Efficient Nondestructive Equality Checking for Trees and Graphs

Michael D. Adams and R. Kent Dybvig

Indiana University Computer Science Department

{adamsmd,dyb}@cs.indiana.edu

Abstract

The Revised[6] Report on Scheme requires its generic equivalence predicate, `equal?`, to terminate even on cyclic inputs. While the terminating `equal?` can be implemented via a DFA-equivalence or union-find algorithm, these algorithms usually require an additional pointer to be stored in each object, are not suitable for multithreaded code due to their destructive nature, and may be unacceptably slow for the small acyclic values that are the most likely inputs to the predicate.

This paper presents a variant of the union-find algorithm for `equal?` that addresses these issues. It performs well on large and small, cyclic and acyclic inputs by interleaving a low-overhead algorithm that terminates only for acyclic inputs with a more general algorithm that handles cyclic inputs. The algorithm terminates for all inputs while never being more than a small factor slower than whichever of the acyclic or union-find algorithms would have been faster. Several intermediate algorithms are also presented, each of which might be suitable for use in a particular application, though only the final algorithm is suitable for use in a library procedure, like `equal?`, that must work acceptably well for all inputs.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—data types and structures; G.2.2 [*Discrete Mathematics*]: Graph Theory—graph algorithms

General Terms Algorithms, Performance

Keywords equality, union-find, DFA equivalence, eq hash tables, Scheme

1. Introduction

The Revised[6] Report on Scheme (R6RS) changed the semantics of `equal?` so that it is now required to terminate even when given cyclic inputs and to return true if and only if the (possibly infinite) unfoldings of its arguments into regular trees are equal as ordered trees (Sperber et al. 2007). This is the same as determining whether the arguments are equivalent when interpreted as deterministic finite automata (DFAs) (Clinger 2006). Thus, it is possible to implement `equal?` using an algorithm designed to test for the equivalence of DFAs, e.g., Hopcroft and Karp's DFA equivalence algorithm (1971), or one of the closely related union-find algorithms (Galler and Fisher 1964; Galil and Italiano 1991).

A complication is that union-find algorithms require a pointer-sized cell to be set aside in each node of the tree or graph, and they update this node as the algorithm progresses. This is unsuitable in multitasking or multithreaded programs, in which multiple threads of execution may be performing concurrent equality checks on the same values.

Moreover, adding an additional cell to each object to support just this one operation is excessively expensive. For pairs, which contain only two cells to start with, the addition of a cell increases the size by 50% or even 100% given typical 64-bit allocation alignment restrictions (to support objects containing double-precision floating-point numbers) on 32-bit machines. In less memory-efficient implementations, where pairs also include a header word, the overhead of an additional word would be less but still significant.

A solution to both problems is to associate the necessary additional cell with each value, as needed, in a separate "eq" hash table, with a fresh hash table allocated for each `equal?` operation.

Unfortunately, when the overhead of union-find (in comparison to a simple tree-equality check) is combined with the additional hash-table overhead, the resulting algorithm becomes unsuitably slow for trees, i.e., inputs without shared structure or cycles, which are likely to be the most common inputs to `equal?`.

To ameliorate this problem, one can use an inexpensive tree-equality pre-check, with a bound on the number of recursive calls to avoid infinite regression when cyclic inputs are provided, and switch to the slower union-find algorithm when the bound is exceeded (Clinger 2006).

Unfortunately, no value for the bound on the tree-equality pre-check works well for all inputs: a large bound penalizes small cyclic graphs or dags with lots of shared structure, and a small bound penalizes larger tree structures.

Our solution to this problem is an algorithm that interleaves an inexpensive tree-equality check with a hash-table-based union-find algorithm in order to achieve good performance for both trees and graphs. Although the idea is simple, several subtle points must be addressed to avoid poor worst-case behavior. This paper presents the algorithm and its implementation, establishes that its asymptotic behavior is the same as the union-find algorithm, and presents benchmarks showing that it is nearly as fast in most cases as whichever is the faster of the tree-equality and union-find algorithms for both tree inputs and inputs with substantial shared structure or cycles. The benchmarks also show that its worst-case behavior is within a small constant factor of the tree-equality and union-find algorithms.

Section 2 describes the new `equal?` procedure in more detail and gives a few examples of its use. Section 3 presents an implementation of a straight hash-table-based union-find algorithm, analyzes its worst-case behavior, and shows how it performs on sev-

©ACM, 2008. This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming* (2008). http://doi.acm.org/10.1145/1411204.1411230

ICFP'08, September 22–24, 2008, Victoria, BC, Canada.

Copyright © 2008 ACM 978-1-59593-919-7/08/09…$5.00

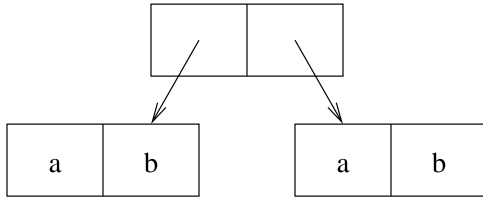

Figure 1. Two equal acyclic values

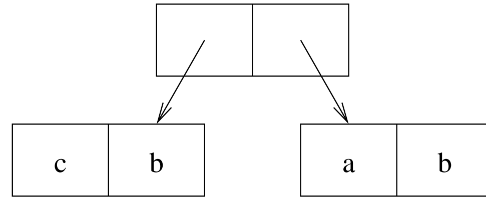

Figure 2. Values no longer equal after mutation

eral kinds of sample input. Section 4 adds an inexpensive, bounded tree-equality pre-check and shows that this helps for some inputs but not others. Section 5 presents one version of our interleaved algorithm and shows that this largely addresses the problems with the other algorithms. Section 6 presents the final version of our interleaved algorithm and shows that it has better worst-case behavior and is usually faster, especially for small acyclic inputs. Section 7 discusses related work, and Section 8 presents conclusions.

