Using Multiset Discrimination To Solve Language Processing Problems Without Hashing [1]

Jiazhen Cai [2] *and Robert Paige* [3]

New York University/ Courant Institute, New York, NY 10012, USA

ABSTRACT

It is generally assumed that hashing is essential to solve many language processing problems efficiently; e.g., symbol table formation and maintenance, grammar manipulation, basic block optimization, and global optimization. This paper questions this assumption, and initiates development of an efficient alternative compiler methodology without hashing or sorting. The methodology rests on efficient solutions to the basic problem of detecting duplicate values in a multiset, which we call *multiset discrimination*.

Paige and Tarjan [22] gave an efficient solution to multiset discrimination for detecting duplicate elements occurring in a multiset of varying length strings. The technique was used to develop an improved algorithm for lexicographic sorting, whose importance stems largely from its use in solving a variety of isomorphism problems [2]. The current paper and a related paper [23] show that full lexicographic sorting is not needed to solve these isomorphism problems, because they can be solved more efficiently using straightforward extensions to the simpler multiset discrimination technique. By reformulating language processing problems in terms of multiset discrimination, we also show how almost every subtask of compilation can be solved without hashing in worst case running time no worse (and frequently better) than the best previous expected time solution (under the assumption that one hash operation takes unit expected time). Because of their simplicity, our solutions may be of practical as well as theoretical interest.

The various applications presented culminate with a new algorithm to solve iterated strength reduction folded with useless code elimination that runs in worst case asymptotic time and auxiliary space $\Theta(|L|+|L'|)$, where $|L|$ and $|L'|$ represent the lengths of the initial and optimized programs respectively. The previous best solution due to Cocke and Kennedy [10] takes $\Omega(|L|^3|L'|)$

1. A preliminary version of this paper appeared in the Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages[6]. Part of this work was done while both authors were visiting the University of Wisconsin at Madison, and while R. Paige was visiting DIKU at the University of Copenhagen.

2. The research of this author was partially supported by National Science Foundation Grant No. CCR-9002428 and by Air Force Office of Scientific Research Grant No. AFOSR-91-0308.

3. The research of this author was partially supported by Office of Naval Research Grant No. N00014-93-1-1036 and by Air Force Office of Scientific Research Grant No. AFOSR-91-0308.

hash operations in the worst case.

Categories and Subject Descriptors: D3.4 [Programming Languages]: Processors -- *Compilers, Optimization, Parsing, Preprocessors*; E.1 [Data Structures] - *graphs, lists, trees*; F2.2 [Analysis of Algorithms and Problem Complexity]: Nonnumerical Algorithms and Problems -- *Sorting and searching*

General Terms: Algorithms, Languages, Theory

Additional Key Words and Phrases: hashing, multiset discrimination, symbol tables, hygienic macro expansion, left factoring, value numbers, acyclic coarsest partition, sequence congruence, reduction in strength, useless code

1. Introduction.

An important practical and theoretical question in Computer Science is whether there are algorithms whose worst case performance can match the expected performance of solutions that utilize hashing. In the context of this broader question, we initiate an investigation of efficient compilation without hashing, and, consequently, raise some doubts about the prevailing view that hashing (e.g. universal hashing[7]) is essential to an efficient implementation of the various subtasks of compilation from symbol table management [3] to reduction in strength by hashed temporaries [10]. Throughout this paper we assume a sequential RAM model of computation under a uniform cost criterion [2] or the more restricted pointer machine [19,34], which prohibits address arithmetic.

Aho, Sethi, and Ullman [3] present only two data structures for storing symbol tables - a linear linked list (with linear search time) and a hash table. They also propose these two data structures for methods to turn an expression tree into a dag, and for the more general basic block optimization of value numbering. Hashing is involved in preprocessing for global optimizations to perform constant propagation [38], global redundant code elimination [5], and code motion [12]. The best algorithms for strength reduction [10,4] rely on hashed temporaries to obtain practical implementations.

There are several reasons why hashing is used in these applications. Given a set of n elements, hashing supports membership testing, element addition, and element deletion of unit-space data in $O(1)$ expected time and $O(n)$ auxiliary space. The method of universal hashing, due to Carter and Wegman [7], is especially desirable, since the expected $O(1)$ time is independent of the input distribution. Universal hashing is well suited to applications such as compilation, where a hash table used to implement the symbol table does not have to persist beyond a single compilation run. In the applications mentioned above hashing leads to simple on-line algorithms supporting immediate storage/access. Consequently, various phases of compilation can be carried out incrementally with few passes and with good space utilization.

However, liberal use of hashing incurs certain costs. Let N and M be two positive integers with $N >> M$, and consider a hash function h: $\{0,...,N\} \rightarrow \{0,...,M\}$ defined by the rule $h(x) = ((ax + b) \ mod \ N) \ mod \ M$, where input argument x and constants a and b belong to $\{0,...,N\}$. Even discounting the costs of collisions and rehashing, the calculation of a single hash operation $h(x)$ is much greater than the cost of an array or pointer access. Mairson proved that for any

’minimal’ class of universal hash functions there exists a bad input set on which every hash function will not perform much better than binary search [21]. The slow speed of SETL [31], observed in the SETL implemented ADA-ED compiler [1], has been attributed to an overuse of hashing [33]. And a hash table implementation involving an array twice the size of the data set is another cost. Arrays lack the benefits offered by linked lists - namely, easy dynamic allocation, dynamic maintenance, and easy integration with other data structures. Finally, although on-line algorithms are vital to incremental compilation, batch processing may often suffice.

This paper presents new algorithms for all of the applications mentioned above with worst case asymptotic time and space that either matches or exceeds the expected performance of the best previous algorithms that utilize hashing (under the assumption that a single hash operation takes unit time). This is achieved by reformulating these language processing problems in terms of subproblems that can be solved efficiently using several simple algorithmic tools (that exclude sorting), the most important of which is multiset discrimination; i.e., finding all duplicate values in a multiset.

In [22] Paige and Tarjan used multiset discrimination of varying length strings to design an efficient lexicographical sorting algorithm. However, the focus of that paper was on lexicographic sorting, whose importance stems largely from its use in solving a variety of isomorphism problems [2]. No other application of multiset discrimination was mentioned. In the current paper together with [23] we uncover the greater significance of multiset discrimination by showing how a straightforward extension of Paige and Tarjan’s technique to a more general class of datatypes, including sequences of trees and dags, can solve these isomorphism problems more efficiently than methods based on full lexicographic sorting. We also demonstrate how the generalized multiset discrimination method can help improve the theoretical performance of over one dozen applications to language processing, and virtually every aspect of compiling from front-end macro processing through optimization by strength reduction. These applications include the following.

- Array- or list-based symbol tables can be formed during lexical scanning with unit-time curser or pointer storage/access. These tables may then be used to support an efficient implementation of hygienic macro expansion [8].
- Many grammar transformations can be implemented efficiently using multiset string discrimination. Included in this paper is a new linear time left factoring transformation. Simpler forms of this ’heuristic’ transformation were previously studied by Stearns [32] and others (see also [20,3]) to turn non-LL context free grammars into LL grammars.
- An expression tree-to-dag transformation is implemented without hashing in a simpler way than before and in linear time and space. Several applications include one in which a linear pattern matching algorithm (e.g., [16]) can be turned into an efficient nonlinear matching algorithm, where each subtree equality check takes unit time. The method is also used to perform basic block optimizations (previously carried out by hashing ’value numbers’ [11,3]) without hashing. It also leads to a faster solution to the program equivalence problem used in integration by Yang, Horwitz, and Reps [40,41].
- Although the main parts of algorithms for global constant propagation [38], global common subexpression detection [5], and code motion [12] do not use hashing, the preprocessing

portions for each of these algorithms do. Such hashing can be eliminated without penalty by efficient construction and maintenance of the symbol table.

- Strength reduction [11,10,4,18,30,15] has remained one of the most complex, least understood, and most practical of the machine independent program optimizations. The strength reduction transformation presented by Cocke and Kennedy [10] may be the most practical reduction in strength algorithm published in the literature. Although the transformation due to Allen, Cocke, and Kennedy [4] is more powerful (since it can reduce multivariate products) and analyzes control flow more deeply, that algorithm can also degrade performance by introducing too many sums in order to remove nests of products (as was shown in [24]). Although Knoop and Steffen's approach [18] is more general, their algorithm is more difficult to implement, and its runtime performance may require exponential time.

  We solve three progressively more complex versions of the Cocke and Kennedy transformation without hashing and with superior worst case time and space than the expected performance in previous hash-based solutions[10]. In the final version an algorithm is presented to solve iterated strength reduction folded with useless code elimination in worst case asymptotic time and auxiliary space linear in the maximum text length of the initial and optimized programs. This solution has additional practical and theoretical significance, because it achieves an important goal in program optimization methodology to combine two technically different program optimizations without sacrificing performance.

2. Partial Tool Kit for Algorithm Design Without Hashing

There are many simple combinatorial problems for which hashing seems like the natural, perhaps only, way to obtain an efficient solution. These include such basic computations as:

- set union, difference, and intersection;
- multiset string discrimination; i.e., finding all duplicates in a multiset of strings;
- given a set S of pairs and an indexed collection of sets $\{T_1,...,T_k\}$, compute the set of pairs $\{[j,c]: \exists\, i \mid ([i,c] \in S \textbf{ and } j \in T_i)\}$.

Although hashing may seem like a panacea, it does incur costs, and one should not overlook the many contexts in which the preceding computations can be solved by an efficient hash-free approach.

In [25] Paige presented a different more general discussion of principles underlying hash-free algorithms for simple set operations. A few sharper techniques with a focus on multiset discrimination are discussed below.

2.1. Terminology

Throughout this paper it is convenient to make use of notations and terminology for specifying algorithms. If x is a variable, then expression $ref(x)$ is a pointer to x, and expression $deref(ref(x))$ is the value stored in x. If x is a record with field g, then the term $x.g$ retrieves the value of field g in record x. Different brackets are used to distinguish between sets, multisets, and sequences. A collection of (not necessarily distinct) values $c_1,...,c_n$ can be formed into a set (which only contains distinct values from the enumerated values $c_1,...,c_n$), a multiset, or a sequence by writing $\{c_1,...,c_n\}$, $<c_1,...,c_n>$, and $[c_1,...,c_n]$ respectively. The empty set, empty multiset, and empty sequence are denoted by $\{\}$, <>, and [] respectively. If R is a finite set, multiset, or sequence, then the number of elements in R is denoted by $\#R$. An arbitrary element chosen from R is denoted by *arbR*; if R has no elements, then *arbR* is undefined. A finite map is defined as a set of pairs, and can be either single- or multi-valued. Single-valued application of a map g to an argument a is denoted by $g(a)$, which is undefined if a doesn't belong to the domain of g or if g is multi-valued at a. Multi-valued map application is denoted by $g\{a\}$, which represents the set $\{b : [a,b] \in g\}$. If R is a finite nonempty set, then a *partition* P of R is any collection of pairwise disjoint nonempty subsets of R (called the *blocks* of P) whose union is all of R. If P and Q are two partitions over the same finite set R, then Q is a *refinement* of P if every block of Q is a subset of some block of P.

If S is a finite set, then S^* represents the set of finite sequences of values belonging to S. If s is a sequence, then $s(i)$ denotes the value occurring in the i-th component of s , where $s(1)$ is the value in the first or leftmost component. Two sequences are equal if they have the same length and are component-wise equal. The term $s(i..j)$ is used to denote the subsequence of s from the i-th to the j-th component. The term $s(..i)$ is an abbreviation for prefix $s(1..i)$, and $s(i..)$ is an abbreviation for suffix $s(i..\#s)$. If $i > j$, then $s(i..j)$ equals []. If s and t are two sequences, then $s\,t$ denotes the concatenation of s and t.

Strings are treated in the conventional way as a special case of sequences over a finite alphabet. They are written with contiguous components and without bracket delimiters; e.g. *abcd* and not $[a,b,c,d]$. Single characters are regarded as one-symbol strings when they are used as arguments in concatenation.

2.2. Basic Multiset Discrimination

Although multiset discrimination has been described in simple terms as finding duplicate values in a finite multiset, we can give the problem greater operational significance by adding one level of indirection. A new, more useful, characterization of multiset discrimination is the problem of function inversion; i.e., partitioning a finite set into preimages of a function that is defined on the set. More formally, the multiset discrimination problem inputs two sets V and S, and a lambda term $\lambda x.e : V \rightarrow S$ called a *discriminator*. It outputs the pair $[F,\ f]$, where

(1) $F = \{(\lambda x.e)y : y \in V\}$ is the subset of distinct discriminator values in S that underly the multiset $M = <(\lambda x.e)y : y \in V>$, and

(2) $f = \{[(\lambda x.e)y,\ y] : y \in V\}$ is a multi-valued map from F to preimage sets contained in V with respect to the discriminator.

This problem with its two main aspects - inverting a function and finding duplicate values in a multiset - is basic. It is central to the database index formation techniques within Willard's database predicate retrieval theory [39]. It is essential to Earley's program optimization by iterator inversion [14]. It was also used by Paige and Tarjan to design new algorithms for the minimization of one-symbol finite automata [26] and lexicographic sorting [22].

For both algorithms found in [26,22], a special instance of multiset discrimination was used in which S is a finite alphabet, and the discriminator is $\lambda x.h(x)(I)$, where $h : V \rightarrow S^*$ maps each element $x \in V$ to a string $h(x)$ over symbols in S, and $h(x)(I)$ is the *I-th* symbol of the string. We extend that solution in order to solve the more general multiset discrimination problem for sets S

with elements that can belong to a wide variety of datatypes, and for a more general class of discriminators.

Set S is implemented by a data structure D called a *directory*, which stores a distinct element of S in each accessible component of D. It is not necessary to distinguish between an endogenous implementation of S in which the elements of S are stored entirely within the directory, and an exogenous implementation in which the directory stores pointers to the elements of S [35]. If x is an element of the directory in an exogenous implementation, then implicit dereferencing is assumed in contexts where it is clear that the actual value is needed. However, it is useful to distinguish two related implementations of the directory. In one of these implementations the directory is an array that is accessed using cursors (i.e., integers 1,..,,#S) so that distinct cursors locate distinct elements of S. In the other implementation the directory is an arbitrary set of locations accessible by pointer. In either case, when two references to the directory are unequal (a unit-time operation), then the values they refer to must also be unequal.

The two implementations just described are combined below in a single abstract basic multiset discrimination procedure with three input parameters. Parameter *directory* is the built-in function name *deref* in pointer-based multiset discrimination. It is the name of the array storing S in the array-based implementation. Parameter *locs* is the set $\{ref(x): x \in V\}$ of all pointers to the elements of V (the discriminator domain) in both implementations. Parameter *discriminator* is a lambda term $\lambda x.e$ that maps pointers $p \in locs$ into references (either cursors or pointers) to elements of S. In the array-based implementation with array *S_array* storing the *directory*, then $discriminator(p)$ is a cursor such that $S\_array(discriminator(p))$ stores the corresponding element of S. In the pointer-based directory implementation $discriminator(p)$ is a pointer to the corresponding element $deref(discriminator(p))$ of S.

