SETL Newsletter # 153

J.T. Schwartz
July 15, 1975

The significance of 'Backtracking', and its Cost.

It is noted in NL 135A that, as a program evolves from its underlying 'rubble' or 'proto-program' toward efficiency, special data arrangements and combinatorial structures which support efficient processing will commonly be introduced. These structures may initially be defined by logical predicates which single them out from some extremely large collection of related structures. To actually build these structures some procedure is needed, and this will generally take one of two forms, a logically efficient form when available, a less efficient form otherwise.

To construct a structured object x efficiently, some variant of the following paradigm, which leads ultimately to procedures consisting of nests and sequences of while-loops, will be used. Invent a set s containing x and a mapping f of s into itself such that the equation f(x') = x' implies that x' satisfies the predicate C(x) defining x. Then pick some $x_o \in s$, and iteratively form the sequence $x_o, f(x_o), f^2(x_o), \ldots$ until it stabilizes, and which point an element satisfying the target predicate will have been constructed.

If in a particular case a desired structure x cannot be found using this kind of 'step to the goal' approach, some much less efficient search technique has to be used. To locate x within s in such cases, one typically starts with some x_o and with a *family* of mappings $f_1, \ldots, f_n$ having the property that by applying the f_j to x_o repeatedly in all possible ways one will eventually generate the whole of s. Then generation begins, and continues till x is found. Various well-known methods to improve the efficiency of such a search exist. For example, whenever a new element $y = f_{i_1}(f_{i_2}(\ldots(f_{i_k}(x_o))\ldots))$ is

generated, one can

a. Make some calculation which determines whether the desired x is reachable from y by the maps f_j, and, if not, suppress y.

b. Determine by calculation that particular maps f_j cannot usefully be applied to y, and suppress the formation of $f_j(y)$ when this is true.

c. In some way, estimate the number of mapping steps that separate y from the desired x, and always work forward from the 'most promising' y.

Procedures based on such a 'guided and pruned search' approach may be said, irrespective of the particular details of their construction, to employ 'backtracking'. (In connection with applications having a numerical flavor, this is sometimes called 'branch and bound' instead.) A 'backtracking' approach is generally much less efficient than procedures of 'step to the goal' type, and for this reason backtracking techniques are avoided when possible; hence in applications they occur less frequently than 'step to the goal' constructions. Nevertheless the use of backtracking will sometimes be unavoidable, for which reason methods for efficient implementation of backtrack, and for the optimization of backtrack programs are of interest.

We first consider the question of efficient implementation of backtrack, and focus first of all on 'strict backtrack' programs, i.e., programs which progress through a tree of data environments, occasionally 'failing' and returning to continue from the immediate ancestor environment of the environment in which failure has occured. To implement this restricted form of backtracking, one logically keeps a stack of environments, saving an environment on it each time the program being run makes a nodeterministic choice, and popping one off each time execution resumes after a failure.

To assess the efficiency of any backtrack implementation, we will compare it to an 'ideal' implementation in which whole data environments can somehow be saved and restored instantaneously. Note that it is desirable that a backtrack system should perform well even if new environments are frequently created, and then quickly found to fail.

The complexity of the problems which will arise in implementing backtracking depends on the characteristics of the language into which backtracking is to be installed. As a first case for study, let us consider a language, something like ALGOL 68, which supports recursion using a stack, and which also provides a garbage collected heap H, holding vectors. We assume that the size of these vectors does not change from the time at which they are allocated up to the time at which they are garbage collected.

A possible implementation of this semantics is as follows: In addition to the main stack MS, keep an auxiliary stack AS. When a new environment is opened, a changed_stack_part pointer CSPP is set to point to the top of MS. Whenever an item of MS is popped or changed for the first time after opening a new environment (which is shown by the position of CSPP), CSPP is lowered to point to this item, and the old value of the item transferred to AS.

To each vector V stored in H we can attach a chain VCH of auxiliary areas AAV of size proportional to that of V. Whenever a new environment NE is opened, and V changed for the first time in this environment, a new block CAAV can be attached to the chain VCH. Then, whenever a component V(i) of V is changed, and provided that it is the first change to V(i) after the opening of NE, the old value of V(i) can be recorded in CAAV. (With each vector component V(i) we associate an auxiliary field giving the serial number of the last environment for which the value of V(i) has been stored in an auxiliary area, and with V as a whole we associate a field defining the last environment in which <u>any</u> component of V was changed.)