2. The equal? procedure

The Revised[6] Report `equal?` procedure accepts two arbitrary arguments and returns true (`#t`) "if and only if the (possibly infinite) unfoldings of its arguments into regular trees are equal as ordered trees" (Sperber et al. 2007). Otherwise, it returns false (`#f`).

In essence, two values are equivalent, in the sense of `equal?`, if the structure of the two values cannot be distinguished by any composition of pair and vector accessors along with the `eqv?`, `string=?`, and `bytevector=?` procedures for comparing atomic data when applied to the leaves of the values. For example, the values of the two expressions below are equal because they have exactly the same structure and same values at the leaves.

```
(cons 'a 'b)
(cons 'a 'b)
```

The values of the expressions below are also equal:

```
(cons (cons 'a 'b) (cons 'a 'b))
(let ([x (cons 'a 'b)]) (cons x x))
```

even though the former consists of three pairs while the latter consists of only two, as illustrated in Figure 1, because any composition of `car` and `cdr` performed in parallel on the two values yields the same (equal) structures.

On the other hand, the values of the following expressions are not equal:

```
(cons (cons 'a 'b) (cons 'a 'c))
(let ([x (cons 'a 'b)]) (cons x x))
```

since they differ at one leaf. The values of the following are also unequal:

```
(cons (cons 'a 'b) (cons 'a 'b))
(cons 'a 'b)
```

since they differ in basic structure.

While two values may be considered equal even if they do not have identical internal structure, they can be distinguished, and hence determined not to have identical internal structure, by some parallel mutation that would not distinguish values with identical internal structure, as illustrated by the following.

```
(define x (cons (cons 'a 'b) (cons 'a 'b)))
(define y (let ([x (cons 'a 'b)]) (cons x x)))

(equal? x y)  ⇒  #t

(set-car! (car x) 'c)
(set-car! (car y) 'c)

(equal? x y)  ⇒  #f
```

Prior to mutation, the values of `x` and `y` are identical to the equal but structurally distinct values illustrated in Figure 1. Their structures after mutation, which are not equal, are illustrated in Figure 2.

The examples so far are all consistent with the Revised[5] Report semantics for equal?, which might be defined as follows:

```
(define (r5rs-equal? x y)
  (let e? ([x x] [y y])
    (cond
      [(eq? x y) #t]
      [(pair? x)
       (and (pair? y)
            (e? (car x) (car y))
            (e? (cdr x) (cdr y)))]
      [(vector? x)
       (and (vector? y)
         (let ([n (vector-length x)])
           (and (= (vector-length y) n)
             (let f ([i 0])
               (or (= i n)
                   (and
                     (e? (vector-ref x i)
                         (vector-ref y i))
                     (f (+ i 1))))))))]
      [(string? x)
       (and (string? y) (string=? x y))]
      [else (eqv? x y)])))
```

In fact, the Revised[6] Report semantics for `equal?` coincides with the Revised[5] Report `equal?` semantics (Kelsey et al. 1998) for

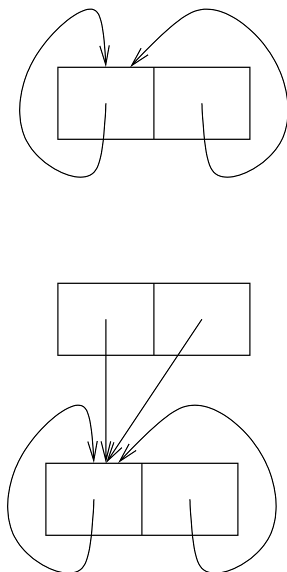

Figure 3. Two equal cyclic values

all inputs upon which the latter is required to terminate, i.e., all directed, acyclic graphs. They differ for cyclic graphs: behavior of the Revised[5] Report `equal?` on cyclic graphs is unspecified (and the implementation above will recur indefinitely), while the Revised[6] Report `equal?` is required to terminate with an answer consistent with its behavior on acyclic graphs. For example, the cyclic values of the following expressions are equal.

```
(let ([x (cons 'x 'x)])
  (set-car! x x)
  (set-cdr! x x)
  x)
(let ([x (cons 'x 'x)])
  (set-car! x x)
  (set-cdr! x x)
  (cons x x))
```

The structure of these values is shown in Figure 3.

3. Straightforward union-find

As stated in Section 1, the problem of determining whether two values are equal per the R6RS semantics is essentially equivalent to the problem of determining whether two DFAs are equivalent. We simply consider the (pair and vector) nodes in our directed graph to be the states of the DFA, the links from one node to another to be the transitions of the DFA, and the accessor names (i.e., `car` and `cdr`) or ordinals (i.e., vector indices) to be the symbols of the DFA's alphabet. The start state of each graph is the root node of the graph, and the final states are the atomic leaves. DFA equivalence can be established via the Hopcroft and Karp DFA equivalence algorithm (1971) or via a union-find algorithm (Tarjan 1975).

Our first version of R6RS `equal?` is thus a straightforward coding of the "splitting" union-find algorithm recommended by Tarjan and van Leeuwen (1984), with an "eq" hash table to associate the required additional cells with the nodes visited by the algorithm. The algorithm traverses the nodes of the graph (a.k.a. states of the DFA), marking equivalent any two nodes reached by the algorithm if they have the same surface structure, then recurring on the subgraphs directly reachable from the two nodes. The equality check fails if two nodes reached during this process have different structure (say if only one of the two is a pair) or if two leaves differ. If a cycle exists in an equal structure, the algorithm sees that they have already been marked equivalent and does not descend again into the subgraphs, thus assuring termination.

Because the equivalence relationship is transitive, the use of equivalence classes rather than pairwise equivalence flags or links sometimes allows the algorithm to terminate more quickly, since it does not recur into the subgraphs of two nodes that are both equivalent to a third node.

