Functional low-level interpreters

Jerzy Karczmarczuk

Dept. d'Informatique, Université de Caen, France

karczma@info.unicaen.fr

Abstract. We show how to construct in a pure, lazy functional language Haskell a low-level, FORTH-like (stack based, postfix style) virtual machine, used as a target platform in our compilation course for computer science students. The implementation is clear, reasonably efficient, and easily extensible. Two models are presented: more "classical", with a separate return stack, and an interpreter based on the continuation-passing style. The idea, although traditional, was a bit experimental, since this was the first contact of our students with Haskell and with advanced functional techniques. We exploited thus this course to teach also those techniques, on a comprehensive and conceptually coherent set of examples.

1 Introduction: Teaching Compilation Functionally

Users of functional languages in a pedagogical context usually insist on a good balance between formal discipline, regularity and capacity to build abstract models, and the clarity and conciseness of concrete, practical codes, their syntactic and semantic simplicity. In our opinion those concrete, applicative examples treated as a vehicle conveying the paradigms of functional programming are often better than a dedicated "language" course. They help to neutralize the popular "curse" of FL: to be good for teaching, but not so useful for a practical work. Thus, we used functional languages to teach image synthesis, and computer graphics in general, and we decided to teach compilation using Haskell as the main implementation tool as well.

Compilation — as seen from this point of view — is an ambiguous domain. We know that the analysis and the modelling of syntactic structures is very well adapted to functional treatment. Parser combinators [1, 2] implemented in a typed functional language are wonderful tools permitting to *understand* the syntactic analysis much easier than through classical techniques proposed in the Dragonbook [3], or in books of Andrew Appel [4]. The laziness is an invaluable tool to process the semantic attributes (the flows of synthesized and inherited attributes are antithetic, and sometimes it is useful to be able to deal easily with cyclic dependencies). The typechecking, the generation of intermediate, arborescent code — all this is *par excellence* functional, at least at the presentation level.

The administration of dynamic environments and the low-level, back-end code is usually much "less functional". If a teacher who cannot during a one-semester course construct a fully-fledged native code compiler, gets satisfied with some assembly-style pseudo-codes, or uses C as the target language, his students might get discouraged from

FP, since the "real work" needs *finally* an imperative language... The fact that professional compilers such as Glasgow Haskell, or Clean have been designed and coded in Haskell or Clean [5, 6], will not help typical students, since the hiatus between teaching introductory compilation and building industrial-strength compilers is still quite wide.

We decided thus to try a little pedagogic experience, to teach compilation with Haskell (initially unknown to students!), beginning with the analysis of the source, and ending with the execution of the compiled (or hand-assembled) programs [1]. It was not our aim to teach the compilation *of* functional languages, but something more orthodox, and rather simple. For the compilation of functional languages we suggested the very abundant literature, e.g., [7–9]. The presented material is a (modified a little) fragment of the compilation course delivered to the 4-th year (Maîtrise) computer science students at the University of Caen. Our students knew Scheme and some imperative languages, and they were competent enough to accept any not too exotic algorithmic construction.

We wanted to show that the construction of low-level, assembly- (or FORTH/Postscript) style, stack-based, reasonably efficient virtual machines is quite straightforward and readable. This readability is largely due to the abstraction facilities of Haskell, notably the higher-order functions.

We exploited the laziness of the language in order to simplify the deployment — the implementation and its presentation — of the "non-linear" low-level code, containing branching (conditionals and loops). Of course, every virtual machine must be anchored on its implementation language and the related run-time support. We made it clear that the usage of Haskell, which provides not only the memory management, but offers the tail recursion and functional composition, facilitates the construction of generic entities through currying and other higher-order constructs, etc., is extremely advantageous for *learning* the construction of low-level code structure.

1.1 Why interpreters?

While nobody claims that in last 15 years the teaching of compilation has undergone a significant revolution, the importance of "small", interpreted, often embeddable languages: Ruby, PHP, Python, Javascript, etc. grows fast, embedding small virtual machines into all kind of applications became explosively popular, and the construction of interpreters *must* be taught. They provide a useful, hardware-independent, but concrete target platform for a typical compilation course. While a typical computer-science student will probably never produce a full, native code compiler, chances that he/she will need some scriptable applications, equipped with "intelligent" linguistic interfaces, are fairly good.

