Multiset Discrimination – a Method for Implementing Programming Language Systems Without Hashing [1]

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

New York University/ Courant Institute
New York, NY 10012

ABSTRACT

It is generally assumed that hashing is essential to many algorithms related to efficient compilation; 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. Underlying this methodology are several generic algorithmic tools, among which special importance is given to *Multiset Discrimination*, which partitions a multiset into blocks of duplicate elements. We show how multiset discrimination, together with other tools, can be tailored to rid compilation of hashing without loss in asymptotic performance. Because of the simplicity of these tools, our results 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 takes $\Omega(|L|^3|L^*|)$ 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, left factoring, value numbers, acyclic coarsest partition, sequence congruence, reduction in strength

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

---

1. A preliminary version of this paper appeared in the Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages[5]. Part of this work was done while both authors were visiting the University of Wisconsin at Madison.

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. N00014-90-J-1890 and by Air Force Office of Scientific Research Grant No. AFOSR-91-0308.

and, consequently, raise some doubts about the prevailing view that hashing (e.g., universal hashing[6]) is essential to the various aspects of compilation from symbol table management [2] to reduction in strength by hashed temporaries [7].

Aho, Sethi, and Ullman [2] 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 the more general basic block optimization of value numbering. Hashing is involved in preprocessing for global optimizations to perform constant propagation [26], global redundant code elimination [4], and code motion [9]. The best methods of strength reduction [3, 7] rely on hashed temporaries to obtain practical implementations.

There are several reasons why hashing is used in these applications. Hashing has O(1) expected time performance and linear auxiliary space. The method of Universal Hashing, due to Carter and Wegman [6], 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 the hash tables do not 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. Even discounting the costs of collisions and rehashing, the calculation of a single hash operation, say $((ax + b) \bmod N) \bmod m$ for input x and constants a, b belonging to $\{0, ..., N\}$, is much greater that 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 [15]. The slow speed of SETL, observed in the SETL implemented ADA-ED compiler, has been attributed to an overuse of hashing [23]. 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 otherwise suffice.

In this paper we show that all of the hashed implementations of applications mentioned above can be replaced by algorithms with matching or superior worst case performance. This is achieved by 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. Multiset discrimination is discussed for various types of elements including pointers, strings, numeric constants, subtrees, and dags. In this paper it is adapted to solve the following problems.

(i) Array or list-based llllll tables can be formed during lexical scanning with unit-time curser or pointer storage/access.

(ii) Many grammar transformations can be implemented efficiently using multiset string discrimination. In this paper we exhibit a new linear time left factoring transformation. Simpler forms of this 'heuristic' transformation were previously studied by Stearns [22] and others (see also [2, 14]) to turn non-LL context free grammars into LL grammars.

(iii) An expression tree-to-dag transformation is implemented without hashing in a simpler way than before and in linear time and space. The numerous applications include one in which any linear pattern matching algorithm (e.g., [11]) can be turned into an efficient nonlinear matching algorithm, where each equality check takes unit-time.

(iv) A new hash-free basic block optimization by value numbering [2, 8] is given, which leads to a faster solution to the program equivalence problem used in integration by Yang, Horwitz, and Reps [27, 28].

(v) Although the main parts of algorithms for global constant propagation [26], global common subexpression detection [4], and code motion [9] 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.

(vi) We regard the strength reduction transformation presented by Cocke and Kennedy [7] to be the most practical reduction in strength algorithm published in the literature. Although the transformation due to Allen, Cocke, and Kennedy [3] is more powerful (since it can reduce multivariate products) and analyzes control flow more deeply, this algorithm can degrade performance by introducing far too

many sums in order to remove nests of products (as was shown in [16]). Although Knoop and Steffen's approach [13] is more general, it is also more expensive to implement.

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[7]. In particular, 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.

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:

(i) set union, difference, and intersection;

(ii) multiset string discrimination; i.e., finding all duplicates in a multiset of strings;

(iii) computing $\{[j,c]: [i,c] \in S, j \in T_i\}$

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

In [17] a different more general discussion of principles underlying hash-free algorithms for simple set operations is presented. Below we discuss a few sharper techniques with a focus on multiset discrimination. Unless otherwise stated, throughout this paper we will assume that sets and multisets are implemented as linked lists.

2.1. Multiset Discrimination of Pointers

We use the following notation for pointer manipulation. If variable x contains a pointer to variable y, expression *deref* (x) retrieves the value stored in y, and *ref* (x) is a pointer to the value stored in x.

Consider a multiset M of pointers to elements in a set S. For each element (i.e., symbolic address) x in M, we want to compute the set f (*deref* (x)) of pointers to all elements in M with the same value as x. (Note here that *deref* (x) is the value of the element in S that x points to.) Assume that f (*deref* (x)) is ini-

tially empty. Multiset M can be partitioned into blocks of duplicates using the following simple procedure

```
F := {}                -- F will be the set underlying M
(for x ∈ M)   -- linear search through M
 if f (deref (x)) = nil then
   F := F ∪ {x}
 end if
 f (deref (x)) := f (deref (x)) ∪ {ref (x)}  --ref (x) is a pointer to element x contained in M
end
```

2.2. Multiset Discrimination of Strings

Solving multiset discrimination of strings is slightly more complicated. Let M be a multiset of n variable-length strings over a k-symbol alphabet $\Sigma = \{1, ..., k\}$. Assume for convenience that each string ends with a sentinel symbol 0.

