Lock-free dynamic hash tables with open addressing

Gao, H.; Groote, J.F.; Hesselink, W.H.

DISTRIBUTED COMPUTING

Lock-free dynamic hash tables with open addressing

H. Gao[1], J.F. Groote[2,3], W.H. Hesselink[1]

[1] Department of Mathematics and Computing Science, University of Groningen, P.O. Box 800, 9700 AV Groningen, The Netherlands
   (e-mail: {hui,wim}@cs.rug.nl)
[2] Department of Mathematics and Computing Science, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven,
   The Netherlands
[3] CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands (e-mail: jfg@win.tue.nl)

Abstract. We present an efficient lock-free algorithm for parallel accessible hash tables with open addressing, which promises more robust performance and reliability than conventional lock-based implementations. "Lock-free" means that it is guaranteed that always at least one process completes its operation within a bounded number of steps. For a single processor architecture our solution is as efficient as sequential hash tables. On a multiprocessor architecture this is also the case when all processors have comparable speeds. The algorithm allows processors that have widely different speeds or come to a halt. It can easily be implemented using C-like languages and requires on average only constant time for insertion, deletion or accessing of elements. The algorithm allows the hash tables to grow and shrink when needed.

Lock-free algorithms are hard to design correctly, even when apparently straightforward. Ensuring the correctness of the design at the earliest possible stage is a major challenge in any responsible system development. In view of the complexity of the algorithm, we turned to the interactive theorem prover PVS for mechanical support. We employ standard deductive verification techniques to prove around 200 invariance properties of our algorithm, and describe how this is achieved with the theorem prover PVS.

1 Introduction

We are interested in efficient, reliable, parallel algorithms. The classical synchronization paradigm based on mutual exclusion is not most suited for this, since mutual exclusion often turns out to be a performance bottleneck, and failure of a single process can force all other processes to come to a halt. This is the reason to investigate lock-free or wait-free concurrent objects, see e.g. [4,9,10,12,15,16,18,22,24–26,28].

Lock-free and wait-free objects

A *concurrent object* is an abstract data type that permits concurrent operations that appear to be atomic [9,20,28]. The easiest way to implement concurrent objects is by means of mutual exclusion, but this leads to blocking when the process that holds exclusive access to the object is delayed or stops functioning.

The object is said to be *lock-free* if any process can be delayed at any point without forcing any other process to block and when, moreover, it is guaranteed that always some process will complete its operation in a finite number of steps, regardless of the execution speeds of the processes [4,10,18,24,28]. The object is said to be *wait-free* when it is guaranteed that any process can complete any operation in a finite number of steps, regardless of the speeds of the other processes [9].

We regard "non-blocking" as synonymous to "lock-free". In several recent papers, e.g. [26], the term "non-blocking" is used for the first conjunct in the above definition of lock-free. Note that this weaker concept does not in itself guarantee progress. Indeed, without real blocking, processes might delay each other arbitrarily without getting closer to completion of their respective operations. The older literature [1,4,12] seems to suggest that originally "non-blocking" was used for the stronger concept, and lock-free for the weaker one. Be this as it may, we use lock-free for the stronger concept.

Concurrent hash tables

The data type of hash tables is very commonly used to efficiently store huge but sparsely filled tables. Before 2003, as far as we know, no lock-free algorithm for hash tables had been proposed. There were general algorithms for arbitrary wait-free objects [2–4,9,13,14], but these are not very efficient. Furthermore, there are lock-free algorithms for different

domains, such as linked lists [28], queues [27] and memory management [12,15].

In this paper we present a lock-free algorithm for hash tables with open addressing that is in several aspects wait-free. The central idea is that every process holds a pointer to a hash table, which is the current one if the process is not delayed. When the current hash table is full, a new hash table is allocated and all active processes join in the activity to transfer the contents of the current table to the new one. The consensus problem of the choice of a new table is solved by means of a test-and-set register. When all processes have left the obsolete table, it is deallocated by the last one leaving. This is done by means of a compare-and-swap register. Measures have been taken to guarantee that actions of delayed processes are never harmful. For this purpose we use counters that can be incremented and decremented atomically.

After the initial design, it took us several years to establish the safety properties of the algorithm. We did this by means of the proof assistant PVS [23]. Upon completion of this proof, we learned that a lock-free resizable hash table based on chaining was proposed in [25]. We come back to this below.

Our algorithm is lock-free and some of the subtasks are wait-free. We allow fully parallel *insertion*, *assignment*, *deletion*, and *finding* of elements. Finding is wait-free, the other three are not. The primary cause is that the process executing it may repeatedly have to execute or join a migration of the hash table. Assignment and deletion are also not wait-free when other processes repeatedly assign to the same address successfully.

Migration is called for when the current hash table is almost filled. This occurs when the table has to grow beyond its current upper bound, but also for maintenance after many insertions and deletions. The migration itself is wait-free, but, in principle, it is possible that a slow process is unable to access and use a current hash table since the current hash table is repeatedly replaced by faster processes.

Migration requires subtle provisions, which can be best understood by considering the following scenario. Suppose that process A is about to (slowly) insert an element in a hash table H_1. Before this happens, however, a fast process B has performed migration by making a new hash table H_2, and copying the content from H_1 to H_2. If (and only if) process B did not copy the insertion of A, A must be informed to move to the new hash table, and carry out the insertion there. Suppose a process C comes into play also copying the content from H_1 to H_2. This must be possible, since otherwise B can stop copying, blocking all operations of other processes on the hash table, and thus violating the lock-free nature of the algorithm. Now the value inserted by A can but need not be copied by both B and/or C. This can be made more complex by a process D that attempts to replace H_2 by H_3. Still, the value inserted by A should show up exactly once in the hash table, and it is clear that processes should carefully keep each other informed about their activities on the tables.

Performance, comparison, and correctness

For a single processor architecture our solution is of the same order of efficiency as sequential hash tables. Actually, only an extra check is required in the main loop of the main func-

tions, one extra bit needs to be set when writing data in the hashtables and at some places a write operation has been replaced by a compare-and-swap, which is more expensive. For ordinary operations on the hashtable, this is the only overhead and therefore a linear speed up can be expected on multiprocessor systems. The only place where no linear speed up can be achieved is when copying the hashtable. Especially, when processes have widely different speeds, a logarithmic factor may come into play (see algorithms for the write all problem [7,16]). Indeed, initial experiments indicate that our algorithm is as efficient as sequential hash tables. It seems to require on average only constant time for insertion, deletion or accessing of elements.

Some differences between our algorithm and the algorithm of [25] are clear. In our algorithm, the hashed values need not be stored in dynamic nodes if the address-value pairs (plus one additional bit) fit into one word. Our hash table can shrink whereas the table of bucket headers in [25] cannot shrink. A disadvantage of our algorithm, due to its open addressing, is that migration is needed as maintenance after many insertions and deletions.

An apparent weakness of our algorithm is the worst-case space complexity in the order of $\mathcal{O}(PM)$ where P is the number of processes and M is the size of the table. This only occurs when many of the processes fail or fall asleep while using the hash table. Failure while using the hash table can be made less probable by adequate use of procedure "releaseAccess". This gives a trade-off between space and time since it introduces the need of a corresponding call of "getAccess". When all processes make ordinary progress and the hash table is not too small, the actual memory requirement in $\mathcal{O}(M)$.

The migration activity requires worst-case $\mathcal{O}(M^2)$ time for each participating process. This only occurs when the migrating processes tend to choose the same value to migrate and the number of collisions is $\mathcal{O}(M)$ due to a bad hash function. This is costly, but even this is in agreement with wait-freedom. The expected amount of work for migration for all processes together is $\mathcal{O}(M)$ when collisions are sparse, as should be the case when migrating to a hash table that is sufficiently large.

A true problem of lock-free algorithms is that they are hard to design correctly, which even holds for apparently straightforward algorithms. Whereas human imagination generally suffices to deal with all possibilities of sequential processes or synchronized parallel processes, this appears impossible (at least to us) for lock-free algorithms. The only technique that we see fit for any but the simplest lock-free algorithms is to prove the correctness of the algorithm very precisely, and to verify this using a proof checker or theorem prover.

As a correctness notion, we take that the operations behave the same as for 'ordinary' hash tables, under some arbitrary linearization [11] of these operations. So, if a *find* is carried out strictly after an *insert*, the inserted element is found. If *insert* and *find* are carried out at the same time, it may be that *find* takes place before *insertion*, and it is not determined whether an element will be returned.

Our algorithm contains 81 atomic statements. The structure of our algorithm and its correctness properties, as well as the complexity of reasoning about them, makes neither automatic nor manual verification feasible. We have therefore chosen the higher-order interactive theorem prover PVS [5, 23] for mechanical support. PVS has a convenient specifica-

tion language and contains a proof checker which allows users to construct proofs interactively, to automatically execute trivial proofs, and to check these proofs mechanically.

Overview of the paper

Section 2 contains the description of the hash table interface offered to the users. The algorithm is presented in Section 3. Section 4 contains a description of the proof of the safety properties of the algorithm: functional correctness, atomicity, and absence of memory loss. This proof is based on a list of around 200 invariants, presented in Appendix A, while the relationships between the invariants are given by a dependency graph in Appendix B. Progress of the algorithm is proved informally in Section 5. Conclusions are drawn in Section 6.

2 The interface

The aim is to construct a hash table that can be accessed simultaneously by different processes in such a way that no process can passively block another process' access to the table.

A hash table is an implementation of (partial) functions between two domains, here called *Address* and *Value*. The hash table thus implements a modifiable shared variable X : $Address \rightarrow Value$. The domains *Address* and *Value* both contain special default elements $0 \in Address$ and $\textbf{null} \in Value$. An equality $X(a) = \textbf{null}$ means that no value is currently associated with the address a. In particular, since we never store a value for the address 0, we impose the invariant

$$X(0) = \textbf{null} .$$

We use open addressing to keep all elements within the table. For the implementation of the hash table we require that from every value the address it corresponds to is derivable. We therefore assume that some function $ADR : Value \rightarrow Address$ is given with the property that

Ax1: $\qquad v = \textbf{null} \equiv ADR(v) = 0$

Indeed, we need **null** as the value corresponding to the undefined addresses and use address 0 as the (only) address associated with the value **null**. We thus require the hash table to satisfy the invariant

$$X(a) \neq \textbf{null} \implies ADR(X(a)) = a .$$

Note that the existence of *ADR* is not a real restriction since one can choose to store the pair (a, v) instead of v. When a can be derived from v, it is preferable to store v, since that saves memory.

There are four principle operations: *find*, *delete*, *insert* and *assign*. The first one is to *find* the value currently associated with a given address. This operation yields **null** if the address has no associated value. The second operation is to *delete* the value currently associated with a given address. It fails if the address was empty, i.e. $X(a) = \textbf{null}$. The third operation is to *insert* a new value for a given address, provided the address was empty. So, note that at least one out of two consecutive *insert*s for address a must fail, except when there is a *delete* for address a in between them. The operation *assign* does the same as *insert*, except that it rewrites the value even if the associated address is not empty. Moreover, *assign* never fails.

We assume that there is a bounded number of processes that may need to interact with the hash table. Each process is characterized by the sequence of operations

$$(\ getAccess \ ; \ (find + delete + insert + assign)^* \ ;$$
$$releaseAccess)^\omega$$

A process that needs to access the table, first calls the procedure *getAccess* to get the current hash table pointer. It may then invoke the procedures *find*, *delete*, *insert*, and *assign* repeatedly, in an arbitrary, serial manner. A process that has access to the table can call *releaseAccess* to log out. The processes may call these procedures concurrently. The only restriction is that every process can do at most one invocation at a time.

The basic correctness conditions for concurrent systems are functional correctness and atomicity, say in the sense of [20], Chapter 13. Functional correctness is expressed by prescribing how the procedures *find*, *insert*, *delete*, *assign* affect the value of the abstract mapping X in relation to the return value. Atomicity means that the effect on X and the return value takes place atomically at some time between the invocation of the routine and its response. Each of these procedures has the precondition that the calling process has access to the table. In this specification, we use auxiliary private variables declared locally in the usual way. We give them the suffix S to indicate that the routines below are the specifications of the procedures. We use angular brackets $\langle$ and $\rangle$ to indicate atomic execution of the enclosed command.

$$\textbf{proc } find_S(a : Address \setminus \{0\}) : Value =$$
$$\quad \textbf{local } rS : Value;$$
(fS) $\quad \langle \ rS := X(a) \ \rangle;$
$$\quad \textbf{return } rS.$$

$$\textbf{proc } delete_S(a : Address \setminus \{0\}) : Bool =$$
$$\quad \textbf{local } sucS : Bool;$$
(dS) $\quad \langle \ sucS := (X(a) \neq \textbf{null}) \ ;$
$$\quad \ \textbf{if } sucS \textbf{ then } X(a) := \textbf{null end} \ \rangle \ ;$$
$$\quad \textbf{return } sucS.$$

$$\textbf{proc } insert_S(v : Value \setminus \{\textbf{null}\}) : Bool =$$
$$\quad \textbf{local } sucS : Bool \ ; \ a : Address := ADR(v) \ ;$$
(iS) $\quad \langle \ sucS := (X(a) = \textbf{null}) \ ;$
$$\quad \ \textbf{if } sucS \textbf{ then } X(a) := v \textbf{ end} \ \rangle \ ;$$
$$\quad \textbf{return } sucS.$$

$$\textbf{proc } assign_S(v : Value \setminus \{\textbf{null}\}) =$$
$$\quad \textbf{local } a : Address := ADR(v) \ ;$$
(aS) $\quad \langle \ X(a) := v \ \rangle \ ;$
$$\quad \textbf{end}.$$

Note that, in all cases, we require that the body of the procedure is executed atomically at some moment between the beginning and the end of the call, but that this moment need not coincide with the beginning or end of the call. This is the reason that we do not (e.g.) specify *find* by the single line **return** $X(a)$.

Due to the parallel nature of our system we cannot use pre and postconditions to specify it. For example, it may happen that $insert(v)$ returns *true* while $X(ADR(v)) \neq v$ since another process deletes $ADR(v)$ between the execution of (iS) and the response of *insert*.

In Section 3.4, we provide implementations for the operations *find*, *delete*, *insert*, *assign*. We prove partial correctness of the implementations by extending them with the auxiliary

variables and commands used in the specification. So, we regard X as a shared auxiliary variable and rS and $sucS$ as private auxiliary variables; we augment the implementations of *find*, *delete*, *insert*, *assign* with the atomic commands (fS), (dS), (iS), (aS), respectively. We prove that each of the four implementations executes its specification command always exactly once and that the resulting value r or *suc* of the implementation equals the resulting value rS or *sucS* in the specification. It follows that, by removing the implementation variables from the combined program, we obtain the specification. This removal may eliminate many atomic steps of the implementation. This is known as removal of stutterings in TLA [19] or abstraction from τ steps in process algebras.

3 The algorithm

An implementation consists of P processes along with a set of variables, for $P \geq 1$. Each process, numbered from 1 up to P, is a sequential program comprised of atomic statements. Actions on private variables can be added to an atomic statement, but all actions on shared variables must be separated into atomic accesses. Since auxiliary variables are only used to facilitate the proof of correctness, they can be assumed to be touched instantaneously without violation of the atomicity restriction.

3.1 Hashing

We implement function X via hashing with open addressing, cf. [17,29]. We do not use direct chaining, where colliding entries are stored in a secondary list, as is done in [25]. A disadvantage of open addressing with deletion of elements is that the contents of the hash table must regularly be refreshed by copying the non-deleted elements to a new hash table. As we wanted to be able to resize the hash tables anyhow, we consider this less of a burden.

