translating a condition

```
let rec condition e truel falsel = match e with
  | Istree.Eand (e1, e2) ->
      condition e1 (condition e2 truel falsel) falsel
  | Istree.Eor (e1, e2) ->
      condition e1 truel (condition e2 truel falsel)
  | Istree.Embinop (Mle, e1, e2) ->
      let tmp1 = Register.fresh () in
      let tmp2 = Register.fresh () in
      expr tmp1 e1
      (expr tmp2 e2
      (generate (Embbranch (Mble, tmp1, tmp2, truel, falsel))
  | e ->
      let tmp = Register.fresh () in
      expr tmp e
      (generate (Emubranch (Mbeqz, tmp, falsel, truel)))
```

translating instructions

to translate `return`, we pass

- a pseudo-register `retr` to receive the function result
- and a label `exitl` corresponding to the function exit

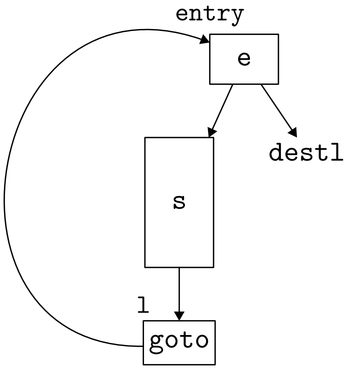
and we pass a label `destl` corresponding to the continuation of the computation for other instructions

```
let rec stmt retr s exitl destl = match s with
  | Istree.Sskip ->
      destl
  | Istree.Sreturn e ->
      expr retr e exitl
  | Istree.Sif (e, s1, s2) ->
      condition e
        (stmt retr s1 exitl destl)
        (stmt retr s2 exitl destl)
  | ...
```

loop

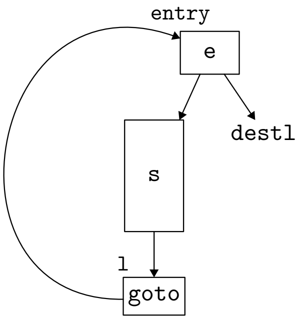
for a `while` loop, we have to build a cycle in the CFG

```
while (e) {
  ...s...
}
```



loop

```
let rec stmt retr s exitl destl = match s with
  | ...
  | Istree.Swhile (e, s) ->
      let l = Label.fresh () in
      let entry = condition e (stmt retr s exitl l) destl in
      graph := Label.M.add l (Egoto entry) !graph;
      entry
```



functions

the formal parameters of a function and its result now are pseudo-registers

```
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;
}
```

as well as actual parameters and result in a function call

```
type instr =
  ...
  | Ecall of register * ident * register list * label
```

translating a function

translating a function involves the following steps:

1. we allocate fresh pseudo-registers for its parameters, its result, and its local variables
2. we start with an empty graph
3. we pick a fresh label for the function exit
4. we translate the function body to RTL code, and the output is the entry label in the CFG

example

let us consider the factorial again

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

we get

```
%2 fact(%1)
  entry : L11
  exit  : L1
  locals:
  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: %4 <- call fact(%5)       --> L2
L2: %2 <- mul %3 %4           --> L1
```

next session

phase 3 : from RTL to **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`)

phase 4 : from **ERTL** to **LTL** (*Location Transfer Language*)
goal: get rid of pseudo-registers, replacing them with

- physical registers preferably
- stack locations otherwise

this is called **register allocation**