OCAMLJIT (pr.rev. 0.33)
a faster Just-In-Time Ocaml Implementation.

Basile Starynkevitch
INRIA Rocquencourt - Cristal project
basile.starynkevitch@inria.fr
basile@starynkevitch.net

ABSTRACT

This paper presents a just in time achine code generator, dynamically translating OCaml (hence MetaOCaml) byte-code into (x86, PowerPC, Sparc) machine code using the Lightning library. The OcamlJit system mimics precisely the OCaml runtime behavior. Its design and implementation is described, and performance measures are given. Alternate designs are proposed to conclude.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processor—*code generation, incremental compiler*

General Terms

Ocaml, just-in-time compilation, byte code, machine code generation

Keywords

Ocaml, Just In Time compilation, machine code generation

1. INTRODUCTION

OCaml[9, 14] and MetaOCaml[18] implementations translate source code into a byte-code for the OCaml virtual machine (inspired by Zinc [8]). OCaml also has a `ocamlopt` native compiler, which produces native machine code (by generating assembler files) on some common platforms (like x86-Linux or Windows, PowerPC-MacOSX, and many others).

MetaOCaml shares a significant part of OCaml code, and uses exactly the same virtual byte-code machine.

This paper presents OcamlJit [16] - an open-source (under LGPL license) faster implementation of the OCaml interpreter (designed to be usable for MetaOCaml) - which uses just-in-time machine code generation. OcamlJit is developed on x86, Linux system (Debian/Sid) - and should be easily portable to PowerPC, Sparc (32bit) architectures or other operating systems[1]. Section 2 presents existing systems (the OCaml and MetaOCaml implementations, and the Lightning library). Section 3 gives the design and implementation of the OcamlJit system. Performance measures are in section 4. The OcamlJit software is mostly a 3900 lines `jitinterp.c` file, which (with the existing `ocaml/byterun/` runtime) compiles into a `libcamljitrun.a` library and a `ocamljitrun` executable. We conclude (§5) by suggesting alternative designs and possible future work.

2. EXISTING SYSTEMS

We present the existing OCaml (and MetaOCaml) systems, and the GNU Lightning library used in our JIT implementation. [readers knowing well the internals of OCaml implementation can skip to 2.3.]

2.1 OCaml compiler usage

The OCaml system is usable either interactively, through the `ocaml` command, which provides a "read an expression, compile it, compute it" interactive loop or in batch mode through the `ocamlc` command which compiles a sequence of `*.ml` or `*.mli`[2] source files or byte-code object files `*.cmi` or `*.cmo`[3]; and produces a byte-code object file (to be reused by another invocation of `ocaml`) or a byte-code executable file. The `ocaml` compiler also handles `*.c` sources and `*.so` shared libraries (for external functions called by OCaml and coded in C) and also deals with (i.e. produce or consume) byte-code libraries `*.cma`.

The byte-code executable file produced by `ocamlc` is interpreted by the `ocamlrun` program[4] every time it is executed.

[1]For the supported architectures (x86,PowerPC,Sparc - 32bit), the main system dependency is in the `mmap` and `munmap` system calls for executable page-aligned memory allocation and release. Patches are welcome for other OSes (e.g. Windows).

[2]Source files are either module interfaces or signatures `*.mli` or module bodies or implementations `*.ml`

[3]A `*.cmi` file is a compiled form of a module interface, from a `*.mli` source; a `*.cmo` is a compiled form of a module implementation, from a `*.ml`.

[4]An OCamlbyte-code executable is technically a `#!/usr/bin/ocamlrun` Unix "script" file (organized in sections), so `execve` of an OCaml byte-code executable invokes `ocamlrun`.

The goal of this work is to provide a `ocamljitrun` program, substitute for the `ocamlrun` program, able to handle any OCaml byte-code executable (without modification or special compilation), running it a bit faster through just-in-time dynamic machine code generation techniques. Hence `ocamljitrun` should be usable by byte-code programs compiled by any future release of `ocamlc` which don't change the byte-code virtual machine, but only the OCaml (or MetaOCaml) language and compiler.

2.2 Overview of the Ocaml compiler

We recall here an overview of OCaml implementation. Feel free to skip this subsection 2.2 if you know the details of it.

2.2.1 compiler phases and representations

The `ocamlc` compiler parses a source tree (loading relevant compiled interfaces) into an abstract syntax tree (AST, see file `ocaml/parsing/parestree.mli` of the ocaml source) like in fig. 1. Then the typing machinery computes type annotations to produce a typed tree (see file `ocaml/typing/typedtree.mli`) like in fig. 2.

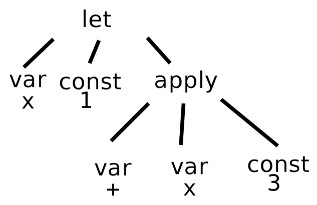

Figure 1: abstract syntax tree

MetaOCaml adds a few primitives [2] to the OCaml language (bracket, escape, run, ...), so extends its AST and its typed tree representations to support multi-stage programming [17, 19].

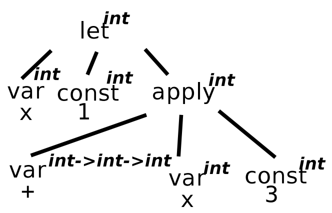

Figure 2: typed tree

From the typed tree, the compiler computes a "lambda" representation (see file `ocaml/bytecomp/lambda.mli`), inspired by the *untyped call-by-value* lambda calculus (see fig. 3). This tree representation is not explicitly related to the source code (because source file positions and some names are lost at this stage), and do not contain any explicit type information.

```
(let (x/60 1) (+ x/60 3))
```

Figure 3: "lambda" representation

The "lambda" tree representation includes leafs for variables, constants, and nodes for application, function definition, let and letrec, primitive operations (like addition, etc..), switch (used in compiled forms of `match` expressions), exception handling (raise, catch, try-with nodes), imperative traits (sequence, for and while loops), method send, assignment to locals, ... The "lambda" representation is produced from the typed tree, and then is subject to peephole optimizations in the byte-code[5] compiler.

MetaOCaml added only the ability to quote an arbitrary OCaml value as a constant inside "lambda" trees.

After some optimizations (transforming "lambda" trees into better or simpler "lambda" trees), the "lambda" tree representation is transformed into a list of "byte-code" instructions (see fig. 4). This list is subject to peephole optimizations and then written into the generated byte-code file (or in memory, by the interactive top-level).

```
const 1
push
acc 0
offsetint 3
pop 1
makeblock 0, 0
setglobal Example!
```

Figure 4: byte-code

The `ocamlc` compiler has some undocumented options `-dlambda` ... `-dinstr` to display most of the various representations above.

