ACM Sigarch *Computer Architecture News* **22**,1 (March 1994), 34-43.

Linear Logic and Permutation Stacks— The Forth Shall Be First

Henry G. Baker

Nimble Computer Corporation
16231 Meadow Ridge Way, Encino, CA 91436 (818) 986-1436 (818) 986-1360 (FAX)

This material is based upon work supported by the National Science Foundation under Grant No. III-9261682.

Girard's linear logic can be used to model programming languages in which each bound variable name has exactly one "occurrence"—i.e., no variable can have implicit "fan-out"; multiple uses require explicit duplication. Among other nice properties, "linear" languages need no garbage collector, yet have no dangling reference problems. We show a natural equivalence between a "linear" programming language and a stack machine in which the top items can undergo arbitrary permutations. Such permutation stack machines can be considered combinator abstractions of Moore's *Forth* programming language.

THE "FORTRAN FALLACY"

For 40 years, programmers have tried to utilize mathematical expressions to program computers. Indeed, "FORTRAN" is a contraction of "FORmula TRANslator", because Fortran's selling point was its close relationship to mathematical formulae. For example, one solution of a quadratic equation $x = \frac{-b+\sqrt{b^2-4ac}}{2a}$ in Fortran is `X = (-B + SQRT(B**2 - 4*A*C))/(2*A).`

Although this one-dimensional form is not as pretty as the mathematician's two-dimensional form, the relationship is clear, and user can look forward to Fortran-0X, which we expect will utilize 2-D input notation pioneered (on punched cards!) by NASA's HAL/S language [Ryer78].

Unfortunately, after 40 years of hard work, computer scientists have not been that successful at efficiently exploiting this mathematical expression metaphor. Furthermore, the single-minded pursuit of this goal has blinded us to the reason for pursuing it in the first place—to find a simple, easy-to-understand metaphor that can be used to efficiently program a variety of applications on a variety of computers.

Fortran has been quite naive about the nature of mathematical expressions, and Fortran novitiates learn to "put that mathematical metaphor up on the shelf next to the Easter Bunny", and use a more implementation-oriented model which includes "storage maps", "by-reference parameters", and the like.

Functional programming languages [Backus78] [Hudak89] [Hughes89] have gone the furthest with the mathematical expression metaphor, and have been extended to cover many programming tasks. Unfortunately, functional programs have been considered "inefficient", and their higher-order functions, "lazy evaluation" and "multiple-return continuations" leave many cold.

The "standard", non-functional programming languages like Fortran, Ada, and C are the bastard progeny of the coupling between a pseudo-mathematical notation and a von Neumann-style random access memory (RAM). *There has never been a simple or particularly effective mathematical model to aid in the deep understanding or compiling of programs in these languages*. A modern optimizing Fortran compiler is still a baroque technical *tour de force,* yet it has difficulty understanding simple Fortran programs. The real Achilles' heel for these languages, however, is their innate inability to deal with parallel or distributed computation. Neither the mathematical expression metaphor nor the von Neumann RAM is up to this task.

"The Emperor has no clothes", you say. "Cut the Gordion knot", you say. "This many smart computer scientists can't be wrong," we say, so the problem—as posed—must be essentially insoluble. In classical logical fashion, computer science has achieved a proof by contradiction. It has proposed the equivalence "programs ≡ mathematical expressions", and has derived a contradiction. Perhaps it is time to move on to the next theorem.

Simula [Dahl66] started a revolution in computer languages which is not yet finished. It proposed the equivalence "programs ≡ physical objects with behavior", but forgot to throw away the previous equivalence. *Smalltalk* [Goldberg83] picked up the ball, but the mathematical expressionists then converted the metaphor of physical objects into polymorphic mathematical type systems—i.e., C++ [Stroustrup86]. The elegance of the mechanical metaphor was thus buried underneath mathematical mysticism.

Linear logic [Girard87] can be viewed as the latest attempt to bring back the physical object metaphor, but stripped of its polymorphic pretensions. For the first time in 50 years of computer science, a metaphor of programming has been proposed that most people can relate to—objects have true identity, and *objects are conserved.* As in the real world, *an object cannot be copied or destroyed* without first filling out a lot of forms, but on the other hand, *the transmission of objects is relatively painless*. An object is localized in space, and can move from place to place. (Only computer-literate people must be told that the transmission of these objects does *not* create copies.) Linear logic finally makes precise the high school notion that a function is like a box into which one puts argument values and receives result values, and that a truly "functional" box does not remember its previous arguments or results.

The mathematical expression metaphor must be sacrificed to make way for this linear/conservative object-oriented metaphor, which—as we have seen—is not a great loss, as computer programs today deal with non-mathematical objects most of the time.[1] The few remaining mathematical expressions—e.g., the quadratic formula—can be computed, but require a more process-oriented description. In such a description, we must first make copies of `b` and `a`, since they are both used twice, and then we can compute the result. The requirement for making explicit copies of `b` and `a` is obvious to any 12-year-old, but computer scientists have spent 40 years eliminating this one trivial task while greatly increasing the costs for everything else.

A linear logic language achieves its elegance through a starkly simple rule—a bound variable name can be "used" only once. Thus, variable *reads are destructive* and hence variables are "read-once". Any attempt to use a name twice (or not at all) is flagged by the compiler. A use of the variable name as an argument in a function call means that the object referred to by the name has been given to the function. Unless the function returns the object as one of its results, the object is gone, and cannot be referenced by the caller. Therefore, many operations on linear objects have the policy of returning them when they are done—e.g., the `length` function returns the length of a list as well as the list itself. A function like "+", which accepts two values and returns their sum, can be thought of as "consuming" its argument values and constructing a result value. Not only is this metaphor appealing, but it can be very efficient in practise—e.g., in multiple-precision arithmetic, the storage utilized by the arguments to "+" can be reused to construct the result.

A linear logic computer language avoids the need for garbage collection by explicit construction and destruction of objects. The beloved "constructors" and "destructors" of C++ can therefore be used, as before. However, the "dangling reference problem" cannot happen in a linear language because 1) the only name occurrence for an object is used to invoke its destructor, and 2) the destructor doesn't return the object. Furthermore, if an object does not wish to be copied, the programmer cannot obtain a copy; thus, linearity achieves the goals that "limited" types in the Ada programming language [AdaLRM83] [Baker91SP] [Baker93Steal] sought in vain.

