Efficient representations for triangular substitutions: A comparison in miniKanren

David C. Bender, Lindsey Kuper, William E. Byrd, and Daniel P. Friedman

Indiana University, Bloomington, IN 47405

Abstract. Unification, a fundamental process for logic programming systems, relies on the ability to efficiently look up values of variables in a substitution. Triangular substitutions, which allow associations to variables that are themselves bound by another association, are an attractive choice for purely functional implementations of logic programming systems because of their fast extension time and linear space requirement, but have the disadvantage of costly lookup. We present several representations for triangular substitutions that decrease the cost of lookup to linear or logarithmic time in the size of the substitution while maintaining most of their desirable properties. In particular, we show that triangular substitutions can be represented efficiently using skew binary random-access lists, and that this representation provides a significant decrease in running time for existing programs written in miniKanren, a declarative logic programming system implemented in a pure functional subset of Scheme.

1 Introduction

Baader and Snyder [1] present the concept of unification as a fundamental process for automated deduction systems, including logic programming systems. In the model of logic programming considered here, the unification of two terms with respect to a substitution s is the process of extending s with bindings for the terms' variables until the terms equate in s (or determining that no satisfactory extension is possible). A substitution maps variables to terms and can be represented simply using an association list.

Substitutions in *triangular* form [1] allow associations of variables to variables that themselves may occur in the domain of the substitution. In a triangular substitution the result of a variable lookup must itself be looked up recursively; this process is known as *walking* the variable in s. For example, walking x in the substitution $((b\ .\ 2)\ (z\ .\ 3)\ (a\ .\ 1)\ (y\ .\ z)\ (x\ .\ y))$ returns 3 because x is bound to y, which is bound to z, which is bound to 3, a constant.

Walking a variable x in a triangular substitution s takes $O(n^2)$ time[1] in the length of s, since each of the n bindings may potentially bind a variable to

[1] This time bound assumes that *walk* terminates, which is guaranteed if the substitution is acyclic.

another variable which must be walked in s.[2] Given an association list representation of triangular substitutions and a *var?* predicate that checks its argument for membership in the set of variables, we can define the *walk* procedure in Scheme as follows:

```
(define walk
  (lambda (v ŝ)
    (let loop ((s ŝ))
      (cond
        ((or (not (var? v)) (null? s)) v)
        ((eq? v (lhs (car s))) (walk (rhs (car s)) ŝ))
        (else (loop (cdr s)))))))
```

If v is not a variable or does not appear as the left-hand side[3] of a binding in s, (*walk* v s) simply returns v. Otherwise, we examine the first binding in s and, if its left-hand side matches v, we recursively walk its right-hand side in s; if not, we continue comparing v with the left-hand sides of the remaining bindings in s until a match is found or s has been exhausted.

A substitution is *idempotent* if no variable in its domain occurs in any term in its range. *walk* is trivial to implement for idempotent substitutions, because once a match has been found, no further recursion is necessary; a *walk* of an idempotent substitution therefore also runs in $O(n)$ worst-case time[4] in the size of the substitution. The substitution in the example above is, of course, not idempotent because z and y appear on both the left-hand and right-hand sides of bindings. Using our association list representation, triangular substitutions are extendable in constant time with *cons*, but idempotent substitutions require a more costly extension operation in order to maintain idempotency.

In summary, the fundamental tradeoff between idempotent and triangular substitutions is in the choice of fast lookup and more costly extension versus fast extension and more costly lookup. However, in a purely functional context, triangular substitutions have a smaller total space requirement since each extension results in a new substitution comprising the new binding and a reference to the unmodified previous substitution. Purely functional idempotent substitutions have no such space bounds, as the addition of one binding may require us to allocate and retain space for modified versions of some (in the worst case, *all*) of the existing bindings. Triangular substitutions are therefore the focus of our attention in this paper. We demonstrate techniques for improving the efficiency of walking in triangular substitutions, first by optimizing *walk* itself, and second,

[2] The name "triangular" can therefore be thought of as referring to the "shape" of the recursion in the worst case, in which all n elements in s must be examined on the first call to *walk*, followed by $n-1$ elements on the first recursive call, and so on.

[3] We use *lhs* and *rhs* selectors for generality; these may be trivially implemented using the Scheme primitives *car* and *cdr*, respectively.

[4] This time bound assumes an association list representation of idempotent substitutions; we note that any non-trivial map data structure has a better worst-case bound.

by choosing representations for triangular substitutions that perform well when tested on real programs written in miniKanren, a declarative logic programming system implemented in a pure functional subset of Scheme [2].

The paper is organized as follows: We begin in section 2 by describing miniKanren's implementation of triangular substitutions and the optimizations it already incorporates. In section 3, we make several attempts to improve the performance of *walk* without altering the association list representation of substitutions. In section 4, we consider more efficient representations for triangular substitutions. In section 5, we benchmark the performance of all our representations on a suite of tests. Finally, in sections 6 and 7, we set out our ideas for future work and summarize our findings.