In principle, hashing is a way to store address-value pairs in an array (hash table) with a length much smaller than the number of potential addresses. The indices of the array are determined by a hash function. In case the hash function maps two addresses to the same index in the array there must be some method to determine an alternative index. The question how to choose a good hash function and how to find alternative locations in the case of open addressing is treated extensively elsewhere, e.g. [17].

For our purposes it is convenient to combine these two roles in one abstract function *key* given by:

$$key(a : Address, \ l : Nat, \ n : Nat) : Nat \ ,$$

where l is the length of the array (hash table), that satisfies

Ax2: $0 \leq key(a, l, n) < l$

for all a, l, and n. The number n serves to obtain alternative locations in case of collisions: when there is a collision, we re-hash until an empty "slot" (i.e. **null**) or the same address in the table is found. The approach with a third argument n is unusual but very general. It is more usual to have a function *Key* dependent on a and l, and use a second function *Inc*, which

may depend on a and l, to use in case of collisions. Then our function *key* is obtained recursively by

$$key(a, l, 0) = Key(a, l)$$
$$key(a, l, n + 1) = Inc(a, l, key(a, l, n)) \ .$$

We require that, for any address a and any number l, the first l keys are all different, as expressed in

Ax3: $0 \leq k < m < l \ \Rightarrow \ key(a, l, k) \neq key(a, l, m) \ .$

3.2 Tagging of values

As is well known [17], hashing with open addressing needs a special value **del** $\in$ *Value* to replace deleted values.

When the current hash table becomes full, the processes need to reach consensus to allocate a new hash table of new size to replace the current one. Then all values except **null** and **del** must be migrated to the new hash table. A value that is being migrated cannot be simply removed, since the migrating process may stop functioning during the migration. Therefore, a value being copied must be tagged in such a way that it is still recognizable. This is done by the function *old*. We thus introduce an extended domain of values to be called *EValue*, which is defined as follows:

$$EValue = \{\textbf{del}\} \cup Value \cup \{old(v) \mid v \in Value\}$$

We furthermore assume the existence of functions *val* : *EValue* $\rightarrow$ *Value* and *oldp* : *EValue* $\rightarrow$ *Bool* that satisfy, for all $v \in$ *Value*:

$$
\begin{array}{ll}
val(v) = v & oldp(v) = false \\
val(\textbf{del}) = \textbf{null} & oldp(\textbf{del}) = false \\
val(old(v)) = v & oldp(old(v)) = true
\end{array}
$$

Note that the *old* tag can easily be implemented by designating one special bit in the representation of *Value*. In the sequel we write **done** for $old(\textbf{null})$. Moreover, we extend the function *ADR* to domain *EValue* by $ADR(v) = ADR(val(v))$.

3.3 Data structure

A *Hash table* is either $\bot$, indicating the absence of a hash table, or it has the following structure:

```
size, bound, occ, dels : Nat;
table : array 0..size-1 of EValue.
```

The field `size` indicates the size of the hash table, `bound` the maximal number of places that can be occupied before refreshing the table. Both are set when creating the table and remain constant. The variable `occ` gives the number of occupied positions in the table, while the variable `dels` gives the number of deleted positions. If h is a pointer to a hash table, we write $h.\texttt{size}$, $h.\texttt{occ}$, $h.\texttt{dels}$ and $h.\texttt{bound}$ to access these fields of the hash table. We write $h.\texttt{table}[i]$ to access the ith *EValue* in the table.

Apart from the *current* hash table, which is the main representative of the variable X, we have to deal with *old* hash tables, which were in use before the current one, and *new* hash tables, which can be created after the current one.

We now introduce data structures that are used by the processes to find and operate on the hash table and allow to delete

hash tables that are not used anymore. The basic idea is to count the number of processes that are using a hash table, by means of a counter `busy`. The hash table can be thrown away when `busy` is set to 0. An important observation is that `busy` cannot be stored as part of the hash table, in the same way as the variables `size`, `occ` and `bound` above. The reason for this is that a process can attempt to access the current hash table by increasing its `busy` counter. However, just before it wants to write the new value for `busy` it falls asleep. When the process wakes up the hash table might have been deleted and the process would be writing at a random place in memory.

This forces us to use separate arrays `H` and `busy` to store the pointers to hash tables and the `busy` counters. There can be $2P$ hash tables around, because each process can simultaneously be accessing one hash table and attempting to create a second one. The arrays below are shared variables.

$$H : \textbf{array } 1 \ldots 2P \textbf{ of pointer to } \textit{Hashtable} \,;$$
$$\texttt{busy} : \textbf{array } 1 \ldots 2P \textbf{ of } \textit{Nat} \,;$$
$$\texttt{prot} : \textbf{array } 1 \ldots 2P \textbf{ of } \textit{Nat} \,;$$
$$\texttt{next} : \textbf{array } 1 \ldots 2P \textbf{ of } 0 \ldots 2P \,.$$

As indicated, we also need arrays `prot` and `next`. The variable $\texttt{next}[i]$ points to the next hash table to which the contents of hash table $H[i]$ is being copied. If $\texttt{next}[i]$ equals 0, this means that there is no next hash table. The variable $\texttt{prot}[i]$ is used to guard the variables $\texttt{busy}[i]$, $\texttt{next}[i]$ and $H[i]$ against being reused for a new table, before all processes have discarded them.

We use a shared variable `currInd` to hold the index of the currently valid hash table:

$$\texttt{currInd} : 1 \ldots 2P \,.$$

Note however that after a process copies `currInd` to its local memory, other processes may create a new hash table and change `currInd` to point to that one.

It is assumed that initially $H[1]$ is pointing to some hash table. The other initial values of the shared variables are given by

$$\texttt{currInd} = \texttt{busy}[1] = \texttt{prot}[1] = 1 \,,$$
$$H[i] = \texttt{busy}[i] = \texttt{prot}[i] = 0 \ \text{for all } i \neq 1 \,,$$
$$\texttt{next}[i] = 0 \ \text{for all } i \,.$$

3.4 Primary procedures

We first provide the code for the primary procedures, which match directly with the procedures in the interface. Every process has a private variable

$$\textit{index} : 1 \ldots 2P;$$

containing what it regards as the currently active hash table. At entry of each primary procedure, it must be the case that the variable $H[\textit{index}]$ contains valid information. In section 3.5, we provide procedure *getAccess* with the main purpose to guarantee this property. When *getAccess* has been called, the system is obliged to keep the hash table at *index* stored in memory, even if there are no accesses to the hash table using any of the primary procedures. A procedure *releaseAccess* is provided to release resources, and it should be called whenever the process will not access the hash table for some time.

3.4.1 Syntax

We use a syntax analogous to Modula-3 [8]. We use := for the assignment. We use the C–operations ++ and -- for atomic increments and decrements. The semicolon is a separator, not a terminator. The basic control mechanisms are

> **loop .. end** is an infinite loop,
>   terminated by **exit** or **return**
> **while .. do .. end** and **repeat .. until ..**
>   are ordinary loops
> **if .. then ..** {**elsif ..**} [**else ..**] **end** is the conditional
> **case .. end** is a case statement.

Types are slanted and start with a capital. Shared variables and shared data elements are in typewriter font. Private variables are slanted or in math italic.

3.4.2 The main loop

We model the clients of the hash table in the following loop. This is not an essential part of the algorithm, but it is needed in the PVS description, and therefore provided here.

```
         loop
0:           getAccess() ;
         loop
1:           choose call; case call of
                 (f, a) with  a ≠ 0 → find(a)
                 (d, a) with  a ≠ 0 → delete(a)
                 (i, v) with  v ≠ null → insert(v)
                 (a, v) with  v ≠ null → assign(v)
                 (r)  → releaseAccess(index); exit
             end
         end
         end
```

The main loop shows that each process repeatedly invokes its four principle operations with correct arguments in an arbitrary, serial manner. Procedure *getAccess* has to provide the client with a protected value for *index*. Procedure *releaseAccess* releases this value and its protection. Note that **exit** means a jump out of the inner loop.

3.4.3 Procedure *find*

Finding an address in a hash table with open addressing requires a linear search over the possible hash keys until the address or an empty slot is found. The kernel of procedure *find* is therefore:

$$n := 0 \,;$$
$$\textbf{repeat } r := h.\texttt{table}[\textit{key}(a, l, n)] \,; \quad n\texttt{++} \,;$$
$$\textbf{until } r = \textbf{null} \vee a = \textit{ADR}(r) \,;$$

The main complication is that, when the process encounters an entry **done** (i.e. *old*(**null**)), it has to join the migration activity by calling *refresh*.

Apart from a number of special commands, we group statements such that at most one shared variable is accessed and label these 'atomic' statements with a number. The labels are chosen identical to the labels in the PVS code, and therefore not completely consecutive.

In every execution step, one of the processes proceeds from one label to a next one. The steps are thus treated as atomic. The atomicity of steps that refer to shared variables more than once is emphasized by enclosing them in angular brackets. Since procedure calls only modify private control data, procedure headers are not always numbered themselves, but their bodies usually have numbered atomic statements.

$$
\begin{aligned}
&\textbf{proc } \textit{find}(a : \textit{Address} \setminus \{0\}) : \textit{Value} = \\
&\quad \textbf{local } r : \textit{EValue} \,;\, n, l : \textit{Nat} \,; \\
&\qquad h : \textbf{pointer to } \textit{Hashtable} \,;
\end{aligned}
$$

5: $h := \mathrm{H}[\textit{index}]\,;\, n := 0\,;\, \{\textit{cnt} := 0\}\,;$
6: $l := h.\texttt{size}\,;$
    repeat
7: $\langle\, r := h.\texttt{table}[\textit{key}(a, l, n)]\,;$
    $\{\textbf{if } r = \textbf{null} \lor a = \textit{ADR}(r) \textbf{ then } \textit{cnt}{+}{+};$
    (fS) **end** $\}\,\rangle\,;$
8: **if** $r = \textbf{done}$ **then**
    $\textit{refresh}()\,;$
10: $h := \mathrm{H}[\textit{index}]\,;\, n := 0\,;$
11: $l := h.\texttt{size}\,;$
    else $n{+}{+}$ **end** ;
13: **until** $r = \textbf{null} \lor a = \textit{ADR}(r)\,;$
14: **return** $\textit{val}(r)$.

In order to prove correctness, we add between braces instructions that only modify auxiliary variables, like the specification variables X and rS and other auxiliary variables to be introduced later. The part between braces is comment for the implementation, it only serves in the proof of correctness. The private auxiliary variable *cnt* of type *Nat* counts the number of times (fS) is executed and serves to prove that (fS) is executed precisely once in every call of *find*.

This procedure matches the code of an ordinary find in a hash table with open addressing, except for the code at the condition $r = \textbf{done}$. This code is needed for the case that the value at address a has been copied, in which case the new table must be located. Locating the new table is carried out by the procedure *refresh*, which is discussed in Section 3.5. In line 7, the accessed hash table should be valid (see invariants *fi4* and *He4* in Appendix A). After *refresh* the local variables n, h and l must be reset, to restart the search in the new hash table. If the procedure terminates, the specifying atomic command (fS) has been executed precisely once (see invariant *Cn1*) and the return values of the specification and the implementation are equal (see invariant *Co1*). If the operation succeeds, the return value must be a valid entry currently associated with the given address in the current hash table. It is not evident but it has been proved that the linear search of the process executing *find* cannot be violated by other processes, i.e. no other process can *delete*, *insert*, or *rewrite* an entry associated with the same address (as what the process is looking for) in the region where the process has already searched.

We require that every valid hash table contains at least one entry **null** or **done**. Therefore, the local variable n in the procedure *find* never goes beyond the size of the hash table (see invariants *Cu1*, *fi4*, *fi5* and axiom *Ax2*). When the bound of the new hash table is tuned properly before use (see invariants *Ne7*, *Ne8*), the hash table will not be updated too frequently, and termination of the procedure *find* can be guaranteed.

3.4.4 Procedure *delete*

To some extent, deletion is similar to finding. Since r is a local variable to the procedure *delete*, we regard 18a and 18b as two parts of atomic instruction 18. If the entry is found in the table, then at line 18b this entry is overwritten with the designated element **del**.

$$
\begin{aligned}
&\textbf{proc } \textit{delete}(a : \textit{Address} \setminus \{0\}) : \textit{Bool} = \\
&\quad \textbf{local } r : \textit{EValue} \,;\, k, l, n : \textit{Nat} \,; \\
&\qquad h : \textbf{pointer to } \textit{Hashtable} \,;\, \textit{suc} : \textit{Bool} \,;
\end{aligned}
$$

15: $h := \mathrm{H}[\textit{index}]\,;\, \textit{suc} := \textit{false}\,;\, \{\textit{cnt} := 0\}\,;$
16: $l := h.\texttt{size}\,;\, n := 0\,;$
    repeat
17: $k := \textit{key}(a, l, n)\,;$
    $\langle\, r := h.\texttt{table}[k]\,;$
    $\{\textbf{if } r = \textbf{null} \textbf{ then } \textit{cnt}{+}{+}\,;\, (\text{dS}) \textbf{ end }\}\rangle;$
18a: **if** $\textit{oldp}(r)$ **then**
    $\textit{refresh}()\,;$
20: $h := \mathrm{H}[\textit{index}]\,;$
21: $l := h.\texttt{size}\,;\, n := 0\,;$
    elsif $a = \textit{ADR}(r)$ **then**
18b: $\langle\, \textbf{if } r = h.\texttt{table}[k] \textbf{ then}$
    $\textit{suc} := \textit{true};\, h.\texttt{table}[k] := \textbf{del};$
    $\{\, \textit{cnt}{+}{+}\,;\, (\text{dS})\,;\, \mathrm{Y}[k] := \textbf{del}\, \}$
    end $\rangle\,;$
    else $n{+}{+}$ **end** ;
    until $\textit{suc} \lor r = \textbf{null}\,;$
25: **if** $\textit{suc}$ **then** $h.\texttt{dels}{+}{+}$ **end** ;
26: **return** $\textit{suc}$.

The repetition in this procedure has two ways to terminate. Either deletion fails with $r = \textbf{null}$ in 17, or deletion succeeds with $r = h.\texttt{table}[k]$ in 18b. In the latter case, we have in one atomic statement a double access of the shared variable $h.\texttt{table}[k]$. This is a so-called compare&swap instruction. Atomicity is needed here to preclude interference. The specifying command (dS) is executed either in 17 or in 18b, and it is executed precisely once (see invariant *Cn2*), since in 18b the guard $a = \textit{ADR}(r)$ implies $r \neq \textbf{null}$ (see invariant *de1* and axiom *Ax1*).

In order to remember the address from the value rewritten to **done** after the value is being copied in the procedure *moveContents*, in 18, we introduce a new auxiliary shared variable Y of type array of *EValue*, whose contents equals the corresponding contents of the current hash table almost everywhere except that the values it contains are not tagged as *old* or rewritten as **done** (see invariants *Cu9*, *Cu10*).

Since we postpone the increment of $h.\texttt{dels}$ until line 25, the field dels is a lower bound of the number of positions deleted in the hash table (see invariant *Cu4*).

3.4.5 Procedure *insert*

The procedure for insertion in the table is given below. Basically, it is the standard algorithm for insertion in a hash table with open addressing. Notable is line 28 where the current process finds that the current hash table too full, and orders a new table to be made. We assume that $h.\texttt{bound}$ is a number less than $h.\texttt{size}$ (see invariant *Cu3*), which is tuned for optimal performance.