A linear logic language avoids the need for most synchronization because there is only one "path" to an object at any given time, and hence only one operation can be performed on the object at a time. The calling program cannot even talk about the object while an operation on it is being performed, but must wait for the object to be returned. On the other hand, complex subexpressions can always be computed in parallel, because there is no possibility of read or write conflict between sub-expressions (linearity allows the environment to be *factored* into disjoint factors that bind the names in each subexpression.). A linear language thus makes explicit any "space-time" tradeoff: space can be minimized by always working with only one copy of an object, and performing each operation serially, while time can be minimized by duplicating the object at some cost in space.

LINEAR LISP AND PERMUTATION STACK MACHINES

Linear Lisp is a "linear" style of Lisp—i.e., every bound name is referenced exactly once. Thus, each function parameter occurs just once, as do the names introduced via other binding constructs—e.g., `let`, `let*`, etc.

Linear Lisp requires the programmer to make explicit any duplication and deletion, but he is repaid through better error checking during compilation and better utilization of resources (time, space) at run-time. Furthermore, distributed and/or parallel execution becomes quite palatable in *Linear* Lisp.

We demonstrate the natural mapping of Linear Lisp onto a permutation stack machine with some examples. The `identity` function is already linear:

```
(defun identity (x) x)
```

Linear Lisp is implemented with a stack on top of which all argument values are placed and all result values are returned. This stack is "unframed"—i.e., there is no Algol-like frame pointer. The elimination of the frame pointer simplifies the model and eliminates the cost of building and disassembling stack frames. To call the `identity` function, an argument is placed on the top of the stack, the `identity` function is called, it *moves* its argument to the top of the stack as its result, and the function then returns. Of course, as the argument is already on the top of the stack, no movement is required, so there is *no* code "[]" actually needed for the `identity` function.

We now consider a function which always returns the constant `5`:

```
(defun five (x) x 5)                    ; x is explicitly killed.
```

`five` is called with its argument on the top of the stack, but it must be destroyed before `5` can be pushed onto the stack. The argument is destroyed by a `drop` operation, which pops the top item and destroys it. The primitive constant `5` is then copied and pushed to finish the `five` function: [drop '5].

`square` requires *two* uses of its argument. A second copy can be obtained by use of the `dup` operation, which accepts one argument and returns *two* values—i.e., two copies of its argument. The `square` function follows:

```
(defun square (x)
  (let* ((x x-prime (dup x)))  ; use Dylan syntax [Shalit92].
    (* x x-prime)))
```

Although the source code for `square` is a bit long, its implementation on our stack machine is very efficient. One argument is passed to `square` on the

[1]Even mathematical APL dispensed with traditional "operator precedences" when Iverson discovered that 1) most programs dealt with very few numerical expressions, and 2) people couldn't remember the precedence table.

stack, `dup` replaces this value by two copies on the top of the stack, and `*` then multiplies them together.[2] The code for `square` is thus "[dup *]".

The quadratic equation function is as follows:

```
(defun quadratic (a b c)                    ; stack is now ...abc
  (let* ((a a-prime (dup a)) (b b-prime (dup b)))3
    (/ (+ (- b-prime) (sqrt (- (square b) (* 4 (* a c)))))
       (* 2 a-prime))))
```

`quadratic` requires variable *movement* on the stack. We must duplicate `a`, but `a` is not on top of the stack. Therefore, we first permute the top three items of the stack to get `a` on top, and then duplicate `a`. The stack then looks like `...bcaa'`. We must then duplicate `b`, but `b` is not on top of the stack. Therefore, we permute the top *four* items to get `b` on top, and then duplicate `b`. The stack then looks like `...caa'bb'`. The next operation negates `b'`, which we arranged to be on the top of the stack. The next operation after that permutes `b` to the top, so that `square` can be called. We continue in this fashion, alternately permuting the stack and calling functions. The code for the quadratic function is therefore: [roll3 dup roll4 dup neg roll2 square '4 roll5 roll6 * * - sqrt + '2 roll3 * /]. We can see in operation the basic variable accessing mechanism `roll`<*n*>, which permutes the top *n* items of the stack by moving the *n*'th item to the top and pushing the other *n*-1 items down by one. A compiler can easily keep track of where everything is.

(The reader may feel that permuting an item to the top of the stack simply for the purpose of copying it is wasteful, as nearly all stack machines have the ability to *copy* an item directly to the top. We believe this intuition to be incorrect. First, while it may be easy to copy objects like small integers from deep in the stack, it is unlikely that the duplication operation for an arbitrary precision integer or an abstract data type is a primitive operation, and when a non-primitive duplication function must be called, the argument and results should be on top of the stack. Second, a key assumption of linear languages is that permuting items to the top of the stack is relatively inexpensive (relative to copying or destroying them), and hence the percolation of the second copy *down* after the `dup` will happen *automatically*, as a side-effect of rolling the other items *up* to the top of the stack.)

Every iterative/recursive function requires some sort of conditional execution. Conditional expressions—e.g., `if`-expressions—in Linear Lisp require sophistication beyond the simple linear rules we have considered so far. Since only one "arm" of the conditional can be executed at any given time, we relax the "one-occurrence" linearity condition to allow a reference in both arms, so long as they are not executed concurrently or speculatively. There is an occurrence in one arm *if and only if* there is an occurrence in the other.

The proper treatment of the boolean expression part of an `if`-expression requires more sophistication. Strict linearity requires that any name used in the boolean part be counted as an occurrence. However, many predicates are "shallow", in that they examine only a small (i.e., shallow) portion of their arguments (e.g., `null`, `zerop`), and therefore a relaxed policy is required. We have not settled on the best syntax to solve this problem, and currently use several `if`-like expressions: `if-atom`, `if-null`, `if-zerop`, etc. These `if`-like expressions require that the boolean part be a simple name, which does not count towards the "occur-once" linearity condition. This modified rule allows a shallow condition to be tested, and the name to be reused within the arms of the conditional.[4] The predicate could alternatively return *two* values: the truth value (on top of the stack) and its unmodified argument(s).