2 Substitutions in miniKanren

miniKanren uses triangular substitutions represented as association lists. Unification is implemented by miniKanren's *unify* procedure, shown below, which takes two terms, $\hat{v}$ and $\hat{w}$, and a substitution, s, and attempts to unify $\hat{v}$ and $\hat{w}$ with respect to s.

```
(define unify
  (lambda (v̂ ŵ s)
    (let ((v (walk v̂ s)) (w (walk ŵ s)))
      (cond
        ((eq? v w) s)
        ((var? v) (ext-s v w s))
        ((var? w) (ext-s w v s))
        ((and (pair? v) (pair? w))
         (let ((s (unify (car v) (car w) s)))
           (and s (unify (cdr v) (cdr w) s))))
        ((equal? v w) s)
        (else #f)))))
```

A call to *unify* proceeds as follows: If $\hat{v}$ and $\hat{w}$ are identical after being walked in s, the incoming terms unify with respect to s and we return s unmodified. Otherwise, if v (or w) is a variable, it was not bound in s and can be bound to w (v), and the resulting extended substitution is returned. If both v and w are unbound variables, *unify* binds v to w instead of w to v, which would be a valid alternative. However, if neither v nor w is a variable, *unify* is recursively called on their subterms. Otherwise, unification fails. (The *equal?* clause catches equal constants that are not identical in the sense of Scheme's *eq?*, which we use in the first clause for efficiency.)

Since the performance of *walk* is critical to *unify*, miniKanren uses two techniques to improve the average cost of *walk*: birth substitutions and right-hand-side check. We discuss these optimizations in the following sections.

2.1 Birth Substitutions

When a variable has no binding, the cost of walking it is linear in the size of the substitution, since every binding must be checked. miniKanren improves

this cost for many variables by exploiting the fact that when a new variable x is created, any bindings in the existing substitution s cannot possibly refer to x, because they were made before x was created. Therefore, when walking x at some point in the future, if x's "birth substitution" s is ever reached, no match for x exists and *walk* can terminate, returning x. In miniKanren, variables are represented as unique objects that contain a pointer to the variable's birth substitution, making it readily available to *walk*. Birth substitutions also allow *walk* to avoid checking explicitly for the empty substitution, since all variables have a birth substitution guaranteed to match either the current substitution or a suffix thereof.

2.2 Right-Hand-Side Check

When walking a variable v, if v is ever found on the right-hand side of a binding, *walk* can terminate, returning v. This "right-hand-side check" is possible because *unify* first *walk*s both $\hat{v}$ and $\hat{w}$. If the substitution is extended, either v will be bound to w or w to v. Consider the case where v is bound to w. If w is a variable, it cannot occur on the left-hand side of a binding deeper in the substitution, or the initial *walk* would have found a match. During some later walk of w, if the pair $(v\ .\ w)$ is reached, *walk* can immediately return w because no deeper binding for w can exist. (If w were bound to some x deeper in the substitution, then v would have been bound to x instead of to w.) The case in which w is bound to v is similar. Another benefit of right-hand-side check is that it guarantees termination of *walk* even for cyclic substitutions.

Because of the above optimizations, a recursive call to *walk* will never reach the end of the substitution because, at the very least, a right-hand-side check on the pair just matched will terminate the recursive *walk*. Using this knowledge, *walk* is split into the following two procedures, *walk-sref* and *step-sref*, which incorporate both birth substitutions and right-hand-side check.

```
(define walk-sref
  (lambda (v ŝ)
    (let loop ((s ŝ))
      (cond
        ((or (not (var? v))
             (null? s)
             (eq? s (var-birth v))
             (eq? v (rhs (car s))))
         v)
        ((eq? v (lhs (car s))) (step-sref (rhs (car s)) ŝ))
        (else (loop (cdr s)))))))

(define step-sref
  (lambda (v ŝ)
    (let loop ((s ŝ))
      (cond
        ((or (not (var? v)) (eq? v (rhs (car s)))) v)
        ((eq? v (lhs (car s))) (step-sref (rhs (car s)) ŝ))
        (else (loop (cdr s)))))))
```

walk-sref performs the search before the first match, while *step-sref* continues the search after the first match and omits the needless check for a birth substitution.

3 Potential Improvements to *walk*

Without altering the underlying representation of substitutions in miniKanren, several changes can be made to *walk* to attempt to improve its performance.

3.1 Forward-Backward *walk*s

The following variations on *walk* limit its worst-case time complexity to linear in the size of the substitution. By taking advantage of the right-hand-side check, we can see that if a variable x is found on the right-hand side of a binding b, it cannot occur on the left-hand side of a binding deeper in the substitution. Therefore, instead of recursively walking x from the front of the list, we begin searching *backwards* from b. If a second match exists, we are certain to find it before reaching the beginning of the list. We therefore traverse the entire substitution at most twice, for an overall time complexity of $O(n)$ in the size of the substitution. *walk-front-back*, shown below, uses stack unwinding as a mechanism for traversing the list backwards upon reaching p.

```
(define walk-front-back
  (lambda (v ŝ)
    (call/cc
      (lambda (k)
        (let loop ((s ŝ))
          (cond
            ((or (not (var? v))
                 (null? s)
                 (eq? s (var-birth v))
                 (eq? v (rhs (car s))))
             (k v))
            ((eq? v (lhs (car s))) (rhs (car s)))
            (else
             (let ((v (loop (cdr s))))
               (cond
                 ((not (var? v)) (k v))
                 ((eq? v (lhs (car s))) (rhs (car s)))
                 (else v)))))))))))
```

A continuation k is saved at the start of *walk-front-back* to permit immediate termination in the case of a successful right-hand-side or birth-substitution check, or if v is ever a non-variable. For each binding in the substitution, the recursive non-tail call (*loop* (*cdr* *s*)) will either invoke k, or will return without invoking k, indicating that we have reached the binding again during the backward traversal.

