ACM *Sigplan Notices* **30**, 1 (January 1995), 45-52.

'Use-Once' Variables and Linear Objects — Storage Management, Reflection and Multi-Threading

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.

Programming languages should have 'use-once' variables in addition to the usual 'multiple-use' variables. 'Use-once' variables are bound to *linear* (*unshared*, *unaliased*, or *singly-referenced*) objects. Linear objects are cheap to access and manage, because they require no synchronization or tracing garbage collection. Linear objects can elegantly and efficiently solve otherwise difficult problems of functional/mostly-functional systems—e.g., in-place updating and the efficient initialization of functional objects. Use-once variables are ideal for directly manipulating *resources* which are inherently linear such as freelists and 'engine ticks' in reflective languages.

A 'use-once' variable must be dynamically referenced *exactly once* within its scope. Unreferenced use-once variables must be explicitly killed, and multiply-referenced use-once variables must be explicitly copied; this duplication and deletion is subject to the constraint that some linear datatypes do not support duplication and deletion methods. Use-once variables are bound only to linear objects, which may reference other linear or non-linear objects. Non-linear objects can reference other non-linear objects, but can reference a linear object only in a way that ensures mutual exclusion.

Although implementations have long had *implicit* use-once variables and linear objects, most languages do not provide the programmer any help for their utilization. For example, use-once variables allow for the safe/controlled use of reified language implementation objects like single-use continuations.

Linear objects and use-once variables map elegantly into dataflow models of concurrent computation, and the graphical representations of dataflow models make an appealing *visual linear programming language*.

INTRODUCTION

Static datatyping enables a Fortran compiler to statically select the correct arithmetic operation to be performed on a numeric quantity—e.g., float v. fixed. In object-oriented languages, however, the cost of managing storage for a complex float object may exceed the cost of the arithmetic. Additional 'type' distinctions may be helpful in reducing these costs; in particular, *linearity* is a 'type' distinction which can reduce storage management and synchronization costs.

What is the result *type* of the expression `cons(a,b)`, which produces a new Lisp-like pair? Common Lisp [Steele90] types this expression with the simple type `cons`, while ML [Harper86] types it as `list(integer)`, assuming that `a` is of `integer` type. These types are good for describing the *shape* of the objects referred to by the expression, but not for describing their *usage*. For example, shape information does not include the fact that these objects are *new*, *unshared* objects. Thus, proving `eq(cons(a,b),cons(a,b))`≡false is non-trivial.

A more fundamental question is: why must the 'function' `cons` be primitive? Why can't `cons` cells be allocated and initialized in separate operations without destroying functionality and efficiency? Consider this definition:

```
(defun cons (x y) (rplacd (rplaca (make-cons) x) y))
```

If `rplaca`, `rplacd` are mathematical *functions*, then so is this definition for `cons`. Unfortunately, in a classical functional language, `rplaca` and `rplacd` must *copy* their arguments in case there are other references to them. An "update in place" optimization for this `cons` requires information not available to `rplaca`, `rplacd`—the fact that their arguments are not referenced elsewhere. Although every update-in-place optimizer—e.g., [Hudak86]—can handle this easy case, update-in-place is not always guaranteed for harder cases.

Linear types elegantly solve this problem by providing an additional dimension to the programmer's type system—*whether the object is shared*. The result type of `make-cons` is then `<cons,unshared>`. Unshared objects can always be updated in place, and since `rplaca`, `rplacd` preserve the nonsharedness of their first argument, the programmer is guaranteed that his code will be efficient, regardless of the quality of the compiler's optimizations.[1] In particular, the same storage that is given to `rplaca`, `rplacd` is returned, and no locking need be performed during the "side-effect" which initializes the `cons` cell. (`coerce-to-shared-readonly` is nearly the identity function—in a static type system it may do nothing, while in a dynamic type system it may just flip one tag bit in a pointer.)

```
(defun cons (x y)
  (coerce-to-shared-readonly (rplacd (rplaca (make-cons) x) y)))
```

The parallel (divide and conquer) in-place initialization of a functional array is trivial using linear types and linear (use-once) variables:

```
(defun (iota-initialize linear) ((v linear) i)
  ;;; takes & returns a linear object.
  (let* ((v n (length v)))        ; take and return linear vector.
    (if (zerop n) v               ; return initialized subvector.
      (let* ((m (floor n 2))           ; m is 1/2 of length of v.
             (vl vm vh (split v m)))   ; split sizes m, 1, n-m-1.
        (catenate (iota-initialize vl i)     ; catenate-in-place.
                  (setf2 (aref vm 0) (+ i m))       ; returns v.
                  (iota-initialize vh (+ i m 1)))))))

(defun iota (n)      ; return constant vector #(0 1 2 3 ... n-1).
  (coerce-to-shared-readonly
    (iota-initialize (make-array n) 0)))
```

Using linear types, the array can be efficiently and incrementally initialized *before* it is "ready for prime time". Once the array is fully initialized, it "comes out"

---

[1] Some compilers type *sharing* [Steele78], a distinction is not provided to the programmer.

[2] `setf` does an explicit `kill` on the (empty) previous contents and returns the vector.

("is published") as a shared, read-only object which is ready for the rigors of access by multiple threads. Linear types elegantly solve the problem of modularly and efficiently initializing complex *functional* objects in an object-oriented system such as the Common Lisp Object System (CLOS) [Steele90] [Kiczales91]. Linear types are better than the "nurseries" of generational garbage collectors; linear objects are "fetal" objects, which have not yet been truly "born" as first-class (shared) objects.