2.2.2 the OCaml byte-code interpreter

The byte-code[6] interpreter (or virtual machine - fig. 5) works with a stack (of OCaml values), and with six "virtual" registers: a stack pointer `sp`, an accumulator `acc`, an environment pointer `env` (pointing to the current closure, which contains the sequence of closed values), a trap frame pointer `trapsp`, an extra arguments counter `xtrargs`, a byte-code program counter `bpc`. The virtual machine also deals with byte code segments (succession of byte codes to be interpreted) and a global data array `caml_global_data` (of OCaml values). Usually, there is only one byte-code segment, the sequence of byte-codes in the interpreted file (produced by `ocamlc`), which also contains the marshalled (or serialized) form of the global data.

The OCaml stack (which is not the native machine C stack in `ocamlrun` or `ocamljitrun`) contains OCaml values, byte-code return addresses and arity counters, organized into OCaml call frames. The byte-code is pointed to either from the stack (return addresses) or from closures (inside the heap). The first slot of a closure is the byte-code address of the closure's code; the other slots are closed values, forming the environment of the closure.

The byte-code interpreter `ocamlrun` starts by reading (unserializing or unmarshalling) the global data, then loads

[5]The native code compiler `ocamlopt` has a similar representation (in file `ocaml/asmcomp/clambda.mli`) which adds explicit direct or indirect calls and closures.

[6]Actually each token of the byte-code is a 32 bits word, so the byte-code is really a "wordcode"

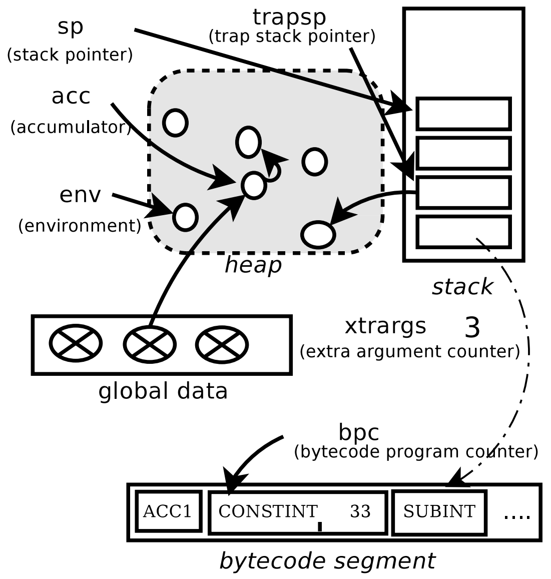

Figure 5: OCaml byte-code virtual machine

the sequence of byte-code into memory and process dynamically linked libraries containing external primitives functions (listed inside the interpreted file). At last, the function `caml_interprete` of file `ocaml/byterun/interp.c` is invoked to interpret the initial byte-code segment (starting from its first byte-code) with an empty stack (and stack and trap-stack pointers to it), and a default environment and accumulator.

OCamlvalues are either pointers to blocks in the garbage collected heap, or tagged integers. Each heap block starts with a header containing a tag, a size, and GC colors. Some tags describe special blocks (like strings, closures, ...), but most of them are used for representing sum types.

2.2.3 the byte-code instruction set

Each instruction of the byte-code is represented by one or more consecutive 32bits words. The first word of the byte-code is for the operation (enumerated in file `ocaml/byterun/instruct.h`). Other words might be constant integer arguments (usually offsets). The byte-code operates on the accumulator and on topmost elements of the OCaml stack. Many operations produce a result inside the accumulator. Some byte-codes convey an offset (noted n,m or by juxtaposition), other instructions are followed by explicit offsets (noted p,q,... or by subscripts). All other byte-codes are only referenced (in the byte-code) by relative offsets, so the byte-code is a “position independent [byte]code”.

Byte-code instructions can be classified as follow :

- stack manipulation instructions: `ACC0` `ACC`n ... `ACC6` `ACC`$_p$ put the n-th or p-th topmost stack element into the accumulator; `PUSH` pushes the accumulator on the stack. `POP`$_p$ removes p elements off the stack. `ASSIGN`$_p$ put the accumulator into the stack, at offset p from top.
- loading instructions: `CONSTINT`$_p$ `CONST0` `CONST`n ... `CONST3` put a tagged integer n or p into `acc`. `GETGLOBAL`$_p$ loads the p-th global value (from the global array `caml_global_data` into `acc`). `ATOM`$_p$ and `ATOM0` loads an atom (the pre-allocated 0-length value) tagged p (resp. 0) into `acc`.
- primitive operations like integer arithmetic (and bitwise operations); unary `NEGINT` (negation) `BOOLNOT` (boolean not) operate on `acc`. Binary operations `ADDINT SUBINT MULINT ANDINT DIVINT MODINT` (for `+ - * land / mod`) etc.... They take an operand in `acc` and another popped off the stack and put their result in `acc`. Division and modulus operations also check for 0 and may raise an exception. Comparison operations `EQ NEQ LTINT LEINT` etc... similarly put their boolean result in `acc`. The `ISINT` operation tests if `acc` is a tagged integer (like an unary compare).
- environment access operations, like `ENVACC1` `ENVACC`n ... or `ENVACC`$_p$ which get the n-th or p-th slot of the current `env` into the `acc`.
- apply operations, both pushing calls (creating a new call frame on the OCaml stack) like `APPLY1` ... `APPLY3` and tail-recursive calls (which overwrite the current call frame on the OCaml stack) like `APPTERM1` `APPTERM3`. The accumulator contains a closure which is applied to arguments on the OCaml stack and becomes the new `env`. General arity case is handled by `APPLY`$_p$ and `APPTERM`$_{p,q}$ (p being the arity, and q being the height of the current frame for tail call). The OCaml call frames contain arguments, return byte-code program counter, and arity `xtrags`; when adding call frames, the stack may be resized if needed.
- function return instructions `RETURN`, with `PUSH_RETADDR`$_p$ (where p is a byte-code offset of the pushed return address), and `RESTART` and `GRAB`$_p$ used for partial application (which allocates a currifying closure).
- closure allocating instructions `CLOSURE`$_{p,q}$ - which allocates a single closure of p slots with byte code at offset q - and `CLOSUREREC`$_{p,q,k_1,...,k_p}$ - which allocates a mutually recursive closure with p functions (ie byte-codes at offsets $k_1, \dots, k_p$) and q slots -.
- other allocating instructions: `MAKEBLOCK1`$_q$ ... `MAKEBLOCK3`$_q$ `MAKEBLOCK`$_{p,q}$ makes a block tagged p of 1, ...3, q values (first in `acc`, next popped off the stack). `MAKEFLOATBLOCK`$_p$ builds an array of floats. All (closure or other block) allocations may trigger the garbage collector and put the newly allocated block into `acc`.
- accessing and modifying operations : `GETGLOBALFIELD`$_{p,q}$ gets from p-th global value its q-th field. `GETFIELD0` ... `GETFIELD`$_p$ get (into `acc`) the first, ... $p+1$-th field from `acc`. Symetrically, `SETFIELD`$_p$ `SETFIELD0` ... `SETFIELD3` set the $p+1$-th (1^{st}, ...) field of `acc` to the popped top of stack. Similarly `SETFLOATFIELD`$_p$ and `GETFLOATFIELD`$_p$ handle a floating point field (in a record of floats). For array access and update, `GETVECTITEM` and `SETVECTITEM` are used (the popped top of stack containing the

