Verified Analysis of List Update Algorithms*

Maximilian P.L. Haslbeck[1] and Tobias Nipkow[2]

1,2 Technische Universität München

Abstract

This paper presents a machine-verified analysis of a number of classical algorithms for the list update problem: 2-competitiveness of move-to-front, the lower bound of 2 for the competitiveness of deterministic list update algorithms and 1.6-competitiveness of the randomized COMB algorithm, the best randomized list update algorithm known to date. The analysis is verified with help of the theorem prover Isabelle; some low-level proofs could be automated.

1998 ACM Subject Classification F.3.1 Specifying and Verifying Programs and F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases Program Verification, Algorithm Analysis, Online Algorithms

Digital Object Identifier 10.4230/LIPIcs.xxx.yyy.p

1 Introduction

Interactive theorem provers have been applied to deep theorems in mathematics [9, 10] or fundamental software components [20, 22] but hardly to quantitative algorithm analysis. This paper demonstrates that nontrivial results from that area are amenable to verification (which for us always means "interactive") by analyzing the best known deterministic and randomized online algorithms for the list update problem with the help of the theorem prover Isabelle/HOL [25, 26]. Essentially, this paper formalizes the main results of the first two chapters of the classic text by Borodin and El-Yaniv [6]: 2-competitiveness of move-to-front (MTF), the lower bound of 2 for the competitiveness of all deterministic list update algorithms and 1.6-competitiveness of COMB [2], the best randomized online list update algorithm known to date. For reasons of space we are forced to refer the reader to the online formalization [12] (15600 lines) for many of the definitions and all of the formal proofs.

The list update problem is a simple model to study the competitive analysis of online algorithms (where requests arrive one by one) compared to offline algorithms (where the whole sequence of requests is known upfront). In the simplest form of the problem we are given a list of elements that can only be searched sequentially from the front and each request asks if some element is in that list. In addition to searching for the element the algorithm may rearrange the list by swapping any number of adjacent elements to improve the response time for future requests. One is usually not interested in the offline algorithm (the problem is NP-hard [3]) but merely uses it as a benchmark to compare online algorithms against.

This paper advocates to extend verification of algorithms from functional correctness to quantitative analysis. There are a number of examples of such verifications for classical algorithms, but this is the first verified analysis of any online algorithm. Our verified proof of the 1.6-competitiveness of COMB appears to be one of the most complex verified analyses of a randomized algorithm to date. Our paper should be read as a contribution to the formalization of computer science foundations, here quantitative algorithm analysis.

* Supported by DFG Koselleck grant NI 491/16-1

Conference title on which this volume is based on.
Editors: Billy Editor and Bill Editors; pp. 1–14

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

It should be noted that although our verification is interactive, some tedious low-level proofs could be automated (see the verifification of algorithm BIT in Section 4.6).

The paper is structured as follows: Section 2 explains our notation. Sections 3 and 4 roughly follow chapters 1 and 2 of [6], with some omissions: Section 3 formalizes deterministic list update algorithms, analyzes MTF and proves a lower bound. Section 4 formalizes randomized list update algorithms, proves two analysis techniques (list factoring and phase partioning), and analyzes the algorithms BIT, TS and finally COMB.

1.1 Related Work

This work grew out of an Isabelle-based framework for verified amortized analysis applied to classical data structures like splay trees [24]. Charguéraud and Pottier [7] verified the almost-linear amortized complexity of Union-Find in Coq. The verification of randomized algorithms was pioneered by Hurd *et al.* [15, 16, 17] who verified the Miller-Rabin probabilistic primality test and part of Rabin's probabilistic mutual exclusion algorithm. Barthe *et al.* (e.g. [5]) verify probabilistic security properties of cryptographic constructions.

An orthogonal line of research is the automatic resource bound analysis of (comparatively simple) deterministic functional or imperative programs, e.g., [13].

The list update problem is still an active area of research [23]; for a survey see [19].

2 Notation

Isabelle's higher-order logic is a simply typed λ-calculus: function application is written $f\ x$ and $g\ x\ y$ rather than $f(x)$ and $g(x,y)$; binary functions usually have type $A \to B \to C$ instead of $A \times B \to C$, and analogously for $n$ary functions; $\lambda x.\ t$ is the function that maps argument x to result t. The notation $t :: \tau$ means that term t has type τ.

The type of lists over a type α is $\alpha\ list$. The empty list is $[]$, prepending an element x in front of a list xs is written $x \cdot xs$ and appending two lists is written $xs\ @\ ys$. The length of xs is $|xs|$. Function set converts a list into a set. The predicate $distinct\ xs$ expresses that there are no duplicates in xs. The ith element of xs (starting at 0) is xs_i. By $index\ xs\ x$ we denote the index (starting at 0) of the first occurrence of x in xs; if x does not occur in xs then $index\ xs\ x = |xs|$. If x occurs before y in xs we write $x < y\ \mathsf{in}\ xs$:

$$x < y\ \mathsf{in}\ xs \longleftrightarrow index\ xs\ x < index\ xs\ y \wedge y \in set\ xs$$

The condition $x \in set\ xs$ is implied by the right-hand-side.

Given two lists xs and ys, we call a pair $(x,\ y)$ an *inversion* if x occurs before y in xs but y occurs before x in ys. The set of inversions is defined like this:

$$Inv\ xs\ ys = \{(x,\ y) \mid x < y\ \mathsf{in}\ xs \wedge y < x\ \mathsf{in}\ ys\}$$

Given a list xs and two elements x and y, let xs_{xy} denote the projection of xs on x and y, i.e., the result of deleting from xs all elements other than x and y.

Note that the LaTeX presentations of definitions and theorems in this paper are generated by Isabelle from the actual definitions and theorems. To increase readability we employed Isabelle's pretty-printing facilities to emulate the notation in [6]. This has to be taken into account when comparing formulas in the paper with the actual Isabelle text [12].

2.1 Probability Mass Functions

Type α *pmf* of *probability mass functions* [14, §4] represent distributions of discrete random variables on a type α. Function $set_{pmf}\ D$ denotes the support set of the distribution D and $pmf\ D\ e$ denotes the probability of element e in the distribution D.