Our interpreters should handle codes generated by small *integrated* compilers (the code is not designed to be loaded from external sources), and provide some reasonable compromise between nice, but formal models, the huge real stuff (Java or Smalltalk virtual machines), and the brutal simplicity of stack-based calculators. For teaching purposes the low-level efficiency (fast access to data, elimination of all redundancy) is less important than

[1] Actually, chronologically it was reversed...

– *simplicity*, compactness, modularity and extensibility of the interpreter kernel,
– readability of the target code (bytecodes, threaded procedure calls, etc.);
– and finally: *seriousness*, practical usability of the model. Neither FORTH, nor PostScript have nothing to be ashamed of. . .

The machine should be reasonable from the high-level point of view, avoiding spaghetti coding (e.g., multiple indirections). We decided thus to construct some small stack-based kernels using Haskell, which at the same time gave us the opportunity to show to students some concrete applications of standard and advanced functional techniques, whose knowledge at the beginning of the course was very rudimentary. However, this was just a small part of the compilation course, nothing to do with the comprehensive approach of Kamin [10]. Some part of our inspiration was due to Knuth [11], and his viewpoint on the role of low-level semantic models in the teaching of programming and algorithms. . .

This low-level stack machine was then used as a model target for the code generators taught during the same course. A typical low-level code is quite "stateful": updateable stacks, environments, and/or registers, etc. Moreover, it is often considered as untyped, e.g. the assembly code may put in the same locations integer or floating-point numbers, and the pointer arithmetic for array access is considered standard. Since the whole idea of our work is based on the representation of low-level constructs using a high-level, typed language, we didn't introduce explicit records containing tags and values, but trivialize the whole issue through Haskell data structures representing tagged types:

```
data Value = Fail String | I Integer | F Double | S String
           | B Bool | L [Value] | P (Value,Value)  -- etc.
```

easy to generalize. They were equipped with all the essential properties: arithmetic (where applicable), comparisons, etc., which by itself was a nice training field for the type class system. The **`Code`** was a linked chain of **`CodeItems`** described below (rather than an array; the assembly of chunks and pattern-based traversal, i.e., its execution, were much easier in that way), but the conversion to a contiguous vector — for efficiency reasons — has also been discussed.

2 A Simple Stack Machine

The basic aim was to construct a *really* simple target, FORTH-style machine, whose programs would have a form similar to: **`[ild 5, call cube, stop]`**, where **`cube`** is a user defined procedure: **`[dup,dup,mul,mul,ret]`**. The meaning of identifiers is evident; **`dup`** is a primitive which duplicates the element at the data stack top, **`mul`** multiplies the last two stack items, **`ret`** returns from a procedure, etc. We wanted that students be able to construct *in a few hours* such small programming *low-level* examples, together with the interpreter of such programs, and with a full plethora of primitives and user-defined functions for lists, numerics, etc. This part of the course preceded parsing.

Since the machine is just a module of an integrated compiler-interpreter system, there was no point in introducing symbolic bytecodes. *Instructions are Haskell function calls*, no decoding is needed during the execution, and the extensibility is unlimited.

The machine has two stacks, as typical FORTH architectures: the data stack which stores values, and the return stack for the flow of control:

```
type Dstack = [Value]
data Rstack = [Code]
```

If we neglect — for the moment — the global environment, and some other secondary issues, the machine ***could*** be coded as a simple loop:

```
evalloop :: Rstack -> Dstack -> Dstack
evalloop Empty st = st
evalloop ((instr:>rcode):rtail) st =
  let (nst,nrts) = instr st (rcode:>rtail)
  in  evalloop nst nrts
```

with code items being:

```
type CodeItem = Dstack -> Rstack -> (Dstack,Rstack)
data Code = Empty | CodeItem :> Code
```

where for technical and training purposes the code is not a list but a specific data structure, which may be constructed from an ordinary list by the converter **`lcode = foldr (:>) Empty`**. Every procedure call should put the rest of the code on the **`Rstack`**, retrieved upon the return. All data operations act on the data stack, which is the final result of the program. (The global tree-like environment has ben implemented as well, but its handling will be omitted from this text, since it doesn't introduce anything particularly fascinating to our presentation).

