UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél.:(1)39635511

Rapports Techniques

N°117

Programme 1
Calcul symbolique, Programmation et Génie logiciel

THE ZINC EXPERIMENT: AN ECONOMICAL IMPLEMENTATION OF THE ML LANGUAGE

Xavier LEROY

Février 1990

L'expérience ZINC: une mise en œuvre à l'économie du langage ML

Xavier Leroy*

Résumé

Ce rapport présente en détail la conception et la réalisation du système ZINC. C'est une mise en œuvre du langage ML, se fixant pour objectif de permettre l'expérimentation de diverses extensions du langage, ainsi que de nouvelles techniques d'implémentation. Ce système est conçu pour permettre la compilation indépendante et la production de petits programmes autonomes; le respect du typage est assuré par par un système de modules à la Modula-2. ZINC utilise des techniques simples et facilement adaptables, telles que l'interprétation de code abstrait; un modèle d'exécution raffiné compense en partie le ralentissement dû à l'interprétation. A noter également une représentation efficace des *records* avec inclusion (sous-typage).

The ZINC experiment: an economical implementation of the ML language

Xavier Leroy*

Abstract

This report details the design and implementation of the ZINC system. This is an implementation of the ML language, intended to serve as a test field for various extensions of the language, and for new implementation techniques as well. This system is strongly oriented toward separate compilation and the production of small, standalone programs; type safety is ensured by a Modula-2-like module system. ZINC uses simple, portable techniques, such as bytecode interpretation; a sophisticated execution model helps counterbalance the interpretation overhead. Other highlights include an efficient implementation of records with inclusion (subtyping).

*Ecole Normale Supérieure et INRIA Rocquencourt, projet Formel.

Contents

Chapter 1

Motivations

There is no cure for it.
(Samuel Beckett)

Originally designed as the control language of the LCF proof assistant [26], the ML language has grown to a general-purpose, very high-level language particularly well-suited to teaching and fast prototyping. A lot of work has been devoted to its implementation, not only to get efficient ML compilers, but also to build good programming environments for ML: among others, the original Edimbourgh ML, the Lazy ML of Göteborg developed at Chalmers University, CAML at INRIA, Standard ML of New Jersey at Bell Labs, Matthews' Poly/ML at Cambridge, ...This leads to believe that the area of ML implementation has been fully investigated, and that it is no more a hot research topic.

However, my own experience with the CAML system, both as a user and a developer, made me feel the need for yet another implementation of ML, with the following goals in mind:

- to get a small, portable implementation of ML. Current ML implementations require lots of resources, both in CPU power and in memory (for instance, CAML needs at least three MIPS and five megabytes of memory to get decent response time). And they work only on specific architectures (e.g. Vaxes and Sun workstations), since they usually generate native machine code. This restricts severely the potential audience of ML. For instance, there is no ML implementation running on a microcomputer, and this is the main obstacle to its getting widely used for teaching.
- to serve as a testbench for extensions of the language. One may fear that ML was a bit hastily standardized, since many extensions have been proposed lately, to integrate features coming from other computational paradigms such as logic programming, object-oriented programming, or communicating processes with the strong static typing, type inference, and higher-orderness of ML. Some of these propositions are extensions of the type system: records with structural subtyping [15, 31, 53], the type classes of Haskell [60], the dynamic objects [1]. Others introduce new mechanisms in the execution model: call-by-unification (narrowing), communicating processes. Finally, new efficient implementation techniques have been proposed, such as high-level program transformations and powerful static analysis [22].

It would be nice to see these extensions at work on a real implementation of ML. But my previous experience with CAML showed me that adding a feature to a full-sized implementation such as CAML requires much more work than strictly needed, due to the fact that this is a large system, not written in a very modular way, with lots of unexpected dependencies among the various parts, serious bootstrapping problems, and so on. Therefore, CAML is not suited to experimentation anymore, and I have felt the need for a much smaller implementation, allowing to toy with new features quite easily.

- for pedagogical purposes. One of the best ways to really understand a language is to look at an actual implementation. While, for instance, toy implementations of Pascal abound in the literature, there is no work describing in detail an actual implementation of ML. Peyton-Jones' book [50] is an excellent introductory text, but not totally relevant to ML, since it uses Miranda, not ML, as its source language, and it focuses on lazy evaluation through graph reduction, while ML usually has strict semantics. A few papers on the Standard ML of New Jersey implementation have been published [5, 6, 4], but they are fairly high-level — they don't show much of the code! And regarding the CAML system, nothing has been published yet, Ascánder Suárez's thesis [59] is still forthcoming, and in the meantime the curious mind is left with 70000 lines of very sparsely commented source code.

Such were my motivations for developing yet another ML implementation. And now, the name of the game. ZINC is a recursive acronym for "ZINC Is Not CAML", and this name emphasize the fact that it does not have exactly the same intents and purposes as one of the existing implementations. (And to insist on this work being mostly practical in nature, I nicknamed it "un projet informel".)

This report presents the state of the current ZINC implementation. Chapter 2 gives the main design decisions, the guiding principles for the implementation. Chapter 3 introduces the execution model used, described as an abstract machine. Chapter 4 shows how ML values are represented, and how memory is structured. Chapter 5 details the compiler, with actual ML code taken from the source. Finally, chapter 6 gives an overview of the runtime system. Concluding remarks are to be found in chapter 7, and benchmark results in appendix A.

Chapter 2

Design principles

This chapter presents the main decisions I made when designing ZINC, and why they lead to a system differing slightly from classical ML implementations.

2.1 Modules

Decomposing programs in several relatively independent parts called *modules* has emerged as an efficient way to tackle programming in the large [49]. Programming with modules can be done in any programming language, even Fortran — provided that the programmer has the required fortitude. It is easier, however, when the language provides features to express dependencies between modules, allowing the compiler to check automatically their consistency. And from a more practical standpoint, language support for modules is the only way to combine separate compilation with strong static typing. Such are the reasons why ZINC had to provide some kind of modules.

2.1.1 Modules in Standard ML

A system of modules for the ML language has been proposed by MacQueen [42], and it is now part of the Standard ML definition [28]. It introduces very powerful notions, such as functors, making it one of the most advanced system of modules ever integrated into a programming language. And it has received a fairly satisfactory formal treatment, if a bit complicated [43]. Nonetheless I had to dismiss it as inadequate to the ZINC system.

First, it is still unclear whether such a sophistication is really needed, and if it really helps when dealing with large software systems. We lack experience with very large software systems written in a modular way in Standard ML. Second, Standard ML modules are not particularly convenient to use. The problem is that to use some values or types defined in another module (to "import" them), the programmer has to insert their signature (their type in case of a value, their constructors in case of concrete types) at the beginning of the current module. This is just unbearable in case of very frequently imported values, such as those of the so-called "prelude" of ML implementations: one has to declare `map : ('a -> 'b) -> 'a list -> 'b list` at the beginning of each module usign `map`, that is to say almost every module. Finally, inconsistencies between modules are not detected at compile time, but only at link time, when all modules have been written and compiled. This goes against "true" separate compilation.

More simple module systems do not have these drawbacks. Take for instance the one of Modula-2 [63]. This is a very simple module system (one can view it as a mere cleanup of the C practice of "include files"). Yet there are huge systems (thousands of modules) written in Modula-2, and such systems demonstrate that even this very simple module system suffices to break these systems into manageable parts and to handle their dependencies. In addition, it is fairly convenient to use. All informations regarding imported globals do not have to be retyped at the beginning of each module using them, since they are stored once for all in the "interface" files of the modules defining those globals. Finally, this allows to report type clashes between modules during compilation, as soon as possible. Therefore, it seems that Modula-2-like modules are more in accordance with the "economy of means" principle of ZINC.

2.1.2 Modula-like modules for ML

As a simpler alternative to Standard ML modules, I propose a system of modules very close to the one found in Modula-2 [63]. It uses an interface file associated with each module and declaring what it exports, and the dot notation (e.g. `module.data`) to refer to a value (or a type) defined in another module. It is extended by a simple mechanism for type abstraction, as in Modula-3 [47]. Among functional languages, a similar module system can be found in Cardelli's Quest language [14], where modules are built on top of a more elementary data structure called abstract tuples (i.e. generalized second-order dependent products). It requires second-order typing, however, and it leads to a slightly different treatment for "manifest types", that is, types exported with their definition. Some foundations for this approach to modules, and especially the way it handles type abstraction, can be found in [16].

A module in ZINC is composed of two parts: an *interface* and an *implementation*. The interface part declares all that the module exports (i.e. makes publicly available to the outside):

- value identifiers, with their type: `value f : int -> int;;`
- definition of concrete types: `type foo = Bar of int | Gee of bool;;`
- abstract types: `type foo;;` The definitions of such types, that is their constructors in the case of ML concrete types, remain hidden, making it impossible to directly build or destructure a value of that type outside of the module. In other words, abstraction is ensured here by representation hiding.
- declaration of exceptions: `exception missed of int;;`

The implementation part is a sequence of usual ML phrases, binding identifiers to (the value of) expressions, defining concrete types and exceptions, or evaluating single expressions. It must provide definitions to the values and the abstract types declared by the interface. It may contain other bindings, but these are private, that is not known outside of the implementation.

As an example, here is the interface of a module `set` implementing the set data structure:

```
type 'a set;;         (* its representation remains hidden *)
value emptyset: 'a set
and singleton: 'a -> 'a set
and union: 'a set -> 'a set -> 'a set
and empty: 'a set -> bool
and member: 'a -> 'a set -> bool;;
```

A corresponding implementation is :

```
type 'a set = EmptySet | Singleton of 'a | Union of 'a set * 'a set;;
let emptyset = EmptySet
and singleton x = Singleton x;;
let rec empty = function
    EmptySet -> true
  | Singleton _ -> false
  | Union(s1,s2) -> empty s1 & empty s2;;
let rec member x = function
    EmptySet -> false
  | Singleton y -> x=y
  | Union(s1,s2) -> member x s1 or member x s2;;
let union s1 s2 = Union(s1,s2);;
```

To use one of the values and types provided by a module outside of this module, the dot notation is used, that is `module.data` to refer to the identifier `data` exported by the module named `module`. The same notation is used for types, and it can be used in interfaces as well as in implementations. In other words, modules are imported implicitly, with the signature found in their interface files. For instance, we could write:

```
type intset_tree =
    Leaf of int set.set
  | Node of intset_tree * intset_tree;;
let rec ints_of_intset_tree = function
    Leaf s -> s
  | Node(t1,t2) -> set.union (ints_of_intset_tree t1) (ints_of_intset_tree t2)
;;
let contains_zero is = set.member 0 (ints_of_intset_tree is)
;;
```

2.1.3 Giving sense to non-qualified idents

Composite identifiers such as `module.data` are sometimes called "qualified identifiers". In Modula-2, non-qualified identifiers, that is `data` without the `module.` part, refer to values and types belonging to the module being defined. This is just syntactic sugar: non-qualified identifiers such as `data` could be transformed into m_c.`data`, where m_c is the name of the module being compiled, and lead to an equivalent program.

In ZINC, this idea of automatic completion of non-qualified identifiers is carried further. The compiler maintains a list of "default" modules, and when it encounters a global identifier s without qualification, it searches through that list to find a module m which defines s; then, it substitutes $m.s$ for s, thereby disambiguating the global name s. The list of default modules always includes the module being compiled (searched first) and a special module `basics` defining very primitive operations such as `+` (searched last).

In addition, two compilation directives are provided, `#open"`*module*`"` to add *module* to the list of default modules, and `#close"`*module*`"` to remove *module* from that list. They are very convenient for library modules defining values and types very frequently used (the "prelude", for instance): just insert `#open"prelude"` at the beginning and you won't have to type `prelude.map prelude.fst (a prelude.:: b)` anymore, `map fst (a::b)` will work just fine.

This disambiguating process is purely syntactical in nature; we only need to know what the binding constructs are (in order to tell a global variable from a local variable) to be able to transform a program into an equivalent one where all global identifiers are qualified. This process does not interfere with typechecking, as opposed to classical overloading, which uses types to resolve ambiguities, making the type inference algorithm much more complex, and causing it to fail sometimes.

Yet this simple mechanism proves very useful in common situations. First, it makes the transition to modular programming easier, since to program in old, non-modular style, without qualified idents, it suffices to "open the world" at the beginning of the file, that is to put a `#open ...` for each module used. Second, it allows to choose between several implementations of the same operations. For instance, operations on strings such as getting character number n, extracting a sub-string, and so on, usually come in two flavors, a cautious one, which checks bounds and so on, and a fast one, which does not check anything. Usually, the user chooses the desired flavor through compilation switches. With modules, two modules are defined, `cautious_string` and `fast_string`, which exports exactly the same names, but with different semantics; the user can explicitly request one or the other (e.g. `cautious_string.nth_char` or `fast_string.nth_char`), or if one of the two semantics, the fast one or the cautious one, is used throughout a file, "open" one of the modules and use non-qualified names (e.g. `#open"fast_string";; ... nth_char ...`).

Similarly, ML systems usually offer several kinds of numbers: integers, floating-point numbers, arbitrary-precision rationals, with the same basic operations (e.g. `+`, `*`) defined on each. These operations are usually named `add_int`, `add_float`, `add_ratio`, to distinguish between them. Of course, it would be nice to use the usual syntax `+` instead, and have the system choose automatically between `add_int` and `add_float`; this is one of the first motivations for overloading. With ZINC modules, we could type `1 int.+ 2` or `1.2 float.+ 3.4` to precisely state which kind of addition we want, while retaining the nice syntax and evocative power of the `+` symbol. And if we use solely integers throughout a file, a `open"int"` at the beginning allows the usual syntax `1 + 2`. To conclude, this mechanism makes the need for overloading less acute, and while it does not offer the same power as "true" overloading, it does not have its drawbacks either.

2.2 Efficient currying *vs.* N-ary functions

There are no N-ary functions in ZINC, but this lack is deliberate, and I shall justify it.

2.2.1 The need for functions with several arguments

Functions with several arguments are a common feature of most programming languages, even those with functions as first-order values (Scheme, well-implemented Pascal, ...). Passing several parameters is just as easy as passing one, after all. Yet ML has been impervious to this notion, and still offers but unary functions. Functions with several arguments, as $f(x, y) = x + 3y$, are implementing either as a function taking a pair:

```
let f = function (x, y) -> x + 3 * y in f(5,7);;
```

or as a function returning a function (this encoding is known as *currying*):

```
let g = function x -> function y -> x + 3 * y in g 4 6;;
```

Both are inefficient, however, with usual execution models such as the FAM [11] or the CAM [20]. Calling the uncurried version means allocating a tuple to hold the arguments; this cannot be avoided, since the called function may use the tuple itself as a value (e.g. `fun (x,y as z) -> (z,z)`). And the curried version builds intermediate closures (one for each argument but the last), which are immediately applied and then never reused. On the other hand, for "true" n-ary functions, we have much more efficient calling conventions (stack-based or register-based) that do not allocate anything in the heap.

Therefore, it seems that to have efficient function calls, we need to add n-ary functions to ML. Yet there is no universally accepted way to do it. A first approach is to explicitly introduce n-ary functions in the language, as in Scheme [58]. But we don't know yet how they should behave w.r.t. higher-order functions, polymorphism, and type inference: should a binary function belong to the type scheme $\alpha \rightarrow \beta$? If not, some polymorphism is lost; if so, then the type variable α represents not only all values, but also sequences of values, yet this is not the case for α in α `list` ...This problem is even more acute if we consider functions with p results as well, for the sake of symmetry, as in Amber [12] for instance. An alternate approach is to leave the ML language unmodified and to transparently replace functions taking n-uples into n-ary functions. But this requires a non-trivial static analysis, which does not work very well with higher-order functions.

2.2.2 The power of currying

Before starting to argue on n-ary function, let me mention a fact which is overlooked in the discussion above: curried functions and the corresponding uncurried ones (functions taking a tuple as well as n-ary functions) are not equivalent. The curried version is in a sense more powerful, since it allows partial application. In the example above, `(g 4)` is a legal object, we can do, for instance,

```
map (g 4) [1; 2; 3; 4; 5];;
```

With the uncurried function `f`, we would have to write instead

```
map (fun y -> f(4,y)) [1; 2; 3; 4; 5];;
```

Here, the differences are not very important. But partial evaluation may become extremely efficient if the called function performs some computation between the passing of the first and the second argument:

```
let h x = let z = fibonacci x in fun y -> y + z;;
map (h 30) L;;
```

In this example, `fibonacci 30` is computed once, not once for each element of the list `L`. Therefore, partial application of a curried function leads to a *partial evaluation* of the body of the function. This may save a lot of computation, and expresses very elegantly invariants when used in conjunction with iterators.

That's why I chose to forget about n-ary functions, and concentrated instead on curried functions. The challenge was to find an abstract machine where a multiple application to k arguments, that is $(\ldots((M\,N_1)\,N_2)\ldots N_k)$, is more efficient than a single application to the k-uple of all arguments $(M\,(N_1,\ldots,N_k))$, and if possible almost as efficient as applying a k-ary function. This means at least not building any intermediate closures to do so. A detailed account of this quest is given in the next chapter.

This approach has a few drawbacks, however. First, with ZINC's abstract machine, it is the case that multiple application to k arguments does not allocate any closures, but it is still slower than applying a k-ary function, since some run-time tests have to be performed. Also, functions computing several results still have to be encoded as functions returning a tuple; functional encodings similar to currying (e.g. transforming `fun x -> e1 e2` into `fun f -> fun x -> f e1 e2` are definitely too clumsy, if more efficient.

2.2.3 Right-to-left evaluation order

This emphasis on multiple application has an unexpected consequence on the evaluation order of application $(M\ N)$. As far as I know, all strict functional languages use left-to-right evaluation, that is they evaluates M first, then N. But left-to-right evaluation does not put up well with multiple application, as we shall see now.

To evaluate $(M\ N_1\ \ldots\ N_k)$, the usual, left-to-right strategy is to reduce M first, then evaluate N_1, then $A_1 = (M\ N_1)$, then N_2, then $A_2 = (A_1\ N_2)$, and so on until $A_k = (A_{k-1}\ N_k)$. The order of computation is therefore $M, N_1, A_1, \ldots, N_k, A_k$. The evaluations of the arguments N_i and of the partial applications A_i are interleaved. That's the reason why we need to actually build the closures representing $A_1, \ldots, A_{k-1}$: when the reduction of M reaches A_i, we must suspend this reduction and start evaluating N_{i+1}, and in term-as-code models, suspending means allocating a closure.

To be more efficient, we need to precompute $N_1, \ldots, N_k$ before starting to reduce inside M; that way, we won't have to stop each time we cross a λ-abstraction in order to receive a value for the parameter, since these values are already available. Hence, an efficient multiple application will be evaluated left-to-right in the following order: $M, N_1, \ldots, N_k, A_1, \ldots, A_k$. This is not consistent with the case of simple applications.

With a right-to-left evaluation order, the evaluation order is $N_k, \ldots, N_1, M, A_1, \ldots, A_k$, regardless of any special treatment for multiple application. Right-to-left evaluation is therefore necessary to have an efficient but transparent mechanism for multiple application.

Evaluation order is more than an implementation issue: due to the imperative features of ML (exceptions, mutable values, input/output performed as side effects, ...), the order in which the subexpressions of an expression are evaluated is meaningful: a different evaluation order may change the semantics of a program. Some especially vicious examples can be found in Huet [30], where a left-to-right evaluation order is assumed, and exceptions are used to share data representations as much as possible; with a right-to-left evaluation order, the functions return the same results, but all sharing is lost, and the program performs much more heap allocation. Therefore, the user must know the evaluation order that the compiler implements, in order to be able to know what his programs do.

Another standpoint is to leave the evaluation order unspecified, except for some constructions with strongly sequential connotations, such as the sequence operator "`;`" and the declaration `let...in...`. This is the case in C, for instance. The motivations are mostly practical: modern code generation algorithms, such as the dynamic programming algorithm of Aho and Ulmann [3, chap. 9], choose the order of evaluation of subexpressions so as to minimize the number of registers needed. In this case, we can only give non-deterministic semantics to programs. Then, programmers have to write their program so that they can only have one meaning, taking advantage of the few constructs whose evaluation order is specified. Notice that this constraint usually leads to clearer programs, since temporal dependencies have been made explicit. This standpoint is not yet widely accepted in the Lisp/ML community, however.

2.3 Execution methods

What kind of code should the ZINC compiler generate? This implementation issue engages many aspects of ZINC, in particular portability, execution speed, and interactivity. I shall review the most common approaches to justify the choice of direct interpretation of abstract machine code.

2.3.1 Native code generation

Real compilers for real languages generate native machine code (or maybe assembly language), in order to produce compiled programs that run as fast as possible on the given hardware. However, this is the "blood, sweat and tears" approach: writing a code emitter that generates good machine code, say, comparable to hand-written assembly code, is by no way trivial. Chapter 9 of [3] gives a good overview of that task. On conventional, CISC architectures such as the VAX or Motorola 68000 family, the main difficulty is to choose the right sequence of instructions among the huge set of instructions and addressing modes, and coping with their asymmetries (some instructions don't work with all addressing modes, some registers are specialized for some operations, ...) On RISC architectures, instruction selection is easier, but new problems arise in trying to exploit new features such as delayed branches (instruction scheduling). In both case, good register allocation is crucial to reduce memory accesses, and this also needs fairly complex algorithms. In addition, a code emitter is specific to one processor, so this work must be done again from scratch for each new target processor. And the correctness of the code emitter is often compromised by clever optimizations.

To automate part of this work, many attempts at automatic generation of a code emitter, starting from a formal description of the target machines, have been made. A popular, well-documented example is the GNU C compiler [57]. However, real architectures are hard to describe formally, so it seems that descriptions are either over-simplified, leading to very naive code generators, or almost as difficult to write than the code emitter itself.

2.3.2 Abstract machines and code expansion

To write more portable compilers, a popular approach is to generate code for a given machine, then translate this code into native code for various target machines. The intermediate machine need not be an actual one: it can be an abstract machine, specially designed to fit the source language. For instance, the Le_Lisp system of Chailloux *et al.* [18] is entirely built around the LLM3 abstract machine [17]. In this approach, it is intended that the machine-independent part of the compiler makes all hard decisions itself (e.g. register allocation). Then, the mapping of the abstract machine onto a real architecture is essentially trivial: each abstract instruction is expanded into a sequence of real ones, regardless of the context. Such code expanders are quite easy to write, therefore porting a system to another architecture becomes manageable (a competent hacker needs but one or two weeks to port the whole Le_Lisp system). The problem is that they usually generate ugly code, with a lot of redundancies, and using a tiny subset of the target machine's abilities.

To get more efficient code, the obvious solution is to perform peephole optimization after expansion, to detect and remove useless instruction sequences, condense several instructions into one, ... The problem is that good peephole optimizers are hard to write, and they are highly machine-dependent, just as the code generators considered above. An alternative is to complicate the expansion of abstract machine code into native code. For instance, a well-known trick to map stack-based abstract machine on register-based processors is to try and simulate stack moves during translation, in order to replace a "push" instruction and its matching "pop" by moves to and from a free register, which is less costly [50, p. 328] (see also [52] for a description of a real-sized implementation, used in the CAML system). In both cases, it is usually too late to generate truly good code, since the abstract machine code contains much less information than the original program. For instance, the abstract machine code fixes the evaluation order, so it is almost impossible to permute the evaluations of subexpressions (and to know if we're allowed to do it, that is if those subexpressions do not have side-effects). When the compiler is particularly dumb, it might be possible to infer such information from the abstract machine code, but this is clearly absurd (it would be easier to skip the compiler entirely and start directly from an abstract syntax tree!)

2.3.3 Translation to another high-level language

The main weakness of the previous approach is that the output of the compiler, that is abstract machine code, is too low-level to allow good code generation. Let us investigate the converse approach: translating the source program to some other high-level language, and then using any good compiler for this language. It seems we could save a lot of work this way: all the hard work, that is generating good code for many machines, would have been done by someone else, all we would have to do is to deal with the unique, very high-level features of ML such as type inference and pattern-matching compilation. In addition, if we choose a sufficiently standardized, popular target language, then our implementation would be highly portable, and fairly efficient as well if good compilers exist for the target language. There is one point we must be careful about, however: to completely hide the target language from the user.

Two well-known languages are obvious candidates to be our target language. The first is some dialect of lexically-scoped Lisp, such as Common Lisp or Scheme. These languages have important points in common with ML: they have functions as first-class values, and everything has a value (i.e. there are no statements, solely expressions). In addition, a lot of work has been devoted to implementing efficient compilers for them (for instance, the Orbit compiler for Scheme [35]). I dismiss Lisp, however, for the following reasons. First, there is not yet a good Lisp compiler on every machine, and the fact that many dialects still coexist (Lisp is not standardized yet) restricts further the number of potential users of our implementation. Second, Lisp implementations are usually built around a toplevel, so they are not well-suited to be used as part of a "pipe" of programs; therefore it might be hard to hide the underlying Lisp system from the user (as developers and users of ML 6.2, the Le_Lisp-based ancestor of CAML, know quite well). Finally, Lisp is too high-level: it does not give access to the innards of the machine. For instance, most Lisp dialects are unable to express their own garbage collector and runtime system, or to talk directly to the operating system. Therefore, we have to use the services provided by the implementation only, and these are not necessarily well adapted to ML. A typical example is the memory model: Lisp puts a strong emphasis on "conses" (pairs), to the point of neglecting other data structures such as arbitrary tuples, or providing them, but with an inefficient implementation (as in Le_Lisp [18], for instance): they are so scarcely used! But, in ML, triples or singletons are almost as frequent as pairs, so the Lisp memory model is not particularly well-adapted to encoding ML values, but at the same time Lisp prevents us from building a memory model specially adapted to ML!

A better target language would therefore be C. A C compiler is available on all Unix machines, Unix is getting widespread, and other platforms, especially in the microcomputer field, are switching to C as their main development language. In addition, C is in the process of being standardized, and even if most Unix C compilers are not ANSI-standard compliant, they all implement a core language fairly precisely defined in the first edition of [33], and their idiosyncrasies are minor (nothing that cannot be overcome with a handful of macros!) Due to very accommodating semantics, C is usually compiled very efficiently. In addition, it gives full access to the innards of the machine, thanks to pointer types, pointer arithmetics, and a very laxist "cast" operator. And Unix C compilers are "batch" compilers, designed to be used as part of a pipe. Therefore, it seems that C is the portable assembly language we were looking at. We are not the only ones to think so, since C is already the target language of the original Bell Labs implementation of C++, of the DEC and Olivetti implementations of Modula-3, and of Kyoto Common Lisp, to name a few.

Generating C has a few drawbacks, however. First, it makes interactivity difficult, since calling a batch C compiler takes too much time to do it for each phrase entered at toplevel; we have to revert to a ML interpretor to run toplevel computations. Second, interfacing a garbage collector with a C program is not easy and quite messy. A piece of non-portable assembly language is needed to get the contents of the registers and give them to the garbage collector, since these contents are roots of the memory graph. But these roots are ambiguous: a register may contain a compiler-generated temporary whose bit pattern looks like a pointer in the heap, without being a pointer in the heap. The object pointed by this ambiguous root cannot be freed, nor relocated. Therefore, the usual, efficient copying collectors cannot be used, and we must use either inefficient non-compacting collectors, or "mostly copying" collectors, such as the one of [8], which are more complex.

2.3.4 Interpreting abstract machine code

Using an abstract machine is not a bad idea *per se*: it is a very convenient way to explain and precisely specify an execution model. What's not very satisfactory is to try and map it on existing hardware. Another approach is to use a simulator for the abstract machine, that is a program which interprets directly abstract machine code (conventionally called *bytecode* in this case).