index). For string access and update GETSTRINGCHAR and SETSTRINGCHAR. $\texttt{SETGLOBAL}_p$ sets the p-th global. the modifying operations have to cooperate with the garbage collector.

- control instructions include (in addition to function application instructions) conditional BRANCHIF BRANCHIFNOT (depending upon boolean in acc), comparing BEQ BNEQ BLTINT .... and unconditional BRANCH jumps. There is also a $\texttt{SWITCH}_{p,q,k_0,\ldots k_{p-1},k'_0\ldots k'_{q-1}}$ instruction (used for compiling match OCaml expressions) testing acc and jumping to offset k_i if it is the tagged integer i and to offset k'_j if it is a block of tag j. Exception handling PUSHTRAP POPTRAP RAISE and signal checking CHECK_SIGNALS are also control instructions, as is the halting STOP instruction, which is the last byte-code of a segment.

- call to external C primitives: $\texttt{C\_CALL1}_p$ ... $\texttt{C\_CALL5}_p$ calls the p-th C primitive with argument from acc and others popped off the OCaml stack. The result of the call goes into acc. Primitives are used for several basic operations, including floating point.

- object oriented operations: GETMETHOD GETPUBMET and GETDYNMET fetch dichotomically the method associated to a message "name", sometimes using a cache inside the byte-code.

- offset operations, e.g. $\texttt{OFFSETINT}_p$ (which adds p to the tagged integer inside acc) or $\texttt{OFFSETCLOSURE}_p$, (which fetchs the $p+1$-th field of env into acc) etc...

- debugger related opcodes - the OCaml debugger puts a breakpoint by *overwriting* the byte-code with BREAK; these are not supported by OcamlJit.

- shortcuts - many byte-codes are actually shortcuts for common small sequence of byte-codes, for example PUSHCONST2 is the same as PUSH CONST2 instruction sequence.

2.3 The Lightning code generator

The Lightning library [1] (free software, LGPL license) by P. Bonzini extends the CCG library of I. Piumarta[12]. Both are coded in C, and are actually a big set of C macros or inline functions[7]. CCG is a big set of macros which generate machine code, using macros reminiscent of assembler syntax. For example, the x86 assembler instruction `add 4,%eax` which increments the `%eax` register by 4 is emitted by coding `ADDWir(4,_EAX);` in C. The execution of this C instruction writes `0x0504000000` (hex) into memory, which is the encoding of the "`add` word" instruction. CCG also provides a specific preprocessor to be able to write in an assembly like syntax (instead of the ADDWir macro) intermixed in C code. CCG exists for several (32 bits) target machines: x86, PowerPC and Sparc.

The Lightning library factors the target-machine dependent macros provided by CCG by providing a single common API for the 3 target platforms (x86, PowerPC, Sparc). To make it possible, Lightning provides a common RISCy abstraction of its targets (load-store memory into registers, and ALU operation in registers), which provides only 6 word registers (and 8 separate floating point registers); the above addition would be emitted by `jit_addi_l(JIT_R0, JIT_R0, 4);`. Both Lightning and CCG are only simple machine code generators, which do not perform any optimization (no clever register allocation, no clever instruction scheduling) - in contrast with the new[8] libjit [21] which uses BURG [4] instruction selection techniques, or with Vcode [3]. Lightning code generation is mostly a quick template expansion of machine instructions.

Generation of forward jumps is handled by Lightning: the origin (machine code) location of each such jump has to be remembered, and when the destination (machine code) address is known, each origin has to be `jit_patch`-ed.

Lightning deals with code cache coherence by requiring applications to call the `jit_flush_code` macro (on the range of generated machine code) after generating machine code. On x86, this macro does nothing, because the architecture requires a combined data and instruction cache.

3. OCAMLJIT DESIGN AND IMPLEMENTATION

3.1 compatibility with byte-code interpreter

Our main goal was maximal compatibility with the OCaml byte-code interpreter, its runtime system (including garbage collector), and its behavior. We do not want to depend upon any assumption regarding the OCaml byte-code compiler or the produced byte-code: any sequence of byte-codes should be handled by OcamlJit, and the OCaml heap and stack should be the same as with `ocamlrun`.

This design goal entails several constraints:

- most of the code (either in C, for the runtime, or in OCaml, for the compiler) is common with the byte-code interpreter. It is expected that OcamlJit source code is reusable without patches as long as the `ocaml/byterun/interp.c` file does not change.

- the runtime (ie the garbage collector and the required C primitives) is the same as in `ocamlc`.

- the heap is the same, in particular all *Ocaml values keep the same representation.* Hence, the code pointer inside closures is still a *byte-code pointer*, not a native code pointer.

- the OCaml program cannot tell (except by performance measures) if it is running under OcamlJit or under the `ocamlrun` byte-code interpreter. A given OCaml program can run both with OcamlJit and with the byte-code interpreter and can exchange closures (by marshalling) between both processes.

- the interactive top-level `ocaml` can run under `ocamljitrun`, and MetaOCaml programs (which

[7]There is no archive library `lib*.a` for them.

[8]Libjit was not yet available when this work was started.

likewise generate byte-code at runtime) should also run[9] under `ocamljitrun`.

- C programs embedding the OCaml byte-code interpreter can embed our OcamlJit instead, using our `libcamljitrun.a` library as a replacement for `libcamlrun.a`.