Another variation, *walk-reverse-list*, builds the reverse list using *cons* cells as it traverses forwards through the list searching for the first match. This eliminates the dependency on *call/cc* and avoids using stack unwinding to reverse the list.

```
(define walk-reverse-list
  (lambda (v ŝ)
    (let loop ((s ŝ) (s-rev '()))
      (cond
        ((or (not (var? v))
             (null? s)
             (eq? s (var-birth v))
             (eq? v (rhs (car s))))
         v)
        ((eq? v (lhs (car s)))
         (walk-reverse-list-back (rhs (car s)) s-rev))
        (else (loop (cdr s) (cons (car s) s-rev)))))))

(define walk-reverse-list-back
  (lambda (v s)
    (cond
      ((or (not (var? v)) (null? s)) v)
      ((eq? v (lhs (car s)))
       (walk-reverse-list-back (rhs (car s)) (cdr s)))
      (else (walk-reverse-list-back v (cdr s))))))
```

Finally, a further variation, *walk-pinch*, reaches the first match and then steps alternatively from the front of the list and the most recent match point, squeezing the search window until a second match is found. This algorithm performs better than *walk-reverse-list* when the second match is more likely to be closer to the front of the list than to the initial match.

```
(define walk-pinch
  (lambda (v ŝ)
    (let loop ((s ŝ) (s-rev '()))
      (cond
        ((or (not (var? v))
             (null? s)
             (eq? s (var-birth v))
             (eq? v (rhs (car s))))
         v)
        ((eq? v (lhs (car s))) (pinch (rhs (car s)) ŝ s-rev))
        (else (loop (cdr s) (cons (car s) s-rev)))))))

(define pinch-find
  (lambda (e s)
    (cond
      ((eq? e (car s)) (cdr s))
      (else (pinch-find e (cdr s))))))
```

```
(define pinch
  (lambda (v ŝ-fwd ŝ-rev)
    (let loop ((s-fwd ŝ-fwd) (s-rev ŝ-rev))
      (cond
        ((or (not (var? v))
             (null? s-rev)
             (eq? s-fwd (var-birth v))
             (eq? v (rhs (car s-fwd))))
         v)
        ((eq? v (lhs (car s-fwd)))
         (pinch (rhs (car s-fwd)) ŝ-fwd (pinch-find (car s-fwd) s-rev)))
        ((eq? v (lhs (car s-rev))) (pinch (rhs (car s-rev)) ŝ-fwd (cdr s-rev)))
        (else (loop (cdr s-fwd) (cdr s-rev)))))))
```

miniKanren's use of purely functional substitutions allows branching computations to share substitution suffixes of various lengths. Unfortunately, this precludes a doubly-linked list representation for substitutions in which the backwards list is built as a side effect of extension. When walking the substitution from front to back, we traverse a list (there is only one path); however, when walking from back to front, any particular branch of the computation has only one path, but the substitutions for all branches together constitute a tree.

3.2 Path Compression

Path compression [4,14] dynamically compresses chains of variables in a substitution, in effect bringing it closer to idempotency. Consider a hypothetical case: walking a variable v initially yields a binding to another variable w, which is then recursively walked to yield a value, say, 1. During the walk, we have gained a valuable piece of information: v and w are members of an equivalence class in that they ultimately walk to the same value. However, this information is not preserved; if we subsequently walk v again, we repeat the entire process. We can preserve the equivalence of v and w by extending the substitution with a new binding of $(v\ .\ 1)$, so that subsequent walks of v immediately return 1. This technique ensures that every variable in a path of equivalent variables is bound directly to the shared value. For instance, if w above walked to x instead of 1, pairs binding both v and w to the final returned value would be added to the substitution. A version of *walk* that trades longer substitutions for shorter paths, *walk-flatten*, is shown below. This version also incorporates the reverse list-building technique from *walk-reverse-list* in the previous section.

```
(define walk-flatten
  (lambda (v ŝ)
    (let loop ((s ŝ) (s-rev '()))
      (cond
        ((or (not (var? v))
             (null? s)
             (eq? s (var-birth v))
             (eq? v (rhs (car s))))
         (values v ŝ))
        ((eq? v (lhs (car s)))
         (walk-flatten-back (rhs (car s)) `(,v) ŝ s-rev))
        (else (loop (cdr s) (cons (car s) s-rev)))))))
```

```
(define walk-flatten-back
  (lambda (v m s-fwd s)
    (cond
      ((not (var? v)) (ret-flatten v m s-fwd))
      ((null? s) (values v s-fwd))
      ((eq? v (lhs (car s)))
       (walk-flatten-back (rhs (car s)) (cons v m) s-fwd (cdr s)))
      (else (walk-flatten-back v m s-fwd (cdr s))))))

(define ret-flatten
  (lambda (v m s)
    (let loop ((m m) (s s))
      (cond
        ((null? m) (values v s))
        (else (loop (cdr m) (cons `(,(car m) . ,v) s)))))))
```