Interpreters have a bad reputation in the field of programming languages; in most people's mind, it is associated with Lisp (at best) or Basic (at worst). The approach considered here is quite different: we do not interpret the source program directly, but first compile it into a very low-level code. Parsing, scoping, symbol table handling, reduction of high-level constructs into low-level ones, and all other complex jobs have already be performed statically by a compiler; all the interpreter has to do dynamically is to emulate some low-level machine.

This approach addresses the issue of portability. The abstract machine is fixed, it does not depend on any actual hardware, therefore the compiler is totally machine-independent. And if the bytecode interpreter is written in some popular, high-level language such as C, it can run on any machine having a C compiler with few, if any, modifications. At worst, one may have to rewrite entirely the bytecode interpreter to port it to another machine; but this is a small and simple program, compared with the compiler, which has not to be modified.

More generally speaking, we get full control over the target code, at last. In previous approaches, the final output of the compilation line was always native machine code, so it was impossible to tailor it to our needs. This is not the case anymore. For instance, it is easy to desing a totally machine-independent bytecode, which can run unchanged on any kind of machine; this is important in case of heterogeneous networks. We can also require the code to be fully "standalone", that is relocatable and without references to external data. Then, a piece of code can be manipulated just as any other data, for instance written to a file, or sent through a network; this is crucial for functional languages, which claim that functions are first-class values, just as integers or strings. Cardelli's Amber language [12] was the first to allow this, thanks to a bytecode designed with this requirement in mind [13]. Finally, bytecode which is machine-independent and "standalone" at the same time opens new perspectives: for instance, one can imagine several interpreters running on the various machines of an heterogeneous network and distributing computation among themselves by exchanging pieces of code through the network ...

The main drawback of interpreting bytecode is, of course, execution speed: the infamous "interpretative overhead". Let us consider this issue more closely. The interpreter contains code to execute each instruction of the abstract machine. This code is roughly speaking the one that expansion to machine code would produce. Therefore, that's not where the slowdown introduced by interpretation comes from. It comes from the following additional computations needed by interpretation:

- additional encodings. If it is written in a high-level language, the interpreter might not be able to manipulate directly objects used by the bytecode, and some kind of encoding becomes necessary. For instance, it may be difficult to store the registers of the abstract machine in actual registers of the host processor. Similarly, an interpreter written in Pascal cannot perform pointer arithmetic, therefore it has to represent the heap of the abstract machine as an array, with addresses as offsets in the array, and memory access in bytecode becomes array indexing, which is slower than pointer dereferencing.

- instruction fetching and decoding. This consists in reading the next instruction from the bytecode stream, extracting the possible operands from the operation code, and branching to the part of the interpreter which will execute the instruction read. All this computation is performed in hardware or microcode in the case of an actual machine.

- unspecialized code. The code executing one bytecode instruction is generic, since it must cope with all possible values of operands. Sometimes, however, cheaper instruction sequences could be used; for instance, it is usually faster to increment an integer by one than to add two integers; but an `Add` instruction whose second operand is 1 will of course be executed by a generic addition, not the cheaper increment operation.

The first issue, additional encodings, is addressed by writing the interpreter in some language giving full access to the innards of the host machine. Assembly language is an obvious candidate, but it is not portable. C is portable, and gives almost the same facilities to "talk" directly to the host processor, thanks to pointer arithmetic, and the ability to put some local variables in registers.

The third source of inefficiency, unspecialized code, is not very important in practice (much less than the cost of instruction fetching!), and can be easily fixed by introducing new, specialized instructions in the bytecode, e.g. a `Succ` instruction to perform the same work as a `Add` instruction with 1 as second operand.

The second issue, the cost of instruction fetching, is critical. First, the decoding of opcodes must be reduced as much as possible. This prohibits putting addressing modes in the opcodes. The location of the operands must be implicit, and not explicitly contained in the opcode. That's the reason why bytecode interpretation is much more suited to stack-based machines than to register-based machines. With a stack-based machine, it is possible to have a very simple format for instructions: one byte to hold the opcode, and nothing else; additional operands (e.g. constants) are stored in the following bytes. Then, instruction decoding and fetching is simply one memory access followed by one jump through table. In C, we would write, for instance:

```
char * pc;
while(1)
  switch(*pc++) {
    case 0: /* Push  */  *--stack_pointer = accu; break;
    case 1: /* Pop   */  accu = *stack_pointer++; break
    case 2: /* Quote */  accu = *(long *)pc; pc += sizeof(int); break;
    /* etc */ }
```

This code looks optimal. Yet in case of simple instructions such as `Push`, instruction fetching (i.e. the `switch` construct) takes more time than actual execution (the code performing the `Push`). This means that interpretation slows down execution by a factor of two. To lower this overhead, there are two things we can do.

First, we can reduce the number of instructions to be executed. This means of course having more complex instructions, performing more work. One way to achieve this is to condense frequently used sequences of instructions into one instruction. For instance, assuming that `Quote` loads a constant in the accumulator and `Push` stores the accumulator on top of a stack, the sequence `Quote <cst>; Push` is quite common, and could be replaced by a single instruction `QuotePush <cst>`, which will execute much faster, since it requires one instruction fetching instead of two. And more complex instructions have an additional benefit: the bigger the C code executing an instruction, the more it can be "optimized" by a good compiler.

We can also perform partial evaluation of the instruction fetching and decoding loop. For instance, we can represent an instruction by the address of the routine interpreting it, instead of its opcode. That way, the jump through table is replaced by a cheaper indexed jump. If C understood indexed jumps, this would read:

```
label * pc;
  goto *pc++;  /* To start */
Push:
  *--stack_pointer = accu; goto *pc++;
Pop:
  accu = *stack_pointer++; goto *pc++;
/* etc */
```

To each instruction, we would associate a routine that executes it, and then fetch the next instruction, that is the address of the next routine to execute, and directly jump to it. This technique is known as *threaded code interpretation*, and is commonly used to implement Forth, for instance [41].

It cuts down interpretative overhead by a factor of two. Alas, I did not succeed in implementing this technique in C, so it requires some assembly language. This seems to be the most efficient interpretative technique. The next step toward partial evaluation is of course to suppress completely instruction fetching by concatenating directly the routines executing the instructions, in the right order, but this is exactly expansion to native machine code, as presented above!

To summarize, interpreting code for an abstract machine seems to be the only way to get a truly portable implementation of ML, as well as to be able to manipulate code as any other value (e.g. writing a function to a file). The overhead of interpretation is not negligible, but can be lowered by careful design of the abstract machine (to reduce the number of instructions needed) and using some tricks for the interpreter (e.g. threaded code instead of bytecode). By using these techniques in ZINC, I tried to see whether they suffice to make this overhead tolerable in practice.

2.4 Toplevels considered harmful

This section deals with the way users can interact with an ML compiler.

2.4.1 The toplevel-oriented approach

Usual ML implementations, such as SML-NJ or CAML, are essentially built around a "toplevel" loop. The spirit is the same as the famous "read, eval, print" loop of Lisp. Unlike Lisp, every phrase is compiled on the fly, and then executed to get its value, so this is actually a "read, compile, execute, print" loop. This leads to fully interactive systems, perfect for learning the language. In particular, it is fascinating to see not only the value, but also the inferred type of the phrase just typed. In addition, toplevels give an easy way to "talk" to the system, that is to specify compilation options, load a file, set a breakpoint on a function, and so on: it suffices to call some system functions, just as if they were user-defined. To trace function `foobar`, one simply evaluates `trace "foobar"`.

Of course, it is possible to load the contents of a file, but this is almost nothing more than input redirection. In particular, all phrases of the file are recompiled at each loading. To save some work here, CAML is able to put the compiled code in another file, and then load the compiled code directly, without recompiling. This separate compilation facility is totally unsafe, however, since the file was typechecked when compiled, in a given global environment, but nothing prevents it from being loaded later on, in a different environment. For instance, a file referring to a function `foo`, and compiled in an environment where `foo` has type `int->int`, can be loaded in a core image where `foo` is undefined, or (worse) is defined but with another type.

To prevent this from happening, module systems were added to SML as well as to CAML. Basically, a module must declare explicitly what global values it expects to find in the current global environment, along with their type, and checks at loading time that these globals are really defined, and with the expected type. That way, type safety is ensured, even in case of separate compilation.

This toplevel-oriented approach has serious flaws, however. First, "batch" use is sometimes useful, especially in the Unix world. The Unix programming environment is designed with "batch" compilers in mind, so that they can easily be composed with other general-purpose utilities — programs that just perform one simple task, but perform it well: dependency handlers (`make`), revision control systems (`sccs`, `rcs`), text preprocessors (`sed`, `m4`, `awk`), and so on. A toplevel-based implementation cannot be easily piped with one of these; for instance, it is almost impossible to use `make` to handle the dependencies of a program. With toplevels, the only way to have all the facilities above is to integrate them in the system. This is impractical, since the system gets bigger and bigger, and users have to learn again how to use these facilities.

Second, the fact that there is but one global symbol table, that gets updated when a module is loaded, is sometimes misleading. After recompiling and reloading a module, the bindings performed by the old module are not thrown away, they are simply hidden behind the bindings performed by the new module, but they can remain apparent if some exported globals were renamed. For instance, I have a module exporting a function `fobar`, and another importing and using `fobar`. Later, I realize that `fobar` is misspelled, so I rename it to `foobar` in the first module, but forget to do so in the second. In addition, I discover a bug in the definition of `foobar`, so I modify it. Then I reload both modules. No errors occur, since there is still a global named `fobar` in the environment. And the second module still uses the old, buggy version of `foobar` (uh, `fobar`, I meant). Of course, if we started again from scratch, the module system would catch this mistake. The problem is that toplevels keep the history of definitions made since the beginning of the session, therefore several successive versions of the user's program may coexist in memory, and interact in strange ways.

Finally, the user's program is loaded into the same process running the toplevel, the compiler, the debugger, ...; there is no way compiled code can be executed otherwise. As a consequence, it is not possible to produce truly standalone programs: even if it is possible to produce executable "core images", which contains the user's program at startup, these "cores" still contain the whole system (several megabytes of code), even if it is no more useful. And this makes bootstrapping more complex: some tricks are necessary to get a core image of the latest version of the system which does not contain in addition the code of the previous version, used to compile the latest one. More generally speaking, there is no distinction at all between the user's program and the meta-level programs (toplevel, compiler, ...), they both live in the same world, and this is often absurd, especially in the case of bootstrapping.

2.4.2 The standalone-oriented approach

As an alternative to toplevel-oriented implementations, I propose an architecture *a la* Unix, oriented toward separate compilation and the production of small, standalone programs. The heart is a standalone compiler, taking one source file, either the interface or the implementation of a module, and producing the corresponding object file, a compiled interface or a stream of symbolic code respectively. When all the modules of a program have been compiled, a standalone linker gathers the object code files together, fixes cross-references and the like, and produces an executable file. In case of bytecode interpretation, this "executable" actually contains bytecode, and must be run through the bytecode interpreter. These three programs, compiler, linker and interpreter, are all we need to produce standalone programs, and to bootstrap the compiler and the linker if they are written in ZINC.

They do not address the issue of learning the language, nor testing and debugging programs, however. To do so, some interactive system has to be provided. The idea is that most parts of the compiler and the linker can be reused in an interactive context, if they are able to work phrase by phrase; then, a phrase can be compiled, linked with the phrases already defined, and executed on the fly. That's all what we need to build a conventional toplevel *a la* CAML. But since the toplevel is no more the basic mechanism provided to use ZINC, it might be interesting to specialize it for teaching and debugging purposes, by restricting it on the one hand (prohibit compilation and loading of entire modules, for instance), extending it on the other (being able to ask for the values of "hidden" variables, such as local variables of a closure, or private global variables of a module), and even changing the semantics of some operations (for instance, `let f = E` where `f` is already defined could be interpreted as a *redefinition* of `f`, replacing all previous occurrences of `f` by `E`, instead of a *rebinding* of `f`, which does not update previous references to `f`; the first is definitely more useful for debugging purposes). In other words, this interactive ZINC system would be a cross-breed between conventional, Lisp-like toplevels and modern source-level debuggers. Close integration with a powerful text editor such as Emacs [56] would be nice, too.

Chapter 3

The abstract machine

This chapter introduces the execution method used by ZINC. As usual, it will be specified using an abstract machine.

Abstract machines for strict functional languages abound in the literature: Landin's seminal SECD [37], Cardelli's FAM [11], Curien's CAM [20], ... Yet it was necessary to develop a new machine for ZINC. Indeed, one of the design requirements is that multiple application to k arguments should be efficient, almost as efficient as applying a k-ary function. As we saw in section 2.2, this is not the case with the existing abstract machines mentioned above, since they all build $k-1$ intermediate closures to evaluate this multiple application.

We may fear that this requirement is totally unrealistic; maybe there is a deep result stating that this apparently bad behavior is unavoidable, and that these closures are the price to pay for the additional flexibility of curried functions (e.g. partial evaluation). However, it seems that this requirement is achieved in lazy graph reducers such as the G-machine [32, 50]. An evidence is their use of supercombinators to speed up reduction: supercombinators are special cases of functions with several arguments, yet they are curried, and applying a supercombinator to several arguments seems much faster than applying it argument by argument, otherwise lambda-lifting (the transformation of a program into a set of supercombinators) would not be very interesting.

To make this intuition clearer, and start the shift from lazy graph reducers to strict environment machines, I shall first present an environment machine performing standard reduction, and enjoying some of the interesting properties of graph reducers: Krivine's machine.

3.1 Krivine's machine

This abstract machine, due to J.-L. Krivine [36, 21], performs reduction to weak head normal form, following the standard (leftmost-outermost) strategy. However, λ-terms are represented by closures, hence substitutions are not performed on the fly, but delayed till variables are reduced.

3.1.1 Presentation

This machine has but three instructions: Access, Push, and Grab. A term in de Bruijn's notation [10] is compiled as follows:

$$[\![n]\!] \quad = \quad \mathsf{Access}(n)$$

Terms	$M ::= n \mid (M\ N) \mid \lambda M \mid M[s]$
Substitutions	$s ::= \mathit{Id} \mid \mathit{Shift} \mid M \cdot s \mid s \circ t \mid \Uparrow(s)$

(Beta)	$(\lambda M)N$	$=$	$M[N \cdot \mathit{Id}]$
(App)	$(M\ N)[s]$	$=$	$M[s]\ N[s]$
(Lambda)	$(\lambda M)[s]$	$=$	$\lambda(M[\Uparrow(s)])$
(Closure)	$(M[s])[t]$	$=$	$M[s \circ t]$
(VarShift1)	$\mathtt{n}[\mathit{Shift}]$	$=$	$\mathtt{n}+\mathtt{1}$
(VarShift2)	$\mathtt{n}[\mathit{Shift} \circ s]$	$=$	$\mathtt{n}+\mathtt{1}[s]$
(FVar)	$\mathtt{1}[M \cdot s]$	$=$	M
(FVarLift1)	$\mathtt{1}[\Uparrow(s)]$	$=$	$\mathtt{1}$
(FVarLift2)	$\mathtt{1}[\Uparrow(s) \circ t]$	$=$	$\mathtt{1}[t]$
(RVar)	$\mathtt{n}+\mathtt{1}[M \cdot s]$	$=$	$\mathtt{n}[s]$
(RVarLift1)	$\mathtt{n}+\mathtt{1}[\Uparrow(s)]$	$=$	$\mathtt{n}[s \circ \mathit{Shift}]$
(RVarLift2)	$\mathtt{n}+\mathtt{1}[\Uparrow(s) \circ t]$	$=$	$\mathtt{n}[s \circ (\mathit{Shift} \circ t)]$
(AssEnv)	$(s \circ t) \circ u$	$=$	$s \circ (t \circ u)$
(MapEnv)	$(M \cdot s) \circ t$	$=$	$M[t] \cdot (s \circ t)$
(Shift)	$\mathit{Shift} \circ (M \cdot s)$	$=$	s
(ShiftLift1)	$\mathit{Shift} \circ \Uparrow(s)$	$=$	$s \circ \mathit{Shift}$
(ShiftLift2)	$\mathit{Shift} \circ \Uparrow(s \circ t)$	$=$	$s \circ (\mathit{Shift} \circ t)$
(Lift1)	$\Uparrow(s) \circ \Uparrow(t)$	$=$	$\Uparrow(s \circ t)$
(Lift2)	$\Uparrow(s) \circ (\Uparrow(t) \circ u)$	$=$	$\Uparrow(s \circ t) \circ u$
(LiftEnv)	$\Uparrow(s) \circ (M \cdot t)$	$=$	$M \cdot (s \circ t)$
(IdL)	$\mathit{Id} \circ s$	$=$	s
(IdR)	$s \circ \mathit{Id}$	$=$	s
(LiftId)	$\Uparrow(\mathit{Id})$	$=$	Id
(IdEnv)	$M[\mathit{Id}]$	$=$	M

Table 3.1: The calculus λenv

$$\begin{aligned}
[\![(M\ N)]\!] &= \mathsf{Push}([\![N]\!]);[\![M]\!] \\
[\![\lambda M]\!] &= \mathsf{Grab};[\![M]\!]
\end{aligned}$$

The machine is equipped with a code pointer, a register holding the current environment (a list of closures, that is, pairs of code pointers and environments), and a stack of closures. The transition function is as follows:

Code	Env.	Stack	Code	Env.	Stack
$\mathsf{Access}(0);c$	$(c_0,e_0)\cdot e$	s	c_0	e_0	s
$\mathsf{Access}(n+1);c$	$(c_0,e_0)\cdot e$	s	$\mathsf{Access}(n);c$	e	s
$\mathsf{Push}(c');c$	e	s	c	e	$(c',e)\cdot s$
$\mathsf{Grab};c$	e	$(c_0,e_0)\cdot s$	c	$(c_0,e_0)\cdot e$	s

At all times the stack represents the *spine* of the term being reduced (that is, the term whose code is in the code pointer). The Push instruction performs one step of unrolling, and Grab corresponds to one step of β-reduction, that is it records the substitution in the environment.

3.1.2 Correctness

For a more formal correctness argument, I shall use the λenv-calculus of Hardin and Lévy [27] (see also [2] for a similar calculus, with less pleasant properties, however). This variant of the λ-calculus manipulates explicitly substitutions, and therefore provides a good framework for reasoning about implementations where substitutions are delayed. Table 3.1 recalls the syntax of terms, along with the equational theory defining the equivalence of two terms. Notice that usual λ-terms are also valid terms of this calculus; indeed, two λ-terms are β-convertible if and only if the corresponding λenv terms are equivalent.

A state of Krivine's machine encodes a $\lambda\sigma$ term in the following way. To a sequence of instructions c, and an environment e, we associate a term $\overline{c}$ and a substitution $\overline{e}$:

$$\begin{aligned}
\overline{\mathsf{Access}(n);c} &= \mathtt{n} \\
\overline{\mathsf{Push}(c_2);c_1} &= (\overline{c_1}\ \overline{c_2}) \\
\overline{\mathsf{Grab};c} &= \lambda\overline{c}
\end{aligned}$$

$$\overline{(c_0,e_0)\cdots(c_n,e_n)} = \overline{c_0}[\overline{e_0}]\cdots\overline{c_n}[\overline{e_n}]\cdot Id$$

Then, to the state (c,e,s) of Krivine's machine, we associate a term $\overline{(c,e,s)}$ as follows:

$$\overline{(c,e,(c_0,e_0)\cdots,(c_n,e_n))} = (\overline{c}[\overline{e}]\ \overline{c_0}[\overline{e_0}]\ \ldots\ \overline{c_n}[\overline{e_n}])$$

Proposition 1 (Correctness of Krivine's machine) *If $(c_1,e_1,s_1)\to(c_2,e_2,s_2)$ is a transition of Krivine's machine, then $\overline{(c_1,e_1,s_1)}=\overline{(c_2,e_2,s_2)}$.*

Proof: It is enough to prove it for all four transitions, that is to check the following equalities:

$$
\begin{array}{rcl}
\mathtt{1}[\overline{c_0}[\overline{e_0}] \cdot \overline{e}] & = & \overline{c_0}[\overline{e_0}] \\
(\mathtt{n}+\mathtt{1})[\overline{c_0}[\overline{e_0}] \cdot \overline{e}] & = & \mathtt{n}[\overline{e}] \\
(\overline{c_1}\ \overline{c_2})[\overline{e}] & = & (\overline{c_1}[\overline{e}]\ \overline{c_2}[\overline{e}]) \\
((\lambda\overline{c})[\overline{e}]\ \overline{c_0}[\overline{e_0}]) & = & \overline{c}[\overline{c_0}[\overline{e_0}] \cdot \overline{e}]
\end{array}
$$

The first three equalities are exactly the *FVar*, *RVar* and *App* axioms. The last one is more complex. Writing $M = \overline{c}$, $s = \overline{e}$ and $N = \overline{c_0}[\overline{e_0}]$, we have:

$$
\begin{array}{rcll}
((\lambda M)[s]\ N) & = & ((\lambda(M[\Uparrow(s)]))\ N) & (Lambda) \\
& = & M[\Uparrow(s)][N \cdot Id] & (Beta) \\
& = & M[\Uparrow(s) \circ (N \cdot Id)] & (Closure) \\
& = & M[N \cdot (s \circ Id)] & (LiftEnv) \\
& = & M[N \cdot s] & (IdR)
\end{array}
$$

which is the expected result. □

3.1.3 Multiple applications

Let us consider the evaluation of a multiple application $(M\ N_1 \ldots N_k)$. This term is compiled into $\mathsf{Push}([\![N_k]\!]); \ldots; \mathsf{Push}([\![N_1]\!]); [\![M]\!]$. At runtime, the closures representing $N_k, \ldots, N_1$ are pushed on the stack, then M is evaluated. Each time a Grab instruction is encountered, that is each time an abstraction is reduced, one of the arguments is popped and added to the environment, then evaluation proceeds with the remaining code, that is the body of the abstraction. The crucial point is that no closures are ever built to represent the intermediate applications $(M\ N_1)$, $(M\ N_1\ N_2)$, $\ldots$, $(M\ N_1 \ldots N_k)$. The sole closures built are those needed to "freeze" the arguments $N_1, \ldots, N_k$, and these are unavoidable with a lazy strategy.

Contrast this behavior to the one of the lazy CAM [20, 44], for instance: the lazy CAM reduces M to weak head normal form first, then applies it to the closure of N_1. The result, the closure representing $(M\ N_1)$ is then applied to the closure of N_2, leading to the closure representing $(M\ N_1\ N_2)$, and so on. Thus, the lazy CAM builds $2k-1$ closures to evaluate $(M\ N_1 \ldots N_k)$, and Krivine's machine builds but k closures. By comparison, the strict CAM as well as the FAM don't build the closures of $N_1, \ldots, N_k$, since these are reduced on the fly to weak head normal form, but they do build closures for the intermediate applications $(M\ N_1), \ldots, (M\ N_1 \ldots N_k)$, totalizing $k-1$ closures. To summarize:

	lazy	strict
without spine	Lazy CAM $2k-1$ closures	Strict CAM $k-1$ closures
with spine	Krivine's machine k closures	? 0 closures

At that point, one may hope to replace the question mark ("The Functional Abstract Machine That Hardly Ever Conses") by some machine analogous to Krivine's, but performing strict evaluation instead of lazy evaluation.

3.2 Krivine's machine with marks on the stack

To perform strict evaluation with some variant of Krivine's machine, we need first to be able to reduce some subterms of a given term to weak head normal form. The problem with Krivine's machine is that it does not stop until the stack is empty. What we need is a way to stop reduction even if there are arguments available on the stack. To this end, let's put a mark on some of the closures awaiting in the stack; this mark says "don't put me in the environment, stop reducing, and resume another reduction". This idea of marks comes from Fairbairn and Wray's Three-Instruction Machine [24], though it uses marks for different purposes (to share reductions); Crégut [21] then applied it to Krivine's machine, again for sharing purposes.

3.2.1 Presentation

The modified Krivine's machine has a fourth instruction, $\mathsf{Reduce}(c)$, to force reduction of c to weak head normal form, and a different semantics for Grab. In the following, marked closures are written $\langle c, e\rangle$ instead of (c, e).

Code	Env.	Stack	Code	Env.	Stack
$\mathsf{Access}(0); c$	$(c_0, e_0) \cdot e$	s	c_0	e_0	s
$\mathsf{Access}(n+1); c$	$(c_0, e_0) \cdot e$	s	$\mathsf{Access}(n); c$	e	s
$\mathsf{Push}(c'); c$	e	s	c	e	$(c', e) \cdot s$
$\mathsf{Grab}; c$	e	$(c_0, e_0) \cdot s$	c	$(c_0, e_0) \cdot e$	s
$\mathsf{Grab}; c$	e	$\langle c_0, e_0\rangle \cdot s$	c_0	e_0	$(\mathsf{Grab}; c, e) \cdot s$
$\mathsf{Reduce}(c'); c$	e	s	c'	e	$\langle c, e\rangle \cdot s$

3.2.2 Correctness

To prove correctness, we proceed as previously, by associating a term to code sequences. An additional case is needed:

$$\overline{\mathsf{Reduce}(c_2); c_1} = (\overline{c_1}\ \overline{c_2})$$

The encoding of machine states is generalized as follows:

$$\begin{array}{rcl} \overline{(c, e, \mathit{empty})} & = & \overline{c}[\overline{e}] \\ \overline{(c, e, s \cdot (c_0, e_0))} & = & (\overline{(c, e, s)}\ \overline{c_0}[\overline{e_0}]) \\ \overline{(c, e, s \cdot \langle c_0, e_0\rangle)} & = & (\overline{c_0}[\overline{e_0}]\ \overline{(c, e, s)}) \end{array}$$

Proposition 2 (Correctness of Krivine's machine with marks) *If* $(c_1, e_1, s_1) \rightarrow (c_2, e_2, s_2)$ *is a transition of Krivine's machine with marks, then* $\overline{(c_1, e_1, s_1)} = \overline{(c_2, e_2, s_2)}$*.*

Proof: The first four transitions are exactly those of Krivine's machine, so it suffices to check the last two transitions. For the penultimate one, the result comes from the following syntactic identity:

$$\overline{(c, e, \langle c_0, e_0\rangle \cdot s)} \equiv \overline{(c_0, e_0, (c, e) \cdot s)}$$

which is easy to prove by induction on s. For the last transition, we have

$$\overline{(\mathsf{Reduce}(c_2); c_1, e, \mathit{empty})} \equiv (\overline{c_1}\ \overline{c_2})[\overline{e}] = (\overline{c_1}[\overline{e}]\ \overline{c_2}[\overline{e}]) \equiv \overline{(c_1, e, \langle c_2, e\rangle \cdot \mathit{empty})}$$

and this result holds for non-empty stacks as well, by trivial induction. □

3.2.3 Compiling call-by-value

A strict application $(M\ N)_s$, that is the application of M to the weak head normal form of N, can be compiled as follows:

$$[\![(M\ N)_s]\!] = \mathsf{Reduce}([\![N]\!]); [\![M]\!]$$

The reader can check that a strict multiple application $(M\ N_1 \dots N_k)_s$, that is

$$(\dots((M\ N_1)_s\ N_2)_s \dots N_k)_s,$$

evaluates without building the closures corresponding to $(M\ N_1)_s, \dots, (M\ N_1 \dots N_k)_s$, as the original Krivine's machine, nor the closures of $N_1, \dots, N_k$, since this is call-by-value. Therefore, strict multiple applications evaluate without allocating any regular closures. Of course, it allocates k marked closures on the stack. But these closures do not need to be heap-allocated: it is easy to check that they cannot be put in an environment. Actually, it suffices to push both components of these marked closures on the stack. Therefore, strict multiple application does not perform any allocation in the heap. This is the expected result.