The absolute value function `abs` demonstrates conditional execution:

```
(defun abs (x)
  (let* ((x truth (minusp2 x)))      ; leave x & truth value.
    (if truth (- x) x)))
```

A more perspicuous version is:

```
(defun abs (x)
  (if-minusp x (- x) x))     ; check x, but don't consume it.
```

The code for `abs` is simply [minusp2 [neg] [] ifelse].

To use Lisp-like cons cells with their car and cdr components, we need a mechanism to *linearly* extract these components, since any use of `(car x)` precludes the use of `(cdr x)`, and vice versa, due to the linearity of `x`. We therefore utilize a function `carcdr`, which takes a cons cell and returns both components. The `carcdr` function is the stack machine *inverse* of `cons`. Using these constructs, we can program the `append` function:

```
(defun append (x y)                ; concatenate lists x and y.
  (if-atom x (progn x y)         ; examine x, kill x, return y.
    (let* ((carx cdrx (carcdr x)))
      (cons carx (append cdrx y)))))
```

`append` is [roll2 atom2 [drop] [carcdr roll3 append cons] ifelse].

Decomposing list structure is common in Linear Lisp, that we provide a "destructuring let" form `dlet*` in which patterns with variable names are matched/unified against a value. Since `dlet*` is linear, `dlet*` patterns

---

[2]We utilize the Postscript™ convention that the first argument is pushed first, and is therefore deepest in the stack—i.e., the stack grows to the right.

[3]This heavy use of explicit copying cries for `'` as a non-quoting character.

[4]*Typestates* [Strom83] can check linearity in complex control structures.

consume the portions of the values that they match—i.e., the parts that are not bound to new names in the process of matching. Of course, linearity requires that a pattern bind a name only once. We can also use a *backquote* list *composing* syntax [Steele90]. A prettier version of `append` follows:

```
(defun append (x y)
  (if-atom x (progn x y)
    (dlet* (((carx . cdrx) x))
      `(,carx ,@(append cdrx y)))))
```

In these `append` functions, we have swept something important under the rug—the code for `append` is itself non-linear, because it references itself in its own body! This is a problem with all iterative and recursive constructs, and we have the same solution for this problem that the lambda calculus has—the Y combinator (and various optimizations of it), which does "lazy" duplication. In other words, we abstract the `append` call out of its own code, and pass it to the kernel of `append` as another argument (the `funcall` special form takes its first argument from the *top* of the stack):

```
(defun append-kernel (x y f) ; f will be append-kernel, itself
  (if-atom x (progn x f y)
    (let* ((carx cdrx (carcdr x)) (f f-prime (dup f)))
      (cons carx (funcall f-prime cdrx y f)))))
```

The code for `append-kernel` is: [roll3 atom2 [drop drop] [carcdr roll4-2 dup funcall-3 check-1 cons] ifelse]. The primitive roll4-2 rolls/rotates the top 4 items of the stack by 2, and is equivalent to [roll4, roll4].

We note that the sequence [funcall-3 check-1] calls a function with 3 arguments and 1 result. The information about the number of arguments and results is used to check for the case of a function which is called with the wrong number of arguments or results. The implementation of this may be a separate implicitly addressed register which keeps track of the number of arguments passed on a call and the number of results on a return. Such numbers could easily be passed on the stack, but this would be inefficient for the vast majority of calls which have fixed numbers of arguments and returned values. To check for 3 arguments to `append-kernel`, use check-3.

```
(defun append (x y)
  (let* ((f f-prime (dup #'append-kernel)))
    (funcall f-prime x y f)))
```

The code for `append` itself is now: [#'append-kernel dup funcall-3].

Using these same ideas, the program for `factorial` is:

```
(defun fact (n f)
  (if-zerop n (progn f (1+ n))
    (let* ((n n-prime (dup n)) (f f-prime (dup f)))
      (* n (funcall f-prime (1- n-prime) f)))))
```

```
(defun factorial (n)
  (let* ((f f-prime (dup #'fact)))
    (funcall f-prime n f)))
```

The code for `fact`: [roll2 zerop2 [roll2 drop 1+] [dup roll3 dup roll3 1- roll3-2 funcall-2 *] ifelse]. The code for `factorial` is [#'fact dup funcall-2].

We now examine an "iterative" factorial function. An *iterative* factorial carries all of its "state" in its arguments, and can therefore make all recursive calls "tail-recursive". Tail-recursive calls are interesting, because they do not require unbounded amounts of either argument stack or return stack space. Furthermore, the complexity of a tail-recursive loop function is independent of the number of its arguments. Below is a linear iterative factorial function.

```
(defun ifact1 (r n f)
  (if-zerop n (progn n f r)
    (let* ((n n-prime (dup n)) (r (* n-prime r))
           (n (1- n)) (f f-prime (dup f)))
      (funcall f-prime r n f))))
(defun ifactorial (n)
  (let* ((f f-prime (dup #'ifact1)))
    (funcall f-prime 1 n f)))
```

The code for `ifact1`: [roll2 zerop2 [drop drop] [dup roll4 * roll2 1- roll3 dup funcall-3] ifelse]. `ifactorial` is ['1 roll2 #'ifact1 dup funcall-3].

This iterative factorial is interesting because 1) its tail-recursive property can be trivially recognized by syntactic means, and 2) it can be implemented in a particularly efficient manner. In fact, since `iterative-fact1` references no "free variables", no closures must be allocated or destroyed during its execution. The entire state of `iterative-fact1` lives on the stack, and consists of the values of the variables `r`, `n`, and `f`. The computations have the effect of permuting these values, but the stack as a whole neither expands or contracts. Finally, the permutation of these items necessary to execute the next iteration is automatically achieved; no special tail-recursion optimization is required [Steele78] [Hanson90]! In short, the execution of tail-recursive loops on our permutation stack machine is not only just as efficient as that achieved with a special iteration construct, it is *exactly the same as* that achieved with the special iteration construct, but without "syntactic sugar".

