

# Systematic Compiler Construction

Michael Sperber    Peter Thiemann    Martin Gasbichler

January 30, 2004

Dieses Werk ist urheberrechtlich geschützt; alle Rechte mit Ausnahme der folgenden sind vorbehalten:

*Studenten der Informatik dürfen diese Dokument für ihre persönlichen Lehrzwecke verwenden.*

Ausdrücklich verboten wird jede andere Verwendung von Ausdrucken, Dateikopien oder Paperkopien. Insbesondere darf dieses Skriptum nicht in irgendeiner Weise vertrieben werden und es dürfen keine Kopien der Dateien auf öffentlich zugänglichen Servern angelegt werden.

Copyright © Michael Sperber, 1999; Peter Thiemann, 2000, 2002; Martin Gasbichler, 2004



# Chapter 7

## Code Generation

All necessary ingredients for code generation are now present: The denotational semantics and the various interpreters describe all features of our language and the cps representation is close enough to machine code to make code generation feasible. The last two chapters also introduced a real machine and the means to store objects directly in memory. This chapter introduces a few small libraries to ease writing of assembler programs before it turns to the final episode of compiler construction: the generation of machine code.

### 7.1 Representing assembler code

The output of our compiler is not machine code but assembler code. Using strings to represent the assembler code and sticking the output together would be possible but rather cumbersome. Instead, we first define some algebraic data types and define functions to turn them into strings.

The representation of instructions uses algebraic data-types for arguments and mnemonics. The list of mnemonics does not include all PowerPC instructions yet:

```
type arg =
  Reg of string (* Register *)
| CReg of string (* Condition register *)
| Con of int32 (* Integer constant *)
| Label of string
| IA of int * arg (* Indexed address, aka explicit based addressing *)
| IAL of arg * arg (* Indexed address using a label as index *)

type mnemonic =
  Li | Mr | Cmpi | Bc | Lwz | St | Addic | Bcr | Bl | Cmp | Bclr | Stwu |
  Stw | Stmw | AddicD | Bgt | Mtlr | Mflr | Lmw | Srwi | Br | Beq | Blr | Cmpwi |
  Subi | Cror | Lis | Ori | Add

type instruction = mnemonic * arg list

Pseudo instructions are defined analogously:

type storage_class = PR | RW | RO

type pseudo_instruction =
  Csect of (string * storage_class)
| TOC
| Long of int
```

```

| Byte of char list
| Globl of string
| Tc of (string * string * storage_class)

```

A Statement is either an instruction, a pseudo instruction, a string to be inserted verbatim, a sole label or a sequence of statements:

```

type statement =
  Instr of
    instruction * string option (* label *) * string option (* comment *)
  | Pseudo of pseudo_instruction * string option
  | Verb of string list
  | Lab of string
  | Seq of statement list

```

Some smart constructors make the life of the programmer easier:

```

let i instr args =
  Instr ((instr, args), None, None)

let withLabel (Label s) instr args =
  Instr ((instr, args), (Some s), None)

let justLabel (Label s) =
  Lab s

let withComment s instr args =
  Instr ((instr, args), None, (Some s))

let p pseudo=
  Pseudo (pseudo, None)

let p_withLabel (Label s) pseudo =
  Pseudo (pseudo, Some s)

let seq stmts = Seq stmts

```

The output of the instructions relies on the following function for printing mnemonics:

```

let mnemonic2string mnemonic =
  match mnemonic with
  | Li -> "li"
  | Mr -> "mr"
  | Cmpi -> "cmpi"
  | Bc -> "bc"
  ...

```

Similar functions exist for arguments and storage classes:

```

let arg2string arg =
  match arg with
  | Reg s -> s
  | CReg s -> s
  | Con i -> Int32.to_string i
  | Label l -> l

```

```
| IA (d,Reg b) -> (string_of_int d) ^ "(" ^ b ^ ")"
| IAL (Label l, Reg b) -> l ^ "(" ^ b ^ ")"
```

```
let storage_class2string sc =
  match sc with
  | PR -> "[PR]"
  | RW -> "[RW]"
  | RO -> "[RO]"
```