The above constraints means that a program running under `ocamljitrun` see the same virtual machine (fig. 5 above) as `ocamlrun`; in particular, it has access to the same virtual registers, and a *byte-code program counter* has to be maintained. The "call stack" is not the native C machine stack, but the same OCaml stack as with the byte-code interpreter.

3.2 implementation description

OcamlJit has to mimic as much as possible the byte-code interpreter. Since the LightningRISCy model provides few registers, common OCaml virtual registers (`sp,acc,env`) go into Lightning registers (hence into machine registers) while other less common OCaml registers are grouped into a C `struct caml_jit_state_st` (here called the OcamlJit state) pointed to by a Lightning register `JML_REG_STATE`. The `ocamljitrun/jitinterp.c` replaces the `caml/byterun/interp.c` source and provides the same entry name `caml_interprete`.

3.2.1 main translation loop

The core of OcamlJit is the `caml_jit_translate` C routine which scans a byte-code sequence and emits appropriate equivalent machine code. It is a loop around a big `switch` (one for each byte-code operation) statement. The case for the `ANDINT` operation (which pops a tagged integer from the OCaml stack and does a bitwise "and" with the OCaml accumulator) is shown in figure 6. It needs a fixed Lightning (and machine) temporary register to load the top of stack.

```
case ANDINT:
  //. emit "tmp1 = *sp; sp++; acc = acc & tmp1"
  jit_ldr_p (JML_REG_TMP1, JML_REG_SP);
  jit_addi_p (JML_REG_SP, JML_REG_SP,
              WORDSIZE);
  jit_andr_l (JML_REG_ACC, JML_REG_ACC,
              JML_REG_TMP1);
  break;
```

Figure 6: a simple case in `caml_jit_translate`

These 3 `jit_*` macro invocations are expanded on x86 by the C preprocessor to a messy code of about 7800 bytes, partly shown in figure 7. However, the GNU `gcc` 3.4 compiler is able to optimize it (with `-O3`) to about thirty x86 machine instructions.

The OcamlJit generated x86 machine code is shown disassembled in figure 8. OCaml `sp` is in the `%eax` machine register (i.e. `JML_REG_SP` Lightning register), `acc` is `%abx` (i.e. `JML_REG_ACC`) and `%ecx` is a temporary (`JML_REG_TMP1`).

[9]OcamlJit *requires* OCaml version 3.08 or greater, and we did not port MetaOCaml to OCaml 3.08, so we did not run MetaOCaml with OcamlJit runtime.

```
case ANDINT:
( ( ((_ul )(((*_jit.x.uc_pc++)= ((_uc )((0x8b)&
0xff)))))),(((0)==0) ? (((((0x40))))==0) ?
(((_ul )(((*_jit.x.uc_pc++)= ((_uc )((((0)<<6)|
((((((((0x41))))&0x7))&0x7))<<3)|(5))&
0xff))))),((_ul )(((*_jit.x.ul_pc++)=
((_ul )(((long)(0)))))))) :(0x44==((((0x40)))
? ((((0)==0) ? (((_ul )(((*_jit.x.uc_pc++)=
```

Figure 7: tiny part of cpp expanded code (x86)

```
movl   (%eax),%ecx
addl   4,%eax
andl   %ecx,%ebx
```

Figure 8: OcamlJit generated x86 code

The equivalent PowerPC code[10] is shown in figure 9.

```
lwzx    r10,r0,r9
addic   r9,r9,4
and     r29,r29,r10
```

Figure 9: OcamlJit generated PowerPC code

The equivalent Sparc (version 9, 32 bits) code[11] is shown in figure 10.

It should be noted that most operations use the OCaml stack, so need to fetch the memory. This is compatible with `ocamlrun` behavior, but makes OcamlJit run slower than code produced by the native `ocamlopt` compiler.

3.2.2 byte-code and native-code addresses

Compatibility requires us to keep the byte-code and to use byte-code addresses, in particular in OCaml closures. This means that closure application (a very common operation in OCaml) has to get the byte-code from the closure, find the corresponding native machine code, and jump to it.

An hash-table `caml_jit_pc_tblptr` is used for this byte-code address to native code address matching. Its size is a power of 2 (so its modulus is just a bit-wise and with `caml_jit_pc_mask`), and it is filled by the `caml_jit_translate` routine for each translated byte-code instruction[12]; hence OcamlJit does not know about OCaml function code boundaries (or yet unseen branch targets). Each bucket is a null terminated array of (byte code pointer, native code pointer) structures (or the empty zero-filled bucket `caml_jit_empty_bucket`).

[10]A bug in GNU Lightning still present in june 2004 - in the prolog generation- currently crashes OcamlJit on PowerPC 32bits.

[11]A bug (probably in Lightning macros) still present in june 2004 crashes some simple test Ocaml code on Sparc.

[12]OcamlJit does not care and does not know if a given byte-code address will be used in a closure or in branch; hence, all byte-code addresses are stored, even if most of them are not jumped to.

```
ld  [ %g0 + %l0 ], %l1
add  %l0, 4, %l0
and  %l3, %l1, %l3
```

Figure 10: OcamlJit generated Sparc code

This hash-table is sparse: it is grown when the number of entries is bigger than half of the hash-table size (a power of 2). Experimentally, buckets are very small (usually one or two entries). The hash-code of a byte-code address `b` is `(b ^ (b >> 13)) & caml_jit_pc_mask` so is computed in a few machine instructions. This hash-table is optimized for speed (in the OcamlJit generated machine code), because finding the machine address for a given byte-code is an extremely common operation.

Most applications have only one byte-code segment (the application program byte code). However, some applications may have more than one segment, and a few applications may have many (e.g. hundreds) segments, in particular long lived interactive top-level `ocaml` sessions (where a byte-code segment may be created at each user interaction with the top-level) and MetaOCaml programs (since code generation is the heart of multi-stage or meta- programming). Byte-code segments should be explicitly removed, because OCaml does not garbage collect code. When a byte-code segment is removed, the generated native code is freed and all its byte-code addresses are removed from this hash-table.

Each byte-code segment is translated (incrementally) into a single (native) code block made of code chunks. See figure 11. The code block knows its byte-code segment. The first code chunk of a code block starts with a prologue, which contains code to fetch the current byte-code address inside the state pointed by `JML_REG_STATE`, find the equivalent native code and jump (indirectly) to it. If there is no native code yet, it means that the byte-code was not yet translated, and the translator should then be called. This byte-code jumping code is a hot spot, and its machine code address is kept (in the `cb_jumppcref` field of the code block).