We eventually get tired of programming each recursive function with its own driver function and its own function duplication code, in which case we would like to program a true Y combinator [Gabriel88]. This Y combinator incorporates all of the lazy duplication machinery necessary to implement iteration and recursion in a linear language. In order to utilize Y, we will use a slightly different form of recursive kernel, in which the function is abstracted separately. Consider again `factorial`:

```
(defun fact-kernel (f)
  #'(lambda (n)
      (if-zerop n (progn n f 1)
        (let* ((n n-prime (dup n)))
          (* n (funcall f (1- n-prime)))))))
```

A *closure* must be compiled within `fact-kernel`. The anonymous inner function of `fact-kernel` is the function which will receive the actual numerical argument. However, this inner function has a free variable `f` which must be carried along with the anonymous function so that when that function is called, the value of this free variable can be obtained. The structure which carries this free value is a *closure*, and consists of a 3-element vector whose last element is the indicator 'funarg, whose second element is the anonymous inner function code, and whose first element is the value of the variable `f`. When the closure is invoked, the anonymous code will be called with a stack which looks like `...nf`. In other words, the entire closure vector is "spread" onto the stack (about the same effort as reading an entire cache line), the indicator 'funarg is dropped, and the code sequence is loaded into the instruction buffer, leaving only the free variables themselves on the stack. The effect of this strategy is to allow the anonymous inner function to be compiled as if its argument list were `(n f)`.

`fact-kernel` is [#'fact-kernel1 'funarg vec3], while the inner function `fact-kernel1` is [roll2 zerop2 [drop drop '1] [dup 1- roll3 funcall-1 *] ifelse].

We need a version of the Y combinator which handles recursive kernels like `fact-kernel` which take 1 argument:

```
(defun y1 (k)
  (let*
    ((g g-prime
       (dup #'(lambda (g)
               #'(lambda (n)
                    (let* ((g g-prime (dup g)))
                      (funcall (funcall k (funcall g-prime g))
                               n)))))))
    (funcall g-prime g)))
```

Using `y1`, we can define non-recursive `factorial` itself:

```
(defun factorial (n) (funcall (y1 #'fact-kernel) n))
```

The code for `y1` is straight-forward: [#'y1-1 'funarg vec3 dup funcall-1]. The code for `y1-1` is [#'y1-2 'funarg vec4]. Finally, the code for `y1-2` is [roll2 dup funcall-1 roll2 funcall-1 funcall-1].

Interestingly, while the source for `y1` handles only kernels of one argument, the compiled code handles kernels of *any* number of arguments, by simply removing the setting and checking of the register keeping track of the number of arguments! This is because `y1` doesn't really deal with the arguments to the kernel itself, but only with the function closure which is lazily copied during the recursion.

With additional effort, a general Y combinator can be programmed which handles mutually recursive kernels [Baker92MC]. If the duplication of short closure vectors and vectors of mutually recursive functions can be done quickly—i.e., these vectors are "copied" by-reference (and thus *reference-counted* [Collins60] [Baker92LLL]), rather than truly copied, then recursion and iteration can be as efficient (O(1)) as in environments having cyclic code. Furthermore, it is well-known how to optimize closure creation and destructuring [Steele78], so our machine need not pay this cost for linearity.

Since a generic Y operator for a mutually-recursive system of functions must constantly recirculate and lazily copy a vector of routines in a strictly linear system, the accessing of the elements in these vectors is an excellent place for implicit copying, if such copying is going to be used anywhere. That efficient mutual recursion requires copying is not surprising, but the explicit copying makes more obvious the problem of implementing mutual recursion in a distributed system. Mutual recursion can be performed as a distributed computation, but the expense of explicit copying may make it unpalatable. Nevertheless, linearity will keep precise the meaning of this distributed computation, and guarantee that a system with similar recursions taking place simultaneously on different machines will have the same result as if all of the computations take place on a single machine.

FORTH IS A SYSTEM OF LINEAR COMBINATORS

Combinatory logic [Curry58] is a logical structure which is closely related to the *lambda calculus* [Church41]. The lambda calculus talks about names and substitutions in expression trees, while combinatory logic achieves the same "computations", but without needing any names. Backus's speech on the advantages of functional programming [Backus78] considers the ability of combinatory logic to eliminate names to be one of its major advantages. Most APL operators are combinators on array-type objects, and the absence of names from APL "one-liners" is quite characteristic of combinators.

When translating from lambda calculus expressions into *combinators*—as the primitives of combinatory logic are called, one must replace *random access* to a value by means of a name with *steering logic* to propagate values to the locations in an expression tree where they will be used. One of the simplest translations involves distributing two *copies* of a value down both branches of a binary tree (the **S** combinator), followed by the *killing* of any copies at leaves where they are not used (the **K** combinator). Obvious optimizations involve sending values only down branches where they will be used by means of the non-copying **B** and **C** combinators, which simply steer the values down either the left or right branch of the binary tree, respectively.

A linear version of the lambda calculus normally translates into only **B** and **C**-type combinators which neither copy nor kill values. Since any copying and killing is done explicitly by the programmer, these operations require "interpretation"—i.e., they are not built into any control structures, but require type-specific code for their implementation. An obvious optimization for a set of linear combinators is to provide more permutation combinators.

Most Forth operators take their operands from the top of the stack and return their values to the top of the stack. A perusal of this Forth code reveals the absence of variable names which is characteristic of combinators. The programming of Forth operators can therefore be seen as the construction of larger combinators from smaller ones. A Forth which incorporates only stack permutation operations like `swap`, `rotate`, and `roll` must be linear, because it has no copying or killing operators.

STACK MACHINE IMPLEMENTATION

The usual implementation of Forth utilizes *two* stacks—an operand stack and a return stack. Many Forth implementations allow the programmer to make temporary use of the return stack to perform more complex permutations of the operand stack. If a relatively complete set of permutation operations for the operand stack is provided, then the user will rarely need to "roll his own".

Since Forth is usually implemented on a traditional von Neumann machine, one assumes that "rolling a stack" is an expensive operation. The von Neumann bottleneck limits the speed with which the information can be rearranged in the stack, because only one word can be moved at a time. However, the RISC revolution makes the basic instruction cycle as fast as possible—i.e., one instruction per clock, and then uses pipelining and compiler technology to remove any impediments to achieving this speed. Unfortunately, a RISC clock cycle is limited by the need to access both a register bank and an instruction cache once per cycle. While these can be done in parallel and overlapped, the basic fact remains that the cycle time of these small RAM's is the limiting factor in RISC performance. The limiting factor in RAM access time is the logn gate delays that are used to address these RAM's. Therefore, the only way to make the fastest RISC architecture is to limit the size of these RAM's, which necessarily limits the amount of data that can be stored in the registers or the number of instructions that can be stored in the instruction cache.

