COMPILING FUNCTIONAL PROGRAMMING CONSTRUCTS TO A LOGIC ENGINE

by

Harvey Abramson and Peter Lüdemann

Technical Report 86-24

November 1986

Compiling Functional Programming Constructs to a Logic Engine

Harvey Abramson[†] and Peter Lüdemann[‡]

Department of Computer Science

University of British Columbia

Vancouver, B.C., Canada V6T 1W5

Abstract

In this paper we consider how various constructs used in functional programming can be efficiently translated to code for a Prolog engine (designed by Lüdemann) similar in spirit but not identical to the Warren machine. It turns out that this Prolog engine which contains a delay mechanism designed to permit co-routining and sound implementation of negation is sufficient to handle all common functional programming constructs; indeed, such a machine has about the same efficiency as a machine designed specifically for a functional programming language. This machine has been fully implemented.

[†] Present address: Department of Computer Science, Bristol University, Bristol, England

[‡] Present address: IBM Canada Ltd., 1 Park Centre, 895 Don Mills Road, North York, Ontario, Canada M3C 1W3

Compiling Functional Programming Constructs to a Logic Engine

Harvey Abramson† and Peter Lüdemann‡
Department of Computer Science
University of British Columbia
Vancouver, B.C., Canada V6T 1W5

Abstract

In this paper we consider how various constructs used in functional programming can be efficiently translated to code for a Prolog engine (designed by Lüdemann) similar in spirit but not identical to the Warren machine. It turns out that this Prolog engine which contains a delay mechanism designed to permit co-routining and sound implementation of negation is sufficient to handle all common functional programming constructs; indeed, such a machine has about the same efficiency as a machine designed specifically for a functional programming language. This machine has been fully implemented.

1. Introduction.

There has been a considerable amount of literature devoted to the topic of combining functional programming and logic programming. We refer the reader to [DeGrLi86] for a collection of such work, and for the extensive bibliographies contained in the chapters of that book. The aspect of the endeavour that we wish to deal with in this paper is not how the two programming paradigms should be combined, but how functional programming constructs can be efficiently realized in code for a Prolog engine.

† Present address: Department of Computer Science, Bristol University, Bristol, England

‡ Present address: IBM Canada Ltd., 1 Park Centre, 895 Don Mills Road, North York, Ontario, Canada M3C 1W3

The proposals for combining the two paradigms have been roughly classified in [BoGi86] as being either *functional plus logic* or *logic plus functional*. In the former approach, invertibility, nondeterminism and unification are added to some higher order functional language; in the latter approach, first-order functions are added to a first-order logic with an equality theory. We do not take any position as to which approach is best, but since a logic machine already deals with unification, nondeterminism and invertibility, we have taken, from an implementation viewpoint, the logic plus functional approach. We believe that a language combining the paradigms can be executed on a Prolog engine (such as the Warren Abstract Machine, for example, or a different Prolog engine with a delaying and coroutining mechanism designed by Lüdemann), and in fact, that a purely functional programming language can be executed on such a machine, without losing much of the efficiency which could be gained by execution on an abstract machine specifically designed for a functional programming language.

In this paper we therefore address one of the open problems cited by [BoGi86], namely the kind of computational mechanism necessary to execute functional and logical programming constructs.

The rest of the paper is organized as follows. We consider various parameters characterizing functional programming languages, and the kinds of constructs derived from these parameter settings (e.g. applicative order evaluation vs. normal order evaluation, etc.). We next describe the architecture necessary to support various functional constructs, and show that the abstract machine to support such constructs is a subset of a Prolog engine's architecture. Finally, we show how the full Prolog engine architecture – including a delay mechanism which can be used for coroutining as well as a sound treatment of negation – encompasses all that is necessary to deal with the most general functional constructs.

2. Functional Programming background.

For illustrative purposes, we will use some of the syntax of SASL and HASL (see [Turn79], [Abra84]).

2.1 Definitions and expressions.

Function definitions are notated:

$$\begin{array}{l} f\ a_{11}\ a1_2\ \ldots\ a_{1n} = expr_1 \\ f\ a_{21}\ a_{22}\ \ldots\ a_{2n} = expr_2 \\ \quad\ldots \\ f\ a_{m1}\ a_{m2}\ \ldots\ a_{mn} = expr_m \end{array}$$

This defines the *m* clauses of an *n*-ary function f.

Expressions may be written as:

```
expr where { definitions }
```

The definitions may include function definitions, as well as definitions of the form:

```
x = 1:x
[a,b,c] = [1,2,3]
```

etc.