Starting with an initial partition P containing only one block M, we can solve this problem by repeatedly splitting blocks of P until all the duplicates in M are found. For each string s we implement a *current position* where the *current symbol* for s is stored. Initially the current symbol for each string is the symbol in its first position. A block is a set of pointers to strings in M. Once we know that a block contains all duplicate strings we say that the block is *finished*; otherwise, we say the block is *unfinished*. Partition P contains two parts - a set of unfinished blocks initially containing the single block M, and a set of finished blocks initially empty.

A partition refinement of P can be implemented using a primitive operation *split*(B) that replaces block B by new blocks, each containing pointers to strings of B with the same current symbol. The technique implements a variant of multiset discrimination of pointers that makes use of an array of $k + 1$ buckets, where the i-th bucket contains the new block with current symbol i. Any new block found in the 0 bucket or containing only one pointer to a string is finished; otherwise it is unfinished. During execution of *split* (B), the current position is incremented in each string belonging to a new unfinished block. It is easy to implement *split* (B) in time $O(|B|)$ and space $O(|B| + k)$.

We can also implement *split* (B) exclusively with lists and list processing. For each symbol $i = 0, 1, ..., k$ form a special list called the i-list with no elements. Strings are represented as lists of pointers to i-lists. Buckets can then be formed using these i-lists instead of arrays, and multiset discrimination can be

solved for pointers as in the preceding subsection.

The following algorithm makes use of $split(B)$ to solve multiset discrimination of strings:

1. Form the initial partition $P = \{M\}$.

2. Repeat Step 3 until all of the blocks in P are finished.

3. Scan the set of unfinished blocks, and replace each such block B in P by $split(B)$.

The algorithm runs in $O(m')$ time and $O(n+k)$ space, where m' is the total length of the prefixes needed to distinguish the strings in M. Both the theoretical time bounds and the simplicity of the implementation make it superior to lexicographic sorting for solving multiset discrimination.

Previously, the lexicographic sorting algorithm found in Aho, Hopcroft, and Ullman's book [1] was used to solve congruence closure [10] and also tree isomorphism [1]. Both these problems can be solved more simply by our solutions to multiset discrimination of strings. 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 strings in M. Their algorithm also has the practical disadvantage of a complex multi-pass implementation.

The array-based version of multiset discrimination of strings was used earlier by Paige and Tarjan to obtain improved solutions to lexicographic sorting [18] and DFA minimization for one symbol alphabets [19]. The implementation that uses pure list processing without arrays has the advantage of easier memory management. Both implementations are simple, and involve one pass through the prefixes of the strings. Consequently, our proposed applications may be practical.

2.3. 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. By treating character strings as bit strings, we can also vary k to obtain space/time tradeoffs in solving string discrimination (see [18]).

3. Applications

3.1. mmmbol Tables

Multiset discrimination of strings can be used directly to implement a two-pass lexical scanner. First the string 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. The performance is linear time and space in the length of the input string. Consequently, parsing and semantic analysis can proceed without hashing, since these processes can store and access the symbol table using pointers. 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.

A symbol table is also used in macro expansion. For example, the hygienic macro expansion algorithm reported in [CR90] frequently performs the the following operations: (1) replace all occurrences of a bound variable in its binding scope by a fresh identifier; (2) paint an expansion, i.e., replace each newly introduced identifier d in an expansion by a fresh identifier sharing the same symbol table entry with d; (3) store a macro definition together with the defining 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 defining environment instead of the using environment.) Although [CR90] assumes $O(1)$ time on each environment operation, straightforward implementation based on their definition would require linear search of 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 x is represented by a pointer to $table(x)$, which is implemented as a stack of pointers to records. In our approach, string substitution is very simple: to replace x by x' in its binding scope, we just push a pointer to a new record representing x' into $table(x)$, 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 paint an expansion, we replace each newly introduced identifier d by a pointer d' to a fresh symbol table entry, and push the top of $table(d)$ into $table(d')$. Finally, to achieve reference transparency, we simply paint each macro definition before storing it in the symbol table.

3.2. Fast Left Factoring

Left factoring is a context free grammar transformation investigated by Stearns [22] and others [2, 14] 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 multiset string discrimination.

Before presenting the factoring algorithm, it is useful to present some terminology and notation for strings. The empty string is denoted by λ. If s is a string, then s_i denotes the i-th symbol of s; the term $s_{i..j}$ is used to denote the substring of s from the i-th to the j-th symbol; $s_{i..}$ is an abbreviation for $s_{i..|s|}$. If $i>j$, then $s_{i..j}$ equals the empty string. Recall from previous discussion that if W is a set of strings over alphabet Σ, then for current symbol in the first position, $split(W) = \{\{x \in W \mid x_1 = a\} : a \in \Sigma \mid (\exists x \in W \mid x_1 = a)\}$.