The functions for turning instructions, pseudo instructions, and statements into strings perform the major work for creating and formatting the output:

```
let with_tab s = "\t" ^ s
```

```
let pseudo_instruction2string pseudo =
  match pseudo with
  | Csect (name, sc) -> ".csect " ^ name ^ (storage_class2string sc)
  | TOC -> ".toc"
  | Long i -> with_tab (".long " ^ (string_of_int i))
  | Byte clist -> with_tab (".byte " ^
    (String.concat "."
      (List.map (function char ->
        "\"" ^ (String.make 1 char) ^ "'") clist)))
  | Globl s -> ".globl " ^ s
  | Tc (name, expr, sc) ->
    ".tc " ^ name ^ "[TC], " ^ expr ^ (storage_class2string sc)
```

```
let instruction2string (mnemonic, args) =
  (mnemonic2string mnemonic) ^ " " ^ (String.concat " " (List.map arg2string args))
```

```
let rec statement2string stmt =
  match stmt with
  | Instr (instr, None, None) -> with_tab (instruction2string instr)
  | Instr (instr, (Some l), None) -> l ^ ": " ^ instruction2string instr
  | Instr (instr, None, Some comment) ->
    with_tab (instruction2string instr ^ "\t;" ^ comment)
  | Pseudo (pseudo, None) -> pseudo_instruction2string pseudo
  | Pseudo (pseudo, (Some l)) -> l ^ ": \t " ^ (pseudo_instruction2string pseudo)
  | Verb v -> with_tab (String.concat "\n\t" v)
  | Lab l -> l ^ ":"
  | Seq stmts -> String.concat "\n" (List.map statement2string stmts)
```

A driver function for printing the obtained string completes the library:

```
let rec output_statement_seq chan is =
  match is with
  | [] -> ();
  | i::is ->
    output_string chan (statement2string i);
    output_string chan "\n";
    output_statement_seq chan is
```

```
let print_statement_seq is =
  output_statement_seq stdout is
```

## 7.2 Creating the TOC

A small library `Toc` provides creation of the Table of Contents for the AIX assembler:

```
open Iset

let uniqueLabel s =
  Label (Rename.generate_unique s)

let newLabel s =
  uniqueLabel "newLabel"

let store_labels = ref []

let csect_with_label contents (Label labelname) storage_class =
  store_labels := (labelname, contents, storage_class)::(!store_labels)

let create_TOC () =
  let lablist = !store_labels in
  let rec loop l =
    match l with
    [] -> []
  | (label, stmts, store_class)::rest ->
    let csect_name = "cs" ^ label in
    ((p (Csect (csect_name, store_class))::stmts),
     p_withLabel (Label label)
      (Tc (csect_name, csect_name, store_class))) ::
    loop rest in
  let (csects, toc_entries) = List.split (loop lablist) in
  (p TOC) ::
  toc_entries @
  List.flatten csects
```

## 7.3 Emitting code

Our programs need to allocate objects on the heap. In a real compiler, the memory is under control of a garbage collector. We have to go without one here and stick to a small piece of code that calls the well-known C function `malloc` to obtain memory. The code expects the number of bytes to be allocated in GPR 12 and returns the address of the allocated memory in GPR 12 as well:

```
let emit_alloc_heap_space () =
  let rr3 = uniqueLabel "rr3" in
  let rr4 = uniqueLabel "rr4" in
  let rr5 = uniqueLabel "rr5" in
  let rescue_lr = uniqueLabel "rescue_lr" in
  List.iter
    (fun label -> csect_with_label [p (Long (Int32.of_int 0))] label RW)
    [rr3; rr4; rr5; rescue_lr];
  (* expects size in r12 *)
  (* returns addr in r12 *)
  let alloc_heap_space_code =
    [ withLabel (Label "alloc_heap_space")
      Stw [r3; IAL (rr3, rtoc)];
```

```

    i Stw [r4; IAL (rr4, rtoc)];
    i Stw [r5; IAL (rr5, rtoc)];
    i Mflr [r5];
    i Stw [r5; IAL (rescue_lr, rtoc)];
    i Mr [r3; r12];
    i Bl [Label ".malloc"];
    i Cror [r31; r31; r31];
    i Mr [r12; r3];
    i Lwz [r3; IAL (rr3, rtoc)];
    i Lwz [r4; IAL (rr4, rtoc)];
    i Lwz [r5; IAL (rescue_lr, rtoc)];
    i Mtlr [r5];
    i Lwz [r5; IAL (rr5, rtoc)];
    i Blr [] in
csect_with_label
    alloc_heap_space_code (Label "alloc_heap_spacecsect") PR