In the examples above, we have coerced linear objects into shared objects before returning them for general use. However, it is usually more efficient to keep the object unshared as long as possible:

```
(defun (cons linear) (x y)        ; syntax indicates result type.
  (rplaca (rplacd (make-cons) x) y))
```

Lists which consist entirely of unshared `cons` cells are useful and efficient; e.g., Common Lisp allows a function to grab its "argument list" with a `&rest` binding:

```
(defun min (x &rest args)       ; minimum of 1 or more arguments.
  (if (null args) x
    (min2 x (apply #'min args))))       ; min2 of exactly 2 args.
```

A naive Common Lisp implementation performs $O(n^2)$ conses in the evaluation of a `min` with n arguments. If Common Lisp implicitly typed `&rest` bindings as *linear*, then only $O(n)$ conses are performed, because the list is passed on, rather than copied. Even if the implementation of `apply` spreads arguments in registers, the linear argument list cells can be immediately reclaimed without GC.

```
(defun min (x &rest (args linear))
  ;;; minimum of 1+ arguments using linear &rest lists.
  (if-null[3] args (progn (kill args) x)      ; note explicit kill.
    (min2 x (apply #'min args))))
```

Multiple linear values can be returned, but *extra linear return values cannot be ignored*, the way Common Lisp does. The linear programming style often returns unchanged arguments, so returning multiple values must be efficient.

Closures may be linear, so that they can refer to linear (use-once) free variables. An occurrence of a linear variable in a closure is its *only* occurrence, so *closure environments are disjoint for linear variables*. A `mapcar` for linear lists that is guaranteed to be sequential (because the function is never copied) is as follows:

```
(defun (mapcar linear) ((f linear) (l linear))
  ;;; f,l are linear, mapcar returns linear list.
  (if-null l (values f l)                  ; return both f and l.
    (dlet* (((lcar . lcdr) l))  ; destructure l into lcar, lcdr.
      (let* ((f flcar (lfuncall f lcar)))       ; bind f & value.
        (cons flcar (mapcar f lcdr))))))
        ;;; compiler can reuse destructured linear cell.
```

Continuations can be linear (enabling an efficient implementation, unlike [Lincoln92]), since they normally return only once ("one-shot" [Haynes87]), even when doing co-routining and time-sharing [Wand80]. (Prolog-like *backtracking* [Haynes87] requires expensive copying—if allowed—or (non-linear) sharing of continuations.) Of course, a linear `call/cc` can have no *implicit* continuation—the *only* way to return from a linear `call/cc` is to invoke the passed continuation, so a function which does not escape via a passed continuation must return it for its caller's use. Escaping via a continuation kills an implicit continuation thereby causing intermediate linear variables to be finalized *á la* Common Lisp `unwind-protect`.

Executable code (with its constants) is non-linear, readonly ([Lincoln92]), allowing for lazy copying into an instruction cache. Efficient interpreters of *linear* executable code use Y combinators for looping and recursion [Baker94LLPS].

Scheme *delays* can be linear, because most uses can be emulated by closures and (when linear) need no value cache. *Futures* [Baker77], however, are non-linear, since even single-use futures require synchronization. We will consider here only languages which are either strict (applicative) or explicitly lazy—e.g., `delay`.

WHAT IS A 'LINEAR' (USE-ONCE) VARIABLE?

Girard's *linear logic* [Girard87] is a logic of limited resources. In the strict form of Girard's logic, the copying and destruction of certain terms are not allowed, thus *conserving* those terms and the resources they represent. Linear (use-once) variables are bound to objects—e.g., *file descriptors*—whose use and accessibility must be tightly controlled. A use-once variable has a short and uneventful life—it is created and bound to a linear value, and this value is then subsequently read exactly *once*—a *destructive read* which *consumes* the value. A function might bind a use-once parameter name to a linear argument object,[4] and this object must either be explicitly consumed during the execution of the function or returned as a return value. A use-once variable can also be locally created, and then either be explicitly killed, or passed along to another procedure for disposal. Use-once *variables* may be bound only to linear *objects*, although these linear objects may indirectly reference non-linear objects and values. Non-linear objects can reference linear objects, but only via mutual exclusion locks.

The acceptance by a function of a linear argument object places a great responsibility on the function—the function must either pass the linear object to another function as an argument, explicitly dispose of it, or return it as a return value. Linear values are thus like controlled substances and toxic waste—the language implementation goes to extraordinary lengths to control their proliferation and disposal—even in exceptional circumstances, where linear objects can be finalized—e.g., by Common Lisp's `unwind-protect` [Steele90].

The limitation on a variable being used exactly once can be statically checked, although it is more difficult in the case of exceptions. The occurrence of the variable in one statement precludes its use in any other statement. In an argument list, the variable can occur in only one argument position. In an `if` or `case` expression, an occurrence in any arm implies its occurrence in every arm. Single-use in loops defined by tail-recursion follows the previous rules, while single-use

[3]
```
(defmacro if-null (a b c)
  `(let* ((truth ,a (lnull ,a))) (if truth ,b ,c)))
```

[4] Linear parameter bindings are *simultaneously* call-by-value *and* call-by-reference!

for loops in Algol-like languages can be defined by converting these loops into tail-recursive functions, or by performing *typestate* analysis [Strom86].

The intuition behind linear types is the *conservation of physical matter*—a linear object cannot be created or destroyed without special permission, although it can be moved or transferred at will. Thus, an occurrence of a linear variable—e.g., in the argument list of a function call—indicates that the function will *consume* the value of the variable. If the function does not later return this value, either as a returned value, or by assigning it to a more global variable, then that function is responsible for properly disposing of any of the resources controlled by the value—even in the case of exceptions. Parameter passing of linear objects has a satisfying physical metaphor—the value is *swapped* into the parameter location, and the empty value previously bound to the parameter location becomes the value of the variable in the caller [Harms91].