At the point when *walk-flatten* normally would terminate, having matched a non-variable value, it instead calls the auxiliary procedure *ret-flatten*. This procedure accepts the return value v, a list of equivalent variables m, and the original substitution s. *ret-flatten* extends the substitution, binding each member of m to the final value v. *ret-flatten*, and hence *walk-flatten*, returns both v and the possibly extended substitution $\hat{s}$ so that $\hat{s}$ can be used in the next call to *walk-flatten*.

The techniques presented thus far optimize for substitutions containing very long chains of variable bindings. Testing with existing miniKanren programs, however, has shown that a typical substitution does not contain very long chains; rather, very large substitutions with hundreds of bindings are common. Next, we consider more fundamental changes to address this observation.

4 Searching for a Better Representation

Substitution representations based on sequential-access lists have an unavoidable problem: the worst-case time complexity of walking a variable is at best linear in the size of the substitution, and as we have seen, some algorithms are quadratic in the worst case. This is true for idempotent substitutions as well as for any of the representations shown in the previous section. In this section, we consider alternative representations for which the asymptotic worst-case performance of *walk* is logarithmic in the size of the substitution. Several of the representations we discuss also have other desirable properties.

4.1 Binary Search Trees

Binary search trees (BSTs) are binary trees in which each node consists of data and two subtrees: left and right. A non-negative integer key is associated with each node such that the keys for all nodes in the left subtree of a node n are

less than or equal to n's key, and keys for nodes in n's right subtree are greater than n's key. If the tree is balanced, insertion, lookup, and update operations in a BST are all logarithmic in the number of elements in the tree [7].

We can represent a triangular substitution with a BST by using the number of previously created variables as the key for each variable. However, because variables may be unified at arbitrary times relative to when they were created, we have no guarantee that the BST will be balanced. Consider the case in which key values are strictly increasing and the substitution is extended with a binding for each variable in the same order in which they were created. In that case, lookup of the most recently added element will be linear in the number of elements in the tree.

An alternative choice for key values, such as those generated by a non-repeating hash function, can ameliorate the balance problem somewhat. Balanced binary tree representations such as red-black trees [11] could also be suitable, but have more complicated insertion operations.

4.2 Prefix Trees

Binary search trees only perform well with specially ordered data and when the cost of key comparison is small. Prefix trees (or tries) offer another approach that takes advantage of the structure of keys—in our case, the bits of binary numbers—and distributes the cost of key comparison throughout the traversal of the tree [10]. Prefix trees are n-ary trees in which each non-leaf node represents a prefix of the key determined by its depth in the tree, and one node exists for every common key prefix.

A prefix tree representation of triangular substitutions that uses integers as keys is a binary tree with data only at the leaf nodes. Each non-leaf node has two subtrees: left and right. The left subtree contains all nodes whose keys have a 0 at the dth bit, where d is the current node's depth. The right subtree similarly contains all nodes whose keys have a 1 at the position corresponding to the node's depth. Each leaf node contains the portion of the key remaining after traversing the tree to that leaf, along with its associated data. In other words, the key stored in a given leaf upon insertion is the original key that was to be inserted, but right-shifted by the leaf's depth.

When performing a lookup for a given key in the prefix tree, the bits in the key are considered in order from least significant to most significant. (At each level, the key is shifted right by one, so the current least significant bit is always the bit under consideration.) If the current node is a leaf and its key matches the current value of the lookup key, the leaf is a match. Otherwise, the key was not found, and the lookup fails. If the current node is not a leaf, the least significant bit of the current search key determines the subtree in which to recur. If the selected subtree is not present, the lookup fails.

Insertion, lookup, and update of a prefix tree all take time linear in the lesser of the length of the key and the number of elements in the tree. As with BSTs, there is no guarantee of balance. Moreover, when using fixed-precision integer keys, distributing the cost of key comparison over the levels of the tree is not significantly more efficient than comparing the full key at every level, since keys

have a fixed range and can be compared in constant time (relative to the size of the prefix tree).

Besides their better average cost for access, prefix trees are preferable to BSTs because common sequences of insertions result in more balanced trees. Consider inserting the keys 0, 1, 2, and 3 into an initially empty BST. After this sequence, the tree is maximally unbalanced with a depth of 4. After inserting the same sequence into a prefix tree, the result is a balanced tree of depth 3.

Although BST and prefix tree representations of substitutions outperform simple association lists in several ways, they both fall short in one respect: extending the substitution with a new binding takes time logarithmic in the size of the substitution or, for prefix trees, linear in the size of the key. With association lists, extension takes constant time. We next consider a representation of substitutions that in many cases allows extension in constant time.