▶ **Example 1.** Our background theory defines the Bernoulli distribution $bernoulli_{pmf}$, a *pmf* on the type *bool* which satisfies amongst others the following properties:

$set_{pmf}\ (bernoulli_{pmf}\ p) \subseteq \{True, False\}$
$pmf\ (bernoulli_{pmf}\ (1/2))\ x = 1/2$
$0 \leq p \wedge p \leq 1 \Longrightarrow pmf\ (bernoulli_{pmf}\ p)\ True = p$

Furthermore the monadic operators $bind_{pmf} :: \alpha\ pmf \to (\alpha \to \beta\ pmf) \to \beta\ pmf$ and $return_{pmf} :: \alpha \to \alpha\ pmf$, as well as the operator $map_{pmf} :: (\alpha \to \beta) \to \alpha\ pmf \to \beta\ pmf$ are defined. With the help of these functions more complex *pmf*s can be synthesized from easier ones. To demonstrate how to work with *pmf*, we define $bv\ n$ the uniform distribution over bit vectors of length n recursively.

```
bv 0 = return_pmf []
bv (n + 1) = do {
  xs ← bv n;
  x ← bernoulli_pmf (1/2);
  return_pmf (x · xs)
}
```

This is an example of probabilistic functional programming [8] with the help of Haskell's *do*-notation (which is just syntax for the *bind* operator).

We further define the simple function *flip* $i\ b$ that flips the ith bit of the bit vector b. When we apply *flip* i to every element of the probability distribution $bv\ n$ we obtain again the same probability distribution: $map_{pmf}\ (flip\ i)\ (bv\ n) = bv\ n$.

3 List Update: Deterministic Algorithms

3.1 Online and Offline Algorithms

We need to define formally what online and offline algorithms are. Our formalization is similar to request-answer systems [6] but we clarify the role of the initial state and replace a history-based formalization with an equivalent state-based one. Everything is parameterized by a type of *requests* $\mathcal{R}$, a type of *answers* $\mathcal{A}$, a type of *states* $\mathcal{S}$, a type of *internal states* $\mathcal{I}$, and by the following three functions: $step :: \mathcal{S} \to \mathcal{R} \to \mathcal{A} \to \mathcal{S}$, $t :: \mathcal{S} \to \mathcal{R} \to \mathcal{A} \to \mathbb{N}$ and $wf :: \mathcal{S} \to \mathcal{R}\ list \to bool$. Answers describe how the system state changes in reaction to a request: $step\ s\ r\ a$ is the new state after r has been answered by a and $t\ s\ r\ a$ is the time or cost of that step. The predicate wf defines the well-formed request sequences depending on the initial state. An offline algorithm is a function of type $\mathcal{S} \to \mathcal{R}\ list \to \mathcal{A}\ list$ that computes a list of answers from a start state and a list of requests. An online algorithm is a pair (ι, δ) of an initialization function $\iota :: \mathcal{S} \to \mathcal{I}$ that yields the initial internal state, and a transition function $\delta :: \mathcal{S} \times \mathcal{I} \to \mathcal{R} \to \mathcal{A} \times \mathcal{I}$ that yields the answer and the new internal state. In the sequel assume $A = (\iota, \delta)$.

Note that we separate the problem specific states $\mathcal{S}$ and step function *step* from the algorithm specific internal states $\mathcal{I}$ and transition function δ to obtain a modular framework. Elements of type $\mathcal{S} \times \mathcal{I}$ are called *configurations*.

In this context we define the following functions:

- *Step A* (s, i) $r = (\mathsf{let}\ (a, i') = \delta\ (s, i)\ r\ \mathsf{in}\ (step\ s\ r\ a, i'))$ transforms one configuration into the next by composing δ and *step*.
- $T\ s\ rs\ as$ is the time it takes to process a request list rs and corresponding answer list as (of the same length) starting from state s via a sequence of *step*s.
- $OPT[s;rs] = Inf\ \{T\ s\ rs\ as \mid |as| = |rs|\}$ where *Inf* is the infimum, is the time of the optimal offline algorithm servicing rs starting from state s.
- $A[s;rs]$ is the time an online algorithm A takes to process a request list rs via a sequence of *Step*s starting from configuration $(s, \iota\ s)$.
- Algorithm A is deemed c-competitive, if its cost is at most c times the optimal, formally: $compet\ A\ c\ S \longleftrightarrow (\forall\, s{\in}S.\ \exists\, b{\geq}0.\ \forall\, rs.\ wf\ s\ rs \longrightarrow A[s;rs] \leq c * OPT[s;rs] + b)$ expresses that the online algorithm A is c-competitive on the set of initial states S and well-formed request sequences.

3.2 On/Offline Algorithms for List Update

The list update problem consists of maintaining an unsorted list of elements while the cost of servicing a sequence of requests has to be minimized. Each request asks to search an element sequentially from the front of the list. A penalty equal to the position of the requested element has to be paid. In order to minimize the cost of future requests the requested element can be moved further to the front of the list by a *free exchange*. Any other swap of two consecutive elements in the list costs one unit and is called a *paid exchange*.

We instantiate our generic model as follows. Given a type of elements α, states are of type $\alpha\ list$, requests of type α, and answers are of type $\mathbb{N} \times \mathbb{N}\ list$. An answer $(n, [n_1,\ldots,n_k])$ means that the requested element is moved n positions to the front at not cost (*free exchanges*) after swapping the elements at index n_i and $n_i + 1$ $(i = k,\ldots,1)$ at the cost of 1 per exchange (*paid exchanges*). Based on two functions $mtf_2 :: \mathbb{N} \to \alpha \to \alpha\ list \to \alpha\ list$ and $swaps :: \mathbb{N}\ list \to \alpha\ list \to \alpha\ list$ we define $step\ s\ r\ (k, ks) = mtf_2\ k\ r\ (swaps\ ks\ s)$ and $t\ s\ r\ (k, ks) = index\ (swaps\ ks\ s)\ r + 1 + |ks|$. There is no need for paid exchanges after the move to front because they can be performed at the beginning of the next step. Corner cases: $mtf_2\ k\ x\ xs$ does nothing if $x \notin set\ xs$ and moves x to the front if $x \in set\ xs$ and $index\ xs\ x < k$; $swaps\ [n_1,\ldots,n_k]\ xs$ ignores indices n_i such that $|xs| \leq n_i + 1$. We focus on the *static list model* by instantiating the well-formedness predicate *wf* by the predicate *static* defined by $static\ s\ rs \longleftrightarrow set\ rs \subseteq set\ s$.