Linear objects have the property that there is only one access path to them at any given time—i.e., their normal and transitive reference counts are both equal to one. Thus, if a linear object is passed to successive levels of a recursive function, only one reference to the object ever appears in the activation stack, because the operation of passing the object (by swapping) destroys the previous reference. An important corollary to these rules is that the resources controlled (owned) by two different linear objects are always disjoint. Thus, the owner of a linear object transitively owns the resources controlled by the linear object, and can manipulate them without interference. Linear objects are quintessentially *passive* objects—*one can never be surprised during their access*.

Although linear types do not allow implicit copying, the definer of a linear abstract data type may provide an explicit copying method. Such a copying method has to strictly (deeply) copy the entire linear structure, so modifications to this copy will not affect the original. Thus, if even one portion of a linear object lacks a copying method, then the incorporating object cannot be copied.

One quintessentially linear object which cannot be copied or destroyed is the *I/O stream* to/from the user. This stream must by synchronized against simultaneous access, yet even the possibility of multiple access is nonsensical—input characters would be distributed randomly to different readers, while the output would consist of a random interleaving of characters from different printers. Such nonsense is trivially eliminated by making `print` take this stream as an argument and return it again as a value, and by making `read` take this stream and return it again as a (multiple) value. If all operations on this stream both accept and return this stream, then synchronization is never necessary, since only one pointer to the stream exists, and hence only one operation can be performed.[5] Opening/closing the terminal stream is nonsensical, hence it is passed at program initialization (*á la* C's `main` arguments), and must be returned to the operating system at program exit.

LINEAR OBJECTS AVOID SYNCHRONIZATION

Programming languages make a crucial distinction between the *name* of an object and the *object* itself. This distinction is most evident in the ubiquitous random access memory (RAM), in which the name of an object is an *address* within the RAM and the object itself is the *contents* of the RAM location at that address. There are two major reasons for the manipulation of names rather than objects themselves. First, the size of the name of an object is small and essentially uniform, because the number of bits to distinguish among objects is usually far fewer than the number of bits required to describe an object's attributes. Second, an object may be used more than once within a computation, and it is easier to refer to the subsequent occurrences of the object by a short name instead of describing the entire object. With a mutable object, however, the convenience of referring to the same object by name becomes a necessity [Baker93ER], because in this case, there is no way other than naming to describe the potentially changed object—especially when the change was not made by the program which is describing the object.

Modern object-oriented languages do not take advantage of the orthogonality of these two different reasons for naming. One may name a large object which is used only once because it is easier to transport a small name than a large object. Such a name is analogous to the single key for an airport rental locker—although the key is never shared, it allows its user to temporarily carry a small key instead of a large bag. Object-oriented languages, however, pessimistically assume the worst—that anyone who has access to a locker has copied its key for all of his acquaintances, and therefore an arbitrary number of keys exist for each locker. One must therefore be prepared when accessing a locker to compete with many others for its contents.

Worse still is the plight of the airport manager, who must determine which lockers are available for new customers. It is not enough to search the pockets of every person in the airport, because many keys may be stored in the lockers themselves. The manager must therefore first search all the people, and then before they can access the lockers, he must quickly search the lockers for keys to other lockers. Only if *no* person has direct or indirect access to a locker can it be made available for re-rental.

The ability to copy a key for one's spouse and friends *appears* to be a great convenience. However, the hassles of contending for one's own locker and the surprises one finds there quickly become inconvenient. Furthermore, the rent charged by the airport manager is expensive because few lockers are available.

The solution proposed by some languages—e.g., Ada [AdaLRM83]—is for each customer to hire an armed guard to stand beside his locker and check the credentials of each person who has a key. This guard establishes a queue and arranges contenders for the locker into priority order. This "solution" may reduce the contention and surprises, but it is extraordinarily expensive, and requires that one also synchronize with his own guard.

Every traveler already knows the elegant solution—provide only one unforgeable key to each locker. Then, if one has the key, he will not face contention and surprises when accessing his locker. The airport manager's job is also made easier. If the customer surrenders something of value—e.g., his passport—to get the key, he is certain to bring it back. When the key is returned, the locker is immediately available for re-rental. The manager may have to clean out baggage and keys from

[5] *Within* the linear stream are non-linear objects which synchronize for flow control.

this locker, but there can be no other claimants for the locker itself. The lockers are cheaper because the manager's job is easier and more lockers are available.[6]

Airport lockers are not the ideal solution for all storage—e.g., one would not want these access restrictions on one's apartment. However, for short-term rental, the one-key system is extremely elegant and efficient. It allows one to quickly and cheaply *cache* one's belongings for later retrieval.

A linear object can be thought of as an airport locker with but one key. The user of this object never has to worry about contention or surprises, and can therefore access it efficiently. He can pass on its key to others, and can also accept keys from others, so long as no copies are made. Because each key provides exclusive access to a locker, keys are valuable, and great efforts are made to avoid their loss. When a locker is cleaned out, any keys found there are immediately recycled. In the case of exceptions, keys are bequeathed to the heirs.

In most object-oriented languages, the key to an object is its name, and this name can be used an arbitrary number of times. For this convenience, however, a high price must be paid—every access to the object is expensive and the cost of real estate (storage) is high. In a language offering linear types, however, one may be willing to operate with just one name for a particular object, and thereby avoid the high cost of access and rental.