A byte-code branch is translated either to a direct jump to the target native code if it is known (i.e. found in `caml_jit_pc_tblptr` hashtable), or to the code jumping to a byte code address otherwise. As soon as the target machine code address is known, the branch is translated again to an efficient direct jump.

Every time forward references are resolved, each "unknown" branch (whose target byte-code is known, but whose target machine code address is still unknown) is re-examined: if its native code address becomes known, a direct machine jump is rewritten, otherwise a jump to the byte code jumping code is rewritten. Care is taken to flush the instruction cache appropriately.

The above scheme permits incremental code generation, in several code chunks and several code blocks (or byte-code segments).

3.2.3 managing code chunks and code memory

Each byte-code segment has its own (native) code block, which references a linked list of (native) code chunks. A code chunk is a page-aligned executable memory segment (allocated through the `mmap` system call with read, write, and execute permissions).

The byte-code is incrementally translated to native machine code. The native code generation is interrupted when the end of chunk is reached, or when a `STOP` byte-code is reached, or when a configurable number of byte-code instructions -the *chunk threshold length*- (default is 2000, 0 meaning infinity i.e. "batch" translation of all the byte-code segment at once) has been translated. Of course the code chunk always ends by the equivalent of a branch to its successor byte-code, if any. Code chunks are filled gradually, and their allocated length is an affine function of the number of byte-codes in the byte-code segment. Since OcamlJit generates machine code, this code is not relocatable, hence code chunks cannot be moved.

Because multiple arguments OCaml functions translate to a sequence of byte-codes starting with `RESTART`, this byte-code is also a hint to interrupt translation, and add a new chunk if the current one is nearly full. If the currently translated byte-code address is already known in the `caml_jit_pc_tblptr` hash-table (because a byte-code sequence containing it was previously translated), translation is of course interrupted.

Since jumping to an unknown byte-code address triggers its translation, translation is done incrementally, on demand, so only byte-codes (or nearby) executed (i.e. jumped to) are translated. Hence, a big sequence of byte-codes which is never executed is not translated.

OcamlJit has to manage the set of byte-code segments and the corresponding code blocks (which refer to code chunks). Usually, there is only one byte-code segment. C call-backs (i.e. applying an OCaml closure from a C primitive) also need a tiny byte-code segment. Each code block knows its byte-code segment (with an hash of the byte-code). An hash table of code blocks is kept in `caml_jit_codtable`, and a small cache `caml_jit_cachecb` of recently seeked code blocks is managed (to quickly find a code block from an internal byte-code pointer, e.g. when incrementally translating it).

3.2.4 interaction with the runtime

The generated machine code has to interact with the runtime. A common interaction is invoking the garbage collector when the young region is full in allocations (byte-codes `MAKEBLOCK`, `CLOSURE`, ...). To speed up common allocations (which allocate small blocks of known tag and size), the decrement of the `caml_young_ptr` and its comparison with the `caml_young_limit` is inlined. Calling the garbage collector follows the same conventions as in OCamlbyte-code interpreter: `acc` and `env` are pushed on the OCaml stack, and `sp` is saved in the `caml_extern_sp` variable. Hence, `MAKEBLOCK2 0` (which allocates a 0-tagged block of 2 pointers, one in `acc`, the other popped from the OCaml stack, and put the new block pointer inside `acc`) is compiled into 29 x86 machine instructions, of which only 17 are executed in the usual case when the garbage collector is not invoked.

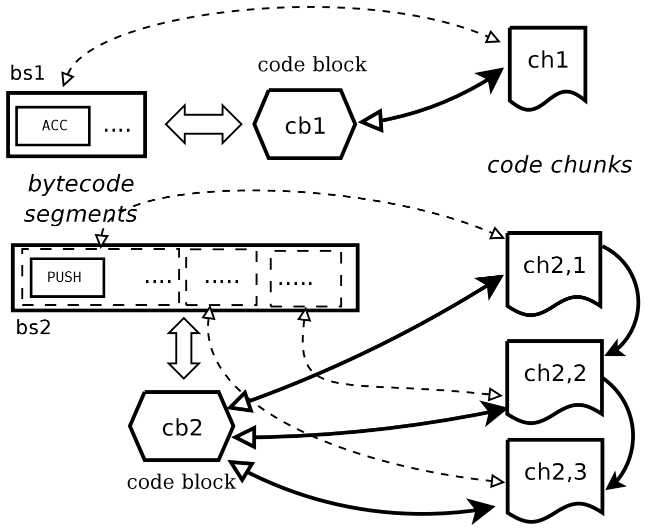

Figure 11: Ocamljit byte-code segments and code blocks, chunks

Another common interaction is calling C primitives (like e.g. system calls, e.g. `caml_sys_chdir`). The generated machine code is usually a call to the C primitive (following calling conventions similar to the GC). Some simple operations are implemented by C primitives, in particular floating point operations. OcamlJit handle specially a few of these; for example, when the called C routine is `caml_add_float`, OcamlJit inlines code which fetch into registers the two added floats from their boxes, add them, and allocated a box for the result, etc... This trick gave a significant boost (20-30%) to some numerical programs.

OcamlJit has other interactions with the runtime; in practice, the generated code prologue is called by `caml_jit_run` with the `caml_jit_state_st` structure address; this single argument is kept in the `JML_REG_STATE` Lightningregister. The generated routine returns (in many places) with an integer code; this code may be `MLJSTATE_RETURN` to return from the interpreter, `MLJSTATE_MODIF` to modify in place a heap value (this may change GC colors), `MLJSTATE_PROCESS_SIGNAL` to process an incoming Unix signal, `MLJSTATE_RAISE` to raise an exception (via C), `MLJSTATE_GRAB` to grab (used for partial application) a closure, `MLJSTATE_JITRANSLATE` to ask for more byte-code translation, `MLJSTATE_MAKEBIGBLOCK` or `MLJSTATE_MAKEBIGFLOATB` to allocate a big pointer or floating point block. In general, any unusual processing which is too complex to be inlined is done by a particular `MLJSTATE_*` code (passing the required data in the state structure) The `caml_interprete` entry point first initialize the state and the code block and then loops around `caml_jit_run`, handling the processing required by one of the returned code above.

3.2.5 minor adaptation of OCaml

Some minor modifications of the OCaml system have been required. They have been added into the 3.08 release of OCaml, and do not affect the byte-code interpreter or the native `ocamlopt` compiler.