Sleator and Tarjan [30], who introduced the list update problem, claimed (their Theorem 3) that offline algorithms do not need paid exchanges. Later Reingold and Westbrook [28] refuted this and proved the opposite: offline algorithms need only paid exchanges. This may also be considered as an argument in favour of verification.

3.3 Move to Front

The archetypal online algorithm is *move to front* (MTF): when an element is requested, it is moved to the front of the list, without any paid exchanges. MTF needs no internal state and thus we identify $\mathcal{I}$ with the unit type that contains only the dummy element $()$. The pair $MTF = (\lambda\_.\ (), \lambda(s, i)\ r.\ ((|s| - 1, []), ()))$ is an online algorithm in the sense of our above model.

Now we verify Sleator and Tarjan's result [30] that MTF is at most 2-competitive, i.e., at most twice as slow as any offline algorithm. We are given an initial state s of distinct

elements, a request sequence rs and an answer sequence as computed by some offline algorithm such that $|as| = |rs|$. The state of MTF after servicing the requests $rs_0, \ldots, rs_{n-1}$ is denoted by $s_{mtf}\ n$, the cost of executing step n is denoted by $t_{mtf}\ n$. The state after answering the requests $rs_0, \ldots, rs_{n-1}$ with the answers $as_0, \ldots, as_{n-1}$ is denoted by $s_{off}\ n$, the cost $t_{off}\ n$ of executing as_n is broken up as follows: $c_{off}\ n$ is the cost of finding the requested element rs_n and $p_{off}\ n$ ($f_{off}\ n$) is the number of the paid (free) exchanges. Following [30] we define the potential as the number of inversions that separates MTF from the offline algorithm ($\Phi\ n = |Inv\ (s_{off}\ n)\ (s_{mtf}\ n)|$) and prove the key lemma

▶ **Lemma 2.** $t_{mtf}\ n + \Phi\ (n+1) - \Phi\ n \leq 2 * c_{off}\ n - 1 + p_{off}\ n - f_{off}\ n$

Its proof is a little bit tricky and requires a number of lemmas about inversions that formalize what is often given as a pictorial argument. By telescoping and defining $T_{mtf}\ n = (\sum_{i<n} t_{mtf}\ i)$ we obtain Sleator and Tarjan's Theorem 1:

▶ **Theorem 3.** $T_{mtf}\ n \leq (\sum_{i<n} 2 * c_{off}\ i + p_{off}\ i - f_{off}\ i) - n$

It follows that $T_{mtf}\ n \leq (2 - 1/|s|) * T_{off}\ n$, where $T_{off}\ n = (\sum_{i<n} t_{off}\ i)$, provided $s \neq []$ and $\forall\, i{<}n.\ rs_i \in set\ s$. By definition of OPT we obtain the following corollary [6]:

▶ **Corollary 4.** $s \neq [] \wedge distinct\ s \wedge set\ rs \subseteq set\ s \Longrightarrow MTF[s;rs] \leq (2 - 1/|s|) * OPT[s;rs]$

Because *compet* is defined relative to *wf* and we have instantiated *wf* with the static list model (which implies $set\ rs \subseteq set\ s$), we obtain the following compact corollary:

▶ **Corollary 5.** $compet\ MTF\ 2\ \{s \mid distinct\ s\}$

The assumption $s \neq []$ has disappeared because we no longer divide by $|s|$.

3.4 A Lower Bound

The following lower bound for the competitiveness of any online algorithm is due to Karp and Raghavan [18]:

▶ **Theorem 6.** $compet\ A\ c\ \{xs \mid |xs| = l\} \wedge l \neq 0 \wedge 0 \leq c \Longrightarrow 2 * l/(l+1) \leq c$

The corresponding Theorem 1.2 in [6] is incorrect because it asserts that every online algorithm is c-competitive for some constant c, but this is not necessarily the case if the algorithm uses paid exchanges. In the proof it is implicitly assumed there are no paid exchanges when claiming "The total cost incurred by the online algorithm is clearly $l * n$".

Our proof roughly follows the original sketch [18, p. 302]. Let $A = (\iota, \delta)$ be an online algorithm. We define a cruel request sequence that always requests the last element in the state of A, given a start configuration and length:

$cruel\ A\ c\ 0 = []$
$cruel\ A\ (s, i)\ (n+1) = last\ s \cdot cruel\ A\ (Step\ A\ (s, i)\ (last\ s))\ n$

We also define a cruel offline adversary for A that first sorts the state in decreasing order of access frequency in the cruel sequence and does nothing afterwards:

$adv\ A\ s\ rs =$
$(\mathsf{if}\ rs = []\ \mathsf{then}\ []$
$\ \mathsf{else\ let}\ crs = cruel\ A\ (Step\ A\ (s, \iota\ s)\ (last\ s))\ (|rs| - 1)$
$\quad \mathsf{in}\ (0, sort\_sws\ (\lambda x.\ |rs| - 1 - count\_list\ crs\ x)\ s) \cdot replicate\ (|rs| - 1)\ (0, []))$

For the first step *sort_sws* computes the necessary paid exchanges according to the frequency count computed by *count_list* from the cruel sequence *crs*; the remaining steps are do-nothing answers (0, []).

For the analysis let $A[s;n]$ (resp. $C[s;n]$) be the time A (resp. the adversary *adv* A) requires to answer the cruel request sequence of length $n+1$ starting in state s. In the rest of the analysis assume $l \neq 0$. First we prove $C[s;n] \leq a + (l+1)*n \ div\ 2$ where $a = l^2 + l + 1$. The cost of the first step of the cruel adversary (searching and sorting) is at most a, and the cost of searching for n requested items is at most $(l+1) * n \ div\ 2$. We obtain the latter bound by writing the cost as a sum of terms $i * f_i$ where f_i is the number of requests of the ith item in the sorted list, $i = 0,\ldots,l-1$. Because the f_i decrease with increasing i, the result follows by Tchebychef's inequality [11, 2.17] that the mean of the product is at most the product of the means. The cost $A[s;n]$ is $(n+1) * l$ if there are no paid exchanges and thus $(n+1)*l \leq A[s;n]$. Combining this with the upper bound for $C[s;n]$ we obtain $2*l/(l+1) \leq A[s;n]/(C[s;n]-a)$ for all large enough n. From the c-competitiveness of A we obtain a constant b such that $(A[s;n]-b)/C[s;n] \leq c$. The additive constants are typically ignored, in which case $2*l/(l+1) \leq c$ is immediate; otherwise it takes a bit of limit reasoning.