Furthermore, in line 35, it can be detected that values in the hash table have been marked *old*, which is a sign that hash table h is outdated, and the new hash table must be located to perform the insertion.

$$\textbf{proc } insert(v : Value \setminus \{\textbf{null}\}) : Bool =$$
$$\textbf{local } r : EValue \,;\, k, l, n : Nat \,;$$
$$h : \textbf{pointer to } Hashtable \,;$$
$$suc : Bool \,;\, a : Address := ADR(v) \,;$$

```
27:        h := H[index] ; {cnt := 0} ;
28:        if h.occ > h.bound then
               newTable() ;
30:            h := H[index] end ;
31:        n := 0 ; l := h.size ; suc := false ;
           repeat
32:            k := key(a, l, n) ;
33:            ⟨ r := h.table[k] ;
                 { if a = ADR(r) then cnt++ ;
                     (iS) end } ⟩ ;
35a:           if oldp(r) then
                   refresh() ;
36:                h := H[index] ;
37:                n := 0 ; l := h.size ;
               elsif r = null then
35b:               ⟨ if h.table[k] = null then
                       suc := true ; h.table[k] := v ;
                       { cnt++ ; (iS) ; Y[k] := v }
                   end ⟩ ;
               else n++ end ;
           until suc ∨ a = ADR(r) ;
41:        if suc then h.occ++ end ;
42:        return suc .
```

Instruction 35b is a version of compare&swap. Procedure *insert* terminates successfully when the insertion to an empty slot is completed, or it fails when there already exists an entry with the given address currently in the hash table (see invariant *Co3* and the specification of *insert*).

3.4.6 Procedure *assign*

Procedure *assign* is almost the same as *insert* except that it rewrites an entry with a given value even when the associated address is not empty. We provide it without further comments.

$$\textbf{proc } assign(v : Value \setminus \{\textbf{null}\}) =$$
$$\textbf{local } r : EValue \,;\, k, l, n : Nat \,;$$
$$h : \textbf{pointer to } Hashtable \,;$$
$$suc : Bool \,;\, a : Address := ADR(v) \,;$$

```
43:        h := H[index] ; {cnt := 0};
44:        if h.occ > h.bound then
               newTable() ;
46:            h := H[index] end ;
47:        n := 0 ; l := h.size ; suc := false ;
           repeat
48:            k := key(a, l, n) ;
49:            r := h.table[k] ;
50a:           if oldp(r) then
                   refresh() ;
51:                h := H[index] ;
52:                n := 0 ; l := h.size ;
               elsif r = null ∨ a = ADR(r) then
```

```
50b:               ⟨ if h.table[k] = r then
                       suc := true ; h.table[k] := v ;
                       { cnt++ ; (aS) ; Y[k] := v }
                   end ⟩
               else n++ end ;
           until suc ;
57:        if r = null then h.occ++ end ;
       end.
```

3.5 Memory management and concurrent migration

In this section, we provide the public procedures *getAccess* and *releaseAccess* and the auxiliary procedures *refresh* and *newTable* which are responsible for allocation and deallocation. We begin with the treatment of memory by providing a model of the heap.

3.5.1 The model of the heap

We *model* the Heap as an infinite array of hash tables, declared and initialized in the following way:

$$\text{Heap} : \textbf{array } Nat \textbf{ of } Hashtable := ([Nat]\bot) \,;$$
$$\text{H\_index} : Nat := 1 \,.$$

So, initially, $\text{Heap}[i] = \bot$ for all indices i. The indices of array Heap are the pointers to hash tables. We thus simply regard **pointer to** *Hashtable* as a synonym of *Nat*. Therefore, the notation $h.\text{table}$ used elsewhere in the paper stands for $\text{Heap}[h].\text{table}$. Since we reserve 0 (to be distinguished from the absent hash table $\bot$ and the absent value **null**) for the null pointer (i.e. $\text{Heap}[0] = \bot$, see invariant *He1*), we initialize H_index, which is the index of the next hash table, to be 2 instead of 1 or 0. Allocation of memory is modeled in

$$\textbf{proc } allocate(s, b : Nat) : Nat =$$
$$\langle \text{Heap}[\text{H\_index}] := \text{ blank hash table}$$
$$\text{with } \text{size} = s, \text{bound} = b,$$
$$\text{occ} = \text{dels} = 0 \,;$$
$$\text{H\_index++} \rangle \,;$$
$$\textbf{return } \text{H\_index} \,;$$

We assume that *allocate* sets all values in the hash table Heap [H_index] to **null**, and also sets its fields size and bound as specified. The variables occ and dels are set to 0 because the hash table is completely filled with the value **null**.

Deallocation of hash tables is modeled by

$$\textbf{proc } deAlloc(h : Nat) =$$
$$\langle \textbf{assert } \text{Heap}[h] \neq \bot \,;\; \text{Heap}[h] := \bot \rangle$$
$$\textbf{end} \,.$$

The **assert** here indicates the obligation to prove that *deAlloc* is called only for allocated memory.

3.5.2 Procedure *getAccess*

The procedure *getAccess* is defined as follows.

$$\textbf{proc } getAccess() =$$
$$\textbf{loop}$$
```
59:            index := currInd;
```

```
60:                    prot[index]++ ;
61:                    if  index = currInd  then
62:                        busy[index]++ ;
63:                        if  index = currInd  then return ;
                           else  releaseAccess(index)  end ;
65:                    else  prot[index]--  end ;
                   end
               end.
```

This procedure is a bit tricky. When the process reaches line
62, the *index* has been protected not to be used for creating a
new hash table in the procedure *newTable* (see invariants *pr2*,
pr3 and *nT12*).

The hash table pointer H[*index*] must contain the valid con-
tents after the procedure *getAccess* returns (see invariants *Ot3*,
He4). So, in line 62, busy is increased, guaranteeing that the
hash table will not inadvertently be destroyed (see invariant
bu1 and line 69). Line 63 needs to check the *index* again in
case that instruction 62 has the precondition that the hash table
is not valid. Once some process gets hold of one hash table
after calling *getAccess*, no process can throw it away until the
process releases it (see invariant *rA7*).

3.5.3 Procedure *releaseAccess*

The procedure *releaseAccess* is given by

```
          proc releaseAccess(i : 1 . . 2P) =
              local h : pointer to Hashtable ;
67:           h := H[i] ;
68:           busy[i]-- ;
69:           if  h ≠ 0 ∧ busy[i] = 0  then
70:               ⟨ if H[i] = h  then  H[i] := 0 ; ⟩
71:                    deAlloc(h) ;
                  end ;
              end ;
72:           prot[i]-- ;
          end.
```

The test $h \neq 0$ at 69 is necessary since it is possible that
$h = 0$ at the lines 68 and 69. This occurs e.g. in the following
scenario. Assume that process p is at line 62 with *index* $\neq$
currInd, while the number $i = index$ satisfies H[i] = 0
and busy[i] = 0. Then process p increments busy[i], calls
releaseAccess(i), and arrives at 68 with $h = 0$.

Since *deAlloc* in line 71 accesses a shared variable, we
have separated its call from 70. The counter busy[i] is used
to protect the hash table from premature deallocation. Only
if busy[i]=0, H[i] can be released. The main problem of the
design at this point is that it can happen that several processes
concurrently execute *releaseAccess* for the same value of i,
with interleaving just after the decrement of busy[i]. Then
they all may find busy[i] = 0. Therefore, a bigger atomic
command is needed to ensure that precisely one of them sets
H[i] to 0 (line 70) and calls *deAlloc*. Indeed, in line 71, *deAlloc*
is called only for allocated memory (see invariant *rA3*). The
counter prot[i] can be decreased since position i is no longer
used by this process.

3.5.4 Procedure *newTable*

When the current hash table has been used for some time, some
actions of the processes may require replacement of this hash
table. Procedure *newTable* is called when the number of occu-
pied positions in the current hash table exceeds the bound (see
lines 28, 44). Procedure *newTable* tries to allocate a new hash
table as the successor of the current one. If several processes
call *newTable* concurrently, they need to reach consensus on
the choice of an index for the next hash table (in line 84).
A newly allocated hash table that will not be used must be
deallocated again.

```
          proc newTable() =
              local i : 1 . . 2P ; b, bb : Bool ;
77:           while next[index] = 0  do
78:               choose i ∈ 1 . . 2P ;
                  ⟨ b := (prot[i] = 0) ;
                   if  b  then prot[i] := 1  end ⟩ ;
                  if  b  then
81:                   busy[i] := 1 ;
82:                   choose bound > H[index].bound
                             −H[index].dels + 2P ;
                      choose size > bound + 2P ;
                      H[i] := allocate(size, bound) ;
83:                   next[i] := 0 ;
84:                   ⟨ bb := (next[index] = 0) ;
                       if  bb  then  next[index] := i  end ⟩ ;
                      if  ¬bb  then  releaseAccess(i)  end ;
                  end end ;
                  refresh() ;
              end .
```

In command 82, we allocate a new blank hash table (see
invariant *nT8*), of which the bound is set greater than
H[*index*].bound − H[*index*].dels + 2P in order to avoid cre-
ating a too small hash table (see invariants *nT6*, *nT7*).

We require the size of a hash table to be more than
bound + 2P because of the following scenario: P processes
find "$h.occ > h.bound$" at line 28 and call *newtable*, *refresh*,
migrate, *moveContents* and *moveElement* one after the other.
After moving some elements, all processes but process p sleep
at line 126 with $b_{mE} = true$ (b_{mE} is the local variable b of
procedure *moveElement*). Process p continues the migration
and updates the new current index when the migration com-
pletes. Then, process p does several insertions to let the occ
of the current hash table reach one more than its bound. Just
at that moment, $P − 1$ processes wake up, increase the occ
of the current hash table to be $P − 1$ more, and return to line
30. Since $P − 1$ processes insert different values in the hash
table, after $P − 1$ processes finish their insertions, the occ of
the current hash table reaches $2P − 1$ more than its bound.

It may be useful to make size larger than bound + 2P
to avoid too many collisions, e.g. with a constraint size $\geq$
$\alpha \cdot$ bound for some $\alpha > 1$. If we did not introduce dels,
every migration would force the sizes to grow, so that our hash
table would require unbounded space for unbounded life time.
We introduced dels to avoid this.

Strictly speaking, instruction 82 inspects one shared vari-
able, H[*index*], and modifies three other shared variables, viz.
H[i], Heap[H_index], and H_index. In general, we split
such multiple shared variable accesses in separate atomic com-

mands. Here the accumulation is harmless, since the only possible interferences are with other allocations at line 82 and deallocations at line 71. In view of the invariant *Ha2*, all deallocations are at pointers $h < \texttt{H\_index}$. Allocations do not interfere because they contain the increment $\texttt{H\_index++}$ (see procedure *allocate*).

The procedure *newTable* first searches for a free index i, say by round robin. We use a nondeterministic choice. Once a free index has been found, a hash table is allocated and the index gets an indirection to the allocated address. Then the current index gets a next pointer to the new index, unless this pointer has been set already.

The variables $\texttt{prot}[i]$ are used primarily as counters with atomic increments and decrements. In 78, however, we use an atomic test-and-set instruction. Indeed, separation of this instruction in two atomic instructions is incorrect, since that would allow two processes to grab the same index i concurrently.

3.5.5 Procedure *migrate*

After the choice of the new hash table, the procedure *migrate* serves to transfer the contents in the current hash table to the new hash table by calling a procedure *moveContents* and to update the current hash table pointer afterwards. Migration is complete when at least one of the (parallel) calls to *migrate* has terminated.

```
      proc migrate() =
          local i : 0 . . 2P; h : pointer to Hashtable ;
              b : Bool ;
94:       i := next[index];
95:       prot[i]++ ;
97:       if index ≠ currInd then
98:           prot[i]-- ;
          else
99:           busy[i]++ ;
100:          h := H[i] ;
101:          if index = currInd then
                  moveContents(H[index], h) ;
103:              ⟨ b := (currInd = index) ;
                  if b then currInd := i ;
                  {Y := H[i].table }
                  end ⟩ ;
                  if b then
104:                  busy[index]-- ;
105:                  prot[index]-- ;
              end end ;
              releaseAccess(i) ;
          end end .
```

According to invariants *mi4* and *mi5*, it is an invariant that $i = next(index) \neq 0$ holds after instruction 94.

Line 103 contains a compare&swap instruction to update the current hash table pointer when some process finds that the migration is finished while $\texttt{currInd}$ is still identical to its *index*, which means that i is still used for the next current hash table (see invariant *mi5*). The increments of $\texttt{prot}[i]$ and $\texttt{busy}[i]$ here are needed to protect the next hash table. The decrements serve to avoid memory loss.

3.5.6 Procedure *refresh*

In order to avoid that a delayed process starts migration of an old hash table, we encapsulate *migrate* in *refresh* in the following way.

```
          proc refresh() =
90:           if index ≠ currInd then
                  releaseAccess(index) ;
                  getAccess() ;
              else migrate() end ;
          end.
```

When *index* is outdated, the process needs to call *releaseAccess* to abandon its hash table and *getAccess* to acquire the present pointer to the current hash table. Otherwise, the process can join the migration.

3.5.7 Procedure *moveContents*

Procedure *moveContents* has to move the contents of the current table to the next current table. All processes that have access to the table, may also participate in this migration. Indeed, they cannot yet use the new table (see invariants *Ne1* and *Ne3*). We have to take care that delayed actions on the current table and the new table are carried out or abandoned correctly (see invariants *Cu1* and *mE10*). Migration requires that every value in the current table be moved to a unique position in the new table (see invariant *Ne19*).

Procedure *moveContents* uses a private variable *toBeMoved* that ranges over sets of locations. The procedure is given by

```
          proc moveContents(from, to :
              pointer to Hashtable) =
              local i : Nat ; b : Bool ; v : EValue ;
                  toBeMoved : set of Nat ;
              toBeMoved := {0, . . . ,from.size − 1} ;
110:          while currInd = index∧
                  toBeMoved ≠ ∅ do
111:              choose i ∈ toBeMoved ;
                  v := from.table[i] ;
                  if v = done then
112:                  toBeMoved := toBeMoved − {i} ;
                  else
114:                  ⟨ b := (v = from.table[i]) ;
                      if b then from.table[i] :=
                      old(val(v)) end ⟩ ;
                      if b then
116:                      if val(v) ≠ null then
                              moveElement(val(v), to) end ;
117:                      from.table[i] := done ;
118:                      toBeMoved := toBeMoved − {i} ;
              end end end ;
          end .
```

Note that the value is tagged as outdated before it is copied (see invariant *mC11*). After tagging, the value cannot be deleted or assigned until the migration has been completed. Tagging must be done atomically, since otherwise an interleaving deletion may be lost. When indeed the value has been copied to the new hash table, it becomes **done** in the old hash

table in line 117. This has the effect that other processes need not wait for this process to complete procedure *moveElement*, but can help with the migration of this value if needed.

Since the address is lost after being rewritten to **done**, we had to introduce the shared auxiliary hash table Y to remember its value for the proof of correctness. This could have been avoided by introducing a second tagging bit, say for "very old".

The processes involved in the same migration should not use the same strategy for choosing i in line 111, since it is advantageous that *moveElement* is called often with different values. They may exchange information: any of them may replace its set *toBeMoved* by the intersection of that set with the set *toBeMoved* of another one. We do not give a preferred strategy here, one can refer to algorithms for the *write-all* problem [7, 16].