3.3 The ZINC machine

The ZINC not-so-abstract machine can be seen as a Krivine's machine with marks specialized to call-by-value only, and extended to handle constants as well. To tell the truth, it was designed well before Pierre-Louis Curien suggested me this viewpoint. My original approach was more down-to-earth: designing a stack-based calling conventions where functions may not consume all their arguments (as in C), but then their result must be applied to the remaining arguments. This explains a few cosmetic differences between the two machines. Yet it is easy to see the underlying Krivine's machine at work, and this is enough to convince oneself of the correctness of the ZAM.

As Krivine's machine, the ZAM is equipped with a code pointer and a register holding the current environment. Environments are lists of values, that is, either closures representing functions, or constants (integers, booleans, ...). An accumulator has been added to hold intermediate results. It is unnecessary in Krivine's machine, since it handles only closures, hence the code pointer and the environment register are enough to represent results. When we add constants, an explicit accumulator becomes necessary.

The stack of Krivine's machine has been split into two stacks. One stack, the argument stack, holds arguments to function calls, that is sequences of values, separated by marks. Marks are no more specially tagged closures, but simply a distinguished value written ϵ. The other stack, the return stack, holds (unallocated) closures, that is pairs of a code pointer and an environment. This design helps reducing stack moves, and allows further refinements in the way environments are represented.

In the following, I shall define two compilation schemes: one, written $\mathcal{T}[\![E]\!]$, is only valid for expressions E in tail-call position, that is expressions whose value is the value of the function body being evaluated; the other, written $\mathcal{C}[\![E]\!]$, is always valid, but usually less efficient. Then, I give the transitions of the ZAM corresponding to the generated instructions. The first line is the state before the transition, the second one is the state after the transition.

3.3.1 Accessing local variables

The compilation scheme for the local variable of index n is:

$$\mathcal{T}[\![n]\!] = \mathcal{C}[\![n]\!] = \mathsf{Access}(n)$$

The Access instruction has the following semantics:

Code	Accu	Env.	Arg. stack	Return stack
$\mathsf{Access}(n); c$	a	$e = v_0 \dots v_n \dots$	s	r
c	v_n	e	s	r

3.3.2 Application

$$\begin{aligned}
\mathcal{T}[\![(M\ N_1 \dots N_k)]\!] &= \mathcal{C}[\![N_k]\!]; \mathsf{Push}; \dots; \mathcal{C}[\![N_1]\!]; \mathsf{Push}; \mathcal{C}[\![M]\!]; \mathsf{Appterm} \\
\mathcal{C}[\![(M\ N_1 \dots N_k)]\!] &= \mathsf{Pushmark}; \mathcal{C}[\![N_k]\!]; \mathsf{Push}; \dots; \mathcal{C}[\![N_1]\!]; \mathsf{Push}; \mathcal{C}[\![M]\!]; \mathsf{Apply}
\end{aligned}$$

The applications we consider are naturally multiple. The compiled code remains correct if k is not maximal, for instance if we consider $(M\ N\ P)$ as $(M\ Q)$ where $Q = (N\ P)$, but less efficient, since unnecessary closures will be built.

Tail applications are treated as in Krivine's machine, since there is no need to allocate a new argument stack by pushing a mark. The sole difference is that the Appterm instruction takes care of consing the first argument with the environment of the closure; this way, we do not have to put a Grab instruction at the beginning of each function. For other applications, we must push a mark on the argument stack to separate the "new" arguments and force reduction to weak head normal form.

Code	Accu	Env.	Arg. stack	Return stack
$\mathsf{Appterm}; c_0$	$a = (c_1, e_1)$	e_0	$v.s$	r
c_1	a	$v.e_1$	s	r
$\mathsf{Apply}; c_0$	$a = (c_1, e_1)$	e_0	$v.s$	r
c_1	a	$v.e_1$	s	$(c_0, e_0).r$
$\mathsf{Push}; c_0$	a	e	s	r
c_0	a	e	$a.s$	r
$\mathsf{Pushmark}; c_0$	a	e	s	r
c_0	a	e	$\varepsilon.s$	r

3.3.3 Abstractions

$$\begin{aligned}
\mathcal{T}[\![\lambda E]\!] &= \mathsf{Grab}; \mathcal{T}[\![E]\!] \\
\mathcal{C}[\![\lambda E]\!] &= \mathsf{Cur}(\mathcal{T}[\![E]\!]; \mathsf{Return})
\end{aligned}$$

In tail-call position, the Grab instruction simply pops one argument from the argument stack, and puts it in front of the environment. If all arguments have already been consumed, that is if there is a mark at the top of the stack, it builds the closure of the current code with the current environment and returns it to the caller, while popping the mark.

Otherwise, we could push a mark, to allocate a new argument stack, and then do the same thing. Of course, Grab would always fail and return immediately the desired closure. To save pushing a mark, and then immediately test it, we use the cheaper Cur instruction, in this case.

The Return instruction that terminates the body of a function does not simply jump back to the caller. It is actually the symmetric of Grab: it has to check if the argument stack is "empty" (i.e. if the top of stack is a mark). If this is the case, it destroys the mark and returns to the caller. But otherwise, it applies the result of the function (necessarily a closure, if the original program is well-typed) to the remaining arguments. This situation is the converse of partial application: a single function is given more arguments than it can use. This is the case of the identity function in the following example:

$$((\lambda x.x)\ (\lambda y.y+1)\ 4)$$

Code	Accu	Env.	Arg. stack	Return stack
$\mathsf{Cur}(c_1); c_0$	a	e	s	r
c_0	(c_1, e)	e	s	r
$\mathsf{Grab}; c_0$	a	e_0	$\varepsilon.s$	$(c_1, e_1).r$
c_1	(c_0, e_0)	e_1	s	r
$\mathsf{Grab}; c$	a	e	$v.s$	r
c	a	$v.e$	s	r
$\mathsf{Return}; c_0$	a	e_0	$\varepsilon.s$	$(c_1, e_1).r$
c_1	a	e_1	s	r
$\mathsf{Return}; c_0$	$a = (c_1, e_1)$	e_0	$v.s$	r
c_1	a	$v.e_1$	s	r

3.3.4 Local declarations

$$\begin{aligned}
\mathcal{T}[\![\texttt{let}\ 1 = N\ \texttt{in}\ M]\!] &= \mathcal{C}[\![N]\!]; \mathsf{Let}; \mathcal{T}[\![M]\!] \\
\mathcal{C}[\![\texttt{let}\ 1 = N\ \texttt{in}\ M]\!] &= \mathcal{C}[\![N]\!]; \mathsf{Let}; \mathcal{C}[\![M]\!]; \mathsf{Endlet} \\
\mathcal{T}[\![\texttt{let rec}\ 1 = N\ \texttt{in}\ M]\!] &= \mathsf{Dummy}; \mathcal{C}[\![N]\!]; \mathsf{Update}; \mathcal{T}[\![M]\!] \\
\mathcal{C}[\![\texttt{let rec}\ 1 = N\ \texttt{in}\ M]\!] &= \mathsf{Dummy}; \mathcal{C}[\![N]\!]; \mathsf{Update}; \mathcal{C}[\![M]\!]; \mathsf{Endlet}
\end{aligned}$$

The special case of `let`, that is $((\lambda x.M)\ N)$, is so common that it deserves a faster and simpler compilation scheme than actually applying an abstraction. It is enough to evaluate N and add its value to the environment, using the Let instruction, then to evaluate M in this modified environment; then, the Endlet instruction restores the original environment, if needed.

For recursive definitions, I use the same trick suggested for the CAM [20]: first, a dummy value is added to the environment (instruction Dummy), and N is evaluated in this modified environment; the dummy value is then physically updated with the actual value of N (instruction Update). This may fail to reach a fixpoint, since the physical update may be impossible (in case of an unboxed value, an integer for instance). However, it works fine for the most commonly used case: when M is an abstraction $\lambda.P$. It is true that in this case, more efficient compilation schemes could be used: for instance, it is possible to loop directly in the code, rather than building a circular environment, as noticed by Mauny and Suárez [46] in the setting of the CAM. Their solutions can be easily applied to the ZAM as well.

Code	Accu	Env.	Arg. stack	Return stack
$\mathsf{Let}; c$	a	e	s	r
c	a	$a.e$	s	r
$\mathsf{Endlet}; c$	a	$v.e$	s	r
c	a	e	s	r
$\mathsf{Dummy}; c$	a	e	s	r
c	a	$?.e$	s	r
$\mathsf{Update}; c$	a	$e = v.e_1$	s	r
c	a	$e[v \leftarrow a]$	s	r

3.3.5 Primitives

$$\mathcal{T}[\![p(M_1, \ldots, M_k)]\!] = \mathcal{C}[\![p(M_1, \ldots, M_k)]\!] = \mathcal{C}[\![M_k]\!]; \mathsf{Push}; \ldots \mathcal{C}[\![M_2]\!]; \mathsf{Push}; \mathcal{C}[\![M_1]\!]; \mathsf{Prim}(p)$$

We write $\mathsf{Prim}(p)$ for the instruction associated with the primitive operation p (e.g. $+$, $=$, `car`) This instruction takes its first argument in the accumulator, the remaining arguments in the argument stack, and puts its result in the accumulator.

Code	Accu	Env.	Arg. stack	Return stack
$\mathsf{Prim}(p); c$	a	e	$v_2 \ldots v_k.s$	r
c	$p(a, v_2, \ldots, v_k)$	e	s	r

3.3.6 Control structures

Conditionals, loops, ..., are compiled in a very classical way, but I shall not describe them formally, since to describe code sharing, we need to introduce labels in the code; this will be done in the description of the implementation.

3.4 Another representation for the environment

The discussion above assumes that environments are always allocated in the heap, so that they can be put in a closure at any time. This is also the case in most environment machines (SECD, CAM, FAM), though they differ on the way environments are represented (linked lists or vectors). Given the high frequency of closure building, little else can be done. For instance, a curried function of several arguments adds a value to the environment and immediately builds a closure for each argument it receives. However, the ZINC machine was specially designed to build less closures. This opens the way for less costly (in terms of heap allocation) representations of environments.

The idea is as follows: when we don't have to build any closures, the current environment does not have to survive the evaluation of the current function body. Therefore we can store it, or part of it, in some volatile location (stack or registers) that will be automatically reclaimed when the current function returns. We can go even further: assuming few closures are built, a sensible policy is to systematically put values being added to the environment in one of these volatile locations, and to copy them back to persistent storage (i.e. in the heap) when a closure is built. (In other terms, we add a write-back cache on the environment). In this approach, the environment $0 \leftarrow a_0, \ldots, n \leftarrow a_n$ is represented by a persistent part $a_k, \ldots, a_n$, which is the environment part of the closure most recently applied or built, and a volatile part $a_0, \ldots, a_{k-1}$, which holds values added to the environment since then. Hence, a given environment has several different representations; this makes the access operation slightly more complex.

More formally, let *Env* be an environment structure, that is an abstract type equipped with a constant *empty* : *Env* and the following operations:

- $access_n : Env \to Value$, to return the value associated to n;
- $add : Value, Env \to Env$, to bind the given value to 0 and shift all other bindings up by one;
- $remove : Env \to Env$, to shift all bindings down by one.

Then we can define a new environment structure Env' as follows: objects of that type are pairs $\langle S, E\rangle$ of a stack S of values and of an environment $E : Env$; operations on Env' are defined as:

$$\begin{array}{rcl} empty' & = & \langle \emptyset, empty\rangle \\ access'_n\langle S, E\rangle & = & S[n] \text{ if } n < \|S\| \\ access'_n\langle S, E\rangle & = & access_{n-\|S\|}E \text{ if } n \geq \|S\| \\ add'(v, \langle S, E\rangle) & = & \langle v.S, E\rangle \\ remove'\langle v.S, E\rangle & = & \langle S, E\rangle \\ remove'\langle \emptyset, E\rangle & = & \langle \emptyset, remove\ E\rangle \end{array}$$

We write $\|S\|$ for the size of the stack S, and $S[n]$ for its n^{th} element, the top of the stack being $S[0]$. Alternatively, if the *remove* operation is too costly, we can define $remove'$ as:

$$remove'\langle \emptyset, E\rangle = \langle access_0 E \ldots access_n E, empty\rangle$$

To produce an equivalent environment, but without volatile components, hence suitable for a closure, we have an additional operation called $perpetuate : Env' \to Env'$, consisting in:

$$perpetuate\langle a_k \ldots a_1.a_0.\emptyset, E\rangle = \langle \emptyset, add(a_0, add(a_1, \ldots add(a_k, E)))\rangle$$

To convince oneself that adding a cache this way does not compromise correctness, it is enough to notice that environments with cache can be mapped to simple environments as follows:

$$\overline{\langle a_0.a_1 \ldots a_k.\emptyset, E\rangle} = add(a_0, add(a_1, \ldots add(a_k, E)))$$

and this maps the transition of some machine with a cache on transitions of the same machine without cache.

Such a cache can be added to any environment structure: vector, linear linked list, linked list of vectors, ... However, adding a cache makes the *add* and *remove* operations much less frequent; in addition, *remove* can be eliminated entirely, and when we extend the persistent environment, we do not add one value at a time any more, but rather several values at once. The vector structure really shines here, since it allows accesses in constant time, and thanks to the cache we can afford the price of full copying when a *perpetuate* operation is needed, that is, when building a closure.

Each active function needs its own cache, and it must be preserved during function call. However, caches follow a stack discipline (at any time, only the most recently allocated cache is active), so we can allocate them in a stack. The return stack of the ZAM is perfectly suited to this purpose. In other words, the return stack still holds unallocated closures, but these closures now include the volatile part of environments as well. More precisely, the return stack is now a sequence of blocks holding:

Code	Accu.	Env.	Size	Arg. stack	Return stack
$\mathsf{Access}(n); c$	a	e	$m > n$	s	$r = v_0 \dots v_n \dots v_{m-1}.r_0$
c	v_n	e	m	s	r
$\mathsf{Access}(n); c$	a	$e = (v_m \dots v_n \dots)$	$m \leq n$	s	$r = v_0 \dots v_{m-1}.r_0$
c	v_n	e	m	s	r
$\mathsf{Appterm}; c_0$	$a = (c_1, e_1)$	e_0	m	$v.s$	r
c_1	a	e_1	1	s	$v.r$
$\mathsf{Apply}; c_0$	$a = (c_1, e_1)$	e_0	m_0	$v.s$	r
c_1	a	e_1	1	s	$v.(c_0, e_0, m_0).r$
$\mathsf{Cur}(c_1); c_0$	a	$(v_m \dots v_n)$	m	s	$v_0 \dots v_{m-1}.r_0$
c_0	$(c_1, (v_0 \dots v_n))$	$(v_0 \dots v_n)$	0	s	r_0
$\mathsf{Grab}; c_0$	a	$(v_m \dots v_n)$	m	$\varepsilon.s$	$v_0 \dots v_{m-1}.(c_1, e_1, m_1).r$
c_1	$(c_0, (v_0 \dots v_n))$	e_1	m_1	s	r
$\mathsf{Grab}; c$	a	e	m	$v.s$	r
c	a	e	$m + 1$	s	$v.r$
$\mathsf{Return}; c_0$	a	e_0	m	$\varepsilon.s$	$v_0 \dots v_{m-1}.(c_1, e_1, m_1).r$
c_1	a	e_1	m_1	s	r
$\mathsf{Return}; c_0$	$a = (c_1, e_1)$	e_0	m	$v.s$	$v_0 \dots v_{m-1}.r_0$
c_1	a	e_1	1	s	$v.r_0$
$\mathsf{Let}; c$	a	e	m	s	r
c	a	e	$m + 1$	s	$a.r$
$\mathsf{Endlet}; c$	a	e	$m > 0$	s	$v_0.r$
c	a	e	$m - 1$	s	r
$\mathsf{Endlet}; c$	a	$(v_0.v_1 \dots v_n)$	0	s	r_0
c	a	$()$	$n - 1$	s	$v_1 \dots v_n.r_0$

Table 3.2: The ZINC abstract machine with cache

- a code pointer
- a (pointer to a) vector representing the persistent part of an environment
- an integer n, the size of the volatile part
- n values representing the volatile part itself.

In addition, the topmost block is not complete, it contains only the volatile part of the current environment, since special registers hold the current code pointer, persistent environment, and size of volatile environment.

We are now ready to give the semantics of the basic instructions of the ZAM with cache (figure 3.2). To make deciphering easier, we write r for a return stack with an incomplete block at the top, as described above, and r_0 for a sequence of complete blocks.

3.5 Conclusion

Trying to reduce the number of closures and environment blocks allocated in the heap has lead us quite naturally to a fairly complex abstract machine, much more complex than the SECD, for instance, and one may fear that the improvement in performances over classical abstract machines is not worth the additional complexity. At that point, it is reassuring to have a look at the results of a few preliminary benchmarks, given in appendix A. They demonstrate that the special mechanisms of the ZAM often lead to dramatic savings of heap space (several orders of magnitude) compared with the SECD or the CAM, while raw execution times remain very close (and these times do not take into account time spent for garbage collection).

Another source of concern is the sheer number of tests performed during execution, either to see whether the argument stack is empty, or when accessing the environment, to know whether the desired value is bound in the persistent part or in the volatile part. It is true that so many tests would lead to very inefficient code, if we were to expand it into native code of a traditional processor. But this is much less serious in the case of bytecode interpretation, since the time spent performing these tests is negligible compared to the interpretative overhead; and these tests helps reducing the number of instructions executed, which is crucial in the case of bytecode interpretation.

Chapter 4

Data representation

To complete the execution model defined in the previous chapter, it remains to find a representation for ML values, and add instructions to manipulate these representations.

4.1 Data structures

ML data types can be split in three classes:

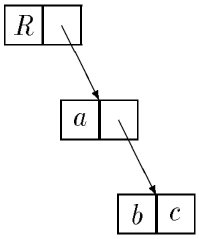
- atomic, predefined types: "small" integers, floating-point numbers, characters, strings, and possibly arbitrary-precision integers.
- predefined type constructors: functions (closures), vectors, dynamics.
- all other type constructors can be defined in ML, through a very general mechanism called "concrete types". For instance:

```
type envir = Empty
           | Regular of int * string list * envir
           | Special of int * string list * envir
           | Weird of string list * envir;;
```

The remaining part of this section investigates more precisely the meaning of such a declaration in terms of more elementary type constructions such as sums and products.

4.1.1 Sums and pairs

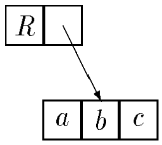
The original ML has only one kind of type constructs, the *sums*. In addition, a built-in pair operator is provided to allow grouping of values. Among the constructors (that is, the summands) of a sum type, classic ML distinguishes between nullary constructors (constant constructors), and unary constructors (functional constructors). This is mostly a question of syntax: a constant constructor `C` can always be viewed as a functional constructor `C of unit`, where `unit` is a built-in type with only one value `()` of that type. In this model, the value $e = \texttt{Regular}(a, b, c)$ is parsed as $\texttt{Regular}(a, (b, c))$ and therefore represented as follows (R stands for a tag associated to the constructor `Regular`):



This representation is very inefficient, both in space and in time to access a, b, and c. Conversely, it allows a very fine-grained destructuring of e, since the sequence of its arguments (a, b, c) and even the suffix (b, c) are legal ML values. For instance, the following phrase is perfectly legal:

```
match e with Regular x     -> Special x
           | Special (x,y) -> Weird y
```

To be more efficient, Lazy ML of Göteborg [7] provides not only a built-in pair type, but also a triple type, a quadruple type, ... Then, `int * string list * envir` is parsed as a triple type, hence `Regular`(a, b, c) has a more efficient representation, that is:



But now (b, c) is no more a valid ML value, and `match e with Special(x,y) -> Weird y` is ill-typed ((x, y) is a pair, and the argument of `Special` is expected to be a triple). It must be replaced by `match e with Special(x,y,z) -> Weird (y,z)`. The price to pay for a more efficient representation is a (very slight) loss of expressiveness. However, `match e with Regular x -> Special x` remains legal.

4.1.2 Sums and products

In both solutions, the components of a tuple are still identified by their position, and this is error-prone. The next step is to give names to the components of a tuple. To this end, Standard ML and CAML 2.6 provide another kind of concrete types, the *records*. These are ordered, named products. For instance:

```
type envir = Empty | Regular of envir_triple
           | Special of envir_triple | Weird of envir_pair
and envir_triple = { size: int; names: string list; rest: envir}
and envir_pair   = { names: string list; rest: envir }
```

Records usually make programs clearer, if less concise, and they eliminate the need for built-in tuple types. However, the previous example illustrates their most unpleasant feature: a given label may belong to several record types. This overloading of labels seems to be necessary, given the natural tendency of programmers to reuse such evocative names as `size`, `rest` and `name`. But it does not interact very well with type inference. For instance, `function {size = x; _} -> x+1` has no principal type if the label `size` belongs to several record types!

4.1.3 Sums of products

ZINC retains sums as the sole type formation operator, but generalizes the nullary and unary constructors of classic ML to *n-ary constructors*. In other words, this operator performs the sum of several products of types. Also, notice the analogy with sorted free algebras.

Such sums of products lend themselves to a compact and "flat" representation. The term `Regular`(a, b, c) is represented by:

R	a	b	c

Such an efficient representation is made possible by the fact that the sequence of the arguments of a constructor is no more a valid ML object. As a consequence, both cases of

```
match e with Regular x     -> Special x
           | Special (x,y) -> Weird y
```

are now ill-typed: `Regular` and `Special` must have exactly three arguments, and `Weird` two arguments. This phrase has to be rewritten as:

```
match e with Regular(x,y,z) -> Special(x,y,z)
           | Special (x,y,z) -> Weird(y,z)
```

This phrase may seem less concise and less elegant than the original one. But one may argue that it is definitely easier to read, since it makes clear that `Regular` takes three arguments, not one that happens to be a triple. Elegant but confusing code might be fun to write, but should be avoided.

There is no need for built-in tuple types, since they can easily be defined (with some special syntax) as:

```
type 'a & 'b = (* the empty string *) of 'a * 'b;;
```

That way, we can also represent old-fashioned concrete types, with unary constructors and trees of pairs as arguments.

Then, for each argument of a constructor, we can give additional information on how it should be computed and updated, as in CAML 2.6 [62, p. 71]. For instance, we can declare that an argument may be physically modified using the keyword `mutable`. Updating is performed by the construction *id* `<-` *expr*, where *id* is an identifier bound by pattern matching. As an example, physical concatenation of lists is done as follows:

```
type 'a mlist = MNil | MCons of 'a * mutable 'a mlist;;
let rec physical_append =
fun MNil  x  ->  x
  | (Mcons(elt, (Mnil as tail)) as whole)  x  ->  tail <- x; whole
  | (Mcons(elt, tail) as whole)  x  ->  physical_append tail x; whole;;
```

Finally, it is possible to add labels naming the arguments of a constructor, and use these labels for pattern matching:

```
type envir = Empty
           | Regular of { size: int; names: string list; rest: envir}
           | ...;;
let firstname = function Regular{ names = n::_ ; _ } -> n | ...;;
```

This time, there is no problem with using several times the same label, since the constructor suffices to identify the concrete type to which the value belongs.

4.1.4 Records with inclusion

Another approach to the problem of reusing label names is to consider that a record r' with more fields than a record r (that is, r' has all the labels of r, and then some) is really a special case of r, that is anywhere r can be used, it should be possible to use r' instead. In other terms, the type of r' is a *subtype* of the one of r. This approach elegantly solves the problem of giving a principal type to, for instance, our previous example `function {size = x; _} -> x+1`. More importantly, subtyping integrates inheritance with static typing, and therefore opens the door to the realm of object-oriented programming. A more detailed introduction, and very convincing examples can be found in Cardelli and Wegner [15]. Such records with inclusion can be integrated to the ML language quite nicely, and still allow type inference, as in the systems proposed by Jategaonkar and Mitchell [31], by Ohori and Buneman [48], and by Rémy [53, 54].

Records with inclusions are not implemented in ZINC yet, but it should not be too difficult, using the typechecker presented in Rémy's thesis [54], and the efficient representation for records presented below (section 4.3.4).

4.1.5 Extensible sums

The last extension we shall consider here is to allow "extending" a sum concrete type by adding new constructors to it, after it has been defined. For instance, after having declared `extensible type 'a list = Nil | Cons of 'a * 'a list`, one should be able to add a special case for, say, single-element lists, by declaring `extend 'a list with Singleton of 'a`. All values of the "old" `list` type also belong to the "new" `list` type. All functions defined before the extension, and taking an argument of the old type `list`, accept arguments of the new type `list`. Of course, previously defined pattern matchings fail on terms containing the new constructor `Singleton`.

This extension seems absolutely necessary to get a clean, general treatment of exceptions. In ML, exceptions have a symbolic name, and possibly one or more arguments. It is very natural to consider them as constructors of a given concrete type, `exc`. That way, the `raise` construct needs no special typing anymore, it is just a primitive with type `exc -> 'a`. Similarly, `try ... with ...` is now just syntactic sugar on top of the basic primitive `handle` with type `'a * (exc -> 'a) -> 'a`. The problem is that this type `exc` have to be extensible, since the user can define new exceptions (e.g. `exception End_of_file of int`), and this corresponds closely to extending the type `exc` with the constructor `End_of_file of int`.

Local extensions of a type raise a subtle typing issue: though the extension is local, say, to a module, it is not possible to check statically that the module does not export a value (or a function returning values) containing the constructor locally added to an extensible type. Continuing the example above, I can define the function `f x = Singleton x`, export it with type `'a -> 'a list`, but without exporting the constructor `Singleton`. This is not harmful by itself. What we must prevent is the extension of the same type with a constructor having the same name, but different arguments, as in `extend 'a list with Singleton of int`. This would allow the following violation of the type system:

```
let f x =
  extend 'a list with Singleton of 'a in Singleton x;;
let coerce_anything_to_int x =
  extend 'a list with Singleton of int in
    match f x with Singleton i -> i;;
```

One solution is to give different representations to all local extensions of a type, that is stop distinguishing on the constructor name, and use some kind of unique stamp instead. Another solution is to simply prohibit extending twice the same type with the same constructor, even if these extensions are local. ZINC implements an intermediate scheme: local extensions are local to a module (extensions that are local to an expression, as above, are not allowed, since I consider them useless), constructors are identified by their full, qualified name (e.g. `module1.Singleton` or `module2.Singleton`), and it is not possible to extend twice a given type with the same constructor in the same module.

For the time being, ZINC implements a restricted form of extensible sums: there is but one extensible type, the type `exc` of exception values, and declaring an exception is treated internally as extending it. This is mostly due to historical reasons (and to my own lazyness). Basically, everything is ready to implement general open sums.

4.2 A memory model

This section describes the format of the data handled by compiled programs. It engages not only the ZINC compiler (which must encode ML values in this format), but also the garbage collector. The garbage collector was written independently by Damien Doligez [23], and was supposed to be usable for other implementations as well; this lead us to try and find a memory model as general as possible.

In this model, a value is a 32-bit word which represents either a small integer, or a pointer in the heap. The GC must be able to distinguish between them at runtime, therefore a tagging scheme is necessary.