Subsequently, when an old environment OE is restored after a failure, and V first accessed in this old environment, the former value of V can be restored using this saved information.

A data structure built up according to these rules can be garbage collected; however, no items reachable from either the main stack MS, the auxiliary stack AS, or thru either a reachable vector V or thru any block AAV of its auxiliary chain can be destroyed. Because of this, data will tend to accumulate as one progresses into a deep nest of environments, and therefore a garbage collector routine may find itself unable to recover any significant amount of space. When this happens, the following technique for spilling excess data to secondary memory might be used.

(a) The total amount of memory occupied by auxiliary blocks generated within each ancestor environment of the current environment CE will always be known. Using this information, one can locate an ancestor environment E_o of CE, where E_o should have the following property: a substantial amount of memory is occupied by auxiliary areas associated with environments AE ancestral to both E_o and CE.

(b) Once the environment E_o is defined, the data state belonging to E_o can be rebuilt and written out to secondary memory (after a suitable compacting garbage collection). Then the set PRIORAAV of all auxiliary blocks belonging either to the environment E_o or to environments ancestral to E_o can be erased, following which we carry out a final garbage collection, which may be able to drop blocks which a garbage collection carried out before erasure of the members of PRIORAAV would have had to retain.

Note however that when one uses the 'strict' backtracking mechanism that has been under consideration is used deep sequences of environments should rarely develop, since search of deep trees of choices is likely to be unfeasible.

If, basing ourselves on this observation, we ignore the time required for sectondary memory spill operations, and assume that garbage collection in a backtracking environment will not be substantially more expensive than garbage collection in a conventional environment, it follows that most of the efficiency impact of backtracking comes from the way that vector component load and store operations are affected.

In the backtracking implementation sketched above, retrieval of a vector component V(i) is performed in exactly the same way as if no backtracking were involved. However, indexed store operations are more complicated, and are implemented as follows:

(a) Before performing a store operation V(i) = ..., check the auxiliary field associated with the component V(i). The value A found in this field defines the last data environment within which V(i) was changed. If A equals the index CE of the current data environment, perform the store operation in the ordinary way.

(b) Otherwise save the old values of V(i), A, and the index i, all in the auxiliary area AAV associated with V, and replace A by CE. Moreover, check the V-associated field value A' which gives the serial number of the last environment in which any component of V was changed. If A' is different from CE, save A' in AAV, put AAV on a *restore list* associated with the current environment, and replace A' by CE. When a failure occurs in a given environment CE, we consult the restore list of CE, and using it restore each vector V which changed since CE was opened returning it to the condition in which V was before CE was opened. This is done simply by replacing the current value of every component V(i) of V for which a value VPRIOR(i) has been saved in AAV; V(i) takes on the saved value VPRIOR(i).

The implementation scheme which has just been outlined should have small impact on the time required to perform indexed load operations, and might approximately double the time required to perform indexed store operations (which will require range-checking whether or not backtracking is supported.) Reckoning stores as 1/4 of all operations performed, we see that to provide 'strict' backtracking in the assumed semantic environment (i.e., recursion and fixedsize vectors in a heap) should approximately double run times. (Garbage collection difficulties could change this estimate.)

Next we consider a somewhat more challenging semantic environment like that of (copy optimized) SETL, in which vectors are used to store set-representing hashtables occasionally requiring rehashing, and in which vectors are allowed to grow dynamically. However, we still continue to assume that only strict backtracking, and not more complicated forms of environment manipulation, are allowed. First consider vectors, which can grow by concatenation operations

(1) $$V = V + \langle x \rangle.$$

If in a particular case a logical copy of V needs to be formed to perform this operation, preparing for backtrack is no problem, since the new copy of V is allocated within the current environment. Suppose on the other hand that (1) can be performed destructively. An implementation supporting variable length vectors will store the length of a vector V with V in some auxiliary component which for notational convenience we shall write as V(0). The concatenation operation (1) can then be handled in the manner suggested by the code sequence V(0) =V(0) +1; V(V(0)) = x and by our earlier discussion. When a vector used destructively expands beyond the space allocated for it and must be recopied, the pointer in the (necessarily unique) reference to it can simply be changed to point to the longer copy.