A definition such as:

```
s f g x = f x (g x)
```

is a "syntactically sugared" version of the lambda expression:

$s = \lambda\ f.\ \lambda\ g.\ \lambda\ x.\ f\ x\ (g\ x)$

2.2 Functional programming varieties.

The syntax above may be interpreted in different ways to derive different programming languages. We consider some of the possible interpretations of the syntax and the effects of the interpretation on the implementation of the language being defined.

2.2.1 Single argument vs. *n*-argument functions.

Some functional programming languages have only single argument functions. The single argument and single result, however, may themselves be single argument functions. Thus, if `f` is defined as an *n*-ary function, it is implemented in such a way that in an evaluation:

$f\ arg_1\ arg_2\ \ldots\ arg_n$

the implicit evaluation is:

$(\ldots(f\ arg_1)\ arg_2)\ldots\ \ arg_n)$

where the result of $(f\ arg_1)$ is a function of a single argument, etc. SASL, HASL treat *n*-ary functions this way; LISP, Scheme, etc. do not.

2.2.2 Normal vs. applicative order evaluation.

In applicative order evaluation, arguments are evaluated before evaluation of the function body; in normal order evaluation, the leftmost reduction (evaluation) is always performed until the normal form (or sometimes just head normal form) of the original expression, if it exists, is obtained. Normal order evaluation will eventually obtain the normal form (if it exists) whereas applicative order evaluation may not: if an argument's value is not needed in the body of the function, evaluation of the argument is not necessary; if the evaluation results in an infinite loop, it is disastrous. Related to normal order evaluation is "evaluation by need" which can be used to prevent repeated evaluations of the same expression: once the value of an expression is evaluated as in normal order, its value is kept for efficiency's sake and used in place of a repeated notrmal order of the same expression.

2.2.3 Lazy vs. eager evaluation.

Lazy evaluation delays evaluations until they are needed, whereas eager evaluation proceeds even though values might not be needed. One effect of lazy evaluation is to permit programming with "infinite" structures such as:

```
x = 1:x
```

With lazy evaluation, an expression such as `hd x` yields the value `1`, the tail of the list not being evaluated. SASL and HASL provide lazy evaluation; LISP, generally, does not, but permits the construction of mechanisms for lazy evaluation.

2.2.4 Lexical scoping vs. dynamic scoping.

In a language where functions are first class objects, a function must be applied in the correct environment, binding free variables in the function body. The data structure consisting of the function body and an environment is called a closure or a thunk [Inge61]. In pure functional programming languages, the closure is formed at definition time: this is called lexical scoping. Other less pure functional languages form a closure at evaluation time: this is known as dynamic scoping. Pure functional programming requires lexical scoping and is used in such languages as SASL, HASL, and Scheme. Many varieties of LISP, however, use dynamic scoping which leads to serious semantic problems, chiefly a kind of destructive assignment.

2.3 Mechanical evaluation of functional programming constructs.

The earliest programming language with a strong functional flavor was McCarthy's LISP. An evaluator (interpreter) for a functional subset of LISP was defined in LISP. LISP, however, was not purely functional, allowing destructive assignment, go to's, etc. Several years after LISP was introduced, Landin described an interpreter for Church's lambda notation, the interpreter being specified by means of an abstract machine which may be regarded as canonical [Land66].

Landin's SECD machine consists of four components:

S - stack

E - environment

C - control

D - the dump, a stack to save states

The stack is used to save intermediate results in a computation, the environment is used to bind variables in a λ-expression, the control is a λ-expression being evaluated, and the dump is a stack which saves earlier states of the machine for later restoration.

The original SECD machine adopted the strategy of applying functions to evaluated arguments, hence, was initially suitable for programming languages using applicative order. Some lambda expressions, therefore, which could be evaluated only with normal order evaluation; applicative order would result in non-terminating computations.

The "procrastinating" SECD machine, a modification of the original SECD machine allowed the postponement of evaluation of arguments until their values became necessary. In a purely functional setting, this is equivalent to normal order evaluation.The evaluation of an expression can be carried out once, and this value saved if the value of the expression is needed again – the above mentioned evaluation by need. (See [Burg75])

Another major technique for evaluating functional programming constructs was introduced by Turner in an implementation of SASL. The primitive operations of SASL are application of a function to a single argument, and the pairing or construction of lists. In this technique, variables are removed from lambda expressions yielding expressions containing only combinators and global names referring either to library or user-defined functions (from which variables of course have been removed). Evaluation of an expression then is accomplished by a combinator machine: a machine whose instructions correspond to the three fundamental combinators **S**, **K** and **I**, and to additional combinators which are not strictly necessary but are introduced to reduce the size of the generated combinator machine code. The leftmost possible reduction is performed to yield an expression's head normal form. Normal order evaluation and "lazy" evaluation of lists fall out from this and the definition of the combinator expressions. Furthermore, SASL introduced a limited kind of unification, extended later in Abramson's HASL which was specified in Prolog.