3.1 Implementation

The code shown below for the union-find `equal?` algorithm creates a hash table used to record equivalence classes, then initiates a recursive equality check.

```
(define (uf-equal? x y)
  (let ([ht (make-eq-hashtable)])
    (define (e? x y)
      (cond
        [(eq? x y) #t]
        [(pair? x)
         (and (pair? y)
           (or (union-find ht x y)
               (and (e? (car x) (car y))
                    (e? (cdr x) (cdr y)))))]
        [(vector? x)
         (and (vector? y)
           (let ([n (vector-length x)])
             (and (= (vector-length y) n)
               (or (union-find ht x y)
                   (let f ([i 0])
                     (or (= i n)
                         (and (e? (vector-ref x i)
                                  (vector-ref y i))
                              (f (+ i 1)))))))))]
        [(string? x)
         (and (string? y) (string=? x y))]
        [else (eqv? x y)]))
    (e? x y)))
```

Each step of the recursion checks to see if the inputs are pointer-equivalent, i.e., "eq," to avoid recurring into the subgraphs of identical values. If not, it handles recursively the cases where both values are pairs or both values are vectors of the same length. It defers to `string=?` for strings and to `eqv?` for other atomic values[1].

Prior to recurring on the elements of pairs or vectors, the code checks for and records the equivalence of the two pairs or vectors via the `union-find` procedure. In the case where `union-find` returns false, `uf-equal?` recurs on the individual elements of the pair or vector in the natural way.

Since we need to record as equivalent any two inputs if they are not already known to be equivalent, the `union-find` procedure, shown in Figure 4, combines the union and find operations of the union-find algorithm, which is less expensive than doing the two operations separately. The procedure returns true if the two values it receives are already in the same equivalence class; otherwise, it destructively merges the equivalence classes and returns false. The `union-find` procedure performs the same operation as the contingent-unite algorithm of Tarjan and van Leeuwen (1984), though it uses a more direct merger of the union and find operations that involves fewer side effects and, for our purposes, appears to be as efficient in practice.

[1] It should also defer to `bytevector=?` for bytevectors, but we have omitted that case from this and the other algorithm variants.

```
(define (union-find ht x y)
  (define (find b)
    (let ([n (unbox b)])
      (if (box? n)
          (let loop ([b b] [n n])
            (let ([nn (unbox n)])
              (if (box? nn)
                  (begin
                    (set-box! b nn)
                    (loop n nn))
                  n)))
          b)))
  (let ([bx (eq-hashtable-ref ht x #f)]
        [by (eq-hashtable-ref ht y #f)])
    (if (not bx)
        (if (not by)
            (let ([b (box 1)])
              (eq-hashtable-set! ht x b)
              (eq-hashtable-set! ht y b)
              #f)
            (let ([ry (find by)])
              (eq-hashtable-set! ht x ry)
              #f))
        (if (not by)
            (let ([rx (find bx)])
              (eq-hashtable-set! ht y rx)
              #f)
            (let ([rx (find bx)] [ry (find by)])
              (or (eq? rx ry)
                  (let ([nx (unbox rx)]
                        [ny (unbox ry)])
                    (if (> nx ny)
                        (begin
                          (set-box! ry rx)
                          (set-box! rx (+ nx ny))
                          #f)
                        (begin
                          (set-box! rx ry)
                          (set-box! ry (+ ny nx))
                          #f)))))))))
```

Figure 4. The `union-find` procedure

Equivalence classes are represented as chains of single-celled boxes, with the last box of each chain serving as the representative of the class. The "box" data type can be declared as a record type or defined via operations on pairs as shown below.

```
(define box? pair?)
(define box list)
(define unbox car)
(define set-box! set-car!)
```

To implement the *weighted union rule* (Tarjan 1975) of the union-find algorithm, the representative holds the size of the class.

Four cases are handled separately by the algorithm: (1) neither input value has been seen before, (2) the first has been seen but not the second, (3) the second has been seen but not the first, and (4) both have been seen. In the first case, we need to associate both values with a box to hold the equivalence class. Since both values will be placed in the same equivalence class, we can get away with creating a single box rather than two separate boxes, reducing memory-allocation and memory-reference overhead and possibly reducing overall chain lengths. In the second and third cases, which are mirror images, the new value is associated with the representative of the previously seen value, effectively adding the new value to the existing equivalence class without any allocation overhead (beyond that incurred by the hash-table mechanism) and without the need to increment the class's size. The representative is located via the `find` procedure, which employs a form of path compression that Tarjan and van Leeuwen refer to as *splitting* (1984). With splitting, each box in the chain becomes linked to the one two beyond it rather than to the representative, allowing the code to avoid a second traversal without affecting the asymptotic behavior.

The fourth case is the most involved. In this case, the code compares the representatives of the two input values. If they are pointer-equivalent, the values are already in the same class and the code returns true. Otherwise, the representative of the smaller class becomes linked to the representative of the larger class and the size is updated to reflect the size of the newly joined class.

The hash-table mechanism used by our implementation of the union-find algorithm is an implementation of the R6RS "eq" hashtable interface. It bases the hash code for an object on its address in memory. Since objects may move during a garbage collection, their hash codes may change over time. The hash-table mechanism handles this efficiently by rehashing only the objects that have actually moved since the last access (Ghuloum and Dybvig 2007).

3.2 Analysis

In order to determine a tight bound on the number of calls to `e?`, we first establish that the number of calls to `e?` is bounded by $nm + 1$, where n is the total number of non-leaf nodes in the two inputs and m is the out-degree of the largest node, i.e., the length of the largest vector, or two if we have only pairs and vectors of length two or less. We do so as follows. At the outset, each node virtually resides in its own equivalence class, so there are exactly n distinct equivalence classes. Each `union-find` operation that returns false merges two distinct equivalence classes. This can happen at most n times before all of the classes have been merged into a single class. Furthermore, recursive calls to `e?` occur only when `union-find` returns false. At most m recursive calls can be made in each of these cases (one for each element of the pair or vector). Thus, we can have at most mn recursive calls to `e?` and at most $mn + 1$ calls total including the initial call.