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

$lcp(W)$ = **if** |W| = 1 **then** (let $W = \{x\}$)
  x
elseif $|split(W)| > 1$ **then**
  λ
else (s is the first symbol of every string in W)
  s $lcp(\{x_{2..} : x \in W\})$
end if

We also need terminology and notation for context free grammars. Let G be a context free grammar in which every nonterminal can derive a nonempty sentence.

Definition (see [2]): 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 $\alpha \Rightarrow^*$ t x and $\beta \Rightarrow^*$ t y;

(2) $\alpha \Rightarrow^* \lambda$ and $\beta \Rightarrow^* \lambda$;

(3) $\alpha \Rightarrow^* \lambda$, and there exists a terminal symbol t such that $\beta \Rightarrow^*$ t x and start symbol $S \Rightarrow^* w$ A t y.

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 a pair of productions $n \rightarrow \alpha\ \beta \mid \alpha\ \tau$ such that *lcp* $(\{\alpha\beta, \alpha\tau\}) = \alpha$ and $n \rightarrow \beta \mid \tau$ 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 (*Left Factoring Transformation*): Let $R \subseteq G$ be a set of productions $A \rightarrow \alpha\ \beta_i$, $i=1..n$ with $n > 1$ and $\alpha \neq \lambda$. If C is a new nonterminal not in *G*, then $LF(G,R,\alpha,C)$ is the set of productions that results from replacing *R* within *G* by the productions $A \rightarrow \alpha\ C$ and $C \rightarrow \beta_i$, $i=1,..,n$.

LEMMA 1. *Grammar LF(G,R, τ,C) is equivalent to grammar G.*

Proof Left factoring preserves the set of sentences derivable from each nonterminal. ■

LEMMA 2. *Nonfactorable conflicts cannot be eliminated by Left Factoring.*

Proof Let $n \rightarrow \alpha \mid \beta$ be a nonfactorable conflict in grammar *G*, and consider $G' = LF(G,R,\tau,C)$. If *R* does not contain either $n \rightarrow \alpha$ or $n \rightarrow \beta$, then by Lemma 1 the nonfactorable conflict remains in *G'*. If *R* contains both of these productions, then $\alpha = \tau\ \alpha_1$, $\beta = \tau\ \beta_1$, and *G'* must contain the nonfactorable conflict $C \rightarrow \alpha_1 \mid \beta_1$. Finally, if $n \rightarrow \alpha$ belongs to *R* but $n \rightarrow \beta$ doesn't, then *G'* contains the nonfactorable conflict $n \rightarrow \tau\ C \mid \beta$. ■

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

LEMMA 3. *If G is factorable, then LF(G,R,α,C) is factorable iff LF(G,R, α,C) is safe.*

Proof Suppose $G' = LF(G,R,\tau,C)$ is not safe. Then there is a production $n \rightarrow \gamma\ \alpha$ in *R* and a production $n \rightarrow \gamma\ \beta$ not in *R* in which the *lcp* $(\gamma\alpha,\gamma\beta) = \gamma$ is not a prefix of τ. Hence, τ must be a proper prefix of γ; i.e, $\gamma = \tau\ \rho$. In this case, *G'* contains nonfactorable conflict $n \rightarrow \tau\ C \mid \tau\ \rho\ \beta$.

Next, suppose that $G' = LF(G,R,\tau,C)$ 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) ($n \rightarrow \alpha \mid \beta$, 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 $n \rightarrow \tau\ \alpha \mid \tau\ \beta$ of *G* contained in *R*. Hence, $C \rightarrow \alpha \mid \beta$ is a factorable LL(1) conflict of *G'*.

(3) ($n \to \tau\ C \mid \rho\ \beta$, with only one occurrence of C) G must contain an LL(1) conflict formed from production $n \to \rho\ \alpha$, which belongs to R, and production $n \to \rho\ \beta$, which does not, with $\rho = lcp(\rho\alpha, \rho\beta)$. Since $LF(G,R,\tau,C)$ is safe, then ρ must be a prefix of τ. Hence, $n \to \tau\ C \mid \rho\ \beta$ is factorable. ■

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

Proof 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. Suppose nonterminal n is the left-hand-side of every production in R. Since the number of LL(1) conflicts within R is $|R|(|R|-1)/2$, then the weight of $G' = LF(G,R,\tau,C)$ equals $w(G) - (|\tau|\,|R|(|R|-1)/2 + (|R|-1)\sum_{n \to \gamma \in G-R} |lcp(\tau,\gamma)|)$. 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) by Lemma 3. ■

Theorem 1 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 $[\alpha,S]$ is a *gap* for nonterminal n if $S = \{n \to x \in G \mid \alpha \text{ is a prefix of } x\}$, $lcp(\{x: n \to x \in S\}) = \alpha$, and $|S| > 1$. A *gap transformation* is a left factoring transformation $LF(G,R,\alpha,C)$ in which $[\alpha,R]$ is a gap for G.