If we examine Forth instruction streams, however, we notice that they are punctuated into two kinds of operations—a number of permutation operations followed by a number of computing operations. These permutation operations are considered "non-productive", while computing operations are "productive". However, stack permutation operations are no less productive than register loads, stores and transfers in a RISC architecture, and due to their ability to move more than one word at a time, there is evidence that permutations are *more* productive than register loads and stores. Furthermore, although permutations are probably more time-consuming than "computing" instructions, a true stack architecture could execute "computing" instructions blindingly fast. For example, AND'ing the top two items of the stack should take only 4-5 gate delays including reading from the top two items and storing back to one of them. Accessing a dual-ported register bank of 64-128 registers should take significantly longer, not counting the time to store the result. If one considers "round-trip time", the result may not be usable in the next clock cycle. Thus, a stack architecture could utilize a clock period for non-permutation instructions which would be a small fraction of that for a register machine, and a permutation unit should take about the same time for its operation as a RAM of the same size.

Most people describe the top several positions of the Forth stack as "locations", but it is more productive to think of them as "busses", since no addressing is required to read from them at all—the ALU is directly connected to these busses. The permutation operations necessary to get operands onto these busses can now be seen as the same sorts of operations that are used to extract values from registers. If one generalizes this idea, then one can conceive of multiple arithmetic operations being performed simultaneously on a number of the top items of the "stack". For example, the top 4 (or 8) items of the stack may be busses which are directly connected to an FFT butterfly network (itself a generalized permutation generator), in which case a radix-4 (or radix-8) FFT could be computed in parallel. Similarly, other operations on independent busses (top items of the stack) could be performed in parallel—e.g., computing an element-wise addition or multiplication of the top 2n items of the stack, considered in pairs. Of course, parallel operations of this sort must conserve the stack size, but any machine with parallel units of this sort would also have a general permutation network capable of "squeezing out" in parallel empty positions from the stack.

A traditional stack cache utilizes its space on the chip and memory bandwidth better than a register bank of the same capacity. The stack cache for a linear stack machine should be even more efficient, since 1) there are no stack frames or frame pointers; 2) all temporaries are handled the same—both named and unnamed temporaries; and 3) the space for each temporary is reclaimed immediately upon the use of its value (i.e., the cache is "self-cleaning"). This last characteristic guarantees that *all* of the data held in the stack cache is live data and is not just tying up space.

Since Forth is usually implemented on a traditional von Neumann machine, one thinks of the return stack as holding "return addresses". However, in these days of large instruction caches, in which entire cache lines are read from the main memory in one transaction, this view should be updated. It is well-known that non-scientific programs have a very high rate of conditional branches, with the mean number of instructions between branches being on the order of 10 or less. Forth programs are also very short, with "straight-line" (non-branching) sequences averaging 10 items or less. In these environments, it makes more sense to view the return stack itself as the instruction buffer cache! In other words, the return stack doesn't hold "return addresses" at all, but the instructions themselves! When a routine is entered, the entire routine is dumped onto the top of the return stack, and execution proceeds with the top item of this stack. Since routines are generally very short, the transfer of an entire routine is about the same amount of work as transferring a complete cache line in present architectures. Furthermore, an instruction stack-cache-buffer is normally accessed sequentially, and therefore can be implemented using shift register technology. Since a shift register can be shifted faster than a RAM can be accessed, the "access time" of this instruction stack-cache-buffer will not be a limiting factor in a machine's speed. Executing a loop in an instruction stack-cache-buffer is essentially the making of connections necessary to create a cyclic shift register which literally cycles the instructions of the loop around the cyclic shift register.

CONCLUSIONS

We have shown a Lisp called *Linear Lisp* which implements a "linear style" of programming suggested by *linear logic* [Girard87]. We have shown this linear style to be competitive with traditional Lisp styles for sparse polynomials [Baker94SP] and Quicksort [Baker94QS]. Furthermore, the mapping of Linear Lisp onto a stack architecture[5] utilizes a frameless Forth-style stack, in which the basic stack operations are *permutations*, rather than loads and stores. A specific kind of permutation—`roll`<*n*>—cycles the top *n* items of the stack so that the *n*-th item becomes the top item. These `roll` instructions are perfectly matched to the needs of linear logic to conserve values on the stack until they are utilized exactly once. Furthermore, any duplication or deletion of values are explicit operations that take place at the top of the stack. Interestingly, the consistent use of these `roll` (move-to-front) and `dup` operators makes the stack into an LRU buffer exactly like those used in "stack algorithms" for simulating LRU caches [Mattson70]!

[5]Linear languages also map elegantly onto *dataflow* architectures.

The idea of permuting the top several items of a stack by means of a "rolling" operation seems to have occurred independently to a number of people as a generalization of the `swap` instruction from the Burroughs stack machines [Lonergan61] [Hauck68]. Although the `swap` instruction was apparently introduced to optimize the code for non-commutative operators (subtraction, division) in arithmetic expressions, general rolling operators were first used in HP calculators [Osborne93] and Forth implementations [Moore93] to maximize the accessibility of operands while minimizing real and conceptual space requirements. Although stack architectures have been extensively written about, we have found *no* references to the more general use of stack permutations to access local variables[6] without the use of frame pointers. The literature on compiler optimizations for stack machines is particularly sparse—probably because the proponents of stack architectures touted trivial translation. With the modern notion of an optimizing compiler as a "partial evaluator", it is time to revisit the stack v. register controversy.

REFERENCES

AdaLRM: *Reference Manual for the Ada® Programming Language*. ANSI/MIL-STD-1815A-1983, U.S. Gov't Print. Off., Wash., DC, 1983.

Aho, A.V., and Johnson, S.C. "Optimal Code Generation for Expression Trees". *JACM 23*,3 (July 1976), 488-501.

Aho, A.V., *et al.* "Code Generation for Expressions with Common Subexpressions". *JACM 24*,1 (Jan. 1977), 146-160.

Aho, A.V., *et al. Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, MA, 1986.