4.3 Skew Binary Random-Access Lists

Numerical representations—data structures that are patterned after number systems—are a natural choice for implementing container data types such as lists. In a numerical representation, a container object of size n is patterned after the representation of the number n in a given number system, and operations on container objects are patterned after their analogous arithmetic operations. *Binary random-access lists* [10] are a general-purpose numerical representation for lists based on binary numbers. For our purposes, a binary random-access list (BRAL) of size n contains a complete binary tree (containing data only at leaf nodes) for each 1 in the binary representation of n. The *cons* operation on BRALs is analogous to the increment operation on binary numbers; likewise, an *uncons* operation that deconstructs a list and returns its head and tail is analogous to decrementing binary numbers. (The *head* and *tail* operations on BRALs can be readily implemented in terms of *uncons*.)

In this section, we present a representation for substitutions based on *skew binary random-access lists* (SBRALs) [10,9]. To motivate our discussion of the advantages of SBRALs over standard BRALs, consider the problem of efficiently incrementing and decrementing binary numbers. Incrementing a binary number with a low-order bit of 0 takes constant time: we simply change the low-order bit to 1. However, incrementing a binary number with a low-order bit of 1 requires us to perform a carry; in the worst case, in which all the bits are 1, carrying requires $\log(n)$ operations in the size of the number being incremented (or n operations in the number of bits). By the same token, it is cheaper to decrement binary numbers containing many 1s and more expensive to decrement those with many 0s. This property of worst-case $\log(n)$ carry operations is not specific to binary numbers; it generalizes to any ordinary base system. If we plan to increment and decrement often, we would do well to instead choose a representation for numbers that allows us to perform these operations in constant time, regardless of the number being incremented or decremented.

Consider an alternate representation in which the weight w_i of the ith digit is $2^{i+1} - 1$, rather than 2^i as in ordinary binary numbers. In this representation, we allow the digits 0, 1, and 2, stipulating that 2 is only allowed as the lowest

non-zero digit. Such numbers are said to be in *canonical skew binary form* [8,9] and have the desirable property that both increment and decrement operations run in $O(1)$ worst-case time, and yield a result that is still in canonical skew binary form.[5]

We would like the representation we use for substitutions to support constant-time *cons* and *uncons* operations corresponding to the constant-time increment and decrement operations that we have seen for skew binary numbers. SBRALs are a list representation based on skew binary numbers that meet this requirement elegantly. An SBRAL comprises a list of complete binary trees, with one tree for each 1 and two trees for each 2 in the corresponding skew binary number.

We can represent a triangular substitution of size n using the SBRAL corresponding to the skew binary number n. Each non-zero digit of n corresponds to a pair containing its skew binary weight and a tree containing the bindings of the substitution. Since the weights are explicitly represented in the SBRAL , the representation can be *sparse*; we can omit zeros from the list of digits without ambiguity.

When we extend the substitution, if the first and second elements of the list have the same weight, they correspond to a single 2 and are replaced with a 0 digit in the same position—omitted because of the sparse representation—and a 1 or 2 in the next higher digit. Again, a 2 is represented as two trees of the same weight. If the first two elements do not have the same weight, or if the list has fewer than two elements, digit 0 is incremented.

The following *cons-sbral* procedure[6] implements the extend operation for SBRALs. For clarity, we show here a simplified version; our full implementation of SBRALs is presented in the accompanying Appendix 7.

```
(define cons-sbral
 (lambda (v ls)
   (pmatch ls
    (((,w1 . ,t1) (,w2 . ,t2) . ,ls*)
     (if (= w1 w2)
       (cons `(,(+ 1 (+ w1 w2)) .
                ,(make-node v t1 t2)) ls*)
       (cons `(1 . ,v) ls)))
    (,ls (cons `(1 . ,v) ls)))))
```

To construct the weight for digit $n+1$ from two weights at n, we must add 1. Equivalently, we can observe that $1 + 2 \cdot 2^{j+1} - 1 = 2^{j+2} - 1$.

SBRAL representations of triangular substitutions have a number of advantages over BST and prefix tree representations. New bindings are added at the front of the list, making lookups for recently bound variables very fast; programs that exhibit a high degree of locality between variable creation and lookup perform especially well. However, because of this, the indices for existing elements

[5] Myers [8] shows that every natural number has a unique skew binary canonical form. We henceforth assume that all skew binary numbers we discuss are in their canonical form.

[6] Here we use *pmatch*, a simple pattern matcher for Scheme developed by Oleg Kiselyov and described in Byrd and Friedman [3].

grow as new elements are added. For example, if an element is located at index 7 and then 3 elements are added, its index becomes 10. To maintain invariant indices, the current list size is stored with the variable instead of the index. When performing a lookup or update, the saved size is subtracted from the current size to determine the current correct index.