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

LEMMA 4. *A gap transformation is safe. If $[\alpha,A]$ and $[\beta,B]$ are two gaps for nonterminal n, then α is a prefix of β iff $B \subseteq A$. Neither α nor β is a prefix of the other iff B and A are disjoint.*

THEOREM 2. *If G is factorable, then the minimal number of applications of safe left factoring 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 for nonterminal n. We show that the number of gaps in G' is either the same as G or one less than G.

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

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

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

G' has one fewer gap than G iff $[\tau,R]$ is a gap for G. The result follows. ■

By Theorem 2, an optimal left factoring strategy must iterate gap transformations $G := LF(G,R,\tau,C)$ until grammar G is free of gaps.

THEOREM 3. *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 Let $[\alpha_1,R_1]$ and $[\alpha_1\alpha_2,R_2]$ be two gaps in grammar G. Grammar $G_1 = LF(G,R_1,\alpha_1,C)$ replaces all productions $n \rightarrow \alpha_1\beta_1,...,n \rightarrow \alpha_2\beta_k$ in R_1 by productions $n \rightarrow \alpha 1C$ and $C \rightarrow \beta_1,...,C \rightarrow \beta_k$. The productions of grammar G_1 have $|\alpha_1|(|R_1|-1)-1$ fewer occurrences of grammar symbols than the productinos 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 $R_3=\{C \rightarrow \beta_i : i=1,..,k \mid \alpha_2$ *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.

■

By Lemma 4, gaps can be partially ordered. We will compute these gaps and apply gap transformations efficiently by always choosing innermost gaps; i.e., gaps $[\tau,S]$ such that no other gap $[\beta,T]$ has $S \subseteq T$. We assume that gaps for productions with different left-hand-side nonterminals are incomparable.

Suppose that grammar G is factorable. For each nonterminal n in G, factor the set W of productions with left-hand-side n according to the following abstract procedure.

1. If $|W| = 1$, then it contains no gaps, so that W is already factored.

2. Otherwise, let R be the set of right-hand-sides of productions in W, and let $C = lcp(R)$.

3. If $C \neq \lambda$, then $[C,R]$ is a gap. If m is a new nonterminal symbol not in G, perform the gap transforma-

tion $G := LF(G,W,C,m)$, and recursively factor the set of productions of G with left-hand-side m.

4. Otherwise, for each block $b \in split(R)$, recursively factor the set of productions of G with left-hand-side n and right-hand-sides in b.

An efficient implementation proceeds by repeatedly partitioning the set of grammar productions starting with an initial partition P in which every set of productions with the same left-hand-side nonterminal forms a block. The data structure for P makes use of a set M of pointers to all the right-hand-sides of productions in the grammar. This set can be obtained by multiset discrimination of all the right-hand-side strings. Each block is represented by a subset $B \subseteq M$, a nonterminal symbol A, and an interval $[i, j]$, where the substring from the i-th to j-th symbol of every string in B must be the same. Initially, for each nonterminal A, P has a block containing the set of pointers to all right-hand-sides rhs such that $A \rightarrow rhs$ is a production. A is the grammar symbol for this block, and [1,0] is the interval, which represents the empty string λ.

After initialization the algorithm computes the new grammar G as described below:

```
G := {}
( while P ≠ {})
  remove block [B, [i,j], A] from P
  (case 1: B contains only one string x)
          G := G ∪ {A → x_{i.. |x|−1}}
  end case
--
-- find a left factor for strings in block B starting from the ith position
--
  j := j +1
  (case 2: split(B) contains only one block)
--
--  part of a nonempty left factor is found for strings in B
--
-- rhs_{i..j} is part of the left factor yet to be found
--
          add [B, [i,j], A] to P
     end if
  end case 2
  (case 3: split (B) contains more than 1 block)
--
-- complete left factor is found for strings in B
--
    if i = j then
```

--
-- left factor is the empty string, and B must be an initial block
--
    $C := A$ -- C represents nonterminal A
  else
--
-- nonempty left factor is $rhs_{i..j-1}$
--
    create new nonterminal C
    $G := G \cup \{A \rightarrow rhs_{i..j-1}\ C\}$
  end if
  (**for** each new block D that results from $split(B)$)
--
-- D contains just a single string, which is the trivial factor $rhs_{j..}$
--
    if $|D| = 1$ **then** $G := G \cup \{C \rightarrow rhs_{j...}\}$
--
-- try to find the left factor for strings in D
--
    elseif $|D| > 1$, **then** add $[D,[j,j],C]$ to P
  end if
 end for
end case 3
end while

The preceding discussion leads to the following theorem.

THEOREM 4. *The preceding algorithm is correct and runs in linear time in the number of the grammar symbol occurrences in the productions of the input grammar.*

3.3. Multiset Discrimination of Trees and Applications

Suppose we have a forest of syntax trees produced by syntactic analysis. Suppose also that the nodes of the syntax tree contain pointers to symbol table entries for function symbols, constants, and variables. There are various applications in which we want to find duplicate subtrees. Multiset subtree discrimination can be solved in a new way without hashing by combining multiset string discrimination with multiset pointer discrimination.