Allmark, R.H., & Lucking, J.R. "Design of an Arithmetic Unit Incorporating a Nesting Store". *Proc. IFIP* 1962, 694-698.

Backus, J. "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs". *CACM 21*,8 (1978), 613-641.

Baer, J.-L. *Computer Systems Architecture*. Computer Science Press, Potomac, MD, 1980.

Baker, H.G. "Structured Programming with Limited Private Types in Ada: Nesting is for the Soaring Eagles". *Ada Letters XI*,5 (1991), 79-90.

Baker, H.G. "Lively Linear Lisp — 'Look Ma, No Garbage!'". ACM *Sigplan Notices 27*,8 (Aug. 1992), 89-98.

Baker, H.G. "NREVERSAL of Fortune—The Thermodynamics of Garbage Collection". *Int'l W/S Mem. Mgmt*, St Malo, 1992, Springer LNCS 637.

Baker, H.G. "Metacircular Semantics for Common Lisp Special Forms". ACM *Lisp Pointers V*,4 (Oct-Dec 1992), 11-20.

Baker, H.G. "How to Steal from a Limited Private Account". ACM *Ada Letters XIII*,3 (May/June 1993), 91-95.

[6][Harms91] [Suzuki82] [Baker92LLL] argue for swapping and permuting program variables. [Herlihy91] argues for swapping lock variables.

Baker, H.G. "Equal Rights for Functional Objects or, The More Things Change, The More They Are the Same". ACM *OOPS Messenger 4*,4 (Oct. 1993), 2-27.

Baker, H.G. "A 'Linear Logic' Quicksort". ACM *Sigplan Notices*, to appear, 1994.

Baker, H.G. "Sparse Polynomials and Linear Logic". ACM *Sigsam Bulletin*, to appear, 1994.

Barton, R.S. "A New Approach to the Functional Design of a Digital Computer". *AFIPS West. Jt. Computer Conf. 19* (1961), 393-396.

Barton, R.S. "Ideas for Computer Systems Organization: A Personal Survey". In Tou, J.T. *Software Engineering, Vol. 1*. Acad. Press, 1970.

Bell, C.G., *et al*. *Computer Engineering: A DEC View of Hardware Systems Design*. Digital Press, Bedford, MA, 1978.

Blake, R.P. "Exploring a Stack Architecture". *IEEE Comp. 10*,5 (1977), 30-39.

Brooks, F.P. "Recent Developments in Computer Organization". In *Advances in Electronics and Electronic Devices*, Acad. Press, NY, 1963.

Bruno, J., and Sethi, R. "Code Generation for a One-Generation Machine". *JACM 23*,3 (July 1976), 502-510.

Bullman, D.M., *ed*. "Stack Computers". *IEEE Comp. 10*,5 (May 1977), 14-52.

Carriero, N., & Gelernter, D. "Linda in Context". *CACM 32*,4 (1989), 444-458.

Case, R.P., & Padegs, A. "Architecture of the IBM System/370". *CACM 21*,1 (Jan. 1978), 73-96.

Castan, M., *et al*. "MaRS: A parallel graph reduction multiprocessor". *Comput. Arch. News 16*,3 (June 1988), 17-24.

Cheese, A. "Combinatory Code and a Parallel Packet Based Computational Model". *Sigplan Not. 22*,4 (April 1987), 49-58.

Chirimar, J., *et al*. "Proving Memory Management Invariants for a Language Based on Linear Logic". *Proc. ACM Conf. Lisp & Funct. Prog*., San Fran., CA, June, 1992, ACM *Lisp Pointers V*,1 (Jan.-Mar. 1992), 139.

Church, A. *The Calculi of Lambda-Conversion*. Princeton U. Press, 1941.

Coffman, E.G., Jr., & Sethi, R. "Instruction Sets for Evaluating Arithmetic Expressions". *JACM 30*,3 (July 1983), 457-478.

Collins, G.E. "A method for overlapping and erasure of lists". *CACM 3*,12 (Dec. 1960), 655-657.

Cragon, H.G. "An Evaluation of Code Space Requirements and Performance of Various Architectures". *Comput. Arch. News 7*,5 (Feb. 1979), 5-21.

Curry, H., and Feys, R. *Combinatory Logic*. North-Holland, 1958.

Dahl, O.-J., Nygaard. "Simula–an Algol-Based Simulation Language". *CACM 9*,9 (1966), 671-678.

Darlington, J., & Reeve, M. "ALICE—A Multiprocessor Reduction Machine for the Evaluation of Applicative Languages". *Proc. FPCA*, 1981.

Denning, P.J. "A Question of Semantics". *Comput. Arch. News 6*,8 (April 1978), 16-18. Discusses stack architectures.

Deutsch, L.P. "A Lisp Machine with Very Compact Programs". *Proc. IJCAI 3*, Stanford, CA, 1973.

Doran, R.W. "Architecture of Stack Machines". In Chu, Y., *ed*. *High-Level Language Computer Architecture*. Acad. Press, NY, 1975, pp. 63-108.

Fasel, J.H., *et al, eds*. "Sigplan Notices Special Issue on the Functional Programming Language Haskell". *Sigplan Not. 27*,5 (May 1992).

Francis, R.S. "Containment Defines a Class of Recursive Data Structures". *Sigplan Not. 18*,4 (Apr. 1983), 58-64.

Friedman, D.P., & Wise, D.S. "Aspects of applicative programming for parallel processing". *IEEE Trans. Comput. C-27*,4 (Apr. 1978), 289-296.

Gabriel, R.P. "The Why of Y". *Lisp Pointers* 2,2 (Oct.-Dec. 1988), 15-25.

Girard, J.-Y. "Linear Logic". *Theoretical Computer Sci. 50* (1987),1-102.

Glass, H. "Threaded Interpretive Systems and Functional Programming Environments". *Sigplan Not. 20*,4 (Apr. 1985), 24-32.

Goldberg, A., and Robson, D. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, Reading, MA 1983.

Grabienski, P. "FLIP-FLOP: A Stack-Oriented Multiprocessing System". *Proc. 2nd Symp. on Parallel Algs. and Archs.*, Crete, July, 1990, 161-168.

Halstead, R.H. "Overview of Concert Multilisp: A Multiprocessor Symbolic Computing System". *Comput. Arch. News 15*,1 (Mar. 1987), 5-14.