SBRALs have logarithmic bounds for update and lookup similar to BSTs and prefix trees. Unlike those representations, they have constant extension time. However, we cannot immediately take advantage of this. Because variables in a substitution are shared across potentially many threads of computation, a variable's index must be set at the time it is created. Different threads cannot attempt to bind the variable to different indices in the SBRAL.

One way to ensure this is to extend the substitution at variable creation time with a binding to a placeholder. If the variable is subsequently bound to a value, an update is necessary. This meets the requirement that a variable's index is determined before the computation can split, but we lose entirely the benefit of constant-time extension; all bindings due to unification require logarithmic-time updates. We note that even with this limitation, the SBRAL implementation is faster than BSTs or prefix trees.

4.4 Deferred-Extension SBRALs

A better solution is to reserve indices in the SBRAL at variable creation time without immediately extending it. During unification, when the substitution is extended, the index of the variable being bound is compared against the instantiated size of the SBRAL. If the variable index is lesser or equal, an update must be performed. If it is greater, extension is performed until the instantiated size reaches the variable index. Any intervening extensions are made with placeholder values. The result is that if variables are bound in the substitution in the same order they are created, extension is always constant time. In practice, some but not all programs exploit this property. However, we can make one simple and semantics-preserving optimization immediately: when two fresh logic variables are unified, binding the newer variable to the older gives us constant-time extension. This optimization can be implemented with a straightforward modification to *unify*.

An additional benefit of deferred extension occurs when performing a lookup. If the variable's index is greater than the instantiated size of the SBRAL, the lookup fails in constant time. There is no need to even check the list, because we know no binding could have been made to an index outside its bounds.

5 Performance

We compared the performance of association-list representations of triangular substitutions using six versions of *walk* discussed in section 3, as well as the four representations discussed in section 4: BSTs, prefix trees, SBRALs, and deferred-extension SBRALs. The tests were performed in miniKanren, with Chez Scheme

7.4 on a 3 GHz Intel Core 2 Duo E8400 machine running Red Hat Linux 5.2. Figure 1 shows the running time (in milliseconds) of each representation on six representative miniKanren programs. *mktests* is a suite of more than three hundred short, functional tests for verifying implementations of miniKanren. The log^o and $append^o$ tests are both adapted from Friedman, Byrd, and Kiselyov [5]; log^o generates all triples of positive integers b, q, r such that the equation $68 = b^q + r$ holds, and $append^o$ computes the "append" relation on lists, generating the first 700 answers when given all fresh variables. The $perm^o$ test generates all permutations of a given input list once, finding the first 7 answers when given all fresh variables. The *leantap* test runs the first 46 Pelletier problems [13] using a version of αKanren, an extension to miniKanren with operators for nominal logic programming [3]. Finally, the *zebra* test is the classic "zebra" logic puzzle run 1000 times.[7]

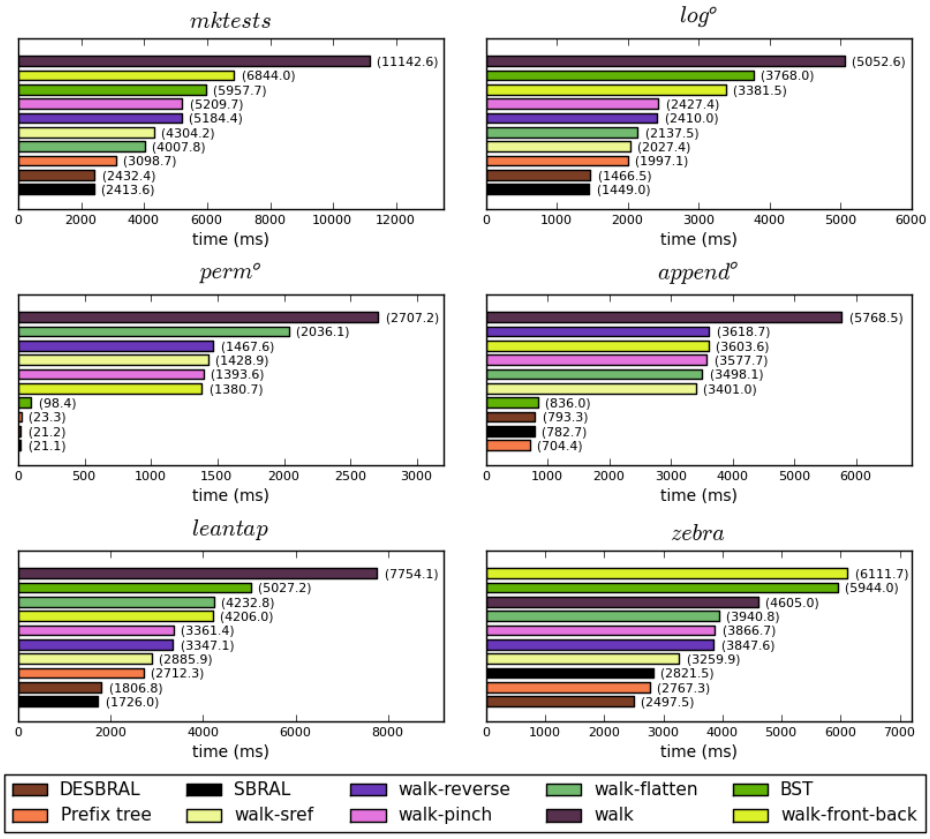

Fig. 1. Mean running time (in milliseconds) for ten runs of six representative miniKanren programs using various representations of triangular substitutions.