```

The function `minicaml_start` creates a minimal runtime environment. It creates a stack frame as required by the AIX subroutine linkage conventions and creates the table of contents:

```

let gpr_save_area_size = 4 * 19
let output_argument_area_size = 4 * 8
let link_area_size = 4 * 6
let minimum_size =
    gpr_save_area_size +
    output_argument_area_size +
    link_area_size
let padding_size = minimum_size mod 16
let total_size = minimum_size + padding_size

let create_stack_frame () =
    [ p (Csect ".text", PR));
      p (Globl ".minicaml");
      withLabel (Label ".minicaml")
        Mflr [r0]; (* save LR *)
        i Stmw [Reg "r13"; IA (-gpr_save_area_size, sp)]; (* save all GPRs *)
        i Stw [r0; IA (8, sp)]; (* save LR *)
        i Stwu [sp; IA (-total_size, sp)] (* allocate our stack frame *)

let minicaml_start () =
    emit_alloc_heap_space ();
    Verb [Preface.preface] ::
    create_TOC () @
    create_stack_frame ()

let remove_stack_frame_and_return =
    [ i Lwz [sp; IA (0, sp)]; (* restore SP *)
      i Lwz [r0; IA (8, sp)]; (* restore LR *)
      i Mtlr [r0];
      i Lmw [Reg "r13"; IA (-gpr_save_area_size, sp)]; (* restore all GPRs *)
      withComment "Back to C" Blr []

```

We can now turn to the actual code generation. The presentation will take a top-down approach.

The main entry point `emit` calls `emit_cps_program` to obtain the assembler text as a string and calls the output function to emit into the specified file.

```
let emit p out_file =
  let program_code = emit_cps_program p in
  let out_channel = open_out out_file in
  output_statement_seq out_channel ((minicaml_start ()) @ program_code);
  close_out out_channel
```

The top-level loop first constructs a compile-time environment, mapping the names of the global variables to labels. It then calls `emit_top` for each top-level definition and for the body of the program:

```
let emit_cps_program (defs, body) =
  let env =
    let rec loop defs env =
      match defs with
      [] -> env
      | (global, _)::rest ->
        extend_env_with_global (loop rest env) global (Label global) in
    loop defs empty_env in
  let rec loop defs =
    match defs with
    [] -> emit_top (Global (Label "mcmmain")) body env remove_stack_frame_and_return
    | (global, cps)::rest ->
      emit_top (env global) cps env (loop rest)
  in
  loop defs
```

For each top-level definition, the compiler emits a csect that holds the value of the definition. To set the value of the global variables, the compiler generates a continuation that stores the contents of GPR 3 in the label. The code of the global variable runs with this continuation:

```
let emit_top (Global glob_label) (WithCont (_, expr)) env nextCode =
  let stopLabel = uniqueLabel "Stop" in
  csect_with_label [p (Long 0)] (*pointer to global*) glob_label RW;
  let stopCode =
    [ p (Long (Int32.to_int (make_code_vector_header 0)));
      i Lwz [r4; IAL (glob_label, rtoc)];
      i Stw [r3; IA (0, r4)];] @
    nextCode in
  csect_with_label stopCode stopLabel PR;
  [ li [r12; Con (Int32.add continuation_size code_template_size)];
    i Bl [Label "alloc_heap_space"];
    li [r4; Con continuation_header];
    i Stw [r4; IA (header_offset, r12)];
    li [r4; Con code_template_header];
    i Stw [r4; IA (Int32.to_int (continuation_size) + header_offset, r12)];
    i Addic [r5; r12; Con (Int32.add
                          continuation_size
                          heap_object_tag)];
    i Stw [r5; IA (cont_ct, r12)]; (* set code-template in conti *)
    li [envr; Con (make_fixnum 0)];
    i Stw [envr; IA (cont_env, r12)];
    li [ehr; Con (make_fixnum 0)]; (* no handler defined for now *)
```

```

i Stw [ehr; IA (cont_exh, r12)];
i Stw [ehr; IA (cont_cont, r12)]; (* no previous cont *)
i Lwz [r6; IAL (stopLabel, rtoc)];
enter_pointer r6 r6;
(* set code-vector in template *)
i Stw [r6; IA (Int32.to_int (continuation_size) + ct_cv, r12)];
enter_pointer conr r12] @
(emit_expr expr 0 env)

```

Two helper functions load constants and the value of identifiers into a register:

```

let enter_const con target_reg =
  match con with
  | Lambda.CInt int ->
    [li [target_reg; Con (make_fixnum int)]]
  | Lambda.CChar char ->
    [li [target_reg; Con (make_char char)]]

let fetch_ident id reg level env =
  match env id with
  | Local loc -> make_access (level - loc.level) loc.offset reg
  | Global glob -> [ i Lwz [reg ; IAL (glob, rtoc)];
                    i Lwz [reg; IA (0,reg)]]