The C callback (applying from a C primitive an OCaml closure to some arguments by calling `caml_callback*` and similar functions) is implemented (in file `ocaml/byterun/callback.c`) by pushing the arguments on the OCaml stack and interpreting a tiny byte-code segment containing `ACC`, `APPLY`, `POP`, `STOP` (with various integer byte-code "operands"). The original (3.07) version had a single array of such byte-codes which was overwritten by each invocation of `caml_callback*` - this could not work (for recursive call-backs - because this array would have been rewritten during its execution) as is with OcamlJit, so some code was added into the OCaml interpreter to copy the call-back byte-code into a C stack allocated small array of byte-codes (so each such pending array had a unique address).

OcamlJit requires obviously some processing on any new byte-code segment (and prohibits byte-code modification[13]), so the runtime added the convention that each new byte-code segment should be prepared (by calling `caml_prepare_bytecode`) before use and released (by calling `caml_release_bytecode`) after. The byte-code interpreter does nothing in these routines, but OcamlJit creates a code block when preparing and release the code block and contained code chunks when releasing. In the compiler, the `Meta.reify_bytecode` primitive (of `ocaml/bytecomp/meta.mli`) calling `caml_reify_bytecode` of `ocaml/byterun/meta.c` and `ocaml/byterun/obj.c` has been changed to prepare byte-code. The `Meta.static_release_bytecode` primitive has been added (called in the `ocaml` toplevel and in MetaOCaml) which releases a byte-code segment after it was used (and before freeing it with `Meta.static_free`).

Also, method cache functions have been enabled in `ocaml/byterun/obj.c` - they are called by the inlining of `GETDYNMETH` byte-code on method cache misses.

3.2.6 debugging issues

[13]The OCamldebugger works by modifying in place the byte-code of the debugged OCaml program - which therefore cannot be executed by OcamlJit.

A lesson learned by this development is that debugging of dynamic machine code generator is more painful than expected. A feature lacking in the `gdb` debugger is the ability to program the debugger in a real scripting language. This feature would have been invaluable.

Some bugs were easy to fix; in particular repeatable faults (e.g. `SEGV` fault because of a bad pointer access) in generated machine code is usually easy to fix if a trivial example exercise it. Such bugs have been corrected traditionally, using `gdb` (and generating from OcamlJit a file of gdb debugging commands). But bugs which affected the OCaml heap or stack where much harder to fix, because execution continued in a faulty way.

In practice, we had to prototype (in Ruby and in OCaml) our own debuggers, using the `ptrace` system call. The debuggers communicate with OcamlJit through textual pipes; OcamlJit sends them the location of every machine code position matching a byte-code, important pointer data (e.g. address of the OcamlJit state), and the debugger set appropriate breakpoints, peeked OcamlJit memory, etc... The detailed execution traces of OcamlJit generated code was compared with the traces of the byte-code interpreter, which serves as reference.

While the calling conventions of some key parts of OCaml (in particular calling the garbage collector) where helpful for the byte-code interpreter, they made the OcamlJit development more complex.

4. PERFORMANCE

All measurements here are done -unless said otherwise- on a x86 desktop computer (AthlonXP 2000+, with a 1.66 GHz clock and 256Kb cache, 512Mb RAM, 120Gb IDE 7200rpm disk) running Debian/Sid (Linux 2.6.7 kernel). Each test is run three times, and the best timing is given. The C compiler is GNU `gcc-3.4`. The OCaml compiler version is almost 3.08[14]. The OcamlJit code is revision 1.42 of `ocamljitrun/jitinterp.c` compiled with `-O3` optimisation.

4.1 translation overhead

While the Just-In-Time translation of byte-code to machine code is done quickly (and crudely), it still has a small overhead. OcamlJit is instrumented[15] to measure it (to the limited precision available to the `times` system call).

Table 1 gives some translation measures on byte-codes inside the `ocaml/` source tree.

The first column gives the invocation: `ocamlc` invocations are byte-code compilation inside `ocaml/stdlib/`; the chunk threshold length (i.e. the number of byte-code instructions after which OcamlJit might interrupt translation) is by default 2000, unless given explicitly (e.g. "*chk*^{500}", or "batch" - which means that all the byte-code instructions of the segment are translated). The second column gives the number of byte-code instructions translated; the third is the

invocation	#by.c. instr.	transl. time *(s)*	speed *(M b.i/s)*
`ocamlc -h`	73713	0.09	0.82
chk^{500} `ocamlc -h`	44817	0.04	1.12
chk^{4000} `ocamlc -h`	90833	0.11	0.82
`ocamlc callback.mli`	80165	0.10	0.80
`ocamlc pervasives.mli`	88749	0.11	0.81
chk^{500} `ocamlc oo.ml`	78524	0.10	0.78
chk^{4000} `ocamlc oo.ml`	118081	0.14	0.84
`ocamlc oo.ml`	108075	0.11	0.98
`ocamlc format.ml`	113269	0.12	0.95
chk^{500} `ocamlc format.ml`	96808	0.12	0.80
chk^{4000} `ocamlc format.ml`	121725	0.14	0.86
`ocamlc *.mli`	89227	0.10	0.89
`ocamlc *.ml`	114876	0.11	1.04
batch `ocamlc -h`	128167	0.13	0.98
chk^{500} `ocamldoc -h`	78469	0.09	0.87
chk^{2000} `ocamldoc -h`	128540	0.12	1.07
batch `ocamldoc -h`	200543	0.24	0.83

Table 1: translation time (x86)

translation CPU time in seconds; the last column is the translation speed, in millions of byte-code instructions per CPU seconds. We also measured while compiling (with `-c` and any other required options, as in the distribution) individual large (`stdlib/format.ml`, `stdlib/pervasives.mli`) and small (`stdlib/oo.ml`, `stdlib/callback.mli`) files, or almost all[16] of them.

The JIT translation cost is about one microsecond per translated byte-code instruction. This cost may actually be significant when the execution is very short (in particular with `ocamlc` on tiny compilations). Hence, OcamlJit may sometimes be slower than the byte-code interpreter `ocamlrun`. For example, in `ocamlc/stdlib`, `make`-ing the standard library with `ocamlrun ocamlc` takes 5.2 seconds, but making it using OcamlJit with `make RUNTIME=ocamljitrun` takes 14.4 seconds with the standard chunk size of 2000 byte-code instructions (or in batch mode), and 10.4 seconds with a smaller chunk size of 500; since `ocamlc` is invoked 84 times, its byte-code is translated 84 times (for a cumulated translation time of about 10 seconds).

