The Architecture of the Quick C-- Compiler

Christian Lindig <lindig@eecs.harvard.edu>

Harvard University

- Compiles C-- to (textual or binary) assembly code.
- Cross compiler with multiple back ends in one binary.
- Current targets: symbolic dummy target, SPARC.
- Back end controlled by built-in Lua interpreter.
- RTL-based translation technique.
- Graph-coloring and linear-scan register allocator.
- Implemented in O'Caml 3.04.
- Extensively documented: literate program in NoWEB.
- Circa 100 modules, about 27 000 lines of hand-written code.
- Machine-generated code in front and back ends: Lex, Yacc, ASDL, Burg, λ-RTL.
- Implementation so far tries to avoid mutable state.

`Scan`	records line breaks in `Srcmap.map`
↓	
`Parse.token`	
↓	
`Parse`	Yacc generated parser
↓	
`Ast.program`,	`Ast.program` generated from ASDL spec.
`Srcmap.map`	source code positions
↓	check static semantics, populate fat env.
`Elab`	evaluate constant expressions (`Rtleval`)
↓	propagates errors (`Error`)
`Ast.program`, `Srcmap.map`,	
`Fenv.Dirty.env`	

The Fat Environment (Fenv)

- Provides meaning for names.
- Two flavors: *clean* and *dirty*.
- Dirty environment can contain errors.
- Mostly functional, but few exceptions.

```
type  kind =
    | Proc      of Symbol.t * scope partial
    | Code      of Symbol.t
    | Data      of Symbol.t
    | Stack     of Rtl.exp option

type denotation =
    | Constant  of Bits.bits
    | Label     of kind
    | Import    of string option * Symbol.t
    | Register  of register
    | Continuation  of string * Symbol.t

type ventry = (* value entry *)
    Srcmap.rgn * (denotation * Types.ty) info
```

A simple value is either `Ok of 'a` or `Error`.

A complex value is `Error`, if one of its internal values is `Error`, and `Ok` otherwise.

Combinators allow to make regular functions error-aware. Heavily used in module `Elab`.

```
val ematch      : 'a error -> ('a -> 'b) -> 'b error
val ematchPair  : 'a error * 'b error -> ('a * 'b -> 'c) -> 'c error
module Raise :
  sig
    val option : 'a error option                -> 'a option error
    val list   : 'a error list                  -> 'a list error
    val pair   : 'a error * 'b error            -> ('a * 'b) error
  end
```

Universal notation for machine instructions: one data structure can describe any machine instruction.

Four important conceptual parts:

1. Constants (boolean, bit vectors, labels).
2. Expressions, denote values and addresses.
3. Locations, unify memory and registers.
4. Operators, primitive function on values.

Module `Rtl` provides two views on RTLs.

Public: abstract representation, constructor functions.

```
bits  : Bits.bits -> width -> exp
late  : string    -> width -> exp
fetch : loc       -> width -> exp
```

Private: exposed representation for pattern matching.

```
type exp =
    | Const of const
    | Fetch of loc * width
    | App   of opr * exp  list
```

Conceptually, values are bit vectors whose interpretation is up to the operators using them.

Practically, we support up to 64 bits as signed and unsigned integers. Floating point value support is planned, but difficult.

```
type t
type width = int
val width : t -> width

module S: sig  (* signed *)
    val of_int:     int       -> width -> t
    val of_native:  nativeint -> width -> t
    val of_string:  string    -> width -> t
    val to_int:     t -> int
    val to_native:  t -> nativeint
end
module U: sig (* unsigned *) ... end
```

1. Creation. `Ast3ir` translates AST to *intermediate representation* (ASM/CFG).
2. Copy in/out. `Copyinout` expands call nodes in CFG to sequence of assignments.
3. Variable Placement. `Placevar` replaces variables in RTLs by temporaries.
4. Code Expansion. Every RTL represents a machine instruction. Yes, but . . .
5. Liveness Analysis. `Live` annotates CFG nodes with sets of live registers.
6. Register Allocation. `Colorgraph` removes temporaries in RTLs.
7. Substitution. `Ast3ir` replaces *late compile time constants* in RTLs.
8. Constant Folding. RTLs are simplified by module `Ast3ir`.
9. Linearization. `Cfg4` linearizes the CFG.
10. Recognition. A recognizer matches and emits instructions.