```

Emitting code for `valexpr` is next. Here, the only exiting case is `Lambda`. The compiler emits the code of the body of the function into a separate csect. A static header precedes the code. The size of the code vector is set to zero to simplify the presentation. We also need code to create the closure and its code template.

```

let rec emit_valexpr valexpr level target_reg env =
  match valexpr with
  | Const c ->
    enter_const c target_reg
  | Ident id ->
    fetch_ident id target_reg level env
  | Builtin builtin ->
    enter_builtin builtin target_reg
  | Lambda (id, contid, body) ->
    let new_level = level + 1 in
    let closure_label = newLabel () in
    let closure_code = p (Long 9) :: (* type of code vector *)
      make_env_frame_from_r3 @
      emit_expr body new_level (extend_env_with_local env id new_level) in
    csect_with_label closure_code closure_label PR;
    [ li [r12; Con (Int32.add closure_size code_template_size)];
      i Bl [Label "alloc_heap_space"];
      li [r4; Con closure_header];
      i Stw [r4; IA (header_offset, r12)]; (* set it *)
      i Stw [envr; IA (clos_env, r12)]; (* store current_env *)
      i Addic [r4; r12; Con (Int32.add (* addr of code template + 10 tag *)
        closure_size
        heap_object_tag)];
      i Stw [r4; IA (clos_ct, r12)]; (* set it *)
      li [r4; Con code_template_header];
      i Stw [r4; IA (Int32.to_int (closure_size) + header_offset, r12)];

```

```

i Lwz [r4; IAL (closure_label, rtoc)]; (*adr of closure-code *)
enter_pointer r4 r4;
(* store adr of code-vector*)
i Stw [r4; IA (Int32.to_int (closure_size) + ct_cv, r12)];
enter_pointer target_reg r12]

```

We now turn to the main procedure for code generation. The function `emit` is responsible for emitting the code of normal expressions. The compiler postpones the generation of binary primitives until both arguments are known. This works by registering a partial application as `IncompleteBinary` in the environment.

```

and emit_expr e level env =
  match e with
  | Return (k, ve) ->
    (emit_valexp ve level r3 env) @
    return r3

  | Let (id, rhs, body) ->
    ...

  | If (test, cons, alt) ->
    ...

  | Call (Builtin b, arg, (x, cont_expr)) ->
    let Some p_info = Camlprim.maybe_primitive_info b in
    begin
      match p_info.Camlprim.arity with
      | 1 ->
        emit_valexp arg level r20 env @
        emit_cont (x, cont_expr) level env @
        emit_unary_builtin b r20 r3 @
        return r3
      | 2 ->
        let c level =
          emit_valexp arg level r20 env @
          emit_binary_builtin b r20 r21 r3 @
          return r3 in
        emit_expr cont_expr level (extend_env env x (IncompleteBinary c))
      end
    end

  | Call (Ident func_id, arg, (x, cont_expr)) ->
    begin
      match env func_id with
      | IncompleteBinary c ->
        emit_valexp arg level r21 env @
        emit_cont (x, cont_expr) level env @
        c level
      | _ ->
        emit_call (Ident func_id) arg (x, cont_expr) level env
      end
    end

  | Call (func, arg, cont) ->
    emit_call func arg cont level env

  | TailCall (Ident func, arg, _) ->
    begin
      match env func with
      | IncompleteBinary c ->
        emit_valexp arg level r21 env @
        c level
    end

```

```

    | _ ->
        emit_tailcall (Ident func) arg level env
    end
| TailCall (Builtin b, arg, _) ->
    emit_valexp arg level r20 env @
    emit_unary_builtin b r20 r3 @
    return r3

| TailCall (func, arg, _) ->
    emit_tailcall func arg level env

| LetCont (ki, cont, e) ->
    ...

```

The code uses a few helper functions. Here is the definition of some of them:

```

and emit_tailcall func arg level env =
    emit_valexp func level r20 env @
    emit_valexp arg level r3 env @

[ i Lwz [envr; IA (untag clos_env,r20)];          (* install closure env *)
  i Lwz [ctr; IA (untag clos_ct,r20)];          (* install code-template *)
  i Lwz [r6; IA (untag ct_cv, ctr)];            (* get pointer of code-vector *)
  extract_pointer_after_header r6 r6;
  i Mtlr [r6];
  i Blr []]                                     (* execute closure *)

let enter_builtin builtin target_reg =
    match builtin with
    | "()" -> [li [target_reg; Con mc_unit]]
    | "true" -> [li [target_reg; Con mc_true]]
    | "false" -> [li [target_reg; Con mc_false]]

let emit_unary_builtin builtin arg_reg target_reg =
    match builtin with
    | "dec" ->
        [ i Subi [target_reg; arg_reg; Con (make_fixnum 1)]]
    | _ -> raise (UII builtin)

let emit_binary_builtin builtin arg1_reg arg2_reg target_reg =
    match builtin with
    | "+" ->
        [withComment "Plus" Add [target_reg; arg1_reg; arg2_reg]]