4.2.1 Unallocated objects

The only kind of unallocated (*unboxed*) objects is small integers. As the heap contains only 32-bit words, all pointers in the heap are 32-bit aligned, so their low-order bits are always 00. Integers are therefore encoded with a 1 as low-order bit: the integer n is represented by the 32-bit field $\overline{n} = 2n + 1$. Arithmetic operations on these 31-bit integers are fairly simple:

$$
\begin{aligned}
\overline{-n} &= 2 - \overline{n} \\
\overline{\mathrm{succ}\, n} &= \overline{n} + 2 \\
\overline{n + m} &= \overline{n} + \overline{m} - 1 \\
\overline{n - m} &= \overline{n} - \overline{m} + 1
\end{aligned}
$$

$$\begin{aligned}
\overline{n.m} &= (\overline{n-1})(\overline{m}/2)+1 \\
\overline{n/m} &= \frac{\overline{n-1}}{\overline{m}/2}+1
\end{aligned}$$

Tag schemes using the high-order bits are much less efficient, since they require to mask all operands and result for each operation. An alternate encoding is to take $\overline{n} = 2n$ and to shift all pointers by one; that way, integer arithmetic is even simpler (e.g. $\overline{n+m} = \overline{n} + \overline{m}$), but memory accesses can be slightly slower or not, depending on the hardware.

4.2.2 Allocated objects

Some structure for the heap is necessary to make garbage collection possible. The heap is therefore divided in blocks of arbitrary size n, preceded by a one-word header. All pointers into the heap must point to a header. The header consists of:

- 2 bits used by the garbage collector for marking purposes.
- 22 bits holding the size of the block, in words (header excluded)
- 8 bits for the kind of the block, since the GC needs to distinguish between blocks containing valid values, which should be recursively traversed, and blocks containing unstructured data such as the characters of a string, which are terminal objects for the GC. For this purpose, one bit would suffice, but we take advantage of the additional tag bits to encode compactly values of concrete types.

size = n	GC	tag = t
field 1		
$\vdots$		
field n		

The value t of the tag field has the following meaning: if $t < C$, the block is structured, it contains valid values in every field; if $t > C$, the block contains unstructured words only; and if $t = C$, the block represents a closure. Distinguishing closures is not very useful, except as a possible hint to the GC, and to write dirty polymorphic functions such as polymorphic equality. The constant C varies according to my mood, but stays fairly close to the highest possible value, $2^8 - 2$, since the tag field is mostly used to encode values of concrete types, and such values are always a structured block.

Pointers outside of the heap are permitted anywhere. This allows allocating statically data which should not be collected, either because such objects are never freed, or because we do not know how to collect them. For instance, this is the case of zero-length blocks, that is blocks consisting only in a header. Such blocks are perfectly legal, and very useful to encode constant constructors, but they cannot be allocated in the heap if the collector is a copying one, since there is no room to store a forwarding pointer in these blocks!

4.3 Encoding ML values

4.3.1 Atomic types

Small integers (type `int`) are of course represented by unboxed, 31-bit integers. This is also the case for characters (type `char`).

Floating-point numbers are allocated in the heap as unstructured blocks of length one, two or three words, depending on the possibilities of the hardware and on the required precision. An unboxed representation is possible, using the 10 suffix for instance, but this gives only 30 bits to represent floating-point numbers. Such a format lacks precision, and does not correspond to any standard format, so it involves fairly brutal truncations. Good old 64-bit, IEEE-standard floating point numbers seem more useful, even if they have to be allocated.

Arbitrary precision integers have to fit into unstructured blocks, that is vectors of 32-bit words. This is fine for the package of Vuillemin *et al.* [55] that I plan to use.

Strings are also stored into unstructured blocks. They are zero-terminated, and padded with additional zeros at the end to fill an entire number of words. To compute their length in constant time, and to avoid prohibiting null bytes in the middle of the string, the following scheme was proposed by Damien Doligez: the last byte of the last word contains an integer $n \in \{2, 3, 4, 5\}$ such that the length of the string is $4m - n$, where m is the size of the block in words. In other terms, assuming the last six characters of the string are $b_6 b_5 b_4 b_3 b_2 b_1$, the last two words of its representation are either $b_6 b_5 b_4 b_3 - b_2 b_1 01$, $b_5 b_4 b_3 b_2 - b_1 003$, $b_4 b_3 b_2 b_1 - 0004$, or $b_3 b_2 b_1 0 - 0005$. (He seems to find this quite enjoyable...)

Most extra atomic types, bit vectors for instance, can also be represented as unstructured blocks.

4.3.2 Functions

Functional values are of course represented by closures, that is pairs of a code pointer and an environment (a vector, that is a structured block). The code itself is not allocated in the heap, but in a separate, static zone, without garbage collection. The reason is that the program counter, and all return addresses stored on the return stack, generally point in the middle of a code block, not necessarily on its first word, and the garbage collector does not know how to cope with these *infix pointers*.

Therefore, code blocks will not be collected. This is not a problem in case of standalone programs, where code blocks are not dynamically created, and all of them are always accessable through global variables containing closures, hence code blocks are never freed, and it would be pointless to try to collect them! In case of interactive use, unused code blocks can appear when a global variable, previously bound to a closure, is rebound. Even in this case, code blocks have a relatively long lifespan, and occupy little space, due to the compactness of the bytecode, so one may hope that the static area will not grow too quickly. And explicit deallocation is often possible.

Once code blocks are put in the static area, they cannot contain pointers to the heap, since the GC would not know about them: it does not walk the static area at all; and chaining these pointers into the heap together, so that they become additional roots for the GC, would make the GC slower and more complex. This means that structured (non-atomic) constants cannot be at the same time allocated in the heap, and directly put into the code, as immediate operands. Either we allocate them in the static area as well (if they do not contain any mutable object, which could be updated later, and then point into the heap). Or we store them in the global table, along with the values of global variables, and access them through one additional indirection. I chose the latter solution, for the sake of simplicity.

4.3.3 Concrete types

The sum-of-products approach allows for a very compact representation of values of concrete types. Not only is the constructor stored in the same tuple as its arguments, but we try to put it in the tag field of the header of the tuple, so that the constructor does not require an extra field in the tuple. More precisely, the constructors $C_0, C_1, \ldots, C_n$ of a concrete type are numbered starting from zero, and the value $C_i(v_1, \ldots, v_k)$ is represented by a block of size k words, tagged i, containing the representations of $v_1, \ldots, v_k$.

size $= k$	GC	tag $= i$
v_1		
$\vdots$		
v_k		

size $= k+1$	GC	tag $= 0$
i		
v_1		
$\vdots$		
v_k		

Of course, this limits the number of constructors in a concrete type, since the tag field of structured blocks must lie between 0 and C (C is something like 253). For concrete types with more than C constructors, we must revert to a less compact representation for $C_i(v_1, \ldots, v_k)$: the $(k+1)$-tuple of the integer i and of the representations of $v_1, \ldots, v_k$. Such large sums are very rare in common programs, however.

Constant constructors

Constant constructors, that is constructors without arguments, are *not* treated specially: they are represented by a block of size 0, whose header holds the number of the constructor. This representation looks a bit unnatural at first; using integers to represent constant constructors seems more natural and more efficient. First, zero-sized blocks are just as inexpensive as integers, since there is no need to allocate a new one each time we evaluate a constant constructor: all zero-sized blocks having the same tag can be safely shared, since they contain no data, so we can preallocate them at link-time, and then we just have to manipulate (unique) pointers to these blocks, which is as efficient as manipulating integers. Then, having a uniform representation for constant and non-constant constructors is more efficient; in particular, discriminating on the constructor, as in

```
match x with C0 -> a | C1 x -> b | C2(x,y) -> c | C3 -> d | ...
```

can be done in a single jump indexed by the tag of the block:

```
case (tag_of x) of
  0 -> a;  1 -> b; 2 -> c; 3 -> d; ...
```

If we used integers for constant constructors, we would have to test whether the representation is an integer, if so jump through one table, if not jump through another:

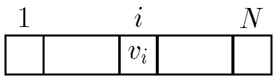
```
if is_int x then
  case x of 0 -> a; 1 -> d; ...
else
  case (tag_of x) of 0 -> b; 1 -> c; ...
```

The latter code is approximately two times slower than the first. This is critical, since all pattern matchings ultimately reduce to a sequence of tests of this kind.

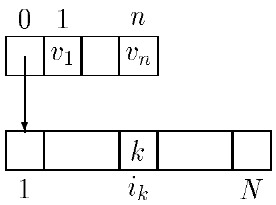
4.3.4 Records with inclusion

Without subtyping, a record with n fields labeled $l_1, \ldots, l_n$ can be represented by a vector of size n, and it is easy to know statically the offset of each label, that is the vector element holding the value associated to this label. With subtyping, we have the additional constraint that the representation of a record with fields $l_1, \ldots, l_n$ must be a valid representation for records whose fields are a subset of $l_1, \ldots, l_n$. This prevents us from having at the same time a compact representation and static determination of offsets.

Indeed, the only way to have statically-known offsets is to gather the set $L = \{l_1, \ldots, l_N\}$ of labels used in the whole program, and represent any record as a vector of size N, with the value v_i of label l_i in slot i.

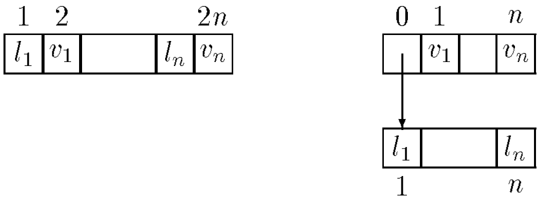


Of course, most records use only a small subset of L, so we waste a lot of space. In addition, we need to look at the whole program to attribute offsets, so this cannot be done until link time, and it prevents interactive use of the language. Notice however that the space wasting can be reduced by the use of a header: the record $(l_{i_1} = v_1, \ldots, l_{i_n} = v_n)$ can be represented by a $(n+1)$-tuple, with fields $1, \ldots, n$ holding values $v_1, \ldots, v_n$, and field 0 pointing to a vector of size N, the header, with slots $i_1, \ldots, i_n$ holding the integers $1, \ldots, n$, and the other slots undefined.



The trick is that this header can be shared among all records whose fields are $l_1, \ldots, l_n$. Conversely, we need an extra indirection to get the value associated with label l_i: first, look up the i^{th} element of the header, and use it as an offset in the tuple. This saves some space, but not enough to make this solution practical.

A more compact representation is to use association tables. For instance, Amber [13] represents the record $(l_1 = v_1, \ldots, l_n = v_n)$ as the $2n$-tuple $l_1, v_1, \ldots, l_n, v_n$. Retrieving the value of label l requires a run-time linear search through the tuple. This representation is fairly compact, and we can even save space by sharing a header containing the labels $l_1, \ldots l_n$ and only put the values $v_1, \ldots, v_n$ and a pointer to this header in the record representation.



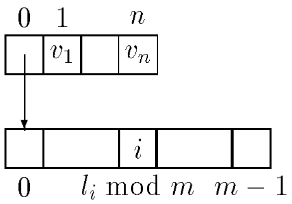
Also, it does not require the "closed program" assumption (i.e. that we know all labels in use). But the linear search makes retrieving quite slow. Indeed, accessing a field in a record takes time proportional to the number of fields in the record. Amber uses a caching scheme to improve performance: it stores the last accessed label and its offset in some fixed location, so that if it is accessed a second time in succession (such as in `r.lbl <- r.lbl+1`, for instance), we won't have to perform the search again. This trick seems quite efficient in practice, but the average behavior is still proportional to the record length. Of course, more efficient representations for association tables can be used, but it is still impossible to get constant-time access.

The discussion above overlooks an important fact, however. Strong static typing guarantees that when at run-time we request the value associated to a label l in a record r, then r will always contain a field labeled l. We do not know where it is stored in r, but we do know it is here. The previous representations do not make use of this assumption; actually, they are more suited to a dynamically-typed setting such as Lisp or Smalltalk, since they are able to detect at run-time that we are trying to access a nonexistent field. In other words, to represent the record $(l_{i_1} = v_1, \ldots, l_{i_n} = v_n)$, these approaches were to represent partial functions from the set of all labels into $\{1, \ldots, n\}$, while static typing allows us to represent only a total function from $\{l_1, \ldots, l_n\}$ to $\{1, \ldots, n\}$. This opens the way to efficient representations, both in space and time, as we shall see now.

First, I assume that all labels are integers. Associating (unique) integers to the label names is quite easy to do at link time, and it can be done incrementally in case of interactive use. Then, it remains to represent a mapping of a set of integers $\{l_1, \ldots, l_n\}$ on $\{1, \ldots n\}$. I suggest the following scheme: take an integer m such that

$$l_i \neq l_j \pmod{m} \qquad \text{for all } i \neq j \tag{4.1}$$

(such an m exists, e.g. $m = 1 + \max_{i \neq j} |l_i - l_j|$). Then, fill a vector h of size m as follows: slot number $l_i \bmod m$ holds the integer i, for all i. (This can be done as soon as labels are numbered, i.e. at compile time or at link time in the worst case.)



Condition 4.1 guarantees that no slot can be assigned two different integers. Then, the record $(l_1 = v_1, \ldots, l_n = v_n)$ itself is represented by the $(n+1)$-tuple $(h, v_1, \ldots, v_n)$.To get the value of the

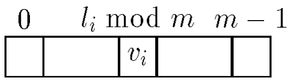
field labeled l, one computes $l \bmod \mathit{sizeof}(h)$, use the value as an offset in h, getting the offset of the desired value in the tuple. Therefore, access is performed in constant time. And the constant is quite low: one division and three memory accesses. This is the same order of magnitude than usual access in simple records (one indirection). And it is one order of magnitude faster than Amber-like schemes for medium-sized records.

It remains to show that the size of the shared header h is manageable. Let m_0 be the smallest m for which condition 4.1 holds. We have the obvious inequalities:

$$n \leq m_0 \leq 1 + \max_{i \neq j} |l_i - l_j|$$

This upper bound is not very reassuring, since it can be equal to the number of labels used in the program. However, it seems very pessimistic in practice; for instance, with $n = 2$, $l_1 = 1$ and $l_2 = 1000000$, we have $m_0 = 2$, while the upper bound is 1000000. Some simulations using random sets of labels between 0 and L, for various values of L, showed that the average m_0 is quite close to its lower bound n. (See figure 4.1.)

Actually, given the experimental results, one may consider skipping the header, and representing $(l_1 = v_1, \ldots, l_n = v_n)$ by a vector of size m_0, with v_i stored in the slot number $l_i \bmod m_0$:



This way, we save one indirection at each access. This might be worth the additional waste of space.

This representation is a first step toward efficient implementations of multiple inheritance, and to the best of my knowledge it has not been considered before, maybe because it requires strong static typing, while most object-oriented languages are dynamically typed. This gives me the opportunity to sing the praises of static typing once more: it is well-known that static typing leads not only to safer programs, but also to faster programs than dynamic typing, since there are less run-time checkings to perform; here, we see that, by not having to check at run-time the presence of a label in a record, we can change the *complexity* of the record access operation, and switch from linear time to constant time! Allelujah.

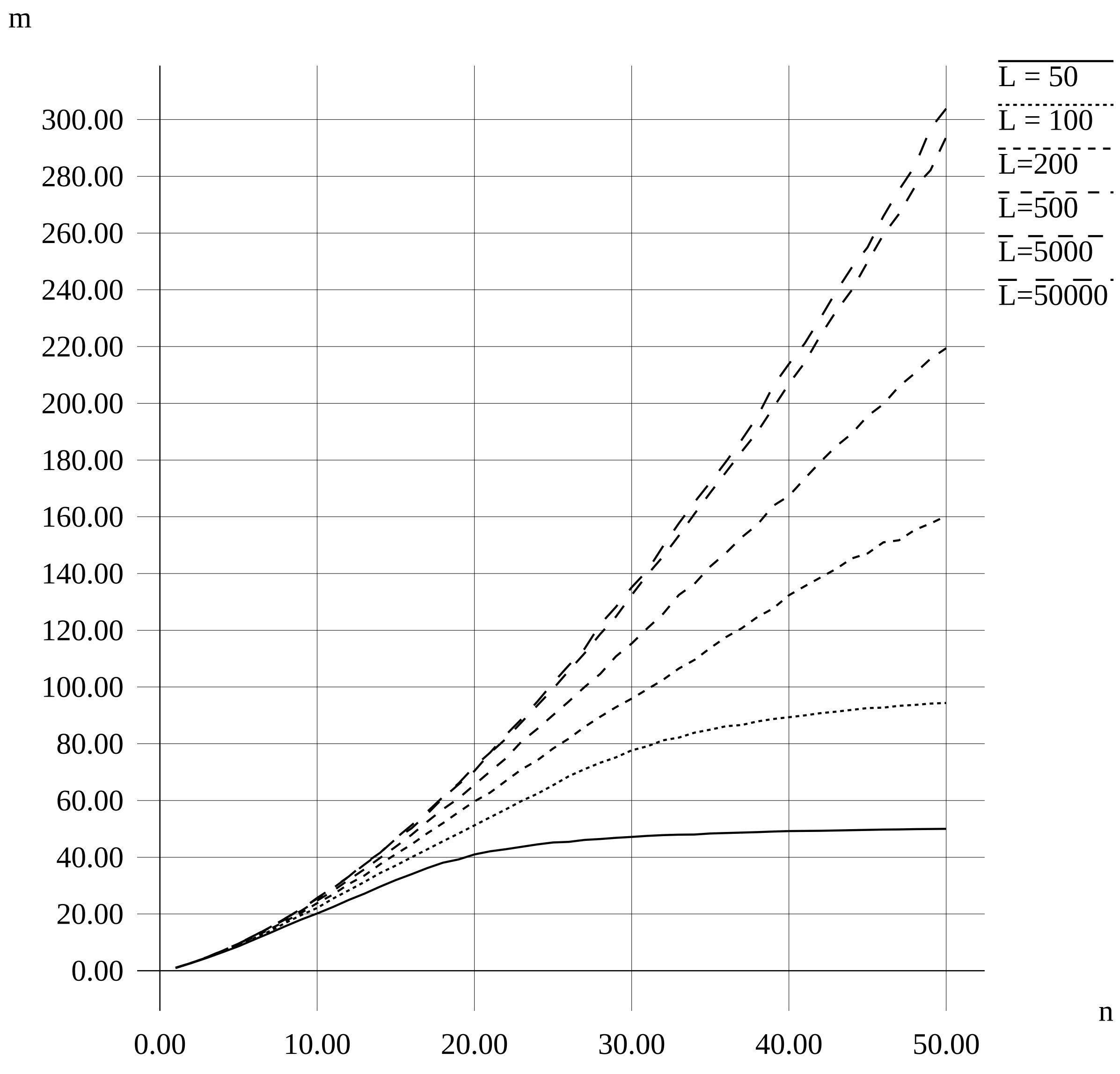


Figure 4.1: Average size m of the header, for random records with n labels between 0 and L.

Chapter 5

The compiler

This chapter describes the compiler for ZINC. It produces symbolic object code files from module implementations, and signature files from module interfaces; a separate program, the linker (described in next chapter), takes all these object code files and produces a bytecode file directly executable by the runtime system. The compiler is written in ML, in CAML for the time being, but I intend to bootstrap it as soon as possible. This chapter is illustrated with actual CAML code taken from the very sources of ZINC implementation.

5.1 Some intermediate representations

The ZINC compiler is a multi-pass compiler, since such compilers are much easier to write, maintain, and expand than single-pass compilers. These passes are partially determined by the kind of program representation they take, and the one they output. Therefore, I shall first present a few intermediate representations worth considering, before describing the various steps of the compiler. Not all of them are used in the ZINC compiler, since so many steps are useless for such a simple compiler.

5.1.1 Abstract syntax tree

The first internal representation, produced during parsing, is an abstract syntax tree. Here is the abstract syntax for expressions used for ZINC:

```
type expression =
    mutable Zident of expr_ident                       (* variable "x" or "module.y" *)
  | Zconstant of struct_constant                       (* constant "3.14" *)
  | Zconstruct of constr_desc global & expression list (* constructor application *)
  | Zapply of expression & expression list             (* multiple application *)
  | Zlet of (pattern & expression) list & expression   (* local binding *)
  | Zletrec of (pattern & expression) list & expression (* local recursive binding *)
  | Zfunction of (pattern list & expression) list      (* functions (with pattern matching) *)
  | Ztrywith of expression & (pattern & expression) list (* exception handling *)
  | Zsequence of expression & expression               (* sequence *)
  | Zcondition of expression & expression & expression (* if...then...else... *)
  | Zwhile of expression & expression                  (* "while" loop *)
  | Zsequand of expression & expression                (* sequential "and" *)
  | Zsequor of expression & expression                 (* sequential "or" *)
```

```
  | Zconstraint of expression & type_syntax            (* type constraint *)
  | Zassign of string & expression                     (* assignment *)
and expr_ident =
    Zglobal of value_desc global        (* global variable, with its descriptor *)
  | Zlocal of string                    (* local variable *)
```

using the following auxiliary type definitions:

```
type pattern =
    Zwildpat                                           (* underscore "_" *)
  | Zvarpat of string                                  (* variable *)
  | Zaliaspat of pattern & string                      (* alias "... as y" *)
  | Zconstantpat of atomic_constant                    (* constant *)
  | Zconstructpat of constr_desc global & pattern list (* construction *)
  | Zorpat of pattern & pattern                        (* alternative *)
  | Zconstraintpat of pattern & type_syntax            (* type constraint *)

type atomic_constant =
    ACnum of num
  | ACint of int
  | ACfloat of float
  | ACstring of string
and struct_constant =
    SCatom of atomic_constant
  | SCblock of constr_tag & struct_constant list
```

It is a compromise between contradictory requirements. On the one hand, the abstract syntax should be as close as possible to the "concrete syntax" (the textual representation), in order to keep parsing simple, and also to be able to pretty-print the abstract syntax and get something close to the original text. On the other hand, the abstract syntax should reflect the "true" meaning of the program, and this means that syntactic sugar of all kinds has been translated into more basic constructs during parsing, and also that ambiguous constructs have been disambiguated. For instance, in genuine ML, an identifier can represent a constructor as well as a global or local variable, and it is not possible to distinguish syntactically between them, yet they perform quite differently, especially in patterns! However, the pretty-printing requirement is crucial only if we choose to report the locations of typing errors by pretty-printing the corresponding piece of abstract syntax, as CAML does [62, pp. 395–396]. We shall soon see an alternate approach. Therefore, I choose to favor the second requirement, and try to reflect the true semantics of the source program. For instance, the type definition above clearly distinguishes constructors from variables, and constructor application from function application.

It would be useful to put some annotations at each node of the syntax tree. For instance, each subtree should contain a pointer to the corresponding source text (e.g. two integers giving the position in the source file of the first and the last characters). Then it would be trivial to report the location of an error, by reprinting the corresponding source code, or even by sending a message to some clever text editor, that will highlight the guilty expression directly in the source text. That way, we get perfectly accurate error locations, and we save the non-trivial work of writing a good pretty-printer for ML. These annotations are very easy to synthesize during parsing: it suffices to transform all grammar rules such as (in Yacc notation)

```
A : B1 B2 B3 { $$ = ... $1 ... $2 ... $3 ...; }
```

into

```
A : B1 B2 B3 { $$.start = $1.start; $$.end = $3.end;
$$.syntax = ... $1.syntax ... $2.syntax ... $3.syntax ...; }
```

and some parser generators (e.g. Bison) even do it automatically. Then, the concrete type `expression` becomes:

```
type expression = { start: int; end: int; syntax: expression_syntax }
and expression_syntax =
    mutable Zident of expr_ident
  | Zconstant of struct_constant
  | Zconstruct of constr_desc global & expression list
  | ...
```

Another useful annotation is the type inferred for each subexpression. This would make type-checking error report easier to understand, especially in conjunction with a clever text editor (e.g. just select a subexpression and see the type inferred for it). It would also allow a debugger to display the value of any subexpression, since to print a value, it is necessary to know its type.

5.1.2 Enriched λ-calculus

A classical intermediate language [50, chap. 3] is the λ-calculus with constants, primitive operations, global variables, conditional constructs, and `let` and `let rec` bindings. The fundamental difference with abstract syntax is that pattern matching has been expanded into sequences of elementary tests. Also, the remaining ambiguities have been fully resolved: local and global identifiers are distinguished; globals bound to primitive operations (e.g. `+`) have been identified and expanded; all global identifiers are now qualified with the module they belong to; and name clashes among local variables have been solved either by generating new, unique names, or by using de Bruijn's indexes [10]. The latter solution is elegant and saves work during the translation to abstract machine code (which uses de Bruijn's indexes anyway), but complicates many clever analysis, where one wants to gather information on all local variables of an expression. Here is an example using de Bruijn's numbers:

```
type lambda =
    Lvar of num                           (* local variable (de Bruijn's number) *)
  | Lconst of struct_constant             (* constants *)
  | Lapply of lambda & lambda list        (* multiple application *)
  | Lfunction of num & lambda             (* curried function (num = arity) *)
  | Llet of lambda list & lambda          (* local binding *)
  | Lletrec of lambda list & lambda       (* local recursive binding *)
  | Lprim of primitive & lambda list      (* primitive operation *)
  | Lcond of lambda & (atomic_constant & lambda) list    (* equality tests with constants *)
  | Lswitch of num & lambda & (constr_tag & lambda) list (* jump through table *)
  | Lstaticfail                           (* as "raise failure", but with static scope *)
  | Lstatichandle of lambda & lambda      (* handler for static failures *)
  | Lhandle of lambda & lambda            (* handler for "real" exceptions (dynamic scope) *)
  | Lifthenelse of lambda & lambda & lambda    (* conditional *)
  | Lsequence of lambda & lambda          (* sequence *)
  | Lwhile of lambda & lambda             (* "while" loop *)
```

```
  | Lsequand of lambda & lambda               (* sequential "and" *)
  | Lsequor of lambda & lambda                (* sequential "or" *)

and primitive =
    Pget_global of string & string          (* access to global variables *)
  | Pset_global of string & string          (* modification of global variables *)
  | Pmakeblock of constr_tag                (* building a tuple in the heap *)
  | Pfield of num                           (* accessing one field of a tuple *)
  | Psetfield of num                        (* modifying one field of a tuple *)
  | Pccall of num                           (* calling a C primitive *)
  | Praise                                  (* raising an exception *)
  | Ptest of bool_test                      (* comparisons *)
  | ...                                     (* arithmetic operations, and much more *)
and bool_test =
    Pequal_test
  | Pnotequal_test
  | Peq_test
  | Pnoteq_test
  | Pnum_test of num prim_test
  | Pint_test of int prim_test
  | Pfloat_test of float prim_test
  | Pstring_test of string prim_test
  | Peqtag_test of constr_tag
  | Pnoteqtag_test of constr_tag
and 'a prim_test =
    PTeq
  | PTnoteq
  | PTnoteqimm of 'a
  | PTlt
  | PTle
  | PTgt
  | PTge
```

`Lcond` and `Lswitch` are both multi-way tests, used for efficient pattern matching: `Lcond` matches its argument against a list of constants, and returns the value of the associated expression. In other terms, `Lcond(exp, [cst1,exp1; ...;cstN,expN])` means

```
let x = exp in
  if x = cst1 then exp1
  elsif ...
  elsif x = cstN then expN
  else raise staticfail
```

Similarly, `Lswitch` discriminates on the tag of its argument, which is assumed to be a tuple: `Lswitch(n, exp, [tag1, exp1; ..., tagN, expN])` means

```
let x = tag_of exp in
  if x = tag1 then exp1
  elsif ...
  elsif x = tagN then expN
  else raise staticfail
```

`Lstaticfail` and `Lstatichandle` implement statically-scoped exceptions (which cannot escape the current function); such exceptions are compiled very efficiently by a mere `goto`, and they permit to express code sharing in this representation.