We can establish a tighter bound by observing that `e?` can recur only if its inputs are both pairs or both vectors of the same length. The n non-leaf nodes consist of p pairs and v_i vectors of each length i represented in the input, for some p and v_i. By the reasoning above, we can generate at most $2p$ recursive calls due to the pairs, and iv_i recursive calls due to vectors of length i. $2p$ and iv_i also happen to be the total numbers of out edges from pairs and vectors of length i, respectively. Summed together, they represent the total number of edges in the two inputs. Thus, the bound on the number of calls to `e?` is actually $l + 1$, where l is the total number of edges.

The amortized cost of each `union-find` call, with weighted union and path compression as implemented above, is $O(\alpha(n))$ where α is the inverse Ackermann function and n is again the number of nodes (Tarjan and van Leeuwen 1984). Hash tables exhibit amortized constant time behavior under assumptions maintained by the underlying implementation and so do not affect the bound. Thus the overall cost of the algorithm is $O(l\alpha(n))$. The value of the inverse Ackermann function is no more than 4 for all reasonable values of n (Tarjan and van Leeuwen 1984) (in particular, for the maximum number of nodes that can fit in a 32- or 64-bit memory space), so the cost is effectively $O(l)$.

This bound assumes that leaves can be compared in constant time, which is the case for most values but not for strings, bytevectors, and bignums.

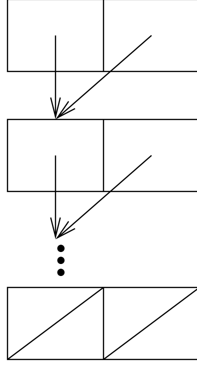

Figure 5. Degenerate DAG

3.3 Benchmarks

We compared the run times of `uf-equal?`[2] and `r5rs-equal?` on a variety of acyclic inputs: lists and inverted lists of various lengths along with balanced binary trees, degenerate directed acyclic graphs (dags), and random dags of various depths. Inverted lists are like lists but are linked through the car rather than cdr field. Degenerate dags consist of a chain of pairs, each connected to the next through the car and cdr fields, as shown in Figure 5. In each case, the values being compared are equal but do not share any structure; comparing unequal values would be the essentially the same as comparing smaller equal values. Bar charts showing the results of all benchmark runs are shown in Figures 8 through 14 of Section 6.2. For purposes of the current comparison, only the first two bars of Figures 8 through 12 are relevant.

As one might expect, `uf-equal?` performs poorly relative to `r5rs-equal?` on lists, inverted lists, trees, and small dags. Indeed, `uf-equal?` outperforms `r5rs-equal?` only for larger dags, where the run time of the latter eventually goes off the scale of the graphs because it treats dags as if they were in their exponentially expanded tree forms. We gave up running `r5rs-equal?` on degenerate dags larger than depth 16, so no results are shown for it at depths 32 and 64.

While `uf-equal?` runs faster for larger dags and can handle cyclic structures upon which `r5rs-equal?` does not terminate, it runs slower for lists and trees. In particular, it runs much slower for the small values that are likely to be common inputs to `equal?`. We would prefer a solution that maintains the useful properties of `uf-equal?` without the performance hit for these smaller values.

4. Union-find with pre-check

As mentioned in the introduction, a way to improve performance on smaller values is to perform a bounded tree-equality pre-check on the inputs before deferring to the slower union-find algorithm. The `precheck/uf-equal?` procedure defined below performs such a check, using the `pre?` procedure, whose definition is given in Figure 6.

[2] In this and subsequent tests, the helpers were made local to the procedure to allow the compiler more latitude for optimization, and fixnum arithmetic operators were used in place of the generic operators. All tests were run in *Chez Scheme* Version 7.4 running as a 32-bit process under the 64-bit Gnu/Linux operating system on a lightly loaded 2.6Ghz AMD Opteron processor with 4GB of memory.

```
(define (pre? x y k)
  (cond
    [(eq? x y) k]
    [(pair? x)
     (and (pair? y)
       (if (<= k 0)
           k
           (let ([k (pre? (car x) (car y) (- k 1))])
             (and k (pre? (cdr x) (cdr y) k)))))]
    [(vector? x)
     (and (vector? y)
       (let ([n (vector-length x)])
         (and (= (vector-length y) n)
           (let f ([i 0] [k k])
             (if (or (= i n) (<= k 0))
                 k
                 (let ([k (pre? (vector-ref x i)
                                (vector-ref y i)
                                (- k 1))])
                   (and k (f (+ i 1) k))))))))]
    [(string? x)
     (and (string? y) (string=? x y) k)]
    [else (and (eqv? x y) k)]))
```

Figure 6. The `pre?` procedure

```
(define k0 400)
(define (precheck/uf-equal? x y)
  (let ([k (pre? x y k0)])
    (and k (or (> k 0) (uf-equal? x y)))))
```

The `pre?` procedure returns false to signify that the values are non-equal, a positive integer to signify that the values are equal, and zero to signify that the equality of the two values could not be determined without exceeding the given bound. The code for `pre?` is similar to the code for `r5rs-equal?`, except that it checks and decrements the bound argument, `k`, each time it encounters a pair and for each vector element. The value 400 for the bound, `k0`, was determined through experimentation, as described in Section 4.2.

4.1 Analysis

The worst-case bound for this version of the algorithm is essentially the same as for the straight union-find algorithm; the pre-check adds only constant overhead.

4.2 Benchmarks

We compared `precheck/uf-equal?` with `r5rs-equal?` and `uf-equal?` on the same set of benchmarks as before, and we separately compared `precheck/uf-equal?` with `uf-equal?` on a set of cyclic inputs, including both random cyclic graphs and graphs constructed specifically to exercise the case where two non-unit equivalence classes must be merged by the `union-find` helper. Again, the results are shown in Figures 8 through 14 of Section 6.2.