2.4 Functional programming via logic programming.

Functional programming languages may be evaluated by logic programs either by interpretation of functional expressions or by compilation of functional programs to logic program goals to be solved. The former technique was used by [Abra84] to specify HASL (an extension of Turner's SASL) in Prolog; the latter technique was used by [BoGi86] to illustrate how various functional programming constructs could be represented as Prolog goals to be solved. Other aspects of the implementation of functional programming extensions of logic programming may be found in [DeGrLi86].

Since functional programming constructs can be compiled to Prolog, and since Prolog itself may be compiled, it is possible to stop at this stage and consider the problem of implementing functional programming constructs in a logic programming language as being solved. However, more efficient handling of functional programming constructs is still possible if one compiles them not into Prolog but directly into code for a Prolog engine. In the remainder of this paper we discuss this, in the process gaining just about as much efficiency as would be possible by compiling functional programming constructs to an abstract machine designed specifically for functional program evaluations.

3. Basic Sequential Inference Machine

We will describe the machine in two stages. The basis for the design is a fairly conventional (von Neuman) register design with a single stack. We then show how constructs for lazy evaluation and combinators fit naturally into the design. The machine also supports full Prolog-like backtracking (described in an appendix). Some simplifications have been made in this presentation – a more detailed description of the machine, including justification of design decisions, is in [Lüde86].

3.1 Structure of the Inference Machine

Every object in the machine is a first class citizen with a tag:
- integer or floating point number,
- string,
- nil (denoted []),
- list element (pointers to two objects, denoted `Hd.Tl` or `[Hd|Tl]`),
- uninstantiated (or not ground) logical variable,
- reference (pointer) to another object which is automatically dereferenced whenever it is accessed,
- code segment,
- "thunk" [Inge61] (code pointer with environment).

We use structure copying rather than structure sharing since its implementation is simpler than structure sharing and efficiency is about the same [Mell82].

Each object is identified solely by its address which remains constant throughout its lifetime. New cells are allocated from the free list. Because all the cells are the same size,[3] this is as efficient as allocating and deallocating on a stack (and we do not need to distinguish between "local" and "global" variables as in WAM).

Each of the 32 registers contains either an object **address** or is flagged as being empty. When an n-ary predicate ((n–1)-ary function) is called, the caller must put the arguments in registers 0 through n–1 and save its registers on the execution stack. The called predicate must ensure that on return all registers are empty – the caller can then pop the

3 String objects point into a separate string area which is divided into segments and is compacted in a manner similar to Smalltalk-80's LOOM (large object oriented memory) [Gold83].

saved registers. "Freeing a register" simply means flagging the register as empty.

The machine's status is kept in registers:

pc: program counter containing the code segment and offset of the next instruction.

cpc: continuation program counter containing the code segment and offset of the next instruction to be executed after a return instruction.

toes: top of execution stack pointer.

In instructions, each register is annotated:

v: contains a value,

n: empty, possibly requiring the allocation of a new object,

f: contains a value which is emptied after use.

x: is empty and the value is unneeded (*v+f*).

With these annotations, the `eq`, `eqLst` and `swAVNL` instructions can copy values, test for equality and unify. Functional programing languages use logical variables only for filling in return values, so unifications with logical variables will always succeed (there is no need for a "trail").

`Eq` r_1, r_2 unifies two registers. It can also be used to move or copy registers:
`eq n1,f2` moves register 2 into register 1;
`eq n3,v4` puts a copy of register 4 into register 3;
`eq n5,'[]'` loads register 5 with nil.

`EqSkip` is like `eq` except that it skips the following instruction if the equality test succeeds (`eq` fails to the most recent choice point – see appendix for details). If the equality test fails, any *f* annotations are ignored.

`EqLst f1,n2,n3` tests for register 1 containing a list element (or, if register 1 contains a logical variable, instantiates it to a list element) with the head being put into register 2 and the tail into register 3; register 1 is then emptied. `EqLst f0,n0,n1` is valid; it replaces register 0 by the list element's head (the tail is put into register 1).