Linear objects can also be used to avoid "infinite descent" in the implementation of a reflective system [Friedman84], where there must be some unshared resources in order to accomplish anything at all. Every real (non-virtual) resource is *ipso facto* linear, since it cannot be copied or destroyed. For example, a real CPU, a stack and a heap are all linear objects, because the program (while it is running) has exclusive access to them; an interpretation of a (serial) function call is that the caller implicitly passes the CPU, the pointer to the free stack location and the freelist pointer to the callee, which then implicitly returns these resources as additional returned values. Use-once variables and linear types allow the programmer to directly manipulate unshared resources as efficiently as can machine language. We therefore suggest that *reified* objects [Friedman84] have linear type.

The reader may object that modern languages have always had unshared resources in the form of locally allocated variables. This objection is not correct, however, because the user can use the names of these local variables as often as he pleases, and even pass them on to other functions. This multiplicity of copies of the name of an assignable local variable produces great uncertainty about which assigned value is associated with which variable use, and a source of great complexity in modern compilers and computers is the sequencing they apply to disambiguate these uses. Thus, local variables reduce the potential for sharing, but do not eliminate it, and the computer requires another level of interpretation before arriving at unshared objects. Occurrences of variables which are truly linear, however, can never have a sequencing problem, because there is no possibility of sharing among them. Dataflow constraints alone are therefore sufficient to avoid contention.

[6]A clever manager at an airport where most of the travelers are similar can recognize identical bags and store them all in the same locker, remembering only how many of them there are. This is the scheme of [Baker92LLL].

LINEAR ABSTRACT DATA TYPES

> "A type may be viewed as a set of clothes (or a suit of armor) that protects an underlying untyped representation from arbitrary or unintended use". [Cardelli85]

An abstract data type whose representation (concrete type) is linear will also be linear. A linear object can have both linear and non-linear components. A linear abstract data type object may be copied if and only if the type exports a copying method—e.g., linear abstract types can be used to represent *resources*—e.g., storage or money—for which copying makes no sense. Similarly, killing a linear object is prohibited unless the type exports a deletion method. Linear abstract objects which represent a resource can usually be subdivided into smaller portions which can be put back together again—e.g., a contiguous range of memory cells can be divided into *buddies*, and these buddies can later be coalesced back into one object. Linear abstract data types allow one object to physically incorporate another linear object without the need for pointers—e.g., linear types eliminate the requirement that Common Lisp `defstruct` structures be vectors of pointers to objects.

Linear objects that are not copyable or deletable, are still *transferable*, since transferring a resource from one ownership to another can be accomplished even when the resource cannot be copied. Indeed, argument passing requires ownership transfer, so all first-class linear values are transferable.

WHAT IS A *NON*-LINEAR TYPE?

Non-linear objects can be shared in an unrestricted manner—i.e., they are the usual objects of object-oriented languages. For example, Lisp *symbols* are quintessentially non-linear. Non-linear objects themselves cannot be copied, for this makes no sense [Baker93ER], but the *names* of non-linear objects can be copied/deleted with little apparent cost. However, every access to a non-linear object must be synchronized and non-linear objects must usually be allocated in a garbage-collected heap, since the detection of the last reference is quite difficult.

A non-linear type is an abstract *naming* type, with no representation structure *per se*. Its own structure may be as simple as a reference to a linear representation object, plus some management information—e.g., a reference count.[7] A mutable non-linear type can be accessed by a simple mechanism such as an atomic `swap` operation (or `compare-and-swap` [Herlihy91]), which produces either a pointer to the linear representation object or an indication of emptiness. Since a `swap` that succeeds leaves the non-linear type empty, mutual exclusion and access is achieved in a single hardware operation. The abstract operations of the non-linear type then operate on the linear representation object, which may itself be abstract. Thus, "within" the non-linear type no further synchronization is required, and any attempt to re-synchronize is nonsensical, since the representation type is a different type from the naming type. Of course, the operations of the non-linear type must eventually restore a representation object to the non-linear object; this new

[7]Non-linear types thus make explicit the invisible pointers of 'lazy' reference counts [Collins60] [Goto88]. They are also reminiscent of the shared header+unshared structure used in many Lisp systems [Steele77] [Yuasa90].

representation object may be different from the earlier representation object (e.g., Smalltalk `become:` [Goldberg83]).

A *readonly* non-linear object is a curious entity, since it is a proxy for *n* (lazy) copies of itself—each reader makes a lazy copy as it traverses the structure. No part of such an object can be modified or reclaimed, however, for fear that another reader inside this object will discover the change. *A shared, readonly object is reminiscent of an object protected by a readers-writers access protocol*. Readers can proceed concurrently with other readers, but *a writer must wait for exclusive access*. In the absence of a reader reference count, writers are held up forever, because only the garbage collector can prove the absence of readers, and allow the object to be recycled. The inability to mutate such an object thus results from lazy copying.

LINEAR TYPES AND 'OPEN'/REFLECTIVE SYSTEMS PROGRAMMING

Systems programs deal explicitly with limited *resources*—e.g., [Appelbe84]:

> The [systems language] implementation of … a file system must support
> (i) management of dynamically allocated resources, for example, I/O buffers, logical and physical files, pipes, and directories;
> (ii) establishing reliable interfaces, so that users can neither maliciously nor unintentionally defeat the resource allocation scheme;
> (iii) sharing of resources at the user level;
> (iv) a high-level representation of the hardware environment.

These programs allocate and recover main memory, file memory, communication channels, etc. Unfortunately, most languages are more concerned with computing the output of a "function" when given an input, than they are with the continual processing of data within resource constraints. These languages are ill-equipped to deal directly with resources, and prefer to sweep their management "under the rug" of a vendor's operating system or run-time system. The management of resources by such "closed" means defies the trend towards modular/"open" systems, in which implementation/policy decisions can be tuned by the user [Friedman84].