Let T be a forest of n nodes. We identify each subtree rooted in node j by a string of length 1 + the number of children of j and with symbols ranging over the alphabet $\{1, ..., n\}$. First we solve multiset pointer discrimination on the symbol table pointers in all the nodes of T. Next, we assign successive

integers, called *local* numbers, starting with 1 to the distinct pointers of *T*. The local number at each node *j* will be the initial symbol of the string that identifies *j*.

To obtain the remaining symbols of the subtree identifier, we exploit the idea that subtrees at different heights must be distinct. This allows us to solve multiset subtree discrimination separately for all nodes of the same height bottom-up starting from the leaves to the tree height *d*. That is,

(1) Solve multiset string discrimination for the leaves, and identify each distinct local number with new numbers, called *value numbers*, with successive values starting with 1.

(2) For height $i = 2, 3, ..., d$, repeat steps 3 and 4:

(3) Identify each node *j* at height *i* with a string formed from the local number of *j* followed by the value numbers of the children of *j*.

(4) Solve multiset string discrimination on the strings described in step 3. This solves the multiset subtree discrimination problem at height *i*. Then identify each distinct subtree at height i with new successive value numbers, starting from the last value number assigned to a subtree at height *i*-1.

The preceding algorithm requires $O(n)$ time and space and is a great deal simpler than the previous best algorithm based on lexicographic sorting. It can be used to obtain new hash-free solutions to many applications including tree-to-dag compression, turning an arbitrary linear tree pattern matching algorithm [11] into a nonlinear matching algorithm [21], deciding structural equivalence of type denotations [2], and preprocessing input in the form required by Wegman and Paterson's unification algorithm [20].

3.4. Multiset Dag Discrimination and Acyclic Coarsest Partitioning

The solution to multiset tree discrimination extends without modification to solve multiset discrimination for dags with *m* edges and *n* nodes in time $O(m)$ and space $O(n)$. Recall that 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 [1]. We 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 [12], has applications in program optimization and program integration. It can be formulated

as follows. Given a directed multi-graph $(V, E_1, ..., E_k)$ (where V is the set of vertices, and $E_1, ..., E_k$ are sets of edges), and an initial partition $P = \{V_1, ..., V_s\}$ of V, find a coarsest refinement P' of P such that for each block C in P' and each $i = 1, ..., k$, there exists a block C_0 in P' such that the image set $E_i[C] \subseteq C_0$, where $E_i[C] = \{y : [x, y] \in E_i \text{ and } x \in C\}$. Here we assume that for each $i = 1, ..., k$, the outdegree of each vertex $v \in V$ in (V, E_i) is at most 1.

An algorithm was given in[12] that solves this problem in time $\Theta(k|V|\log|V|)$ and space $\Theta(k|V|)$ in the worst case, which is true even when the graph $(V, E_1 \cup ... \cup E_k)$ is acyclic. However, when the graph $(V, E_1 \cup ... \cup E_k)$ is acyclic, we can solve the problem in time and space $O(k|V|)$ using a solution to multiset discrimination for dags.

THEOREM 5. *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 $\Theta(k|V|\log|V|)$ and space $\Theta(k|V|)$ in the worst case, and by multiset dag discrimination in time $O(k|V|)$ and space $O(|V|)$.*

3.5. The Sequence Congruence Problem

The sequence congruence problem [27, 28] arises in the context of program integration. It asks how to partition program components into classes whose members have equivalent execution behaviors. The algorithm presented in [27, 28] solves this problem in two phases: the program components are first partitioned with respect to the flow dependence graph, and then refined with respect to the 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. Since 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 Analysis by Value Numbering

Value numbering is a standard program analysis technique of determining equalities of the values computed by instructions within basic blocks [2, 8]. Although the technique is mostly implemented with hashing, multiset discrimination can be used to obtain a more efficient implementation.

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, *rhs* is either a constant, a variable, or an expression of the form $op(x_1, ..., x_t)$ in which *op* is some t-nary 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 assign an integer (i.e. a value number) to each *lhs* and *rhs* so that if two occurrences of expressions have the same value number, then they have the same run-time value. We compute value numbers in three steps as follows:

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 represents 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 $[v, v_i]$ 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 $[v, node(x_i)]$ 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 on all the constant entries, and then adjust the pointers to these entries accordingly.

3. To recognize common subexpressions, we dagify D using the method described in the previous sections, which gives a value number to each node in D.

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 seems to stem 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 [7]. The algorithm runs in worst case time/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. Like their algorithm we are careful not to compute the potentially costly data flow relation.

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 [3], the earlier transformation [7] 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 [7] may be defined as follows. Let L be a strongly connected region of code. We assume that this code consists of assignments to simple 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 a new value. All concern for control flow is simplified by taking a most conservative position that L forms a clique; i.e., that every two statements in L can be executed one after the other.