If after backtracking this copy of V turns out to be unnecessarily long (which might be discovered either during garbage collection or when, after backtracking to a prior environment, a component is concatenated to V) then some appropriate part of the superfluous space occupied by V can be released to the space allocator.

Next we shall describe a way in which sets can be treated in an implementation which supports strict backtracking. For sets not used as maps, the crucial operation are

(2) s with x

and

(3) s less x.

A reasonable way to proceed is as follows. With each set s associate an auxiliary chain having one node for each environment. To each node, attach an 'additions set' *sa* and a 'deletion set' *sd* (if both these sets are null, the node to which they would be attached can be omitted.) All nodes belonging to an environment e are linked together in an associated *restore list* RL(e). To perform the operation s with x, we first make the test x∈s; if this yields true, nothing remains to be done. Otherwise we insert x in s, and make the test x∈sd. If x∈sd, we delete x from sd; otherwise we insert x into sa.

To perform s less x, we make the test x∈s; if false, nothing remains to be done. Otherwise we delete x from s, and make the test x∈sa. If x∈sa, we delete x from sa; otherwise, we insert x into sd.

These operations should be no worse than twice as slow as the same operations as performed by non-backtracking SETL. Moreover, backtracking should not seriously hamper our ability to perform operations destructively (i.e., without making full copies of large compound objects).

Next consider the case of a set s used as a map, for which the crucial operations are

(4) $s(x) = y;$

and more generally

(5) $s(x_1,\ldots,x_n) = y;$

To handle (4), we first test to see if $\langle x,y\rangle \in s$ and also if s{x} contains only a single element; in this case, there is nothing to do. Otherwise, if $\langle x,y\rangle$ not $\in s$ then we remove $\langle x,z\rangle$ from s for each $z \in s\{x\}$ and $\langle x\rangle + z$ from s for each tuple $z \in s\{x\}$. Then we test each element t removed from s to see if $t \in sa$; if so, we remove it from sa, if not, we add it to sd. Finally, we insert $t=\langle x,y\rangle$ into s, and test t to see if $t \in sd$; if so, we remove it from sd, otherwise add it to sa. If $\langle x,y\rangle \in s$ but s{x} contains elements z different from y, we remove $\langle x,z\rangle$ from s, and also $\langle x\rangle + z$ from s if z is a tuple. We test each element t removed from s to see if $t \in sa$; if so, we remove it from sa, if not, add it to sd. The operation (5) is handled in a very similar way.

These operations also should be no more than twice as slow as the same operations in non-backtracking SETL. Overall then, we estimate that a version of SETL supporting 'strict' backtracking can run at approximately one half to one third the speed of deterministic SETL. Note that in backing out of an environment e we examine all the items on the restore list RL(e); using the information which these items record, we restore every data object to the condition which it had in the parent environment of e.

Generalised backtracking. Next we consider the implementation impact of 'generalised' backtracking, i.e., the overhead cost of a system which admits data environments as semantic objects that can be manipulated explicitly. (An example of a language which allows this is G. Sussman's CONNIVER.) Since for reasons already explained a 'differential' implementation is desired, we assume that the data environments to be manipulated form a tree.

Let e be the parent node of e' in this tree; then the logical situation which prevails in e' is, with the exception of changes explicitly recorded in the implementation level representation of e', exactly that which prevails in e. Inter-environment transition is assumed to be accomplished by use of the following coroutine-call like primitive:

valuereturned = cocall (newenvironment, valuesent).

This cocall exits from the current environment *ce*, enters *newenvironment*, and leaves *ce* suspended 'halfway thru' the cocall, ready to receive a value back when and if control is eventually returned to *ce*. Note accordingly that with the exception of the one environment that is currently running all other environments will represent processes which have lost control by executing some cocall from which they are awaiting a value. The value transmitted by *ce* to *newenvironment* when the cocall shown above is executed is the pair <ce, valuesent>.