One possibility for achieving "open" systems is to allow the user to modify the resource managers. Such systems are called "reflective" systems, because they have knowledge of their own implementation [Friedman84]. Unfortunately, a reflective system may be an insecure system, because modification of the implementation by the user can destroy any assurance about its correctness. Existing reflective languages provide few tools to assure the end-user of the application's correctness. Thus, the end-user is faced with two unpleasant alternatives—a closed, correct system, or an open, incorrect system.

Resource managers must be both *safe* and *leakproof*. A *safe* manager will not allocate the same resource for multiple uses, and a *leakproof* manager will not allow resources to become misplaced. Linear types for reified objects can provide help in assuring these properties—something not possible with existing types.

Some "open" systems utilize memory management by tracing garbage collection solely to avoid the unpleasantness that might arise through the activities of an erroneous user-provided policy manager. While we believe in the advantages of garbage collection, it should also be possible to provide safe open systems which require no garbage collection. Linear types provide a significant level of control over user-provided code which can be used to eliminate the most egregious errors at compile time. Like a double-entry bookkeeping system, linear types require the user to account for all resources, whether they are incorporated within other resources or explicitly returned to the resource pool—e.g., linear types are ideal for typing the *engine ticks* in [Haynes84] (except the clock itself, whose surprises are non-linear—e.g., how time flies!).

HARDWARE IMPLICATIONS OF EXPLICIT LINEARITY

Several other researchers have considered the hardware implications of explicit linearity [Wakeling91] [Lincoln92] [Chirimar92], with the most popular interpretation involving two heaps—the linear heap and the non-linear heap. We agree with this dichotomy, and believe that its extension to the hardware level can yield significant savings. Swapping access to linear memory is not more expensive, because the outgoing data is sent on data wires that are unused during normal (copying) read cycles. Cache 'consistency' protocols for linear objects are trivial, so all consistency effort can be aimed at non-linear objects. Furthermore, non-linear objects can be of fixed size and quite small, because the bulk of the data structures are stored in linear memory. Linear memory (at least for non-arrays) can even allocate itself, since any read is the last access to the stored object.

RISC architectures which use assignment are not really reduced, because assignment is a complex operation—*assignment = copy + kill + move*. By factoring assignment into more fundamental operations, we can further simplify and optimize RISC architectures—especially for distributed applications.

A linear language is ideal for a *dataflow architecture,* which must explicitly duplicate/delete (linear) tokens, as well as for a frameless stack architecture—e.g., the Forth model [Rather93]. A Forth-type stack architecture has no frame pointer, but manipulates values on a stack by means of indices relative to the top of the stack (TOS). If the fundamental operation of fetching the value of a local variable *permutes* the k'th deep object to TOS—i.e., a Forth or Postscript `roll` operation—then `roll` is the destructive read that is distinctive of linearity. When one wishes to utilize a value more than once in such an architecture, one must explicitly duplicate the value at TOS (assuming that the type exports a copy operation). Similarly, if one wishes to discard a value in such an architecture, one must bring it to TOS and explicitly delete (drop/pop) it (assuming that the type exports a deletion operation). Duplication/deletion are thus "uninterpreted" (i.e., type-specific) operations rather than "interpreted" operations that are uniform for every type. By forcing duplication/deletion to take place at TOS, one has a uniform calling sequence for these operations for both hardware and software data types. In short, a Forth-like permutation stack architecture is the ideal combinator basis for a "linear λ-calculus". We have prototyped a compiler from Linear Lisp into Postscript and find that the mapping is elegant [Baker94LLPS]. Code itself is linear and uses lazy copying (λ-calculus Y operator [Gabriel88]) to efficiently implement recursion.

LINEAR TYPES AND GRAPHICAL PROGRAMMING LANGUAGES

Due to their heavy use of multiple returned values, programs in linear languages tend to be somewhat difficult to read in a printed representation. These same programs look quite elegant, however, when shown in a graphical representation. In this graphical representation, the linear variables become simple wires connecting the output of one operation to the input of another. Each routine is itself acyclic, since loops are implemented by means of tail-recursion, so that the diagram can always be drawn with the input end of the wire higher than the output end of the wire. Thus, the values "flow down" the diagram. Because these programs tend to be small and highly modular, it should rarely be necessary to look at large "spaghetti" diagrams.

Rather than developing a printed representation—e.g., Algol-ish infix notation—which is "prettier" than a Lisp-like parenthesized expression, we are focussing on developing a true graphical representation for programs in a linear language. A significant benefit of a graphical representation is the elimination of *spurious sequencing* that is present in the printed representation. This spurious sequencing arises because of the need to choose *some* ordering to present the computation, but the actual execution ordering for a linear language is constrained only by dataflow constraints. Given the ubiquity of "graphical user interfaces" (GUI's), the "lowest common denominator" representation for a program need no longer be a linear sequence of ASCII characters, but can instead be a more readable graphical structure.

RESULTS OF EXPERIMENTS WITH A LINEAR STYLE OF PROGRAMMING

We programmed several benchmarks—*Quicksort*, *FRPOLY* and *Boyer* [Gabriel85]—in a linear style of Lisp. We implemented a fragment of this 'Linear Lisp', which uses only linear objects and immutable nonlinear objects, in Coral Common Lisp using macros, and performed timing experiments.

We programmed both list and vector Quicksort routines in the linear style [Baker94QS], and found that the list version of linear Quicksort—which includes a cell reuse optimization [Wakeling91], and hence does *no* consing—is the fastest Quicksort we could program, period, and also 1.81X faster than the built-in Common Lisp `sort` primitive. The vector version of a linear Quicksort (which depends upon *swaps* not supported in hardware) is only 3.5% slower than a traditional non-linear Quicksort. All Quicksorts call a generic comparison predicate. The most elegant linear Quicksort for vectors decomposes the vector into disjoint pieces (by means of separate linear headers)—i.e., the arguments to recursive calls are not *indices into* a vector, but the actual (linear) *subvectors* themselves. Reconstitution of the subvectors is performed by a smart concatenation which notices that the concatenation can be performed in-place. This linear Quicksort is ideal for a shared-memory multiprocessor where disjoint subvectors can be concurrently Quicksorted.