3.5.8 Procedure *moveElement*

The procedure *moveElement* moves a value to the new hash table. Note that the value is tagged as outdated in *moveContents* before *moveElement* is called.

$$\textbf{proc } moveElement(v : Value \setminus \{\textbf{null}\}, to :$$
$$\textbf{pointer to } Hashtable) =$$
$$\textbf{local } a : Address \; ; \; k, m, n : Nat \; ;$$
$$w : EValue \; ; \; b : Bool \; ;$$

120: $n := 0 \; ; \; b := \textit{false} \; ; \; a := ADR(v) \; ;$
       $m := to.\texttt{size} \; ;$
       repeat

121:       $k := key(a, m, n) \; ; \; w := to.\texttt{table}[k] \; ;$
           if $w = \textbf{null}$ **then**

123:           $\langle \; b := (to.\texttt{table}[k] = \textbf{null}); $
               if b **then** $to.\texttt{table}[k] := v$ **end** $\rangle \; ;$
           else n++ **end** ;

125:       **until** $b \lor a = ADR(w) \lor \texttt{currInd}$
                  $\neq index \; ;$

126:       **if** b **then** $to.\texttt{occ}$++ **end**
       end .

The value is only allowed to be inserted once in the new hash table (see invariant *Ne19*), since otherwise the main property of open addressing would be violated. In total, four situations can occur in the procedure moveElement:

- the current location k contains a value with a different address. The process increases n to inspect the next location.
- the current location k contains a value with the same address. This means that the value has already been copied to the new hash table, the process therefore terminates.
- the current location k is an empty slot. The process inserts v and returns. If insertion fails, since another process filled the empty slot in between, the search is continued.
- when *index* happens to differ from currInd, the entire migration has been completed.

While the current hash table pointer is not updated yet, there exists at least one **null** entry in the new hash table (see invariants *Ne8*, *Ne22* and *Ne23*), hence the local variable n in the procedure *moveElement* never goes beyond the size of the hash table (see invariants *mE3* and *mE8*), and the termination is thus guaranteed.

4 Correctness (safety)

In this section, we describe the proof of safety of the algorithm. The main aspects of safety are functional correctness, atomicity, and absence of memory loss. These aspects are formalized in eight invariants described in section 4.1. To prove these invariants, we need many other invariants. These are listed in Appendix A. In section 4.2, we sketch the verification of some of the invariants by informal means. In section 4.3, we describe how the theorem prover PVS is used in the verification. As exemplified in 4.2, Appendix B gives the dependencies between the invariants.

Notational Conventions. Recall that there are at most P processes with process identifiers ranging from 1 up to P. We use p, q, r to range over process identifiers, with a preference for p. Since the same program is executed by all processes, every private variable name of a process $\neq p$ is extended with the suffix "." + "process identifier". We do not do this for process p. So, e.g., the value of a private variable x of process q is denoted by $x.q$, but the value of x of process p is just denoted by x. In particular, $pc.q$ is the program location of process q. It ranges over all integer labels used in the implementation.

When local variables in different procedures have the same names, we add an abbreviation of the procedure name as a subscript to the name. We use the following abbreviations: *fi* for *find*, *del* for *delete*, *ins* for *insert*, *ass* for *assign*, *gA* for *getAccess*, *rA* for *releaseAccess*, *nT* for *newTable*, *mig* for *migrate*, *ref* for *refresh*, *mC* for *moveContents*, *mE* for *moveElement*.

In the implementation, there are several places where the same procedure is called, say *getAccess*, *releaseAccess*, etc. We introduce auxiliary private variables *return*, local to such a procedure, to hold the return location. We add a procedure subscript to distinguish these variables according to the above convention.

If V is a set, $\sharp V$ denotes the number of elements of V. If b is a boolean, then $\sharp b = 0$ when b is false, and $\sharp b = 1$ when b is true. Unless explicitly defined otherwise, we always (implicitly) universally quantify over addresses a, values v, non-negative integer numbers k, m, and n, natural number l, processes p, q and r. Indices i and j range over $[1, 2P]$. We abbreviate H(currInd).size as *curSize*.

In order to avoid using too many parentheses, we use the usual binding order for the operators. We give "$\land$" higher priority than "$\lor$". We use parentheses whenever necessary.

4.1 Main properties

We have proved the following three safety properties of the algorithm. Firstly, the access procedures *find*, *delete*, *insert*, *assign*, are functionally correct. Secondly they are executed atomically. The third safety property is absence of memory loss.

Functional correctness of *find*, *delete*, *insert* is the condition that the result of the implementation is the same as the result of the specification (fS), (dS), (iS). This is expressed by the required invariants:

Co1: $pc = 14 \Rightarrow val(r_{fi}) = rS_{fi}$
Co2: $pc \in \{25, 26\} \Rightarrow suc_{del} = sucS_{del}$
Co3: $pc \in \{41, 42\} \Rightarrow suc_{ins} = sucS_{ins}$

Note that functional correctness of *assign* holds trivially since it does not return a result.

According to the definition of atomicity in chapter 13 of [20], atomicity means that each execution of one of the access procedures contains precisely one execution of the corresponding specifying action (fS), (dS), (iS), (aS). We introduced the private auxiliary variables *cnt* to count the number of times the specifying action is executed. Therefore, atomicity is expressed by the invariants:

Cn1: $pc = 14 \Rightarrow cnt_{fi} = 1$
Cn2: $pc \in \{25, 26\} \Rightarrow cnt_{del} = 1$
Cn3: $pc \in \{41, 42\} \Rightarrow cnt_{ins} = 1$
Cn4: $pc = 57 \Rightarrow cnt_{ass} = 1$

We interpret absence of memory loss to mean that the number of allocated hash tables is bounded. More precisely, we prove that this number is bounded by $2P$. This is formalized in the invariant:

No1: $\sharp\{k \mid k < \texttt{H\_index} \wedge \texttt{Heap}(k) \neq \bot\} \leq 2P$

An important safety property is that no process accesses deallocated memory. Since most procedures perform memory accesses, by means of pointers that are local variables, the proof of this is based on a number of different invariants. Although, this is not explicit in the specification, it has been checked because the theorem prover PVS does not allow access to deallocated memory as this would violate type correctness conditions.

4.2 Intuitive proof

The eight correctness properties (invariants) mentioned above have been completely proved with the interactive proof checker of PVS. The use of PVS did not only take care of the delicate bookkeeping involved in the proof, it could also deal with many trivial cases automatically. At several occasions where PVS refused to let a proof be finished, we actually found a mistake and had to correct previous versions of this algorithm.

In order to give some feeling for the proof, we describe some proofs. For the complete mechanical proof, we refer the reader to [6]. Note that, for simplicity, we assume that all non-specific private variables in the proposed assertions belong to the general process p, and general process q is an active process that tries to threaten some assertion (p may equal q).

Proof of invariant *Co1* (as claimed in 4.1). According to Appendix B, the stability of *Co1* follows from the invariants *Ot3*, *fi1*, *fi10*, which are given in Appendix A. Indeed, *Ot3* implies that no procedure returns to location 14. Therefore all return statements falsify the antecedent of *Co1* and thus preserve *Co1*. Since r_{fi} and rS_{fi} are private variables to process p, *Co1* can only be violated by process p itself (establishing pc at 14) when p executes 13 with $r_{fi} = $ **null** $\vee a_{fi} = ADR(r_{fi})$. This condition is abbreviated as $Find(r_{fi}, a_{fi})$. Invariant *fi10* then implies that action 13 has the precondition $val(r_{fi}) = rS_{fi}$, so then it does not violate *Co1*. In PVS, we used a slightly different definition of *Find*, and we applied invariant *fi1* to exclude that r_{fi} is **done** or **del**, though invariant *fi1* is superfluous in this intuitive proof. □

Proof of invariant *Ot3*. Since the procedures *getAccess*, *releaseAccess*, *refresh*, *newTable* are called only at specific locations in the algorithm, it is easy to list the potential return addresses. Since the variables *return* are private to process p, they are not modified by other processes. Stability of *Ot3* follows from this. As we saw in the previous proof, *Ot3* is used to guarantee that no unexpected jumps occur. □

Proof of invariant *fi10*. According to Appendix B, we only need to use *fi9* and *Ot3*. Let us use the abbreviation $k = key(a_{fi}, l_{fi}, n_{fi})$. Since r_{fi} and rS_{fi} are both private variables, they can only be modified by process p when p is executing statement 7. We split this situation into two cases

1. with precondition $Find(h_{fi}.\texttt{table}[k], a_{fi})$
   After execution of statement 7, r_{fi} becomes $h_{fi}.\texttt{table}[k]$, and rS_{fi} becomes $\texttt{X}(a_{fi})$. By *fi9*, we get $val(r_{fi}) = rS_{fi}$. Therefore the validity of *fi10* is preserved.
2. otherwise.
   After execution of statement 7, r_{fi} becomes $h_{fi}.\texttt{table}[k]$, which then falsifies the antecedent of *fi10*. □

Proof of invariant *fi9*. According to Appendix B, we proved that *fi9* follows from *Ax2*, *fi1*, *fi3*, *fi4*, *fi5*, *fi8*, *Ha4*, *He4*, *Cu1*, *Cu9*, *Cu10*, and *Cu11*. We abbreviate $key(a_{fi}, l_{fi}, n_{fi})$ as k. We deduce $h_{fi} = \texttt{H}(index)$ from *fi4*, $\texttt{H}(index)$ is not $\bot$ from *He4*, and k is below $\texttt{H}(index).\texttt{size}$ from *Ax2*, *fi4* and *fi3*. We split the proof into two cases:

1. $index \neq \texttt{currInd}$: By *Ha4*, it follows that $\texttt{H}(index) \neq \texttt{H}(\texttt{currInd})$. Hence from *Cu1*, we obtain $h_{fi}.\texttt{table}[k] = $ **done**, which falsifies the antecedent of *fi9*.
2. $index = \texttt{currInd}$: By premise $Find(h_{fi}.\texttt{table}[k], a_{fi})$, we know that $h_{fi}.\texttt{table}[k] \neq $ **done** because of *fi1*. By *Cu9* and *Cu10*, we obtain $val(h_{fi}.\texttt{table}[k]) = val(\texttt{Y}[k])$. Hence it follows that $Find(\texttt{Y}[k], a_{fi})$. Using *fi8*, we obtain

$$\forall m < n_{fi} : \neg Find(\texttt{Y}[key(a_{fi}, curSize, m)], a_{fi})$$

We get n_{fi} is below *curSize* because of *fi5*. By *Cu11*, we conclude

$$\texttt{X}(a_{fi}) = val(h_{fi}.\texttt{table}[k]) \qquad\qquad □$$

4.3 The model in PVS

Our proof architecture (for one property) can be described as a dynamically growing tree in which each node is associated with an assertion. We start from a tree containing only one node, the proof goal, which characterizes some property of the system. We expand the tree by adding some new children via proper analysis of an unproved node (top-down approach, which requires a good understanding of the system). The validity of that unproved node is then reduced to the validity of its children and the validity of some less or equally deep nodes.

Normally, simple properties of the system are proved with appropriate precedence, and then used to help establish more complex ones. It is not a bad thing that some property that was taken for granted turns out to be not valid. Indeed, it may

uncover a defect of the algorithm, but in any case it leads to new insights in it.

We model the algorithm as a transition system [21], which is described in the language of PVS in the following way. As usual in PVS, states are represented by a record with a number of fields:

```
State : TYPE = [#
% global variables
    ...
    busy : [ range(2*P) → nat ],
    prot : [ range(2*P) → nat ],
    ...
% private variables:
    index : [ range(P) → range(2*P) ],
    ...
    pc : [ range(P) → nat ], % private program counters
    ...
% local variables of procedures,
    also private to each process:
% find
    a_find : [ range(P) → Address ],
    r_find : [ range(P) → EValue ],
    ...
% getAccess
    return_getAccess : [ range(P) → nat ],
    ...
#]
```

where *range*(P) stands for the range of integers from 1 to P.

Note that private variables are given with as argument a process identifier. Local variables are distinguished by adding their procedure's names as suffixes.

An action is a binary relation on states: it relates the state prior to the action to the state following the action. The system performed by a particular process is then specified by defining the precondition of each action as a predicate on the state and also the effect of each action in terms of a state transition. For example, line 5 of the algorithm is described in PVS as follows:

```
% corresponding to statement find5: h := H[index]; n := 0;
find5(i,s1,s2) : bool =
    pc(s1)(i)=5 AND
    s2 = s1 WITH [ (pc)(i) := 6,
                   (n_find)(i) := 0,
                   (h_find)(i) := H(s1)(index(s1)(i)) ]
```

where i is a process identifier, s1 is a pre-state, s2 is a post-state.

Since our algorithm is concurrent, the global transition relation is defined as the disjunction of all atomic actions.

```
% transition steps
step(i,s1,s2) : bool =
    find5(i,s1,s2) or find6(i,s1,s2) or ...
    delete15(i,s1,s2) or delete16(i,s1,s2) or ...
    ...
```

Stability for each invariant is proved by a PVS *Theorem* of the form:

```
% Theorem about the stability of invariant fi10
IV_fi10: THEOREM
    forall (u,v : state, q : range(P) ) :
        step(q,u,v) AND fi10(u) AND fi9(u) AND ot3(u)
        => fi10(v)
```

To ensure that all proposed invariants are stable, there is a global invariant INV, which is the conjunction of all proposed invariants.

```
% global invariant
INV(s:state) : bool =
    He3(s) and He4(s) and Cu1(s) and ...
    ...
% Theorem about the stability of the global invariant INV
IV_INV: THEOREM
    forall (u,v : state, q : range(P) ) :
        step(q,u,v) AND INV(u) => INV(v)
```

We define Init as all possible initial states, for which all invariants must be valid.

```
% initial state
Init: { s : state |
        (forall (p: range(P)):
            pc(s)(p)=0 and ...
            ...) and
        (forall (a: Address):
            X(s)(a)=null) and
        ...
        }
% The initial condition can be satisfied
    by the global invariant INV
IV_Init: THEOREM
    INV(Init)
```

The PVS code contains eleven preconditions to imply well-definedness: e.g. in *find7*, the hash table must be non-NIL and ℓ must be its size.

```
% corresponding to statement find7
find7(i,s1,s2) : bool =
    i?(Heap(s1)(h_find(s1)(i))) and
    l_find(s1)(i)=size(i_(Heap(s1)(h_find(s1)(i))))
    and pc(s1)(i)=7 and
    ...
```

All preconditions are allowed, since we can prove lock-freedom in the following form. In every state $s1$ that satisfies the global invariant, every process q can perform a step, i.e., there is a state $s2$ with $(s1, s2) \in step$ and $pc(s1, q) \neq pc(s2, q)$. This is expressed in PVS by

```
% theorem for lock-freedom
IV_prog: THEOREM
    forall (u: state, q: range(P) ) :
        INV(u) => exists (v: state): pc(u)(q) /= pc(v)(q)
                  and step(q,u,v)
```

5 Correctness (progress)

In this section, we prove that our algorithm is lock-free, and that it is wait-free for several subtasks. Recall that an algorithm is called *lock-free* if always at least some process will finish its task in a finite number of steps, regardless of delays or failures by other processes. This means that no process can block the applications of further operations to the data structure, although any particular operation need not terminate since a slow process can be passed infinitely often by faster processes. We say that an operation is *wait-free* if any process involved in that operation is guaranteed to complete it in a finite number of its own steps, regardless of the (in)activity of other processes.