4 List Update: Randomized Algorithms

4.1 Randomized Online Algorithms

Now we generalize our model of online algorithms of §3.1 to randomized online algorithms. We view a randomized algorithm not as a distribution of deterministic algorithms, but an algorithm working on a distribution of configurations. The monad described in §2.1 suggests this view and enables us to formulate randomized algorithms concisely. Furthermore we expect that proofs can be mechanized more easily that way.

The initialization function now not only yields one initial internal state but a distribution over the type of internal states: $\mathcal{S} \to \mathcal{I}\ pmf$. Similarly the transition function of randomized online algorithms has the type $(\mathcal{S} \times \mathcal{I})\ pmf \to \mathcal{R} \to (\mathcal{A} \times \mathcal{I})\ pmf$. We now give definitions to analogous functions like in the deterministic case, overloading is unproblematic because deterministic online algorithms can be embedded canonically into randomized ones. Whether $A = (\iota, \delta)$ is a randomized or deterministic online algorithm will be clear from the context.

- A compound *Step* on configurations consists of two steps: first the online algorithm will produce a distribution of answer and new internal states, then the problem (*step*) will process the answer and yield a new configuration distribution:
  Step $A\ r\ (s, i) =$ *do* {
  $(a, is') \leftarrow \delta\ (s, i)\ r$;
  $return_{pmf}\ (step\ s\ r\ a, is')$
  }
- *config* $A\ s\ rs$ formalizes the execution of A by denoting the distribution of configurations after servicing the request sequence *rs* starting in state s.
- $A[s;rs]$ denotes the expected time A takes to process a request list *rs* via a sequence of *Step*s starting from the distribution of configurations obtained by combining s with $\iota\ s$.
- *compet* $A\ c\ S_0 \longleftrightarrow (\forall\, s{\in}S_0.\ \exists\, b{\geq}0.\ \forall\, rs.\ wf\ s\ rs \longrightarrow A[s;rs] \leq c * OPT[s;rs] + b)$ expresses that A is c-competitive against an oblivious adversary on the set of initial states S_0.
- The embedding $embed\ (\iota, \delta) = (\lambda s.\ return_{pmf}\ (\iota\ s), \lambda s\ r.\ return_{pmf}\ (\delta\ s\ r))$ for example preserves *compet* $A\ c\ S_0 \longleftrightarrow$ *compet* $(embed\ A)\ c\ S_0$ for an deterministic algorithm A.

4.2 BIT

In this section we study a simple randomized algorithm for the list update problem called BIT due to Reingold and Westbrook [28]. BIT breaks the 2-competitive barrier for deterministic online algorithms (Theorem 6): we will prove that BIT is 1.75-competitive.

BIT keeps for every element x in the list a bit $b(x)$. The $b(x)$ are initialized randomly, independently and uniformly. When some x is requested, its bit $b(x)$ is complemented; then, if $b(x)$, x is moved to the front. Formally, the internal state is a pair $(b,\ s_0) :: \mathit{bool\ list} \times \alpha\ \mathit{list}$ where s_0 is the initial list and $|b| = |s_0|$. The informal $b(x)$ becomes $b_{\mathit{index}\ s_0\ x}$.

▶ **Definition 7** (BIT). $BIT = (\iota_{BIT},\ \delta_{BIT})$ where $\iota_{BIT}\ s_0 = \mathit{map}_{pmf}\ (\lambda b.\ (b,\ s_0))\ (\mathit{bv}\ |s_0|)$
$\delta_{BIT}\ (s,\ b,\ s_0)\ x = \mathit{return}_{pmf}\ ((\mathit{if}\ b_{\mathit{index}\ s_0\ x}\ \mathit{then}\ 0\ \mathit{else}\ |s|,\ []),\ \mathit{flip}\ (\mathit{index}\ s_0\ x)\ b,\ s_0)$

Function ι_{BIT} generates a random bit vector (for bv see §2.1) of length $|s_0|$ and pairs it with s_0. Function δ_{BIT} is given a configuration $(s,\ b,\ s_0)$ and a request x, *flip*s x's bit in b, and if it was set, the answer is move-to-front, otherwise it is do-nothing. BIT is a barely random algorithm: only the initialization function is randomized, the transition function is deterministic.

▶ **Theorem 8** ([6, Theorem 2.1]). $\mathit{compet}\ BIT\ (7/4)\ \{\mathit{init} \mid \mathit{init} \neq [] \wedge \mathit{distinct}\ \mathit{init}\}$

The proof of this theorem is similar to the proof that MTF is 2-competitive: the potential function involves weighted inversions. Therefore we do not discuss the details (see [12]).

4.3 List Factoring

The list factoring method enables us to reduce competitive analysis of list update algorithms to lists of size two. The main idea is to modify the cost measure. The cost of accessing some element will be the number of elements that precede it. We attribute a "blocking cost" of 1 to every element that precedes the requested element. For the requested element and all following the blocking cost is 0. In summary, the cost of accessing the ith item is no longer $i + 1$ but i. This is called the *partial* cost model, in contrast to the *full* cost model. Costs in the partial cost model are marked with an asterisk; for example, $t^*\ s\ r\ a = t\ s\ r\ a - 1$. Upper bounds on the competitive ratio in the partial cost model are also upper bounds on the competitive ratio in the full cost model [6, Lemma 1.3]

Let $A^*[s;rs](x;i)$ denote the expected blocking cost of element x in the ith step of the execution of algorithm A on the request sequence rs starting from state s. The notations $\sum x{\in}M_x.\ m_x$ and $\sum x \mid P_x.\ m_x$ denote summations over a set or restricted by a predicate. We will need a set of all pairs $(x,\ y) \in M \times M$ of distinct elements where only one of the two pairs $(x,\ y)$ and $(y,\ x)$ is included. For simplicity we assume M is linearly ordered and define $\mathit{Diff}_2\ M = \{(x,\ y) \mid x \in M \wedge y \in M \wedge x < y\}$.