If c is either a region constant variable of L or a constant, and if i is a variable that is defined in L, then product $i \times c$ is *reducible* if 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 $i := -j - k$, where in each such form $j \times c$ and $k \times c$ must also be reducible. For each reducible product $i \times c$ occurring in L, strength reduction transforms L as follows:

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

(ii) 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.

(iii) Within L and just prior to each definition to i of the forms either $i := \pm j$ or $i := \pm j \pm k$, insert the code $t_{ic} := \pm j \times c$ or $t_{ic} := \pm j \times c \pm k \times c$ respectively.

(iv) If any of the products introduced in step (iii) 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 (iii) 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 solution shares the first four steps of the Cocke and Kennedy algorithm; i.e.,

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

(ii) Compute the set IV of *induction variables*; that is, the set of all variables x with definitions occurring in L such that any product $x \times c$ would be reducible. This procedure was also described by Cocke and Schwartz [8].

(iii) Find the set $Cands$ of all reducible products $x \times c$ actually appearing in L, and the associated places where they occur.

(iv) 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 in $L\}$.

The preceding steps can be performed in worst case time and space linear in the program text. 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 5. *The set of all expressions removed from L by strength reduction is defined by* $Rm = \{j \times c: i \times c \in Cands, j \in Afct^*(i)\}$.

Calculation of *Rm* is central to the implementation of strength reduction, and it is important to observe three sources of redundancy in computing this set naively.

(i) when $i \times c$ and $j \times c$ belong to *Cands* and $Afct^*(i) \cap Afct^*(j)$ is nonempty;

(ii) when $i \times c_1$ and $j \times c_2$ belong to *Cands*, $c_1 \in Afct^*(j)$, and $c_2 \in Afct^*(i)$;

(iii) when two different products of constants evaluate to the same constant

Because only the first source of redundancy can lead to an asymptotic blowup in time and space, we avoid it during the calculation of *Rm*. Because the other two sources of redundancy only contribute constant factors in complexity, we avoid them during a postpass cleanup. Our approach combines multiset discrimination with 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[25] (see also[1]), where n is the number of variables and constants contained in *Afct*, and m is the number of assignments to induction variables. They also use a greedy strategy committed to hashing each product removed by strength reduction. In contrast, we compute the strong component decomposition of *Afct* inverse (i.e., we consider decomposition of a graph with directed edge $i \rightarrow j$ iff $i \in Afct(j)$) in $\Theta(m)$ time and space using Tarjan's algorithm [24]. The dag structure *Scd* of this decomposition is used to efficiently compute *Rm* in time *O*(final text length). The algorithm rests on the following obvious fact:

LEMMA 6. *Let* $Cs = \{c: i \times c \in Cands\}$. *For each* $c \in Cs$ *let* $Cmps(c)$ *be the set of strong components containing some variable i for which* $i \times c \in Cands$. *If c is any region constant variable or constant, then the set of all expressions* $j \times c$ *removed by strength reduction is defined by* $Rm(c) = \{j \times c:$ *j belongs to a component of Scd from which there is a path in Scd to any component of* $Cmps(c)\}$.

The remaining steps of the algorithm are given just below:

(v) Compute the set *Cs* using multiset pointer discrimination. At the same time, for each constant $c \in Cs$, form a set of pointers to strong components $Cmps(c)$ as described in Lemma 6, and mark variables v within these components such that $v \times c$ belongs to *Cands*.

(vi) Initialize an empty multiset *Mrc* of subtrees and an empty multiset *Mc* of numeric constants. For

each constant $c \in Cs$ repeat steps (vii) and (viii)

(vii) 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 $Cmps(c)$. Observe that for each strong component of *Scd*, if it has no edges leading in, then its entries are constants or region constant variables; otherwise, its entries are induction variables. Link each induction variable $v \in Scd_c$ to a new symbol table entry containing unique identifier t_{vc}, and insert assignment $t_{vc} := v \times c$ on entry to *L* if *v* is live 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}. Link each region constant variable $v \in Scd_c$ to a new entry in *Mrc* containing subtree $v \times c$. For each constant $c' \in Scd_c$, if *c* is a region constant variable, then link c' to a new entry in *Mrc* containing subtree $c \times c'$; otherwise, link c' to a new entry in *Mc* containing the computed value of $c \times c'$.

(viii) For each induction variable *v* in Scd_c and each assignment to *v* in $Defs(v)$, introduce update code to t_{vc} according to the definition of the strength reduction transformation described earlier. 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.

(ix) Use multiset subtree discrimination and constant discrimination to find duplicate region constant expressions and constants in *Mrc* and *Mc*, and augment the symbol table with new variables for each distinct item in *Mrc* and *Mc*. At the same time readjust pointers inside *L* to the symbol table, and insert an assignment $t_{c_1 c_2} := c_1 \times c_2$ on entry to *L* for each product $c_1 \times c_1 \in Mrc$.

THEOREM 6. *The preceding algorithm is correct and has worst case time and space O(length of the final program text).*

4.2. Strength Reduction With Cleanup

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 amongst 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. Let *Afcti* denote *Afct* inverse. As before, we compute the strong component decomposition dag *Scd* of *Afcti*.