[7] Omitted for brevity are standard deviation and memory usage statistics; these are available, along with complete source code, at `http://iucs-relational-research.googlecode.com/files/subst.tgz`.

Although all of the random-access representations perform well, the versions using SBRAL representations generally perform the best. The superior performance of random-access list representations is especially evident in the results of the $perm^o$ test, in which they are several orders of magnitude faster than any of the association-list representations, regardless of the version of *walk* being used.

We also performed a test to measure the performance of variable lookup independent of particular miniKanren programs. Figure 2 shows the running time (in milliseconds) of these tests. The *worst-case* test performs a single lookup in a 60,000-element substitution in which the first binding is v_0 to v_1 and each subsequent binding is v_i to v_{i+1}, requiring the maximum number of recursions during lookup. Here we see clearly the difference among the implementations with linear-, logarithmic-, or quadratic-bounded lookup.

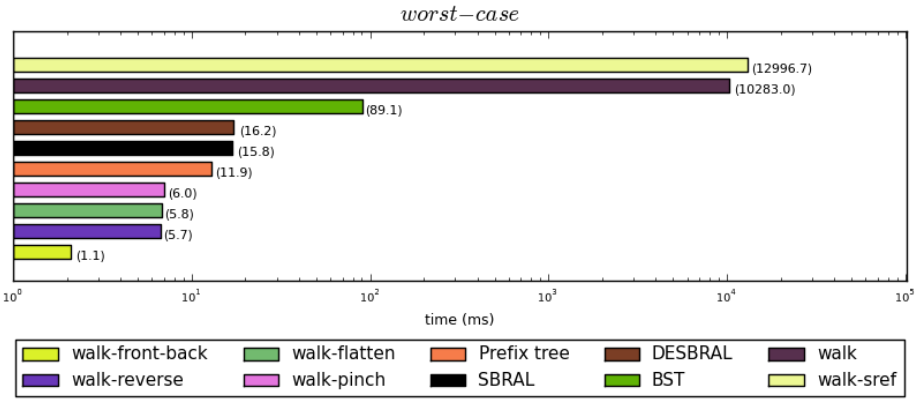

Fig. 2. Mean running time (in milliseconds) for ten variable lookups in a worst-case substitution of 60,000 elements for various representations of triangular substitutions.

6 Future Work

We note that skew binary random-access lists are one of a host of numerical representations that are potential candidates for efficiently representing triangular substitutions. Of these, we hypothesize that finger trees [6], which are relatively straightforward to implement and support amortized constant-time extension, are a promising possibility.

SBRAL representations support constant-time extension if variables are bound in the order they are created. This property suggests automatic reordering of variable creation as a promising direction for future research. In principle, compile-time or run-time program transformations that automatically reorder variable creation could guarantee constant-time extension for a broad class of miniKanren programs. More work remains to determine whether such transformations would be feasible in practice.

Finally, prefix trees that represent keys in big-endian format [12] have been shown to have better sequential insertion performance than little-endian prefix trees such as those shown in this paper. It may be that, for our application, the

benefit of fast sequential insertion outweighs prefix trees' more expensive average random insertion cost; future work could test this hypothesis.

7 Conclusion

The fast extension time and linear space requirement of triangular substitutions make them an attractive choice for purely functional implementations of logic systems such as miniKanren. However, typical implementations of triangular substitutions have the disadvantage of costly lookup. We have demonstrated a variety of representations for triangular substitutions that decrease the cost of lookup while maintaining most of their desirable properties. Our most efficient representations, based on skew binary random-access lists, combine logarithmically bounded update and lookup operations with an often constant-time extension operation, resulting in a substantial performance benefit for existing miniKanren programs.

References

1. Franz Baader and Wayne Snyder. Unification theory. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, pages 445–532. Elsevier and MIT Press, 2001.
2. William E. Byrd and Daniel P. Friedman. From variadic functions to variadic relations: A miniKanren perspective. In Robby Findler, editor, *Proc. of the 2006 Scheme and Functional Programming Workshop, Portland, Sep. 17, 2006*, University of Chicago Technical Report TR-2006-06, pages 105–117, 2006.
3. William E. Byrd and Daniel P. Friedman. αKanren: A fresh name in nominal logic programming. In *Proceedings of the 2007 Workshop on Scheme and Functional Programming*, 2007.
4. Luca Cardelli. Basic polymorphic typechecking. *Science of Computer Programming*, 8:147–172, 1987.
5. Daniel P. Friedman, William E. Byrd, and Oleg Kiselyov. *The Reasoned Schemer*. The MIT Press, July 2005.
6. Ralf Hinze and Ross Paterson. Finger trees: a simple general-purpose data structure. *Journal of Functional Programming*, 16(2):197–217, 2006.
7. Donald E. Knuth. *The art of computer programming*, volume 3. Addison-Wesley Longman Publishing Co., Boston, MA, 2nd edition, 1998.
8. Eugene W. Myers. An applicative random-access stack. *Information Processing Letters*, 17(5):241–248, December 1983.
9. Chris Okasaki. Purely functional random-access lists. In *Functional Programming Languages and Computer Architecture*, pages 86–95. ACM Press, 1995.
10. Chris Okasaki. *Purely functional data structures*. Cambridge University Press, 1998.
11. Chris Okasaki. Red-black trees in a functional setting. *Journal of Functional Programming*, 9(4):471–477, 1999.
12. Chris Okasaki and Andrew Gill. Fast mergeable integer maps. In *Workshop on ML*, pages 77–86, 1998.
13. Francis Jeffry Pelletier. Seventy-five problems for testing automatic theorem provers. *Journal of Automated Reasoning*, 2(2):191–216, 1986.
14. Tim Sheard. Generic unification via two-level types and parameterized modules. In *ICFP '01: Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*, volume 36, pages 86–97. ACM Press, October 2001.