- `Ast3ir` controls compilation of a translation unit (a file).
- A compilation unit contains data and procedures. Everything except procedures is immediately emitted using the assembler interface `Asm3`.

```
method import    : string -> Symbol.t
method align     : int -> unit
method zeroes    : int -> unit
method value     : Bits.bits -> unit
```

- Procedures are translated into a control-flow graph (CFG) and pass the phases outlined before. Finally, they are emitted as well:

```
method cfg_instr : Cfg4.cfg -> Symbol.t -> unit
```

In summary: only procedures have a tangible intermediate representation. For the back end, a procedure is packed up as a `Proc.t` value.

Back End Procedure Representation (Proc.t)

```
type t =
    { symbol:   Symbol.t            (* of procedure *)
    ; cc:       Target2.cc          (* calling convention                 *)
    ; target:   Target2.t           (* target of this procedure           *)
    ; temps:    Talloc.Multiple.t   (* allocator for temporaries          *)
    ; cfg:      Cfg4.cfg            (* control-flow graph                 *)
    ; incoming: Block.t             (* stack - incoming area              *)
    ; outgoing: Block.t             (* stack - outgoing area              *)
    ; stackd:   Block.t             (* stack - user stack data            *)
    ; conts:    Block.t             (* pairs of pointers for conts *)
    ; priv:     Automaton2.t        (* stack - spill slots etc - still open *)
    ; eqns:     Const2.t list       (* eqns for compile time consts *)
    }
```

Together with RTLs, our most important data structure.

- Imperative: nodes are objects, edges are pointers.
- Every node carries an RTL.
- Internal implementation is object-oriented.
- CFG is built bottom up by `Ast3ir`.
- Functions for mutation and traversal.

```
mk: target:Target2.t -> nop:Rtl.rtl -> cfg
entry:      cfg -> node
exit:       cfg -> node
gm_assign:  cfg -> Rtl.rtl -> succ:node
                           -> node
gm_return:  cfg -> Rtl.rtl -> int * int
                           -> node
gm_goto:    cfg -> Rtl.rtl -> node list
                           -> node
next:       node -> node option
instr:      node -> Rtl.rtl
```

A stack frame is an algebraic composition of memory blocks, represented by `Block.t`.

```
type t
type placement = High | Low
val mk         : base:Rtl.exp -> size:int -> alignment:int -> t
val cat        : t -> t -> t
val overlap    : placement -> t -> t -> t
```

Sources of blocks: stack data, initialized data, global variables, slot allocation, calling conventions.

Most blocks are allocated piecewise using an `Automaton2.t` value that encapsulates the allocation policy.

Automata for Resource Allocation (Automaton2)

Abstraction for a pool of abstract *locations* that can be requested.

Locations can be complex, like two registers, or a register and a memory cell.

```
type t
type loc

val mk       : spec -> address:Rtl.exp -> t
val allocate : t -> width:int -> hint:(string option) -> loc
val fetch    : loc -> width:int -> Rtl.exp
val store    : loc -> Rtl.exp -> width:int -> Rtl.rtl
val freeze   : t -> Block.t
```

Important applications: slot allocation in stack frame, implementation of calling conventions. Both are part of a target description (`Target2.t`).

Target (*Target2*)

Complex record to describe all aspects of a target. Important entries:

```
spaces:  Space.t list;      (* memory, register, temporaries *)
spill :  (Rtl.space -> Space.t) -> Register.t -> Rtl.loc -> Rtl.rtl list;
reload:  (Rtl.space -> Space.t) -> Register.t -> Rtl.loc -> Rtl.rtl list;
cc    :  string -> cc;      (* calling convention *)
stack_slots:    Automaton2.spec;

type cc =
    { sp:           Rtl.loc          (* stack pointer                      *)
    ; return:       Rtl.rtl          (* machine instr passed to Cfg.return *)
    ; proc:         Automaton2.spec  (* pass parameter to a procedure      *)
    ; cont:         Automaton2.spec  (* pass parameter to a continuation   *)
    ; ret:          Automaton2.spec  (* return values                      *)
    ; allocatable:  Register.t list  (* regs for reg-allocation            *)
    }
```

```
function Main.Util.cc (target,file)
    local ast = Driver.parse(file)
    local asm = target.asm(Driver.stdout)
    local env = Driver.check(ast,asm)
    local asm =
        Driver.compile
        (target.opt, ast, target.id, env)
    Driver.assemble(asm)
end

function Opt.sparc(proc)
    local action = seq
        { B.single(Expand.stages.placeVars)
        , B.single(Expand.stages.sparc)
        , Default.allocator
        }
    B.run(action, proc, {})
end
```

- Lua 2.5 implementation in O'Caml.
- Extensions distributed accross modules.
- At startup, `qc--` executes `qc--.lua` that controls the compiler.
- All important phases are accessible from Lua.
- Command line arguments are handled with Lua code.
- Interactive sessions with the Lua interpreter.