As expected, `precheck/uf-equal?` reduces the run times for small inputs to levels approaching those of `r5rs-equal?`, but does not help larger lists and trees, i.e., those whose size exceeds the k_0 bound. A larger bound would improve the performance for larger inputs, but `precheck/uf-equal?` already runs slower than `uf-equal?` for larger dags and for cyclic values. The difference may be acceptable with `k0` set to 400, but not for much larger values of `k0`. Indeed, we selected 400 for the value of `k0` because we felt that higher values penalized cyclic inputs to an unacceptable degree.

5. Interleaved algorithm

For comparisons of large lists and trees, the preceding algorithm is actually slower than the straight `union-find` algorithm, because the pre-check effort is wasted. It is also much slower than `r5rs-equal?` for such inputs, because the `union-find` algorithm adds overhead without any compensating benefits. Yet, we cannot use `r5rs-equal?` because it does not handle cyclic inputs, and we cannot determine *a priori* which algorithm to use since we do not know until we traverse the inputs how large they are and whether they contain cycles.

Our solution to this problem is to interleave the tree-equality checking algorithm with the `union-find` algorithm. We briefly considered independent coroutines, but abandoned the idea because of the overhead involved and also because a tighter interleaving can sometimes result in beneficial synergy, since the tree-shaped portions of a cyclic structure may be tested less expensively with a tree-equality check than with the slower union-find algorithm.

The `interleave-equal?` procedure defined below, along with the `interleave?` procedure shown in Figure 7 implements the interleaved algorithm.

```
(define k0 400)
(define kb -40)
(define (interleave-equal? x y)
  (interleave? x y k0))
```

The `interleave?` procedure is passed the two inputs plus the initial value, `k0`, for its bound argument, `k`. If `k` is greater than zero, `interleave?` uses its `fast?` helper, which is similar to `pre?`. If `k` is less than or equal to zero but greater than the lower bound on `k`, `kb`, `interleave?` defers to its `slow?` helper, which is similar to `uf-equal?`, but augmented to decrement `k` when appropriate. If `k` is exactly `kb`, `interleave?` restarts `k` at a random value between 0 and twice `k`. The random restart value reduces the likelihood of repeatedly tripping on worst-case behavior in cases where the sizes of the input graphs happen to be related to the chosen bounds in a bad way.

Because the algorithm starts out with `k` greater than zero and may terminate before `k` ever reaches zero, `interleave?` puts off creating the hash table until it is first needed, with the help of the `call-union-find` procedure.

When `union-find` returns true, `slow?` returns 0 instead of the current value of `k`, on the theory that if one equivalence is found, more are likely to be found, so we should give `slow?` more time in which to find them.

5.1 Analysis

To establish a bound on this algorithm, we first assume that `interleave?` starts in "slow" mode, i.e., `k` starts out at 0 rather than `k0`, then discuss what happens when `k` starts out at `k0`.

Let r be the number of calls to `union-find` that return false. As discussed in Section 3.2, this can happen at most n times (where, again, n is the number of nodes), so $r \leq n$. Because we perform at least $-k_b$ calls to `slow?` for which `union-find` returns false before making each set of at most $2k_0$ calls to `fast?`, the number of calls f to `fast?` is at most $2(k_0/k_b)r$. Since $r \leq n$, it follows that $f \leq 2(k_0/k_b)n$.

What remains is to count the number of calls s to `slow?` where `union-find` returns true. With the exception of the initial call to `interleave?`, such calls can occur only $rm + fm$ times (where, again, m is the length of the longest vector, or two if there are no vectors with more than two elements), since `slow?` does not recur when `union-find` returns true. Thus, $s \leq rm + fm + 1$. Since $r \leq n$ and $f \leq 2(k_0/k_b)n$, $s \leq nm + 2(k_0/k_b)nm + 1$. By our earlier reasoning in Section 3.2, we can replace the nm terms with

```
(define (interleave? x y k)
 (let ([ht #f])
  (define (call-union-find x y)
   (unless ht (set! ht (make-eq-hashtable)))
   (union-find ht x y))
  (define (e? x y k)
   (if (<= k 0)
       (if (= k kb)
           (fast? x y (random (* 2 k0)))
           (slow? x y k))
       (fast? x y k)))
  (define (slow? x y k)
   (cond
    [(eq? x y) k]
    [(pair? x)
     (and (pair? y)
       (if (call-union-find x y)
           0
           (let ([k (e? (car x) (car y) (- k 1))])
             (and k (e? (cdr x) (cdr y) k)))))]
    [(vector? x)
     (and (vector? y)
       (let ([n (vector-length x)])
         (and (= (vector-length y) n)
           (if (call-union-find x y)
               0
               (let f ([i 0] [k (- k 1)])
                 (if (= i n)
                     k
                     (let ([k (e? (vector-ref x i)
                                  (vector-ref y i) k)])
                       (and k (f (+ i 1) k))))))))))]
    [(string? x) (and (string? y) (string=? x y) k)]
    [else (and (eqv? x y) k)]))
  (define (fast? x y k)
   (let ([k (- k 1)])
    (cond
     [(eq? x y) k]
     [(pair? x)
      (and (pair? y)
        (let ([k (e? (car x) (car y) k)])
          (and k (e? (cdr x) (cdr y) k))))]
     [(vector? x)
      (and (vector? y)
        (let ([n (vector-length x)])
          (and (= (vector-length y) n)
            (let f ([i 0] [k k])
              (if (= i n)
                  k
                  (let ([k (e? (vector-ref x i)
                               (vector-ref y i) k)])
                    (and k (f (+ i 1) k))))))))]
     [(string? x) (and (string? y) (string=? x y) k)]
     [else (and (eqv? x y) k)])))
  (and (e? x y k) #t)))
```

Figure 7. The `interleave?` procedure

l (where, again, l is the total number of out-edges in the input), so $s \leq 2(k_0/k_b + 1)l + 1$.