Now consider the cost incurred by an online algorithm without paid exchanges:

$$\begin{aligned}
A^*[s;rs] = &\ \textstyle\sum_{i<|rs|} \sum x{\in}\mathit{set}\ s.\ A^*[s;rs](x;i)\\
= &\ \textstyle\sum x{\in}\mathit{set}\ s.\ \sum_{i<|rs|} A^*[s;rs](x;i)\\
= &\ \textstyle\sum x{\in}\mathit{set}\ s.\ \sum y{\in}\mathit{set}\ s.\ \sum i \mid i < |rs| \wedge rs_i = y.\ A^*[s;rs](x;i)\\
= &\ \textstyle\sum (x,\ y){\in}\{(x,\ y) \mid x \in \mathit{set}\ s \wedge y \in \mathit{set}\ s \wedge x \neq y\}.\\
&\ \textstyle\sum i \mid i < |rs| \wedge rs_i = y.\ A^*[s;rs](x;i)\\
= &\ \textstyle\sum (x,\ y){\in}\mathit{Diff}_2\ (\mathit{set}\ s).\\
&\ \textstyle\sum i \mid i < |rs| \wedge (rs_i = y \vee rs_i = x).\ A^*[s;rs](y;i) + A^*[s;rs](x;i)
\end{aligned}$$

The inner summation of the last expression is abbreviated by $A^*_{xy}[s;rs]$ and interpreted as the expected cost generated by x blocking y or vice versa. We can condense the above derivation:

▶ **Lemma 9** ([6, Equation (1.4)]). $A^*[s;rs] = (\sum (x,\ y){\in}\mathit{Diff}_2\ (\mathit{set}\ s).\ A^*_{xy}[s;rs])$

As the value of any summand on the right hand side only depends on the relative order of x and y during the execution and the relative order may only change when x or y are requested (as we disallowed paid exchanges for now) one might think that this is exactly the same as the cost incurred by the algorithm when run on the projected request list rs_{xy} starting from the projected initial state s_{xy}. While this is not the case in general, this equality yields a good characterization of a subset of all list update algorithms and is thus referred to as the pairwise property. Most of the list update algorithms studied in the literature share this property, including MTF, BIT, TS and COMB (see Table 1 in [19] where "projective" means "pairwise").

▶ **Definition 10** (pairwise property). Algorithm A satisfies the *pairwise property* if
$\mathit{distinct}\ s \wedge \mathit{set}\ rs \subseteq \mathit{set}\ s \wedge (x,\ y) \in \mathit{Diff}_2\ (\mathit{set}\ s) \Longrightarrow A^*[s_{xy};rs_{xy}] = A^*_{xy}[s;rs]$

With a similar development as for Lemma 9 we can split the costs of OPT^* into the costs that are incurred by each pair of elements: first the costs incurred by blocking each other and second the number of paid exchanges that change the elements' relative order:

▶ **Theorem 11** ([6, Equation (1.8)]).
$OPT^*[s;rs] = (\sum (x,\ y){\in}\mathit{Diff}_2\ (\mathit{set}\ s).\ OPT^*_{xy}[s;rs] + OPT^*_{P;xy}[s;rs])$

If we consider the summand for a specific pair (x,y), we see that it gives rise to an (not necessarily optimal) algorithm servicing the projected request sequence rs_{xy}. Thus this term is an upper bound for the optimal cost for servicing rs_{xy}. This fact is established by constructing this projected algorithm and showing that its cost is equal to the right-hand side of the inequality:

▶ **Lemma 12** ([6, Equation (1.7)]). $OPT^*[s_{xy};rs_{xy}] \leq OPT^*_{xy}[s;rs] + OPT^*_{P;xy}[s;rs]$

Now we are in the position to describe the *list factoring technique*:

▶ **Theorem 13** ([6, Lemma 1.2]). *Let A be an algorithm that does not use paid exchanges, satisfies the pairwise property and is c-competitive for lists of length 2. Then A is c-competitive for arbitrary lists.*

Proof.
$$
\begin{aligned}
A^*[s;rs] &\overset{\text{Lemma 9}}{=} \textstyle\sum (x,\ y){\in}\mathit{Diff}_2\ (\mathit{set}\ s).\ A^*_{xy}[s;rs] \\
&\overset{\text{pairwise}}{=} \textstyle\sum (x,\ y){\in}\mathit{Diff}_2\ (\mathit{set}\ s).\ A^*[s_{xy};rs_{xy}] \\
&\overset{c\text{-compet.}}{\leq} \textstyle\sum (x,\ y){\in}\mathit{Diff}_2\ (\mathit{set}\ s).\ c * OPT^*[s_{xy};rs_{xy}] + b \\
&\overset{\text{Lemma 12}}{\leq} c * (\textstyle\sum (x,\ y){\in}\mathit{Diff}_2\ (\mathit{set}\ s).\ OPT^*_{xy}[s;rs] + OPT^*_{P;xy}[s;rs]) + \\
&\qquad b * (|s| * (|s| - 1))/2 \\
&\overset{\text{Lemma 11}}{=} c * OPT^*[s;rs] + b * (|s| * (|s| - 1))/2
\end{aligned}
$$
◀

For showing that algorithm A has the pairwise property it suffices to show that the probability that x precedes y during the service of the projected request sequence is equal to the probability when servicing the original request sequence.

▶ **Lemma 14** ([6, ⇒ of Lemma 1.1]). *For an online algorithm A without paid exchanges, let ${s_{xy}}^{A;rs_{xy}}$ and $s^{A;rs}$ denote the configuration distribution after servicing the projected respectively full request list rs starting from s. Then*
$\mathit{map}_{pmf}\ (\lambda(s,is).\ x < y\ \mathit{in}\ s)\ {s_{xy}}^{A;rs_{xy}} = \mathit{map}_{pmf}\ (\lambda(s,is).\ x < y\ \mathit{in}\ s)\ s^{A;rs} \Longrightarrow \mathit{pairwise}\ A$

4.4 OPT_2: an Optimal Algorithm for Lists of Length 2

We formalize OPT_2, an optimal offline algorithm for lists of length 2 due to Reingold and Westbrook [29], verify its optimality, and determine the cost of OPT_2 on different specific request sequences. The informal definition of OPT_2 is as follows [29]:

▶ **Definition 15** (OPT_2 informally)**.** After each request, move the requested item to the front via free swaps if the next request is also to that item. Otherwise do nothing.