Recall that those single node components of *Scd* with no edges leading in contain only constants and region constant variables. Also, any product $x \times c \in Prods$ is reducible (i.e., belongs to *Cands*) iff there is no path in *Scd* 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 *Scd* 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 with edges leading in.

Consequently, we can proceed to solve strength reduction starting with step (v) of the previous algorithm. We now have an alternative linear time strength reduction algorithm, where the first four steps of Cocke and Kennedy's solution are simplified. This new algorithm can also be extended to support efficient analysis for useless code.

Consider how the new strong component dag Scd_{new} of the program loop L after strength reduction differs from the initial dag Scd_{old}.

LEMMA 7.

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

ii. The only new components in Scd_{new} are ones containing only new temporaries; the only edges incident to these components are between them 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 ways:

i Assignments are introduced within L to modify compiler-generated temporaries t_{xc}. The right-hand-side of any such assignment must contain only compiler-generated temporaries. Hence, these assignments cannot create new edges from Scd_{old} to any strong components in Scd_{new} containing compiler-generated variables.

ii 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 that belongs to a marked component $Scd_z \in Scd_{old}$, and x must appear in an unmarked component $Scd_x \in Scd_{old}$. Moreover, there must be an edge from Scd_x to Scd_z. After replacement, there would be an edge from the strong component in Scd_{new} containing t_{xc} to Scd_z. If the edge count between Scd_x and Scd_z after replacement becomes zero, indicating no assignments in L from a right-hand-side variable in Scd_x to a variable in Scd_z, then this edge is deleted in Scd_{new}. ■

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 *Scd* containing them, which we call the critical set *crit*, are also useful. The useful components include *crit* and all strong components of *Scd* 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 constants, and constants can become useless. Temporaries generated 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 steps (vii)-(ix) of the algorithm in the previous subsection to facilitate useless code elimination as follows. In step (vii), for each assignment $z:=v\times c$ replaced by assignment $z:=t_{vc}$, decrement the edge count from Scd_v to Scd_z. If the edge count reaches zero, then delete the edge from Scd_v to Scd_z. This is implemented using a pointer linking a record for assignment $z:=v\times c$ into the adjacency list for *Scd*. Also, add edges from t_{vc} to z in *Afcti*. In step (viii) introduce a new edge in *Afcti* for

each assignment to a temporary introduced. In step (ix) multiset discrimination will determine the new vertices corresponding to new temporaries in *Afcti*. Add a final step (x) in which the useful components of *Scd* are computed. Within L all assignments to variables not in useful components can be removed.

By the preceding discussion we have

THEOREM 7. *The preceding algorithm is correct and has worst case time and space O(length of the initial plus final program text).*

4.3. Iterated Strength Reduction

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 [7]. 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 the iterated strength reduction. Let *Cands** be the set of products in L reduced by the iterated strength reduction. Following Cocke and Schwartz [8], 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 string $c_1 \ldots c_j$ is a *tail* of v. The set of tails of a variable or constant v is denoted by $tails(v)$. By default, $t_v = v$. Thus, $\lambda \in tails(v)$ iff v is not useless in L^*, where λ denotes the empty string. 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 label the edges in *Afcti* as follows. For each instruction $v := \pm j$, $v := \pm j \pm k$ and $v := \pm k \pm j$, we add λ to $labels(j,v)$. For each instruction $v := j \times c$, where $c \in RC$, we add c to $labels(j,v)$. For each instruction $v := \ldots j \ldots$ not mentioned above, we mark v as a *spoiler* and add λ to

labels(j,v). We further extend the definition of labels to edges in *Scd*. For each edge $[C_1, C_2]$ in *Scd*, we define $Labels(C_1, C_2) = \{c: c \in labels(x, y), x \in C_1, y \in C_2\}$.

We say that a component $C \in Scd$ is *clean* if none of its elements are spoilers and for all edges $[x, y]$ in C, $labels(x, y) = \{\lambda\}$. We say that C is *reducible* if all its ancestors in *Scd* are clean. It is not difficult to see that a variable v occurring in L becomes an inductive variable in some round 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 reducible components in a single topological search through *Scd* in the direction of its edges: for each component C, if it is clean and if its predecessors are all reducible, then it is reducible; otherwise it is not reducible. It follows that,

LEMMA 8.

i. The set Cands consists of all those products $v \times c$ occurring in L such that $c \in RC$ and v belongs to a reducible component.*

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

iii. If $[v, x]$ is an edge in Afcti, $c \in labels(v,x)$, $p \in tails(x)$, and v belongs to a reducible component, then the string $cp \in tails(v)$.