5.1 The easy part of progress

It is clear that *releaseAccess* is wait-free. It follows that the wait-freedom of *migrate* depends on wait-freedom of *moveContents*. The loop of *moveContents* is clearly bounded. So, wait-freedom of *moveContents* depends on wait-freedom of *moveElement*. It has been proved that n is bounded by m in *moveElement* (see invariants *mE3* and *mE8*). Since, moreover, $to.\texttt{table}[k] \neq \textbf{null}$ is stable, the loop of *moveElement* is also bounded. This concludes the sketch that *migrate* is wait-free.

5.2 Progress of newTable

The main part of procedure *newTable* is wait-free. This can be shown informally, as follows. Since we can prove the condition $\texttt{next}(index) \neq 0$ is stable while process p stays in the region $[77, 84]$, once the condition $\texttt{next}(index) \neq 0$ holds, process p will exit *newTable* in a few rounds.

Otherwise, we may assume that p has precondition $\texttt{next}(index) = 0$ before executing line 78. By the invariant

Ne5: $\quad pc \in [1, 58] \;\vee\; pc \geq 62 \;\wedge\; pc \neq 65$
$\qquad\qquad \wedge\; \texttt{next}(index) = 0$
$\qquad\quad \Rightarrow\; index = \texttt{currInd}$

we get that $index = \texttt{currInd}$ holds and $\texttt{next}(\texttt{currInd}) = 0$ from the precondition. We define two sets of integers:

$$
\begin{aligned}
prSet1(i) \;=\; & \{ r \mid index.r = i \;\wedge\; pc.r \\
& \notin \{0, 59, 60\} \} \\
prSet2(i) \;=\; & \{ r \mid index.r = i \;\wedge\; pc.r \\
& \in \{104, 105\} \\
& \vee\; i_{rA}.r = i \;\wedge\; index.r \neq i \\
& \wedge\; pc.r \in [67, 72] \\
& \vee\; i_{nT}.r = i \;\wedge\; pc.r \in [81, 84] \\
& \vee\; i_{mig}.r = i \;\wedge\; pc.r \geq 97 \}
\end{aligned}
$$

and consider the sum $\sum_{i=1}^{2P}(\sharp(prSet1(i)) + \sharp(prSet2(i)))$. While process p is at line 78, the sum cannot exceed $2P - 1$ because there are only P processes around and process p contributes only once to the sum. It then follows from the pigeon hole principle that there exists $j \in [1, 2P]$ such that $\sharp(prSet1(j)) + \sharp(prSet2(j)) = 0$ and $j \neq index.p$. By the invariant

pr1: $\quad \texttt{prot}[j] = \sharp(prSet1(j)) + \sharp(prSet2(j))$
$\qquad\qquad + \sharp(\texttt{currInd} = j)$
$\qquad\qquad + \sharp(\texttt{next}(\texttt{currInd}) = j)$

we can get that $\texttt{prot}[j] = 0$ because of $j \neq index.p = \texttt{currInd}$.

While $\texttt{currInd}$ is constant, no process can modify $\texttt{prot}[j]$ for $j \neq \texttt{currInd}$ infinitely often. Therefore, if process p acts infinitely often and chooses its value i in 78 by round robin, process p exits the loop of *newTable* eventually. This shows that the main part of *newTable* is wait-free.

5.3 The failure of wait-freedom

Procedure *getAccess* is not wait-free. When the active clients keep changing the current index faster than the new client can observe it, the accessing client is doomed to starvation. In that case, however, the other processes repeatedly succeed. It follows that *getAccess*, *refresh*, and *newTable* are lock-free.

It may be possible to make a queue for the accessing clients which is emptied by a process in *newTable*. The accessing clients must however also be able to enter autonomously. This would at least add another layer of complications. We therefore prefer to treat this failure of wait-freedom as a performance issue that can be dealt with in practice by tuning the sizes of the hash tables.

According to the invariants *fi5*, *de8*, *in8* and *as6*, the primary procedures *find*, *delete*, *insert*, *assign* are loops bounded by $n \leq h.\texttt{size}$, and n is only reset to 0 during migration. If n is not reset to 0, it is incremented or stays constant. Indeed, the atomic **if** statements in 18b, 35b, and 50b have no **else** parts. In *delete* and *assign*, it is therefore possible that n stays constant without termination of the loop. Since *assign* can modify non-**null** elements of the table, it follows that *delete* and *assign* are not wait-free. This unbounded fruitless activity is possible only when *assign* actions of other processes repeatedly succeed. It follows that the primary procedures are lock-free. This concludes the argument that the system is lock-free.

6 Conclusions

Lock-free shared data objects are inherently resilient to halting failures and permit maximum parallelism. We have presented a new practical, lock-free algorithm for concurrently accessible hash tables, which promises more robust performance and reliability than a conventional lock-based implementation. Moreover, the new algorithm is dynamic in the sense that it allows the hash table to grow and shrink as needed.

The algorithm scales up linearly with the number of processes, provided the function *key* and the selection of i in line 111 are defined well. This is confirmed by some experiments where random values were stored, retrieved and deleted from the hash table. These experiments indicated that 10^6 insertions, deletions and finds per second and per processor are possible on an SGI powerchallenge with 250Mhz R12000 processors. This figure should only be taken as a rough indicator, since the performance of parallel processing is very much influenced by the machine architecture, the relative sizes of data structures compared to sizes of caches, and even the scheduling of processes on processors.

The correctness proof for our algorithm is noteworthy because of the extreme effort it took to finish it. Formal deduction by human-guided theorem proving can, in principle, verify any correct design, but doing so may require unreasonable amounts of effort, time, or skill. Though PVS provided great help for managing and reusing the proofs, we have to admit that the verification for our algorithm was very complicated due to the complexity of our algorithm. The total verification effort can roughly be estimated to consist of two man years excluding the effort in determining the algorithm and writing the documentation. The whole proof contains around 200 invariants. It takes an 1Ghz Pentium IV computer around two days to re-run an individual proof for one of the biggest invariants. Without suitable tool support like PVS, we even doubt if it would be possible to complete a reliable proof of such size and complexity.

It may well be possible to simplify the proof and reduce the number of invariants slightly, but we did not work on this. The complete version of the PVS specifications and the whole proof scripts can be found at [6]. Note that we simplified some definitions in the paper for the sake of presentation.

Appendix A: Invariants

We present here all invariants whose validity has been verified by the theorem prover PVS.

Conventions. We abbreviate

$$Find(\mathtt{r}, \mathtt{a}) \triangleq \mathtt{r} = \mathbf{null} \vee \mathtt{a} = ADR(\mathtt{r})$$
$$LeastFind(a, n) \triangleq$$
$$(\forall m < n : \neg Find(\mathtt{Y}[key(a, curSize, m)], a))$$
$$\wedge Find(\mathtt{Y}[key(a, curSize, n)], a))$$
$$LeastFind(h, a, n) \triangleq$$
$$(\forall m < n : \neg Find(h.\mathtt{table}$$
$$[key(a, h.\mathtt{size}, m)], a))$$
$$\wedge Find(h.\mathtt{table}[key(a, h.\mathtt{size}, n)], a))$$

Axioms on functions *key* and *ADR*

Ax1: $\quad v = \mathbf{null} \equiv ADR(v) = \mathbf{0}$
Ax2: $\quad 0 \leq key(a, l, k) < l$
Ax3: $\quad 0 \leq k < m < l \Rightarrow key(a, l, k) \neq key(a, l, m)$

Main correctness properties

Co1: $\quad pc = 14 \Rightarrow val(r_{fi}) = rS_{fi}$
Co2: $\quad pc \in \{25, 26\} \Rightarrow suc_{del} = sucS_{del}$
Co3: $\quad pc \in \{41, 42\} \Rightarrow suc_{ins} = sucS_{ins}$
Cn1: $\quad pc = 14 \Rightarrow cnt_{fi} = 1$
Cn2: $\quad pc \in \{25, 26\} \Rightarrow cnt_{del} = 1$
Cn3: $\quad pc \in \{41, 42\} \Rightarrow cnt_{ins} = 1$
Cn4: $\quad pc = 57 \Rightarrow cnt_{ass} = 1$

The absence of memory loss is shown by

No1: $\quad \sharp(nbSet1) \leq 2 * P$
No2: $\quad \sharp(nbSet1) = \sharp(nbSet2)$

where $nbSet1$ and $nbSet2$ are sets of integers, characterized by

$$nbSet1 = \{k \mid k < \mathtt{H\_index} \wedge \mathtt{Heap}(k) \neq \bot\}$$
$$nbSet2 = \{i \mid \mathtt{H}(i) \neq 0 \vee (\exists r : pc.r = 71 \wedge i_{rA}.r = i)\}$$

Further, we have the following definitions of sets of integers:

$$deSet1 = \{k \mid k < curSize \wedge \mathtt{Y}[k] = \mathbf{del}\}$$
$$deSet2 = \{r \mid index.r = \mathtt{currInd} \wedge pc.r = 25$$
$$\wedge suc_{del}.r\}$$
$$deSet3 = \{k \mid k < \mathtt{H}(\mathtt{next}(\mathtt{currInd})).\mathtt{size}$$
$$\wedge \mathtt{H}(\mathtt{next}(\mathtt{currInd})).\mathtt{table}[k] = \mathbf{del}\}$$
$$ocSet1 = \{r \mid index.r \neq \mathtt{currInd}$$
$$\vee pc.r \in [30, 41] \vee pc.r \in [46, 57]$$
$$\vee pc.r \in [59, 65] \wedge return_{gA}.r \geq 30$$
$$\vee pc.r \in [67, 72] \wedge (return_{rA}.r = 59$$
$$\wedge return_{gA}.r \geq 30$$
$$\vee return_{rA}.r = 90 \wedge return_{ref}.r \geq 30)$$
$$\vee (pc.r = 90 \vee pc.r \in [104, 105]) \wedge return_{ref}.r$$
$$\geq 30\}$$
$$ocSet2 = \{r \mid pc.r \geq 125 \wedge b_{mE}.r$$

$$\wedge to.r = \mathtt{H}(\mathtt{currInd})\}$$
$$ocSet3 = \{r \mid index.r = \mathtt{currInd} \wedge pc.r = 41$$
$$\wedge suc_{ins}.r$$
$$\vee index.r = \mathtt{currInd} \wedge pc.r = 57$$
$$\wedge r_{ass}.r = \mathbf{null}\}$$
$$ocSet4 = \{k \mid k < curSize \wedge val(\mathtt{Y}[k]) \neq \mathbf{null}\}$$
$$ocSet5 = \{k \mid k < \mathtt{H}(\mathtt{next}(\mathtt{currInd})).\mathtt{size}$$
$$\wedge val(\mathtt{H}(\mathtt{next}(\mathtt{currInd})).\mathtt{table}[k]) \neq \mathbf{null}\}$$
$$ocSet6 = \{k \mid k < \mathtt{H}(\mathtt{next}(\mathtt{currInd})).\mathtt{size}$$
$$\wedge \mathtt{H}(\mathtt{next}(\mathtt{currInd})).\mathtt{table}[k] \neq \mathbf{null}\}$$
$$ocSet7 = \{r \mid pc.r \geq 125 \wedge b_{mE}.r$$
$$\wedge to.r = \mathtt{H}(\mathtt{next}(\mathtt{currInd}))\}$$
$$prSet1(i) = \{r \mid index.r = i \wedge pc.r \notin \{0, 59, 60\}\}$$
$$prSet2(i) = \{r \mid index.r = i \wedge pc.r \in \{104, 105\}$$
$$\vee i_{rA}.r = i \wedge index.r \neq i \wedge pc.r \in [67, 72]$$
$$\vee i_{nT}.r = i \wedge pc.r \in [81, 84]$$
$$\vee i_{mig}.r = i \wedge pc.r \geq 97\}$$
$$prSet3(i) = \{r \mid index.r = i \wedge pc.r$$
$$\in [61, 65] \cup [104, 105]$$
$$\vee i_{rA}.r = i \wedge pc.r = 72$$
$$\vee i_{nT}.r = i \wedge pc.r \in [81, 82]$$
$$\vee i_{mig}.r = i \wedge pc.r \in [97, 98]\}$$
$$prSet4(i) = \{r \mid index.r = i \wedge pc.r \in [61, 65]$$
$$\vee i_{mig}.r = i \wedge pc.r \in [97, 98]\}$$
$$buSet1(i) = \{r \mid index.r = i$$
$$\wedge (pc.r \in [1, 58] \cup (62, 68] \wedge pc.r \neq 65$$
$$\vee pc.r \in [69, 72] \wedge return_{rA}.r > 59$$
$$\vee pc.r > 72)\}$$
$$buSet2(i) = \{r \mid index.r = i \wedge pc.r = 104$$
$$\vee i_{rA}.r = i \wedge index.r \neq i \wedge pc.r \in [67, 68]$$
$$\vee i_{nT}.r = i \wedge pc.r \in [82, 84]$$
$$\vee i_{mig}.r = i \wedge pc.r \geq 100\}$$

We have the following invariants concerning the Heap

He1: $\quad \mathtt{Heap}(0) = \bot$
He2: $\quad \mathtt{H}(i) \neq 0 \equiv \mathtt{Heap}(\mathtt{H}(i)) \neq \bot$
He3: $\quad \mathtt{Heap}(\mathtt{H}(\mathtt{currInd})) \neq \bot$
He4: $\quad pc \in [1, 58] \vee pc > 65 \wedge \neg(pc \in [67, 72]$
$$\wedge i_{rA} = index)$$
$$\Rightarrow \mathtt{Heap}(\mathtt{H}(index)) \neq \bot$$
He5: $\quad \mathtt{Heap}(\mathtt{H}(i)) \neq \bot \Rightarrow \mathtt{H}(i).\mathtt{size} \geq P$
He6: $\quad \mathtt{next}(\mathtt{currInd}) \neq 0 \Rightarrow$
$$\mathtt{Heap}(\mathtt{H}(\mathtt{next}(\mathtt{currInd}))) \neq \bot$$

Invariants concerning hash table pointers

Ha1: $\quad \mathtt{H\_index} > 0$
Ha2: $\quad \mathtt{H}(i) < \mathtt{H\_index}$
Ha3: $\quad i \neq j \wedge \mathtt{Heap}(\mathtt{H}(i)) \neq \bot \Rightarrow \mathtt{H}(i) \neq \mathtt{H}(j)$
Ha4: $\quad index \neq \mathtt{currInd} \Rightarrow \mathtt{H}(index) \neq \mathtt{H}(\mathtt{currInd})$

Invariants about counters for calling the specification.