(The bound on s is overly loose, since if all of the r calls really result in s calls, there can be no f calls, and the second term for s will be zero. Furthermore, we cannot reach the maximum number of f calls without some of the f calls resulting directly in other f calls, since at most n can result from r calls. This in turn further reduces the possible number of s calls. So we believe a significantly tighter bound on s should be possible, but at the time of this writing we have not yet established one.)

The total number of calls t can be determined by summing the three kinds of calls (calls to `slow?` where `union-find` returns true, calls to `slow?` where `union-find` returns false, and calls to `fast?`), i.e., $t \leq n + i2(k_0/k_b+)n + i2(k_0/k_b)l + 1$. If we treat empty vectors as atoms and ignore the special case where `equal?` is called on atomic inputs, each node counts for at least one out-edge, so $n \leq l$, and $t \leq l + 2(k_0/k_b + 1)l + 2(k_0/k_b)l + 1$. This reduces to $t \leq 4(k_0/k_b)l + 1$. Thus, since k_0/k_b is a constant, we have the same asymptotic bound as for `uf-equal?`, but with a larger constant factor.

The most expensive calls are the calls to `slow?` for which `union-find` returns false and hence must merge equivalence classes, and the number of such calls is still bounded by n. Thus the additional calls are likely to be of a less expensive variety.

We now return to the case where k starts out at k_0, as it does in the call to `interleave?` from the `interleave-equal?` procedure defined above. In this case, we can have at most k_0 additional calls to `fast?` and, because of that, at most $k_0 l$ additional calls to `slow?`, so t is now proportional to k_0 rather than to k_0/k_b. While we have been willing to accept a pre-check that has, effectively, k_0 additional calls to `fast?`, the $k_0 l$ additional calls to `slow?` are problematic, which in part leads to our final algorithm in Section 6.

5.2 Benchmarks

Again, benchmark results are shown in Figures 8 through 14 of Section 6.2. The results show that the `interleaved-equal?` is substantially faster than `precheck/uf-equal?` on most inputs and especially so on larger lists and trees. It suffers relative to `precheck/uf-equal?` on many of the cyclic inputs. Interestingly, the disadvantage disappears for the union-exercising graphs shown in Figure 14 as the number of nodes increases. Once it gets past the initial `fast?` calls to the first `slow?` call, it remains in "slow" mode for the duration and wastes no more time in "fast" mode. This appears to justify our choice to return `0` rather than `k` when `union-find` returns true.

6. Interleaved with pre-check

The final version of our algorithm is like the `interleaved-equal?` algorithm except that it performs a pre-check identical to the one performed by the `precheck/uf-equal?` algorithm described in Section 4. The purpose of the pre-check is to allow us to start the interleaving in "slow" mode to avoid the potential for the $k_0 l$ additional calls to `slow?` that can occur when the `interleaved?` algorithm starts out in "fast" mode, as described at the end of Section 5.1.

The `precheck/interleave-equal?` procedure defined below implements the algorithm.

```
(define k0 400)
(define kb -40)
(define (precheck/interleave-equal? x y)
  (let ([k (pre? x y k0)])
    (and k (or (> k 0) (interleave? x y 0)))))
```

The `pre?` procedure is the same as before, but the version of `interleave?` used by this algorithm (not shown) differs from the

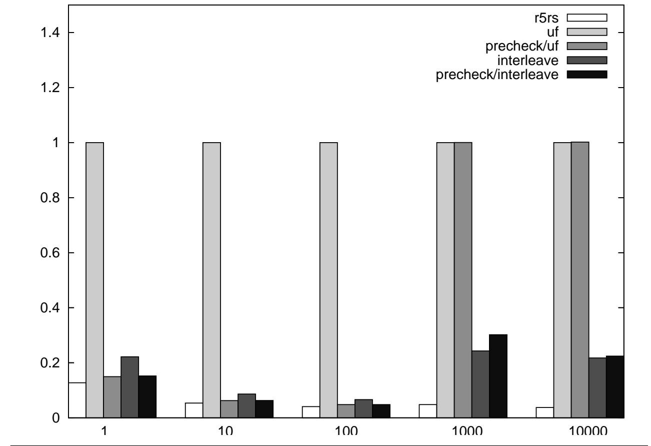

Figure 8. Lists of various lengths

one shown in Figure 7 in that the hash table is created immediately upon entry to `interleave?`. There is no benefit for delaying its creation in this case, since the initial value of `k` causes the algorithm to start out in "slow" mode, where the hash table would be created immediately anyway.

6.1 Analysis

The bound on the number of calls made during the interleaved portion of this algorithm (starting in "slow" mode) is described in Section 5.1; the pre-check adds only constant overhead.

6.2 Benchmarks

Figures 8 through 12 compare the algorithms defined in this paper on various shapes and sizes of acyclic inputs. Figures 13 and 14 compare all but `r5rs-equal?` on cyclic inputs. All of the run times are normalized to the `uf-equal?` run times, which we consider the baseline for our comparison because it is the most straightforward of the algorithms that work for all inputs.

While `precheck/interleaved-equal?` outperforms straight `interleaved-equal?` in many cases, and its worst-case behavior may be better, it is sometimes slower than `interleaved-equal?`, generally for larger inputs for which the pre-check does not pay off.

In general, it appears that `precheck/interleaved-equal?` achieves our goal of finding an algorithm that is within a small constant factor of the faster of the simple tree equality and straight union-find algorithms. This can be seen most clearly in Figure 15. which shows the worst-case behavior for each of the tested algorithms on each of the benchmarks. Each individual bar within the set of bars for each algorithm shows the largest observed ratio between the run time of the algorithm and the minimum of the run times of the `r5rs-equal?` and `uf-equal?` algorithms on one benchmark, with the benchmark that tests lists of various lengths first followed by inverted lists, balanced binary trees, degenerate dags, random dags, random graphs, and union-exercising graphs. Bars for the last two are not shown for `r5rs-equal?`, since it cannot handle arbitrary graphs, and the fourth bar for degenerate dags understates the worst case for that algorithm, since it was not tested on degenerate dags of depth 32 or 64.