This intermediate representation is both simpler (due to the expansion of pattern matching) and more precise than abstract syntax, while retaining the overall program structure; it is therefore the representation of choice to perform program analysis and transformation, such as compile-time β-reduction ("function inlining") and constant propagation.

5.1.3 Graph of code

Enriched λ-calculus is still a tree-like structure. Machine code is essentially "flat", that is list-like. Hence we will have to sequentialize λ-expressions at some point. It might be convenient to break this process into two parts: first, generate *basic blocks*, that is chunks of code whose execution is purely sequential, and connect their entry and exit points; then, flatten this graph-like structure by inserting labels and jumps. The intermediate graph of code is well-suited to express code sharing and flow of control. For instance, `while` constructs simply become loops in the graph. Here is such a structure for ZINC machine code (it is not used in the current ZINC compiler):

```
type graph_code =
      Gstop                          (* end of toplevel phrase *)
    | Greturn                        (* end of function *)
    | Gtermapply                     (* tail function application *)
    | Gquote of quoteconst & code    (* loading a constant *)
    | Ggrab of code                  (* eat up one argument *)
    | Gpush of code                  (* push one argument *)
    | ...                            (* a lot more simple instructions *)
    | Gclosure of labeled_code & code (* build up a closure *)
    | Gbranch of labeled_code        (* unconditional jump *)
    | Gtest of labeled_code & labeled_code (* conditional jump (if-then-else) *)
    | Gswitch of labeled_code vect   (* jump through table *)

and labeled_code =
    { mutable lbl : code_label;      (* the label *)
      code : graph_code }            (* the contents of the basic block *)

and code_label = Unlabeled | Label of num
```

Most instructions take as argument the code that should be executed after them (their continuation). Some don't have any continuation. Other have several continuations, for instance conditional tests. To have a more compact representation, we use two different types: `graph_code` and `labeled_code`. Sharing is ignored for objects of type `graph_code`; therefore, they represent the body of a basic block. Objects of type `labeled_code` bear an additional "label" field, which allows to detect sharing during linearization. Labels are initially undefined, but they get defined during the flattening phase, when they are visited the first time; the second time, the labeled code is not emitted again, but replaced by a jump to the code generated the first time. Most instructions are intended to continue execution sequentially, so their continuation is of type `graph_code`. Branch instructions explicitly connect basic blocks together, so their continuations are of type `labeled_code`. Notice how these subtle structure constraints are automatically enforced by the type system.

5.1.4 Linear code

The last representation is a linear list of instructions, as close as possible to the string of bytes given to the interpretor. Some informations have to remain symbolic, however. For instance, global variables are still named, while the bytecode refers to them by their address; this is because global variable allocation can only be done at link time. Similarly, jumps refer to labels, while the bytecode contains the offsets of the destinations; the reason is that the size of bytecode instructions are not known to the compiler. Here is the type of instructions used in ZINC:

```
type instruction =
    Kquote of struct_constant         (* put a constant in the accumulator *)
  | Kget_global of string & string    (* get the value of a global ident *)
  | Kset_global of string & string    (* set the value of a global ident *)
  | Kaccess of num                    (* fetch a local variable from the env. *)
  | Kgrab                             (* eat one argument and put it in the env. *)
  | Kpush                             (* push one argument *)
  | Kpushmark                         (* push a mark *)
  | Klet                              (* add a value in front of the env. *)
  | Kendlet of num                    (* remove the first values from the env. *)
  | Kdummies of num                   (* add dummy closures to the env. *)
  | Kupdate of num                    (* update a dummy closure with its final value *)
  | Kapply                            (* function call *)
  | Ktermapply                        (* tail function call *)
  | Kstartfun                         (* entry point in a function *)
  | Kreturn                           (* end of a function *)
  | Kclosure of label                 (* builds a closure *)
  | Kmakeblock of constr_tag & num    (* allocates a tuple in the heap *)
  | Kprim of primitive                (* various primitive operations *)
  | Kpushtrap of label                (* set up an exception handler *)
  | Kpoptrap                          (* remove an exception handler *)
  | Klabel of label                   (* defines a label *)
  | Kbranch of label                  (* unconditional jump *)
  | Kbranchif of label                (* conditional jump *)
  | Kbranchifnot of label             (* reverse conditional jump *)
  | Kstrictbranchif of label          (* almost the same thing as above *)
  | Kstrictbranchifnot of label       (* (quick fix for a subtle bug) *)
  | Ktest of bool_test & label        (* various conditional jumps (= < > etc) *)
  | Kswitch of label vect             (* jump through table *)

and label = Nolabel | Label of num
```

5.2 Compilation steps

Before delving into the details of the compiler, let me give here an overview of the compilation process. Zinc compiles files phrase by phrase. For interface files, each phrase is a declaration (of value, type, or exception); all these declarations are parsed, then stored into an initially empty symbol table. At the end of the file, the symbol table contains the full signature of the module; it is written to the compiled interface file.

Implementation files can hold three different kinds of phrases. Simple expressions (such as `1+2;;`) are parsed, typechecked, then translated to the `lambda` intermediate representation, then to symbolic ZAM code, which is appended to the object code file. Value definitions (such as `let f,g = h 3;;`) are compiled similarly; in addition, the defined globals (`f` and `g` here) are entered into the symbol table, with their inferred types. Type and exception declarations are simply stored in the symbol table. At the end of the file, the symbol table is compared with the one contained in the compiled interface file, to check that the implementation defines everything the interface declares.

5.2.1 Parsing

The parsing phase produces the abstract syntax tree for each phrase of the input file. For the time being, it uses the grammar facility of CAML [62, chap. 16], which combines a $LR(1)$ parser generator built on top of Yacc with type checking and inference for the semantic actions and grammar attributes, and a parameterized lexical analyzer. Here are the main entries of ZINC grammar definition, in CAML syntax. (This grammar was written by Yannick Martel.)

```
(* Parsing auxiliaries (must come first for precedence reasons!) *)

rule Simple_expr_list =
  parse Simple_expr e; Simple_expr_list el   -> e :: el
      | Simple_expr e                        -> [e]

and Expr_comma_list =
  parse Expr e; ","; Expr_comma_list el      -> e :: el
      | Expr e  with precedence ","          -> [e]

and Expr_constr_list =
  parse "("; Expr_comma_list el; ")"         -> el
      | Simple_expr e                        -> [e]

and Expr_sm_list =
  parse Expr e; ";"; Expr_sm_list el         -> Zconstruct(cons_desc, [e;el])
      | Expr e                               -> Zconstruct(cons_desc, [e; Zconstruct(nil_desc, [])])

(* Expressions *)

and entry Expr =
  parse Expr e; Simple_expr_list el  with precedence APP -> Zapply(e,el)
      | Constr c; Expr_constr_list el with precedence APP-> Zconstruct(c, el)
      | Simple_expr e                        -> e
      | Expr e; ","; Expr_comma_list el      -> Zconstruct(tuple_desc, e::el)
      | "-"; Expr e1 with precedence UMINUS  -> Zapply(Zident(Zlocal "~"), [e1])
      | "!"; Expr e1 with precedence UMINUS  -> Zapply(Zident(Zlocal "!"), [e1])
      | Expr e1; "*"; Expr e2                -> Zapply(Zident(Zlocal "*"), [e1;e2])
      | Expr e1; "/"; Expr e2                -> Zapply(Zident(Zlocal "/"), [e1;e2])
      | Expr e1; "+"; Expr e2                -> Zapply(Zident(Zlocal "+"), [e1;e2])
      | Expr e1; "-"; Expr e2                -> Zapply(Zident(Zlocal "-"), [e1;e2])
      | Expr e1; "::"; Expr e2               -> Zconstruct(cons_desc, [e1;e2])
      | Expr e1; "@"; Expr e2                -> Zapply(Zident(Zlocal "@"), [e1;e2])
      | Expr e1; "^"; Expr e2                -> Zapply(Zident(Zlocal "^"), [e1;e2])
      | Expr e1; "<"; Expr e2                -> Zapply(Zident(Zlocal "<"), [e1;e2])
      | Expr e1; ">"; Expr e2                -> Zapply(Zident(Zlocal ">"), [e1;e2])
```

```
      | Expr e1; "="; Expr e2               -> Zapply(Zident(Zlocal "="), [e1;e2])
      | Expr e1; "=="; Expr e2              -> Zapply(Zident(Zlocal "=="), [e1;e2])
      | Expr e1; ">="; Expr e2              -> Zapply(Zident(Zlocal ">="), [e1;e2])
      | Expr e1; "<="; Expr e2              -> Zapply(Zident(Zlocal "<="), [e1;e2])
      | Expr e1; "<>"; Expr e2              -> Zapply(Zident(Zlocal "<>"), [e1;e2])
      | "not"; Expr e                       -> Zapply(Zident(Zlocal "not"), [e])
      | Expr e1; "&"; Expr e2               -> Zsequand(e1,e2)
      | Expr e1; "or"; Expr e2              -> Zsequor(e1,e2)
      | Expr e1; ":="; Expr e2              -> Zapply(Zident(Zlocal ":="), [e1;e2])
      | IDENT s; "<-"; Expr e               -> Zassign(s,e)
      | "if"; Expr e1; "then"; Expr e2; "else"; Expr e3  -> Zcondition(e1,e2,e3)
      | "if"; Expr e1; "then"; Expr e2      -> Zcondition(e1,e2, Zconstruct(ok_desc,[]))
      | "while"; Expr e1; "do"; Expr e2     -> Zwhile(e1,e2)
      | Expr e1; ";"; Expr e2               -> Zsequence(e1,e2)
      | "match"; Expr e; "with"; Function_match M -> Zapply(Zfunction M, [e])

      | "let"; Binding_list L; "in"; Expr e        with precedence LET  -> Zlet(L,e)
      | "let"; "rec"; Binding_list L; "in"; Expr e with precedence LET  -> Zletrec(L,e)
      | "fun"; Fun_match M                  -> Zfunction M
      | "function"; Function_match M        -> Zfunction M
      | "try"; Expr e; "with"; Try_match M   -> Ztrywith(e,M)

      | Expr e; "where"; Binding_list L        with precedence WHERE   -> Zlet(L,e)
      | Expr e; "where"; "rec"; Binding_list L  with precedence WHERE   -> Zletrec(L,e)

and Simple_expr =
  parse Struct_constant sc                  -> Zconstant sc
      | Ext_ident gr                        -> ident_of_gr gr
      | "("; ")"                            -> Zconstruct(ok_desc,[])
      | "["; Expr_sm_list e; "]"            -> e
      | "["; "]"                            -> Zconstruct (nil_desc, [])
      | "("; Expr e; ":"; Type ty; ")"      -> Zconstraint(e,ty)
      | "("; Expr e; ")"                    -> e
      | "begin"; Expr e; "end"              -> e
      | Constr c                            -> Zconstruct (c, [])

and entry Struct_constant =
  parse Atomic_constant ac                  -> SCatom ac

and entry Atomic_constant =
  parse NUM n                               -> ACnum n
      | INT i                               -> ACint i
      | FLOAT f                             -> ACfloat f
      | STRING s                            -> ACstring s

(* Bindings and matchings *)

and Fun_match =
  parse Simple_pattern_list pl; "->"; Expr e; "|"; Fun_match M  -> (pl,e) :: M
      | Simple_pattern_list pl; "->"; Expr e                    -> [pl,e]

and Function_match =
  parse Pattern p; "->"; Expr e; "|"; Function_match M  -> ([p],e) :: M
      | Pattern p; "->"; Expr e                          -> [[p],e]
```

```
and Try_match =
  parse Pattern p; "->"; Expr e; "|"; Try_match M -> (p,e) :: M
      | Pattern p; "->"; Expr e                   -> [p,e]

and Binding_list =
  parse Binding b; "and"; Binding_list bl -> b :: bl
      | Binding b                         -> [b]

and Binding =
  parse Pattern p; "="; Expr e  with precedence DEFINE  -> p,e
      | Ide s; Simple_pattern_list pl; "=";
Expr e  with precedence DEFINE                  -> Zvarpat s, Zfunction [pl,e]

(* Patterns *)

and Pattern_sm_list =
  parse Pattern p; ";"; Pattern_sm_list pl  -> Zconstructpat(cons_desc, [p;pl])
      | Pattern p                           -> Zconstructpat(cons_desc,
                                                  [p; Zconstructpat(nil_desc,[])])

and Pattern_constr_list =
  parse "("; Pattern_comma_list pl; ")"     -> pl
      | Simple_pattern p                    -> [p]

and Pattern_comma_list =
  parse Pattern p; ","; Pattern_comma_list pl -> p :: pl
      | Pattern p  with precedence ","        -> [p]

and Simple_pattern_list =
  parse Simple_pattern p; Simple_pattern_list pl -> p :: pl
      | Simple_pattern p                         -> [p]

and entry Pattern =
  parse Simple_pattern p                       -> p
      | Pattern p; "as"; IDENT s               -> Zaliaspat(p,s)
      | Pattern p1; "::"; Pattern p2           -> Zconstructpat(cons_desc, [p1;p2])
      | Pattern p1; ","; Pattern_comma_list pl -> Zconstructpat(tuple_desc, p1::pl)
      | Constr c; Pattern_constr_list pl       -> Zconstructpat (c, pl)
      | Pattern p1; "|"; Pattern p2            -> Zorpat(p1,p2)

and Simple_pattern =
  parse Atomic_constant ac               -> Zconstantpat ac
      | "_"                              -> Zwildpat
      | Ide s                            -> Zvarpat s
      | Constr c                         -> Zconstructpat(c,[])
      | "("; ")"                         -> Zconstructpat(ok_desc,[])
      | "["; "]"                         -> Zconstructpat(nil_desc,[])
      | "["; Pattern_sm_list p; "]"      -> p
      | "("; Pattern p; ":"; Type ty; ")" -> Zconstraintpat(p,ty)
      | "("; Pattern p; ")"              -> p

(* etc., etc. *)
```

Parsing ML is not easy: there is a huge number of possible ambiguities, since most constructs are "open"(i.e. without closing delimitor), and even more so if one wants to provide the same amount of syntactic sugar provided by CAML, for compatibility reasons. The conventional approach is to multiply the nonterminals of the grammar (e.g. having several entry points for expressions, corresponding to expressions with various priorities), in order to resolve these ambiguities in the grammar definition itself. This leads to very convoluted and hard to read grammars; the actual grammar of CAML [62, appendix Z] is a very convincing example of this tendency!

Yannick Martel and I dismissed this approach and chose instead to write a very natural grammar, but highly ambiguous, and rely on Yacc's disambiguating mechanisms (precedences, associativities, etc.) to get correct parsing. The result is a concise grammar (about five times smaller than the corresponding fragment of CAML grammar), fairly easy to read (even for a non-specialist like me), but with some 50 reduce/reduce conflicts and 500 shift/reduce conflicts! As it is impossible to check each conflict to see if it was resolved correctly, it is not possible to know exactly what the resulting parser does, and getting a parser that works correctly on test CAML programs required quite a bit of trial and error...Yacc is partly responsible for this situation, since the conflict report and disambiguation tools it provides are fairly crude. For instance, precedences and associativities are *not* taken into account to solve reduce/reduce conflicts. And to understand where the conflicts lie, one has to look at the generated automata, which requires training and fortitude. We conclude that common parser generator technology is not yet powerful enough to easily tackle languages with such a tricky syntax as CAML.

I experienced a few problems with lexical analysis, too. CAML does not provide a lexer generator such as Lex, indeed it always uses the same lexer. Admittedly, this lexer can be parameterized by e.g. the set of keywords or the string delimitor. However, it is not possible to add new lexical units unused in CAML, for instance the token `CHAR` corresponding to character constants such as `‘a‘`. Also, it is not possible to perform specific actions when a token has been recognized. For instance, when a valid identifier is read, one may want to query the symbol table to determine whether it is a constructor or a variable name, and return accordingly the token `CONSTR` with the constructor description, or the token `VAR` with the variable description. Admittedly, this could be done in the actions of the grammar, but complicates it dramatically, so much that I had to use different syntaxes for constructors and variable names: constructors are prefixed with a quote, as in `'true`.

bootstrap, I have written a simple lexer generator in the same vein as Lex. Inputs look like functions defined by pattern matching, except that patterns are replaced by regular expressions. Actions are arbitrary ML expressions; most of the time, they compute values of a user-defined concrete type, representing all the possible tokens along with their attributes. For instance, a lexer for ZINC would look like:

```
type token =
    Tint of int
  | Tstring of string
  | Tident of string
  | Tconstructor of constr_desc
  | Tfunction | Tlet | Tin | ...;;

<:lexer<let lex_main = function
    [‘0‘-‘9‘]+ as s -> Tint(int_of_string_deci s)
  | ‘0‘ (‘x‘|‘X‘) ([‘0‘-‘9‘, ‘A‘-‘F‘, ‘a‘-‘f‘]+ as s) -> Tint(int_of_string_hexa s)
  | ‘"‘ ( ([except ‘"‘] | (‘\\‘ [‘"‘, ‘n‘, ‘r‘, ‘\\‘]))* as s) ‘"‘ -> Tstring(parse_string s)
  | "(*" -> lex_comment instream; lex_main instream
```

```
  | "function" -> Tfunction
  | (* etc. *)
and lex_comment = function
    "*)" -> ()
  | `"` ([except `"`] | (`\\` [`"`, `n`, `r`, `\\`]))*  `"` -> lex_comment instream
  | "(*" -> lex_comment instream; lex_comment instream
  | _ -> lex_comment instream >>
```

The sets of regular expressions are transformed into deterministic automata using the Berry-Sethi algorithm [9] (see also [3, pp. XXX]). The resulting automata are *not* output as transition tables to be interpreted by some driver program, but directly as a set of mutually recursive functions, one per state of the automaton, performing pattern-matching on the next character, as for instance:

```
let rec state_0_2 lexbuf =
 match getchar lexbuf with
    `9`|`8`|`7`|`6`|`5`|`4`|`3`|`2`|`1` -> state__0_1 lexbuf
  | `0` -> state__0_1_3_4 lexbuf
  | _ -> backtrack lexbuf

and state__0_1_3_4 lexbuf =
 remember lexbuf action_0;
 match getchar lexbuf with
    `9`|`8`|`7`|`6`|`5`|`4`|`3`|`2`|`1`|`0` -> state__0_1 lexbuf
  | `x`|`X` -> state_5 lexbuf
  | _ -> backtrack lexbuf

and state_5 lexbuf =
 match getchar lexbuf with
    `f`|`e`|`d`|`c`|`b`|`a`|`F`|`E`|`D`|`C`|`B`|`A`|`9`|`8`|`7`|`6`|`5`|
    `4`|`3`|`2`|`1`|`0` -> state__1 lexbuf
  | _ -> backtrack lexbuf

and ...
```

This way, execution can be very fast (especially if the compiler is able to generate indexed jumps for pattern-matching), and this representation is much more compact than plain transition tables, without reverting to intricate compaction schemes. It is quite easy to implement, thanks to CAML's extensive possibilities to manipulate its own syntax trees [62, chap. 18]. There is no need for a special typechecking mechanism, since a lexer definition is well-typed (i.e. all actions belonging to the same entry point have the same type) if and only if it translates to a well-typed ML program.

For bootstrapping purposes, similar needs arise for parsing. It should be possible to reuse most of the parser generator of CAML, and to modify only the output routines. Currently, it generates tables driving a pushdown automaton. A functional encoding, in the same vein as above, can be considered.

5.2.2 Type inference

For the time being, there is no typechecker in the ZINC implementation. Several simple implementations of the type inference algorithm of ML can be found in the literature. First, [50] explains the algorithm in the setting of another language (Miranda) and presents a very naive, very inefficient implementation, due to the fact that it does not use destructive unification, Miranda *oblige*. A more realistic typechecker for a subset of ML can be found in Michel Mauny's course notes [45]. It does use destructive unification, and the sole source of inefficiency is the way types are generalized. Otherwise, it is comparable with "real" typecheckers such as the one used in the CAML system. For very efficient type inference, the reader is referred to Didier Rémy's thesis [54]. He gives the sources of two prototypes, one for the core ML language, the other extended with records and variants with subtyping. It is the latter one that I plan to integrate to the ZINC compiler.

5.2.3 Compiler, front end

The first pass of the compiler properly speaking translates the abstract syntax tree into a term of the enriched λ-calculus. It performs the following transformations:

- expansion of pattern-matching into sequences of simple tests
- replacing local variables by de Bruijn indexes and access paths
- recognizing primitive operations
- simple constant folding.

I shall first give the source code of the main function (the one that compiles an expression) interleaved with a few comments. Then, the handling of local environments will be explained. Compiling pattern-matching shall be explained in the next section.

```
let rec translate env = transl where rec transl = function
    Zconstant cst ->
      Lconst cst
  | Zconstruct(c,args) ->
      let tr_args = map transl args in
        if c.desc.pure then
          try
            Lconst(SCblock(c.desc.tag,
                           map (fun Lconst c -> c
                                  |    _     -> failwith "const") tr_args))
          with failure _ ->
            Lprim(Pmakeblock c.desc.tag, tr_args)
        else
          Lprim(Pmakeblock c.desc.tag, tr_args)
```

In case of constructor application $C(e_1, \ldots, e_n)$ where C is "pure", that is none of its arguments is mutable, we trap the case where $e_1, \ldots, e_n$ are all constants. In this case, there is no need to build a new block in the heap at each evaluation, we can share a single one, built once for all at link time. Therefore, the whole expression is transformed into a (structured) constant.

```
  | Zapply(Zfunction matchlist, args) ->
      if size_of_match matchlist = length args then
        Llet(transl_let env args, transl_match partial_fun matchlist)
      else
        Lapply(transl funct, map transl args)
```

```
| Zapply((Zident(Zglobal
              { desc = { prim = ValuePrim(arity,prim); _ }; _}) as f),
       args) ->
    if arity = length args then
      Lprim(prim, map transl args)
    else
      Lapply(transl f, map transl args)
| Zapply(funct, args) ->
    Lapply(transl funct, map transl args)
```

Translating an application is straightforward, but we have to trap two special cases. The first is a function immediately applied to all its arguments; this situation corresponds to a `match...with...` construct. In this case, there is no need to actually evaluate the function and immediately apply it, it suffices to store the values of the argument in the environment and perform pattern-matching on the fly. The second special case is the application of a primitive operation to all its arguments. A primitive operation is a global variable bearing a `ValuePrim` annotation, along with its arity and the corresponding value of the concrete type `primitive`. This case is transformed into a `Lprim(prim, ...)` term.

```
| Zident(Zlocal s) as E ->
    translate_access s env
| Zident(Zglobal{ desc = { prim = ValuePrim(arity,prim); _ }; _}) ->
    let rec vars_list = fun 0 -> [] | n -> Lvar(n-1) :: vars_list(n-1) in
      Lfunction(arity, Lprim(prim, vars_list arity))
| Zident(Zglobal g) ->
    Lprim(Pget_global(g.modname, g.name), [])
| Zassign(s,E) ->
    translate_update s env (transl E)
```

Local variables are fetched in the environment; `translate_access` generates the right access sequence based on the structure of the environment described by `env`. Global variables are translated into the special primitive `Pget_global`. We have to trap here the case of primitives used as functions (e.g. `it_list (prefix +) 0 L`) or partially applied (`prefix + 5`). Then, we actually have to build the corresponding (curried) function, for instance `fun x y -> x+y`.

```
| Zsequence(E1, E2) ->
    Lsequence(transl E1, transl E2)
| Zcondition(Eif, Ethen, Eelse) ->
    Lifthenelse(transl Eif, transl Ethen, transl Eelse)
| Zwhile(Econd, Ebody) ->
    Lwhile(transl Econd, transl Ebody)
| Zsequand(E1, E2) ->
    Lsequand(transl E1, transl E2)
| Zsequor(E1, E2) ->
    Lsequor(transl E1, transl E2)
| Zconstraint(E, _) ->
    transl E
```

Simple control structures have exact counterparts in the intermediate representation, so we just have to translate recursively their arguments.

```
  | Zfunction(matchlist) ->
      Lfunction(size_of_match matchlist, transl_match partial_fun matchlist)
  | Ztrywith(body, pat_exprlist) ->
      Lhandle(transl body,
              transl_match partial_try
                          (map (fun (pat,expr) -> [pat], expr) pat_exprlist))
  | Zlet(pat_exprlist, body) ->
      let cas = map fst pat_exprlist
      and args = map snd pat_exprlist in
        Llet(transl_let env args, transl_match partial_let [cas, body])
  | Zletrec(pat_exprlist, body) ->
      let cas = map fst pat_exprlist
      and args = map snd pat_exprlist in
      let transl_in_newenv = translate (make_env env cas) in
        Lletrec(map transl_in_newenv args,
                transl_match partial_let [cas, body])

  and transl_match failure_code casel =
    let transl_action (patlist, expr) =
      patlist, translate (make_env env patlist) expr in
    translate_matching failure_code (map transl_action casel)

  and transl_let env = function
     []  -> []
  | a::L -> translate env a :: transl_let (Treserved env) L
;;
```

Here come the cases actually performing pattern-matching. The hard work — transforming sets of patterns into simple tests — is done by the `transl_match` function (see below). Its first argument indicates what should be done if the match fails: raising a special exception `match_fail` in case of a function or a `let`, or raising again the trapped exception in case of a `try...with...`. The second argument is a "square" set of patterns and λ-expressions:

$$\begin{array}{ccccccc} p_{11} & p_{12} & \cdots & p_{1n} & \rightarrow & e_1 \\ \vdots & \vdots & \ddots & \vdots & & \vdots \\ p_{k1} & p_{k2} & \cdots & p_{kn} & \rightarrow & e_k \end{array}$$

It represents the simultaneous matching of local variables number n, $n-1$, ..., 1 (in de Bruijn's notation) against the first, second, ..., n^{th} columns of patterns. (To be pedantic: it evaluates to e_i where i is the smallest j such that for all m, p_{jm} filters the value of variable number m.) It is stored internally as a list of pairs (list of patterns, expression).

Notice that this matching is exactly the one performed by functions (more precisely `fun` rather than `function`), since the n arguments of a curried function have numbers $n, n-1, \ldots, 1$. Therefore, the case for `Zfunction` is straightforward. For `Ztrywith`, we just have to transform a simple matching on but one value (the exception bucket) into the more general form above.

The case of `let` and `letrec` bindings is more delicate. An expression such as:

```
let pat1 = expr1 and pat2 = expr2 and pat3 = expr3 in expr4
```

is basically compiled as:

```
let var1 = expr1 and var2 = expr2 and var3 = expr3 in
  match var1 var2 var3 with
      pat1 pat2 pat3 -> expr4
    | _    _    _    -> raise match_fail
```

This simple let (without matching) corresponds exactly to the semantics of the `Llet` term constructor; its body is the one-line matching on `var1 var2 var3`, which is translated into a λ-term by `transl_match`.

Another subtlety is the way expressions `expr1`, `expr2`, ...are compiled. This is done by the function `transl_let`. What we want to do is evaluate `expr1`, add its value in front of the environment, evaluate `expr2`, add it, and so on. So, after evaluating `expr1`, all de Bruijn indexes are off by one: assume the variable `x` had number 1 before, it has number 2 now. We must take this into account and compile `expr2` in a compilation environment where all bindings are shifted, and that's the aim of the function `Treserved` (a constructor, actually). Hence, `expr1` is compiled in environment `env`, `expr2` in `Treserved env`, `expr3` in `Treserved (Treserved env)`, and so on.