Output set F is a list of cursors (respectively pointers) to elements of the directory for the array-based (respectively pointer-based) implementation of S. Output map f has the same representation in both implementations. For each element x belonging to F, image set $f\{directory(x)\}$ equals $\{y : y \in locs$ **and** $discriminator(y) = x\}$, which is stored as a list of back pointers to elements of V.

Assume that $f\{x\}$ is initially empty for each x in S. Then procedures *md_array* and *md_pointer* below implement the two approaches just described. Data structures that abstract both approaches are shown in Figure 1.

```
--Pointer based implementation
proc md_pointer(locs,discriminator)
  return md_basic(locs,discriminator,deref)
end
--Array based implementation
proc md_array(locs,discriminator,directory)
  return md_basic(locs,discriminator,directory)
end
--Multiset Discrimination Driver
proc md_basic(locs,discriminator,directory)
F := {}                                  -- F will be the set underlying M
f := {}
for each pointer y in locs loop          -- linear search through locs
```

```
 if f{directory(discriminator(y))} = {} then     -- directory(discriminator(y)) is an element of S
   F := F ∪ {discriminator(y)}                   -- discriminator(y) locates an element of S
 end if
 f{directory(discriminator(y))} := f{directory(discriminator(y))} ∪ {y}
end loop
return [F, f]
end
```

Basic Multiset Discrimination Algorithm

In the implementation depicted in Figure 1, the domain of mapping f and the set S are shared. Each element z of S is implemented as a record with a field that contains a pointer (with initial value **nil** representing the empty set) to the set $f\{z\}$ of back pointers to V. It is easy to see that both methods just described can be implemented to run in time and space $O(\#locs)$ whenever execution of $discriminator(y)$ takes unit time for each $y \in locs$.

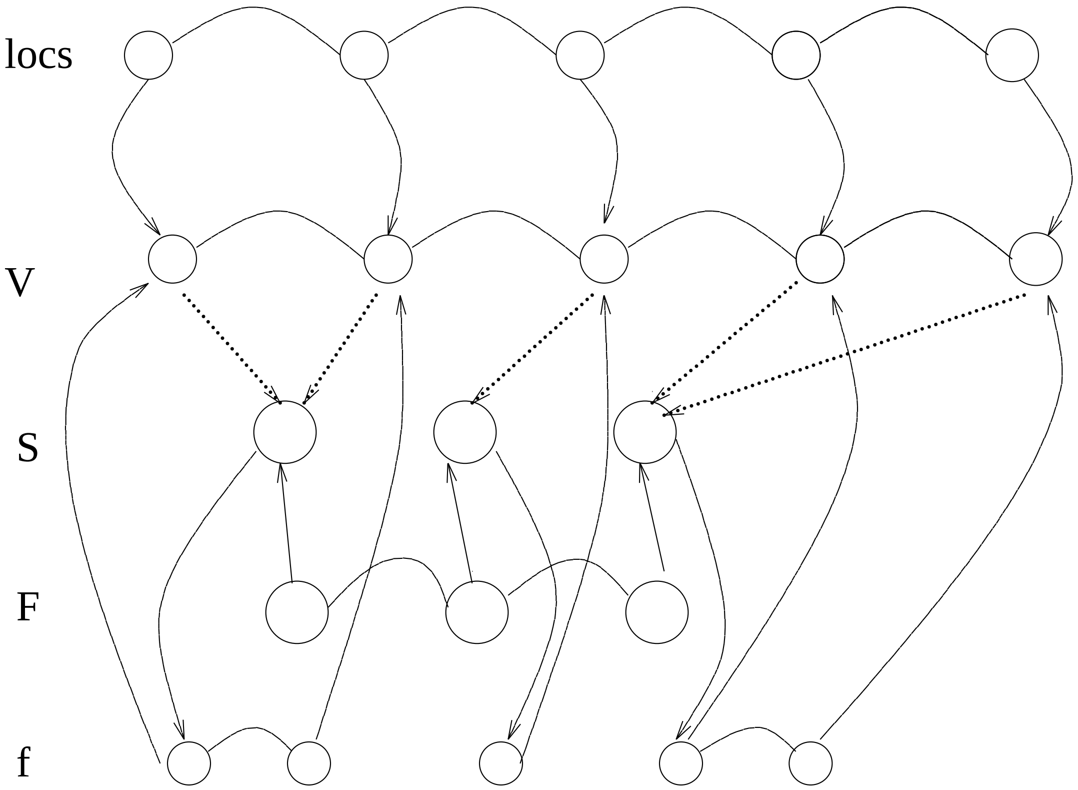

Figure 1. Data Structure Abstraction For Basic Multiset Discrimination

Of course, the efficiency of the two solutions to multiset discrimination rests on the key assumption that S is a SET, and that distinct values of $discriminator(y)$ locate distinct values in S. Since both methods manipulate unit-space cursors or pointers respectively, their efficiency is independent of the type of element of S. Ironically, even though Paige and Tarjan only considered the simplest form of multiset discrimination in which the elements of S are of type **char** [26,22], their array-based algorithm is easily turned into the polymorphic procedure *md_array* shown above.

The alternative implementation by procedure *md_pointer* is presented for pragmatic as well as theoretical reasons. Nonsequential access to an array component involves address arithmetic, which is more expensive than pointer access. A numeric representation of arbitrary values is somewhat more complex than a pointer representation. When numeric representations are used in array access to elements of S, then the space for S is proportional to the range of these numbers, which can be much greater than the number of elements in either of the sets F, V, or even S itself.

2.3. Multiset Discrimination of Sequences

Multiset discrimination of sequences is a natural extension of basic multiset discrimination. In abstract terms, the problem inputs two sets V and S, and *discriminator* $\lambda\, x.e : V \rightarrow S^*$. It outputs the pair $[F,\ f]$, where

(1) $F = \{(\lambda\, x.e)y : y \in V\}$ is the subset of distinct values in S^* that underly the multiset $M = <(\lambda\, x.e)y : y \in V>$, and

(2) $f = \{[(\lambda\, x.e)y,\ y\,] : y \in V\}$ is a multi-valued map from F to preimage sets contained in V with respect to the discriminator.

This problem is harder than basic multiset discrimination, because the discriminator range is not implemented as a directory. Only S is. Our solution essentially builds F as a directory for the discriminator range.

Paige and Tarjan solved multiset discrimination of sequences for the special case in which S is a finite alphabet implemented as an array-based directory, and the sequences belonging to M are implemented as arrays (or more precisely, as strings) [22]. In order to investigate the general utility of multiset discrimination of sequences, we extend their solution by making it polymorphic in S; i.e., by allowing the elements of S to belong to a general class of datatypes. We also make the approach more widely applicable by allowing the sequences belonging to M to be implemented by lists.

As in the case of basic multiset discrimination, both array-based (with array *S_array* storing the directory for S) and pointer-based directories are considered for implementing S. It is convenient to augment set S with a unique sentinel value denoted by **0**, which does not belong to any of the input sequences. We implement V as a set of *header* records, each providing pointer access to a distinct *body* that stores a sequence q of elements of S. The *body* is implemented as a one-way linked list of cells with two fields. The *data* field in the ith cell of a body that stores sequence q makes reference (by either pointer or cursor) to the element of S that corresponds to $q(i)$ for $i = 1, ..., \#q$. The $\#q + 1$st list cell is the last cell, and its *data* field stores a reference to the sentinel element of S. Each list cell also has a *next* field that stores a pointer to the next cell in the list, or **nil** in the case of the last cell. Each header record r contains two fields - a *first* field containing a pointer to the first element of the *body*, and a *current_position* field containing a pointer to an element (initially the first element) of the *body*. See Figure 2.

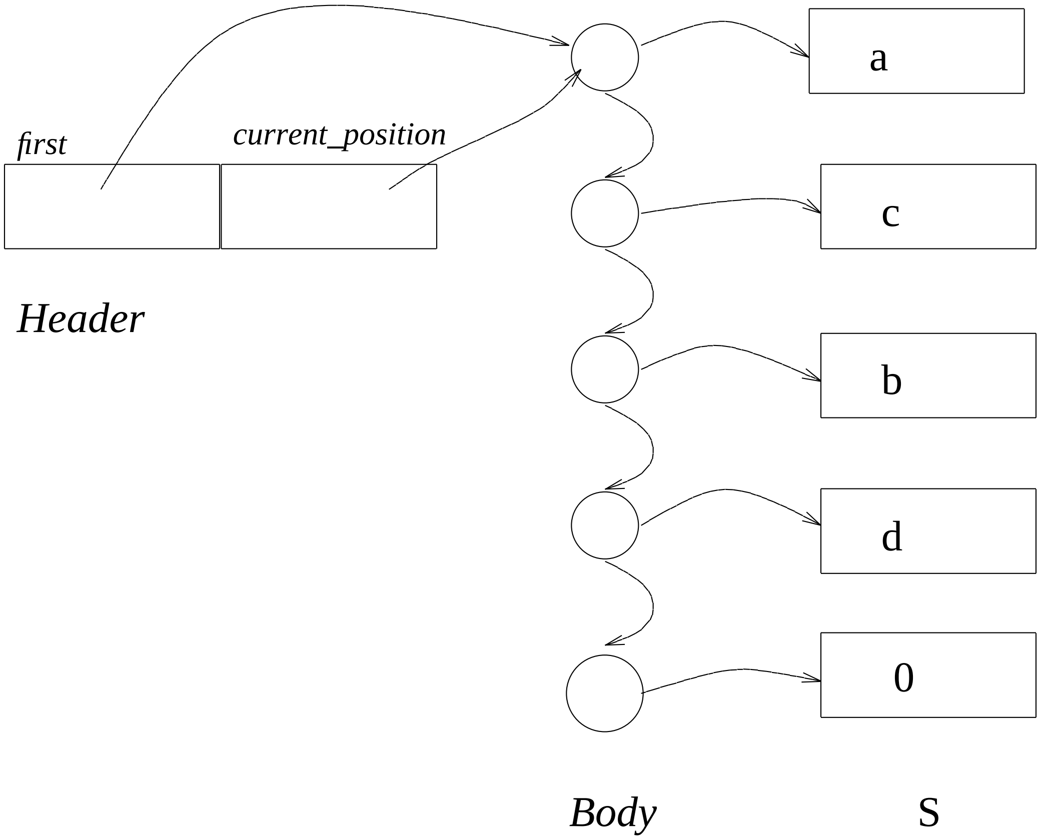

Figure 2. Implementation of the initial value of input sequence [a,c,b,d].

The abstract formulation of multiset sequence discrimination given below has four parameters. As in basic multiset discrimination, parameter *directory* is used to abstract the array- and pointer-based implementations of S. That is, *directory* equals the name of the array (storing S) in the array-based implementation, and *deref* otherwise. As before, parameter *locs* stores the set of pointers to the elements of V. Parameter *discriminator* is a lambda term $\lambda\, x.e$ that maps pointers $p \in locs$ to references $discriminator(p)$ to elements of S. Below we assume that lambda term $\lambda\, x.\ deref\ (deref\ (x).current\_position).data$ is the discriminator. Finally, we have a new parameter *update*, which is a lambda assignment $\lambda\, x.lhs(x){:=}rhs(x)$. It maps a pointer $p \in locs$, which references header record r, into an assignment $lhs(p){:=}rhs(p)$ that updates $r.current\_position$ so that it points to the next sequential list cell in the body for r. Below we assume that *update* equals $\lambda\, z.\ (deref\ (z).current\_position) := deref\ (deref\ (z).current\_position).next)$.

It is helpful to explain the abstract multiset sequence discrimination algorithm below using virtual fields (i.e., fields not actually implemented) associated with each header record $r \in V$. Let sequence q be stored in the body associated with header r. Among the virtual fields, let $r.pos$ equal the position in q corresponding to the list cell referenced by $r.current\_position$, let $r.prefix$ equal prefix $q(..r.pos-1)$ of q, and let $r.body$ equal q. Since for each record $r \in V$ the initial value of $r.current\_position$ is a pointer to the first element of its corresponding *body*, then the initial values of $r.pos$ and $r.prefix$ are, respectively, 1 and the empty sequence [].

A state of the multiset sequence discrimination algorithm is a partition P of *locs*. Each block of P will be designated *solved* if we know that it is a maximum subset of *locs* with pointers to records with equal bodies. Any block that isn't *solved* will be called *unsolved*. Initially, P contains the whole set *locs* as a single *unsolved* block. The algorithm repeatedly refines P until every block is *solved*, and each refinement step maintains the following three invariants:

(1) for each block B of P, every record r referenced by pointers belonging to B has the same *r.prefix*;

(2) header records referenced by different blocks of P have different *prefix*'s; and

(3) for each pointer $p \in$ *locs* to record r, *r.prefix* = *r.body* *iff* *directory*(*discriminator*(p)) = **0**.

Before describing the refinement step, we see immediately that the three invariants hold initially. The initial partition P has only one block *locs* containing pointers to records with *prefix* []. Each $p \in$ *locs* refers to a header r in which *r.body* is empty *iff* *directory*(*discriminator*(p)) = **0**.

Starting with this initial state, the algorithm proceeds as long as P contains an *unsolved* block by performing the following refinement step, which is easily shown to preserve the three invariants.

Refine. Remove an arbitrary *unsolved* block B from P, and perform basic multiset discrimination *md_basic*(B,*discriminator*,*directory*). The call to *md_basic* will compute the set FF of distinct references *directory*(*discriminator*(p)) (pointers or cursors) to elements of S (stored in the directory) for each pointer $p \in B$. For each pointer y belonging to FF, it also computes the set *ff*{*directory*(y)} of back pointers to all those header records r referenced by pointers p in B such that *discriminator*(p) equals y. The images of *ff*{*directory*(y)} for each $y \in FF$ partition B. These images are added back to P as new blocks that are designated *solved* or *unsolved* according to the following three cases that can arise.

Case 1. If *ff*{*directory*(y)} contains only a single back pointer to a record, say r, then no other record in V has the same body as *r.body* by invariant (2) and the semantics of *md_basic*. Hence, this block is *solved*.

Case 2. Otherwise, if *directory*(y) is the sentinel **0**, then *ff*{*directory*(y)} is the set of all pointers to records r referenced by pointers in B such that *r.current_position* points to the last list cell of the body for r, so that *r.prefix* equals *r.body* by invariant (3). By invariant (2) and the semantics of *md_basic*, we know that no other record in V has the same body as *r.body*. Hence, this block is *solved*.