A complete data environment consists of a control stack, a logical address structure which changes dynamically as recursive calls and returns are made, a map which binds program variables to addresses, and a map binding addresses to values. If actions taking place in one environment e are allowed to have side effects on the values which these maps have in any other environment e', then one must define rules determining the way in which all these maps and structures are affected. To avoid the issues in which this would involve us, we shall simply rule out inter-environment side effects by insisting that control can only pass to an environment e (by a cocall) if in the tree of all environments e has no descendants.

We assume two mechanisms for creating new environments. The first, which we will write in the form

env' = copy (environment),

generates an exact copy of *environment* and assigns it to *env'*. The newly created *env'* is an immediate descendant of *environment* in the tree of all environments. The second way of creating a new environment is to make the cocall

cocall (Ω, valuesent).

This generates copy *env* of the currently running environment *env*; *env'* becomes an immediate descendent of *env* in the environment tree, and control passes to *env'*.

A primitive which allows environments to be destroyed is also required. We shall assume that this is written

destroy(env),

and that it destroys *env* and all its descendants in the environment tree. Note that by destroying *env* we can make its parent environment executable again.

Now suppose that the generalised backtracking primitives that have just been described are to be supported in a language providing both recursion and a garbage collected heap H. We first assume that the heap H holds vectors whose size does not change from the time at which they are allocated up to the time at which they are garbage collected. We shall describe an implementation of this semantics, and first discuss the way in which vectors V stored in H are to be represented. A possible approach is this: number environments as they are generated, and with each environment e record both its serial number n(e) and a bit-vector b(e) which shows (the serial number of) all environments ancestral to e. With each component V(i) of each vector V, associate a bitstring b(V,i) showing (the serial numbers of) all environments in which V(i) has been changed, and a hash table h(V) in which all these changes are recorded (as triples <e,i, value_assigned_to_V(i)_in_e>).

If these conventions are used, retrieval of a vector component is performed by forming the bitvector b(e)∧b(V,i), locating the position e' of its most significant nonzero bit, and locating a triple <e',i,x> in the hash table h(V); x gives the value of V(i) in the environment e. Note that if not more than 32 or so environments are being manipulated at any one time (which should generally be the case) the bitvectors b(e) and b can be held in a single word. When more environments than this must be managed, sections of these bitvectors can be held in a list of words (words which consist exclusively of zeroes being skipped).

To perform the storage operation V(i) = y in an environment e, one first checks the appropriate bit of b(V,i) to see if V(i) has yet been changed in e. If not, this bit is set, and an entry <e,i,y> made in h(V). Otherwise an existing entry <e,i,x> is located in h(V), and x changed to y.

When and if environments are destroyed in an order differing from the inverse of their creation order, gaps will appear in the sequence of identifying serial numbers assigned to environments, which means that some bit-vector positions will temporarily fall out of use. However, the next following garbage collection will examine all accessible vectors V, and after this can repack all bit-vectors b(V,i), eliminating unused bit positions; at the same time, environment serial numbers e appearing in triples <e,i,x> can be changed to their new values.

A scheme for handling recursion in languages providing generalised backtracking has been described by Bobrow and Wegbreit. The following is also a plausible stack management scheme: With each environment e, maintain a stack section vector SS(e), representing the condition of the stack, in the environment e, between an upper limit U(e) and lower limit L(e).

When the environment e is first created, SS(e) will be null and U(e) will equal L(e). Stack locations between U(e) and L(e) will be read from and written to SS(e); whenever a stack location ℓ below U(e) is consulted, all stack locations from U(e) to ℓ will be copied from the stack section SS(e') associated with an appropriate ancestor e' of e to SS(e), and U(e) will be lowered to ℓ.

Data will tend to accumulate as multiple coexisting environments are built up, increasing the cost of garbage collection and making it impossible for the garbage collector to recover any significant amount of space. This means that it is important to develop some technique allowing excess data to be spilled to secondary memory. However, we shall not attempt to discuss this problem now.

The implementation scheme which we have just outlined retrieves vector components V(i) by hashing. This should be about 20 times slower than a machine level indexing operation, and about 7 times slower than a range-checked indexing operation. About the same figures should apply to indexed store operations. Thus an ALGOL 68 variant providing generalised backtracking should run approximately 7 times slower than normal ALGOL 68. (Note however that this estimate ignores garbage collection difficulties which may develop for programs manipulating large numbers of environments.)