Observe that this algorithm only needs knowledge of the current and next request. Thus it is almost an online algorithm, except that it needs a lookahead of 1.

Function $OPT_2\ rs\ [x,\ y]$ that takes a request sequence rs and a state $[x,\ y]$ and returns an answer sequence is defined easily by recursion on rs.

▶ **Theorem 16** (Optimality of OPT_2)**.**
$set\ rs \subseteq \{x,\ y\} \wedge x \neq y \Longrightarrow T^*\ [x,\ y]\ rs\ (OPT_2\ rs\ [x,\ y]) = OPT^*[[x,\ y];rs]$

The proof is by induction on rs followed by a "simple case analysis" [29] the formalization of which is quite lengthy.

In an execution of OPT_2, after two consecutive requests to the same element that element will be at the front. This enables us to partition the request sequence into phases and restart OPT_2 after each phase:

▶ **Lemma 17.** If $x \neq y \wedge s \in \{[x,\ y],\ [y,\ x]\} \wedge set\ us \subseteq \{x,\ y\} \wedge set\ vs \subseteq \{x,\ y\}$ then $OPT_2\ (us\ @\ [x,\ x]\ @\ vs)\ s = OPT_2\ (us\ @\ [x,\ x])\ s\ @\ OPT_2\ vs\ [x,\ y]$.

Thus we can partition a request sequence into phases ending with two consecutive requests to the same element and we know the state of OPT_2 at the end of each phase. Such a phase can have one of four forms, for any of these we have "calculated" the cost of OPT_2 (see Table 1). This involves inductive proofs in the case of the Kleene star.

In the following two subsections the *phase partitioning* technique is described in more detail and is then used to prove that BIT is 7/4-competitive.

4.5 Phase Partitioning

In the following we will partition all request sequences into *complete phases* that end with two consecutive requests to the same element and possibly a trailing *incomplete phase*. The set of all request sequences can be described by $(x{+}y)^*$, the complete phases by $\varphi_{xy} = (x^?(yx)^*yy{+}y^?(xy)^*xx)$ and the incomplete phases (that do not contain two consecutive occurrences of the same element) by $\overline{\varphi}_{xy} = (x^?(yx)^*y{+}y^?(xy)^*x)$. The regular expression $r^?$ is short for $r + \varepsilon$. In order to prove identities like $\mathcal{L}\ ({\varphi_{xy}}^*{\overline{\varphi}_{xy}}^?) = \mathcal{L}\ ((x{+}y)^*)$ we use a regular expression equivalence checker available in Isabelle/HOL [21], which we extend to regular expressions with variables.

We now want to compare costs of an online algorithm A and OPT_2 on a complete phase rs and lift results to arbitrary request sequences σ containing two different elements. Recall that OPT_2 will have element $rs_{|rs|\ -\ 1}$ in front of the state after servicing rs. Let S^A be a configuration distribution of A, then $S^{A;rs}$ denotes the configuration distribution after the service of rs by A starting from S^A and $A^*[S^A;rs]$ its cost. Let $inv_A\ S^A\ x\ s$ be a predicate on a configuration distribution of A, a request and a state. Suppose for any S^A, x and s (i) $inv_A\ S^A\ x\ s \Longrightarrow A^*[S^A;rs] \leq c * T^*\ [x,\ y]\ rs\ (OPT_2\ rs\ [x,\ y])$ and (ii) $inv_A\ S^A\ x\ s \Longrightarrow inv_A\ S^{A;rs}\ (rs_{|rs|\ -\ 1})\ s$. Then we can conclude $A^*[S^A;\sigma] \leq c * T^*\ [x,\ y]\ \sigma\ (OPT_2\ \sigma\ [x,\ y]) + c$ if the predicate $inv_A\ S^A\ x\ [x,\ y]$ holds initially.

This fact follows by well-founded induction on the length of σ. If we additionally verify that the invariant $inv_A\ S_A\ x\ [x,\ y]$ holds for S_A being the configuration distribution after initializing algorithm A from state $[x,\ y]$ we can finally conclude $A^*[[x,y],\ \sigma] \leq c * T^*\ [x,\ y]\ \sigma\ (OPT_2\ \sigma\ [x,\ y]) + c = c * OPT^*[[x,\ y];\sigma] + c$.

Note that $inv_A\ S^A\ x\ s$ must imply that all states in the configuration distribution S^A have x in front. If this is not the case and the state $[y,\ x]$ has nonzero probability, for the complete phase $rs = [x,\ x]$, A has nonzero costs whereas OPT_2 pays nothing. This makes showing property (i) impossible. Consequently not all pairwise algorithms can be analyzed with this technique (e.g. RMTF [6]).

To further facilitate the analysis all complete phases can be classified into four forms, described by the regular expressions found in the first column of Table 1. Together with the list factoring technique, the proof of an algorithm being c-competitive is reduced to determining the costs on request sequences of these forms. This kind of analysis is conducted in the next section for BIT.

4.6 Analysis of BIT

We now show that BIT is 7/4-competitive using both the list factoring method and the phase partitioning technique.

With the help of Lemma 14 we proved that BIT has the pairwise property. The result can be established by induction on the original request sequence and a case distinction whether the requested element is one of x and y or not, the rest is laborious bookkeeping. We refer the reader to [12] for the details.

▶ **Lemma 18.** *pairwise BIT*

Let us turn to showing that BIT is 7/4-competitive on lists of length 2. To that end we analyze BIT on the different forms of complete phases as explained above. At the end of each complete phase, BIT will have the last request in front of the state (because the element was requested twice and one of the two requests moved it). BIT and OPT_2 thus are synchronized before and after each phase. BIT's invariant $inv_{BIT}\ S\ x\ s$ (in the sense of the previous subsection) is defined as saying that in every configuration in the distribution S, element x is in front of the state and the second component of the internal state is s.

	σ	BIT	OPT_2	TS
A	$x^?yy$	1.5	1	2
B	$x^?yx(yx)^*yy$	$1.5 * (k+1) + 1$	$(k+1) + 1$	$2 * (k+1)$
C	$x^?yx(yx)^*x$	$1.5 * (k+1) + 0.25$	$(k+1)$	$2 * (k+1) - 1$
D	xx	0	0	0

Table 1 Costs of BIT, OPT_2 and TS for request sequence of the four phase forms; x is the first element in the state; k is the number of iterations of the Kleene star expression.