Case 3. Otherwise, *ff*{*directory*(y)} is the set of pointers to all those records r referenced by pointers in B with bodies that have the same prefix formed from concatenating the value *directory*(y) $\in S$ to the right end of *r.prefix*. By invariant (2) and the semantics of *md_basic*, we know that no other record in V has such a prefix. However, longer prefixes for these records must be examined before their bodies can be distinguished. Hence, the *current_position* field of each of these records must be updated to point to the next list cell in its body; i.e., we perform *update*(p) for each back pointer $p \in$ *ff*{*directory*(y)}. Since this block must be processed further, it is *unsolved*.

The preceding arguments show that invariants (1 - 3) will be preserved by each refinement step. The update operation within *Case* (3) above ensures that the algorithm progresses, and that every pointer in *locs* will eventually end up in a block that is *solved* according to cases (1) or (2).

The abstract code for multiset sequence discrimination appears just below. In an efficient implementation P is represented by a list of blocks, and each block is represented by a list of pointers to records in V.

```
--Multiset Sequence Discrimination
proc md_seq (locs,discriminator,directory,update)
F := {}                          -- F will be the set underlying M
f := {}
P := {locs}
while ∃ B ∈ P loop
 P := P - {B}
 [FF, ff] := md_basic (B, discriminator, directory)
 for y ∈ FF loop
   if #ff{directory (y)} = 1 or directory (y) = 0 then
-- ff{directory (y)} contains a maximal set of headers in V whose bodies store the same sequence
-- witness points to an arbitrary body among these copies
     witness := deref (arbff{directory (y)}).first
     F := F ∪{witness}
     f{deref (witness)} := ff{directory (y)}
     for x ∈ ff{directory (y)} loop
--make all headers for the same sequence point to the witness body
1       deref (x).first := witness
2       deref (x).current_position := witness
     end loop
   else
     for z ∈ ff{directory (y)} loop
       update (z)          -- make deref (z).current_position point to next list cell
     end loop
     P := P ∪{ff{directory (y)}}
   end if
 end loop
end loop
return [F, f]
end
```

Multiset Sequence Discrimination Algorithm

The preceding algorithm is an abstracted and extended formulation of Paige and Tarjan's technique for multiset discrimination of varying length strings over alphabet S [22]. The earlier algorithm used an array-based directory for S and implemented strings as 'packed' arrays of cursors referencing the directory. The earlier algorithm had running time $O(m')$ and space $O(n+k)$, where n is the number of strings, m' is the total length of the prefixes needed to distinguish the strings, and $k=\#S$. Our list-based implementation has the same time and space bounds, but it has the advantage of easier memory management. Both this and the earlier array-based implementations are simple, and involve one pass through the prefixes of the sequences. Consequently, our proposed applications may be practical.

Previously, the lexicographic sorting algorithm found in Aho, Hopcroft, and Ullman's book [2] was used to solve congruence closure [13] and also tree isomorphism [2]. However, their sorting algorithm has the theoretical disadvantage of an $\Theta(m+k)$ complexity in time and auxiliary space, where m is the total length of all the input strings. Their algorithm also has the practical disadvantage of a complex multi-pass implementation, and a requirement to map graph structures into explicit ‘symbols’ in a totally ordered alphabet. Both these problems can be solved more simply by using the preceding list-based method to solve multiset discrimination of sequences.

2.4. Multiset Discrimination of Numeric Constants

Multiset discrimination of numeric constants can be solved by treating these constants as strings over a k-bit alphabet for arbitrary k, and using an array-based directory of size 2^k.

3. Applications

3.1. Symbol Tables

Multiset discrimination of strings can be used directly to implement a two-pass lexical scanner. First the program text in string form is scanned to produce tokens and initial pointers to symbol table entries. The symbol table is a multiset of lexemes (implemented as a doubly linked list). An additional pass is needed to remove redundant entries, and redirect pointers (the lexical values) to distinct entries in the modified table, now implemented as a pointer-based ’directory’. The performance is linear time and space in the length of the input string. Consequently, during the compiler phases for syntactic and semantic analysis the symbol table can be accessed by pointers instead of by hashing.

This approach supports the scope rule of block-structured languages conveniently if we implement each symbol table entry as a stack of pointers to records that store the identifier name (i.e. the symbolic address or l-value) and other attributes. Such a symbol table can also be used to implement macro expansion. For example, the hygienic macro expansion algorithm reported by Clinger and Rees [8] frequently performs the following operations: (1) replace all occurrences of a bound variable in its binding scope by a fresh identifier; (2) store a macro definition together with its environment into the symbol table (so that reference transparency can be achieved, i.e., when a macro is expanded, identifiers are referenced with respect to the environment where the macro is defined - not where it is used); and (3) paint an expansion, i.e., replace each identifier d introduced by expansion (i.e., identifiers occurring explicitly in the macro body and not within arguments to the macro) by a fresh identifier d' sharing the same symbol table entry with d (so that d' is interpreted as d in the environment in which the macro was defined). By making a new copy of the environment that was originally stored with the macro definition, the original environment is preserved.

Although Clinger and Rees [8] assume $O(1)$ time for each environment operation, a straightforward implementation based on their definition would require a linear search through lists that can be as long as the input text. We suggest the following more efficient implementation. Let $table(x)$ represent the symbol table entry for identifier x. Recall that, within the parsed program text, x is represented by a pointer to the symbol table entry $table(x)$, which is implemented as a stack of pointers to records (containing the identifier name, whether it is a program variable, keyword, macro name, etc., and other attributes).

In our approach, string substitution is very simple: to replace identifier x by a fresh identifier x' in its binding scope, we just push a pointer to a new record representing x' onto $table(x)$, and pop it from $table(x)$ when exiting from the binding scope. The cost is $O(1)$ time, independent of the number of occurrences of x to be replaced. To store a definition of macro m, we paint the symbols that will be introduced during expansion of m, and then push the painted macro definition onto $table(m)$; i.e., we replace each identifier d within the macro body that would be introduced during expansion by a pointer to a fresh symbol table entry for d', and push the top of $table(d)$ onto $table(d')$. When leaving the scope of the macro definition we pop $table(m)$. To paint an expansion, we replace each newly introduced identifier d by a pointer to a fresh symbol table entry for d', and push the top of $table(d)$ onto $table(d')$.

3.2. Fast Left Factoring

Left factoring is a context free grammar transformation investigated by Stearns [32] and others [20,3] as a tool for turning non-LL grammars into LL grammars. They did not describe optimal forms of factoring or algorithmic details. We define a new class of *factorable* grammars that can be turned into equivalent LL(1) grammars by applying an ‘optimal’ sequence of left factoring transformations. We show how to find and apply this optimal sequence to obtain an LL(1) grammar in linear time with respect to the number of symbol occurrences in the input grammar. The solution depends on basic multiset discrimination.

The following terminology and notation for context free grammars can be found in [3]. A context free grammar $G = [N, T, S \in N, P \subseteq N \times (N \cup T)^*]$ is a 4-tuple, where N is a nonempty finite set of nonterminal symbols, T is a nonempty finite set of terminal symbols disjoint from N, S is a distinguished nonterminal called the *start* symbol, and P is a finite nonempty set of productions of the form $M \rightarrow \alpha$, where M is a nonterminal and α is a finite string of grammar symbols (either terminals or nonterminals). Binary relation => is defined on strings of grammar symbols by the rule, $\alpha N \beta => \alpha\delta\beta$ *iff* there exists a production $N \rightarrow \delta \in P$. The reflexive transitive closure of =>, denoted by =>*, is defined by the rule, $\alpha =>^* \beta$ *iff* $\alpha = \beta$ or there exists a sequence of strings $\alpha_i \in (N \cup T)^*$ for $i=1,...,n$ and $n > 1$ such that $\alpha_1 = \alpha$, $\alpha_n = \beta$ and $\alpha_i => \alpha_{i+1}$ for $i=1,...,n-1$. The language generated by G is $\{\alpha \in T^* \mid S =>^* \alpha\}$, also called the set of sentences of G. The set of sentential forms of G is the set $\{\alpha \in (T \cup N)^* \mid S =>^* \alpha\}$. More generally, the set of sentences (respectively sentential forms) of G derived from a string $\beta \in (T \cup N)^*$ of grammar symbols is defined to be $\{\alpha \in T^* \mid \beta =>^* \alpha\}$ (respectively $\{\alpha \in (T \cup N)^* \mid \beta =>^* \alpha\}$). Two context free grammars are *equivalent* if they generate the same language.

We use upper case italic letters to denote nonterminals, lower case italic letters to denote terminals, and w, x, y, z and Greek letters to denote strings of terminal and nonterminal symbols. It is convenient to use alternation notation $A \rightarrow \alpha \mid \beta$ to abbreviate the two context free grammar productions $A \rightarrow \alpha$ and $A \rightarrow \beta$. Throughout this section it will sometimes be convenient to identify a grammar G with its set of productions. We also assume that each context free grammar G contains only nonterminals that can derive nonempty sentences and can occur within some sentential form of G.

Definition 3.1. (see [3]): Two productions $A \rightarrow \alpha \mid \beta$ belonging to G form an LL(1) *conflict iff* one of the following conditions holds

(1) there exists a terminal symbol t such that α =>* $t\ x$ and β =>* $t\ y$;

(2) α =>* [] and β =>* [];

(3) α =>* [], and there exists a terminal symbol t such that β =>* $t\ x$, and start symbol S =>* $w\ A\ t\ y$.

Definition 3.2. If W is a set of strings over alphabet Σ, then define

$$split(W,i) = \{\{x \in W \mid x(i)=a\}: a \in \Sigma \textbf{ and } \exists x \in W \mid x(i)=a\}$$

The longest common prefix of a nonempty set of strings W, denoted by $lcp(W)$, is defined recursively according to the following rule:

```
lcp(W) = if #W = 1 then -- If W is the singleton set, say {x},
            arbW              -- then select x from W
         elseif #split(W, 1) > 1 then
            []
         else                 -- s is the first symbol of every string in W
            s lcp({x(2..):x∈W}) where s = (arbW)(1)
         end if
```

Grammar G is said to be LL(1) if it has no LL(1) conflicts. We divide LL(1) conflicts into two kinds:

(1) A *factorable* LL(1) *conflict* is an LL(1) conflict $A \rightarrow \alpha\beta \mid \alpha\tau$ such that $\alpha = lcp(\{\alpha\beta,\alpha\tau\})$ is nonempty, and after replacing this conflict by productions $A \rightarrow \alpha C$, $C \rightarrow \beta$, and $C \rightarrow \tau$, where C is a new nonterminal symbol, then $C \rightarrow \beta \mid \tau$ is not an LL(1) conflict.

(2) A *nonfactorable* LL(1) *conflict* is an LL(1) conflict that is not factorable.

G is said to be a *factorable* grammar if all of its conflicts are factorable.

Definition 3.3. (*Left Factoring Transformation*): Let R be a set of productions $A \rightarrow \alpha\ \beta_i$, i=1..n in grammar G, where $n > 1$ and $\alpha \neq []$. If C is a new nonterminal not in G, then $G' = LF(G,R,\alpha,C)$ is the grammar formed from G by adding nonterminal C, and by replacing productions R within G by the productions $A \rightarrow \alpha\ C$ and $C \rightarrow \beta_i$, i=1,..,n.

Lemma 3.1. *Consider grammar $G' = LF(G,R,\tau,C)$. If A is a nonterminal of G, then the set of sentential forms derived from A in G is contained in the set of sentential forms derived from A in G', and the set of sentences derived from A in G equals the set of sentences derived from A in G'. In particular G' and G are equivalent.*

Proof. This follows immediately from *Definition* 3.3. □

Lemma 3.2. *Nonfactorable conflicts cannot be eliminated by left factoring.*

Proof. Let $A \rightarrow \alpha \mid \beta$ be a nonfactorable conflict in grammar G, and consider grammar $G' = LF(G,R,\tau,C)$ that results from any left factoring transformation. If R does not contain either $A \rightarrow \alpha$ or $A \rightarrow \beta$, then these two productions remain in G', where they must also form a nonfactorable conflict. If R contains both of these productions, then by *Definition* 3.3 there exists a nonempty string τ and strings α_1 and β_1 such that $\alpha = \tau\ \alpha_1$, $\beta = \tau\ \beta_1$, and G' contains productions $C \rightarrow \alpha_1 \mid \beta_1$. Productions $C \rightarrow \alpha_1 \mid \beta_1$ must form a nonfactorable LL(1) conflict of G'. Otherwise, we would reach a contradiction that $A \rightarrow \alpha \mid \beta$ is a factorable LL(1) conflict. Finally, if $A \rightarrow \alpha$ belongs to R but $A \rightarrow \beta$ does not, then G' contains the productions $A \rightarrow \tau\ C \mid \beta$. By

Lemma 3.1 any sentential form derived from α and from the start symbol in grammar G can be derived respectively from τC and the start symbol in grammar G'. Hence by *Definition* 3.1, if $A \rightarrow \alpha \mid \beta$ forms an LL(1) conflict in G, then productions $A \rightarrow \tau\, C \mid \beta$ must form an LL(1) conflict in G'. Suppose that $A \rightarrow \tau\, C \mid \beta$ were factorable. Then by definition there exist strings ρ, α_1, and β_1 such that $\tau C = \rho\alpha_1 C$ and $\beta = \rho\beta_1$, where $\rho = lcp(\{\tau C, \beta\})$ is nonempty. Also, after replacing productions $A \rightarrow \tau C \mid \beta$ by productions $A \rightarrow \rho C_1$ and $C_1 \rightarrow \alpha_1 C \mid \beta_1$, where C_1 is a new nonterminal symbol, we know that the two C_1 productions do not form an LL(1) conflict. But this leads to a contradiction that productions $A \rightarrow \alpha \mid \beta$ form a factorable conflict in G. □

A left factoring transformation $LF(G,R,\tau,C)$ is *safe* if for each production $A \rightarrow \alpha$ in R, every production $A \rightarrow \beta$ in G must also belong to R whenever $lcp(\{\alpha,\beta\})$ is not a prefix of τ.

Lemma 3.3. *If G is factorable, then LF(G,R,τ,C) is factorable iff LF(G,R,τ,C) is safe.*

Proof. Suppose $G' = LF(G,R,\tau,C)$ is not safe. Then there is a production $A \rightarrow \gamma\,\alpha$ in R and a production $A \rightarrow \gamma\,\beta$ in G but not in R in which $lcp(\{\gamma\alpha,\gamma\beta\}) = \gamma$ is not a prefix of τ. Hence, τ must be a proper prefix of γ; i.e, $\gamma = \tau\,\rho$ for some nonempty string ρ. Moreover, G' must contain productions $A \rightarrow \tau\rho\beta \mid \tau C$ and $C \rightarrow \rho\alpha$. Since we assume that every nonterminal in G must derive a nonempty string of terminals, then ρ must derive a sentential form that starts with some terminal symbol t. Hence, both C and the string $\rho\beta$ must derive sentential forms that start with t. Since C does not occur in string $\rho\beta$, then $A \rightarrow \tau\rho\beta \mid \tau C$ must form a nonfactorable LL(1) conflict in G'.