Cn5: $\quad pc \in [6, 7] \Rightarrow cnt_{fi} = 0$
Cn6: $\quad pc \in [8, 13]$
$$\vee pc \in [59, 65] \wedge return_{gA} = 10$$
$$\vee pc \in [67, 72] \wedge (return_{rA} = 59$$
$$\wedge return_{gA} = 10$$
$$\vee return_{rA} = 90$$
$$\wedge return_{ref} = 10)$$
$$\vee pc \geq 90 \wedge return_{ref} = 10$$
$$\Rightarrow cnt_{fi} = \sharp(r_{fi} = \mathbf{null} \vee a_{fi} = ADR(r_{fi}))$$

Cn7: $pc \in [16, 21] \;\wedge\; pc \neq 18$
     $\vee\; pc \in [59, 65] \wedge return_{gA} = 20$
     $\vee\; pc \in [67, 72] \wedge (return_{rA} = 59$
                    $\wedge return_{gA} = 20$
                  $\vee\; return_{rA} = 90$
                    $\wedge return_{ref} = 20$
     $\vee\; pc \geq 90 \;\wedge\; return_{ref} = 20$
     $\Rightarrow\; cnt_{del} = 0$

Cn8: $pc = 18 \Rightarrow\; cnt_{del} = \sharp(r_{del} = \textbf{null})$

Cn9: $pc \in [28, 33]$
     $\vee\; pc \in [59, 65] \wedge return_{gA} = 30$
     $\vee\; pc \in [67, 72] \wedge (return_{rA} = 59$
                   $\wedge\; return_{gA} = 30$
                  $\vee\; return_{rA} = 77$
                   $\wedge\; return_{nT} = 30$
                  $\vee\; return_{rA} = 90$
                   $\wedge\; return_{ref} = 30$
     $\vee\; pc \in [77, 84] \wedge return_{nT} = 30$
     $\vee\; pc \geq 90 \;\wedge\; return_{ref} = 30$
     $\Rightarrow\; cnt_{ins} = 0$

Cn10: $pc \in [35, 37]$
     $\vee\; pc \in [59, 65] \wedge return_{gA} = 36$
     $\vee\; pc \in [67, 72] \wedge (return_{rA} = 59 \wedge return_{gA} = 36$
                   $\vee\; return_{rA} = 90 \wedge return_{ref} = 36$
     $\vee\; pc \geq 90 \;\wedge\; return_{ref} = 36$
     $\Rightarrow\; cnt_{ins} = \sharp(a_{ins} = ADR(r_{ins}) \vee suc_{ins})$

Cn11: $pc \in [44, 52]$
     $\vee\; pc \in [59, 65] \wedge return_{gA} \in \{46, 51\}$
     $\vee\; pc \in [67, 72] \wedge (return_{rA} = 59$
                   $\wedge\; return_{gA} \in \{46, 51\}$
                  $\vee\; return_{rA} = 77$
                   $\wedge\; return_{nT} = 46$
                  $\vee\; return_{rA} = 90$
                   $\wedge\; return_{ref} \in \{46, 51\}$
     $\vee\; pc \in [77, 84] \wedge return_{nT} = 46$
     $\vee\; pc \geq 90 \;\wedge\; return_{ref} \in \{46, 51\}$
     $\Rightarrow\; cnt_{ass}sign = 0$

Invariants about old hash tables, current hash table and the auxiliary hash table Y. Here, we universally quantify over all non-negative integers $n < curSize$.

Cu1: $\text{H}(index) \neq \text{H}(\texttt{currInd}) \;\wedge\; k < \text{H}(index).\texttt{size}$
     $\wedge\; (pc \in [1, 58] \vee pc > 65 \;\wedge\; \neg(pc \in [67, 72]$
     $\wedge\; i_{rA} = index)$
     $\Rightarrow\; \text{H}(index).\texttt{table}[k] = \textbf{done}$

Cu2: $\sharp(\{k \mid k < curSize \;\wedge\; \text{Y}[k] \neq \textbf{null}\}) < curSize$

Cu3: $\text{H}(\texttt{currInd}).\texttt{bound} + 2 * P < curSize$

Cu4: $\text{H}(\texttt{currInd}).\texttt{dels} + \sharp(deSet2) = \sharp(deSet1)$

Cu5:   Cu5 has been eliminated. The numbering has been
     kept, so as not to endanger the consistency
     with Appendix B and the PVS script.

Cu6: $\text{H}(\texttt{currInd}).\texttt{occ} + \sharp(ocSet1) + \sharp(ocSet2)$
     $\leq \text{H}(\texttt{currInd}).\texttt{bound} + 2 * P$

Cu7: $\sharp(\{k \mid k < curSize \;\wedge\; \text{Y}[k] \neq \textbf{null}\}$
     $= \text{H}(\texttt{currInd}).\texttt{occ} + \sharp(ocSet2) + \sharp(ocSet3)$

Cu8: $\texttt{next}(\texttt{currInd}) = 0 \Rightarrow$
     $\neg\; oldp(\text{H}(\texttt{currInd}).\texttt{table}[n])$

Cu9: $\neg(oldp(\text{H}(\texttt{currInd}).\texttt{table}[n])) \Rightarrow$
     $\text{H}(\texttt{currInd}).\texttt{table}[n] = \text{Y}[n]$

Cu10: $oldp(\text{H}(\texttt{currInd}).\texttt{table}[n])$
     $\wedge\; val(\text{H}(\texttt{currInd}).\texttt{table}[n]) \neq \textbf{null}$

     $\Rightarrow val(\text{H}(\texttt{currInd}).\texttt{table}[n]) = val(\text{Y}[n])$

Cu11: $LeastFind(a, n) \Rightarrow$
     $\text{X}(a) = val(\text{Y}[key(a, curSize, n)])$

Cu12: $\text{X}(a) = val(\text{Y}[key(a, curSize, n)]) \neq \textbf{null} \Rightarrow$
     $LeastFind(a, n)$

Cu13: $\text{X}(a) = val(\text{Y}[key(a, curSize, n)]) \neq \textbf{null}$
     $\wedge\; n \neq m < curSize$
     $\Rightarrow\; ADR(\text{Y}[key(a, curSize, m)]) \neq a$

Cu14: $\text{X}(a) = \textbf{null} \;\wedge\; val(\text{Y}[key(a, curSize, n)]) \neq \textbf{null}$
     $\Rightarrow\; ADR(\text{Y}[key(a, curSize, n)]) \neq a$

Cu15: $\text{X}(a) \neq \textbf{null}$
     $\Rightarrow \exists m < curSize : \text{X}(a)$
     $= val(\text{Y}[key(a, curSize, m)])$

Cu16: $\exists(f : [\{m : 0 \leq m < curSize) \wedge val(\text{Y}[m]) \neq \textbf{null}\}$
     $\rightarrow \{v : v \neq \textbf{null} \wedge (\exists k < curSize : v$
     $= val(\text{Y}[k]))\}]) :$
     f is bijective

Invariants about `next` and `next(currInd)`:

Ne1: $\texttt{currInd} \neq \texttt{next}(\texttt{currInd})$

Ne2: $\texttt{next}(\texttt{currInd}) \neq 0 \Rightarrow$
     $\texttt{next}(\texttt{next}(\texttt{currInd})) = 0$

Ne3: $pc \in [1, 59] \vee pc \geq 62 \;\wedge\; pc \neq 65 \Rightarrow$
     $index \neq \texttt{next}(\texttt{currInd})$

Ne4: $pc \in [1, 58] \vee pc \geq 62 \;\wedge\; pc \neq 65 \Rightarrow$
     $index \neq \texttt{next}(index)$

Ne5: $pc \in [1, 58] \vee pc \geq 62 \;\wedge\; pc \neq 65$
     $\wedge\; \texttt{next}(index) = 0$
     $\Rightarrow\; index = \texttt{currInd}$

Ne6: $\texttt{next}(\texttt{currInd}) \neq 0$
     $\Rightarrow\; \sharp(ocSet6) \leq \sharp(\{k \mid k < curSize \;\wedge\; \text{Y}[k] \neq \textbf{null}\}$
     $- \text{H}(\texttt{currInd}).\texttt{dels} - \sharp(deSet2)$

Ne7: $\texttt{next}(\texttt{currInd}) \neq 0$
     $\Rightarrow\; \text{H}(\texttt{currInd}).\texttt{bound} - \text{H}(\texttt{currInd}).\texttt{dels}$
     $+ 2 * P \leq \text{H}(\texttt{next}(\texttt{currInd})).\texttt{bound}$

Ne8: $\texttt{next}(\texttt{currInd}) \neq 0$
     $\Rightarrow\; \text{H}(\texttt{next}(\texttt{currInd})).\texttt{bound}$
     $+ 2 * P < \text{H}(\texttt{next}(\texttt{currInd})).\texttt{size}$

Ne9: $\texttt{next}(\texttt{currInd}) \neq 0 \Rightarrow$
     $\text{H}(\texttt{next}(\texttt{currInd})).\texttt{dels} = \sharp(deSet3)$

Ne9a: $\texttt{next}(\texttt{currInd}) \neq 0 \Rightarrow$
     $\text{H}(\texttt{next}(\texttt{currInd})).\texttt{dels} = 0$

Ne10: $\texttt{next}(\texttt{currInd}) \neq 0 \;\wedge\; k < h.\texttt{size} \Rightarrow$
     $h.\texttt{table}[k] \notin \{\textbf{del}, \textbf{done}\},$
     where $h = \text{H}(\texttt{next}(\texttt{currInd}))$

Ne11: $\texttt{next}(\texttt{currInd}) \neq 0 \;\wedge\; k$
     $< \text{H}(\texttt{next}(\texttt{currInd})).\texttt{size}$
     $\Rightarrow\; \neg oldp(\text{H}(\texttt{next}(\texttt{currInd})).\texttt{table}[k])$

Ne12: $k < curSize \;\wedge\; \text{H}(\texttt{currInd}).\texttt{table}[k] = \textbf{done}$
     $\wedge\; m < h.\texttt{size}$
     $\wedge\; LeastFind(h, a, m)$
     $\Rightarrow\; \text{X}(a) = val(h.\texttt{table}[key(a, h.\texttt{size}, m)]),$
     where $a = ADR(\text{Y}[k])$ and
     $h = \text{H}(\texttt{next}(\texttt{currInd})))$

Ne13: $k < curSize \;\wedge\; \text{H}(\texttt{currInd}).\texttt{table}[k] = \textbf{done}$
     $\wedge\; m < h.\texttt{size}$
     $\wedge\; \text{X}(a) = val(h.\texttt{table}[key(a, h.\texttt{size}, m)]) \neq \textbf{null}$
     $\Rightarrow\; LeastFind(h, a, m),$
     where $a = ADR(\text{Y}[k])$ and $h = \text{H}(\texttt{next}(\texttt{currInd}))$

Ne14: $\texttt{next}(\texttt{currInd}) \neq 0 \;\wedge\; a \neq \textbf{0} \;\wedge\; k < h.\texttt{size}$
     $\wedge\; \text{X}(a) = val(h.\texttt{table}[key(a, h.\texttt{size}, k)]) \neq \textbf{null}$

$\Rightarrow LeastFind(h, a, k)$,
where $h = \mathrm{H}(\mathrm{next}(\mathrm{currInd}))$

Ne15: $k < curSize \wedge \mathrm{H}(\mathrm{currInd}).\mathtt{table}[k]$
$= \mathbf{done} \wedge \mathrm{X}(a) \neq \mathbf{null}$
$\wedge\ m < h.\mathtt{size}$
$\wedge\ \mathrm{X}(a) = val(h.\mathtt{table}[key(a, h.\mathtt{size}, m)])$
$\wedge\ n < h.\mathtt{size} \wedge m \neq n$
$\Rightarrow ADR(h.\mathtt{table}.[key(a, h.\mathtt{size}, n)]) \neq a$,
where $a = ADR(\mathrm{Y}[k])$ and $h = \mathrm{H}(\mathrm{next}(\mathrm{currInd}))$

Ne16: $k < curSize \wedge \mathrm{H}(\mathrm{currInd}).\mathtt{table}[k] = \mathbf{done}$
$\wedge\ \mathrm{X}(a) = \mathbf{null}$
$\wedge\ m < h.\mathtt{size}$
$\Rightarrow val(h.\mathtt{table}[key(a, h.\mathtt{size}, m)]) = \mathbf{null}$
$\vee ADR(h.\mathtt{table}[key(a, h.\mathtt{size}, m)]) \neq a$,
where $a = ADR(\mathrm{Y}[k])$ and $h = \mathrm{H}(\mathrm{next}(\mathrm{currInd}))$

Ne17: $\mathrm{next}(\mathrm{currInd}) \neq 0 \wedge m < h.\mathtt{size}$
$\wedge\ a = ADR(h.\mathtt{table}[m]) \neq 0$
$\Rightarrow \mathrm{X}(a) = val(h.\mathtt{table}[m]) \neq \mathbf{null}$,
where $h = \mathrm{H}(\mathrm{next}(\mathrm{currInd}))$

Ne18: $\mathrm{next}(\mathrm{currInd}) \neq 0 \wedge m < h.\mathtt{size}$
$\wedge\ a = ADR(h.\mathtt{table}[m]) \neq 0$
$\Rightarrow \exists n < curSize : val(\mathrm{Y}[n]) = val(h.\mathtt{table}[m])$
$\wedge\ oldp(\mathrm{H}(\mathrm{currInd}).\mathtt{table}[n])$,
where $h = \mathrm{H}(\mathrm{next}(\mathrm{currInd}))$

Ne19: $\mathrm{next}(\mathrm{currInd}) \neq 0 \wedge m < h.\mathtt{size} \wedge m \neq n$
$< h.\mathtt{size}$
$\wedge\ a = ADR(h.\mathtt{table}[key(a, h.\mathtt{size}, m)]) \neq 0$
$\Rightarrow ADR(h.\mathtt{table}[key(a, h.\mathtt{size}, n)]) \neq a$,
where $h = \mathrm{H}(\mathrm{next}(\mathrm{currInd}))$

Ne20: $k < curSize \wedge \mathrm{H}(\mathrm{currInd}).\mathtt{table}[k] = \mathbf{done}$
$\wedge\ \mathrm{X}(a) \neq \mathbf{null}$
$\Rightarrow \exists m < h.\mathtt{size} : \mathrm{X}(a)$
$= val(h.\mathtt{table}[key(a, h.\mathtt{size}, m)])$,
where $a = ADR(\mathrm{Y}[k])$ and $h = \mathrm{H}(\mathrm{next}(\mathrm{currInd}))$

Ne21: *Ne21 has been eliminated.*

Ne22: $\mathrm{next}(\mathrm{currInd}) \neq 0$
$\Rightarrow \sharp(ocSet6) = \mathrm{H}(\mathrm{next}(\mathrm{currInd})).\mathtt{occ}$
$+ \sharp(ocSet7)$

Ne23: $\mathrm{next}(\mathrm{currInd}) \neq 0$
$\Rightarrow \mathrm{H}(\mathrm{next}(\mathrm{currInd})).\mathtt{occ}$
$\leq \mathrm{H}(\mathrm{next}(\mathrm{currInd})).\mathtt{bound}$

Ne24: $\mathrm{next}(\mathrm{currInd}) \neq 0 \Rightarrow \sharp(ocSet5) \leq \sharp(ocSet4)$

Ne25: $\mathrm{next}(\mathrm{currInd}) \neq 0$
$\Rightarrow \exists(f : [\{m : 0 \leq m < h.\mathtt{size}$
$\wedge\ val(h.\mathtt{table}[m]) \neq \mathbf{null}\} \rightarrow$
$\{v : v \neq \mathbf{null} \wedge (\exists k < h.\mathtt{size} : v$
$= val(h.\mathtt{table}[k]))\}]) :$
f is bijective,
where $h = \mathrm{H}(\mathrm{next}(\mathrm{currInd}))$

Ne26: $\mathrm{next}(\mathrm{currInd}) \neq 0$
$\Rightarrow \exists(f : [\{v : v \neq \mathbf{null} \wedge (\exists m < h.\mathtt{size} : v$
$= val(h.\mathtt{table}[m]))\} \rightarrow$
$\{v : v \neq \mathbf{null} \wedge (\exists k :< curSize : v$
$= val(\mathrm{Y}[k]))\}]) :$
f is injective,
where $h = \mathrm{H}(\mathrm{next}(\mathrm{currInd}))$