Appendix

We present here the complete Scheme implementation for the deferred-extension SBRAL representation of substitutions described in section 4.4.

```
(module skew-bral-def (kd:empty kd:size kd:associate
                       kd:lookup kd:update kd:get-value kd:reserve)
  (import (scheme))
  (define-syntax make-bral
    (syntax-rules ()
      ((_ size realized ls) `(,size ,realized . ,ls))))
  (define bral-size car)
  (define set-bral-size! set-car!)
  (define bral-realized cadr)
  (define bral-ls cddr)
  (define-record node ((immutable val)
                       (immutable even)
                       (immutable odd)))
  (define kd:empty (lambda () (make-bral 0 -1 '())))
  (define kd:reserve
    (lambda (b)
      (let ((n (bral-size b))) (set-bral-size! b (+ 1 n)) n)))
  (define kd:size (lambda (b) (bral-size b)))
  (define kd:associate
    (lambda (i v b)
      (if (> i (bral-realized b))
          (make-bral
            (bral-size b)
            i
            (consˆ v (bral-ls b) (- i (bral-realized b))))
          (kd:update i v b))))
  (define kd:lookup
    (lambda (i b)
      (if (> i (bral-realized b))
          #f
          (lookup (reverse-idx i b) (bral-ls b)))))
  (define kd:update
    (lambda (i v b)
      (make-bral
        (bral-size b)
        (bral-realized b)
        (update (reverse-idx i b) v (bral-ls b)))))
  (define kd:get-value
    (lambda (v) (cond ((node? v) (node-val v)) (else (car v)))))
  (define br (vector 'br))
  (define reverse-idx
    (lambda (i b) (- (bral-realized b) i)))
  (define shift (lambda (n) (fxsra n 1)))
  (define consˆ
    (lambda (v ls n)
```

```
    (let* ((v* (if (= 1 n) v br))
           (res (cond
                  ((and (pair? ls)
                        (pair? (cdr ls))
                        (= (caar ls) (caadr ls)))
                   (cons
                     `(,(+ 1 (+ (caar ls) (caadr ls)))
                       .
                       ,(make-node v* (cdar ls) (cdadr ls)))
                     (cddr ls)))
                  (else (cons `(1 . ,v*) ls)))))
      (if (= 1 n) res (cons^ v res (- n 1))))))
(define lookup-tree
  (lambda (w i t)
    (cond
      ((node? t)
       (if (zero? i)
           (if (eq? (node-val t) br) #f t)
           (let ((w/2 (shift w)))
             (if (<= i w/2)
                 (lookup-tree w/2 (- i 1) (node-even t))
                 (lookup-tree w/2 (- (- i 1) w/2) (node-odd t))))))
      (else
       (if (zero? i) (if (eq? t br) #f (cons t '())) #f)))))
(define lookup
  (lambda (i ls)
    (cond
      ((null? ls) #f)
      (else
       (let ((t (car ls)))
         (if (< i (car t))
             (lookup-tree (car t) i (cdr t))
             (lookup (- i (car t)) (cdr ls)))))))))
(define update-tree
  (lambda (w i v t)
    (cond
      ((node? t)
       (if (zero? i)
           (make-node v (node-even t) (node-odd t))
           (let ((w/2 (shift w)))
             (if (<= i w/2)
                 (make-node
                   (node-val t)
                   (update-tree w/2 (- i 1) v (node-even t))
                   (node-odd t))
                 (make-node
                   (node-val t)
                   (node-even t)
                   (update-tree
                     w/2
```

```
                    (- (- i 1) w/2)
                    v
                    (node-odd t)))))))
      (else
       (if (zero? i)
           v
           (error 'update-tree "illegal index ~s" i))))))
(define update
  (lambda (i v ls)
    (cond
      ((null? ls) (error 'k:update "illegal index ~s" i))
      (else
       (let ((t (car ls)))
         (if (< i (car t))
             (cons
               `(,(car t) . ,(update-tree (car t) i v (cdr t)))
               (cdr ls))
             (cons t (update (- i (car t)) v (cdr ls)))))))))))
```