Next, suppose that G is factorable, and $G' = LF(G,R,\tau,C)$ is safe. We show that any LL(1) conflict in G' must be factorable. Any LL(1) conflict in G' must be one of the following three kinds.

(1) ($A \rightarrow \alpha \mid \beta$, with no occurrence of C) This is factorable, since it must be a factorable LL(1) conflict in G.

(2) ($C \rightarrow \alpha \mid \beta$) There must be a corresponding factorable LL(1) conflict $A \rightarrow \tau\,\alpha \mid \tau\,\beta$ of G contained in R. Hence, $C \rightarrow \alpha \mid \beta$ is a factorable LL(1) conflict of G'.

(3) ($A \rightarrow \tau\, C \mid \beta$, with only one occurrence of C) G must contain a factorable LL(1) conflict formed from some production $A \rightarrow \tau\,\alpha$, which belongs to R, and production $A \rightarrow \beta$, which does not. Since $LF(G,R,\tau,C)$ is safe, then $\rho = lcp(\{\tau\alpha,\beta\})$ must be a prefix of τ. Hence, $A \rightarrow \tau\, C \mid \beta$ is factorable in G'. □

Theorem 3.4. *If G is factorable, then a finite number of successive applications of safe left factoring transformations will yield an LL(1) grammar.*

Proof. Let $G' = LF(G,R,\tau,C)$ be a safe left factoring transformation, where each production in R has nonterminal A on the left-hand-side. Let the *weight* of G, denoted by $w(G)$, be the sum of the lengths of the longest common prefixes of the right-hand-sides of each LL(1) conflict. An LL(1) conflict of G can either be (1) two productions not belonging to R, (2) two productions both belonging to R, or (3) one production belonging to R and the other not. The first kind of conflict also occurs in G', according to case (1) in the proof of Lemma 3.3. The second kind is either eliminated or shows up as a distinct case (2) conflict in the proof of Lemma 3.3. Since the number of different conflicts within R is $\#R(\#R-1)/2$, then the total weight reduction that results from replacing R by productions that have C on their left-hand-sides is $\#\tau\#R(\#R-1)/2$. Finally, the third kind of conflict in G contains one production $A \rightarrow \tau\alpha$ that belongs to R and the other production $A \rightarrow \beta$ that does not. Moreover, by analysis of case (3) in the proof of Lemma 3.3, $A \rightarrow \tau\alpha$ must form an LL(1) conflict with every production in R. However, all of these conflicts associated with $A \rightarrow \beta$ are replaced by a single pair of productions $A \rightarrow \tau C \mid \beta$ in G'. Hence, left factoring reduces the weight for this kind of conflict by $(\#R-1) \sum_{A \rightarrow \gamma \in G-R} \#lcp(\{\tau,\gamma\})$. The weight of G' is, therefore, $w(G) - (\#\tau \#R(\#R-1)/2 + (\#R-1) \sum_{A \to \gamma \in G-R} \#lcp(\{\tau,\gamma\}))$. Since weight is monotonically decreasing with respect to safe left factoring, a finite number of applications of safe left factoring will yield an equivalent LL(1) grammar with weight 0. □

Theorem 3.4 give rise to a variety of strategies for turning factorable grammars into LL(1) grammars. We say that a strategy is *optimal* if it applies the smallest number of left factoring transformations. An optimal strategy will introduce the smallest number of new nonterminal symbols. As we shall see, it will also produce a grammar whose productions contain the fewest occurrences of grammar symbols.

In order to investigate optimal strategies we need to introduce some additional terminology and notation. A pair $[\alpha, Q]$ is a *gap* for nonterminal A in grammar G if $Q = \{A \to \beta \in G \mid \alpha$ is a prefix of $\beta\}$, $lcp(\{\beta : A \to \beta \in Q\}) = \alpha$, $\alpha \neq []$, and $\#Q > 1$. A *gap transformation* is a left factoring transformation $LF(G, Q, \alpha, C)$ in which $[\alpha, Q]$ is a gap for G.

The following obvious properties of gaps and gap transformations are stated without proof.

Lemma 3.5. *A gap transformation is safe. If $[\alpha, Q]$ and $[\beta, R]$ are two gaps for nonterminal A and grammar G, then α is a prefix of β iff $R \subseteq Q$. Neither α nor β is a prefix of the other iff R and Q are disjoint.*

By Lemma 3.5, gaps can be partially ordered. That is, gap $[\alpha, Q]$ is less than or equal to gap $[\beta, R]$ *iff* $Q \subseteq R$. Gap $[\alpha, Q]$ is *maximal* if there is no other gap $[\beta, R]$ in which $Q \subseteq R$. If $[\alpha, Q]$ and $[\beta, R]$ are two different maximal gaps, then Q and R must be disjoint. A production $A \to \alpha$ is said to be *factored* if no other production $A \to \beta$ has $\beta(1) = \alpha(1)$. The set of productions in a grammar can be partitioned into factored productions and sets R of productions in maximal gaps $[\alpha, R]$.

Theorem 3.6. *If G is factorable, then the minimum number of applications of safe left factoring transformations to obtain an LL(1) grammar equals the total number of gaps in G.*

Proof. Let $G' = LF(G, R, \tau, C)$ be a safe left factoring transformation, where each production in R has left-hand-side nonterminal A. First, we show that the number of gaps in G' is either the same as G or one less than G. Next, we show that the number of gaps in G is less than the number of gaps in G' *iff* $[\tau, R]$ is a gap.

Gaps can be divided into three corresponding cases for grammars G and G'. A gap $[\alpha, Q]$ within G either has Q disjoint from R, Q contained in R, or Q has a production in R and one not in R. These cases correspond to gaps $[\alpha, Q]$ within G' in which Q contains no productions with an occurrence of nonterminal C, Q contains only productions with C as the left-hand-side nonterminal, and Q contains occurrences of C on the right-hand-side.

(1) If $Q \subseteq G \cap G'$, then we claim that $[\beta, Q]$ is a gap in G' *iff* it is a gap in G. In this case Q and R are disjoint.

The 'only if' part is obvious. To prove the 'if' part, suppose that $[\beta, Q]$ is a gap in G' but not in G. Since $Q \subseteq G$, it follows from Lemma 3.5, from the partial ordering on gaps, and from $\beta = lcp(\{x : \exists B \mid B \to x \in Q\})$ being nonempty, that G contains a unique maximum gap $[\beta, W]$ such that $Q \subseteq W$. Suppose that production $A \to \beta\alpha$ belongs to W and G but not Q and G'. Then $A \to \beta\alpha$ belongs to R, there exists a nonempty string ρ such that $\beta = \tau\rho$, and productions $A \to \tau C$ and $C \to \rho\alpha$ belong to G'. Since $LF(G, R, \tau, C)$ is safe and $lcp(\{x : A \to x \in Q\} \cup \{\beta\alpha\})$ is not a prefix of τ, then $Q \subseteq R$, a contradiction.

(2) If $\beta \neq []$, then $[\beta, \{C \rightarrow \beta\ \alpha_i,\ i{=}1,..,k\}]$ is a gap of G' *iff* $[\tau\ \beta, \{A \rightarrow \tau\ \beta\ \alpha_i,\ i{=}1,...,k\}]$ is a gap of G. In this case $\{A \rightarrow \tau\ \beta\ \alpha_i,\ i{=}1,...,k\}$ is a subset of R. When $\beta{=}[]$, then gap $[\tau\ \beta, \{A \rightarrow \tau\ \beta\ \alpha_i,\ i{=}1,...,k\}]$ of G does not correspond to any gap in G'.

(3) Finally, $[\beta, (Q - R) \cup \{A \rightarrow \alpha\ C\}]$ is a gap of G' *iff* $[\beta, Q]$ is a gap of G, $Q - R \neq \{\}$, and Q and R are not disjoint. In this case, R must be a strict subset of Q or else left factoring is not safe.

By case (2) G' has exactly one fewer gap than G *iff* $[\tau, R]$ is a gap for G. The result follows. □

By Theorem 3.6, an optimal left factoring strategy must iterate gap transformations $G := LF(G, R, \tau, C)$ until grammar G is free of gaps.

Theorem 3.7. *The number of grammar symbols occurring in the grammar produced by an optimal left factoring strategy is the same independent of the order in which gap transformations are chosen.*

Proof. Recall from Lemma 3.5 that incomparable gaps have disjoint sets of productions. Thus, it suffices to consider two gaps $[\alpha_1, R_1]$ and $[\alpha_1\alpha_2, R_2]$ for some nonterminal A in grammar G. By Lemma 3.5 we know that $R_2 \subseteq R_1$. Grammar $G_1 = LF(G, R_1, \alpha_1, C)$ replaces all productions $A \rightarrow \alpha_1\beta_i$ for $i=1,...,k$ in R_1 by productions $A \rightarrow \alpha_1 C$ and $C \rightarrow \beta_i$ for $i=1,...,k$. The productions of grammar G_1 have $\#\alpha_1(\#R_1-1)-1$ fewer occurrences of grammar symbols than the productions of grammar G. Gap $[\alpha_1\alpha_2, R_2]$ in G is transformed into gap $[\alpha_2, R_3]$ appearing in grammar G_1, where $R_3 = \{C \rightarrow \beta_i : i=1,..,k$ **and** α_2 is a prefix of $\beta_i\}$. Next, we see that the productions of grammar $G_2 = LF(G_1, R_3, \alpha_2, D)$ have precisely $\#\alpha_2(\#R_3-1)-1$ fewer occurrences of grammar symbols than the productions in grammar G_1. Since $\#R_3 = \#R_2$, then the total reduction in grammar symbols resulting from transforming G into G_2 is $\#\alpha_1(\#R_1-1) + \#\alpha_2(\#R_2-1)-2$. The reader can verify that factoring G with respect to the alternative order of gap transformations yields the same reduction in grammar symbols. □

Our algorithm will repeatedly perform gap transformations using maximal gaps until the grammar is free from gaps, and only *factored* productions remain. The key strategic idea is to maintain a partition P of productions, and a map *prefix* from blocks of P to strings of grammar symbols such that the following invariant holds.

Invariant. Every maximal gap $[\alpha, R]$ corresponds to a block $B \in P$ such that $R \subseteq B$, *prefix*(B) is a common prefix of all the right-hand-side strings of productions included in B, and *prefix*(B) is a prefix of α.

The initial partition P stores all productions $A \rightarrow \alpha$ for nonterminal A in a separate block for each nonterminal A in the grammar. It is convenient to let the right-hand-side of every production end in a special sentinel symbol (outside the set of grammar symbols). For each such initial block B we have *prefix*$(B) = []$. Clearly, the invariant holds initially.

The algorithm proceeds by repeatedly removing a block B from P and performing the following refinement step:

Refine. Assume that B contains only productions with left-hand-side nonterminal A. Assume also the induction hypothesis that the invariant holds just before the refinement step is performed. There are two cases to consider.

Case 1. If B contains only one production $A \rightarrow \alpha$, then, by the induction hypothesis, it must be factored. Hence, it is included in the final grammar. For this simple case, the invariant is clearly preserved.

Case 2. Otherwise, split the right-hand-side strings of productions in B by the symbol occurring just after $prefix(B)$ in each such string (cf *Definition* 3.2 of *split*); i.e., perform

$D := split(\{\alpha\colon A \rightarrow \alpha \in B\}, \#prefix(B) + 1)$

There are three subcases to consider.

2.1 If D contains only one set, so that one symbol, say s, occurs just after $prefix(B)$ in the right-hand-side of every production in B, then redefine $prefix(B)$ to be $prefix(B)\ s$, and add B back to P. Since B must be a set with more than one string, s cannot be the sentinel. It follows from the definition of gaps and the induction hypothesis that the invariant is maintained.

2.2 If $\#D > 1$ and $prefix(B) = []$, then B must be an initial block. For every set $DD \in D$, add the new block $BB = \{A \rightarrow \alpha\colon \alpha \in DD\}$ to P, and define $prefix(BB) = s$, where s is the first symbol occurring immediately after $prefix(B)$ in every string belonging to DD. If s is the sentinel, then DD must contain only one string, so that block BB will be handled later by *Case* 1. By the definition of gaps, and by the particular fact that string α must be nonempty for any gap $[\alpha, R]$, the invariant is maintained

2.3 If $\#D > 1$ and $prefix(B) \neq []$, then $[prefix(B), B]$ must be a maximal gap by the induction hypothesis and by the definition of gaps. Perform part of the corresponding gap transformation by adding the single factored production $A \rightarrow prefix(B)\ C$ to the final grammar, where C is a new nonterminal. Next, for every set $DD \in D$, add the new block $BB = \{C \rightarrow \alpha(\#prefix(B){+}1..)\colon \alpha \in DD\}$ to P, and define $prefix(BB) = s$, where s is the symbol occurring just after $prefix(B)$ in every string belonging to DD. If s is the sentinel, then DD must contain only one string, so that block BB will be handled later by *Case* 1. The invariant is maintained by the proof of Theorem 3.6 case (2) and the proof of Theorem 3.7. The algorithm proceeds to search for gaps and factored productions among the newly introduced C productions.

To design an efficient implementation, we note, first of all, that the right-hand-sides of productions re-introduced to P in Step (2) are all suffixes of right-hand-sides in the original grammar input. In particular, the new grammar symbols C introduced in Step (2.3) never appear on the right-hand-side of any production introduced into partition P. We also note that the *prefix*'s are all substrings of right-hand-sides in the original input grammar. Hence, when performing gap transformations , we can reuse the right-hand-side strings of the original grammar. That is, we can represent a block B of productions (with left-hand-side nonterminal A) stored in P together with $prefix(B)$ using a record $[B', [i, j], A]$, where B' is a subset of right-hand-side's of productions in the initial grammar input, $B = \{A \rightarrow \beta(i..)\colon \beta \in B'\}$, and $prefix(B) = \beta(i..j-1)$ for any string $\beta \in B'$; i.e., the ith through $j-1$st symbol of every string in B' is the same.