```

# Bibliography

- [ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers Principles, Techniques, and Tools*. Addison-Wesley, 1986.
- [Boc76] G. V. Bochmann. Semantic evaluation from left to right. *Communications of the ACM*, 19:55–62, 1976.
- [Cha87] Nigel P. Chapman. *LR parsing: theory and practice*. Cambridge University Press, Cambridge, UK, 1987.
- [DeR69] Franklin L. DeRemer. *Practical Translators for LR(k) Parsers*. PhD thesis, Department of Electrical Engineering, Massachusetts Institute of Technology, Cambridge, Ma., 1969.
- [DeR71] Franklin L. DeRemer. Simple LR(k) grammars. *Communications of the ACM*, 14(7):453–460, 1971.
- [DF92] Olivier Danvy and Andrzej Filinski. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science*, 2:361–391, 1992.
- [DP82] Franklin DeRemer and Thomas Pennello. Efficient computation of LALR(1) look-ahead sets. *ACM Transactions on Programming Languages and Systems*, 4(4):615–649, October 1982.
- [DS95] Charles Donnelly and Richard Stallman. *Bison—The YACC-compatible Parser Generator*. Free Software Foundation, Boston, MA, November 1995. Part of the Bison distribution.
- [FH95] Christopher W. Fraser and David R. Hanson. *A Retargetable C Compiler: Design and Implementation*. Benjamin/Cummings, 1995.
- [FHP92] Christopher W. Fraser, David R. Hanson, and Todd A. Proebsting. Engineering a simple, efficient code generator generator. *Letters on Programming Languages and Systems*, 1(3):213–226, 1992.
- [Fis93] Michael J. Fischer. Lambda-calculus schemata. *Lisp and Symbolic Computation*, 6(3/4):259–288, 1993.
- [Ive86] Fred Ives. Unifying view of recent LALR(1) lookahead set algorithms. *SIGPLAN Notices*, 21(7):131–135, July 1986. Proceedings of the SIGPLAN’86 Symposium on Compiler Construction.
- [Ive87a] Fred Ives. An LALR(1) lookahead set algorithm. Unpublished manuscript, 1987.
- [Ive87b] Fred Ives. Response to remarks on recent algorithms for LALR lookahead sets. *SIGPLAN Notices*, 22(8):99–104, August 1987.

- [Joh75] S. C. Johnson. Yacc—yet another compiler compiler. Technical Report 32, AT&T Bell Laboratories, Murray Hill, NJ, 1975.
- [Lee93] Rene Leermakers. *The Functional Treatment of Parsing*. Kluwer Academic Publishers, Boston, 1993.
- [PC87] Joseph C.H. Park and Kwang-Moo Choe. Remarks on recent algorithms for LALR lookahead sets. *SIGPLAN Notices*, 22(4):30–32, April 1987.
- [PCC85] Joseph C. H. Park, K. M. Choe, and C. H. Chang. A new analysis of LALR formalisms. *ACM Transactions on Programming Languages and Systems*, 7(1):159–175, January 1985.
- [Plo75] Gordon Plotkin. Call-by-name, call-by-value and the  $\lambda$ -calculus. *Theoretical Computer Science*, 1:125–159, 1975.
- [Spe94] Michael Sperber. Attribute-directed functional LR parsing. Unpublished manuscript, October 1994.
- [SSS90] Seppo Sippu and Eljas Soisalon-Soininen. *Parsing Theory*, volume II (LR(k) and LL(k) Parsing) of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Berlin, 1990.
- [ST95] Michael Sperber and Peter Thiemann. The essence of LR parsing. In William Scherlis, editor, *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation PEPM '95*, pages 146–155, La Jolla, CA, June 1995. ACM Press.
- [ST00] Michael Sperber and Peter Thiemann. Generation of LR parsers by partial evaluation. *ACM Transactions on Programming Languages and Systems*, 22(2):224–264, March 2000.
- [WM92] Reinhard Wilhelm and Dieter Maurer. *Übersetzerbau — Theorie, Konstruktion, Generierung*. Lehrbuch. Springer-Verlag, Berlin, Heidelberg, 1992.