Next we consider the additional problems which arise in a SETL - like semantic millieu which provides vectors that can grow dynamically and also provides set-representing hashtables.

Vectors V of dynamically variable length do raise some problems, but none terribly severe. The vector of bitstrings b(V,i) associated with V must always be maintained for all i from 1 to the greatest length which V has in <u>any</u> existing environment. A nominal component V(0) will then store V's actual length (in each particular environment.) During garbage collection all the entries in the V-associated hash table h(V) will be examined, and this will reveal the actual length which V has in any currently existing environment, after which any

superfluous bitstrings b(V,i) can be handed back to the space allocator.

Next let us consider the way in which sets, both sets used as collections and sets of tuples used as mappings, can be handled. Here more difficulties seem to be encountered, and considerably more thought is needed; but the following remarks will begin to suggest the outlines of one possible approach. Let s be a set, and first suppose that s does not contain any tuples. We can represent s, or more precisely the various different values that s will have in different environments e, by using two mappings f_s and g_s. The mapping f_s is defined for every element x that belongs to s in any environment e, and maps x into a pair of bitstrings b_1 and b_2, whose individual bit positions refer to particular environments. (We continue to assume that serial numbers are assigned to environments e as they are generated.) The e-th bit of b_1 will be nonzero of and only if x was either added to or deleted from s in the environment e, and the e-th bit of b_2 will distinguish the 'addition' from the 'deletion' case. Using the map f_s, a membership test $y \in s$ is performed as follows: Calculate $f_s(y)$; if it is undefined then $y \in s$ is false. Otherwise $f_s(x)$ gives two bitvectors b_1 and b_2; form $b' = b(e) \wedge b_1$, where as before b(e) is a bitvector showing (the serial numbers of) all environments ancestral to e; and let e' be the position of the most significant nonzero bit of b'. Then $y \in s$ is true if and only if bit e' of b_2 is nonzero.

The map g_s sends each environment e into a list of all the elements added to s in the environment e, and is used to expedite iterations over s. Let $e_1, \ldots e_n$ be the environments ancestral to e (we include e itself as $e_n = e$). To carry out the iteration $\forall x \in s$, we let x vary over the list $g_s(e_j)$ for all $j = 1, \ldots, n$ in turn.

For each x on any of these lists, we make the test $x \in s$ in the manner explained in the preceeding paragraph; if it succeeds (and if x is not a member of the set aux(e) introduced below) then x legitimately belongs to the range of the iteration $\forall x \in s$; otherwise not. When, during the iteration $\forall x \in s$, a particular element x is reached, we add x to an auxiliary set aux(e) associated with the iteration $\forall x \in s$ and with the current environment e (the set aux(e) is initialised to null at the start of the iteration $\forall x \in s$). As elements $x \in s$ are reached in the course of iteration the test $x \in aux(e)$ is made, and x is bypassed if this test succeeds since x will have been reached at least once before in the same iteration.

We shall call the representation of s that has just been described its *standard multienvironment representation*. Additional complications arise in connection with sets containing tuples, since these sets can be used as maps, and therefore operations like $s\{x\}$ and $s\{x_1,\ldots,x_n\}$ must be supported efficiently. Suppose, to begun with, that such a set s contains ordered pairs but no ordered triples, so that only the operation $s\{x\}$ comes in question. Then we can keep s (or rather that part of it consisting of ordered pairs) as a hash-table-based mapping S, where, for each x, S(x) is exactly the standard multienvironment representation of $s\{x\}$. Then $s\{x\}$, s(x), etc., can easily be calculated from S(x) in any given environment; and the membership test $\langle x,x' \rangle \in s$ can be executed as $x' \in S(x)$ (which of course is executed in the manner appropriate for sets having standard multienvironment representation, which we have just explained). In order to expedite the iteration $\forall y \in s$ we will probably also want the representation of s to include an auxiliary mapping h_s which sends each environment e into the list of all x which first enter the domain of S in the environment e.

Note however that if it is known *a priori* that s will never be iterated over, then we can suppress part of the representation that has just been described, thus saving both time and space.