FRPOLY [Gabriel85] performs exponentiation on sparse multivariate polynomials. FRPOLY's addition and exponentiation routines are functional, but its multiplication routine is ugly imperative code. We reprogrammed FRPOLY in the linear style to see whether this ugliness could be removed without sacrificing efficiency [Baker93SP]. Our linear FRPOLY (which includes GC effort) is a great success—it runs only 6% slower than the standard benchmark (which does no GC) in only 1/10 the space. The linear FRPOLY is more cache-friendly, and will be faster on a modern cacheful RISC ([Nishida90] backs up this intuition).

Boyer [Gabriel85] performs a Prolog-like rule-directed rewriting that accounts for nearly all of its time and space. Boyer is a "worst-case" for the linear style, because most rule matches fail, leaving partially-matched structures which must either be thrown away or reconstituted, and even successful rule matches throw away the rest of the rules in a list. A linear Boyer (which includes GC effort) was 1.59X slower than a standard Boyer (which does no GC), although the linear Boyer does 1/5 as many conses. We also tested linear and non-linear Boyers with rules *compiled* by an optimizing rule compiler. This compiled linear Boyer (with GC effort) was 1.02X *faster* than a non-linear compiled Boyer (no GC), with only 28% of its conses. When the standard Boyer fails to match a rule, it copies the expression anyway, producing an unshared result which makes the linear Boyer look good. We therefore tested Boyers which conserve sharing, and found that the result was compressed by a factor of 7.85. With this change, the non-linear Boyers (with no GC effort) speeded up by 15-30%. We then used reference counts to represent sharing in the linear Boyers, and even when *deferred* pointers [Baker94MinRC] were used to minimize count updating, the linear reference count Boyers (which do GC) were 25-27% slower. It is not linearity, but uncounted GC effort and count updating (inefficient on current architectures) that are to blame for this slower performance.

Our experiments show that the linear style is easy to write, and can save time and sometimes a dramatic amount of space. The best efficiency comes when linearity is known at compile time. In such cases, the compiler can arrange for cells which are dissolved to be immediately reused without first explicitly putting them onto the freelist [Wakeling91]—e.g., the linear rule compiler can reuse the cells from the matched expression in the construction of the result expression. This powerful optimization saves not only freelist linking time, but also the redundant tracing time that is required by a garbage collector or a count decrementer to recursively decompose the structure. Unfortunately, this reuse optimization is not applicable to reference-counted cells, whose counts may be >1. Thus, the lack of reuse and the additional count checking causes our reference-counted linear compiled Boyer to run at virtually the same speed as the strictly linear compiled Boyer, even though the reference-counted Boyer uses only 1/7'th the space of the strictly linear Boyer.

PREVIOUS WORK

The explicit killing of linear variables is reminiscent of the automatic *finalization* found in languages like C++. Linear objects are much more dynamic than the local finalized variables of C++, however, and can therefore be used to communicate results to a caller as well as act as "envelopes" for messages and "argument lists" in remote procedure calls in distributed computation.

There has been much heat, but relatively little light, shed during the past 25 years on the proper treatment of aliasing in the presence of side-effects. It is becoming clear that the static detection of aliasing in general is hopelessly exponential—e.g., sharing analysis for functional languages appears to be at least as hard as ML type inference [Baker90], which is already exponential [Mairson90]. Sharing analysis for non-functional languages is assumed to be even harder. The alternative of making sharing an explicit part of the programmer's type system has only recently gained momentum [Lafont88] [Hogg91] [Wakeling91].

Hoare's concept of *monitors* [Hoare74] can be considered an early "near-miss" to the concept of a non-linear object with a linear representation. Monitors "allocate" and "schedule" resources, but the resources themselves are only referenced indirectly. A linear object which references a 1 megabyte array in memory, on the other hand, is not the *manager* for this storage—it actually *is* that storage.

Ada's *limited* (non-assignable) types [AdaLRM83] were an unsuccessful early attempt at the concept of a linear type. Limited types did not appear in requirements documents, but were introduced as an *ad hoc* solution to the problem of inadvertently losing file descriptors and tasks. Because Ada refused to first factor assignment into copy+kill+move, the resulting rules for limited types became overly complex, yet they still failed to solve the problem they were introduced to solve. Curiously, the original Ada9X proposal for 'controlled' types [Taft91] (Ada's second unsuccessful approximation to linear types) focussed on the correct problems—duplication and deletion methods—but the final Ada9X proposal [Taft94] unfortunately followed the C++ model. Thus, Ada became saddled with all of the complexity of limited and controlled types, but neither provided efficient solutions to aliasing, resource management or synchronization.

A few voices cried out in the wilderness. [Kieburtz76] and [Francis83] discuss ways to manipulate linear structure objects without using the concept of pointers. [Bawden86] describes connection graphs in which duplication and deletion must be explicit. *NIL* [Strom85] (now *Hermes* [Lowry92]) introduced the concept of *typestates* [Strom86], which are a way to check for linearity in a language with *regular* (i.e., `if`,`while`) control structures. *Janus* [Saraswat90] is a logic language with only producer-consumer (linear) variables. The *Linda* language is based on a shared database of unshared (linear) structures [Carriero89]. [Hogg91] discusses "unique" (uniquely referenced) types.