But we didn't code the interpreter in such a way. Our philosophy was formulated as follows: *don't think about the "interpreter loop" at all! The instruction dispatching in a modern hardware is distributed. Think of the underlying Haskell infrastructure as of an "intelligent hardware", permitting to each instruction to know its successor, and to "jump to it" (tail-call it). Instructions themselves drive the control flow. This locality makes everything easier to read, and actually it is also easier to make experiments with the semantic model of the machine*.

2.1 Threaded Code

Thus, the most interesting property of the main interpreter loop is that it doesn't exist. It is replaced by a threaded variant, which might be considered as a *specific version* of the continuation-passing style programming. The real **`CodeItem`** within this model has the type

```
type CodeItem = Dstack -> Code -> Rstack -> Dstack
```

i.e., an instruction needs the two stacks and the *following* code as its parameters, and the program starts by executing a function **`start :: Code->Dstack`** defined as

```
start (instr:>prog) = instr [] prog ([stop:>Empty])
```

The program stops upon the execution of `stop stk _ _ = stk`. *All* other instructions pass the data to their successors, the machine follows the protocol of the *threaded code* [12, 13]. Among these instructions, particular role is played by the operators which acts only on the data stack without specific control behaviour. They are defined in a most generic way, showing to students the power of functional abstraction. For example the addition and similarly all other arithmetic operators are defined as

```
add = op2 (+)                 -- This (+) acts on Values
sub = op2 (-)
mul = op2 (*)
```

etc., where

```
op2 op = stkop (\(x:y:stk) ->y `op` x : stk)

stkop :: (Dstack -> Dstack) -> CodeItem
stkop op stk (instr:>code) rstack =
      instr (op stk) code rstack
```

The generic operator **stkop** defines also the comparisons, and other **Dstack** manipulations, for example

```
relop :: (Value->Value->Bool) -> CodeItem
relop op = stkop (\(x:y:stk) ->B (y `op` x) : stk)
eq = relop (==)
gt = relop (>)                -- etc.

ld c = stkop (c :)            -- Load a known constant
tld t c = ld (t c)            -- ... a specific constructor
ild = tld I                   -- e.g., Integer
fld = tld F                   -- Double, etc.

dup  = stkop (\s@(x:_) ->x:s)  -- Clasical FORTH/PS stack ops
pop  = stkop (\(_:stk) ->stk)  -- (or drop.)
exch = stkop (\(x:y:stk) ->y:x:stk)   -- (or swap.)
```

etc. The usual control structures such as procedure calls and returns take the form

```
call (instr:>prc) stk cod rstk = instr stk prc (cod:rstk)
jmp (instr:>proc) stk _ = instr stk proc  -- Yes, the 'goto'...

ret stk _ ((instr:>code):rstk)  = instr stk code rstk
```

We want to underline the fact that this part of the course played a major part in acquainting the students with the typical construction of a relatively complex functional package, with pattern-matching, currying [2], higher-order functions, polymorphism, type classes, and simple, but non-trivial data structures. In our opinion such comprehensive, coherent programming project is more effective in teaching those issues than a much bigger set of heterogeneous small examples.

[2] E.g., several definitions omit the **Rstack** parameter, since it is just a trailing spectator.

2.2 Decisional mechanisms

In PostScript the **`if`** and **`ifelse`** instructions require that the conditional code wrapped as procedures is put first on the data stack, and retrieved thereof by the corresponding primitive. The same technique is used in the implementation of the **`while`** loop. While this is *de facto* a practical model, we wanted explicitly to deal with a more primitive, low-level control structures, a linear code with branching, implementing literally such diagrams as on Fig.1 The primitive conditional statements, e.g., a conditional branching