Hanson, C. "Efficient Stack Allocation for Tail-Recursive Languages". ACM *Conf. Lisp & Funct. Progr*., 1990, Nice, France, 106-118.

Harms, D.E., & Weide, B.W. "Copying and Swapping: Influences on the Design of Reusable Software Components". *IEEE Tr. SE 17*,5 (1991), 424-435.

Harrison, P.G., & Khoshnevisan, H. "Efficient Compilation of Linear Recursive Functions into Object Level Loops". *Symp. on Compiler Constr. '86, Sigplan Not. 21*,7 (July 1986), 207-218.

Hauck, E.A., and Dent, B.A. "Burroughs B6500/B7500 Stack Mechanism". *AFIPS Spring Jt. Computer Conf. 32* (1968), 245-251.

Hayes, J.R., *et al*. "An Architecture for the Direct Execution of the Forth Programming Language". *ASPLOS II, Sigplan Not. 22*,10 (1987), 42-49.

Herlihy, M. "Wait-Free Synchronization". *TOPLAS 11*,1 (1991), 124-149.

Hesselink, W.H. "Axioms and Models of Linear Logic". *Formal Aspects of Comput. 2*,2 (Apr-June 1990), 139-166.

Hewitt, C., & Baker, H. "Laws for Communicating Parallel Processes". *IFIP 77 Proceedings*, North-Holland.

Hill, D.D. "An Analysis of C Machine Support for Other Block-Structured Languages". *Comput. Arch. News 11*,4 (Sept. 1983), 6-16.

Hudak, P. "Conception, evolution and application of functional programming languages". ACM *Comput. Surv. 21*,3 (Sept. 1989), 359-411.

Hughes, J. "Why functional programming matters". *The Computer J. 32*,2 (April 1989), 98-107.

Ito, T., and Halstead, Jr., R.H., *Eds*. *Parallel Lisp: Languages and Systems*. Springer LNCS 441, 1990.

James, J.S. "FORTH for Microcomputers". *Sigplan Not. 13*,10 (1978), 33-39.

Keedy, J.L. "On the Use of Stacks in the Evaluation of Expressions". *Comput. Arch. News 6*,6 (Feb. 1978), 22-28.

Keedy, J.L. "On the Evaluation of Expressions using Accumulators, Stacks and Store-to-Store Instructions". *Comput. Arch. News 7*,4 (1978), 24-27.

Keedy, J.L. "More on the Use of Stacks in the Evaluations of Expressions". *Comput. Arch. News 7*,8 (June 1979), 18-21.

Keller, R.M., and Lin, F.C. "Simulated Performance of a Reduction-Based Multiprocessor. *IEEE Computer* (July 1984), 70-82.

Keown, W.F., Jr., *et al.* "Performance of the Harris RTX 2000 Stack Architecture versus the Sun4 SPARC and the Sun3 M68020 Architecture". *Comput. Arch. News 20*,3 (June 1992).

Kieburtz, R.B. "Programming without pointer variables". *Proc. Conf. on Data: Abstr., Defn. and Struct., Sigplan Not. 11* (spec. iss. 1976), 95-107.

Kieburtz, R.B. "The G-machine: a fast, graph-reduction evaluator". *Proc. IFIP FPCA*, Nancy, France, 1985.

Kingdon, H, *et al.* "The HDG-machine: a highly distributed graph-reducer for a transputer network". *The Computer J. 34*,4 (1991), 290-301.

Koopman, P. *Stack Computers, The New Wave*. Ellis Horwood, Chichester, W. Sussex, England, 1989.

Koopman, P., & Lee, P. "A Fresh Look at Combinator Graph Reduction". ACM *PLDI'89, Sigplan Not. 24*,7 (1989), 110-119.

Koopman, P.J., Jr., *et al.* "Cache Behavior of Combinator Graph Reduction". ACM *TOPLAS 14*,2 (April 1992), 265-297.

Lafont, Y. "The Linear Abstract Machine". *Th. Comp. Sci. 59* (1988), 157-180.

Lawson, H.W., Jr. "Programming-Language-Oriented Instruction Streams". *IEEE Trans. Comput. C-17*,5 (1968), 476-485. Discusses stacks.

Lieu, C.T., & Sarni, D. "O.Lisp Theoretical Basis and Description". *Sigplan Not. 25*,12 (Dec. 1990), 37-44.

Lipovski, G.J. "Just a Few More Words on Microprocessors of the Future". *Comput. Arch. News 6*,6 (Feb. 1978), 18-21.

Lonergan, W., & King, P. "Design of the B5000 System". *Datamation 7*,5 (1961), 28-32.

Lunde, Å. "Empirical Evaluation of Some Features of Instruction Set Processor Architectures". *CACM 20*,3 (Mar. 1977), 143-153.

Martí-Oliet, N., & Meseguer, J. "From Petri nets to linear logic". *Math. Struct. in Comp. Sci. 1*,1 (Mar. 1991), 69-101.

Mason, I.A. *The Semantics of Destructive Lisp*. CSLI LN 5, Stanford, CA, 1986.

Mattson, R.L., *et al.* "Evaluation Techniques for Storage Hierarchies". *IBM Sys. J. 9*,2 (1970), 78-117.

Maurer, W.D. "A Theory of Computer Instructions". *JACM 13*,2 (1966), 226-235.

McKeeman, W.M. "Stack Processors". In Stone, H., *ed. Introduction to Computer Architecture*. SRA, Chicago, 1975, pp. 281-317.

Moore, C.H. "The Evolution of Forth, An Unusual Language". *Byte 5*,8 (Aug. 1980), 76-92.

Moore, C.H. Personal communication, Aug.18, 1993.

Myers, G.J. "The Case Against Stack-Oriented Instruction Sets". ACM *Sigarch News 6*,3 (Aug. 1977), 7-10.

Myers, G.J. "The Evaluation of Expressions in a Storage-to-Storage Architecture". *Comput. Arch. News 6*,9 (June 1978), 20-23.

Nordstrom, D.J. "Threading Lisp". *Sigplan Not. 25*,2 (Feb. 1990), 17-24..

Omondi, A.R. "On function languages and parallel computers". *Fut. Gen. Comput. Sys. 6*,4 (Sept. 1991), 355-372.