Unshared references have long been used by language implementors. [Collins60] does *not* provide a reference count field for unshared objects; counts for shared objects are stored in invisible indirect nodes. Collins's *lazy reference counts* were rediscovered by [Goto88]. [Clark78] shows that the vast majority of Lisp cons cells are unshared. [Wise77] shows how the singly-referenced/multiply-referenced distinction ("1-bit reference count") can be used to promptly reclaim a significant fraction of garbage cells. [Stoye84] shows how the 1-bit reference count is most efficient as part of the pointer, thus becoming a dynamic type bit. [Chikayama87] [Goto88] [Inamura89] and [Nishida90] show the application of this "multiple reference bit" (MRB) to logic programming languages. [Hudak86] uses abstract interpretation to *infer* non-sharing for Quicksort, while [Guzmán90] uses an extended ML type system and type inference for a similar purpose. Explicit linear types achieve 90% of the goals of "effect" types [Lucassen87] with only 10% of the effort. [Wadler91] [Chirimar92] use *implicit* linear types to solve the array update problem. "Eval server" parallel Lisps [Yuasa89] and parallel Lisps that copy [Kessler89] traffic in linear objects. [Baker91] discusses how unshared objects do not require forwarding pointers during copying garbage collection, and [Wise92] restores maybe-shared pointers to unshared pointers by means of garbage collection.

CONCLUSIONS

Explicit linear types for programming languages have had a long gestation, and are now ready to enter the workaday world. They provide a rare combination of efficiency and elegance, by dispensing with the one-size-fits-all, everthing-is-first-class philosophy. Most volatile objects cluttering up generational garbage collector nurseries can be more efficiently handled as linear objects—e.g., mathematical objects (bignums, complex numbers, vectors, matrices, polynomials), intermediate and final objects produced by Lisp's `read` function, reified language implementation objects (argument lists, result lists, heaps, freelists, continuations), etc. Linear types offer trivial, user-controllable solutions to the aggregate initialization and update problems of functional objects. Linear types are important in *fractal* (useful/efficient at many scales) parallel, distributed, and systems languages for avoiding synchronization.

Due to their destructive reading, linear variables require a new programming style, for which *linearity-checking* by the compiler is necessary. Because linear types are very visible to the programmer, it makes no sense to infer linearity *á la* ML type inference, although some linear/non-linear polymorphism may prove useful. Far from being obtrusive, we have found the additional redundancy of linearity to be helpful in finding bugs—e.g., there must be a good reason if an argument to a function is not used within its body. Linear types do require cooperation from the language implementation during exception handling and other non-local exits, but this cooperation is similar to that required by C++ destructors.

It is time to beat turnstiles into technology. The elegance and efficiency of linear types should allow functional languages to finally go "main-stream".

REFERENCES

AdaLRM: *Reference Manual for the Ada® Programming Language*. ANSI/MIL-STD-1815A-1983, 1983.

Appelbe, W.F., and Ravn, A.P. "Encapsulation Constructs in Systems Programming Languages". ACM *TOPLAS 6*,2 (Apr 1984), 129-158.

Baker, H.G., and Hewitt, C. "The Incremental Garbage Collection of Processes". *Sigplan Not. 12*,8 (Aug. 1977).

Baker, H.G. "Unify and Conquer (Garbage, Updating, Aliasing, ...) in Functional Languages". ACM *LFP*. (1990).

Baker, H.G. "Cache-conscious copying collectors". *OOPSLA'91 GC Workshop*, Oct. 1991.

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

Baker, H.G. "Equal Rights for Functional Objects". ACM *OOPS Mess. 4*,4 (Oct. 1993), 2-27.

Baker, H.G. "The Boyer Benchmark Meets Linear Logic". ACM *Lisp Pointers* VI,4 (Oct/Dec 1993), 3-10.

Baker, H.G. "Sparse Polynomials and Linear Logic". ACM *Sigsam Bull.* 27,4 (Dec 1993), 10.

Baker, H.G. "A 'Linear Logic' Quicksort". ACM *Sigplan Not*. 29,2 (Feb 1994), 13-18.

Baker, H.G. "Linear Logic and Permutation Stacks—The Forth Shall Be First". ACM *Comput. Arch. News* 22,1 (Mar 1994), 34-43.

Baker, H.G. "Minimizing Reference Count Updating with Deferred and Anchored Pointers for Functional Data Structures". *Sigplan Not. 29*,9 (Sept 1994), 38-43.

Bawden, A. "Connection Graphs". *Proc. Lisp & Funct. Progr. 1986*, 258-265.

Cardelli, L, and Wegner, P. "On Understanding Types, Data Abstraction, and Polymorphism". ACM *Comput. Survs. 17*,4 (Dec. 1985), 471-522.

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

Chikayama, T., and Kimura, Y. "Multiple reference management in flat GHC". In Lassez, J.-L., *ed. Logic Programming, Proc. 4th Intl. Conf.* MIT Press, Camb., MA, 1987, 276-293.

Chirimar, J., *et al*. "Proving Memory Management Invariants for a Language Based on Linear Logic". *ACM LFP*, June, 1992, also ACM *Lisp Pointers V*,1 (Jan.-Mar. 1992), 139.

Clark, D.W., and Green, C.C. "A note on shared list structure in Lisp". *Inf. Proc. Lett. 7*,6 (Oct. 1978), 312-315.

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

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.

Friedman, D.P., and Wand, M. "Reification: Reflection without Metaphysics". *Proc. LFP 1984*, 348-355.

Gabriel, R.P. Performance and Evaluation of Lisp Systems. MIT Press, Camb., MA 1985.