Two register allocators: *linear scan* and *graph-coloring* register allocator.

The graph-coloring allocator is split up into phases that are exported to Lua and can be configured using the Lua startup code.

```
CG.stages =
    { liveness         = B.mk("liveness",         CG.liveness)
    , build            = B.mk("build",            CG.build)
    , setColors        = B.mk("setColors",        CG.setColors)
    , makeWorklist     = B.mk("makeWorklist",     CG.makeWorklist)
    , simplify         = B.mk("simplify",         CG.simplify)
    , coalesce         = B.mk("coalesce",         CG.coalesce)
    , freeze           = B.mk("freeze",           CG.freeze)
    , selectSpill      = B.mk("selectSpill",      CG.selectSpill)
    , assignColors     = B.mk("assignColors",     CG.assignColors)
    , haveSpilledTemps = B.mk("haveSpilledTemps", CG.haveSpilledTemps)
    , ...
    , applyColors      = B.mk("applyColors",      CG.applyColors)
    , printCG          = B.mk("printCG",          CG.printCG)
    , printCFGLive     = B.mk("printCFGLive",     CG.printCFGLive)
    , pCFG             = B.mk("pCFG",             CG.pCFG)
    }
```

An expander is target specific; it rewrites every RTL in the CFG such that each resulting RTL represents a machine instruction.

`Expander` is manually written, `Sparcexpander` and `Dummyexpander` are generated from Burg rules.

```
addr:       App2("add", reg, rc)
            {:
                reg >>= fun r  ->
                rc  >>= fun rc ->
                return (I.generala r rc)
            :}

addr:       reg         {: reg >>= fun r -> return (I.indirecta r) :}
addr:       const13     {: return (I.absolutea const13) :}

mem:        Cell('m', agg, width, addr, ass) {: addr :}

stmt:       Store(mem, reg, width) [1]
            {: mem >>= fun m -> reg >>= fun r -> exec (I.st r m) :}
```

Currently manually modified machine-generated code.

Encoder:

```
let ld address rd =
    Rtl.store rd
        (Rtl.fetch (Rtl.cell Rtl.none 'm' Rtl.BigEndian 32 address) 32) 32
```

Decoder and Emitter:

```
| RP.Rtl [(RP.Const (RP.Bool true), RP.Store (RP.Cell ('r', Rtl.Identity,
    32, RP.Const (RP.Bits rd), _), RP.Fetch (RP.Cell ('m',
      Rtl.BigEndian, 32, RP.App (("add", [32]), [RP.Fetch (RP.Cell ('r',
          Rtl.Identity, 32, RP.Const (RP.Bits rs1), _), 32);
          RP.Const (RP.Bits arg13)]), _), 32), 32))]
when Base.to_bool (Bitops.fits_signed arg13 13) ->
   Instruction.ld (Instruction.generala (Bits.U.to_native rs1)
         (Instruction.imode (Bits.U.to_native arg13)))
     (Bits.U.to_native rd)
```

```
CVS/        config/    mwb/
INSTALL     doc/       ocaml-3.02.patch
LAUNDRY     examples/  ocaml-bug/
README      figures/   rtl/
asdl/       gen/       specialized/
aug99/      lib/       src/
bin/        lua/       tdpe/
camlburg/   man/       test/
ccl-suite/  mk/        tmp/
cllib/      mkfile     tools/
```

`src/`	main compiler source
`lua/`	Lua interpreter source
`rtl/`	RTL declaration
`gen/`	λ-RTL-generated files: emitter, constructors, recognizer
`doc/`	musings about problems
`cllib/`	generic modules: pretty printer, parser combinators
`config/`	shared mkfile rules, LaTeX macros

No need to remember the details: `mk` does it for you.