Ne27: $\mathrm{next}(\mathrm{currInd}) \neq 0$
$\wedge\ (\exists n < h.\mathtt{size} : val(h.\mathtt{table}[n]) \neq \mathbf{null})$
$\Rightarrow \exists(f : [\{m : 0 \leq m < h.\mathtt{size}$
$\wedge val(h.\mathtt{table}[m]) \neq \mathbf{null}\} \rightarrow$
$\{k : 0 \leq k < curSize \wedge val(\mathrm{Y}[k])$

$\neq \mathbf{null}\}])$
f is injective,
where $h = \mathrm{H}(\mathrm{next}(\mathrm{currInd}))$

Invariants concerning procedure *find* (5...14)

fi1: $a_{fi} \neq \mathbf{0}$
fi2: $pc \in \{6, 11\} \Rightarrow n_{fi} = 0$
fi3: $pc \in \{7, 8, 13\} \Rightarrow l_{fi} = h_{fi}.\mathtt{size}$
fi4: $pc \in [6, 13] \wedge pc \neq 10 \Rightarrow h_{fi} = \mathrm{H}(index)$
fi5: $pc = 7 \wedge h_{fi} = \mathrm{H}(\mathrm{currInd}) \Rightarrow n_{fi} < curSize$
fi6: $pc = 8 \wedge h_{fi} = \mathrm{H}(\mathrm{currInd}) \wedge \neg Find(r_{fi}, a_{fi})$
$\wedge\ r_{fi} \neq \mathbf{done}$
$\Rightarrow \neg Find(\mathrm{Y}[key(a_{fi}, curSize, n_{fi})], a_{fi})$
fi7: $pc = 13 \wedge h_{fi} = \mathrm{H}(\mathrm{currInd}) \wedge \neg Find(r_{fi}, a_{fi})$
$\wedge\ m < n_{fi}$
$\Rightarrow \neg Find(\mathrm{Y}[key(a_{fi}, curSize, m)], a_{fi})$
fi8: $pc \in \{7, 8\} \wedge h_{fi} = \mathrm{H}(\mathrm{currInd}) \wedge m < n_{fi}$
$\Rightarrow \neg Find(\mathrm{Y}[key(a_{fi}, curSize, m)], a_{fi})$
fi9: $pc = 7 \wedge Find(t, a_{fi}) \Rightarrow \mathrm{X}(a_{fi}) = val(t)$,
where $t = h_{fi}.\mathtt{table}[key(a_{fi}, l_{fi}, n_{fi})]$
fi10: $pc \notin (1, 7] \wedge Find(r_{fi}, a_{fi}) \Rightarrow val(r_{fi}) = rS_{fi}$
fi11: $pc = 8 \wedge oldp(r_{fi}) \wedge index = \mathrm{currInd}$
$\Rightarrow \mathrm{next}(\mathrm{currInd}) \neq 0$

Invariants concerning procedure *delete* (15...26)

de1: $a_{del} \neq \mathbf{0}$
de2: $pc \in \{17, 18\} \Rightarrow l_{del} = h_{del}.\mathtt{size}$
de3: $pc \in [16, 25] \wedge pc \neq 20 \Rightarrow h_{del} = \mathrm{H}(index)$
de4: $pc = 18 \Rightarrow k_{del} = key(a_{del}, l_{del}, n_{del})$
de5: $pc \in \{16, 17\} \vee Deleting \Rightarrow \neg suc_{del}$
de6: $Deleting \wedge sucS_{del} \Rightarrow r_{del} \neq \mathbf{null}$
de7: $pc = 18 \wedge \neg oldp(h_{del}.\mathtt{table}[k_{del}]) \Rightarrow$
$h_{del} = \mathrm{H}(\mathrm{currInd})$
de8: $pc \in \{17, 18\} \wedge h_{del} = \mathrm{H}(\mathrm{currInd}) \Rightarrow$
$n_{del} < curSize$
de9: $pc = 18 \wedge h_{del} = \mathrm{H}(\mathrm{currInd})$
$\wedge\ (val(r_{del}) \neq \mathbf{null} \vee r_{del} = \mathbf{del})$
$\Rightarrow r \neq \mathbf{null} \wedge (r = \mathbf{del} \vee ADR(r) = ADR(r_{del}))$,
where $r = \mathrm{Y}[key(a_{del}, h_{del}.\mathtt{size}, n_{del})]$
de10: $pc \in \{17, 18\} \wedge h_{del} = \mathrm{H}(\mathrm{currInd})$
$\wedge\ m < n_{del})$
$\Rightarrow \neg Find(\mathrm{Y}[key(a_{del}, curSize, m)], a_{del})$
de11: $pc \in \{17, 18\} \wedge Find(t, a_{del}) \Rightarrow \mathrm{X}(a_{del}) = val(t)$,
where $t = h_{del}.\mathtt{table}[key(a_{del}, l_{del}, n_{del})]$
de12: $pc = 18 \wedge oldp(r_{del}) \wedge index = \mathrm{currInd}$
$\Rightarrow \mathrm{next}(\mathrm{currInd}) \neq 0$
de13: $pc = 18 \Rightarrow k_{del} < \mathrm{H}(index).\mathtt{size}$

where $Deleting$ is characterized by

$Deleting \equiv$
     $pc \in [18, 21] \vee pc \in [59, 65] \wedge return_{gA} = 20$
     $\vee pc \in [67, 72] \wedge (return_{rA} = 59$
                           $\wedge return_{gA} = 20$
                           $\vee return_{rA} = 90$
                           $\wedge return_{ref} = 20)$
     $\vee pc \geq 90 \wedge return_{ref} = 20$

Invariants concerning procedure *insert* (27...52)

in1: $a_{ins} = ADR(v_{ins}) \wedge v_{ins} \neq \mathbf{null}$
in2: $pc \in [32, 35] \Rightarrow l_{ins} = h_{ins}.\mathtt{size}$

in3: $pc \in [28, 41] \ \wedge \ pc \notin \{30, 36\} \Rightarrow \ h_{ins} = \text{H}(index)$
in4: $pc \in \{33, 35\} \Rightarrow \ k_{ins} = key(a_{ins}, l_{ins}, n_{ins})$
in5: $pc \in [32, 33] \vee \ Inserting \Rightarrow \ \neg suc_{ins}$
in6: $Inserting \ \wedge \ sucS_{ins} \Rightarrow \ ADR(r_{ins}) \neq a_{ins}$
in7: $pc = 35 \ \wedge \ \neg \, oldp(h_{ins}.\texttt{table}[k_{ins}]) \Rightarrow$
       $h_{ins} = \text{H}(\texttt{currInd})$
in8: $pc \in \{33, 35\} \ \wedge \ h_{ins} = \text{H}(\texttt{currInd}) \Rightarrow$
       $n_{ins} < curSize$
in9: $pc = 35 \ \wedge \ h_{ins} = \text{H}(\texttt{currInd})$
       $\wedge \ (val(r_{ins}) \neq \textbf{null} \vee r_{ins} = \textbf{del})$
       $\Rightarrow r \neq \textbf{null} \ \wedge \ (r = \textbf{del} \vee ADR(r) = ADR(r_{ins})),$
       where $r = \text{Y}[key(a_{ins}, h_{ins}.size, n_{ins})]$
in10: $pc \in \{32, 33, 35\}$
       $\wedge \ h_{ins} = \text{H}(\texttt{currInd}) \ \wedge \ m < n_{ins}$
       $\Rightarrow \ \neg Find(\text{Y}[key(a_{ins}, curSize, m)], a_{ins})$
in11: $pc \in \{33, 35\} \ \wedge \ Find(t, a_{ins}) \Rightarrow$
       $\text{X}(a_{ins}) = val(t),$
       where $t = h_{ins}.\texttt{table}[key(a_{ins}, l_{ins}, n_{ins})]$
in12: $pc = 35 \ \wedge \ oldp(r_{ins}) \ \wedge \ index = \texttt{currInd}$
       $\Rightarrow next(\texttt{currInd}) \neq 0$
in13: $pc = 35 \Rightarrow \ k_{ins} < \text{H}(index).\texttt{size}$

where $Inserting$ is characterized by

$Inserting \ \equiv$
       $pc \in [35, 37] \vee pc \in [59, 65] \ \wedge \ return_{gA} = 36$
       $\vee \ pc \in [67, 72] \ \wedge \ (return_{rA} = 59$
                       $\wedge \ return_{gA} = 36$
                       $\vee \ return_{rA} = 90$
                       $\wedge \ return_{ref} = 36)$
       $\vee \ pc \geq 90 \ \wedge \ return_{ref} = 36$

Invariants concerning procedure *assign* (43. . . 57)

as1: $a_{ass} = ADR(v_{ass}) \ \wedge \ v_{ass} \neq \textbf{null}$
as2: $pc \in [48, 50] \Rightarrow \ l_{ass} = h_{ass}.\texttt{size}$
as3: $pc \in [44, 57] \ \wedge \ pc \notin \{46, 51\} \Rightarrow \ h_{ass} = \text{H}(index)$
as4: $pc \in \{49, 50\} \Rightarrow \ k_{ass} = key(a_{ass}, l_{ass}, n_{ass})$
as5: $pc = 50 \ \wedge \ \neg \, oldp(h_{ass}.\texttt{table}[k_{ass}]) \Rightarrow$
       $h_{ass} = \text{H}(\texttt{currInd})$
as6: $pc = 50 \ \wedge \ h_{ass} = \text{H}(\texttt{currInd}) \Rightarrow \ n_{ass} < curSize$
as7: $pc = 50 \ \wedge \ h_{ass} = \text{H}(\texttt{currInd})$
       $\wedge \ (val(r_{ass}) \neq \textbf{null} \vee r_{ass} = \textbf{del})$
       $\Rightarrow r \neq \textbf{null} \ \wedge \ (r = \textbf{del} \vee ADR(r) = ADR(r_{ass})),$
       where $r = \text{Y}[key(a_{ass}, h_{ass}.size, n_{ass})]$
as8: $pc \in \{48, 49, 50\}$
       $\wedge \ h_{ass} = \text{H}(\texttt{currInd}) \ \wedge \ m < n_{ass}$
       $\Rightarrow \ \neg Find(\text{Y}[key(a_{ass}, curSize, m)], a_{ass})$
as9: $pc = 50 \ \wedge \ Find(t, a_{ass}) \Rightarrow \ \text{X}(a_{ass}) = val(t),$
       where $t = h_{ass}.\texttt{table}[key(a_{ass}, l_{ass}, n_{ass})]$
as10: $pc = 50 \ \wedge \ oldp(r_{ass}sign) \ \wedge \ index = \texttt{currInd}$
       $\Rightarrow next(\texttt{currInd}) \neq 0$
as11: $pc = 50 \Rightarrow \ k_{ass} < \text{H}(index).\texttt{size}$

Invariants concerning procedure *releaseAccess* (67. . . 72)

rA1: $h_{rA} < \text{H\_index}$
rA2: $pc \in [70, 71] \Rightarrow \ h_{rA} \neq 0$
rA3: $pc = 71 \Rightarrow \ Heap(h_{rA}) \neq \bot$
rA4: $pc = 71 \Rightarrow \ \text{H}(i_{rA}) = 0$
rA5: $pc = 71 \Rightarrow \ h_{rA} \neq \text{H}(i)$

rA6: $pc = 70 \Rightarrow \ \text{H}(i_{rA}) \neq \text{H}(\texttt{currInd})$
rA7: $pc = 70$
       $\wedge \ (pc.r \in [1, 58] \vee pc.r > 65 \wedge$
       $\neg(pc.r \in [67, 72] \wedge i_{rA}.r = index.r))$
       $\Rightarrow \text{H}(i_{rA}) \neq \text{H}(index.r)$
rA8: $pc = 70 \Rightarrow \ i_{rA} \neq next(\texttt{currInd})$
rA9: $pc \in [68, 72] \ \wedge \ (h_{rA} = 0 \vee h_{rA} \neq \text{H}(i_{rA}))$
       $\Rightarrow \text{H}(i_{rA}) = 0$
rA10: $pc \in [67, 72] \ \wedge \ return_{rA} \in \{0, 59\} \Rightarrow$
       $i_{rA} = index$
rA11: $pc \in [67, 72] \ \wedge \ return_{rA} \in \{77, 90\} \Rightarrow$
       $i_{rA} \neq index$
rA12: $pc \in [67, 72] \ \wedge \ return_{rA} = 77 \Rightarrow$
       $next(index) \neq 0$
rA13: $pc = 71 \ \wedge \ pc.r = 71 \ \wedge \ p \neq r \Rightarrow \ h_{rA} \neq h_{rA}.r$
rA14: $pc = 71 \ \wedge \ pc.r = 71 \ \wedge \ p \neq r \Rightarrow \ i_{rA} \neq i_{rA}.r$

Invariants concerning procedure *newTable* (77. . . 84)

nT1: $pc \in [81, 82] \Rightarrow \ Heap(\text{H}(i_{nT})) = \bot$
nT2: $pc \in [83, 84] \Rightarrow \ Heap(\text{H}(i_{nT})) \neq \bot$
nT3: $pc = 84 \Rightarrow \ next(i_{nT}) = 0$
nT4: $pc \in [83, 84] \Rightarrow \ \text{H}(i_{nT}).\texttt{dels} = 0$
nT5: $pc \in [83, 84] \Rightarrow \ \text{H}(i_{nT}).\texttt{occ} = 0$
nT6: $pc \in [83, 84] \Rightarrow$
       $\text{H}(i_{nT}).\texttt{bound} + 2 * P < \text{H}(i_{nT}).\texttt{size}$
nT7: $pc \in [83, 84] \ \wedge \ index = \texttt{currInd}$
       $\Rightarrow \ \text{H}(\texttt{currInd}).\texttt{bound}$
       $-\text{H}(\texttt{currInd}).\texttt{dels} + 2 * P < \text{H}(i_{nT}).\texttt{bound}$
nT8: $pc \in [83, 84] \ \wedge \ k < \text{H}(i_{nT}).\texttt{size} \Rightarrow$
       $\text{H}(i_{nT}).\texttt{table}[k] = \textbf{null}$
nT9: $pc \in [81, 84] \Rightarrow \ i_{nT} \neq \texttt{currInd}$
nT10: $pc \in [81, 84] \ \wedge \ (pc.r \in [1, 58] \vee pc.r$
                       $\geq 62 \ \wedge \ pc.r \neq 65)$
       $\Rightarrow \ i_{nT} \neq index.r$
nT11: $pc \in [81, 84] \Rightarrow \ i_{nT} \neq next(\texttt{currInd})$
nT12: $pc \in [81, 84] \Rightarrow \ \text{H}(i_{nT}) \neq \text{H}(\texttt{currInd})$
nT13: $pc \in [81, 84]$
       $\wedge \ (pc.r \in [1, 58] \vee pc.r > 65 \wedge \neg(pc.r \in [67, 72]$
                       $\wedge i_{rA}.r = index.r))$
       $\Rightarrow \ \text{H}(i_{nT}) \neq \text{H}(index.r)$
nT14: $pc \in [81, 84] \ \wedge \ pc.r \in [67, 72] \Rightarrow \ i_{nT} \neq i_{rA}.r$
nT15: $pc \in [83, 84] \ \wedge \ pc.r \in [67, 72] \Rightarrow$
       $\text{H}(i_{nT}) \neq \text{H}(i_{rA}.r)$
nT16: $pc \in [81, 84] \ \wedge \ pc.r \in [81, 84] \ \wedge \ p \neq r \Rightarrow$
       $i_{nT} \neq i_{nT}.r$
nT17: $pc \in [81, 84] \ \wedge \ pc.r \in [95, 99]$
       $\wedge \ index.r = \texttt{currInd}$
       $\Rightarrow \ i_{nT} \neq i_{mig}.r$
nT18: $pc \in [81, 84] \ \wedge \ pc.r \geq 99 \Rightarrow \ i_{nT} \neq i_{mig}.r$