Other facts deducible by global program analysis will also make significant efficency improvements possible even in a generalised backtracking environment. Suppose, for example, that s is known to be a set of ordered pairs used only as a map, and that s is also known to be single-valued (in every environment.) Then s can be represented in a manner resembling the representation of vectors suggested a few paragraphs above. Specifically, with each x which belongs to the domain of s in any environment, we associate a bitstring b(s,x) showing the serial numbers of all environments in which s(x) has been changed; we also store a map h(s) in which all these changes are recorded (as triples <e x, value-of-s(x)-in-e>). Then to retrieve or to modify the value s(x) we proceed in much the same way as for vectors. Observe that when this representation is used, the time required to retrieve or store s(x) should not be more than three times the time required for these same operations in a nondeterministic SETL environment.

However, let us return to our discussion of set representations in the general case. We have described a way in which sets consisting of non-tuple elements, plus tuples of length at most 2, can be represented. We shall continue to call the representation that was described for these sets their 'standard multienvironment representation'. To handle sets s containing tuples of more general type, we can simply proceed inductively, associating with s a map S such that, for each x, S(x) gives the standard multienvironment representation of s{x} Then to evaluate $s\{x_1,s_2\}$, one calculates $(S(x_1))(x_2)$, etc. If s must support iteration, we may also want the representation of s to include an auxiliary mapping which sends each environment e into the list of all x which enter the domain of s for the first time in the environment e.

The time required to calculate $s\{x_1,\ldots,x_n\}$ from the multienvironment representation of s should stand in the same proportion to the time needed for an ordinary SETL calculation of $S\{x_1,\ldots,x_n\}$ as the time needed for a multi-environment calculation of $s\{x_1\}$ stands to the time needed for an ordinary calculation of $s\{x_1\}$. A very crude estimate of this latter time ratio can be obtained as follows. To calculate $s\{x_1\}$ in an environment e, one will have to examine all the elements y which belong to s{x} in any environment ancestral to e, and determine which of these y still belong to s{x} in e. To examine a single element y, and to insert it into a developing representation of s{x} if this is needed, should take roughly twice as long as the like operations would take in a non-backtracking environment. Thus to evaluate s{x} in a generalised backtracking environment should be roughly 2/P as expensive as it would be in a backtracking environment, where P measures the probability that an element x which belongs to s{x} in <u>any</u> environment ancestral to an environment at e lso belongs to s{x} in e. The quantity P can be regarded as a measure of the rate at which s{x} changes as one passes between environments.

In a generalised backtracking environment, the membership test $x \in s$ will involve a hashing operation followed by several bit-string operations; it should therefore have roughly half the speed of the corresponding non-backtracking test.

All in all, if we suppose that the number of environments being manipulated never comes to exceed the number of bits in a word by more than a small factor (a reasonable assumption since the space cost of maintaining a large number of environments will often be considerable), than a SETL-like language supporting generalised backtracking should run no more than in order of magnitude slower than standard SETL.

Global Optimization in Backtracking Environments.

In both 'strict' and 'generalised' backtracking environments global analysis algorithms can be expected to uncover many optimization opportuniities. One such optimization has already been sketched in an earlier discussion of the representation of single-valued maps in a generalised backtracking envoronment. Even though really accurate estimation of the value of particular optimizations will depend on the accumulation of more experience with backtracking versions of set-theoretic languages, we shall now list a few more optimizations which should be useful.

a) As already observed, each operation which a set x must support in a backtracking environment will add substantially to the size of x's representation and to the cost of every other operation involving x. Thus precise determination of the operations which will be applied to every individual object in a program is particularly important.

b) Suppose that some vector or set y can be shown by global analysis to be strictly 'local' to a single environment e, i.e., to be created within e, and to be dead at any program point at which e splits itself into two environments or passes control to another environment which might split e. Then backtracking need not be supported for y, i.e., y can be maintained in the same form that non-backtracking SETL would use, which avoids all backtracking overhead when y is accessed.