With the preceding representation partition P can be implemented as a one-way list of blocks, each implemented as a triple. For grammar G the initial partition P contains triple $[\{\alpha\colon A \rightarrow \alpha \in G\}, [1,1], A]$ for each nonterminal A in the grammar. After initialization the algorithm computes the new grammar G' as described below:

$G' := \{\}$
while $P \neq \{\}$ **loop**

```
  remove block [B, [i,j], A] from P
  if B contains only one string α then
    G' := G' ∪ {A → α(i..)}  -- found a factored production
  else
-- find a left factor for strings in block B starting from the ith position
    BB := split(B,j)
    if BB contains only one block then
--  the ith through jth symbols of every string in B are the same, and form part of a nonempty left factor
1     add [B, [i,j+1], A] to P
    else -- BB contains more than 1 block
-- the ith through j−1st symbols of every string in B are the same, and form a complete left factor τ
2     if [i,j] = [1,1] then -- left factor τ = [], and B must be an initial block
        C := A   -- C represents nonterminal A
      else -- left factor τ ≠ []
        create new nonterminal C
        G' := G' ∪ {A → τ C}
      end if
-- efficient joint implementation of cases (2.2) and (2.3)
      for each block D in BB loop
        if D contains only one string β  then
          G' := G' ∪ {C → β(j..)}
        else
3         add [D,[j,j+1],C] to P -- try to find the left factor for strings in D
        end if
      end loop
    end if
  end if
end loop
```

Optimal Left Factoring Algorithm

We can implement the preceding algorithm with only minor modification to the pointer-based multiset discrimination method for strings described in the last section. A directory can be used to store the set $N \cup T \cup \{\mathbf{0}\}$ of grammar symbols and sentinel. Then each right-hand-side α of a production in G can be implemented by a *header* record with a *first* and *current_position* field and a body storing α in a list of pointers to $N \cup T \cup \{\mathbf{0}\}$. Using multiset discrimination of sequences, we can form the set W of distinct right-hand-sides of productions in grammar G. Then for each triple $[D,[i,j],A]$, let D be implemented as a set of pointers to header records associated with strings in W. We modify the implementation of the last section only slightly in the following way. In each such header record r let *r.first* and *r.current_position* point respectively to the ith and jth list cell of *r.body*. Initially, the *first* and *current_position* pointers point to the first list cell in each header record. At program points (1) and (3) of the optimal left factoring algorithm, the *current_position* pointer for each header record in block B (respectively D) is updated to the next sequential list cell as in routine *md_seq* of the last section. At program point (3) the *first* pointer of each header record in block D is replaced by the *current_position* pointer. At program point (2) the test for the initial block can be performed in unit time by checking whether the *current_position* and *first* pointers are the same in an artibtrary header record of block B.

We implement *split* efficiently by basic multiset discrimination. If each record referenced by elements of W has a *current_position* field pointing to the jth list cell, then $split(B,j)$ can be implemented by the call $md\_pointer(W, \lambda x.deref(deref(x).current\_position).data)$. Recall that such a call returns the pair $[F,f]$, where f maps symbols of $N \cup T \cup \{\mathbf{0}\}$ to back pointers to header records of strings in B that have the same jth symbol.

The preceding discussion leads to the following theorem.

Theorem 3.8. *The optimal left factoring algorithm presented above is correct and runs in linear time in the number of the grammar symbol occurrences in the productions of the input grammar.*

Proof. Every basic operation in the preceding algorithm takes unit time except for $split(B,j)$, which takes time $O(\#B)$. Because j is incremented just before blocks are added to P, we see that new symbol occurrences are processed in the input strings in each call to *split*. The result follows. □

3.3. Multiset Discrimination of Trees and Applications

Suppose we have a forest of syntax trees produced by syntactic analysis. Each internal node in the syntax tree is associated with a *node list* that begins with an identifier (a function symbol of arity greater than 0) followed by the ordered successors of the node. Each successor is either an identifier (constant or variable) or another internal node.

There are many applications that depend on identifying subtrees that are *equivalent* in the following sense. We say that two identifiers are equivalent if they are equal; two nodes are equivalent if the components of their node lists are pairwise equivalent, and two subtrees are equivalent if their respective root nodes are equivalent. The problem of deciding subtree equivalence arises in common subexpression detection [9], turning an arbitrary linear tree pattern matching algorithm [16] into a nonlinear matching algorithm [28], deciding structural equivalence of type expressions[3], and preprocessing input in the form required by Wegman and Paterson's unification algorithm [27].

Cocke and Schwartz [11] solved this problem by representing distinct node identifiers and nodes with distinct integers called 'value numbers'. Their method involves extensive hashing. An alternative method related to the tree isomorphism solution found in [2] uses lexicographic sorting to compute value numbers. Although it runs in linear worst case time in the size of the forest, it is a complex multi-pass method that is not likely to outperform the value number method in practice.

This problem can be solved by multiset sequence discrimination more efficiently without hashing and without any numeric representation for identifiers or nodes. Suppose that the symbol table is implemented as a pointer-based directory of identifiers. Suppose also that each identifier component of a node list is represented as a pointer to its symbol table entry, and each component that represents a successor node n is represented (in the same way as sequences in the last section) by a header record that references the node list for *n*.

Our solution rests on two simple ideas. First, since the symbol table forms a directory, equivalence is solved at the outset for components of node lists that represent identifiers. Define the *height* of an identifier to be 0, and the *height* of an internal node *n* to be one plus the maximum height of the components of the node list for *n*. The second idea is that nodes at distinct heights in the forest cannot be equivalent. This allows us to solve multiset subtree discrimination separately for all nodes of the same height bottom-up starting from the nodes of height one.

Suppose that the forest has maximum height *d*. Then the following procedure will make use of multiset sequence discrimination to solve multiset subtree discrimination.

1. For each tree root *n* in the forest, create a dummy header record that references the node list for *n*.
2. For height $i = 1, 2 \ldots, d$, repeat the following step.
   2.1 Let *locs* be the set of pointers to all header records (for the forest roots or contained within node lists) that reference nodes of height *i*. Solve multiset discrimination of sequences by calling procedure *md_seq* with arguments *locs*, etc., but modified in the following minor way. Instead of updating each header record referenced by *locs* to point to a witness at program points (1) and (2), completely replace each such header record by a pointer to the witness; i.e., perform assignment *deref* (x) := *witness*.

The preceding algorithm solves the subtree equivalence problem, and compresses a syntax forest into a dag with no redundant subtrees (i.e., no two occurrences of equivalent subtrees) in worst case time and space linear in the number of nodes in the forest. It is a great deal simpler than the previous theoretically best algorithm based on lexicographic sorting. We expect it to outperform the method of value numbering in practice.

3.4. Multiset Dag Discrimination and Acyclic Coarsest Partitioning

The solution to multiset tree discrimination extends without modification to solve multiset discrimination for ordered dags with *m* edges and *n* nodes in worst case time $O(m)$ and auxiliary space $O(n)$. This space bound improves the $O(m)$ space bound that could be obtained to solve this problem using Aho, Hopcroft, and Ullman's lexicographic sorting algorithm [2]. We also show how multiset dag discrimination can be used to obtain an improved solution to acyclic instances of the many-function coarsest partition problem.

The many-function coarsest partition problem, used by Hopcroft to model the problem of DFA minimization [17], has applications in program optimization and program integration. It can be formulated as follows. Given a directed multi-graph $(V, E_1, \ldots, E_k)$ (where V is the set of vertices, and $E_1, \ldots, E_k$ are sets of edges), and an initial partition $P = \{V_1, \ldots, V_s\}$ of V, find a coarsest refinement P' of P such that for each block C in P' and each $i = 1, \ldots, k$, there exists a block C_0 in P' such that the image set $E_i[C] \subseteq C_0$, where $E_i[C] = \{y : \exists\, x \mid ([x, y] \in E_i \textbf{ and } x \in C)\}$. Here we assume that for each $i = 1, \ldots, k$, the out-degree of each vertex $v \in V$ in graph (V, E_i) is at most 1.

Hopcroft gave an algorithm that solves this problem in time $\Theta(kn \log n)$ and space $\Theta(k\ n)$ in the worst case[17], where $n = \#V$. As described in [2] Hopcroft's algorithm is nondeterministic, and could behave in the following way for the case where $k=1$ and $f = E_1$:

1. Push each block of the inital partition P onto a stack W.
2. While W is nonempty, pop block B from W, and use B to split blocks of P in the following way.
   2.1 Split each block Q in partition P that is neither disjoint from nor a subset of $f^{-1}[B]$ into two new blocks $Q' = Q \cap f^{-1}[B]$ and $Q'' = Q - f^{-1}[B]$. If Q belongs to W, then push both Q' and Q'' onto W. Otherwise, push either Q' or Q'', whichever is not larger than the other, onto W.

Consider an input instance for a single function $f: V \to V$ defined on $V = \{1,..., 2^r\}$ with $r>0$ according to the rule $f(i) = i+1$ for $i=1..2^r-1$, and $f(2^r) = 2^r$. Let initial partition $P = \{B_1,...,B_{r+1}\}$ such that $B_i = \{2^{i-1}(1+2j): j=0,...,2^{r-i}-1\}$ for $i=1,...,r$, and $B_{r+1} = \{2^r\}$.

Lemma 3.9. *The preceding problem instances satisfy the following properties.*

(A) $\#B_i = 2^{r-i}$ for $i=1,...,r$.

(B) $B_i = f^{1-2^{i-1}}[B_1 - f^{-1}[\bigcup_{j=1}^{i} B_j]]$ for $i=1,...,r$.

(C) $f^{-m}[B_1 - f^{-1}[\bigcup_{j=1}^{i-1} B_j]]$ *is the disjoint union of the two sets* $f^{-m-2^{i-1}}[B_1 - f^{-1}[\bigcup_{j=1}^{i} B_j]]$ and $f^{-m}[B_1 - f^{-1}[\bigcup_{j=1}^{i} B_j]]$, *each of cardinality* 2^{r-i} for $i=2,...,r$ and $m=0,...,2^{i-1}-2$.

Proof. Property (A) is trivial. Property (B) is proved by first establishing the two simple identities (1) $f[B_1] = \bigcup_{i=2}^{r+1} B_i$, and (2) $f^{2^{i-1}}[B_i] = \bigcup_{j=i+1}^{r+1} B_j$ for $i=1,...,r$. Next, we combine these identities to obtain (3) $f^{2^{i-1}}[B_i] = f[B_1] - \bigcup_{j=1}^{i} B_j$ for $i=1,...,r$. Finally, by applying $f^{-2^{i-1}}$ to both sides of identity (3), we confirm that $B_i = f^{1-2^{i-1}}[B_1] - f^{-2^{i-1}}[\bigcup_{j=1}^{i} B_j] = f^{1-2^{i-1}}[B_1 - f^{-1}[\bigcup_{j=1}^{i} B_j]]$ for $i=1,...,r$.

To prove property (C), we first note that f is a one-to-one function on $V - \{2^r\}$ and that $f^{-1}[B_i] \subset B_1$ for $i=1,...,r$. Consequently, for $i=1,...,r$, set $B_1 - f^{-1}[\bigcup_{j=1}^{i-1} B_j]$ is the disjoint union of sets $f^{-1}[B_i]$ and $B_1 - f^{-1}[\bigcup_{j=1}^{i} B_j]$, each having cardinality 2^{r-i} by property (A). Since set disjointness is preserved under function preimage operations, we see that $f^{-m}[B_1 - f^{-1}[\bigcup_{j=1}^{i-1} B_j]]$ is the disjoint union of $f^{-m}[B_1 - f^{-1}[\bigcup_{j=1}^{i} B_j]]$ and $f^{-m-1}[B_i]$, each having cardinality 2^{r-i} for $i=1,...,r$ and $m=0,...,2^{i-1}-2$. The result follows from using property (B) to replace the term B_i within expression $f^{-m-1}[B_i]$. □

Theorem 3.10. *i. If $(V, E_1 \cup ... \cup E_k)$ is acyclic, then the many function coarsest partition problem is solved by Hopcroft's algorithm in time $\Theta(kn\log n)$ and space $\Theta(kn)$ in the worst* *case. ii. It can be solved by multiset dag discrimination in time $O(kn)$ and space $O(n)$.*

Proof. i. It suffices to give an example of an input instance for just one function on which Hopcroft's algorithm can run in $\Omega(n \log n)$ steps. Consider any instance satisfying Lemma 3.9. Assume that Hopcroft's algorithm starts out by initializing stack W to be $[B_1,...,B_{r+1}]$. We can show that each time a block is popped from stack W, either partition P remains unchanged during the splitting step, or a single block not belonging to stack W is split in half. Furthermore, the final partition contains only singleton blocks. Since splitting a block in half takes time proportional to half the size of the block, the cumulative cost of splitting input block B_1 into singleton blocks is $\Theta(\#B_1 \log \#B_1) = \Theta(n \log n)$.

We use an inductive argument to establish the following claim. Right before each block B_i is popped from stack W for the first time $i=1,...,r$, then the following conditions hold.

(1) Stack W contains the original input blocks $[B_i, B_{i+1},...,B_{r+1}]$.

(2) The nonstack blocks of partition P are $f^{-m}[B_1 - f^{-1}[\bigcup_{j=1}^{i-1} B_j]]$ for $m=0,...,2^{i-1}-2$.

(3) The nonstack blocks of P all have twice the number of elements as B_i.

(4) No block in P can be split by any of the nonstack blocks.

(Basis) For $i=1$ properties (1–4) follow from the definition of the input instances and the fact that there are no nonstack blocks. Since no block in P is split by block B_1, the algorithm proceeds with stack $W = [B_2,...,B_{r+1}]$ and nonstack block B_1. Hence, conditions (1) and (4) hold for $i=2$. Condition (2) holds by inspection, and (3) holds by property (A) of Lemma 3.9.

(Induction) Assume that conditions (1-4) hold for any i > 1. By property (B) of Lemma 3.9, $B_i = f^{1-2^{i-1}}[B_1 - f^{-1}[\bigcup_{j=1}^{i} B_j]]$. Hopcroft's algorithm can be made to use blocks $f^{1-m-2^{i-1}}[B_1 - f^{-1}[\bigcup_{j=1}^{i} B_j]]$ to split nonstack blocks $f^{-m}[B_1 - f^{-1}[\bigcup_{j=1}^{i-1} B_j]]$ into two new blocks $f^{-m-2^{i-1}}[B_1 - f^{-1}[\bigcup_{j=1}^{i} B_j]]$ and $f^{-m}[B_1 - f^{-1}[\bigcup_{j=1}^{i} B_j]]$, each of cardinality 2^{r-i} for $m=0,...,2^{i-1}-2$ (which follows from properties (A) and (C) of Lemma 3.9). Moreover, the last of these blocks $f^{2-2^i}[B_1 - f^{-1}[\bigcup_{j=1}^{i} B_j]]$ splits partition P in the same way as its 'twin' $f^{2-2^{i-1}}[B_1 - f^{-1}[\bigcup_{j=1}^{i} B_j]]$, which cannot split P at all, because it equals B_i (by property (B) of Lemma 3.9). Therefore, conditions (1-4) hold for $i=1,...r$.