For recursive bindings, we proceed as above, with `Lletrec` instead of `Llet`. This time, expressions `expr1`, `expr2`, `expr3` must be compiled in an environment where the variables introduced by `pat1`, `pat2`, `pat3` are already defined.

We can now explain what the function `transl_match` does: it simply translates the actions (to the right of the arrows) into λ-expressions and transmits this modified matching to the `transl_matching` function, described below. This way, we avoid messy mutual recursion between `translate_expr` and the pattern-matching compiler. Notice that the action e_i must be translated in an environment where the free variables of $p_{1i}, \ldots, p_{ni}$ are defined.

Environment handling

There are two ways to handle variables bound by pattern matching: either the pattern matching code actually puts the values of bound variables into new slots in the environment; or it does not extend the environment, which is assumed to contain the term being matched, and in the right-hand side of the arrow, each reference to a variable is translated in an access in the environment, followed by a sequence of indirections (the *access path*) to extract the value given to this variable. The former way is presumably more efficient, especially with a flat environment structure, but it is slightly more complex to implement, so I chose the latter way for ZINC.

Therefore, we need to associate to each local variable a position in the environment and an occurrence (a sequence of integers $a_1 \ldots a_n$ representing the ${a_n}^{\text{th}}$ son of the ${a_{n-1}}^{\text{th}}$ son of ...of the ${a_1}^{\text{th}}$ son of the root found in the environment). However, as positions in the environment are shifted when a new binding is performed, we won't record them as is.

Compilation environments have the following representation:

```
type transl_env =
    Tnullenv
  | Treserved of transl_env
  | Tenv of (string & num list) list & transl_env
```

A `Tenv` node represents the introduction of a new value in front of an existing environment. In this slot, we put the list of local variables that refer to subterms of this value, associated to the corresponding occurrences (the integer list $[a_n; \ldots; a_1]$ representing the occurrence $a_1 \ldots a_n$). A `Treserved` node simply reserves one slot in front of the environment without declaring local variables; `Treserved E` is equivalent to `Tenv([],E}`. It is only used to compile `let` bindings, as described above.

The function `search_env` retrieves the de Bruijn's number and access path of a variable:

```
let search_env s = search_rec 0 where rec search_rec i = function
    Tnullenv      -> zinc_error "search_env : unknown local"
  | Treserved env -> search_rec (i+1) env
  | Tenv(L,env)   -> try (i, assoc s L)
                     with failure "find" -> search_rec (i+1) env
```

Using this function, it's easy to generate the term accessing or modifying the value of a local variable:

```
let translate_access s env =
  let (i,path) = search_env s env in
    list_it (fun n lambda -> Lprim(Pfield n, [lambda])) path (Lvar i)

and translate_update s env lambda =
  match search_env s env with
      (i, n::path) ->
        Lprim(Psetfield n,
              [list_it (fun m lambda -> Lprim(Pfield m, [lambda]))
                       path (Lvar i);
               lambda])
    | _ -> zinc_error "translate_update : not mutable"
;;
```

Keeping track of bindings is easy, since there is but one binding construct, multiple pattern matching:

$$\begin{array}{ccccccc} p_{11} & p_{12} & \cdots & p_{1n} & \rightarrow & e_1 \\ \vdots & \vdots & \ddots & \vdots & & \vdots \\ p_{k1} & p_{k2} & \cdots & p_{kn} & \rightarrow & e_k \end{array}$$

The action e_i must be compiled in an environment extended by the definitions of the free variables of p_{i1}, ..., p_{in}. First, we compute the occurrences of all variables of a given pattern:

```
let rec paths_of_pat path = function
    Zvarpat s ->
      [s, path]
  | Zaliaspat(pat,s) ->
      (s, path) :: paths_of_pat path pat
  | Zconstructpat(_, patlist) ->
      let rec paths_of_patlist i = function
           []    -> []
        | p::pl -> paths_of_pat (i::path) p @ paths_of_patlist (i+1) pl
      in paths_of_patlist 0 patlist
  | Zconstraintpat(pat,_) ->
      paths_of_pat path pat
  | _ -> []
```

Then, the extension of environment E used to compile e_i is simply

$$\texttt{Tenv(paths\_of\_pat}\ [\,]\ p_{in}, \ldots \texttt{Tenv(paths\_of\_pat}\ [\,]\ p_{i1}, E) \ldots)$$

since patterns p_{i1}, …, p_{in} correspond to variables number n, …, 1. More concisely, we write in ML:

```
let make_env = it_list (fun env pat -> Tenv(paths_of_pat [] pat, env))
```

5.2.4 Compiling pattern matching

Compiling pattern matching — that is transforming a pattern matching into a sequence of elementary `if...then...else` tests — is not easy, hence I tried to keep this part of the compiler as simple as possible. In particular, I imposed a left-to-right walk of the term being matched. Therefore, the sequence of tests produced is not necessarily optimal: some tests may be performed while they were not strictly necessary to recognize one of the cases. More sophisticated methods choose the ordering of tests in order to produce an optimal sequence of tests whenever possible, for instance the one of Laville [39, 40], or the one of Puel and Suárez [51]. This problem is crucial in the framework of lazy evaluation, since unnecessary tests may trigger the evaluation of parts of the arguments, which may not terminate; therefore, in some cases, non-optimal pattern matching may fail to terminate, while optimal pattern matching would have succeeded. With strict semantics, this cannot occur: an optimal pattern matching is faster, but has the same semantics as a non-optimal one.

In the following, we call *matching* a matrix of patterns, with one additional row of `lambda` expressions, the *actions*, and one additional line of `lambda` expressions, the *access paths*:

$$\begin{pmatrix} path_1 & \ldots & path_k & & \\ \downarrow & & \downarrow & & \\ p_{11} & \ldots & p_{1k} & \rightarrow & act_1 \\ \vdots & \ddots & \vdots & & \vdots \\ p_{n1} & \ldots & p_{nk} & \rightarrow & act_n \end{pmatrix}$$

The patterns belong to the concrete type `pattern`, but actually we perform some identifications between patterns. For instance, variable names are meaningless (ML patterns are linear), so we identify variable patterns `Zvarpat` and "wildcard" `Zwildpat`, and we write Ω for all these patterns. Also, aliases $\texttt{Zaliaspat}(p, s)$, that is p `as` s, are identified with p. Type constraints `Zconstraintpat` disappear similarly. Finally, for alternatives $(p_1 \mid p_2)$, we shall use the following transformation:

$$\begin{pmatrix} path_1 & \ldots & path_k & & \\ \downarrow & & \downarrow & & \\ \vdots & \ddots & \vdots & & \vdots \\ (p_1|p'_1) & \ldots & p_k & \rightarrow & act_i \\ \vdots & \ddots & \vdots & & \vdots \end{pmatrix} \longmapsto \begin{pmatrix} path_1 & \ldots & path_k & & \\ \downarrow & & \downarrow & & \\ \vdots & \ddots & \vdots & & \vdots \\ p_1 & \ldots & p_k & \rightarrow & act_i \\ p'_1 & \ldots & p_k & \rightarrow & act_i \\ \vdots & \ddots & \vdots & & \vdots \end{pmatrix}$$

Matchings are represented by the following ML concrete type:

```
type pattern_matching = Matching of (pattern list & lambda) list & lambda list
```

Let me define a few trivial functions manipulating this representation. The first simply adds one line to a matching:

```
let add_to_match (Matching(casel,pathl)) cas =
  Matching(cas :: casel, pathl)
```

The next two build matchings from a list of patterns and an action, by adding the right list of access paths:

```
let make_constant_match (path :: pathl) cas =
  Matching([cas], pathl)

and make_construct_match numargs (path :: pathl) cas =
  Matching([cas], make_path 0)
  where rec make_path i =
    if i >= numargs then pathl else Lprim(Pfield i, [path]) :: make_path (i+1)
```

In other terms, `make_constant_match` extracts one line from a matching, discards the leftmost pattern (assumed to be a leaf, e.g. a constant pattern), and builds the corresponding matching:

$$\begin{pmatrix} path_1 & \dots & path_k & & \\ \downarrow & & \downarrow & & \\ \vdots & \ddots & \vdots & & \vdots \\ p_1 & \dots & p_k & \rightarrow & act_i \\ \vdots & \ddots & \vdots & & \vdots \end{pmatrix} \longmapsto \begin{pmatrix} path_2 & \dots & path_k & & \\ \downarrow & & \downarrow & & \\ p_2 & \dots & p_k & \rightarrow & act_i \end{pmatrix}$$

Similarly, `make_construct_match` correspond to the "flattening" of a pattern with a constructor:

$$\begin{pmatrix} path_1 & \dots & path_k & & \\ \downarrow & & \downarrow & & \\ \vdots & \ddots & \vdots & & \vdots \\ C(q_1, \dots, q_m) & \dots & p_k & \rightarrow act_i & \\ \vdots & \ddots & \vdots & & \vdots \end{pmatrix} \longmapsto$$

$$\begin{pmatrix} path_1;[0] & \dots & path_1;[m-1] & path_2 & \dots & path_k & \\ \downarrow & & \downarrow & & & \downarrow & \\ q_1 & \dots & q_m & p_2 & \dots & p_k & \rightarrow act_i \end{pmatrix}$$

where we write $path_1;[i]$ for `Lprim`(`Pfield` $i, [path_1]$).

The pattern matching compiler is of course a divide-and-conquer algorithm. Let us try to divide a matching into smaller matchings first. We distinguish two cases: whether the leftmost-topmost pattern is a variable or not. If it is a variable, we perform the following transformation:

$$\begin{pmatrix} path_1 & path_2 & \dots & path_k & & \\ \downarrow & \downarrow & & \downarrow & & \\ \Omega & p_{12} & \dots & p_{1k} & \rightarrow & a_1 \\ \vdots & \vdots & \ddots & \vdots & & \vdots \\ \Omega & p_{n2} & \dots & p_{nk} & \rightarrow & a_n \\ q_{11} \neq \Omega & q_{12} & \dots & q_{1k} & \rightarrow & b_1 \\ \vdots & \vdots & \ddots & \vdots & & \vdots \\ q_{m1} & q_{m2} & \dots & q_{mk} & \rightarrow & b_m \end{pmatrix} \longmapsto$$

$$\begin{pmatrix} path_2 & \dots & path_k & & \\ \downarrow & & \downarrow & & \\ p_{12} & \dots & p_{1k} & \rightarrow & a_1 \\ \vdots & \ddots & \vdots & & \vdots \\ p_{n2} & \dots & p_{nk} & \rightarrow & a_n \end{pmatrix} \qquad \begin{pmatrix} path_1 & path_2 & \dots & path_k & & \\ \downarrow & \downarrow & & \downarrow & & \\ q_{11} & q_{12} & \dots & q_{1k} & \rightarrow & b_1 \\ \vdots & \vdots & \ddots & \vdots & & \vdots \\ q_{m1} & q_{m2} & \dots & q_{mk} & \rightarrow & b_m \end{pmatrix}$$

The corresponding ML function is as follows:

```
let split_matching (Matching(casel, (_ :: endpathl as pathl))) =
  split_rec casel where rec split_rec = function
      ((Zwildpat | Zvarpat _) :: patl, action) :: rest ->
        let vars, others = split_rec rest in
          add_to_match vars (patl, action),
          others
    | (Zaliaspat(pat,v) :: patl, action) :: rest ->
        split_rec ((pat::patl, action) :: rest)
    | (Zconstraintpat(pat,ty) :: patl, action) :: rest ->
        split_rec ((pat::patl, action) :: rest)
    | (Zorpat(pat1, pat2) :: patl, action) :: rest ->
        split_rec ((pat1::patl, action) :: (pat2::patl, action) :: rest)
    | casel ->
        Matching([], endpathl), Matching(casel, pathl)
```

Let us now consider the case where the leftmost-topmost pattern is not a variable. As in the previous case, we split the matching horizontally just before the first variable in the left column:

$$\begin{pmatrix} path_1 & \dots & path_k & & \\ \downarrow & & \downarrow & & \\ p_{11} \neq \Omega & \dots & p_{1,k-1} & \rightarrow & a_1 \\ \vdots & \ddots & \vdots & & \vdots \\ p_{n1} \neq \Omega & \dots & p_{n,k-1} & \rightarrow & a_n \\ \Omega & \dots & q_{1k} & \rightarrow & b_1 \\ \vdots & \ddots & \vdots & & \vdots \\ q_{m1} & \dots & q_{mk} & \rightarrow & b_m \end{pmatrix} \longmapsto$$

$$\left(\begin{array}{ccccc} path_1 & \dots & path_k & & \\ \downarrow & & \downarrow & & \\ p_{11} & \dots & p_{1,k} & \rightarrow & a_1 \\ \vdots & \ddots & \vdots & & \vdots \\ p_{n1} & \dots & p_{n,k} & \rightarrow & a_n \end{array}\right) \qquad \left(\begin{array}{cccccc} path_1 & path_2 & \dots & path_k & & \\ \downarrow & \downarrow & & \downarrow & & \\ \Omega & & \dots & q_{1k} & \rightarrow & b_1 \\ \vdots & & \ddots & \vdots & & \vdots \\ q_{m1} & & \dots & q_{mk} & \rightarrow & b_m \end{array}\right)$$

Then, in the first matching thus obtained (the one without variables in the first row), we group together the lines whose leftmost pattern have the same toplevel symbol (i.e. either constant patterns with the same constant, or constructor patterns with the same constructor), and build sub-matchings with them. For instance:

$$\left(\begin{array}{ccccc} path_1 & path_2 & \dots & path_n & \\ \downarrow & \downarrow & & \downarrow & \\ 1 & p_2 & \dots & p_n & \rightarrow a \\ 2 & q_2 & \dots & q_n & \rightarrow b \\ 1 & r_2 & \dots & r_n & \rightarrow c \end{array}\right) \longmapsto$$

$$1, \left(\begin{array}{cccc} path_2 & \dots & path_n & \\ \downarrow & & \downarrow & \\ p_2 & \dots & p_n & \rightarrow a \\ r_2 & \dots & r_n & \rightarrow c \end{array}\right) \qquad 2, \left(\begin{array}{cccc} path_2 & \dots & path_n & \\ \downarrow & & \downarrow & \\ q_2 & \dots & q_n & \rightarrow b \end{array}\right)$$

For constructor patterns, we also add the arguments of the constructors to the resulting matchings:

$$\left(\begin{array}{ccccc} path_1 & path_2 & \dots & path_n & \\ \downarrow & \downarrow & & \downarrow & \\ C(p', p'') & p_2 & \dots & p_n & \rightarrow a \\ D(q') & q_2 & \dots & q_n & \rightarrow b \\ C(r', r'') & r_2 & \dots & r_n & \rightarrow c \end{array}\right) \longmapsto$$

$$C, \left(\begin{array}{cccccc} path_1;[0] & path_1;[1] & path_2 & \dots & path_n & \\ \downarrow & \downarrow & \downarrow & & \downarrow & \\ p' & p'' & p_2 & \dots & p_n & \rightarrow a \\ r' & r'' & r_2 & \dots & r_n & \rightarrow c \end{array}\right) \qquad D, \left(\begin{array}{ccccc} path_1;[0] & path_2 & \dots & path_n & \\ \downarrow & \downarrow & & \downarrow & \\ q' & q_2 & \dots & q_n & \rightarrow b \end{array}\right)$$

To summarize:

```
let divide_matching (Matching(casel, (_ :: tailpathl as pathl))) =
  divide_rec casel where rec divide_rec = function
      [] ->
        [], [], Matching([], pathl)
    | ([],_) :: _ ->
        zinc_error "divide_matching"
    | (Zaliaspat(pat,v) :: patl, action) :: rest ->
        divide_rec ((pat::patl, action) :: rest)
    | (Zconstraintpat(pat,ty) :: patl, action) :: rest ->
        divide_rec ((pat::patl, action) :: rest)
    | (Zorpat(pat1, pat2) :: patl, action) :: rest ->
        divide_rec ((pat1::patl, action) :: (pat2::patl, action) :: rest)
```

```
  | (Zconstantpat(cst) :: patl, action) :: rest ->
      let (constant, others) = divide_rec rest in
        add_to_division (make_constant_match pathl) constant
                  cst (patl, action),
        others
  | (Zconstructpat(c,args) :: patl, action) :: rest ->
      let (constant, constrs, others) = divide_rec rest in
        constant,
        add_to_division (make_construct_match (length args) pathl) constrs
                        c.desc.tag (args @ patl, action),
        others
  | casel ->
      [], [], Matching(casel, pathl)
```

The `divide_matching` function returns a triple: the last component is the remaining matching, the one that starts by a variable; the first component is a list of pairs of an atomic constant and a reference to a matching; the second component is a list of pairs of a constructor descriptor and a reference to a matching. The auxiliary function `add_to_division` is defined as follows:

```
let add_to_division make_match divlist key cas =
  try
    let matchref = assoc key divlist in
      matchref := add_to_match !matchref cas; divlist
    with failure "find" ->
      (key, ref (make_match cas)) :: divlist
```

Now, we can start to conquer. We need to define the `lambda` term $[m]$ corresponding to a matching m. The empty matching (the one with no lines) always fails:

$$\begin{bmatrix} path_1 & \dots & path_n \\ \downarrow & & \downarrow \end{bmatrix} = \texttt{staticfail}$$

A flat matching (one whose rows have zero width) always succeeds, and executes the action of its first line:

$$\begin{bmatrix} \rightarrow & act_1 \\ \vdots & \vdots \\ \rightarrow & act_n \end{bmatrix} = act_1$$

If the matching p "starts" by a variable (i.e. if its leftmost-topmost pattern is a variable), then, writing p_1, p_2 for the two matchings returned by $\texttt{split\_matching}(p)$:

$$[p] = [p_1]\ \texttt{statichandle}\ [p_2]$$

Indeed, in the upper half of p, the constraint on $path_1$ is always satisfied (variables match everything), so we just have to satisfy the other constraints, the one of p_1; but if it fails, we execute the remaining part of p, that is p_2.

Finally, if the matching m does not start by a variable, we discriminate on the value of $path_1$ to dispatch among the sub-matchings returned by `divide_matching`. For instance, in the case $\texttt{divide\_matching}(m) = ([1, m_1; 2, m_2; 3, m_3], [\,], m_0)$, we have:

$$[p] = (\texttt{case}\ path_1\ \texttt{of}\ 1 : [m_1]; 2 : [m_2]; 3 : [m_3])\ \texttt{statichandle}\ [m_0]$$

Similarly, if $\texttt{divide\_matching}(m) = ([\,], [C, m_1; D, m_2;], [\,], m_0)$, and assuming that C has tag 0 and D has tag 1,

$$[p] = (\texttt{case tag\_of}(path_1)\ \texttt{of}\ 0 : [m_0]; 1 : [m_1])\ \texttt{statichandle}\ [m_0]$$

To summarize:

```
let rec conquer_matching =
  let rec conquer_divided_matching = function
    [] ->
      [], Total, 0
  | (key, matchref) :: rest ->
      let lambda1, partial1, unused1 = conquer_matching !matchref
      and list2,   partial2, unused2 = conquer_divided_matching rest in
        (key, lambda1) :: list2,
        (match (partial1, partial2) with
              (Total,Total) -> Total
            | (Partial, _)  -> Partial
            | (_, Partial)  -> Partial
            |      _        -> Dubious),
        unused1 @ unused2
  in function
    Matching([], _) ->
      Lstaticfail, Partial, 0
  | Matching(([],action) :: rest, _) ->
      action, Total, rest
  | Matching(_, (path :: _)) as matching ->
      if starts_by_a_variable matching then
        let vars, rest = split_matching matching in
        let lambda1, partial1, unused1 = conquer_matching vars
        and lambda2, partial2, unused2 = conquer_matching rest in
          if partial1 = Total then
            lambda1, Total, unused1 @ lines_of_matching rest
          else
    Lstatichandle(lambda1, lambda2),
            (if partial2 = Total then Total else Dubious),
            unused1 @ unused2
      else (match divide_matching matching with
          [], [], vars ->
    conquer_matching vars
        | constants, [], vars ->
            let condlist1,   _   , unused1 = conquer_divided_matching constants
            and lambda2, partial2, unused2 = conquer_matching vars in
              Lstatichandle(Lcond(path, condlist1), lambda2),
              partial2,
              unused1 @ unused2
        | [], constrs, vars ->
            let switchlst, partial1, unused1 = conquer_divided_matching constrs
            and lambda,    partial2, unused2 = conquer_matching vars in
            let span = get_span_of_matching matching in
              if length constrs = span & partial1 = Total then
```

```
              Lswitch(span, path, switchlst),
              Total,
              unused1 @ lines_of_matching vars
            else
              Lstatichandle(Lswitch(span, path, switchlst), lambda),
              (if partial1 = Total then partial2
               if partial2 = Total then Total
               else Dubious),
              unused1 @ unused2
      | _ -> zinc_error "conquer_matching : ill_typed matching")
  | _ -> zinc_error "conquer_matching"
```

The careful reader has noticed that this function does not simply return a `lambda` expression, but also a value of the concrete type `partiality = Partial | Total | Dubious`, and a list of "lines" (that is, a list of pattern and an action). These two additional results are a modest attempt at detecting partial matchings and unused cases. As the case `Dubious` suggests, the "partiality" information is approximated fairly brutally. The problem is that when we combine two partial matchings p_1 and p_2 into p_1 `statichandle` p_2, the resulting matching can be total if the cases of p_1 and p_2 overlap correctly, but this overlapping is very hard to detect. More sophisticated frameworks, such as the constrained terms of Puel and Suárez [51], directly address this issue.

Finally, here is the code that glues the pattern matching compiler to the first pass of the compiler:

```
let translate_matching failure_code casel =
  let rec make_path =
    function 0 -> []
        |    n -> Lvar(n-1) :: make_path(n-1) in
  let (lambda, partial, unused) =
    conquer_matching (Matching(casel, make_path (length (hd casel)))) in
 (match unused with
      []  -> ()
    | [_] -> message "1 unused case in this match."
    | _   -> print_num (length unused); message " unused cases in this match.");
 (match partial with
      Total -> lambda
    |   _   -> Lstatichandle(lambda, failure_code partial))
```

The function `failure_code` passed as argument is responsible for printing a suitable warning and returning the `lambda` expression raising the right exception. For instance, here are the corresponding functions for `function ...` and `try...with...` constructs:

```
let partial_fun par =
  if par = Partial then print_string "Partial function.\n";
  Lprim(Praise, [Lconst(SCblock(ConstrExtensible "match_fail", []))])
and partial_try par =
  Lprim(Praise, [Lvar 0])    (* reraise the trapped exception *)
```

5.2.5 Compiler, back end

The second pass of the compiler transform intermediate λ-terms into linear lists of ZAM code. This pass performs sequentialization of intermediate code, and nothing else: it does not attempt to fix inefficiencies in the given term, by analyzing it, reorganizing evaluations, and so on. However, it goes to great length not to introduce new inefficiencies during sequentialization. For instance, it never generates a jump to a jump, nor a jump to a `return` instruction, and does a good job at compiling complex tests such as `if a=b or c<d then e else f`, which becomes:

```
   if a=b goto 1;
   if c>=d goto 2;
1: e;
   goto 3;
2: f;
3:
```

This pass produces a list of ZAM code. There are two ways to do so: the first is to generate sublists (one for each subterm of the original λ-term) and combine them using the list concatenation operator `@`; the second way is to build the list in one right-to-left pass, by taking as arguments not only the term to compile, but also a list of code, and consing the instructions corresponding to the term in front of that list. The second way is usually preferred, since list concatenation is an expensive operation (even if it can be done by physical modifications). More importantly, it allows the compiler to know the *continuation* of the expression to be compiled (i.e. the code that will be executed after it), and this continuation encodes a lot of useful information, allowing to generate much better code. For instance, to know whether the given expression is in tail-call position, it suffices to look at the continuation and see if it starts by a `Kreturn` instruction:

```
let is_tail = function Kreturn :: _ -> true | _ -> false
```

Similarly, when we are about to generate a jump to the continuation, we first look if it starts by a `Kreturn` or `Kbranch` instruction; in the first case, it suffices to insert a `Kreturn`L instruction; in the second case, it is possible to jump directly to the label L:

```
let make_branch = function
    Kreturn :: _ as C -> Kreturn, C
  | (Kbranch _ as branch) :: _ as C -> branch, C
  | C -> let lbl = new_label() in Kbranch lbl, Klabel lbl :: C
```

In the same vein, the `label_code` function, which puts a label on a piece of code, is written:

```
let label_code = function
    Kbranch lbl :: _ as C -> lbl, C
  | Klabel lbl :: _ as C -> lbl, C
  | C -> let lbl = new_label() in lbl, Klabel lbl :: C
```

The main function, `compile_expr`, takes three arguments. The first is a label, used to compile `Lstaticfail` and `Lstatichandle` instructions. Remember that these are statically-scoped, nullary exception handling, therefore `Lstaticfail` is simply a jump to the handler part of the first enclosing `Lstatichandle`. The first argument of `compile_expr` is precisely the label of this handler. The second argument is a term of the intermediate syntax `lambda`. The third is the continuation. First, we perform a large pattern matching on the term. Then, we may test the continuation to recognize special cases.

```
let rec compile_expr staticfail = compexp
  where rec compexp = fun
    (Lvar n) C ->
        Kaccess n :: C
  | (Lconst cst) C ->
        Kquote cst :: C
  | (Lapply(body, args)) ->
       (function
          Kreturn :: C ->
            compexplist args (Kpush :: compexp body (Ktermapply :: C))
        | C ->
            Kpushmark ::
              compexplist args (Kpush :: compexp body (Kapply :: C)))
```

The `Lapply` case is the first example of matching on the continuation. Here, we detect tail calls, and use the more efficient `Ktermapply` instruction in this case (and we suppress the following `Kreturn`, which will never be executed anyway). The `compexplist` function, defined below, simply generates code for a list of expressions, with `Kpush` instructions between two expressions.

```
  | (Lfunction(n, body)) C ->
        if is_tail C then
          iterate (cons Kgrab) n (compexp body C)
        else
         (let lbl = new_label() in
            add_to_compile (n, body, lbl);
            Kclosure lbl :: C)
```

Similarly, functions in tail position do not need to build a closure, and can be expanded in-line, with the right number of `Kgrab` instructions first, to bind their parameters to their arguments. Otherwise, we emit a `Kclosure` instruction; but we cannot generate code for the body of the abstraction on the fly, since we are in the middle of a block of code, and we have no place to store another block; therefore, we push it on a stack, using `add_to_compile`, and we shall pop it when the current expression is fully compiled (see below).

```
  | (Llet(args, body)) C ->
        let C1 = if is_tail C then C else Kendlet(length args) :: C in
        let C2 = compexp body C1 in
        let rec comp_args = function
               []     -> C2
          | exp::rest -> compexp exp (Klet :: comp_args rest) in
        comp_args args
  | (Lletrec(args, body)) C ->
        let size = length args in
        let C1 = if is_tail C then C else Kendlet size :: C in
        let C2 = compexp body C1 in
        let rec comp_args i = function
               []     -> C2
          | exp::rest -> compexp exp (Kupdate i :: comp_args (i-1) rest) in
        Kdummies size :: comp_args (size-1) args
  | (Lprim(Pget_global qualid, [])) C ->
```

```
      Kget_global qualid :: C
  | (Lprim(Pset_global qualid, [exp])) C ->
      compexp exp (Kset_global qualid :: C)
  | (Lprim(Pmakeblock tag, explist)) C ->
      compexplist explist (Kmakeblock(tag, length explist) :: C)
  | (Lprim(Pnot, [exp])) ->
     (function
        Kbranchif lbl :: C ->
          compexp exp (Kbranchifnot lbl :: C)
      | Kbranchifnot lbl :: C ->
          compexp exp (Kbranchif lbl :: C)
      | C ->
          compexp exp (Kprim Pnot :: C))
  | (Lprim((Ptest tst as p), explist)) ->
     (function
        Kbranchif lbl :: C ->
          compexplist explist (Ktest(tst,lbl) :: C)
      | Kbranchifnot lbl :: C ->
          compexplist explist (Ktest(invert_bool_test tst,lbl) :: C)
      | C ->
          compexplist explist (Kprim p :: C))
  | (Lprim(p, explist)) C ->
      compexplist explist (Kprim p :: C)
```