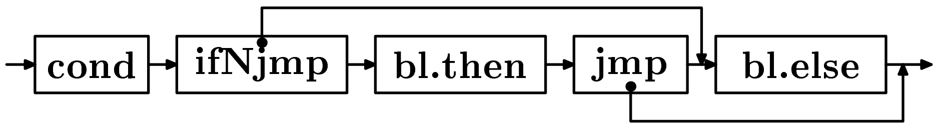

Fig. 1. If-then-else

statement **`ifjmp proc`** is straightforward:

```
ifjmp (instr:>ccode) (B cond : stk) (nxt:>ncode)
  | cond = instr stk ccode
  | otherwise = nxt stk ncode
```

and its inverse, **`ifNjmp`** simply reverses the condition. The first parameter of this procedure: **`(instr:>ccode)`** is the chunk of code executed conditionally, whenceupon the "statically linked" rest of the code: **`(nxt ...)`** is simply abandoned. This is a delicate point during the presentation. In assembly language a branching statement needs a label, and several textbooks devoted to compilation introduce labels while discussing the low-level code generation. They are obviously artificial entities, and references to them, especially forward references need some gymnastics. Our philosophy goes as follows: *labels are needed for identifying some code chunks. But here these chunks are tangible, they are sequences of functional objects, you use* ***them*** *directly as targets of your referring statements. Yes, you are constructing a linear code, but the assembly process need not be linear, your Haskell underlying machinery is sufficiently powerful to construct all kind of cross-referring chunks, also cyclic.*

So, if the compiler (or a human) identifies the chunks: the condition, and both conditional blocks in an **if-then-else** statement, and if it/she knows the "future" (i.e., the continuation) **`continue`** of the conditional, the code depicted on the Fig.1 is *generated* by the following construction

```
ifelse_gen cond thcode elcode continue =
  let othwise = elcode +> continue
  in cond +> (ifNjmp othwise :> thcode)
          +> (jmp continue :> othwise)
```

where **`(+>)`** is the concatenation operator for the **`(:>)`** sequences. Exactly in the same way we generate the **while** loop, graphically represented on Fig.2. This gave us the

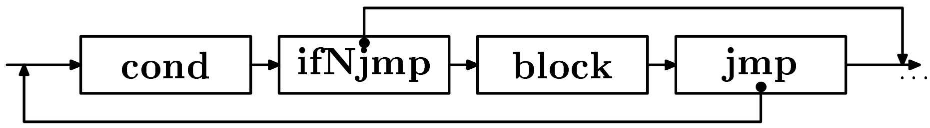

Fig. 2. While loop

opportunity to advocate the use of laziness in order to handle cyclic (here: self-referring) dependencies. The code deployment generator:

```
while_gen cond block continue =
 let wchunk = cond +> (ifNjmp continue :> block)
                   +> (jmp wchunk :> continue)
 in  wchunk
```

is so compact that it is difficult to imagine something shorter. But it is possible; we shall get to it later, see the section (4)...

2.3 Some examples and extensions

This model was designed for the manual construction of low-level programs, the students coded such procedures as the recursive:

```
factorial = ifelse_gen initf tproc eproc (ret:>Empty)
 where
   initf = lcode [dup,ild 0,eq]
   tproc = lcode [pop,ild 1]
   eproc = lcode [dup,ild 1,sub,call factorial,mul]
ftest = lcode [ild 6, call fact, ret]
```

and iterative (with a buffer variable) factorial:

```
itfact = init +> while_gen (ild 0 :> gt :> Empty) loop fin
 where
  init = lcode [dup,ild 1,exch]
  loop = lcode [param 1,mul,exch,ild 1,sub,exch,param 1]
  fin  = lcode [exch,pop,ret]
itftest = lcode [ild 7, jmp itfact] -- note the abbrev. of call/ret
```

where the primitive **param**

```
param n = stkop (\stk ->stk!!n : stk)
```

loads on the stack one of its previously stored elements. It is obvious that such stack manipulations as in the examples above are *never* generated by a typical compiler. The function **param** and a few others, e.g. the stack "rotation", dropping an arbitrary number of stack items, or assembling/disassembling lists from/to stack item sequences were introduced in order to prepare the machine for handling codes produced automatically by the generator from high-level constructs.

The purpose of these examples was to pinpoint the appearance of typical coding patterns and facilitate the future construction of the code generators, and also to signal the importance of decent debugging/tracing tools. The machine provided not only the numerical primitives, but also some simple list handling utilities, permitting e.g. to construct the iterative list reversal procedure:

```
lreverse = lcode [lld [], exch, jmp rev]
rev = lcode [dup, lld [], ne, ifjmp rec, pop, ret] +> rec
 where
  rec = lcode [hdtl, exch, roll 2, cons, exch, jmp rev]
```

where **`hdtl`** is a **`stackop`** separating the head and the tail of a list on the stack, **`roll`** is a stack rotation operator:

```
roll n = stkop rollp where
  rollp (x:l) = let (t,d)=splitAt n l in t ++ (x:d)
```

and **`lld`** loads a list on the stack. The **`splitAt`** function separates the first n elements of a list from the rest. The students were encouraged to get acquainted with the standard Haskell libraries, and to recode in postfix style other popular functions.