Table 1 shows the costs of BIT for the four respective forms. We now verify both the preservation of inv_{BIT} and the cost incurred by BIT for a phase rs of form B, i.e., $rs \in \mathcal{L}\ (x^?yx(yx)^*yy)$.

We start with the configuration distribution S satisfying $inv_{BIT}\ S\ x\ s$ (see above). First we observe that serving an optional request x does not alter the configuration distribution nor add any cost. Now serving the request y moves y to the front for two out of four configurations and in any case adds cost 1. In one of the former two configurations the next

request to element x brings x to the front again. Consequently the state after serving yx is $[y, x]$ iff y's bit is set and x's bit is not set. This distribution of configurations is preserved by any number of further requests yx. Serving the first request to yx costs $1 + 1/2$ as well as any further request to yx has cost $3/4 + 3/4$. Let σ be the part of rs with the Kleene star, by induction on the Kleene star in σ the cost for serving σ is $3/4 * |\sigma|$. The trailing request to yy then costs $3/4 + 1/4$ and y is moved to the front in all configurations. Thus finally the invariant $inv_{BIT}\ S'\ y\ s$ is satisfied, where S' is the configuration distribution after serving rs. Note that here the order of y and x have changed. In the analysis of the next phase, x and y take the swapped positions. But as they are interchangeable in all the theorems this does no harm.

Determining the costs of Table 1 usually is presented in the style of the last paragraph ([6, §2.4] and [2, Lemma 3]). While these calculations are tedious for humans, the proof assistant is able to carry them out almost automatically: With the help of some lemmas about how BIT transforms the configuration distribution on single requests, the costs of complete phases can be calculated and proved mechanically.

Note that in contrast to Table 1.1 in [6, §1.6.1] we define the phase forms differently. They allow more than one initial x in forms A, B and C, we only allow zero or one. Our forms precisely capture the idea of splitting the request sequence into phases that end with two consecutive requests to the same element. Moreover, form D is missing from their table.

The results for the other phase forms follows in a similar way and finally we prove $BIT^*[[x, y];\sigma] \leq 7/4 * OPT^*[[x, y];\sigma] + 7/4$. Together with the pairwise property of BIT and the list factoring method we can lift the result to lists of arbitrary length and obtain another proof of Theorem 8.

Actually the ratio between costs of BIT and OPT_2 tends to $3/2$ for long phases, only the poor performance of short phases of form C leads to the competitive ratio of $7/4$.

4.7 TS to the Rescue

The deterministic online algorithm TS due to Albers [1] performs well in the cases where BIT performs badly. We now present the analysis of TS and in the following subsection will show how the two algorithms can be combined. TS does the following:

▶ **Definition 19** (TS informally)**.** After each request, the accessed item x is inserted immediately in front of the first item y that precedes x in the list and was requested at most once since the last request to x. If there is no such item y or if x is requested for the first time, then the position of x remains unchanged.

This algorithm has an internal state of type $\alpha\ list$, the history of requests already processed. The transition function δ_{TS} is formalized as follows:

▶ **Definition 20.**

$\delta_{TS}\ (s,\ is)\ r =$
  $(let\ V_r = \{x \mid x < r\ in\ s \wedge count\_list\ (take\ (index\ is\ r)\ is)\ x \leq 1\}$
   $in\ ((if\ index\ is\ r < |is| \wedge V_r \neq \emptyset\ then\ index\ s\ r - Min\ (index\ s\ `\ V_r)\ else\ 0,\ []),$
     $r \cdot is))$

Note that $take\ n\ xs$ returns the length n prefix of xs; because the history is stored in reverse order, $take\ (index\ is\ r)\ is$ is the part of the history since the last request to element r.

For the analysis of TS we employ the proof methods developed in the preceding subsections. We first examine the costs of the four phase forms by simulation and induction. Then, by the phase partitioning method, we extend the result to any request sequence. The

invariant needed for TS essentially says $is = [] \vee (\exists x\ s'\ hs.\ s = x \cdot s' \wedge is = x \cdot x \cdot hs)$: either TS has just been initialized or the last two requests were to the first element in the state. Intuitively this invariant implies that for the next request to y, the element would not be moved to the front of x. The last column of Table 1 shows the costs of TS for the four respective phase forms. TS performs better than BIT for short phases of forms B and C.

To lift this result to arbitrary initial lists, it remains to show that TS satisfies the pairwise property. With Lemma 14 and because we are in the deterministic domain it suffices to show that the relative order of x and y is equal both in the service of the projected as well as the original request sequence. This fact is not as obvious as for MTF or BIT; for showing this equality at any point in time during the service of TS, we do a case distinction on the history and look at most at the last three accesses to x and y. For most cases it is quite easy to determine the current relative order both in the projected as well as the full request sequence. For example, after two requests to the same element x, x must be before y in both cases. The only tricky case is when the last requests were xxy: then a proof involving infinite descent is used along the lines of Lemma 2 in [2].

Finally we obtain the fact *pairwise TS* and hence

▶ **Theorem 21.** $compet\ TS\ 2\ \{s \mid distinct\ s \wedge s \neq []\}$

4.8 COMB

Our development finally peaks in the formalization of the 8/5-competitive online algorithm COMB due to Albers *et al.* [2] that chooses with probability 4/5 between executing BIT and TS. COMB's internal state type is the sum type of the internal state types of BIT and TS, function ι_{COMB} initializes like BIT and TS with respective probabilities and function δ_{COMB} employs δ_{BIT} or δ_{TS} depending on the type of the internal state it receives. As COMB is a combination of BIT and TS several properties carry over directly: COMB is barely random, does not use paid exchanges and *pairwise COMB* holds.

Table 1 shows that BIT outperforms TS for long phases. TS is cheaper only for short phases of forms B and C. The combination of the two algorithms yields an improved competitive ratio of 8/5. The result is established by analyzing the combined cost for the different phase forms and then use the phase partitioning and list factoring method.

▶ **Theorem 22.** $compet\ COMB\ (8/5)\ \{x \mid distinct\ x \wedge x \neq []\}$

It can also be shown (we did not verify this) that the probability for choosing between BIT and TS is optimal and that COMB attains the best competitive ratio possible for pairwise algorithms in the partial cost model [4].