When block B_{r+1} is finally popped from stack W, P is already a collection of singleton blocks. Thus part (i.) of the theorem holds.

ii. This problem can be reduced to multiset dag discrimination as follows. First form a pointer-based directory of k elements, each one uniquely associated with a set V_i, $i=1,...,k$. Next, for $i=1,...,k$ and each element $x \in V_i$, define $label(x)$ to be a pointer to the 'directory' element uniquely associated with set V_i. Then remove all self-loops from 'leaves'. Finally, for each node $n \in V$ form its node list $[label(n), E_1(n),...,E_k(n)]$, which consists of its label followed by its ordered successors. If we interpret all leaf nodes as having height 1, we can then solve acyclic coarsest partitioning by multiset dag discrimination. □

Since the original version of our paper (which included Theorem 3.10 without proof) appeared in [6], we learned that Revuz [29] independently observed that acyclic coarsest partitioning could be solved in linear time. Revuz rediscovered the multiset discrimination algorithm originally described in [22], and independently used this procedure without modification to obtain the result. He did not show that Hopcroft's algorithm can take $\Omega(n \log n)$ steps on acyclic input instances. Our approach is likely to be more efficient than his, since our pointer-based dag discrimination avoids the numeric representation that is needed in order to make direct use of the algorithm found in [22].

3.5. The Sequence Congruence Problem

The sequence congruence problem [40,41] arises in the context of program integration. The problem is how to partition program components into classes whose members have equivalent execution behavior. The algorithm presented in [40,41] solves this problem in two phases: the program components are first partitioned with respect to a 'flow dependence' graph, and then refined with respect to a 'control' graph. Hopcroft's coarsest partition algorithm is used in both phases, giving the $O(m_1 \log m_1 + m_2 \log m_2)$ time complexity, where m_1 and m_2 are the sizes of the flow dependence graph and control graph respectively. Because their control graph is essentially acyclic, the linear time multiset dag discrimination method can be used for the second phase to improve their time bound to $O(m_1 \log m_1 + m_2)$.

3.6. Basic Block Optimization

Value numbering has been used as a standard program analysis technique for determining equalities of the values computed by instructions within basic blocks [11, 3]. Although the technique is mostly implemented with hashing, multiset discrimination can be used to obtain a more efficient implementation without hashing and without numeric representations.

Consider a basic block B consisting of a sequence of assignment statements $s_1, ..., s_k$, each of the form *lhs* := *rhs*, where *lhs* is a variable, and *rhs* is either a constant, a variable, or an expression of the form $op(x_1, ..., x_t)$ in which *op* is some t-ary operator, and $x_1, ..., x_t$ can be constants or variables. Assume that B is lexically scanned, and that variables and constants are represented by pointers to a symbol table as described previously. We want to determine equivalence classes of variable and expression occurrences in B that have the same run-time value. This is achieved in three steps.

1. Construct an initial dag representation $D = (V, E_1, ..., E_{tmax})$ of B with vertex set V and edge sets $E_1, ..., E_{tmax}$, where *tmax* is the maximum arity of the operators appearing in the instructions in B. The leaves of D represent the constants and initial values of variables, and internal nodes represent the values computed by right-hand-side expressions. If v is an internal node representing the value of right hand expression $op(x_1, ..., x_t)$ and if $v_1, ..., v_t$ are vertices in D representing the values of $x_1, ..., x_k$ used in $op(x_1, ..., x_t)$, then E_i contains the edge $v \rightarrow v_i$ for $i = 1, ..., t$.

   We construct D by scanning the statements in B in order from s_1 to s_k. During the scan, the vertex $node(x)$ in D, representing the current value of variable or constant x, is accessed through a pointer stored in the symbol table entry for x.

Let $z := rhs$ be the statement being scanned. For each argument x in *rhs* such that $node(x)$ is not defined, we assign a new node to $node(x)$ labeled by a pointer to the symbol table entry for x. Then consider the following cases. If *rhs* is a variable or a constant y, we simply set $node(z) = node(y)$. Otherwise *rhs* is of the form $op(x_1, ..., x_t)$. If $x_1, ..., x_t$ are all constants, then we enter the computed value $c = op(x_1, ..., x_t)$ into the symbol table, create a new node v labeled with a pointer to c, and set $node(z) = v$. Otherwise, create a new vertex v labeled *op*, set $node(z) = v$, and add the edge $v \rightarrow node(x_i)$ to E_i for $i = 1, ..., t$.

2. Step (1.) may create duplicate entries in the symbol table for the newly computed constants. Therefore we compress the symbol table by performing multiset discrimination of constants on all the new constant entries, and then adjust the pointers to these entries accordingly.
3. To recognize common subexpressions, we dagify D using multiset dag discrimination.

4. Reduction in Strength

The final three examples use the preceding techniques to obtain new solutions to strength reduction with worst case performance asymptotically better than the expected performance of the previous best algorithms. Ironically, the efficiency obtained stems from using batch techniques to implement strength reduction, which itself uses incremental techniques to improve program performance.

4.1. Basic Strength Reduction

First we consider a new hash-free algorithm that implements Cocke and Kennedy's strength reduction transformation [10]. The algorithm runs in worst case time and space linear in the length of the final program text, which, as we will show, can be as much as two orders of magnitude better than their hash-based algorithm.

Cocke and Kennedy's transformation is concerned with replacing hidden costs of linear polynomials involved in the array access formula used in programming languages like Fortran or Algol. As was suggested by Allen, Cocke, and Kennedy [4], the earlier transformation [10] can be improved by sharper analysis of control flow and taking safety of code motion into account. However, such improvement is orthogonal to the solution presented here.

The strength reduction transformation of [10] may be defined as follows. Let L be a strongly connected region of code. We assume that this code consists of assignments to simple (non-array) variables of the form $z{:=}op(x,y)$ or $z{:=}op(x)$ and conditional branches with boolean valued variables as predicates. We assume implicit assignment to certain designated input variables, and implicit output variables that are printed whenever they are assigned. All concern for control flow is simplified by accepting a most conservative assumption that the *control flow* graph representation of L forms a clique; i.e., that every two statements in L can be executed one after the other at runtime.

In accordance with conventional terminology (cf. [3]) variable occurrences on the left of an assignment are called *definitions*; all other variable occurrences are called *uses*. A variable that has no definitions in a region of code is called a *region constant variable*. A variable v is *live* on entry to a program region R if there is a sequence of instructions entirely within R beginning with an instruction on first entering R, ending with a use of v, and free from any definition to v save perhaps for the last instruction in the sequence.

Definition 4.1. If c is either a region constant variable of L or a constant, and if i is a variable defined in L, then product $i \times c$ is *reducible* if

(1) all definitions to i occurring in L are among the following forms: $i := j$, $i := -j$, $i := j + k$, $i := j - k$, $i := -j + k$, or $i := -j - k$, and

(2) for each definition of the form $i := j$ or $i := -j$ that occurs in L, then product $j \times c$ must be reducible; for each definition of the form $i := j + k$, $i := j - k$, $i := -j + k$, or $i := -j - k$ that occurs in L, then both products $j \times c$ and $k \times c$ must be reducible.

For each reducible product $i \times c$ that actually OCCURS within L, strength reduction transforms L as follows:

T1. Replace each occurrence of $i \times c$ in L by a new variable t_{ic} uniquely associated with text expression $i \times c$.

T2. If variable i is live on entry to L, then introduce assignment $t_{ic} := i \times c$ in a unique entry block (a detail we add to their transformation for correctness), which must be entered before entering L.

T3. Within L and just prior to each definition to i insert the following assignments to newly generated temporary variables t_{ic} uniquely associated with product $i \times c$.

T3.1 For definitions of the form $i := j$ and $i := -j$, insert the code $t_{ic} := j \times c$ and $t_{ic} := -j \times c$, respectively.

T3.2 For definitions of the the form $i := j + k$, $i := j - k$, $i := -j + k$, and $i := -j - k$, insert the code $t_{ic} := j \times c + k \times c$, $t_{ic} := j \times c - k \times c$, $t_{ic} := -j \times c + k \times c$, and $t_{ic} := -j \times c - k \times c$, respectively.

T4. If any of the products introduced in Step (T3.) has been previously eliminated by either code motion or strength reduction, replace it by its associated temporary variable. Remove all other products introduced in Step (T3.) by either code motion or recursive application of strength reduction as appropriate.

Like Cocke and Kennedy we assume that strength reduction is performed after redundant code elimination, constant propagation, and code motion. Given a strongly connected program region L as input, our initial solution to the preceding transformation shares the following first four steps of the Cocke and Kennedy algorithm.

A1. Compute the set RC of region constant variables of L and the set $Defs(v)$ of all definitions in L to each variable v defined in L.

A2. Compute the set IV of *induction variables*; that is, the set of all variables x that have definitions occurring in L such that product $x \times c$ would be reducible for any constant c. This procedure was also described by Cocke and Schwartz [11].

A3. Find the set *Cands* of all reducible products $x \times c$ that actually occur in L, and find the associated program points in L where these products occur.

A4. For each induction variable x, compute the set $Afct(x) = \{x\} \cup \{y$: y is a variable or constant on the right-hand-side of any assignment to x that occurs in $L\}$.

Although no algorithmic analysis was provided in [10], it is well known that Cocke and Kennedy's strength reduction algorithm carries out Steps (A1.-A4.) in worst case time and space linear in the length $\#L$ of program text L. Before describing the remaining steps of their and our algorithms, we need to take a closer look at the problem structure.

If *Afct* is regarded as a binary relation and *Afct** represents its transitive closure, then the following fact immediately follows from Cocke and Kennedy's paper.

Lemma 4.1. *The set of all reducible products removed from L by recursive application of strength reduction is defined by* $Rm = \{j \times c: \exists\, i \mid (i \times c \in Cands$ **and** $j \in Afct^*(i))\}$.

Calculation of *Afct* and *Rm* is central to the implementation of strength reduction, and the linearity of our algorithm depends on the following fact.

Lemma 4.2. *If L represents the initial program text, and L′ represents the final program text after strength reduction has been applied, then* $\#Rm = O(\#L + \#L')$ *and* $\#Afct = \Theta(L)$.

Proof. For any product $v \times c \in Rm$, there are three cases to consider. If the product also belongs to *Cands*, then it appears in L. Otherwise, if v is an induction variable, but $v \times c$ doesn't belong to *Cands*, then L' will have at least one definition to variable t_{vc} that is introduced by transformation Step (T3.) and uniquely associated with the product. Otherwise, L' will have at least one use of variable t_{vc} that is introduced by Steps (T3.) and (T4.), and can be uniquely associated with the product.

For any x and y such that $y \in Afct(x)$, there are two cases to consider. In the first case y is the same as induction variable x, and we can uniquely associate the pair $[x,x]$ with at least one definition to x that occurs in L. Otherwise, we can uniquely associate the pair $[x,y]$ with a use of variable or constant y occurring in an assignment to x inside L. □

In order to compute *Rm* in time $O(\#Rm)$, we need to avoid the redundant computation that arises when products $i \times c$ and $j \times c$ both belong to *Cands*, and $Afct^*(i) \cap Afct^*(j)$ is nonempty. In this case there may be substantial overlap between subsets $\{k \times c \in Rm \mid k \in Afct^*(i)\}$ and $\{k \times c \in Rm \mid k \in Afct^*(j)\}$ of *Rm*.

Two other sources of redundancy in *Rm* are more innocuous, but worth mentioning. The first case arises when $i \times c_1$ and $j \times c_2$ belong to *Cands*, $c_1 \in Afct^*(j)$, and $c_2 \in Afct^*(i)$. In this case *Rm* will contain both $c_2 \times c_1$ and $c_1 \times c_2$. The second case arises when *Rm* contains different products with completely different constant arguments that evaluate to the same value. Redundant products for both of these cases may be included in our initial calculation of *Rm*, and eliminated during a postpass cleanup. Such simplification will reduce the number of variables in L', but will not eliminate the variable uses on which the counting argument of Lemma 4.2 is based. Our algorithm combines multiset discrimination with other data structuring techniques.

It is at this point that our solution differs from Cocke and Kennedy. They go on to compute the transitive closure *Afct** in time $\Theta(n^3+m)$ using, say, Warshall's algorithm[37] (see also[2]), where n is the number of variables and constants contained in *Afct*, and m is the number of assignments to induction variables. Because this transformation is based on a most conservative assumption that the control flow graph for region L forms a clique, the transitive closure *Afct** is likely to be large, so that any heuristic approach to compute it is not likely to improve substantially on Warshall's algorithm. Cocke and Kennedy also use a greedy strategy committed to hashing each product removed by strength reduction.

In contrast, we compute the strong components of the directed graph G representing *Afct* inverse; i.e., G has directed edge $i \rightarrow j$ *iff* $i \in Afct(j)$. Graph G is constructed in $\Theta(m)$ time and space using multiset discrimination of sequences (of length two) to eliminate multi-edges that arise from different assignments to the same variable that have uses of the same variables or constants. The strong component decomposition of G is computed by Tarjan's algorithm [36]. Note that the roots of G (i.e., nodes with in-degree 0) represent constants and region constant variables. All other nodes of G represent induction variables. For convenience we will sometimes refer to the nodes of G in terms of the variables and constants that they represent.

The dag structure of the strong component decomposition of G is used to efficiently compute *Rm* in time $O(\#L')$. The algorithm rests on the following obvious fact:

Lemma 4.3. *Let* $Cs = \{c: \exists\, i \mid i\times c \in Cands\}$. *For each* $c \in Cs$ *let* $Cmps(c)$ *be the set of strong components of G containing some variable i for which* $i\times c \in Cands$. *Then for each* $c \in Cs$, *the set of all products* $j\times c \in Rm$ *removed by strength reduction is defined by* $Rm(c) = \{j\times c$: *there is a path in G from j to any component of* $Cmps(c)\}$.

Proof. By Lemma 4.1 a product $j\times c$ is reducible for fixed constant or region constant variable c *iff* $\exists\, i \mid (i\times c \in Cands$ **and** $j \in Afct^*(i))$. Based on our graph representation G of *Afct* inverse, there is a path from node j to node i in G *iff* $j \in Afct^*(i)$. Based on the definition of strongly connected component (i.e., a subgraph S of a directed graph in which there is a path entirely inside S from every node in S to every node in S), there is a path from node j to node i in G *iff* there is a path from node j to the strong component containing node i. Hence, $j\times c$ is reducible for constant or region constant variable c *iff* $\exists\, Cm \in Cmps(c)$ such that there is a path in G from j to Cm. □

The remaining steps of our algorithm are given just below along with the algorithmic analysis:

A5. Simultaneously compute the sets *Cs* and $Cmps(d)$ for each $d \in Cs$ (as described in Lemma 4.3) using basic multiset discrimination of identifiers c occurring in products $i\times c \in Cands$. At the same time mark those variables v in G such that $v\times c$ belongs to *Cands*. If the symbol table is a pointer-based directory, and each product in *Cands* is a pair of pointers to the directory, then this whole step takes $O(\#Cands)$ time and space.