3 More Elaborate Extensions

3.1 Error handling

The division by zero, an attempt to split an empty list, etc. exceptional situations are not handled by this machine. While such problems might be secondary in many subdomains of computer science, during a compilation course this is a primary issue. It is easy to augment the concerned primitives so that they return on the stack the special value **`(Fail`** ***`message`*****`)`**, and that the generic data processor **`stkop`** refuse to apply any operator to such a stack, which propagates the failure ([14]) until the end of the concerned chunk, but this solution is clearly incomplete: a failure is a *control*, not only *data* problem. After a failure the program may execute some idle instructions, but it shouldn't be allowed to call or branch.

We wanted to implement a simple, but universal "trapping" solution, something similar to **throw/catch**, or the "escape continuation" mechanism. In some models there is a global "trap stack" where the program deposes the snapshot of its state before entering a potentially dangerous code. We implemented this as well; the escaping instruction retrieved all the information from this stack, and continued with the recovery code, and the previously stored stacks. But adding one more parameter (almost never used) to all **`CodeItems`** is ridiculous [3]. Our solution is local, and demands a specific code generation protocol.

The idea is to be able to write a code of the form

[3] Inefficient, of course, but — which is even more sinful in a pedagogical context — it is plainly boring...

```
lcode [ ... , trap block escode, nextInstr, ...]
```

which makes a snapshot of the stacks before entering the **`block`** chunk. If this block terminates, the control passes automatically to **`nextInstr`**, but if it executes some special "escape" instruction: **`block = lcode [ ...,throw, ...]`**, the **`throw`** instruction (not a general command; just an appropriately defined identifier) passes the control to **`escode`**, with the previously stored stacks. The machine executes **`escode`** and stops, but the last stage is conventional, other possibilities are also plausible, e.g., a return, or the same continuation as previously. The question is: how a **`trap`** instruction, which performs the checkpoint (execution time) enables the coding (assembly time) of the **`throw`** instruction? The argumentation goes as follows:

*The natural way to make an object — here: a program chunk — dependent on something external, is to parameterize it. We shall discover now the power of **closures**, entities which compose a piece of code and some data available at the moment of the closure call, but referred during the closure coding. These closures may be used anywhere, even outside the definition scope of those data. Any chunk of code may be constructed and used in the following form:* **`block throw = lcode [ ...,throw, ...]`**, *where* **`throw`**, *the escaping instruction is passed from* **`trap`**. *In modern functional languages closures are relatively cheap. If we agree to exploit parametrized primitive (Haskell) functions in our machine, there is no reason not to parameterize code chunks during the assembly process. During a normal work this won't introduce any penalties.*

Thus, the **`trap`** construct may be defined as

```
trap block (instr:>escode) stk code rstk =
 let throw _ _ _ = instr stk (escode +> endprog) rstk
 in  jmp (block throw +> code) stk code rstk
```

where **`endprog = stop :> Empty`**. Let us show a simple example, the coding of a product function, literally translated from

```
prod [] = 1.0
prod (x:q) = x*prod q
```

We shall use the exception trapping mechanism as a control structure which stops the recursion immediately when a zero is encountered. Here is the original code:

```
prod = lcode [dup, lld [], eq, ifNjmp nonil,
              pop, fld 1, ret] +> nonil
  where  nonil = lcode [hdtl, exch, call prod, mul, ret]
```

and here is an augmented code, contrived a little, but clear. It is obvious that if such mechanism is to be used as a "normal" control structure, the escape code should not stop the program.

```
xprod = lcode [trap blk escape, ret] where
 blk exit = lcode [jmp prod]  where
   prod = lcode [dup, lld [], eq, ifNjmp nonil,
                 pop, fld 1, ret] +> nonil
```

```
   nonil = lcode [hdtl, dup, fld 0, eq, ifjmp bang,
                  exch, call prod, mul, ret] +> bang
   bang = lcode [exit]
 escape = lcode [sld "Zero. You lost!"]
```

Students were encouraged to propose and to code several shortcuts: conditional exceptions avoiding thus some branching instructions, augmented **stkop** operators which throws the exception upon a **Fail**ure, etc.