Boolean primitives, such as `not` and the comparisons, are treated specially here. Sometimes, their actual value is not needed, all we want to do is to perform a conditional branch on their result. For instance, a `not` operation followed by a conditional branch need not be computed, it suffices to invert the conditional branch. Similarly, when a comparison is followed by a conditional branch, we can generate the `Ktest` instruction, which is a conditional branch on that condition, instead of computing the boolean value and jump if it is true or false. As its name implies, the `invert_bool_test` function maps `Pequal_test` to `Pnotequal_test`, `Pint_test(PTlt)` to `Pint_test(PTge)`, and so on.

```
  | (Lstatichandle(body, Lstaticfail)) C ->
      compexp body C
  | (Lstatichandle(body, handler)) C ->
      let branch1, C1 = make_branch C
      and lbl2 = new_label() in
        compile_expr lbl2 body (branch1 :: Klabel lbl2 :: compexp handler C1)
  | (Lstaticfail) C ->
      Kbranch staticfail :: C
```

The `Lstatichandle` does not generate any code *per se*, it simply changes the value of the current static handler label, the parameter `staticfail`. The "static raise" `Lstaticfail` is a mere jump.

```
  | (Lhandle(body, handler)) C ->
      let branch1, C1 = make_branch C in
      let lbl2 = new_label() in
      let C2 = if is_tail C1 then C1 else Kendlet 1 :: C1 in
        Kpushtrap lbl2 ::
```

```
          compexp body
                  (Kpoptrap :: branch1 :: Klabel lbl2 :: compexp handler C2)
  | (Lifthenelse(cond, ifso, ifnot)) C ->
        comp_test2 cond ifso ifnot C
  | (Lsequence(exp1, exp2)) C ->
        compexp exp1 (compexp exp2 C)
  | (Lwhile(cond, body)) C ->
        let lbl1 = new_label() and lbl2 = new_label() in
          Kbranch lbl1 :: Klabel lbl2 ::
            compexp body (Klabel lbl1 :: compexp cond (Kbranchif lbl2 :: C))
  | (Lsequand(exp1, exp2)) ->
       (function
          Kbranch lbl :: _ as C ->
            compexp exp1 (Kbranchifnot lbl :: compexp exp2 C)
        | Kbranchifnot lbl :: _ as C ->
            compexp exp1 (Kbranchifnot lbl :: compexp exp2 C)
        | Kbranchif lbl :: C ->
            let lbl1, C1 = label_code C in
              compexp exp1 (Kbranchifnot lbl1 ::
                              compexp exp2 (Kbranchif lbl :: C1))
        | C ->
            let lbl = new_label() in
              compexp exp1 (Kstrictbranchifnot lbl ::
                              compexp exp2 (Klabel lbl :: C)))
  | (Lsequor(exp1, exp2)) ->
       (function
          Kbranch lbl :: _ as C ->
            compexp exp1 (Kbranchif lbl :: compexp exp2 C)
        | Kbranchif lbl :: _ as C ->
            compexp exp1 (Kbranchif lbl :: compexp exp2 C)
        | Kbranchifnot lbl :: C ->
            let lbl1, C1 = label_code C in
              compexp exp1 (Kbranchif lbl1 ::
                              compexp exp2 (Kbranchifnot lbl :: C1))
        | C ->
            let lbl = new_label() in
              compexp exp1 (Kstrictbranchif lbl ::
                              compexp exp2 (Klabel lbl :: C)))
```

What we said about `not` and the comparisons also holds for the left-to-right `and` and `or` constructs: when they are followed by a conditional jump, it is possible to "short-circuit" this jump (this amounts to transforming `if a and b then c else d` into `if a then if b then c else d else d`, without duplicating the code for `d`, however). Otherwise, when the actual boolean value of `a and b` is needed, we compile it as `if a then b else a`, without re-evaluating `a`, of course, that is we evaluate `a`, jump to the continuation if it is false, evaluate `b`, and continue. A subtle bug arises here: in the discussion above, we have assumed that when an expression is followed by a conditional branch, then its actual value is not used. This is not the case here: we test the value of `a`, but if it's false, then it is also the value of the whole `a and b` expression, so the boolean `false` must be in the accumulator. For instance, assume we compile `(a=b) or c` as follows:

```
<code for a=b>; Kbranchifnot 100; <code for c>; Klabel 100; ...
```

Then the compilation of `a=b` will be "optimized", getting:

```
<code for a>; Kpush; <code for b>; Ktest(Pnotequal_test,100);
<code for c>; Klabel 100; ...
```

and if $\mathtt{a} \neq \mathtt{b}$, we reach label 100 with garbage in the accumulator, instead of `false`, as expected. To fix this, I had to introduce two variants of `Kbranchif` and `Kbranchifnot`, `Kstrictbranchif` and `Kstrictbranchifnot`, that have exactly the same semantics, but prohibit all the optimizations above from being applied. Therefore, `(a=b) or c` is compiled as

```
<code for a=b>; Kstrictbranchifnot 100; <code for c>; Klabel 100; ...
```

and the equality test will not be optimized:

```
<code for a>; Kpush; <code for b>; Kprim(Ptest(Pnotequal_test));
Kstrictbranchifnot 100; <code for c>; Klabel 100; ...
```

Let us go back to the main function.

```
  | (Lcond(arg, casel)) C ->
        let branch1, C1 = make_branch C in
        let rec comp_tests = function
            [] ->
              zinc_error "compile_exp (cond)"
          | [a,exp] ->
              Ktest(test_for_atom a, staticfail) :: compexp exp C1
          | (a,exp)::rest ->
              let lbl = new_label() in
                Ktest(test_for_atom a, lbl) ::
                compexp exp (branch1 :: Klabel lbl :: comp_tests rest) in
        compexp arg (comp_tests casel)
  | (Lswitch(1, arg, [ConstrRegular _, exp])) C ->
        compexp exp C
  | (Lswitch(2, arg, [ConstrRegular(0,_), exp0;
                      ConstrRegular(1,_), exp1])) C ->
        comp_test2 arg exp1 exp0 C
  | (Lswitch(2, arg, [ConstrRegular(1,_), exp1;
                      ConstrRegular(0,_), exp0])) C ->
        comp_test2 arg exp1 exp0 C
  | (Lswitch(size, arg, casel)) C ->
        let branch1, C1 = make_branch C in
        if switch_through_jumptable size casel then
          let switchtable = vector size of staticfail in
          let rec comp_case = function
              [] ->
                zinc_error "compile_exp (comp_case)" in
            | [ConstrRegular(i,_), exp] ->
                let lbl = new_label() in
                  switchtable.(i) <- lbl;
                  Klabel lbl :: compexp exp C1
            | (ConstrRegular(i,_), exp) :: rest ->
```

```
              let lbl = new_label() in
                switchtable.(i) <- lbl;
                Klabel lbl :: compexp exp (branch1 :: comp_case rest)
        in compexp arg (Kswitch switchtable :: comp_case casel)
      else
        let rec comp_tests = function
            [] ->
              zinc_error "compile_exp (switch)"
          | [tag,exp] ->
              Ktest(Pnoteqtag_test tag, staticfail) :: compexp exp C1
          | (tag,exp)::rest ->
              let lbl = new_label() in
                Ktest(Pnoteqtag_test tag, lbl) ::
                compexp exp (branch1 :: Klabel lbl :: comp_tests rest)
        in compexp arg (comp_tests casel)
```

The last cases deal with the multi-way branches `Lcond` and `Lswitch`. A `Lcond` construct is simply transformed into a sequence of elementary comparisons against constants; no attempt is done yet to try to use jumps through tables in case of "dense" matchings with integers or characters. For a `Lswitch` construct, which discriminates on the tag of its argument, we trap first the cases where there are only one or two possible values for the tag (as guaranteed by typing); if there is but one, we discard the test entirely (assuming the argument of the `Lswitch` is a pure expression, which is always the case, since this construct is generated by the pattern-matching compiler only); if there are two possible tags, we use a simple `if ... then ... else` test. Otherwise, we are left to choose between generating an indexed branch, or a sequence of simple tests, as above. The choice is performed by the function `switch_through_jumptable`. There is one important case where we cannot generate an indexed branch: when we perform matching on values of an open sum. In this case, the tag numbers of the constructors are not known at compile-time (they are still represented by their name, through the `ConstrExtensible` constructor), so we cannot build a jump table. The other case where a sequence of simple tests is preferred is when the matching is "sparse", that is when at least 80 % of the cases fall into the default action, which is implicitly `Lstaticfail`, and there are less than 10 cases not defaulting to `Lstaticfail`.

```
and compexplist = fun
    [] C -> C
  | [exp] C -> compexp exp C
  | (exp::rest) C -> compexplist rest (Kpush :: compexp exp C)

and comp_test2 cond ifso ifnot C =
  let branch1, C1 = make_branch C
  and lbl2 = new_label() in
    compexp cond (Kbranchifnot lbl2 ::
              compexp ifso (branch1 :: Klabel lbl2 :: compexp ifnot C1))
```

As mentioned previously, the `compile_expr` function does not compile the body of the abstractions it encounters (by the way, where would it put the corresponding code ?), but save these abstractions in a stack. It is time to describe this little machinery. The stack is maintained by three functions sharing it as a local variable:

```
let reset_to_compile, add_to_compile, get_to_compile =
  let still_to_compile = ref ([] : (num & lambda & label) list) in
    (fun () -> still_to_compile := []; ()),
    (fun bc -> still_to_compile := bc :: !still_to_compile; ()),
    (fun () -> match !still_to_compile with
                   []     -> failwith "finished"
                 | bc::rest -> still_to_compile := rest; bc)
```

To compile entirely a `lambda` expression, the procedure is as follows: apply `compile_expr` on it; while the stack of abstractions is not empty, pop an expression from it, compile it, and add its code to the code already generated. In our "continuation" style, this translates to:

```
let compile_lambda =
  let rec compile_rest C =
    try
      match get_to_compile() with
        (1,exp,lbl) ->
           compile_rest (Klabel lbl :: Kstartfun ::
                         compile_expr Nolabel exp (Kreturn :: C))
      | (n,exp,lbl) ->
           compile_rest (Klabel lbl :: Kstartfun ::
                         iterate (cons Kgrab) (n-1)
                                 (compile_expr Nolabel exp (Kreturn :: C)))
    with failure "finished" ->
      C in
  fun expr ->
    reset_to_compile(); reset_label();
    (compile_expr Nolabel expr [Kbranch(Label 0)], compile_rest [])
```

Notice that we actually return *two* lists of code, one for the original `lambda` expression, the other for all the closures it uses. The idea is that the first piece of code will be executed only once (since it is not part of a closure!), so the runtime system can throw it away just after, while the second piece of code really needs to stay in memory, since some closures may refer to it.

For each toplevel phrase, the pair of instruction lists obtained at the end of the compilation line is written to the object code file, using the `extern` facility for structured data output (see [62, chap. 9] for a presentation, and [1] for the problem of its integration into a statically-typed language such as ML).

5.2.6 Global environment

The discussion above focuses on the compilation of single expressions; it is now time to see how global declarations are handled.

A global environment comprises three independent name spaces, one for global variables ("values"), one for constructors, and one for types. These name spaces have a regular structure: for each global identifier, they hold its fully qualified name (name of defining module + name inside this module), and associated informations.

```
type 'a global =
  { modname : string;
```

```
    name : string;
    desc : 'a }
```

For a global value, these informations are its type, and whether it is a primitive operation or not; if it is, we record here the corresponding primitive, and its arity.

```
type value_global == value_desc global
and value_desc =
  { valtype: type_expr;
    prim:    prim_desc
  }
and prim_desc =
    ValueNotPrim
  | ValuePrim of num * primitive
```

For a constructor, we store the concrete type it belongs to, the type of its arguments (and for each argument, whether it is mutable or not), the tag given to the constructor, and a flag telling if it has some mutable arguments or not (used for allowing structured constant propagation).

```
type constr_global == constr_desc global
and constr_desc =
  { result_type: type_expr;
    arg_types  : (bool * type_expr) list;
    tag        : constr_tag;
    pure       : bool
  }
```

Finally, assuming all concrete types are sums of products, the informations associated to a type identifier are its arity, and the list of its constructors:

```
type type_global == type_desc global
and type_desc =
  { arity  : num;
    constrs: constr_desc list
  }
```

To make symbol table lookup faster, I use hashing to associate a description to the name of a global. Actually, this is a general-purpose implementation of hashed association tables, fully polymorphic, thanks to the excellent "universal" (i.e. polymorphic) hash function provided in the CAML library [62, pp. 343–345]. It uses dynamic hashing, and doubles the size of the table when a bucket becomes too long, but without hashing again the whole table. This ensures average access time logarithmic in the number of elements in the table.

The global environment of a module is therefore a triple of hashed tables, associating value descriptors, constructor descriptors, and type descriptors to global names:

```
type module_desc =
  { name : string;
    values : (string, value_desc) hashtable;
    constrs : (string, constr_desc) hashtable;
    types : (string, type_desc) hashtable
  }
```

Compiled interfaces or modules are also represented that way; of course, they just contain the globals that are actually exported.

Then, the global state of the compiler comprises the description of the module being compiled, as well as a list of description of the modules currently "opened", that is those modules that are searched when disambiguating non-qualified global names (as described in section 2.1.3).

```
type global_environment =
  { defined_module: module_desc;
    mutable opened_modules: module_desc }
```

Global table lookup is performed as soon as a global reference was recognized: at parsing for constructors and type identifiers, during the typechecking for value identifiers. Their descriptors are put directly in the syntax tree. Here are the functions that actually fetch the descriptor, given a "global reference", which may be already qualified or not:

```
type global_reference =
    GRname of string
  | GRmodname of string & string
;;
let find_value_desc, find_constr_desc, find_type_desc =
  let find_desc extract =
    let find_with_name (md : module) s =
      let desc = find_in_hashtable (extract md) s in
        { modname = md.name; name = s; desc = desc } in
    function GRname s ->
        (let gl = current_globalenv() in
          try
            find_with_name gl.defined_module s
          with failure "find" ->
            find_rec gl.used_modules where rec find_rec = function
              []       -> failwith "find"
            | [md]     -> find_with_name md s
            | md::rest -> try find_with_name md s
                          with failure "find" -> find_rec rest)
    | GRmodname(modname,s) ->
        (let gl = current_globalenv() in
          if modname = gl.defined_module.name then
            find_with_name gl.defined_module s
          else
            find_rec gl.used_modules where rec find_rec = function
              []       -> failwith "find"
            | md::rest -> if md.name = modname then
                            try find_with_name md s
                            with failure "find" -> find_rec rest
                          else
                            find_rec rest) in
  find_desc (fun x -> x.values),
  find_desc (fun x -> x.constrs),
  find_desc (fun x -> x.types)
;;
```

Chapter 6

The linker and the runtime system

This chapter presents the remaining two parts of ZINC: the runtime system, which executes bytecode, and the linker, which produces executable bytecode from the object code files produced by the compiler. The former is written in C, the latter in ML.

6.1 The complete instruction set

It is time to give the exact instruction set of the bytecode interpreter, with the format of their operands. The following notations will be used: identifiers such as Grab are operation codes, they take up one byte in the case of bytecode, and two or four in the case of threaded code. The operands of an instruction are written in braces, as in Cur(*ofs*); operands are stored just after the opcode, in the given order. The following kinds of operands are used:

n	a small integer (the size of an opcode)
ofs	an offset for a relative branch, relative to the address where it is stored; it uses two bytes
tag	the tag of a block (one byte)
header	a well-formed block header (four bytes)
int_8	a small integer constant (one byte)
int_{16}	a medium integer constant (two bytes)
int_{32}	a large integer constant (four bytes)
float	a floating-point number (four, eight or ten bytes, depending on the hardware)
string	a character string, stored as if it was in the heap (section 4.3.1).

Notice that the format of executable bytecode depends on the interpretation method used (bytecode and threaded code), and also on the host machine. For instance, integer and floating-point constants are stored using the format of the host machine; in particular, the endianness of the host processor must be respected. Also, some processors may impose alignment constraints on the operands: a 16-bit operand may have to be aligned on a 16-bit boundary. This must be ensured by padding the bytecode with Nop operations, if necessary.

Constants and literals

Constbyte(int_8), Constshort(int_{16}), Constlong(int_{32})	Put an integer constant in the accumulator. Constlong allows loading any constant, as long as it is not a pointer in the heap.
Atom(n), Atom0, ..., Atom9	Put a pointer to a zero-sized block tagged n in the accumulator.
GetGlobal(int_{16}), SetGlobal(int_{16})	Load (resp. store) the accumulator from the global variable number int_{16}

Function handling

Push, Pushmark	Push the accumulator (resp. a mark) on the argument stack
Apply, Appterm	Call (resp. jump to) the closure contained in the accumulator
Return	If there is a mark on top of the argument stack, pop it and return to the caller; otherwise, jump to the closure contained in the accumulator
Grab	Pop one value on the argument stack and put it in the environment; if the top of stack is a mark, build a closure and return it to the caller
Cur(ofs)	Build the closure of ofs with the current environment, and put it in the accumulator

Environment handling

Access(n), Access0, Access1, Access2, Access3, Access4, Access5	Fetch the n^{th} slot of the environment, and put it in the accumulator
Let	Put the value of the accumulator in front of the environment
Endlet(n), Endlet1	Throw away the first n local variables from the environment
Dummies(n)	Put n dummy closures in front of the environment
Update(n)	Physically update the n^{th} slot of the environment with the value of the accumulator
Letrec1(ofs)	Same as Dummies(1); Closure(ofs); Update(0) (a very frequent sequence, corresponding to `let rec f = function ... in ...`

Building and destructuring blocks

Makeblock(*header*), Makeblock1(*tag*), Makeblock2(*tag*), Makeblock3(*tag*), Makeblock4(*tag*)	Allocate a block with given header, initialize field 0 with the accumulator, and the remaining fields with values taken from the argument stack
Getfield(n), Getfield0, Getfield1, Getfield2, Getfield3	Access the n^{th} field of the block pointed to by the accumulator
Setfield(n), Setfield0, Setfield1, Setfield2, Setfield3	Physically replace the n^{th} field of the block pointed to by the accumulator with the value popped from the argument stack

Integers

SuccInt, PredInt, NegInt, AddInt, SubInt, MulInt, DivInt, ModInt, AndInt, OrInt, XorInt, ShiftLeftInt, ShiftRightInt	Usual arithmetic operations on integers

Floating-point numbers

Floatop(n)	Allocates room for one floating result, and executes the sub-instruction n, one of AddFloat, SubFloat, MulFloat, DivFloat, and the usual transcendental functions
FloatOfInt, IntOfFloat	Conversion from an integer, and truncation to an integer

Strings

Makestring	Allocates a string of given length (in the accumulator)
StringLength	Length of the string contained in the accumulator
GetChar, SetChar	Read or modify one char in a string
FillString, BlitString	Fill a substring with a given character, or copy one substring into another

Predicates

Boolnot	Negation: returns "true" (the zero-sized block tagged 1) if the block in the accumulator is tagged 0, and "false" (the zero-sized block tagged 0) otherwise

Eq, Equal	Pointer equality (resp. structural equality) between the accumulator and the top of stack
EqInt, NeqInt, LtInt, GtInt, LeInt, GeInt	Usual comparison predicates on integers
EqFloat, NeqFloat, LtFloat, GtFloat, LeFloat, GeFloat	Usual comparison predicates on floating-point numbers
EqString, NeqString, LtString, GtString, LeString, GeString	Usual comparison predicates on strings

Branches and conditional branches

Branch(ofs)	Unconditional relative jump
Branchif(ofs), Branchifnot(ofs), Branchifeqtag(tag, ofs), Branchifneqtag(tag, ofs)	Conditional branches on the tag t of the block pointed to by the accumulator: Branchif jumps if $t \neq 0$, Branchifnot jumps if $t = 0$, Branchifeqtag jumps if $t = tag$, Branchifneqtag jumps if $t \neq tag$
Switch($ofs_0, \ldots, ofs_k$)	Jumps to the offset ofs_t, where t is the tag t of the block contained in the accumulator
BranchifEq(ofs), BranchifNeq(ofs), BranchifEqual(ofs), BranchifNequal(ofs), BranchifLtInt(ofs), ..., BranchifGeString(ofs)	Conditional branches corresponding to the binary predicates above
BranchifNeqImmInt(int_{32}, ofs), BranchifNeqImmFloat($float, ofs$), BranchifNeqImmString($string, ofs$)	Compare the accumulator with the constant given as argument, and jumps if different. (This gadget is useful for fast pattern matching.)

Miscellaneous

CCall0(n), ..., CCall5(n)	Call a C function, with 0 to 5 arguments. C functions are put in a special table; n is the number of the desired function. The first argument is the value if the accumulator, the remaining arguments are popped from the argument stack. The result is put in the accumulator.
StartFun	Perform various checks such as stack overflow, pending break condition, and so on. Intended to be inserted at the beginning of each function and loop body.

Nop1, Nop2, Nop3	Do nothing, but skip respectively one, two, and three bytes. Used to align code on 16-bit or 32-bit boundaries.

6.2 The linker

The linker gathers together the object code files produced by the compiler, and produces a file of bytecode (or threaded code), directly executable by the runtime system. The linker itself is written in ML. It performs the following transformations:

- lists of ML-represented instructions (the concrete type `instruction` of section 5.1.4) are transformed into a stream of opcodes and operands, with the format described above.
- labels and references to labels are replaced by offsets. This uses a classical one-pass algorithm, with backpatching for forward references.
- global variables, which are still represented by their qualified name, are given a slot in the data space of the executable, and global references are represented in the code by the address of their slot. Slots are also reserved to hold structured constants, since the garbage collector prevents them from being put directly in the code.
- constructors belonging to extensible sums (for the time being, only exceptions are in this case) are given a tag number. This cannot be done at compile-time, since two modules can extend the same type in independent directions.
- constants are converted into the format expected by the host processor (e.g. big-endian or little-endian in the case of integers). Similarly, Nop instructions are inserted to align operands on 16-bit or 32-bit boundaries, if the host requires it.
- initialization code sequences for each phrase are chained together via Branch instructions.

The sources of the linker are very simple and basically uninteresting, so I shall not detail them here. Let me simply give the format of executable files:

- a header, which happens to be the following string (Unix specialists can easily guess the reason): `#!/usr/local/zinc/runtime/interp`
- the number of global values used.
- the values of initialized globals, that is a sequence of one integer (the slot number of the global) and one ZINC value (in prefix form). The integer -1 terminates this list.
- the bytecode (or threaded code), which can be directly executed. Entry point is at offset 0.

6.3 The runtime system

The runtime system is entirely written in portable C, with the exception of the optional threaded-code interpreter, which is written in assembly language. It comprises the following parts:

6.3.1 The loader

The loader is responsible for reading the executable file, allocating and initializing the table of globals, and loading the bytecode in memory. It is completely trivial (100 lines), since all the hard work has been performed by the linker.

6.3.2 The interpreter

The interpreter executes bytecode, or threaded code. In the case of bytecode, it is just one large C function of about 800 lines. (The reason is that the registers of the ZAM must be local variables, in order to be put in actual registers by the compiler.) Here are the first and last lines:

```
void interprete(prog)
     code prog;
{
  register code pc;
  register value * asp;         /* argument-stack pointer */
  register value * rsp;         /* return-stack pointer */
  register value * alloc_ptr;   /* for allocation in the heap */
  register int cache_size;      /* number of values in the volatile env. */
  register value env;           /* the persistent env. */
  register value accu;          /* the accumulator */
  register value tmp;           /* to hold uninitialized allocated objects */

  asp = extern_asp;  rsp = extern_rsp;  alloc_ptr = extern_alloc_ptr;
  pc = prog;         env = null_env;    cache_size = 0;
  accu = MLINT(0);

  while(1) switch(*pc++){
    case CONSTBYTE:
      accu = (value) *(char *) pc; pc += 1; break;
    case CONSTSHORT:            /* Pour les moins petits entiers */
      accu = (value) *(short *) pc; pc += 2; break;
    case CONSTLONG:             /* Pour une valeur hors du tas quelconque */
      accu = (value) *(long *) pc; pc += 4; break;
    case PUSH:
      *--asp = accu; break;
    case POP:
      accu = *asp++; break;
    /* lots of stuff deleted */
  }
}
```

I shall not detail here the whole list of cases, since this would be a lengthy paraphrase of the abstract machine description (sections 3.3 and 6.1). Let me simply mention how the interpreter interacts with the outside, and especially the garbage collector. First, the registers of the ZAM must be known to the garbage collector (to find the roots), yet they are local to the `interprete` function, so we have to save them in global variables (heap and stack pointers) or in globally-accessible areas (the accumulator is pushed on the argument stack, the environment and cache size on the return stack), before calling the garbage collector, and reload some of them after, since the

GC might have moved some objects in the heap. For instance, here is the macro allocating in the heap. We perform linear allocation in the young generation (see next section), and call the minor garbage collector when the end of the generation is reached:

```
#define ALLOC(hdr, size)                                                    \
{                                                                           \
  tmp = VALUE(alloc_ptr);                                                   \
  alloc_ptr += (size);                                                      \
  if (alloc_ptr > young_end) {                                              \
    rsp -= sizeof(struct return_frame) / sizeof(value);                     \
    (struct return_frame *) rsp->env = env;                                 \
    (struct return_frame *) rsp->cache_size = cache_size;                   \
    *--asp = accu;                                                          \
    extern_asp = asp; extern_rsp = rsp; extern_alloc_ptr = alloc_ptr - (size);\
    tmp = call_minor_gc (size);                                             \
    alloc_ptr = extern_alloc_ptr;                                           \
    accu = *asp++;                                                          \
    env = (struct return_frame *) rsp->env;                                 \
    rsp += sizeof(struct return_frame) / sizeof(value);                     \
  }                                                                         \
  HEADER(tmp) = (hdr);                                                      \
}
```

In the case of threaded code, the interpreter is written in assembly language. The reason is that C does not know about indirect jumps, but only about indirect calls, and I found no C compiler able to transform a tail indirect call into an indirect jump, without pushing anything on the stack. I cannot resist the pleasure of putting some 68020 assembly code in this report, so here are a few interpretation routines:

```
retsp   = a7
argsp   = a5
hp      = a4
mypc    = a3
env     = a2
accu    = a1
csize   = d7
base    = d6

CONSTSHORT:
        movw mypc@+, accu
        movw mypc@+, a0
        jmp a0@(base:l)
CONSTLONG:
        movl mypc@+, accu
        movw mypc@+, a0
        jmp a0@(base:l)
PUSH:
        movl accu, argsp@-
        movw mypc@+, a0
        jmp a0@(base:l)
POP:
```

```
        movl argsp@+, accu
        movw mypc@+, a0
        jmp a0@(base:l)
```

6.3.3 The garbage collector

ZINC uses a high-tech garbage collector written by Damien Doligez. It is based upon the Lang and Dupont algorithm [38], which unifies and generalizes the well-known "mark and sweep" and "stop and copy" algorithms. The heap is divided in three zones: the from-space, the to-space and the mark-space. The GC walks the memory graph; blocks in the from-space are copied to the to-space; blocks in the mark-space are marked in use. Then, a sweep of the mark-space collects all unused blocks. By choosing the sizes of the spaces, one can mix "mark and sweep" and "stop and copy" in any proportions, and therefore enjoy the compacting properties of copying collectors, without limiting heap occupancy to 50 %, as in the case of genuine "stop and copy".