This suggests that OcamlJit is better suited for not too short executions, i.e. for OCaml processes running more than a few seconds. It also suggests possible improvements: caching on disk the result of a translation (i.e. the generated machine code chunks, which could be reused at the same address only); mixing the byte-code interpreter and the OcamlJit machine code generator (e.g. translating to machine code a sequence of byte-codes only after it has been executed once by the byte-code interpreter, i.e. at it second invocation).

4.2 execution time speedup

The significant time is not the code generation time or the generated code execution time, but the total execution time

[14]It is the OCaml CVS tree of june 30th 2004.

[15]For translation time measures, use the debug variant `ocamljitrund` with the environment variable `OCAMLJITOPTION` containing `time`.

[16]In this paper `*.ml` means all the source files from `ocaml/stdlib` which do not require special compilation arguments like `-nopervasives` needed for `stdlib/pervasives.ml[i]` or `-nolabels` needed for `stdlib/*Labels.ml[i]`.

(which is their sum). Table 2 gives some timings (cpu combined system and user time), comparing the byte-code interpreter t_i `ocamlrun` time with the OcamlJit `ocamljitrun` time t_j (both in CPU seconds), and the speedup $\sigma = t_i/t_j$ (bigger values are better, and < 1 is a slowdown - short executions where OcamlJit is in fact *slower* than the byte-code interpreter because of the translation overhead.

Short executions are actually slower ($\sigma < 1$, a slowdown marked with $\neg$), and the short execution time is very unprecise (marked $\sim$). Natively compiled [with `ocamlopt`] programs (e.g. running `ocamlopt.opt *.ml`) perform about twice as fast ($t_o = 2.71$ vs $t_j = 5.91$) as OcamlJit running the equivalent byte-code (e.g. running `ocamlopt *.ml`).

The test programs from `ocamlc/tests` gives better speedup than compilation tests (i.e. running `ocamlc` or `ocamlopt`), perhaps because the OCaml compilers parses using lex and yacc technologies driven by C primitives (which are not accelerated by the OcamlJit program). When most of the compiler's work is not related to parsing (e.g. for `ocamlopt x86imkasm.ml`) OcamlJitgives a significant speedup w.r.t. `ocamlrun`, typically a factor approaching two. A speedup factor approaching two is not unusual with OcamlJitexecution.

We could not compile with the required OCaml compiler some interesting programs (like e.g. the Coq proof assistant `http://coq.inria.fr/` or the Cil C parsing kit `http://manju.cs.berkeley.edu/cil/`) and even MetaOCaml because they have not yet been ported to OCaml **3.08**[17], so we were not able to run them with our OcamlJit runtime.

5. CONCLUSIONS AND FUTURE WORK

Our OcamlJit did achieve some significant speedup (usually almost twice as fast as the bytecode interpreter), but some design choices may be questionable and suggest possible future work.

5.1 compatibility with OCaml runtime

Total compatibility between `ocamlrun` and `ocamljitrun` was our objective, and has been reached by OcamlJit(except for debugging OCaml programs with `ocamldebug`), even during execution, since the OCaml stack and heap always remains the same, and because OcamlJit maintains the mapping between byte-code addresses and equivalent generated machine code addresses (see §3.2.2 above). However, this mapping is costly (many branches and all closure applications goes thru the hash-table) and could be avoided[18] if closures remembered not only their byte-code address but also their native address. However, adding such a field to closure would have a significant memory cost, and would require a tailored runtime (e.g. changing the garbage collector, the serializer, etc...).

[17]Minor porting issues (to 3.08, or the OCaml CVS snapshot of late june 2004) include the `camlp4` preprocessor (whose API changed w.r.t. locations), and the name of C primitives (now prefixed with `caml_`).

[18]In an `ocamlc` execution, 15% of the executed byte-codes depends on this: they are applications, branches, returns, grabs, ...

Should changing the byte-code instruction set (hence changing the OCamlcompiler) be considered, adding one single new operation would help OcamlJit a lot: the `LABEL` byte-code (followed by a skipped word [to contain the native code address for JIT]) would tell that a function application is possible at that point. The OCaml compiler has to be changed to insert such byte-codes (when translating every **lambda** node in the "lambda" representation) at appropriate places; the `ocamlrun` byte-code interpreter would handle `LABEL` instructions as no-op (i.e. skipping them). But OcamlJit could write the generated machine code address in the word after this byte-code, so closures won't be changed (they still contain a byte-code pointer to the `LABEL` instruction), but could be applied in OcamlJit much quickly without any `caml_jit_pc_tblptr` hashtable access (see §3.2.2 above).

5.2 better code representation

The OCaml byte-code was designed for making byte-code programs compact, while being easily interpretable. Our OcamlJit system does not make any assumption on the byte-code sequence (as long as it is read-only), and should even work with byte-code files generated by other means than the OCaml compiler.

The stack based organization of the OCaml virtual machine (a feature shared with the Java JVM) may be questionable, since most operations require stack memory access. A register-based virtual machine (like Parrot[20]) might be more suitable for JIT translation, in particular on RISCy processors (on x86, there are too few registers so all are used, but with more registers -like on PowerPC or AMD64- the current OcamlJit approach leaves some of them unused). Alternatively, the JIT translator could analyze the byte-code and generate machine code using all the registers. The register-based VM approach requires some register allocator in the OCaml system, but the byte-code analysis would need a more complex and slower JIT translator. Another possibility would be to have the system translates (incrementally to machine code) a higher-level representation, like the "lambda" one.

If OCaml had *garbage collected code* pieces, i.e. if each closure contained a pointer to a garbage collected piece of byte-code, it would help both the toplevel and MetaOCaml systems; currently, most of the generated code (even if it is dead) is kept as long as it might be used in a closure. Garbage collecting the byte-code would help, but would require a significant change to the GC, which would have to scan the OCaml call stack to find currently active code pieces. The current OcamlJitwould be adaptable to such a scheme, since it keeps the OCaml (byte-code based) call stack, provided each code value is at a fixed address and is finalized by the GC, who would ask the JIT part to destroy the equivalent generated machine code. Garbaged-collected code sequences is in our opinion necessary for meta-programming reflexive systems (like J. Pitrat's Maciste [11]) and would be useful to MetaOCaml, and should help marshalling[7]. Hacking the actual Ocaml garbage collector might be tricky, so we could extend a simpler one[15].