3.2 Tracing

The purity of the machine was — apparently — an obstacle to trace the execution of the program, and to debug it.

The changes took less than 30 minutes of presentation, and it was the first occasion to seee the Haskell IO in a concrete application, after some general introduction and some practical exercises, but before the discussion of all the intricacies of monads introduced together with parsers. We changed the type of the instruction:

```
type CodeItem = Dstack -> Code -> Rstack -> IO Dstack
```

and embedded (almost) everything in the IO monad. in such a way every instruction could *perform an action*.

Some of the (simplified) modifications of the code are presented below.

```
start (instr:>prog) =
  putStrLn "START TRACING" >>
  instr [] prog ([stop:>Empty]) >>=
  print >>                              --  the stack
  putStrLn "END TRACING"


stkop msg op stk (instr:>code) rstack
  = putStr (msg++"; Stack: ") >>
    putStrLn (show stk) >>
    case stk of
      (Fail _ :_) -> instr stk code rstack
      _           -> instr (op stk) code rstack


add = op2  "ADD"  (+)    --  op2 inherits msg from stkop
sub = op2  "SUB"  (-)
mul = op2  "MUL"  (*)

stop stk _ _ =
  putStrLn "STOPPING" >> return stk

ret stk _ ((instr:>code):rstk) =
```

```
  putStrLn "RETURNING" >>
  instr stk code rstk
```

etc. The **call** instruction contained the called procedure name. The code assemblers (**while_gen**, etc.) did not change, only the **jmp** instruction inside took a message parameter,

```
ifjmp (instr:>ccode) (B cond : stk) (nxt:>ncode) r
  | cond = putStrLn "C.BRANCHING" >> instr stk ccode r
  | otherwise = putStrLn "SKIPPING" >> nxt stk ncode r

ifelse_gen cond thcode elcode continue =
  let othw = elcode +> continue
  in cond +> (ifNjmp othw :> thcode) +>
             (jmp "CONTINUE" continue :> othw)
```

and the interpreted code (**lcode [...]**) remained almost untouched. The trace took the following form:

```
*Tmachine> start (lcode [ild 2, jmp "Here we go" factorial])
Start tracing
LOAD 2; Stack: []
BRANCHING TO Here we go
DUP; Stack: [2]
LOAD 0; Stack: [2,2]
EQ; Stack: [0,2,2]
C.BRANCHING
DUP; Stack: [2]
LOAD 1; Stack: [2,2]
SUB; Stack: [1,2,2]
CALLING Fact
DUP; Stack: [1,2]
LOAD 0; Stack: [1,1,2]
EQ; Stack: [0,1,1,2]
C.BRANCHING
DUP; Stack: [1,2]
LOAD 1; Stack: [1,1,2]
SUB; Stack: [1,1,1,2]
CALLING Fact
DUP; Stack: [0,1,2]
LOAD 0; Stack: [0,0,1,2]
EQ; Stack: [0,0,0,1,2]
SKIPPING
POP; Stack: [0,1,2]
LOAD 1; Stack: [1,2]
BRANCHING TO CONTINUE
```

```
RETURNING
MUL; Stack: [1,1,2]
RETURNING
MUL; Stack: [1,2]
RETURNING
STOPPING
[2]
End Tracing
```

Of course several questions were raised, e.g. how to control the tracing selectively, what kind of diagnostics was really useful, how to report the state of the return stack, etc. but these were technicalities. The students were asked to re-implement the concept of user procedure so that the instruction **`call proc`** did not need the message argument; the **`proc`**edure should present itself. However, the main issue was: If we can add tracing in such a simplistic way, we know already how to augment the code by something yielding "side-effects", perhaps a more complex monad would be useful in other contexts.