A6. Let *Scd* be the dag representation of the strong component decomposition of G. Initialize an empty multiset *Mrc* of products $v\times c$ and an empty multiset *Mc* of numeric constants. For each constant $c \in Cs$ repeat Steps (A6.1) and (A6.2)

A6.1 Compute the set $Scd_c = \{v : v\times c \in Rm(c)\}$ using a depth-first-search through dag *Scd* in the reverse direction of its edges and starting from components belonging to $Cmps(c)$. (Recall from previous discussion that for each strong component of *Scd*, if it has no incoming edges, then its entries are constants or region constant variables; otherwise, its entries are induction variables.) For each identifier $v \in Scd_c$, consider three cases. (1) If v is an induction variable, then link it to a new symbol table entry containing unique identifier t_{vc}. If v is live on entry to L, then insert assignment $t_{vc} := v \times c$ on entry to L. If v is marked, indicating that $v \times c \in Cands$, then replace each occurrence of $v \times c$ in L by a pointer to the symbol table entry for t_{vc}. (2) If $v \in Scd_c$ is a region constant variable, then link it to a new entry in *Mrc* containing product $v \times c$. (3) if $c' \in Scd_c$ is a constant, then link c' to a new entry in *Mc* containing the computed value of $c \times c'$.

A6.2 For each induction variable v in Scd_c and each assignment to v in $Defs(v)$, introduce update code to t_{vc} according to strength reduction transformation Step (T3.). Replace products that are introduced within this update code by references to the symbol table, *Mrc*, or *Mc* as is indicated by the links in Scd_c.

The depth first search through *Scd* in Step (A6.1) takes linear time in the number of edges traversed. Each assignment to an induction variable $v \in Scd_c$ in L corresponds to one or two edges in the portion of the dag traversed in Step (A6.1). Hence, the number of such assignments must be greater or equal to half the number of edges traversed in Step (A6.1). For each such assignment, update code is introduced by Step (A6.2). Hence, Step (A6.) takes time and space $O(\#L')$.

A7. Use multiset discrimination of sequences (of length two in this case) belonging to *Mrc* and discrimination of constants belonging to *Mc* respectively to find and eliminate duplicate region constant expressions in *Mrc* and duplicate values in *Mc*. Consequently, augment the symbol table with new variables for each distinct item in *Mrc* and *Mc*. At the same time re-adjust pointers inside L to the new symbol table entries, and insert an assignment $t_{c_1 c_2} := c_1 \times c_2$ on entry to L for each product $c_1 \times c_1 \in Mrc$. This whole Step takes time and space $O(\#Mrc + \#Mc)$, which is bounded by the number of edges traversed in Step (A6.).

The preceding discussion establishes the following theorem.

Theorem 4.4. *The Strength Reduction Transformation (T1 - T4) can be performed in worst case time and space O(#L').*

4.2. Strength Reduction With Useless Code Elimination

Cocke and Kennedy noted that after strength reduction is applied, it is necessary to apply global cleanup transformations such as useless code elimination (i.e., elimination of statements not contributing to the output) and variable subsumption (i.e., eliminating useless copy operations). In this section we show how to fold useless code elimination together with strength reduction. Our hash-free solution runs in worst case time and space linear in the sum of the lengths of the initial and final program texts.

As before we assume that L is a strongly connected region of code, and $Defs(v)$ is a set of all definitions in L to each variable v defined in L. Instead of computing *Cands* directly, we compute the set *Prods* of all products appearing in L and the places where they occur. Also, the set *IV* of induction variables is not computed explicitly, but is detected implicitly in a simpler way.

By a *spoiler* we mean any variable v for which $Defs(v)$ contains a definition not among the forms $v := \pm j$ or $v := \pm j \pm k$. We compute the set *Spoilers* of all such variables.

Finally, we generalize relation *Afct* so that $Afct(x)$ is defined for each variable x (and not just induction variables) that is assigned within L. As before, let directed graph G denote *Afct* inverse. Once again, we compute the strong component decomposition dag *Scd* of G.

Recall that those strong components of G with no incoming edges contain only constants and region constant variables. All other strong components of G contain induction variables or spoilers. Any product $x \times c \in Prods$ also belongs to *Cands iff* there is no path in G from a spoiler to x. If we mark all strong components containing spoilers, and mark all other components reachable from these marked components, then the unmarked portion of G corresponds precisely to the data structure at the heart of the strength reduction algorithm in the preceding subsection. Recall that the induction variables are all those variables contained in unmarked strong components that have incoming edges.

The preceding discussion simplifies the first four Steps (A1. - A4.) of Cocke and Kennedy's algorithm, and leads to an alternative linear time strength reduction algorithm that continues with Step (A5.) of the last subsection. It also supports a new algorithm that includes efficient analysis for and elimination of useless code.

Consider how the new graph G_{new} of the program region L' after strength reduction differs from the initial graph G_{old} of L.

Lemma 4.5.

i. The subgraph of G induced by unmarked strong components, and the subgraph of G induced by marked strong components are both invariant with respect to strength reduction.

ii. The new strong components in G_{new} *only contain temporary variables; the only edges incident to these components are between each other and from them to marked components.*

iii. Edges only go from unmarked to marked components, and these can only be deleted by strength reduction.

Proof. Strength Reduction alters loop L in the following two ways:

(1) Assignments are introduced within L to modify temporary variables t_{xc} by Steps (T3.) and (T4.). The right-hand-side of any such assignment must contain only temporary variables. Hence, these assignments cannot create new edges from G_{old} to any strong components in G_{new} containing temporary variables.

(2) An assignment $z := x \times c$ appearing in L can be replaced by assignment $z := t_{xc}$. In this case, variable z must be a spoiler (since a product is assigned to it in L) that belongs to a marked strong component Scd_z of G_{old}, and x must appear in an unmarked component Scd_x of G_{old}. Moreover, there must be an edge from Scd_x to Scd_z. After product $x \times c$ is replaced, there would be an edge from the strong component in G_{new} containing t_{xc} to Scd_z. Moreover, the edge from x (respectively c) to z would be deleted in G_{new} when no assignment to z remains with a use of x (respectively c) on the right-hand-side. □

Let *inputs* be the set of input variables, *outputs* be the set of output variables, and *controls* be the set of predicate variables of control statements. We will assume that these variables are all *useful*, and that the strong components of G containing them, which we call the critical set *crit*, are also useful. The useful components include *crit* and all strong components of G that can reach the components in *crit*.

If we assume that all statements in L are initially useful, then after strength reduction is applied to L once, only induction variables, region constant variables, and constants can become useless. Temporary variables introduced by strength reduction must all be useful. Consequently, only the replacement of products by temporaries can create useless code. And all statements that undergo such replacement will be useful in the end.

Hence, we can modify the algorithm in the previous subsection to facilitate useless code elimination as follows. For each edge $v \rightarrow z \in G$ store a count of the number of distinct uses of v on the right hand side of all assignments to z occurring L. This is implemented using a pointer linking each assignment $z{:=}v{\times}c$ in L directly into the adjacency list for G. In step (A6.1), for each assignment $z{:=}v{\times}c$ replaced by assignment $z{:=}t_{vc}$, decrement the edge counts for edges $v \rightarrow z$ and $c \rightarrow z$ in G. If any edge count reaches zero, that edge is deleted from G. Also, add new edges from t_{vc} to z in G. In Step (A6.2) introduce a new edge in G for each assignment to a temporary variable introduced. In Step (A7.) multiset discrimination will determine the new vertices corresponding to new temporary variables in G. Add a final step (A8.) in which the useful variables of G are computed by a linear time search through the inverse of G (i.e., in the opposite direction of edges in G) starting from vertices in *crit*. Within L all assignments to variables not in useful components can be removed.

By the preceding discussion we have

Theorem 4.6. *The Strength Reduction Transformation (T1 - T4) combined with useless code elimination can be performed in worst case time and space O(#L+#L').*

4.3. Iterated Strength Reduction With Useless Code Elimination

Cocke and Kennedy noted that after strength reduction is applied, the new compiler generated variables t_{vc} and other variables can become new induction variables, and new products defined in terms of these variables can be removed by further applications of strength reduction [10]. In this section we show how iterated strength reduction folded with useless code elimination can be solved in worst case time and space linear in the maximum length of the initial and final program texts.

Note, first of all, that iterated strength reduction terminates, because each iteration except the last must eliminate at least one product in the original strongly connected region L. In order to achieve the promised linear time complexity, we must be careful to generate only temporaries that are not useless.

Let L' be the final code resulting from iterated strength reduction. Let *Cands*$'$ be the set of products in the initial code L reduced by iterated strength reduction. Following Cocke and Schwartz [11], we say that a temporary $t_{vc_1 \ldots c_j}$ is *available* in program region L' if, whenever it is referenced during execution of L', it stores the value of $v{\times}c_1{\times}\ldots{\times}c_j$. In this case, we say that the sequence $[c_1,\ldots,c_j]$ of region constant variables and constants is a *tail* of v. The set of tails of a variable or constant v is denoted by $tails(v)$. By default, $t_v = v$, and $[] \in tails(v)$ *iff* v is not useless in L' (recall that [] denotes the empty sequence). The main task of the algorithm is to determine $tails(v)$ for each variable and constant v appearing in L. It is then straightforward to introduce temporary variables and generate the code to keep them available.

First consider the preprocessing. We use the same graph representation G as in the last subsection. For each edge $x \rightarrow y \in G$, we can compute $tails(x)$ from $tails(y)$ by making use of a set $label(x,y)$, which is defined inductively as the smallest set satisfying the following rules.

(1) $labels(j,v)$ includes [] for each instruction $v := \pm\, j$, $v := \pm\, j \pm k$ or $v := \pm\, k \pm j$ that occurs in L.

(2) $labels(j,v)$ includes c for each instruction $v{:=}j{\times}c$ that occurs in L, where $c \in RC$ or c is a constant.

(3) $labels(j,v)$ includes [] for each instruction $v := ...j...$ not mentioned above. In this case we also mark v as a *spoiler*.

Let *Scd* be the dag representation of the strong component decomposition of G. The vertices of *Scd* are the strong components of G. *Scd* has an edge $C_1 \rightarrow C_2$ between two strong components C_1 and C_2 *iff* G contains an edge from a node in C_1 to a node in C_2. We further extend the definition of labels to edges in *Scd*. For each edge $C_1 \rightarrow C_2$ in *Scd*, we define $Labels(C_1, C_2) = \{c\colon c \in labels(x, y), x \in C_1, y \in C_2\}$.

We say that a component $C \in Scd$ is *clean* if none of its elements are spoilers, and if every edge $x \rightarrow y$ in G between any two nodes x and y in C has $labels(x, y) = \{[]\}$. We say that C is *reducible* if all of its ancestors in *Scd* are clean. It is not difficult to see that a variable v occurring in L becomes an induction variable in some iteration of Cocke and Kennedy's algorithm *iff* v belongs to a reducible component. Therefore if v belongs to a non-reducible component, then none of the products $v{\times}x$ in L are reducible. Since we assume that all variables occurring in L are useful initially, then the variables belonging to non-reducible components remain useful in L'.

It is straightforward to compute the reducible components of *Scd* by processing the components of *Scd* in topological order in the same direction as the edges. Each clean component $C \in Scd$ is reducible if all of its predecessors are reducible; otherwise it is not reducible. It follows that *tails* can be computed inductively according to the following lemma.

Lemma 4.7.

i. The set Cands′ consists of all those products $v{\times}c$ occurring in L such that $c \in RC$ or c is a constant, and v belongs to a reducible component.

ii. If $v \rightarrow x$ is an edge in G, $[] \in labels(v,x)$, and $[] \in tails(x)$, then $[] \in tails(v)$.

iii. If $v \rightarrow x$ is an edge in G, $c \in labels(v,x)$, $p \in tails(x)$, and v belongs to a reducible component, then the sequence $c\ p \in tails(v)$.

iv. If v belongs to a non-reducible component, then $tails(v) = \{[]\}$.

Let C be a component in *Scd*, and let $x, y \in C$. If C is reducible, then $tails(x) = tails(y)$ by Lemma 4.7 (iii). If C is not reducible, then $tails(x) = tails(y) = \{[]\}$ by Lemma 4.7 (iv). In either case, we define $Tails(C) = tails(x)$ for some arbitrary $x \in C$. Thus, instead of computing the tails of variables, we can compute the tails of the components in *Scd*.

One simple way of computing *Tails* is as follows:

for C in *Scd* in topological order in the opposite direction of its edges **loop**
    if C is not reducible **then**
        $Tails(C) := \{[]\}$;
1   **else** $Tails(C) := \bigcup_{C_i \in succ(C)} \{\, x\ s \colon x \in Labels(C, C_i),\ s \in Tails(C_i) \,\}$;
        if C contains output variables **then**
            $Tails(C) := Tails(C) \cup \{[]\}$;
        end if;
    end if;
end loop;

where $succ(C_i)$ is the set of successors of C_i in *Scd*. Although multiset sequence discrimination could be used in computing the union in line 1, the $\Omega(\#s)$ worst case cost due to each sequence s in $Tails(C)$ is too slow. More efficient is to modify the preceding algorithm to generate all tails of a given length before applying multiset discrimination.

Suppose Strength Reduction Transformation (T1 - T4) is iterated k times before no new products are reduced. The ith such iteration reduces products of $i+1$ arguments, which corresponds to our computation of tails of length i for $i=1,\ldots,k$.

Our algorithm will compute tails of length $i=0,\ldots,k$ in ‘round’ i. Initially, $Tails(C)$ is empty for all components $C \in Scd$. In round $i=0,\ldots,k$, we compute the tails of length i for each component C, assuming that no new tails are generated in round $k+1$. When $i = 0$, useful program variables are detected. After each round $i=1,\ldots,k$ we assign a unique identifier for each distinct tail of length i. Thus, in round $i+1$, each newly generated sequence $[c_1, c_2, \ldots, c_{i+1}]$ can be represented by a pair $[c_1, n_1]$, where n_1 is the name for $[c_2, \ldots, c_{i+1}]$. Consequently, in order to determine distinct tails generated in each ith round, where $i > 1$, multiset sequence discrimination is only needed for sequences of length 2. Implementation details are given below.