As the figure illustrates, our algorithm is at worst about a factor of six slower than the minimum of the simple tree-equality and straight union-find algorithms. As can be seen in Figure 8, this worst-case occurs when it is compared with the simple tree equality algorithm on long lists. Each of the other algorithms exhibits poorer worst-case behavior.

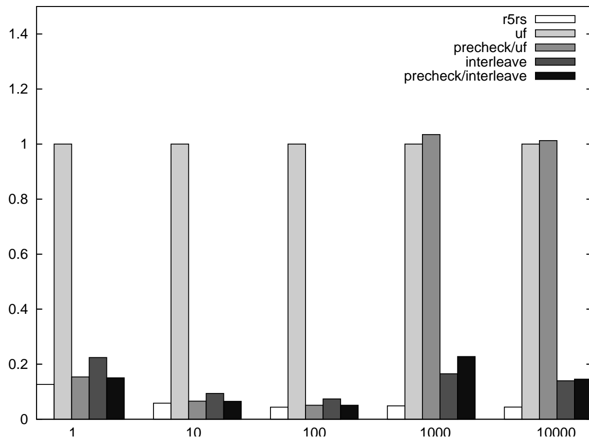

Figure 9. Inverted lists of various lengths

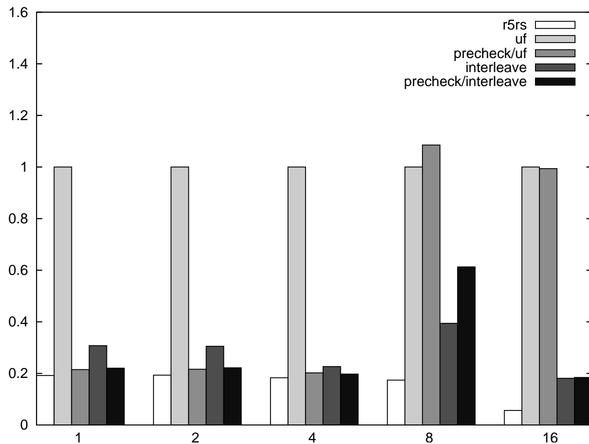

Figure 10. Balanced binary trees of various depths

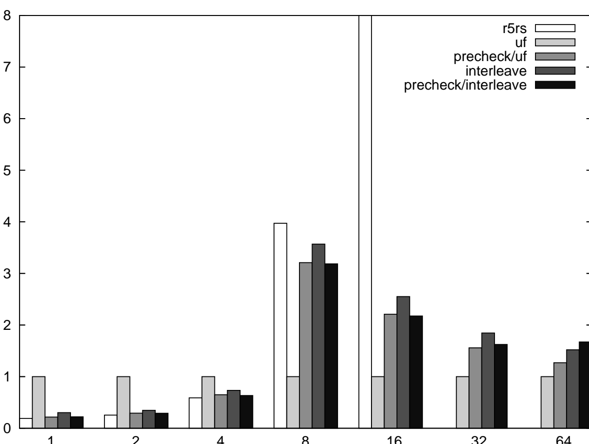

Figure 11. Degenerate dags

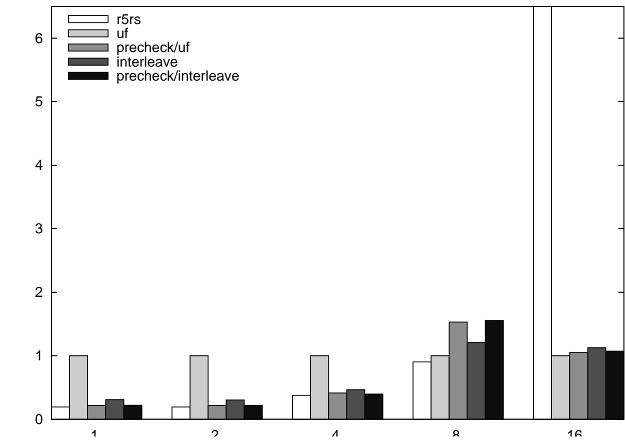

Figure 12. Random dags

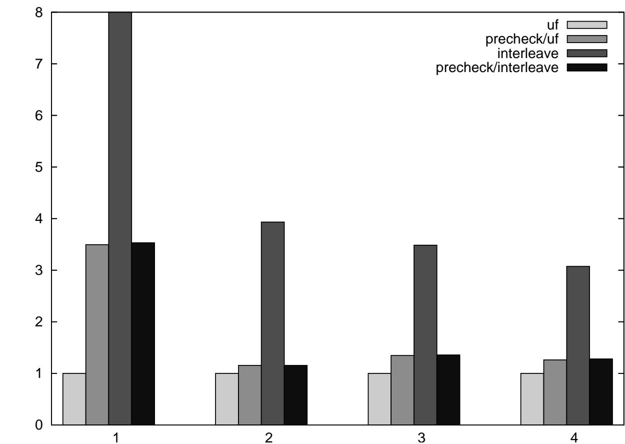

Figure 13. Random graphs

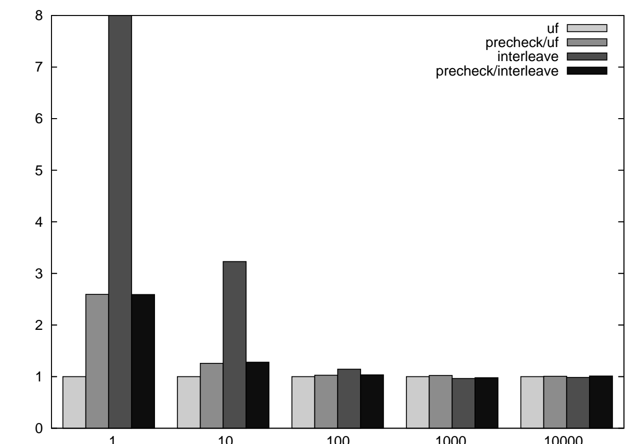

Figure 14. Union-exercising graphs

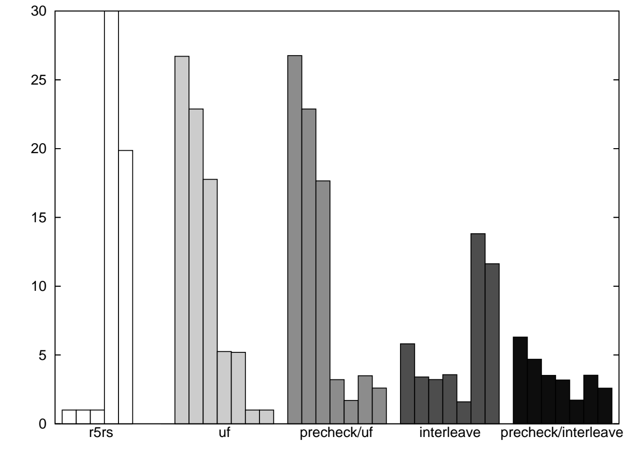

Figure 15. Worst-case performance of the algorithms.

7. Related work

Will Clinger proposed the R6RS `equal?` procedure in "SRFI 85: Recursive Equivalence Predicates," which refers to the predicate as `equiv?` (2006). Our `precheck/uf-equal?` procedure is similar to his corrected reference implementation (2007) in that both perform a bounded tree-equivalence pre-check followed by a DFA-equivalence check. The reference implementation does not interleave the tree-equality and DFA-equivalence checks, as our penultimate and final algorithms do.

Although the SRFI makes no specific claims about performance, it is worth noting a few other problems that can cause it to run multiple orders of magnitude slower than our algorithm and thus make it unsuitable for use in an actual Scheme implementation. First, the reference implementation sets the pre-check bound at 100000, while we set the bound at 400; this makes the reference implementation somewhat faster for lists and trees with between 400 and 100000 nodes but slower for larger lists and trees and much slower for small cyclic structures. Second, the reference implementation pre-check does not decrement the bound for every vector element; thus the effective bound on the pre-check is actually 100000 times the length of the longest vector, which is bad if the inputs contain large vectors. Third, the reference implementation touches every element of a merged equivalence class when two nonempty equivalence classes are combined, which introduces a linear factor into each such operation and can thus cause `equal?` to exhibit quadratic behavior. A comment in the reference implementation alludes to the third problem.

The reference implementation avoids entering some nodes into its graph by looking ahead at the subnodes to see if they can be compared without recursion. We tried this and found it not only complicated the code but also performed worse even on inputs contrived to make the look-ahead succeed, presumably due to the additional checks required.

As an aside, the SRFI document claims that `equal?` (`equiv?`) structures cannot be distinguished by mutation, but as we show in Section 2, this is not the case, i.e., some `equal?` structures *can* be distinguished by parallel mutations.

8. Conclusions