4 Alternative Sequencing Model: CPS Codes

A major proposed modification (which could not be pursued exhaustively because our time was limited) was based on the introduction of the explicit continuation-passing style in order to sequence the primitive and the user-defined instructions. It could have been – of course – squeezed into the monadic protocol, but we preferred even to avoid that name... We abandoned the representation of **`Code`** as a list, introducing a special sequencing operator. In such a way we introduced practically the CPS paradigms, and we signalled their use in the compilation, more restricted than in [15], or in hundreds of other works, see e.g., [16–18] but fairly suggestive. We returned later to it while discussing parsing and program transformations. Here we applied the following reasoning:

Until now every instruction ***fetched*** *its successor from the code list. But almost all the time the successor is known statically, during the code generation, and we shall put it explicitly as a parameter of every instruction. We will introduce a special sequencing operator which will eliminate the necessity of the special* **`call`** *instruction, of the return, of* **`stop`***, and of the return stack.*

The code proposed will be higher-level, and may be less efficient than our basic interpreter (because of the thunk creation). Its main advantage is that it is ***very short*** *and still very clear, and it will inspire us later how to exploit the idea in other contexts. Now you should keep in your minds that instead of speaking about values, say, the stack* **`s`** *we deal with a "lifted" object* **`lift s`** *which is a function passing the value to some future treatment* **`nxt`***. Thus*

```
lift s nxt = nxt s
```

A general instruction has the form:

```
type CodeItem a = Dstack->(Dstack->a)->a
```

and we must say now a few words about the sense of this type parameter “`a`” (. . .)

The **`stkop`** operator which makes an instruction from a “pure” **`Opstack`** operator (**`type Opstack = Dstack->Dstack`**), is defined as

```
stkop :: Opstack -> CodeItem a
stkop = (lift .)
```

and the remaining data processing operators remain unchanged. The introduction of such compact combinatoric definitions as above *is* a slight shock for the students, but it is a good occasion to work on the relation between some curried definition and the type system; the principal type of **`(lift .)`** is **`(a -> b) -> a -> (b -> t) -> t`**, and the students are asked to derive it.

The next ingredient is the infix chaining operator, whose typing is another interesting issue. . . :

```
instr1 .> instr2 = \stk nxt ->
   instr1 stk (\nstk ->instr2 nstk nxt)
```

which permits the construction of such compact definitions and “main programs” as

```
cube = dup .> dup .> mul .> mul
test1 = fld 5.0 .> cube .> sld "done"
```

In order to test the code we write **`eval test1`**, where **`eval`** terminates the chain by the application of the code to something which ignores the continuation:

```
eval code = code [] const undefined
```

(and whose typing is a yet another challenging question for the students. . .)

The typical control structures are also remarkably compact. The **`ifelse`** generator is just the lifted **if-then-else**, as expected:

```
ifelse cond thcode elcode = cond .>
  \(B v : s) ->(if v then thcode else elcode) s
```

and the “simpler” conditional statement **`ifthen`** which does ***nothing*** if the condition is false, is left as an exercice to more ambitious students, who should finally deduce how to “lift nothing”:

```
ifthen cond thcode = ifelse cond thcode lift
```

Finally, the **while** loop is a cyclic structure as previously. Here it is, together with the iterative loop which constructs the sequence **`[0,1,...`*n*`]`** on the stack

```
while cond code = chunk where
 chunk = ifthen cond (code .> chunk)

nezero = dup .> ild 0 .> ne
whtest n = eval
 (ild n .> while nezero (dup .> ild 1 .> sub))
```

As usually with CPS, there is plenty of place to play with the organisation (e.g., flattening) of recursive structures, beginning with the simplest

```
factorial = ifelse nezero
  (dup .> ild 1 .> sub .> factorial .> mul)
  (pop .> ild 1)
```

etc. This was a nice experience, whose advantage w.r.t. the first contact with CPS as a program transformation tool was the possibility to test immediately everything.

5 Final Remarks

5.1 Other low-level issues

The presentation above is obviously incomplete. Several other issues have been treated, albeit often superficially.

- The global environment, storeable variables, etc. were discussed as well.
- We discussed some non-trivial optimizations, e.g., the presence of one "register", a variable threaded through the interpreted code, whose role was to represent the directly accessible stack top; stored on the **`Dstack`** only in case of necessity. This is an important issue in the compiler optimisation.
- Our machine was prepared — after some small modifications — to the implementation of coroutines and similar objects (generators, some simplistic dataflow constructs, etc.). The organisation of the control flow was not too difficult, but such entities as coroutines *must* be based on a "stateful" computation model, otherwise they are practically useless. This required a thorough re-design of the machine concepts, and could not be completed during our course.
- We duly mentioned the relation between continuations, and some techniques in logic programming, e.g., the backtracking.