Let $pred_1 = \{\, [C_1, C_2] \colon [C_2, C_1] \in Scd \mid C_2$ *is reducible and* $[] \in Labels(C_2, C_1)\}$, and let $pred_2 = \{\, [C_1, C_2] \colon [C_2, C_1] \in Scd \mid C_2$ *is reducible and* $Labels(C_2, C_1)$ *contains some label* $c \neq []\}$. We will use $pred_2$ to generate tails of length i from tails of length $i-1$, and use $pred_1$ to propagate tails between components. Let $Tails(C, i)$ be the set of tails of C of length i. Let $Heads(i)$ be the set of components such that $Tails(C, i)$ is not empty. Initially, we set $Heads(0) = \{\}$. Then for $i=0,\ldots,k$, we compute tails of length i:

(1) generate tails of length i
    propagate tails of length i using $pred_1$

Tails of length 0 are generated according to Lemma 4.7 (iv) as follows.

(2) **for** C in *Scd* **loop**
        if C is not reducible or C contains any output variable **then**
            $Tails(C, 0) := \{[]\}$
            $Heads(0) := Heads(0) \cup \{C\}$
        end if
    end loop

Tails of length 1,...,k are generated according to Lemma 4.7 (iii) as follows.

```
(3)     for C_1 in Heads(i−1) loop
            for C in pred_2(C_1) loop
                Tails(C,i) := {}
                Heads(i) := Heads(i) ∪ {C}
            end loop
        end loop
        for C_1 in Heads(i−1) loop
            for C in pred_2(C) loop
2               Tails(C,i) := Tails(C,i) ∪ { c s: c ∈ Labels(C,C_1) − {[]}, s ∈ Tails(C_1,i−1) }
            end loop
        end loop
        perform multiset sequence discrimination on  ∪_{C ∈ Heads(i)} Tails(C,i)
```

For $i = 0, 1, ..., k$, tails of length i are propagated in *Scd* according to Lemma 4.7 (ii):

```
(4)     Heads(i) := {C ∈ Scd | C is reachable from the components in Heads(i) through edges in pred_1}
        for C_1 in Heads(i) in topological order w.r.t. pred_1 in the direction of its edges loop
            perform multiset sequence discrimination on Tails(C_1,i)
            for C in pred_1(C_1) loop
3               Tails(C,i) := Tails(C,i) ∪ Tails(C_1,i)
            end loop
        end loop
```

The above representation of sequences can also be used to initialize temporaries. If $s = [c_1, ..., c_k]$ is a tail generated in round k for some $k > 1$, and if n_1 is the name of $[c_2, ..., c_k]$, then we use t_s to store the value of $c_1 \times ... \times c_k$, and insert an assignment $t_s := c_1 \times t_{n_1}$ at the end of the initialization block. Once all the tails are initialized, we insert an assignment $t_{vs} := v \times t_s$ at the end of the initialization block for each temporary t_{vs}.

The rest of the algorithm includes: (1) replacing products in *Cands′* by temporaries, (2) inserting code to keep temporaries available, and (3) eliminating dead code. The first two tasks are straightforward, and the third one can be done easily with the help of the *Tails* sets.

To analyze our algorithm, we note that for each tail $c\ s$ ever added to $Tails(C,i)$ at line 2 in code (3), there exists at least one instruction $v := j \times c$ in L such that $v \in C_1$ and $j \in C$. Thus, a distinct instruction should be inserted to keep the temporary t_{vp} available. Similarly, for each tail $p \neq []$ ever added to $Tails(C,i)$ at line 3 in code (4), there exists an instruction $v := \pm x$ or $v := \pm x \pm y$ in L with respect to which we need to insert an instruction to keep the temporary t_{vp} available. Therefore the accumulated cost of code (1) is bounded from above by the size of the output code. Consequently, we have,

Theorem 4.8. *The iterated strength reduction problem with useless code elimination can be solved in time and auxiliary space* $O(\#L + \#L')$.

Our algorithm is theoretically superior to an iterated form of Cocke and Kennedy's algorithm. Let L_i be the program text before the ith iteration of Cocke and Kennedy's algorithm. Even if we perform dead code elimination after each iteration, the size of L_i could still be as large as $\Omega(\#L\#L')$, since some inserted code may stay in the program for $\Omega(\#L)$ iterations before becoming dead. Because of Lemma 4.3, each induction variable or region constant in L_i can be in the set $Afct^*(x)$ for at most $\#L$ induction variables x. Thus the transitive closure $Afct*$ can be computed in $\Theta(\#L\#L_i) = \Theta(\#L^2\#L')$ time if we use hashing for set element addition. Therefore iterating Cocke and Kennedy's algorithm can take $\Theta(\#L^3\#L')$ hash operations in the worst case. A closer look at their algorithm reveals that $Afct*(x)$ must only be computed for those variables x such that $x \times c$ is a candidate product in L_i for some c. Even with this optimization, iterated strength reduction with Cocke and Kennedy's algorithm takes $\Theta(\#L\#L')$ hash operations in the worst case.

4.4. Extensions

Two possible approaches that exploit commutative and associative laws of products may reduce the number of strings and therefore temporaries generated in the preceding strength reduction algorithms. One approach is to use a weak form of the Paige/Tarjan lexicographic sorting algorithm[22] to generate strings of constants in some arbitrarily chosen order. Another more effective, but less efficient, approach, would be to actually compute the product of constants identifying each temporary, and to use multiset constant discrimination.

We are currently investigating these ideas as well as extensions that implement a more powerful transformation integrating strength reduction of sums, products, quotients, exponentiations, and multivariate expressions. Such extensions would allow different kinds of spoilers for different arguments of candidate expressions. Development of simpler hash-based algorithms is another promising direction.

5. Conclusion

We have presented new and theoretically efficient hash-free solutions to an assortment of programming language problems. Each of our solutions has worst case asymptotic time and space complexities that either match or improve upon the expected time and worst case space of the best previous solution that involved hashing (under the assumption that each hash operation takes $O(1)$ expected time). In the case of iterated strength reduction, our solution has worst case time three orders of magnitude better than the expected time of the best previous solution.

All of our solutions have been based in large part on efficient multiset discrimination methods for datatypes built up from basic datatypes implemented using directories and from sequence constructors. Such datatypes include strings, lists, ordered trees, and ordered dags. Multiset discrimination of arbitrary datatypes formed from identifiers and constructors for sequences, sets, and multisets are found in [23]. The generic nature of these methods, and their wide ranging successful applications to language processing problems demonstrates a basic algorithm design tool for solving problems in various other areas of computer science.

The replacement of solutions based on hashing with solutions based on multiset discrimination illustrates a fundamental principle of algorithm improvement. All of the problems considered in this paper are batch problems; i.e., problems in which all of the input is made available at the beginning of computation. However, each of these problems was previously solved by reducing it to a simpler on-line problem; i.e., by breaking the problem up into a sequence of simpler subproblems each of which uses only a portion of the original input and contributes to only a portion of the output. Each solution to these on-line reformulations made use of hashing. This approach has the advantage of a simple easily implementable algorithm with local strategy in which decisions are made based on a small portion of input. In this paper each of these problems has been reformulated in terms of multiset discrimination, which is a batch subproblem. Although our algorithms are more complicated, they have better theoretical performance, partly because they depend on a more global strategy in which decisions are made based on larger portions of input, and because multiset discrimination can be solved efficiently.

Although all of the algorithms given in this paper have better theoretical performance than their predecessorrs, it is not clear whether they have any computational advantage. An empirical investigation comparing our hash-free alternatives with their conventional hash-based counterparts would be worthwhile future work.

6. Acknowledgement

We thank Ralph Wachter, whose workshop on randomized algorithms brought to our attention questions about randomized versus deterministic algorithms, which, we felt, raised similar questions about hash-based versus hash-free algorithms, and led to the current paper.

References

1. no author, *ADA UK News*, Vol 6, Num 1, pp. 14-15, Jan., 1985.
2. Aho, A., Hopcroft, J., and Ullman, J., *Design and Analysis of Computer Algorithms,* Addison-Wesley, 1974.
3. Aho, A., Sethi, R. and Ullman, J., *Compilers,* Addison-Wesley, 1986.
4. Allen, F. E., Cocke, J., and Kennedy, K., ‘‘Reduction of Operator Strength,’’ in *Program Flow Analysis,* ed. Muchnick, S. and Jones, N., pp. 79-101, Prentice Hall, 1981.
5. Alpern, B., Wegman, N., and Zadeck, K., ‘‘Detecting Equality of Variables in Programs,’’ in *Proc. 15th ACM POPL*, Jan, 1988.
6. Cai, J. and Paige, R., ‘‘"Look Ma, No Hashing, And No Arrays Neither",’’ in *ACM POPL,* pp. 143 - 154, Jan, 1991.
7. Carter, J. and Wegman, M., ‘‘Universal Classes of Hash Functions,’’ *JCSS*, vol. 18, no. 2, pp. 143-154, 1979.
8. Clinger, W. and Rees, J., ‘‘Macros That Work,’’ in *Proceedings 8th ACM Symposium on Principles of Programming Languages,* pp. 155-162, Jan, 1991.
9. Cocke, J., ‘‘Global common subexpression elimination,’’ *ACM SIGPLAN Notices*, vol. 5, no. 7, pp. 20 - 24, 1970.
10. Cocke, J. and Kennedy, K., ‘‘An Algorithm for Reduction of Operator Strength,’’ *CACM*, vol. 20, no. 11, pp. 850-856, Nov., 1977.
11. Cocke, J. and Schwartz, J. T., *Programming Languages and Their Compilers,* Lecture Notes, CIMS, New York University, 1969.

12. Cytron, R., Lowry, A., and Zadeck, K., ‘‘Code Motion of Control Structures in High-level Languages,’’ IBM Research Center/Yorktown Heights, 1985.

13. Downey, P., Sethi, R., and Tarjan, R., ‘‘Variations on the Common Subexpression Problem,’’ *JACM*, vol. 27, no. 4, pp. 758-771, Oct., 1980.

14. Earley, J., ‘‘High Level Iterators and a Method for Automatically Designing Data Structure Representation,’’ *J of Computer Languages*, vol. 1, no. 4, pp. 321-342, 1976.

15. Fong, A., ‘‘Elimination of Common Subexpressions in Very High Level Languages,’’ in *Proceedings Fourth ACM Symposium on Principles of Programming Languages*, pp. 48-57, Jan, 1977.

16. Hoffmann, C. and O’Donnell, J., ‘‘Pattern Matching in Trees,’’ *JACM*, vol. 29, no. 1, pp. 68-95, Jan, 1982.

17. Hopcroft, J., ‘‘An n log n Algorithm for Minimizing States in a Finite Automaton,’’ in *Theory of Machines and Computations*, ed. Kohavi and Paz, pp. 189-196, Academic Press, New York, 1971.

18. Knoop, J. and Steffen, B., *Strength Reduction based on Code Motion*, Institute Fur Informatik und Praktische Mathematik, Christian-Albrechts-University, Kiel, Germany, Feb., 1991.

19. Knuth, D. E., *The Art of Computer Programming, Vol 1: Fundamental Algorithms*, Addison-Wesley, 1973.

20. Lewis, F., Rosencrantz, D., and Stearns, R., *Compiler Design Theory*, Addison-Wesley, 1976.

21. Mairson, H., ‘‘The Program Complexity of Searching a Table,’’ in *24th IEEE FOCS*, pp. 40-47, Nov., 1983.

22. Paige, R and Tarjan, R., ‘‘Three Efficient Algorithms Based on Partition Refinement,’’ *SIAM Journal on Computing*, vol. 16, no. 6, pp. 973-989, Dec., 1987.

23. Paige, R., ‘‘Efficient Translation of External Input in a Dynamically Typed Language,’’ in *Proc. IFIP Congress 94 - Vol 1*, ed. B. Pehrson and I. Simon, pp. 603-608, Elsevier North-Holland, 1994.

24. Paige, R., ‘‘Symbolic Finite Differencing - Part I,’’ in *Proc. ESOP 90*, ed. N. Jones, Lecture Notes in Computer Science, vol. 432, pp. 36-56, Springer-Verlag, 1990.

25. Paige, R., ‘‘Real-time Simulation of a Set Machine on a RAM,’’ in *ICCI ’89*, ed. R. Janicki and W. Koczkodaj, Computing and Information, Vol II, pp. 69-73, 1989.

26. Paige, R., Tarjan, R., and Bonic, R., ‘‘A Linear Time Solution to the Single Function Coarsest Partition Problem,’’ *TCS*, vol. 40, no. 1, pp. 67-84, Sep, 1985.

27. Paterson, M. and Wegman, M., ‘‘Linear Unification,’’ *Journal of Computer and System Science*, no. 16, pp. 158 - 167, 1978.

28. Pelegri-Llopart, E., ‘‘Rewrite Systems, Pattern Matching, and Code Generation,’’ Ph.D. Dissertation, U. of CA - Berkeley, 1987.

29. Revuz, D., ‘‘Minimisation of acyclic deterministic automata in linear time,’’ *Theoretical Computer Science*, vol. 92, no. 1, pp. 181 - 189, 1992.

30. Rosen, B. K., ‘‘Degrees of Availability,’’ in *Program Flow Analysis*, ed. Muchnick, S., Jones, N., pp. 55-76, Prentice Hall, 1981.

31. Schwartz, J., Dewar, R., Dubinsky, D., and Schonberg, E., *Programming with Sets: An introduction to SETL,* Springer-Verlag, 1986.

32. Stearns, R., ‘‘Deterministic top-down parsing,’’ in *Proc. 5th Princeton Conf. on Information Sciences and Systems*, pp. 182-188, 1971.

33. Straub, R., ‘‘Taliere: An Interactive System for Data Structuring SETL Programs,’’ Ph.D. Dissertation, Dept. of Computer Science, New York University, New York, NY, May 1988.

34. Tarjan, R., ‘‘A Class Of Algorithms Which Require Nonlinear Time To Maintain Disjoint Sets,’’ *J. Comput. Sys. Sci.,* vol. 18, pp. 110-127, 1979.

35. Tarjan, R., *Data Structures and Network Algorithms*, SIAM, 1984.

36. Tarjan, R., ‘‘Depth first search and linear graph algorithms,’’ *SIAM J. Comput*, vol. 1, no. 2, pp. 146-160, 1972.

37. Warshall, S., ‘‘A Theorem on Boolean matrices,’’ *JACM*, vol. 9, no. 1, pp. 11-12, 1962.

38. Wegman, M. N. and Zadeck, F. K., ‘‘Constant Propagation with Conditional Branches,’’ in *Proc. 12th ACM POPL*, Jan, 1985.

39. Willard, D., ‘‘Quasilinear Algorithms for Processing Relational Calculus Expressions,’’ in *Proc. PODS*, pp. 243-257, 1990.

40. Yang, W., Horwitz, S., and Reps, T., ‘‘Detecting program components with equivalent behaviors,’’ TR-840, Computer Sciences Dept., Univ. of Wisconsin, Madison, WI, April 1989.

41. Yang, W., ‘‘A new algorithm for semantics-based program integration,’’ Ph.D. Dissertation, TR 962, Computer Sciences Dept., Univ. of Wisconsin, Madison, WI, August 1990.