Osborne, Tom. Personal communication, Aug. 27, 1993.

Passia, J., and Löhr, K.-P. "Fips: A Functional-Imperative Language for Exploratory Programming". *Sigplan Not. 28*,5 (May 1993), 39-48.

Patterson, D.A., and Ditzel, D.R. "The Case for the Reduced Instruction Set Computer". *Comput. Arch. News 8*,6 (Oct. 1980), 25-32.

Peyton-Jones, S.L., & Salkild, J. "The Spineless Tagless G-machine". ACM *FPCA 4*, (1989), 184-201.

Ponder, C.G., *et al.* "Are Applicative Languages Inefficient?" *Sigplan Not. 23*,6 (June 1988), 135-139.

Prabhala, B., & Sethi, R. "Efficient Computation of Expressions with Common Subexpressions". *JACM 27*,1 (Jan. 1980), 146-163.

Randell, B., & Russell, L. *Algol 60 Implementation*. Acad. Press., 1964.

Rather, E.D., *et al.* "The Evolution of Forth". *HOPL-II, Sigplan Not. 28*,3 (Mar. 1993), 177-199.

Robison, A.D. "The Illinois Functional Programming Interpreter". *Symp. on Interp. and Interpretive Techs., Sigplan Not. 22*,7 (July 1987), 64-73.

Ryer, M.J. *Programming in HAL/S*. Intermetrics, Inc., Camb., MA, 1978.

Sarwar, S.M., *et al.* "Implementing Functional Languages on a Combinator-Based Reduction Machine". *Sigplan Not. 23*,4 (Apr. 1988), 65-70.

Shalit, A. *Dylan™: An object-oriented dynamic language*. Apple Computer, Camb., MA, 1992.

Sites, R.L. "A Combined Register-Stack Architecture". *Comput. Arch. News 6*,8 (April 1978), 19.

Smith. A.J. "Cache Memories". *Comput. Surv*. (Sept. 1982), 473-530.

Steele, G.L. *Rabbit: A Compiler for SCHEME (A Study in Compiler Optimization)*. AI-TR-474, MIT AI Lab., May 1978.

Steele, G.L. *Common Lisp, The Language; 2nd Ed*. Digital Press, 1990.

Stevenson, J.W. "Efficient Encoding of Machine Instructions". *Comput. Arch. News 7*,8 (June 1979), 10-17.

Strom, R.E. "Mechanisms for Compile-Time Enforcement of Security". *Proc. POPL 10*, Jan. 1983.

Stroustrup, B. *The C++ Programming Language*. Addison-Wesley, 1986.

Suzuki, N. "Analysis of Pointer 'Rotation'". *CACM 25*,5 (1982), 330-335.

Tanenbaum, A.S. "Implications of Structured Programming for Machine Architecture". *CACM 21*,3 (Mar. 1978), 237-246.

Vegdahl, S.R. "A Survey of Proposed Architectures for the Execution of Functional Languages". *IEEE Trans. Comput. C-33*, 12 (1984), 1050-1071.

Wadler, P. "Linear types can change the world!". *IFIP TC2 Conf. on Progr. Concepts & Meths.*, April 1990.

Wadler, P. "Is there a use for linear logic?". *Proc. ACM PEPM'91*, New Haven, June 1991, 255-273.

Wakeling, D., & Runciman, C. "Linearity and Laziness". *Proc. Funct. Progr. & Comp. Arch.*, LNCS 523, Springer-Verlag, Aug. 1991, 215-240.

Wirth, N. "Stack vs. Multiregister Computers". *Sigplan Not.* (Mar. 1968), 13-19.

APPENDIX: POSTSCRIPT™[7] CODE FOR EXAMPLES

Below are the Linear Lisp examples as compiled into *Postscript*™—a widely available Forth-like language—by our Linear Lisp compiler.

We first define various primitive stack operations.

```
/roll1 {} def                   /roll2 {exch} def
/roll3 {3 -1 roll} def          /roll4 {4 -1 roll} def
/roll5 {5 -1 roll} def          /roll6 {6 -1 roll} def
/roll7 {7 -1 roll} def          /roll8 {8 -1 roll} def
/minusp2 {dup 0 lt} def
/cons {2 array astore} def
/carcdr {aload pop} def
/atom2
{dup type /arraytype eq {dup length 2 eq not} {true} ifelse}
def
/zerop2 {dup 0 eq} def
/sub1 {1 sub} def               /add1 {1 add} def
/function {userdict exch get} def
/arrayp2 {dup type /arraytype eq} def
/last {dup length sub1 get} def
/closure {/funarg exch 2 add array astore} def
/closurep2
{arrayp2 not {false}
             {dup length 0 eq {false}
                              {dup last /funarg eq}
                              ifelse}
             ifelse}
def
/apply {closurep2 {aload pop pop exec} {exec} ifelse} def
```

We now show actual compiled examples.

```
/identity {} def
/five {pop 5} def
/square {dup mul} def
/quadratic
{roll3 dup roll4 dup neg roll2 square 4 roll5 roll6 mul mul
 sub sqrt add 2 roll3 mul div}
def
/my-abs {minusp2 {neg} {} ifelse} def
/append
{roll2 atom2 {pop} {carcdr roll3 append cons} ifelse}
def
/append-kernel
{roll3 atom2 {pop pop}
             {carcdr roll4 roll4 dup apply cons}
             ifelse}
def
/append {/append-kernel function dup apply} def
/fact
{roll2 zerop2 {roll2 pop add1}
              {dup roll3 dup roll3 sub1 roll3 roll3 apply mul}
              ifelse}
def
/factorial {/fact function dup apply} def
/ifactorial
{1 roll2 {roll2 zerop2 {pop pop}
                       {dup roll4 mul roll2 sub1 roll3 dup
                        apply}
                       ifelse} dup apply}
def
/fact-kernel
{{roll2 zerop2 {pop pop 1} {dup sub1 roll3 apply mul} ifelse}
 1 closure}
def
/y1
{{{roll2 dup apply roll2 apply apply} 2 closure}
 1 closure dup apply}
def
/factorial {/fact-kernel function y1 apply} def
```

---

[7] *Postscript* is a trademark of Adobe Systems, Inc.