In general, we found that functional languages used as the description tool of lower-level architectures are comcise and readable. Even the presentation of the garbage-collection mechanisms could be nicely done in Haskell, although obviously no serious real implementation thereof was possible.

5.2 Conclusion

For those teachers who think that functional languages are good for parsing and some other compilation techniques, it may be interesting to try to exploit the fact that functional languages are really good at modelling in general, and to apply them at much lower level of the code organisation, somehow outside the traditional functional "folklore". A functional — high-level — construction of a low-level interpreter as the target platform seems to be a fruitful idea. The design of parsers, attribute analysers, and of code generators is then easier, since the practical aspects of program semantics may be treated very coherently, the students can apply the same coding (and reasoning) style throughout the whole course.

The package (available from the author) is very short. With its aid (and some parsing utilities based on combinators, say, [1] and others) the students constructed compilers/interpreters for subsets of Scheme, and played with a procedural graphical language. We, the teacher and the students, enjoyed this work very much, and the compilation course based on lazy functional techniques should be considered as rather successful. It began as a solitary pedagogical project, since the author's colleagues had different visions of how to teach compilation, but now, when some students *who passed this course* graduated, they offered their help.

References

1. Graham Hutton, Erik Meijer, *Monadic Parser in Haskell*, J. Funct. Progr. **8**, (1998), pp. 437–444. Also: Graham Hutton, *Higher Order Functions for Parsing*, J. Funct. Programming **2**, (1992), pp. 323-343.
2. Doaitse Swierstra, Pablo Azero, *Fast, Error Correcting Parser Combinators: A Short Tutorial*, SOFSEM'99 Theory and Practice of Informatics. LNCS 1725, (1999) pp 111–129.
3. Alfred V. Aho, Ravi Seth, Jeffrey D. Ullman, *Compilers: Principles, Techniques and Tools*, Addison-Wesley, Reading, (1986).
4. Andrew W. Appel, *Modern Compiler Implementation in ML*, Cambridge University Press, (1998).
5. Simon Peyton Jones, John Hughes (editors) et al., *Haskell 98: A Non-strict, Purely Functional Language*, The language report, available from **`http://haskell.org/report`**, (2002).
6. Rinus Plasmeijer, Marko van Eekelen, *Clean version 2.0, Language Report*, University of Nijmegen and Hilt - High Level Software Tools B.V., available from **`http://www.cs.kun.nl/~clean/`**, (2001).
7. Luca Cardelli, *The Functional Abstract Machine*, AT&T Bell Lab. Tech. Report TR-107, (1983).
8. Simon L. Peyton Jones *al*, *The Implementation of Functional Programming Languages*, Prentice-Hall, (1987).
9. Rinus Plasmeijer, Marco van Eekelen, *Functional Programming and Parallel Graph Rewriting*, (1993)
10. Samuel N. Kamin *Programming Languages: An Interpreter-Based Approach*, Addison-Wesley, (1990).
11. Donald E. Knuth, *The Art of Computer Programming*, Fascicle 1: MMIX, (**`www-cs-faculty.stanford.edu/~knuth/taocp.html`**). (Copyright 1999 by Addison-Wesley).
12. James R. Bell, *Threaded Code*, CACM **16**, (1973), pp. 370–372.
13. Peter M. Kogge, *An Architectural Trail to Threaded-Code Systems*, IEEE Computer, (1982), pp. 22–32.
14. P. Wadler, *The Essence of Functional programming*, 19'th Symposium on Principles of programming Languages, Santa Fe, (1992).
15. Andrew W. Appel, *Compiling with Continuations*, Cambridge University Press, (1992).
16. D.P. Friedman, M. Wand, C.T. Haynes, *Essentials of Programming Languages*, MIT Press, (1992).
17. D.P. Friedman, C.T. Haynes, E. Kohlbecker, *Programming with Continuations*, In: *Program Transformations and Programming Environments* (ed. P. Pepper), Springer-Verlag (1985), pp. 263–274.
18. Christian Tismer, *Continuations and Stackless Python, or "How to change a paradigm of an existing program"* (1999).