5.3 another stack organisation

invocation	t_i byte	t_j jit	σ speedup	*notes*
`ocamlc -h`	$.01^{\sim}$	.15	$.06^{\neg}$	short exe.
chk^{500} `ocamlc -h`	$.01^{\sim}$	.06	$.17^{\neg}$	short exe.
`ocamlc callback.mli`	$.02^{\sim}$	.16	$.13^{\neg}$	short exe.
`ocamlc pervasives.mli`	$.06^{\sim}$	.18	$.33^{\neg}$	short exe.
`ocamlc oo.ml`	$.02^{\sim}$	.15	$.13^{\neg}$	short exe.
`ocamlc format.ml`	.29	.39	$.74^{\neg}$	short exe.
chk^{500} *idem*	.29	.37	$.78^{\neg}$	short exe.
`ocamlc *.ml`	2.89	2.55	1.13	
`ocamlopt *.ml`	7.44	5.91	1.25	
`ocamldoc` like *latex_test*	25.3	18.8	1.35	string, IO, GC
`ocamlopt x86imkasm.ml`	59.8	30.3	1.97	compiling a *C*-- file [13] of 3253 lines
`tests/sorts.byt`	49.3	22.6	2.18	sorting test
`tests/bdd.byt`	4.63	1.92	2.42	array
`tests/nucleic.byt`	0.67	0.30	2.23	small floating
`tests/almabench.byt`	8.8	4.16	2.11	floating benchmark
`tests/almabench.fast.byt`	8.64	4.02	2.14	floating benchmark (no bound checks)
`tests/kb.byt`	1.13	0.6	1.88	Knuth Bendix
`tests/quicksort.byt`	0.75	0.23	3.20	array quicksort
`tests/quicksort.fast.byt`	0.68	0.23	2.90	array quicksort (no bound checks)
`tests/soli.byt`	0.44	0.10	4.40	small solitaire puzzle
`tests/soli.fast.byt`	0.39	0.07	5.50	small solitaire puzzle (no bound checks)
`spamoracle` on Bible	4.92	3.17	1.55	learning Bayesian filter (on 4.4Mb text)

Table 2: running time and speedup (on x86)

Using the native machine stack might be helpful for a JIT implementation. The layout of a machine stack is constrained by the processor and the operating system, and can be formally described[10]. Perhaps OcamlJitmight share the native `ocamlopt` stack (and runtime), thus permitting a mix of native and byte code in OCaml.

Assuming that the implementation does not use any C callbacks, we could have some abstractly similar stack layouts on several machine architectures, hence giving the language a common (abstrace) view of the call stack. This requires of course a well designed JIT code generator, and take into account that OCaml(or similar mostly functional languages) do not need to point from the heap (or from variables) to (inside) the stack. Such an abstract stack organization might enable the reification of call stack subsequences, useful for partial continuations [6] and other powerful control structures [5]. Providing primitives to inspect the call stack is useful for reflection, and also for debugging.

We welcome any feedback regarding OcamlJitand its use in real size programs (running for more than a few seconds of CPU).

6. ACKNOWLEDGMENTS

Many thanks to the Cristal team at INRIA, in particular to François Pottier and Xavier Leroy for many fruitful discussions and for proof-reading a draft. Special thanks to Paolo Bonzini and Laurent Michel for their support on the GNU Lightning library, and to Walid Taha for explanations about MetaOCaml.

7. REFERENCES

[1] P. Bonzini and L. Michel. GNU Lightning library. `http://savannah.gnu.org/projects/lightning/`, 2004.

[2] C. Calcagno, W. Taha, L. Huang, and X. Leroy. Implementing multi-stage languages using ASTs, gensym, and reflection. In *GPCE'03*, 2003.

[3] D. R. Engler. Vcode: a retargetable, extensible, very fast dynamic code generation system. In *Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, pages 160–170. ACM Press, 1996.

[4] C. W. Fraser, D. R. Hanson, and T. A. Proebsting. Engineering a simple, efficient code-generator generator. *ACM Lett. Program. Lang. Syst.*, 1(3):213–226, 1992.

[5] C. A. Gunter, D. Remy, and J. G. Riecke. A generalization of exceptions and control in ML-like languages. In *Functional Programming Languages and Computer Architecture*, pages 12–23, 1995.

[6] Y. Kameyama. A type-theoretic study on partial continuations. In *IFIP TCS*, pages 489–504, 2000.

[7] J. J. Leifer, G. Peskine, P. Sewell, and K. Wansbrough. Global abstraction-safe marshalling with hash types. In *Proc. 8th ICFP*, 2003. To appear. Available from `http://pauillac.inria.fr/~leifer/research.html`.

[8] X. Leroy. The ZINC experiment: an economical implementation of the ML language. Technical report 117, INRIA, 1990.

[9] X. Leroy et al. the OCAML language (version 3.08). `http://caml.inria.fr/`, 2004.

[10] C. Lindig and N. Ramsey. Declarative composition of stack frames. In *13th Int'l. Conf. on Compiler Construction*, Barcelona, april 2004.

[11] J. Pitrat. Implementation of a reflective system. *Future Generation Computer Systems*, 12:234–242, 1996.

[12] I. Piumarta. CCG : dynamic code generation for C and C++. `http://www-sor.inria.fr/projects/vvm/realizations/ccg/`, 1999.

[13] N. Ramsey, S. Peyton-Jones, et al. The C-- language. `http://www.cminusminus.org/`, 2004.

[14] D. Rémy. Using, Understanding, and Unraveling the OCaml Language. In G. Barthe, editor, *Applied Semantics. Advanced Lectures. LNCS 2395.*, pages 413–537. Springer Verlag, 2002.

[15] B. Starynkevitch. QISH runtime software. `http://starynkevitch.net/Basile/qishintro.html`, 2003.

[16] B. Starynkevitch. OCAMLJIT software. `http://cristal.inria.fr/~starynke/ocamljit.html`, 2004.

[17] W. Taha. A gentle introduction to multi-stage programming. In D. Batory, C. Consel, C. Lengauer, and M. Odersky, editors, *Dagstuhl Workshop on Domain-specific Program Generation*, LNCS. Springer-Verlag, 2004. To appear.

[18] W. Taha et al. the MetaOCAML language. `http://metaocaml.org/`, 2004.

[19] W. Taha and T. Sheard. MetaML: Multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1-2), 2000.

[20] L. Totsch et al. The Parrot virtual machine. `http://parrotcode.org/`, 2004.

[21] R. Weatherley. libjit library. `http://www.southern-storm.com.au/libjit.html`, 2004.