`SwAVNL` r_1, r_{Hd}, r_{Tl} jumps to one of the following four instructions depending on whether the register contains an atom, a variable (not needed for functional programming), nil, or a list element. In the case of a list element, the head and tail are put into the indicated registers. Other switch instructions exist for multi-way branching on strings or numbers but we will ignore them because they can be emulated by sequences of `eqSkips`.

`Push` *r* and `pop` *r* are used to push and pop registers on the execution stack before and after a call instruction.

The `call` instruction assumes that the argument registers are already loaded; *cpc* is pushed onto the execution stack, *pc* is copied into *cpc* and *pc* is set to the first instruction in the new code segment.

`Return` does the inverse of `call` by copying *cpc* into *pc* and popping the execution stack into *cpc* (all the registers must already be empty when a `return` is executed).

Using *cpc* this way allows the `lastCall` instruction for tail recursion optimisation (TRO) to be like a `goto` [Warr77].

3.2 Sample code

Here is how a function `p` which produces a new list by applying the function `q` to each element of a list may be written in SASL or HASL:

```
p []      = []
p (Hd:Tl) = (q Hd) : (p Tl)
```

In Prolog, this is:

```
p([], []) :- !.
p(Hd.Tl, HdX.TlX) :-
    q(Hd, HdX), !, p(Tl, TlX).
```

The cuts (!) are necessary to make this deterministic. In general, an *n*-ary function can be turned into an (*n*+1)-ary predicate by adding one parameter to hold the result. This extra parameter must always be initialised as an uninstantiated logical variable before a call and the called predicate must always instantiate it before returning.

Here is our code (comparable WAM code is in a separate Technical Report):

```
  swAVNL    f0,n0,n2 % switch on parm0
  builtin   "error"  % invalid parm
  builtin   "error"  % can't be variable
  goto      nil      % parm0=[]
lst:                 % parm0=Hd.Tl
  eqLst     f1,n1,n3 % parm1:=HdX.TlX
                     % (can't fail)
  push      f2       % save Tl
  push      f3       % save TlX
  call      'q/2'    % q(Hd,HdX): regs 0, 1
                     % are already set
  pop       n1       % arg1:=restore TlX
  pop       n0       % arg0:=restore Tl
  lastCallSelf       % p(Tl,TlX)
nil:
  eq        f1,'[]'  % result:=[]
  return
```

The `lastCallSelf` instruction has the same meaning as `goto 0` (the different opcode helps in debugging). This is a tail recursive call – recursion has been turned into iteration.

The code can also be written in a purely functional way (see appendix). The functional code is not tail recursive because of the "cons", so much more stack is needed. Typically, using predicates instead of functions results in about the same number of machine instructions. The slight amount of extra execution time required by using logical variables is more than compensated by the better opportunities for detecting tail recursion optimization (TRO).

Without loss of generality, we can use deterministic predicates instead of functions. When we say that a predicate returns a value we mean that the last parameter gets instantiated to the value returned by the equivalent function.

So far, we have treated functions as if they were compiled into Prolog. This can introduce large inefficiencies because of the more general nature of unification and the possibility of backtracking. The `eqSkip` instruction is used to avoid creating choice points. For example:

```
p [] = []
p ('a':rst) = 'x' : (p rst)
p ('b':rst) = 'y' : (p rst)
```

is compiled to:

```
  swAVNL    f0,n2,n0  % switch on parm0
  builtin   "error"   % invalid parm
  builtin   "error"   % can't be variable
  goto      nil       % parm0=[]
lst:                  % parm0=hd.rst
  eqSkip    f2,'a'    % test hd = 'a'
  goto      else
  eqLst     f1,'x',n1 % result := 'x' :
  lastCallSelf        %            p(rst)
else:
  eqSkip    f2,'b'    % test hd = 'b'
  builtin   "error"   % else: invalid parm
  eqLst     f1,'y',n1 % result := 'y' :
  lastCallSelf        %            p(rst)
nil:
  eq        f1,'[]'   % result := []
  return
```

A boolean functions returns either `true` or `false`, so calling a boolean function and testing the result can be done in a similar fashion – there is no need to create choice points.

3.3 Thunks, lazy evaluation and higher order functions

A thunk contains the address of an n-ary predicate and the values of the first m arguments ($m \leq n$). The thunk can be considered as an $(n-m)$-ary predicate. When the thunk is called, the first $n-m$ registers are moved into registers m through $n-1$ and registers 0 through $(n-m)-1$ are loaded with the values in the thunk.