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

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

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

Goto, A., *et al*. "Lazy reference counting: An incremental garbage collection method for parallel inference machines". *Proc. Intl. Conf. on Logic Programming*, MIT Press, 1988, 1241-1256.

Gudeman, D., *et al*. "`jc`: An Efficient and Portable Sequential Implementation of Janus". *Logic Programming, Proc. Jt. Intl. Conf. & Symp.* MIT Press, 1992, 399-413.

Guzmán, J.C., and Hudak, P. "Single-threaded polymorphic lambda calculus". *5th IEEE LICS*, 1990, 333-343.

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

Harper, R., *et al*. "Standard ML". TR ECS-LFCS-86-2, Comp. Sci. Dept., Edinburgh, UK, Mar. 1986.

Haynes, C.T., & Friedman, D.P. "Engines Build Process Abstractions". *Proc. Lisp & Funct. Progr.*, 1984, 18-24.

Haynes, C.T., & Friedman, D.P. "Embedding Continuations in Procedural Objects". ACM *TOPLAS 9*,4 (1987).

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

Hoare, C.A.R. "Monitors: An operating system structuring concept". *CACM 17*,10 (Oct 1974), 549-557.

Hogg, J. "Islands: Aliasing Protection in Object-Oriented Languages". *OOPSLA'91, Sigplan Not. 26*,11 (1991), 271.

Hudak, P. "A Semantic Model of Reference Counting and its Abstraction". *Proc. LFP 1986*, 351-363.

Imai, A., and Tick, E. "Evaluation of Parallel Copying Garbage Collection on a Shared-Memory Multiprocessor". *IEEE Trans. Par. & Distr. Sys. 4*,9 (Sept. 1993), 1030-1040.

Inamura, Y., *et al*. "Optimization techniques using the MRB and their evaluation on the Multi-PSI/V2". In Lusk, E.L., and Overbeek, R.A., *eds*. *Logic Programming, Proc. of North American Conf.* MIT Press, 1989, 907-921.

Ito, T., and Halstead, Jr., R.H. *Parallel Lisp: Languages and Systems*. Springer-Verlag, Berlin, LNCS 441, 1989.

Kessler, R.R., and Swanson, M.R. "Concurrent Scheme". In [Ito89], 200-235.

Kiczales, G., *et al*. *The Art of the Metaobject Protocol*. MIT Press, Camb., MA, 1991.

Kieburtz, R.B. "Programming without pointer variables". *Proc. Conf. on Data: Abstraction, Definition and Structure, Sigplan Not. 11* (special issue 1976), 95-107.

Lafont, Yves. "The Linear Abstract Machine". *Theor. Computer Sci. 59* (1988),157-180.

Lincoln, P., and Mitchell, J.C. "Operational aspects of linear lambda calculus". *Proc. 7th IEEE Logic in CS*, 1992.

Lowry, A. "The Hermes Language in Outline Form". *Sigplan Not. 27*,8 (Aug. 1992), 51-70.

Lucassen, J.M. *Types and Effects: Towards the Integration of Functional and Imperative Programming*. PhD Thesis, MIT/LCS/TR-408, MIT, Aug. 1987.

Mairson, H.G. "Deciding ML Typeability is Complete for Deterministic Exponential Time". ACM *POPL 17* (1990).

Mendelson, A. "A Single Cached Copy Data Coherence Scheme for Multiprocessor Systems". *Comput. Arch. News 17*,6 (Dec. 1989), 36-49.

Nishida, K., *et al*. "Evaluation of MRB garbage collection on parallel logic programming architectures". *Proc. Intl. Conf Logic Programming*, MIT Press, 1990, 83-95.

Odersky, M. "How to Make Destructive Updates Less Destructive". *Proc. ACM POPL 18*, Jan. 1991, 25-36.

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

Saraswat, V., *et al*. "Janus: A step towards distributed constraint programming". *Logic Programming, Proc. of 1990 North Amer. Conf.*, MIT Press, Camb., MA, 1990, 431-446.

Steele, G.L. "Data Representations in PDP-10 Maclisp". AI Memo 420, MIT AI Lab., Camb., MA, Sept. 1977.

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

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

Stoye, W.R., *et al*. "Some practical methods for rapid combinator reduction". ACM *LFP 1984*, 159-166.

Strom, R.E., & Yemini, S. "The NIL Distributed Systems Programming Language: A Status Report". *Sigplan Not. 20*,5 (May 1985), 36-44.

Strom, R.E. & Yemini, S. "Typestate: A Programming Language Concept for Enhancing Software Reliability". *IEEE Trans. SW Engrg. SE-12*,1 (Jan. 1986), 157-171.

Taft, S.T., *et al*. Ada9X Mapping Documents. DoD, Wash., DC, Feb. 1991.

Taft, S.T., *et al*. Ada9X Reference Manual, v. 5.0. Intermetrics, Inc., Camb., MA, June, 1994.

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. FPCA*, LNCS 523, Springer-Verlag, 1991, 215-240.

Wand, M. "Continuation-based Multiprocessing". *Proc. of Lisp Conf.*, Stanford, CA, 1980, 19-28.

Wise, D.S., and Friedman, D.P. "The one-bit reference count". *BIT 17*,3 (Sept. 1977), 351-359.

Wise, D.S. "Stop-and-copy and One-bit Reference Counting". *Info. Proc. Lett. 46*,5 (July 1993), 243-249.

Yuasa, T. "Design and Implementation of Kyoto Common Lisp". *J. Info. Proc. 13*,3 (1990), 284-295.

Yuasa, T., and Kawana, T. "PM1 and PMLisp: An Experimental Machine and Its Lisp System for Research on MIMD Massively Parallel Computation". In [Ito89], 322-347.