Damien Doligez combined this algorithm with the generation mechanism: objects are allocated in a small, separate space (the young generation) and when this space is full, all active objects in the young generation are identified and copied to the large, main heap (the old generation). This operation, the minor collection, takes little time, yet it may recover a lot of space, since most objects are short-lived; major collection, that is collecting the main heap, becomes all the less frequent.

The algorithms, the implementation, and the performances are fully detailed in [23]. The implementation is fairly complex, due to intricacies such as the need to perform a major GC in the middle of a minor GC, and then resume the minor GC as if nothing happened. The whole garbage collector and memory allocator occupy about 1200 lines of C.

6.3.4 Primitive functions

The rest of the runtime system is composed of primitive functions. Some are utility functions on integers and strings, implemented in C for efficiency. It is planned to include the arbitrary-precision integer arithmetic package of Vuillemin *et al.* [55]. The rest is mostly interface with the operating system. This is trivial stub code converting data back and forth between ZINC and C representations. It ought to be generated automatically from the type of the functions. Here is a sample (and once you've seen it, you've seen them all):

```
value unix_open(path, mode, rights)
     value path, mode, rights;
{
  return MLINT(open(CSTRING(path), CINT(mode), CINT(rights)));
}
```

Chapter 7

Assessment

At the time of this writing, the following parts of the ZINC system are operational. First, a compiler and a linker, both written in CAML. The compiler does not perform typing yet, nor consistency checks among modules; also, error reporting is almost nonexistent. Second, a runtime system written in C, comprising a bytecode interpreter, memory allocator, garbage collector, and a few primitives. I wrote also a threaded-code interpreter, and a translator to 68020 assembly code, but these are not yet interfaced with the garbage collector. Primitives and libraries are almost nonexistent, with the exception of a few Unix system calls and a small I/O library. All these parts work well together, and not only on trivial programs; for instance, I was able to run a medium-sized (500 lines) example taken directly from the CAML anthology [25]: an implementation of the Knuth-Bendix completion procedure (see appendix A). I haven't tried to bootstrap the compiler and the linker yet, but I am confident in the ability of the compiler to compile itself; the bottleneck is the lack of libraries and of tools such as a parser generator.

It took me about three months to get a first version of the system able to run this program. The major part of this time went into writing and debugging the bytecode interpreter and its interface with the garbage collector. The compiler took comparatively less time, and was less buggy; this is especially true of the core functions (the compiler for expressions, including the expansion of pattern-matching) which were written at the very beginning and almost did not change since then. This first version of the system had some design flaws, however: it did not have true modules, but simply separate compilation; and linking was performed on the fly, just before execution. Two additional weeks were needed to implement the standalone linker and the module system.

Regarding performance, and especially execution speed, it is common saying that bytecode interpretation is quite slow, so I did not expected very good results. However, ZINC behaves much better than expected, especially on non-trivial programs. Some figures can be found in appendix A. To summarize, it is true that on simple benchmarks such as the Fibonacci function, ZINC is three to four times slower than CAML, garbage collection included. But on real programs such as the Knuth-Bendix completion procedure, the bytecode version of ZINC is slightly faster than CAML. This is partially due to the very inefficient GC of CAML; but even if we discard GC time, the threaded code version of ZINC is as fast as CAML, and no more than 50 % slower than Standard ML of New Jersey, claimed to be the fastest ML implementation to date. This is quite unexpected, since CAML and SML-NJ generates native machine code, while poor little ZINC has to cope with the overhead of interpretation. It demonstrates that the "baroque" mechanisms of the ZAM (special treatment for multiple application, cache on the environment) are really efficient in case of large programs. It also demonstrates that the compilers of CAML and SML-NJ are fairly naive, or more precisely that the rather complex transformations they perform are not very rewarding, and that they do not address the real performance bottlenecks. (A similar conclusion can be drawn from Pierre Weis' metacompiler [61], which is able to generate an ML compiler sometimes more efficient than the CAML compiler.)

The other common saying about bytecode interpretation is that it is very easy to implement, and this one, too, proved to be wrong. Compared to machine code generation, it introduces one additional source of bugs: when a piece of assembly code crashes, we can be sure that the compiler is the culprit; with bytecode interpretation, the bug can be in the interpreter as well. And detecting bugs in the compiler is more complex. First, a disassembler for bytecode must be written, to be able to read the output of the compiler. Then, it is not easy to see bugs at first sight, since the semantics of bytecode instructions change from time to time. Finally, dynamic debugging (e.g. putting breakpoints on some bytecode instructions) requires either adding a breakpoint and trace facility in the bytecode interpreter itself, or putting some breakpoints on the interpreter program itself, and trying to guess what the bytecode program is doing. This is almost as inconvenient as debugging a program using an oscilloscope plugged on the bus of the machine ....

It remains to see now whether the current ZINC implementation fulfills its initial goals:

- getting a small, portable implementation of ML. ZINC is indeed fairly small: the runtime system occupies about 100 K, compiled programs are very compact, thanks to the use of bytecode, and they can run with fairly small heaps, thanks to the efficiency of the garbage collector. And ZINC is indeed fully portable, at least for the bytecode version.

  But ZINC can hardly be called a usable implementation of ML. An evidence is that it still needs CAML to run its compiler and its linker! What prevents bootstrapping yet is not missing language constructs in ZINC (it implements all the constructs used in the compiler), but missing parts of the compiler, such as a lexical analyser, a parser, a typechecker, a debugger and a toplevel; some libraries, such as input/output functions; and lots of primitives. None of these is very difficult to implement, well-known techniques would suffice, but all together they represent a fairly large amount of (rather uninteresting) work.

- providing a testbench for extensions of ML. For the time being, ZINC implements a very classical ML, and no extensions at all. The simplicity of the system would certainly make extensions not very difficult, though I realized the overall design is presumably not as flexible as it could be, and this might make some extensions hard to integrate in it.

  On the other hand, the ZINC implementation itself investigated some new techniques. In particular, it puts to question the dogma according to which a good analysis is a static analysis. ZINC demonstrates that some additional runtime tests are not always absolutely evil, since they can save a lot of computation later on. This is the main idea behind the ZINC execution model, and even if it leads to a slightly baroque abstract machine, it seems to work better than traditional abstract machines in practice. The main problem with this approach is that it requires full control over the execution line: the ZAM works fine with bytecode interpretation, but it is not possible to map it efficiently on existing hardware (the existing code is huge, and with lots of redundancies), since these hardware systems were designed with different purposes in mind. To go further than bytecode interpretation, one would have to microcode the ZAM instructions, in order to keep code size small and take advantage of the fact that several steps of complex ZAM instructions can be executed in parallel. Microcodable architectures are almost inexistent, however.

- learning how ML can be implemented. I definitely learned a lot while designing and implementing ZINC, and this report is an attempt to share this experience. If you find it useful, you should thank Gérard Huet, who caused me to write it; however, all criticism should be directed to me.

Appendix A

Benchmarks

A.1 Simple benchmarking of abstract machines

These preliminary benchmarks compare the ZINC machine with the CAM as defined in [20], and some flavour of SECD [37] (more precisely, a CAM with an additional register to hold the environment). Three variants of the ZINC machine are considered here, differing only in the way environments are represented: by a linear list, by a linear list and a cache, and by a vector and a cache. For each of those five machines, I wrote a bytecode interpreter in C, and a simple compiler in CAML. Compilers and interpreters are all written in the same spirit, so one may hope the comparison is fair. The compilers are unable to compile more than one expression, so I had to use fairly simple and well-known benchmarks. Here are the ML source of the tests:

```
let rec fib = fun n -> if n<2 then 1 else fib(n-1)+fib(n-2) in fib 26;;

let rec tak = fun x -> fun y -> fun z ->
  if x > y then tak (tak (x-1) y z) (tak (y-1) z x) (tak (z-1) x y)
           else z
in tak 18 12 6;;

let rec sum = function [] -> 0 | a::L -> a + sum L in
let rec interval = fun n -> if n=0 then [] else n :: interval(n-1) in
sum (interval 10000);;

let double = fun f -> fun x -> f(f x) in
let quad = double double in
let oct = quad quad in
double oct (fun x -> x+1) 1;;

let rec interval = fun n -> if n=0 then [] else n :: interval(n-1) in
let double = fun f -> fun x -> f(f x) in
let quad = double double in
let succ = fun n -> n+1 in
map (quad quad succ) (interval 1000);;
```

	`fib 26`	`tak 18 12 6`	`sum(interval..`	`map (quad quad succ..`
CAM	25.6 s 785674 words	6.3 s 636094 words	1.2 s 60012 words	13.4 s 1030088 words
SECD	19.1 s 785674 words	5.1 s 636094 words	0.9 s 60012 words	11.2 s 1030088 words
ZAM no cache	21.7 s 785674 words	4.6 s 381658 words	1.1 s 60012 words	12.3 s 1030086 words
ZAM cache+list	24.0 s 4 words	4.9 s 4 words	1.2 s 20008 words	14.0 s 4056 words
ZAM cache+vector	23.3 s 4 words	4.6 s 4 words	1.2 s 20009 words	13.3 s 4078 words
ZAM threaded cache+vector	10.6 s 4 words	2.2 s 4 words	0.5 s 20009 words	6.0 s 4078 words
CAML 40 version 2.6	3.5s (+5.2s) 785738 words	2.6s (+3.9s) 636158 words	0.5s (+0.7s) 60076 words	2.5s (+3.3s) 518118 words
SML-NJ version 0.33	2.0 s n/a	1.0 s n/a	0.12 s n/a	1.5s n/a
Handwritten C	1.4 s 0 words	0.2 s 0 words	0.1 s 20000 words	n/a

Table A.1: Simple preliminary benchmarks

The classical `fib` function tests mostly the speed of function application (to one argument). The `tak` function is highly recursive, too, but with three arguments; since it is curried here, it tests multiple application. The `sum (interval 10000)` test mixes envenly recursive calls and data building and destructuring. Finally, the test using the `double` functional is the only one essentially functional, it shows how well functional values (e.g. partially applied functions) are handled.

The results of the benchmarks can be found in the first part of table A.1. For each test, I give the execution time (measured on a Sun 3/60), and also the amount of heap requested (measured in 32-bit words). Since there is no garbage collector in the bytecode interpreters used here (they simply perform linear allocation in a huge heap), the cost of allocation is underestimated, compared with an actual implementation. Therefore, the raw execution time must be tempered by the volume of heap used.

These figures show that the CAM and the SECD have exactly the same (bad) memory behavior, since they both allocate one pair for each function call. This is dramatic for functions performing only arithmetic computations, such as `fib`; and even functions performing mostly structure building, and little else computation, such as `interval` and `sumint`, still use three times more heap than necessary (about 60000 words instead of the 20000 needed to represent a 10000-element list). The SECD is noticeably faster (about 25 %), because the additional register holding the current environment saves a lot of stack shuffling (the `Swap` instruction of the CAM becomes unnecessary).

The simple ZINC machine (without cache) is not really better, except in the `tak` test, where its efficient multiple application mechanism saves some time and cuts down the heap used by a factor of two; this saving corresponds to the intermediate closures not built by the ZAM. Otherwise, execution times are intermediate between the CAM and the SECD: the ZAM has an environment register, so it is faster than the CAM, but not as fast as the SECD because of the handling of marks on the argument stack.

Adding a cache to the environment cuts down heap allocation by several orders of magnitude. Indeed, for non-functional tests, the volume of heap used is almost the minimal one requested to do the work, that is none for `fib` and `fact`, and 20000 words for `sum(interval...`. The functional test `map(quad quad succ) ...` still allocates about twice as much as strictly necessary (2000 words), due to the building of closures which requires to flush the cache to the heap. Execution times are higher, of course, due mostly to the additional tests for environment access. Using a vector instead of a list for the persistent part helps keeping this overhead acceptable. After all, in the case of `fib`, a 785000-word space-saving easily compensates (in GC time) for 1.6 seconds of additional execution time.

In the second part of table A.1, I tried to compare bytecode interpretation with other execution methods. First, "threaded" code interpretation (that is, code consisting of sequences of pointers, the addresses of the routines executing instructions), is twice as fast as bytecode interpretation, for the same execution model (the ZAM with cache and vectors). Two factors contributes equally to this speedup: first, instruction decoding is made faster (one indirect jump instead of one jump through table); second, the interpreting routines are written directly in assembly language, since C does not know about true indirect jumps, and I am able to write better 68020 code than usual C compilers. As a reference point, I have included figures obtained with the CAML system itself, that is the CAM execution model coupled with expansion of CAM code into machine code. CAML turns out to be two to three times faster than threaded ZAM code for raw execution time, but its memory behavior is almost exactly the one of the original CAM (with the exception of the `(quad quad succ)` test, where a special optimization for combinator applications saves a lot of conses), that is to say awful, hence if we take into account garbage collection times (written in braces), it is not really faster than threaded ZAM code. On these simple tests, Standard ML of New Jersey behaves much better (the given times include garbage collection). Finally, to recall the reader what a well-compiled language would do on these tests, I have included times for equivalent programs written in C; of course, the last test, using full functionality, cannot be translated easily.

A.2 Real-sized benchmarking of ZINC

For more realistic benchmarks, I have used an implementation of Knuth and Bendix's completion procedure for equational theories [34]. This implementation is due to Gérard Huet, and the original CAML source can be found in [29] and [25]. It is a simple, yet not totally naive, implementation, written in "classical ML" style (no side-effects, lots of functionals, ...), and I think it is representative of the use of ML for prototyping and teaching.

The original sources required very little adaptation to run on ZINC. Indeed, the sole incompatibility was due to the fact that the original code relied on CAML parsing `(a,b,c)` as `(a,(b,c))`, while in ZINC the first is a triple and the latter is a pair. A few extra braces fixed it immediately. For conveniency, I also split the program in six small modules. The benchmark itself consists in the completion of the axioms of groups. It was run on a Sun 3/280, using the following implementations: ZINC with a bytecode interpreter written in C; ZINC with a threaded code interpreter written in 68020 assembly language; ZINC with a translator producing native 68020 code; the distribution CAML version 2.6; an experimental version of CAML using the same efficient garbage

System	Time	Size
ZINC, interpreted bytecode	29.0 s	5.4 K
ZINC, interpreted threaded code	15.5 s	8.2 K
ZINC, expansion to 68020 native code	9.7 s	31.7 K
CAML 40 (version 2.6)	36.6 s (15.4 s + GC 21.2 s)	11.7 K
CAML (new runtime)	14.7 s	n/a
SML-NJ (version 0.33)	10.5 s	n/a

Table A.2: The Knuth-Bendix completion benchmark

collector than ZINC, and a new runtime system; and finally, Standard ML of New Jersey, release 0.33. The results are summarized in table A.2.

The first conclusion is that interpretation performs quite well compared to naive generation of machine code. Indeed, expansion to 68020 assembly language is just one-third faster than threaded code interpretation. Going from bytecode interpretation to threaded-code interpretation is almost as efficient as going from bytecode interpretation to expansion to native code. This is due to the fact that in case of the threaded-code interpreter, the routines executing the instructions are hand-written in assembly language, therefore they are as efficient as the pieces of assembly language generated by the expander, while interpretation overhead is reduced to a minimum. It is not negligible yet, however: both interpreters spend 37 % of their time fetching and decoding the next instruction (10.7 s in case of bytecode, 5.6 s in case of threaded code).

The second conclusion is that a toy implementation such as ZINC compares quite favorably with mature implementations such as CAML and SML-NJ. Even the portable, bytecode-interpreted version of ZINC is faster than CAML. This is mostly due to the inefficiency of CAML garbage collector; the new runtime, using the same garbage collector as ZINC, cuts down execution times by a factor of two. Even then, the threaded-code interpreter is almost as fast as CAML, and only one-third slower than SML-NJ, claimed to be one of the fastest ML implementations to date. These results are quite unexpected, given the results of the preliminary benchmarks. They show that the special mechanisms of the ZAM (efficient multiple application, cache on the environment, ...) are really efficient for non-trivial programs.

To confirm this intuition, I ran an extensive profiling of the bytecode interpreter. A summary of instruction frequencies and execution times is given in table A.3. The special mechanisms for multiple application prove quite efficient, since the Grab instruction almost never fails: over 345581 executions, only 43 found an empty argument stack and had to build a closure. This shows that partial application of a curried functions seldom happens. Symmetrically, the case where Return does not go back to the caller, but applies the result to the next remaining argument, is not uncommon at all (78315 times out of 193431, that is about 40%).

Regarding the cache on the environment, it cuts down heap allocation for environments by a factor of four: among the 1237285 words of data that were stored in the environment during the test, only 410450 had to be copied back to the heap. Conversely, accesses to the environment have a cache hit rate of 69.8%. Average size of active caches (those in the return stack) is between two and three words.

Memory behavior is as follows: 1178465 words (without taking headers into account) are allocated in the heap. Among these, 35.3% hold data structures, 36.5% persistent environments, and the remaining 28.2% are closures. In contrast, CAML allocates 2027906 conses to run this

Instruction	Frequency
Access0	1449497
Push	1427445
Getfield1	792583
Getfield0	655375
Access1	512027
Appterm	433327
Getglobal	394174
Access2	355109
Branchifnot	352176
Grab	345581
Pushmark	285221
Apply	284121
Return	193431
Letrec1	157697
Switch	153208

Instruction	Time (in seconds)
Letrec1	5.0 + GC 2.1
Push	3.7
Grab	3.2
Access0	3.1
Appterm	3.0
Makeblock2	1.8 + GC 1.0
Access2	2.4
Getfield1	2.3
Getfield0	2.2
Apply	2.2
Access1	2.2
Getglobal	2.0
Branchifnot	1.7
Branchifnequal	1.4
Return	1.2

Table A.3: The top 15 instruction counts and interpretation times

benchmark, that is over 4 million words, and among these, only 8.3% hold data structures.

From these figures, and especially those for the Letrec1 instruction, it seems that the main performance bottleneck is the treatment of local function definitions: building a closure each time is the main source of cache flushing.

Bibliography

[1] Martín Abadi, Luca Cardelli, Benjamin C. Pierce, Gordon D. Plotkin. "Dynamic Typing in a Statically Typed Language." *16th Ann. ACM Symp. on Principles of Programming Languages*, 1989.

[2] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, Jean-Jacques Lévy. "Explicit Substitutions." To appear in *17th Ann. ACM Symp. on Principles of Programming Languages*, 1990.

[3] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman. *Compilers, principles, techniques, and tools.* Addison-Wesley, 1986.

[4] A. Appel. *A runtime system.* Draft, Princeton University, 1989.

[5] Andrew Appel, David MacQueen. "A Standard ML Compiler." *Functional Programming Languages and Computer Architecture*, Lecture Notes in Computeer Science 242, Springer-Verlag, 1987.

[6] Andrew Appel, Trevor Jim. "Continuation-passing style, closure-passing style." *16th Ann. ACM Symp. on Principles of Programming Languages*, 1989.

[7] L. Augustsson. "A compiler for Lazy ML". *1984 Lisp and Functional Programming Conference*, 1984.

[8] Joel F. Bartlett. *Compacting Garbage Collector with Ambiguous Roots.* Technical report, DEC Western Research Laboratory, February 1988.

[9] Gérard Berry, Ravi Sethi. "From regular expressions to deterministic automata." *Theoretical Computer Science* 48 (1986), 117–126.

[10] N. G. de Bruijn. "Lambda-Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation with Application to the Church-Rosser Theorem." *Indag. Math.* 34, 5 (1972), 381–392.

[11] Luca Cardelli. "The Functional Abstract Machine." *Polymorphism*,

[12] Luca Cardelli. "Amber." *Combinators and Functional Programming Languages, Proc. 13th Summer School of the LITP, Le Val d'Ajol, May 1985*, Lecture Notes in Computer Science 242, Springer-Verlag, 1986.

[13] Luca Cardelli. “The Amber Machine.” *Combinators and Functional Programming Languages, Proc. 13th Summer School of the LITP, Le Val d'Ajol, May 1985*, Lecture Notes in Computer Science 242, Springer-Verlag, 1986.

[14] Luca Cardelli. “Typeful Programming.” Research report 45, DEC Systems Research Center, 1989. To appear in *Proc. IFIP State of the Art Seminar on Formal Description of Programming Concepts (Rio de Janeiro, April 1989).*

[15] Luca Cardelli, Peter Wegner. “On understanding types, data abstraction, and polymorphism.” *Computing surveys*, vol. 17(4), 1985.

[16] Luca Cardelli, Xavier Leroy. “Abstract Types and the Dot Notation.” To appear in *Proc. IFIP Working Conference on Programming Concepts and Methods (Sea of Galilee, April 1990).*

[17] Jérôme Chailloux. *La machine LLM3.* INRIA technical report 55, 1985.

[18] Jérôme Chailloux *et al. Le_Lisp version 15.22, manuel de référence.* INRIA.

[19] Guy Cousineau, Gérard Huet. *The CAML Primer.* INRIA.

[20] Guy Cousineau, Pierre-Louis Curien, Michel Mauny. “The Categorical Abstract Machine.” In *Functional Programming Languages and Computer Architecture*, J. P. Jouannaud ed., Lecture Notes in Computer Science 201, Springer-Verlag, 1985.

[21] Pierre Crégut. *Machines abstraites pour la réduction de λ-termes.* Thèse de doctorat, Université Paris VII, forthcoming.

[22] Alain Deutsch. “On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications. ” To appear in *17th Ann. ACM Symp. on Principles of Programming Languages*, 1990.

[23] Damien Doligez. *Réalisation d'un glaneur de cellules de Lang et Dupont à générations.* Mémoire de D.E.A., Université Paris VII, Sept. 1989.

[24] J. Fairbairn, S. C. Wray. “TIM: a simple abstract machine for executing supercombinators”. *Functional Programming and Computer Architecture*, 1987.

[25] Formel project. *The CAML Anthology.* INRIA.

[26] M. Gordon, R. Milner, C. Wadsworth. *Edinburgh LCF.* Lecture Notes in Computer Science 78, Springer-Verlag, 1978.

[27] Thérèse Hardin, Jean-Jacques Lévy. *A confluent calculus of substitutions.* Draft.

[28] Robert Harper, David MacQueen, Robin Milner. “The definition of Standard ML, Version 3”. Laboratory for Foundations of Computer Science, University of Edinburgh, 1989.

[29] Gérard Huet. *Formal structures for computation and deduction.* Course notes, Carnegie-Mellon University, May 1986.

[30] Gérard Huet. “The constructive engine.” In *The Calculus of Constructions*, INRIA technical report 110, 1989

[31] Lalita A. Jategaonkar, John C. Mitchell. "ML with Extended Pattern Matching and Subtypes". *Proc. 1988 Conference on Lisp and Functional Programming.*

[32] T. Johnsson. "Efficient compilation of lazy evaluation." *ACM Conference on Compiler Construction*, 58–69, 1985.

[33] Brian W. Kernighan, Dennis M. Ritchie. *The C programming language.* Prentice Hall, 1978, 1988.

[34] D. Knuth, P. Bendix. "Simple word problems in universal algebras." In *Computational Problems in Abstract Algebra*, ed. J. Leech, Pergamon Press (1970), 263–297.

[35] David Krantz, Richard Kelsey, Jonathan Rees *et al.* "ORBIT: An Optimizing Compiler for Scheme." *SIGPLAN Notices*, 21(7), July 1986.

[36] J.-L. Krivine. Unpublished.

[37] P. J. Landin. "The mechanical evaluation of expressions. " *The Computer Journal* 6 (1964), 308–320.

[38] Bernard Lang, Francis Dupont. "Incremental Incrementally Compacting Garbage Collection." *SIGPLAN Notices*, 22(7), July 1987, 253–263.

[39] Alain Laville. *Evaluation Paresseuse des Filtrages avec Priorité, application au langage ML.* Thèse de doctorat, Université Paris VII, 1988.

[40] Alain Laville. "Lazy pattern matching." Draft.

[41] R. G. Loeliger. *Threaded interpretive languages.* Byte Books, 1981.

[42] David MacQueen. "Modules for Standard ML." *1984 Conference on Lisp and Functional Programming*, 198–207.

[43] David MacQueen. "Using Dependent Types to express Program Structure." *13th Ann. ACM Symp. on Principles of Programming Languages*, 1986.

[44] Michel Mauny. *Compilation des langages fonctionnels dans les combinateurs catégoriques, application au langage ML.* Thèse de troisième cycle, Université Paris VII, September 1985.

[45] Michel Mauny. *Spécification CAML d'un sous-ensemble de CAML.* Course notes, D.E.A., Université Paris VII, 1989.

[46] Michel Mauny, Ascander Suárez. "Implementing Functional Languages in the Categorical Abstract Machine." *Proc. 1986 ACM conference on Lisp and Functional Programming*, 266–279.

[47] Greg Nelson *et al. Modula-3 Report.* Research report 31, DEC Systems Research Center, 1988.

[48] Atsushi Ohori, Peter Buneman. "Type Inference in a Database Language." *Proc. 1988 ACM Conference on LISP and Functional Programming*, 174–183.

[49] D. L. Parnas. "On the criteria to be used in decomposing systems into modules." *Communications of the ACM*, 15(12), 1053–1058, December 1972.

[50] S. Peyton-Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.

[51] Laurence Puel, Ascánder Suárez. *Compiling pattern matching by term decomposition.* Research report 4, DEC Paris Research Laboratory, 1989.

[52] Didier Rémy. *Etude de la génération de code du langage CAML et de son optimisation.* Mémoire de D.E.A., Université Paris VII, 1987.

[53] Didier Rémy. "Records and variants as a natural extension of ML." 16th Ann. ACM Symp. on Principles of Programming Languages, 1989.

[54] Didier Rémy. *Algèbres touffues. Application au typage polymorphe des objets enregistrements dans les langages fonctionnels.* Thèse de doctorat, Université Paris VII, forthcoming.

[55] Bernard Serpette, Jean Vuillemin, Jean-Claude Hervé. *BigNum: A Portable and Efficient Package for Arbitrary-Precision Arithmetic.* Research report number 2, DEC Paris Research Laboratory, 1989.

[56] Richard Stallman. *GNU Emacs Manual.* Free Software Foundation, 1985.

[57] Richard Stallman. *Using and Porting GNU CC.* Free Software Foundation, 1988.

[58] G. L. Steele, G. J. Sussman. *Revised report on Scheme, a dialect of Lisp.* MIT AI memo 452, 1978.

[59] Ascánder Suárez. *Une implémentation de ML en ML.* Thèse de doctorat, Université Paris VII, forthcoming.

[60] P. Wadler, S. Blott. *How to make* ad-hoc *polymorphism less* ad-hoc. 16th Ann. ACM Symp. on Principles of Programming Languages, 1989.

[61] Pierre Weis. *Le système SAM: métacompilation très efficace à l'aide d'opérateurs sémantiques.* Thèse de doctorat, Université Paris VII, 1987.

[62] Pierre Weis *et al. The CAML Reference Manual, version 2.6.* INRIA, 1989.

[63] N. Wirth. *Programming in Modula-2.* Texts and Monographs in Computer Science, Springer-Verlag, 1983.