In applicative order, the code for `plus(1,2,Z)` is:

```
  eq    n0,'1'
  eq    n1,'2'
  push  n2           % Z
  call  'plus/3'
  pop   nZ           % Z
```

In normal order, the code is transformed to compute

```
z where {
    z=p1(2),
    p1=pluss(1),
    pluss=λxλy.x+y }
```

Plus 1 2 is reduced to *p1 2* (with $p1(x) = \lambda x\{x+1\}$) and finally to *3*.

On our machine, we compile the functional expression

`z=p1(2) where p1=pluss(1)`

to the predicate calls

```
?- pluss(1,P1),
   P1(2,Z).
```

which then becomes:

```
eq      f0,'1'          % argument
push    n1              % P1
call    'pluss/2'
pop     n2              % P1
eq      n0,'2'          % argument
push    n1              % Z
callThunk f2/2          % call P1(2,Z)
pop     nZ              % Z
```

`Pluss` is a 1-ary function (2-ary predicate) `plus`:

`pluss(X,Z) :- thunk(plus(X)/2,Z).`

which is compiled to:

```
mkThunk f1,1,'plus/2' % Z:=thunk(...)
return
```

`MkThunk f1,1,'plus/2'` means that register 1 is unified to a thunk pointing to `plus/2`, the first argument already having been set in register 0. `MkThunk` is like a call, so it frees all the arguments (in this case, register 0 is marked as empty after its value is saved in the thunk). The third operand to `mkThunk` may be a register so that we can make a thunk from a thunk. `MkThunk` turn atomic objects can be made into thunks by using the "=" predicate (defined `X=X`).

Whenever an `eq`, `eqSkip`, `eqLst` or `swAVNL` instruction requires a value, the thunk is "woken up" and evaluated. The non-empty registers are put into a new thunk (pointing to the current instruction) which is pushed onto the execution stack. This "suspends" the currently executing predicate (unification is repeatable, so we do not need to store any other information to aid in restarting the suspended instruction). The registers are then loaded from the woken thunk and execution proceeds within it until its last `return` instruction is reached. Normally, when a `return` is executed, the top element on the stack is a code segment but in this case it is a thunk so the registers are restored from the thunk (recall that all the registers must be empty before a return) and execution resumes where it was earlier suspended.

Because a thunk saves the registers, pure normal order execution does not need to push and pop registers on the execution stack. But the `mkThunk` and `callThunk` instructions are quite expensive, so using thunks and normal order evaluation is less efficient than applicative order evaluation. There is, however, a compensating optimisation possible. Rather than using a logical variable to return the value of a predicate (with the associated `push` and `pop`), the value can be returned directly in a register:

```
eq     f0,'1'      % argument
call   'pluss/2'   % result in reg 31
eq     n0,'2'      % argument
callThunk f31/2
eq     nZ, f31     % z = result
```

A simple (not very useful) example of delayed evaluation:

```
intsFrom m i  =  m : (intsFrom (m+i) i)
```

which returns a list of every *i*th integer starting at *m*. We can sum all the even integers up to *n* by:

```
sumLim (x:r) n = if x > n
                 then 0
                 else x + (sumLim r n)
sumEven n = sumLim (intsFrom 2 2) n
```

At each step of `sumLim`, a new list element `(x:r)` is required. This wakes up the `intsFrom` thunk which makes a list element containing the next number and a thunk for generating the next list element. Eventually `sumLim` reaches the limit `n` and no more elements are needed.

Naïvely translating this to producer-consumer co-routines:

```
intsFrom(M, I, M.L) :- M2 is M + I,
                       intsFrom(M2, I, L).
sumLim(X?.R, N, 0) :- X > N.
sumLim(X?.R, N, S) :- sumLim(R, N, S2),
                        S is X + S.
sumEven(N, S) :- sumLim(L, N, S),
                 intsFrom(2, 2, L).
```

where the ? notations mean that the predicate must delay until the value is instantiated. A delay is implemented by using a `swAVNL` or `varGoto` instruction to detect that a value is uninstantiated – a `delay` *r* instruction suspends the predicate by saving a thunk with all the non-empty registers on the delay list associated with the variable and then executing a `return`. When the variable becomes instantiated, all associated delayed predicates are made eligible for resumption (the current predicate is suspended and all the delayed predicates are pushed onto the execution stack except for the oldest which is resumed).