Any of the algorithms presented in this paper might be suitable for a given application's mix of inputs. A general-purpose library equivalence predicate, like Scheme's `equal?`, must, however, be suitable (if not optimal) for all inputs. The final algorithm presented in Section 6 appears to satisfy this requirement.

For the interleaved versions of the algorithm, it is worth considering placing an effective bound on the m factor in the number of calls that `fast?` can make to `slow?` by calling `union-find` even in "fast" mode after processing a certain number of vector elements, if the algorithm has not yet found an unequal element. Based on our current (loose) analysis, this would reduce the cost of the algorithms when large vectors are present in the input, but whether this would be beneficial in practice is not clear.

We leave for future work a proof of correctness and the establishment of tighter bounds for the final algorithm. It would also be nice to benchmark real programs that use `equal?` for cyclic inputs, but no such programs are likely to exist until R6RS achieves wider adoption.

Acknowledgments

Aziz Ghuloum observed that only one cell need be allocated for each pair of nodes being equivalenced when neither has been seen before. This observation led directly to the further observation that the existing cell can be reused when only one of a pair of nodes is seen for the first time. Comments by the anonymous reviewers led to several improvements in the presentation.

References

William D. Clinger. SRFI 85: Recursive equivalence predicates, March 2006. URL `http://srfi.schemers.org/srfi-85/`.

William D. Clinger. SRFI 85: Recursive equivalence predicates (corrected reference implementation), December 2007. URL `http://srfi.schemers.org/srfi-85/post-mail-archive/msg00001.html`.

Zvi Galil and Giuseppe F. Italiano. Data structures and algorithms for disjoint set union problems. *ACM Comput. Surv.*, 23(3):319–344, 1991. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/116873.116878.

Benrard A. Galler and Michael J. Fisher. An improved equivalence algorithm. *Commun. ACM*, 7(5):301–303, 1964. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/364099.364331.

Abdulaziz Ghuloum and R. Kent Dybvig. Generation-friendly eq hash tables. In *2007 Workshop on Scheme and Functional Programming*, pages 27–35, 2007. URL `http://sfp2007.ift.ulaval.ca/programme.html`.

John E. Hopcroft and R. M. Karp. A linear algorithm for testing equivalence of finite automata. Technical Report 71–114, Cornell University, Ithaca, NY, 1971. URL `http://ecommons.library.cornell.edu/handle/1813/5958`.

Richard Kelsey, William Clinger, and Jonathan Rees (eds.). Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998. Also appears in ACM SIGPLAN Notices 33(9), September 1998.

Michael Sperber, R. Kent Dybvig, Matthew Flatt, and Anton van Straaten (eds.). Revised^6 report on the algorithmic language Scheme, September 2007. URL `http://www.r6rs.org/`.

Robert E. Tarjan and Jan van Leeuwen. Worst-case analysis of set union algorithms. *J. ACM*, 31(2):245–281, 1984. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/62.2160.

Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, 1975. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/321879.321884.