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

Let C be a component in *Scd*, $x, y \in C$. If C is reducible, then $tails(x) = tails(y)$ by Lemma 8.iii. If C is not reducible, then $tails(x) = tails(y) = \{\lambda\}$ by Lemma 8.iv. In either case, we define $Tails(C) = tails(x)$. 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
    if C is not reducible **then**
        $Tails(C) := \{\lambda\}$;
1   **else** $Tails(C) := \bigcup_{C_i \in succ(C)} \{ xs: x \in Labels(C, C_i), s \in Tails(C_i) \}$;
        if C contains output variables **then**

$Tails(C) := Tails(C) \cup \{\lambda\}$;
end if;
end if;
end for;

where $succ(C_i)$ is the set of successors of C_i in *Scd*. Although multiset string discrimination could be used in computing the union in line 1, the $\Omega(|s|)$ worst case cost contributed by each string 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. Initially, $Tails(C)$ is empty for all components $C \in Scd$. In round $i = 0, 1, ..., k$, 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 $i = 1, ..., k$ we assign a unique identifier for each distinct tail of length i. Thus, in round $i+1$, each newly generated string $c_1c_2...c_{i+1}$ can be represented by a pair $[c_1, n_1]$, where n_1 is the name for $c_2...c_{i+1}$. Consequently, in order to determine distinct tails generated in each ith round, where $i > 1$, multiset string discrimination is only needed for strings of length 2. Following are the implementation details.

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

(1) generate tails of length i;
propagate tails of length i using Scd_1;

Tails of length 0 are generated according to Lemma 8.iv:

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

Tails of length 1, ..., k are generated according to Lemma 8.iii:

(3) **for** C_1 in $Heads(i-1)$
for C in $pred_2(C_1)$
$Tails(C, i) := \{\}$;

Heads(i) **with**:= C;
end for;
end for;
for C_1 in *Heads*$(i-1)$
for C in $pred_2(C)$
2 *Tails*(C,i) := *Tails*$(C,i) \cup \{ cs: c \in Labels(C,C_1) - \{\lambda\}, s \in Tails(C_1,i-1) \}$;
end for;
end for;
perform a multiset discrimination on $\bigcup_{C \in Heads(i)} Tails(C,i)$;

For $i = 0, 1, ..., k$, tails of length i are propagated in *Scd* according to Lemma 8.ii:

(4) *Heads*(i) := $\{C \in Scd \mid C$ 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
perform a multiset discrimination on *Tails*(C_1,i);
for C in $pred_1(C_1)$
3 *Tails*(C,i) := *Tails*$(C,i) \cup$ *Tails*(C_1,i);
end for;
end for;

The above representation of strings 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: replacing products in *Cands** by temporaries, inserting code to keep temporaries available, and 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 see the complexity of our algorithm, we note that for each tail *cs* 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 $p \neq \lambda$ 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 8. *The iterated strength reduction problem with useless code elimination can be solved in time and auxiliary space linear in the maximum length of the initial and final program texts.*

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 become dead after $\Omega(|L|)$ iterations. Because of Lemma 7, each inductive variable or region constant in L_i can be in the set $Afct^*(x)$ for at most $|L|$ inductive 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 $Afct^*(x)$ need 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[18] 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 suggested hash-free methods for solving various aspects of optimizing compilation. These methods have been based in large part on efficient algorithms for solving multiset discrimination for different datatypes. Multiset discrimination of ordered flow graphs, unordered trees and dags, and unordered graphs with respect to given depth-first-search spanning trees are straightforward. An empirical investiga-

tion comparing our hash-free alternatives with their conventional hash-based counterparts would be worthwhile future work.

6. Acknowledgements

We are grateful to Alan Siegel, whose independent investigation of lexicographic sorting and great interest in its applications to algorithm design provided motivation for our work. We thank Bob Tarjan for describing a list based data structure known to Knuth for implementing fast string matching, which is related to our list based implementation of multiset discrimination. We also 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. Aho, A., Hopcroft, J., and Ullman, J., *Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

2. Aho, A., Sethi, R. and Ullman, J., *Compilers*, Addison-Wesley, 1986.

3. 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.

4. Alpern, B., Wegman, N., and Zadeck, K., ‘‘Detecting Equality of Variables in Programs,’’ in *Proc. 15th ACM POPL*, Jan, 1988.

5. Cai, J. and Paige, R., ‘‘Look Ma, No Hashing, And No Arrays Neither",’’ in *ACM POPL*, pp. 143 - 154, Jan, 1991.

6. Carter, J. and Wegman, M., ‘‘Universal Classes of Hash Functions,’’ *JCSS*, vol. 18, no. 2, pp. 143-154, 1979.

7. Cocke, J. and Kennedy, K., ‘‘An Algorithm for Reduction of Operator Strength,’’ *CACM*, vol. 20, no. 11, pp. 850-856, Nov., 1977.

8. Cocke, J. and Schwartz, J. T., *Programming Languages and Their Compilers*, Lecture Notes, CIMS, New York University, 1969.

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

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

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

12. 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.

13. Knoop, J. and Steffen, B., *Strength Reduction based on Code Motion*, Bericht Nr.9103, Institute Fur Informatik und Praktische Mathematik, Christian-Albrechts-University Kiel, Germany, Feb, 1991.

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

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

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

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

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

19. 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.

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

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

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

23. 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.

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

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

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

27. 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.

28. 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.