The above program has a subtle error. `SumLim` is first started and immediately delays on `L`. `IntsFrom` is then entered – it instantiates the first element of `L`. This wakens `sumLim` (and suspends `intsFrom`) which calls itself and then delays on the next element of `L`. Eventually, `sumLim` terminates but `intsFrom` continues generating elements in the list even though they are not needed. Here is a correct version

```
intsBetween(M, N, I, M.L) :- M ≤ N,
    M2 is M + I,
    intsBetween(M2, N, I, L).
intsBetween(M, N, I, []) :- M > N.
sum([]?, 0).
sum(X?.R, S) :- sum(R S2), S is X + S.
sumEven(N, S) :- sum(L, S),
                 intsBetween(2, N, 2, L).
```

This should not be taken to mean that thunks are more powerful than delayed predicates. In some cases, delayed predicates are easier to use because they allow more than one predicate to delay on a single variable. Predicates also backtrack.

The main difference between the two concepts is in how they handle an "infinite" list. For the computation to terminate, the list must be made finite. Thunks do this by eventually leaving the tail uncomputed; delayed predicates eventually instantiate the tail to nil. In both cases, the list need not actually exist (it is, after all, just a communication channel) – reference counting (if used) ensures that only the current element exists, all other elements being deallocated as soon as they are finished with.

3.4 Equality: `is` and =

Standard Prolog has a simple syntactic equality theory given by the predicate "=" (defined `X=X`). Another kind of equality is provided by the builtin predicate `is`. This can be considered to be defined:

```
X is Y :- atomic(X), X=Y.
X is F :- F =.. Fname.FArgs,
          isArgs(FArgs, FAx),
          append(Fname.FAx, [X],FL2),
          F2 =.. FL2,
          call(F2).
isArgs([], []).
isArgs(H.T, H2.T2) :- H2 is H,
                      isArgs(T, T2).
+(X,Y,Z) :- "Z:=X+Y". % builtin predicate
-(X,Y,Z) :- "Z:=X-Y".
  etc.
```

The predicates `is`, `+`, `–`, etc. delay if any of their parameters are not sufficiently instantiated.

The first clause is a slight extension of the usual definition (removing the "only numbers" restriction). The second clause expects the right-hand parameter to be of the form `F(A1,A2,…,An)`: it evaluates arguments `A1` through `An` (recursively using `is`) to produce `B1` through `Bn`, then computes `X` by calling `F(B1,B2,…,Bn,X)`. If we assume that `call`, `=..`, `+`, `–`, etc. are defined by an infinite number of rules, `is` is definable in first-order logic.

This definition of `is` gives a kind of semantic equality. For convenience, let us introduce the notation `p({F})` to mean `Fv is F, p(Fv)` (`{F}` is pronounced "evaluate `F`"). Using this for factorial:

```
f(0, 1).
f(N, {N * f({N-1})}).
```

The second clause is an abbreviation for:

```
f(N, NF) :- NF is N*F, Nsub is N-1,
        f(Nsub, F).
```

This definition depends on arithmetic predicates delaying when their arguments are insufficiently instantiated. Although tail recursive, it is much less efficient than the non tail recursive version because of the overhead of processing the delays.

3.5 Combinators

The implementation described so far corresponds to a lazy SECD machine [Hend80]. But it can also be used as a combinator machine [Turn79]. Thunks and code segments, being "first class" objects, can be passed as arguments and be returned as values.

A thunk is evaluated only when required by unification or by a builtin. When a thunk is evaluated, it may return a structure containing another thunk which itself requires evaluation when its value is needed.

The traditional combinators can be restated as predicates. Given the definitions:

```
I = λx.x
K = λx λy.x
S = λf λg λx.(f x)(g x)
```

we get the following predicate definitions (using `thunk/2` as described earlier):

```
comb_I(F,F).
comb_K(F,Z)  :- thunk(comb_kk(F)/2,Z).
comb_S(F,Z)  :- thunk(comb_ss(F)/3,Z).
```

with the auxilliary predicates:

```
comb_kk(X,Y,X).
comb_ss(F,G,Z)  :-
    thunk(comb_sss(F,G)/4,Z).
comb_sss(F,G,X)  :-
    F(X,Z1), G(X,Z2), Z1(Z2,Z).
```

Here is a computation using predicates `pluss` (1-ary addition) and `succ` (successor):

```
a = S plus succ 3
```

is compiled to the predicate definitions and calls

```
pluss(X,Z) :- thunk(plus(X)/2, Z).
succ(X,Z)  :- Z is X+1.
?- comb_s(pluss,A1), A1(succ,A2), A2(3,A).
```

resulting in:

```
A1= thunk comb_ss: parm0=pluss
A2= thunk comb_sss: parm0=pluss,parm1=succ
A= Z where pluss(3,Z1), succ(3,Z2), Z1(Z2,Z)
     leading to: Z1= thunk plus: parm0=3
                 Z2= 4
     and finally (when evaluation of Z is forced all the way):
                 Z = A = 7
```

When code is written using the combinators **K**, **I** and **S**, the predicate calls to `pred_K`, `pred_I` and `pred_S` look just like normal predicate calls. When they are executed, the returned values are thunks which can be handled like any other objects. They will be evaluated only when needed and only as much as needed, possibly returning structures containing other thunks.