c) Suppose next that y is a set or vector-valued variable whose value is never used destructively, i.e., that whenever y is changed its value is reassigned completely. Then y can be represented using a mapping y, where y(e) is either the value of y in the environment e or is Ω if y has the same value in e as in e's ancestor environment; and also using a bit-string defining the environments e in which y has a value different from y's value in e's ancestor environment. If y is represented in this way, the time required to perform an operation on y in a backtracking environment will exceed its nonbacktracking time requirement only by a single hash-access time.

d) Next suppose that the set s is known to be a subset of another set s', i.e, that the relationship $s \subseteq s'$ is known to hold in every environment. In the standard multienvironment representation described in the preceeding section, s and s' will be represented by a pair of maps f_s and $f_{s'}$, realised by hashtables H, H', and sending element each x into either a pair of bitstrings or into Ω. If $s \subseteq s'$ is known to hold, it can be useful to conglomerate these two maps into one single hashtable $\bar{H}$, whose individual entries will then in effect be quintuples $\langle x, b_1, b_2, b_1', b_2' \rangle$, where x is any element belonging to s' in some environment, and where the bitstrings b_1 and b_2 define x's relationship to s every environment while b_1' and b_2' define x's relationship to s in every environment. This joint representation becomes useful if, for example, we wish to perform the test $y \in s$, provided that $y \in s'$ is known by global analysis, and that y is represented 'as an element of s'', i.e., by a pointer to the quintuple $\langle x, b_1, b_2, b_1' b_2' \rangle$ which defines x's relationship to s'. If y is represented in this way, then the test $y \in s$ requires no hashing operation. Note also that if the subset s'-s of s' is known to be small in most environments, then we can suppress the lists that might otherwise have to be maintained to expedite iterations over s.

Appendix: Representation of 'strict' backtracking in terms of 'generalised' backtracking .

In the present appendix we tie up an irritating loose end by explaning how the generalised backtracking primitives described in the preceeding pages can be used to represent 'strict' backtracking. The strict backtracking primitives which need to be represented are ok, which nondeterministically returns t or f, and fail, which terminates an environment.

To make strict backtracking available to a program p, we compile it with the following prologue:

```
<otherenv, valret> = cocall (Ω,Ω);
/* the preceeding cocall operation forms a second copy of the */
/* data environment of p, and returns it as otherenv. The     */
/* environment otherenv will appear to have just lost control*/
/* thru the cocall, and to be awaiting a reply. In the environment */
/* tree (which consists of precisely two nodes) the running */
/* environment will be an immediate descentant of otherenv */

if valret ne Ω then go to start;

/* in the originally running environment, this branch will not */
/* be taken. In the environment which we shall now form, it will */

otherenv = copy (otherenv);

/* this copy operation forms a copy e' of the original otherenv */
/* which can run alternately with the initially running en- */
/* vironment. After this, the original otherenv will never be */
/* used; but it remains as the root node of the tree of environments */
envstack = nult;      /* initialise environment stack */
valret = t;           /* for use in following cocall */
(while t)             /* loop until explicit quit */
```

```
        <otherenv,valret> = cocall (otherenv,valret);
    /* this transfers control to e', valret will always have the */
    /* value t or f. The t passed originally merely forces the  */
    /* conditional jump to start  (see above) to be taken;      */
    /* subsequently valret will be interpreted (in e') as the   */
    /* value returned by the function ok                        */
        if valret then /* this is the code for a call to ok in e' */
            envstack(#envstack + 1) = otherenv;
            otherenv = split (otherenv);
                    /* the split environment copy will now be run */
        else /* the fail primitive has been called in e' */
            destroy(otherenv); /* delete the failed environment */
            if envstack eq nult then quit while;; /* exit test */
            otherenv = envstack(# envstack); /* pop off old environment */
            envstack (# envstack) = Ω;
        end if valret;
    end while;
    print 'backtracking fails'; exit; /* a complete backtracking failure */
start: /* entry will be made here when the alternate environment e' */
    /* (or otherenv) runs.                                      */
    .... (here follows a body of code using the 'strict backtracking'
                                                        primitives)
```

If every program begins with the prologue shown above, then the strict backtracking primitive ok can be regarded simply as a macro for

```
(cocall (otherenv, t))(2),
```

while the fail primitive can be regarded as a macro for

```
junk = cocall (otherenv, f);
```