5 Conclusion

This paper has demonstrated that state of the art randomized list update algorithms can be analyzed with a theorem prover. In the process we found mistakes and omissions in the published literature.

The field of programming languages is full of verified material (e.g., [25, 27]) which has lead to achievements like Leroy's verified C compiler [22]. We believe that eventually both functional correctness and performance of critical software components will be verified. Such verifications will require verified algorithm analyses such as presented in this paper.

References

1 Susanne Albers. Improved randomized on-line algorithms for the list update problem. *SIAM J. Comput.*, 27(3):682–693, 1998.

2 Susanne Albers, Bernhard von Stengel, and Ralph Werchner. A combined BIT and TIMESTAMP algorithm for the list update problem. *Inf. Process. Lett.*, 56(3):135–139, 1995.

3 Christoph Ambühl. Offline list update is NP-hard. In Mike Paterson, editor, *Algorithms - ESA 2000*, volume 1879 of *LNCS*, pages 42–51. Springer, 2000.

4 Christoph Ambühl, Bernd Gärtner, and Bernhard Von Stengel. Optimal projective algorithms for the list update problem. In *Automata, Languages and Programming*, pages 305–316. Springer, 2000.

5 Gilles Barthe, Juan Manuel Crespo, Benjamin Grégoire, César Kunz, and Santiago Zanella Béguelin. Computer-aided cryptographic proofs. In L. Beringer and A. Felty, editors, *Interactive Theorem Proving (ITP 2012)*, volume 7406 of *LNCS*, pages 11–27. Springer, 2012.

6 Allan Borodin and Ran El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.

7 Arthur Charguéraud and François Pottier. Machine-checked verification of the correctness and amortized complexity of an efficient union-find implementation. In C. Urban and X. Zhang, editors, *ITP 2015*, volume 9236 of *LNCS*, pages 137–153. Springer, 2015.

8 Martin Erwig and Steve Kollmansberger. Functional pearls: Probabilistic functional programming in Haskell. *Journal of Functional Programming*, 16(1):21–34, 2006.

9 Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry. A machine-checked proof of the odd order theorem. In S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors, *Interactive Theorem Proving (ITP 2013)*, volume 7998 of *LNCS*, pages 163–179. Springer, 2013.

10 Thomas C. Hales, Mark Adams, Gertrud Bauer, Dat Tat Dang, John Harrison, Truong Le Hoang, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Thang Tat Nguyen, Truong Quang Nguyen, Tobias Nipkow, Steven Obua, Joseph Pleso, Jason Rute, Alexey Solovyev, An Hoai Thi Ta, Trung Nam Tran, Diep Thi Trieu, Josef Urban, Ky Khac Vu, and Roland Zumkeller. A formal proof of the Kepler conjecture. *CoRR*, abs/1501.02155, 2015.

11 G. Hardy, J.E. Littelwood, and G. Pólya. *Inequalities*. Cambridge University Press, 1934.

12 Maximilian P.L. Haslbeck and Tobias Nipkow. List update. *Archive of Formal Proofs*, 2016. http://devel.isa-afp.org/entries/List_Update.shtml, Formal proof development.

13 Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Multivariate amortized resource analysis. *ACM Trans. Program. Lang. Syst.*, 34(3):14, 2012.

14 Johannes Hölzl, Andreas Lochbihler, and Dmitriy Traytel. A formalized hierarchy of probabilistic system types. In *Interactive Theorem Proving*, pages 203–220. Springer, 2015.

15 Joe Hurd. *Formal Verification of Probabilistic Algorithms*. PhD thesis, University of Cambridge, 2002.

16 Joe Hurd. Verification of the Miller-Rabin probabilistic primality test. *J. Logic Algebraic Programming*, 56(1-2):3–21, 2003.

17 Joe Hurd, Annabelle McIver, and Carroll Morgan. Probabilistic guarded commands mechanized in HOL. *Theoretical Computer Science*, 346(1):96–112, 2005.

18 Sandy Irani. Two results on the list update problem. *Inf. Process. Lett.*, 38(6):301–306, 1991.

19 Shahin Kamali and Alejandro López-Ortiz. A survey of algorithms and models for list update. In A. Brodnik, A. López-Ortiz, V. Raman, and A. Viola, editors, *Space-Efficient Data Structures, Streams, and Algorithms*, volume 8066 of *LNCS*, pages 251–266. Springer, 2013.

20 Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: Formal verification of an OS kernel. In Jeanna Neefe Matthews and Thomas E. Anderson, editors, *Proc. 22nd ACM Symposium on Operating Systems Principles 2009*, pages 207–220. ACM, 2009.

21 Alexander Krauss and Tobias Nipkow. Proof pearl: Regular expression equivalence and relation algebra. *J. Automated Reasoning*, 49:95–106, 2012. Published online March 2011.

22 Xavier Leroy. A formally verified compiler back-end. *J. Automated Reasoning*, 43:363–446, 2009.

23 Alejandro López-Ortiz, Marc P. Renault, and Adi Rosén. Paid exchanges are worth the price. In E.W. Mayr and N. Ollinger, editors, *Symposium on Theoretical Aspects of Computer Science, STACS 2015*, volume 30 of *LIPIcs*, pages 636–648. Schloss Dagstuhl, 2015.

24 Tobias Nipkow. Amortized complexity verified. In C. Urban and X. Zhang, editors, *ITP 2015*, volume 9236 of *LNCS*, pages 310–324. Springer, 2015.

25 Tobias Nipkow and Gerwin Klein. *Concrete Semantics with Isabelle/HOL*. Springer, 2014. http://concrete-semantics.org.

26 Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

27 Benjamin C. Pierce and Stephanie Weirich, editors. *Special Issue: The* PoplMark *Challenge*, volume 49(3) of *J. Automated Reasoning*. 2012.

28 Nick Reingold and Jeffery Westbrook. Optimum off-line algorithms for the list update problem. Technical Report 805, Yale University, Department of Computer Science, 1990.

29 Nick Reingold and Jeffery Westbrook. Off-line algorithms for the list update problem. *Inf. Process. Lett.*, 60(2):75–80, 1996.

30 Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Comm. ACM*, 28(2):202–208, 1985.