This definition of combinators requires that thunks be created only for the basic combinators **K**, **I** and **S** and of course for any other combinators (such as **B**, **C**, etc.) which are introduced to control the size of the compiled code (from which variables have been removed). All other definitions are done without reference to thunks.

The combinator machine uses a subset of the inference machine's instructions in a small number of ways, so a number of optimisations are possible. Call instructions invariably have two arguments, so the "load arguments, push new variable for result, call, pop result" sequence can be optimised into one instruction. By adding these optimisations to the machine, the programmer is given the flexibility of efficiently using either the lambda machine or the combinator machine models of functional computation – or mixing them – as the problem requires.

In Turner's combinator machine, combinator reduction occurs in-place. Turner observed that when an expression occurs within a function, it will only be evaluated once, the first time it is needed. Because logical variables share, we get the same effect with our predicate translations of combinators. In the combinator machine, values do not actually get replaced – rather, new values are computed and the replaced – rather, new values are computed and the old values are abandonded (and eventually garbage collected).

4. Conclusions

We have described a new logic inference machine which can efficiently implement functional programming. It can directly execute functional constructs or execute deterministic predicates which are equivalent to functional constructs. The latter is often preferable because it offers more scope for tail recursion optimization.

The machine has been implemented and runs at about the same speed as other logic inference machines (such as WAM implementations). Functional programs run efficiently because they are compiled to a true functional subset of the inference machine – there is no overhead for creating unnecessary choice points as would happen if we simply translated from functional constructs to Prolog.

Having designed a machine which can efficiently handle both functional and logic programs, the problem remains of designing a single integrated programming language which can take advantage of the many possibilities offered.

References

[Abra84] Abramson, H.: *A Prological Definition of HASL a Purely Functional Language with Unification Based Conditional Binding Expressions.* New Generation Computing, 2(1984) 3-35.

[BoGi86] Bosco, P.G. and Giovanetti, E.: *IDEAL: An Ideal DEductive Applicative Language*, Proceedings IEEE 1986 Symposium on Logic Programming, pp. 89-94.

[Burg75] Burge, W.H.: *Recursive Programming Techniques*, Addison-Wesley, 1975.

[ClMcC84] Clark, K.L. and McCabe, F.G.: *micro-PROLOG: Programming in Logic*, Prentice-Hall, 1984.

[DeGrLi86] DeGroot, D. and Lindstrom, G. *Logic Programming Functions, Relations and Equations* , Prentice-Hall, 1986.

[GLLO85] Gabriel, Lindholm, Lusk, Overbeek: *A Tutorial on the Warren Abstract Machine for Computational Logic.* Argonne National Laboratory Report ANL-84-84.

[Gold83] Goldberg, A.: *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, 1983.

[Inge61] Ingerman, P.Z.: *Thunks – A way of compiling procedure statements with some comments on procedure declarations*, Comm. A.C.M. 4, 1, pp. 55-58.

[Kowa79] Kowalski, R.: *Logic for Problem Solving.* Elsevier North Holland, 1979.

[Hend80] Henderson, P.: *Functional Programming: Application and Implementation.* Prentice-Hall, 1980.

[Land66] Landin, P.J.: *An abstract machine for designers of computing languages.* Proc. IFIP Congress 65, Vol. 2, Washington: Spartan Books, 1966, pp. 438-439.

[Lüde86] Lüdemann, P.: *Efficiently Implementing Pure Prolog* or: *Not "YAWAM"*, University of British Columbia, Department of Computer Science, Technical Report 86-25. Submitted to Fourth International Logic Programming Conference.

[Mell82] Mellish, C.S.: *An Alternative to Structure Sharing in the Implementation of a Prolog Interpreter.* In *Logic Programming*, Clark, K.L. and Tärnlund, S-A. (ed.), Academic Press, 1982.

[Nais85] Naish, L: *Negation and Control in Prolog.* Ph.D. Thesis, University of Melbourne.

[Turn79] Turner, D.A.: *A New Implementation Technique for Applicative Languages*, Software Practice and Expirence, 9(1979), 31-49.

[Warr77] Warren, D.H.D.: *Implementing Prolog – Compiling Predicate Logic Programs.* Technical Reports 39 and 40, Department of Artificial Intelligence, University of Edinburgh.

[Warr83] Warren, D.H.D.: *An Abstract Prolog Instruction Set.* SRI Technical Note 309.

Appendix: backtracking

The deterministic inference machine is easily extended to allow backtracking by adding two stacks: a backtrack (choice point) stack and a reset stack ("trail"). The backtrack stack can be embedded in the execution stack as in WAM; we have separated the stacks for simplicity of explanation.

When unification instantiates a value cell which is older than the top choice frame, the cell's address is put on the reset stack (if the cell is newer, backtracking would simply free it, so there is no need to record it). The age number of an uninstantiated variable is the depth of the choice stack when the variable was created, so an entry is pushed onto the reset stack only if the cell's age number is less than the depth of the choice stack.[1] Deterministic predicates will not create reset stack entries because such predicates do not create choice points.

Wherever backtracking is possible, choice points must be created on the backtrack stack using the `mkCh` instruction. Each choice point frame contains sufficient information to reset the machine to the state it was in when the `mkCh` was executed:

values of all non-empty registers,
value of *cpc*,
top of the execution stack and reset stacks,
failure instruction address.

When failure occurs, by a unification failing or by an explicit `fail` instruction, all registers are emptied and the top choice point frame is used to fill the registers. The reset and execution stacks are popped to what they were when the choice point was created. As the reset stack is popped, its entries are used to reset objects to uninstantiated. Execution then resumes at the failure instruction address.

When backtracking occurs, the execution stack must be restored to what it was when the choice entry was made. This means that a return instruction may not pop the execution stack if a choice entry needs it – the choice entry "protects" the entry in the execution stack [GLLO85]. The *toes* value in the top choice point frame is used to determine whether or not the execution stack can be popped. Each entry in the execution stack has a back pointer to the previous frame, skipping frames which are protected by the choice stack. If only deterministic predicates are executed, nothing is put onto the choice stack and the execution stack grows and shrinks just like the execution stack in a conventional machine (Algol, Pascal, etc.).

When a predicate delays it must also be recorded on the reset stack so that it can be removed from the delay list upon backtracking – when a delayed predicate is woken, it is recorded a second time on the reset stack so that backtracking can put it back on the delay list. An optimisation similar to TRO is performed: if no choice point has been created since the original delay entry was created, both entries are removed from the reset stack (shuffling the stack down if necessary).

Deterministic predicates run slightly slower on the full backtracking machine than on a purely deterministic machine. There are three overheads:

- making choice points rather than just branching to a failure address for *if-then-else* (that is, using the `mkCh` instruction instead of switch instructions).
- testing whether or not an instantiation should push an entry onto the reset stack (for deterministic execution, nothing will ever be pushed).
- recording delay information on the reset stack.

The first item can be avoided by a smart compiler. We can avoid recording delay information on the reset stack by having a "set deterministic mode" instruction which causes such information to be not recorded.

1 A similar technique is used in WAM except that the relative positions on the stack are used rather than age numbers – "global" cells are considered to be older than "local" cells.

Appendix: pure functional code

For comparison, here is the pure function version of the function p which applies q to each element of a list. The result is returned in register 1 which is not pre-initialised to be a logical variable (actually, register 0 should be used to make higher order functions easier to implement – doing this would require more machine instructions for moving values into the correct registers):

```
  swAVNL   f0,n0,n2 % switch on parm0
  builtin  "error"  % invalid parm
  builtin  "error"  % can't be variable
  goto     nil      % parm0=[]
lst:                % parm0=Hd.Tl
  push     f2       % save Tl
                    % arg0: already set
  call     'q/1'    % reg1:=q(Hd)
  pop      n0       % restore Tl
  push     f1       % save q(Hd)
  call     'p/1'    % r1:=p(Tl)
  pop      n2       % restore q(Hd)
  eqLst    n0,f2,f1 % result:=q(Hd).p(Tl)
                    % (can't fail)
  return
nil:
  eq       n1,'[]'  % result:=[]
  return
```

This code is one instruction longer than the code which uses logical variables (because of the `return` instruction). The functional version builds the return value more efficiently using `eqLst n0,f2,f1` – the predicate version does it by `eqLst f1,n1,n3` and then filling in the head and tail. Thus, the TRO in the predicate has a price: slightly slower construction of the result and one extra level of indirection (using "reference" objects).