Invariants concerning procedure *migrate* (94. . . 105)

mi1: $pc = 98 \vee pc \in \{104, 105\} \Rightarrow \ index \neq \texttt{currInd}$
mi2: $pc \geq 95 \Rightarrow \ i_{mig} \neq index$
mi3: $pc = 94 \Rightarrow \ next(index) > 0$
mi4: $pc \geq 95 \Rightarrow \ i_{mig} \neq 0$
mi5: $pc \geq 95 \Rightarrow \ i_{mig} = next(index)$
mi6: $pc.r = 70$
       $\wedge \ (pc \in [95, 102) \ \wedge \ index = \texttt{currInd}$
       $\vee \ pc \in [102, 103] \vee pc \geq 110)$
       $\Rightarrow \ i_{rA}.r \neq i_{mig}$

mi7: $pc \in [95, 97] \ \wedge \ index = \text{currInd} \vee pc \geq 99$
$\Rightarrow \ i_{mig} \neq \text{next}(i_{mig})$

mi8: $(pc \in [95, 97] \vee pc \in [99, 103] \vee pc \geq 110)$
$\wedge \ index = \text{currInd}$
$\Rightarrow \ \text{next}(i_{mig}) = 0$

mi9: $(pc \in [95, 103] \vee pc \geq 110) \ \wedge \ index = \text{currInd}$
$\Rightarrow \ \text{H}(i_{mig}) \neq \text{H}(\text{currInd})$

mi10: $(pc \in [95, 103] \vee pc \geq 110) \ \wedge \ index = \text{currInd}$
$\wedge \ (pc.r \in [1, 58] \vee pc.r \geq 62 \ \wedge \ pc.r \neq 65)$
$\Rightarrow \ \text{H}(i_{mig}) \neq \text{H}(index.r)$

mi11: $pc = 101 \ \wedge \ index = \text{currInd} \vee pc = 102$
$\Rightarrow \ h_{mig} = \text{H}(i_{mig})$

mi12: $pc \geq 95 \ \wedge \ index = \text{currInd}$
$\vee \ pc \in \{102, 103\} \vee pc \geq 110$
$\Rightarrow \ \text{Heap}(\text{H}(i_{mig})) \neq \bot$

mi13: $pc = 103 \ \wedge \ index = \text{currInd} \ \wedge \ k < curSize$
$\Rightarrow \ \text{H}(index).\text{table}[k] = \textbf{done}$

mi14: $pc = 103 \ \wedge \ index = \text{currInd}$
$\wedge \ n < \text{H}(i_{mig}).\text{size}$
$\wedge \ LeastFind(\text{H}(i_{mig}), a, n)$
$\Rightarrow \ \text{X}(a) = val(\text{H}(i_{mig})[key(a, \text{H}(i_{mig}).\text{size}, n)])$

mi15: $pc = 103 \ \wedge \ index = \text{currInd}$
$\wedge \ n < \text{H}(i_{mig}).\text{size}$
$\wedge \ \text{X}(a) = val(\text{H}(i_{mig}).\text{table}$
$[key(a, \text{H}(i_{mig}).\text{size}, n)] \neq \textbf{null}$
$\Rightarrow \ LeastFind(\text{H}(i_{mig}), a, n)$

mi16: $pc = 103 \ \wedge \ index = \text{currInd}$
$\wedge \ k < \text{H}(i_{mig}).\text{size}$
$\Rightarrow \ \neg oldp(\text{H}(i_{mig}).\text{table}[k])$

mi17: $pc = 103 \ \wedge \ index = \text{currInd}$
$\wedge \ \text{X}(a) \neq \textbf{null} \ \wedge \ k < h.\text{size}$
$\wedge \ \text{X}(a) = val(h.\text{table}$
$[key(a, h.\text{size}, k)])$
$\wedge \ k \neq n < h.\text{size}$
$\Rightarrow \ ADR(h.\text{table}.[key(a, h.\text{size}, n)]) \neq a,$
where $h = \text{H}(i_{mig})$

mi18: $pc = 103 \ \wedge \ index = \text{currInd}$
$\wedge \ \text{X}(a) = \textbf{null} \ \wedge \ k < h.\text{size}$
$\Rightarrow \ val(h.\text{table}[key(a, h.\text{size}, k)]) = \textbf{null}$
$\vee \ ADR(h.\text{table}[key(a, h.\text{size}, k)]) \neq a,$
where $h = \text{H}(i_{mig})$

mi19: $pc = 103 \ \wedge \ index = \text{currInd} \ \wedge \ \text{X}(a) \neq \textbf{null}$
$\Rightarrow \ \exists m < h.\text{size} : \text{X}(a)$
$= val(h.\text{table}[key(a, h.\text{size}, m)]),$
where $h = \text{H}(i_{mig})$

mi20: $pc = 117 \ \wedge \ \text{X}(a) \neq \textbf{null}$
$\wedge \ val(\text{H}(index).\text{table}[i_{mC}]) \neq \textbf{null}$
$\vee \ pc \geq 126 \ \wedge \ \text{X}(a) \neq \textbf{null} \ \wedge \ index = \text{currInd}$
$\vee \ pc = 125 \ \wedge \ \text{X}(a) \neq \textbf{null} \ \wedge \ index = \text{currInd}$
$\wedge \ (b_{mE} \vee val(w_{mE}) \neq \textbf{null}$
$\wedge \ a_{mE} = ADR(w_{mE}))$
$\Rightarrow \exists m < h.\text{size} : \text{X}(a)$
$= val(h.\text{table}[key(a, h.\text{size}, m)]),$
where $a = ADR(\text{Y}[i_{mC}])$ and
$h = \text{H}(\text{next}(\text{currInd}))$

Invariants concerning procedure *moveContents* (110...118):

mC1: $pc = 103 \vee pc \geq 110 \Rightarrow \ to = \text{H}(i_{mig})$
mC2: $pc \geq 110 \Rightarrow \ from = \text{H}(index)$
mC3: $pc > 102 \ \wedge \ m \in toBeMoved \Rightarrow$
$m < \text{H}(index).\text{size}$

mC4: $pc = 111 \Rightarrow$
$\exists m < from.\text{size} : m \in toBeMoved$
mC5: $pc \geq 114 \ \wedge \ pc \neq 118 \Rightarrow \ v_{mC} \neq \textbf{done}$
mC6: $pc \geq 114 \Rightarrow \ i_{mC} < \text{H}(index).\text{size}$
mC7: $pc = 118 \Rightarrow \ \text{H}(index).\text{table}[i_{mC}] = \textbf{done}$
mC8: $pc \geq 110 \ \wedge \ k < \text{H}(index).\text{size}$
$\wedge \ k \notin toBeMoved$
$\Rightarrow \ \text{H}(index).\text{table}[k] = \textbf{done}$
mC9: $pc \geq 110 \ \wedge \ index = \text{currInd}$
$\wedge \ toBeMoved = \emptyset$
$\wedge \ k < \text{H}(index).\text{size}$
$\Rightarrow \ \text{H}(index).\text{table}[k] = \textbf{done}$
mC10: $pc \geq 116 \ \wedge \ val(v_{mC}) \neq \textbf{null}$
$\wedge \text{H}(index).\text{table}[i_{mC}] = \textbf{done}$
$\Rightarrow \ \text{H}(i_{mig}).\text{table}[key(a, \text{H}(i_{mig}).\text{size}, 0)]$
$\neq \textbf{null}$, where $a = ADR(v_{mC})$
mC11: $pc \geq 116 \ \wedge \ \text{H}(index).\text{table}[i_{mC}] \neq \textbf{done}$
$\Rightarrow \ val(v_{mC}) = val(\text{H}(index).\text{table}[i_{mC}])$
$\wedge \ oldp(\text{H}(index).\text{table}[i_{mC}])$
mC12: $pc \geq 116 \ \wedge \ index = \text{currInd} \ \wedge \ val(v_{mC})$
$\neq \textbf{null} \Rightarrow \ val(v_{mC}) = val(\text{Y}[i_{mC}])$

Invariants concerning procedure *moveElement* (120...126):

mE1: $pc \geq 120 \Rightarrow \ val(v_{mC}) = v_{mE}$
mE2: $pc \geq 120 \Rightarrow \ v_{mE} \neq \textbf{null}$
mE3: $pc \geq 120 \Rightarrow \ to = \text{H}(i_{mig})$
mE4: $pc \geq 121 \Rightarrow \ a_{mE} = ADR(v_{mC})$
mE5: $pc \geq 121 \Rightarrow \ m_{mE} = to.\text{size}$
mE6: $pc \in \{121, 123\} \Rightarrow \ \neg b_{mE}$
mE7: $pc = 123 \Rightarrow \ k_{mE} = key(a_{mE}, to.\text{size}, n_{mE})$
mE8: $pc \geq 123 \Rightarrow \ k_{mE} < \text{H}(i_{mig}).\text{size}$
mE9: $pc = 120$
$\wedge \ to.\text{table}[key(ADR(v_{mE}), to.\text{size}, 0)] = \textbf{null}$
$\Rightarrow \ index = \text{currInd}$
mE10: $pc \in \{121, 123\}$
$\wedge \ to.\text{table}[key(a_{mE}, to.\text{size}, n_{mE})] = \textbf{null}$
$\Rightarrow \ index = \text{currInd}$
mE11: $pc \in \{121, 123\} \ \wedge \ pc.r = 103$
$\wedge \ to.\text{table}[key(a_{mE}, to.\text{size}, n_{mE})] = \textbf{null}$
$\Rightarrow \ index.r \neq \text{currInd}$
mE12: $pc \in \{121, 123\} \ \wedge \ \text{next}(\text{currInd}) \neq 0$
$\wedge \ to = \text{H}(\text{next}(\text{currInd}))$
$\Rightarrow \ n_{mE} < \text{H}(\text{next}(\text{currInd})).\text{size}$
mE13: $pc \in \{123, 125\} \ \wedge \ w_{mE} \neq \textbf{null}$
$\Rightarrow \ ADR(w_{mE}) = ADR(to.\text{table}[k_{mE}])$
$\vee \ to.\text{table}[k_{mE}] \in \{\textbf{del}, \textbf{done}\}$
mE14: $pc \geq 123 \ \wedge \ w_{mE} \neq \textbf{null}$
$\Rightarrow \ \text{H}(i_{mig}).\text{table}[k_{mE}] \neq \textbf{null}$
mE15: $pc = 117 \ \wedge \ val(v_{mC}) \neq \textbf{null}$
$\vee \ pc \in \{121, 123\} \ \wedge \ n_{mE} > 0$
$\vee \ pc = 125$
$\Rightarrow \ h.\text{table}[key(ADR(v_{mC}), h.\text{size}, 0)] \neq \textbf{null},$
where $h = \text{H}(i_{mig})$
mE16: $pc \in \{121, 123\}$
$\vee \ (pc = 125 \ \wedge \ \neg b_{mE}$
$\wedge \ (val(w_{mE}) = \textbf{null} \vee a_{mE} \neq ADR(w_{mE})))$
$\Rightarrow \ \forall m < n_{mE} :$
$\neg Find(to.\text{table}[key(a_{mE}, to.\text{size}, m)], a_{mE})$

Invariants about the integer array prot.

pr1: $\quad\mathtt{prot}[i] \;=\; \sharp(prSet1(i)) \;+\; \sharp(prSet2(i))$
$\qquad\qquad +\; \sharp(\mathtt{currInd} = i)$
$\qquad\qquad +\; \sharp(\mathtt{next}(\mathtt{currInd}) = i)$
pr2: $\quad\mathtt{prot}[\mathtt{currInd}] > 0$
pr3: $\quad pc \in [1, 58] \vee pc \geq 62 \;\wedge\; pc \neq 65 \Rightarrow$
$\qquad\mathtt{prot}[index] > 0$
pr4: $\quad\mathtt{next}(\mathtt{currInd}) \neq 0 \Rightarrow$
$\qquad\mathtt{prot}[\mathtt{next}(\mathtt{currInd})] > 0$
pr5: $\quad\mathtt{prot}[i] = 0 \Rightarrow\; \mathtt{Heap}(\mathtt{H}[i]) = \bot$
pr6: $\quad\mathtt{prot}[i] \leq \sharp(prSet3(i)) \;\wedge\; \mathtt{busy}[i] = 0 \Rightarrow$
$\qquad\mathtt{Heap}(\mathtt{H}[i]) = \bot$
pr7: $\quad pc \in [67, 72] \Rightarrow\; \mathtt{prot}[i_{rA}] > 0$
pr8: $\quad pc \in [81, 84] \Rightarrow\; \mathtt{prot}[i_{nT}] > 0$
pr9: $\quad pc \geq 97 \Rightarrow\; \mathtt{prot}[i_{mig}] > 0$
pr10: $\quad pc \in [81, 82] \Rightarrow\; \mathtt{prot}[i_{nT}] = \sharp(prSet4(i_{nT})) + 1$

Invariants about the integer array `busy`.

bu1: $\quad\mathtt{busy}[i] \;=\; \sharp(buSet1(i)) \;+\; \sharp(buSet2(i))$
$\qquad\qquad +\; \sharp(\mathtt{currInd} = i)$
$\qquad\qquad +\; \sharp(\mathtt{next}(\mathtt{currInd}) = i)$
bu2: $\quad\mathtt{busy}[\mathtt{currInd}] > 0$
bu3: $\quad pc \in [1, 58]$
$\qquad\vee\, pc > 65 \;\wedge\; \neg(i_{rA} = index \;\wedge\; pc \in [67, 72])$
$\qquad\Rightarrow\; \mathtt{busy}[index] > 0$
bu4: $\quad\mathtt{next}(\mathtt{currInd}) \neq 0 \Rightarrow$
$\qquad\mathtt{busy}[\mathtt{next}(\mathtt{currInd})] > 0$
bu5: $\quad pc = 81 \Rightarrow\; \mathtt{busy}[i_{nT}] = 0$
bu6: $\quad pc \geq 100 \Rightarrow\; \mathtt{busy}[i_{mig}] > 0$

Some other invariants we have postulated:

Ot1: $\quad\mathtt{X}(\mathbf{0}) = \mathbf{null}$
Ot2: $\quad\mathtt{X}(a) \neq \mathbf{null} \Rightarrow\; ADR(\mathtt{X}(a)) = a$

The motivation of invariant (Ot1) is that we never store a value for the address 0. The motivation of invariant (Ot2) is that the address in the hash table is unique.

Ot3: $\quad return_{gA} = \{1, 10, 20, 30, 36, 46, 51\}$
$\qquad\qquad\wedge\; return_{rA} = \{0, 59, 77, 90\}$
$\quad\wedge\, return_{ref} = \{10, 20, 30, 36, 46, 51\}$
$\qquad\qquad\wedge\; return_{nT} = \{30, 46\}$

Ot4: $\quad pc \in \{0,\, 1,\, 5,\, 6,\, 7,\, 8,\, 10,\, 11,\, 13,\, 14,\, 15,\, 16,$
$\qquad\qquad 17,\, 18,\, 20,\, 21,\, 25,\, 26,\, 27,\, 28,\, 30,\, 31,\, 32,$
$\qquad\qquad 33,\, 35,\, 36,\, 37,\, 41,\, 42,\, 43,\, 44,\, 46,\, 47,\, 48$
$\qquad\qquad 49,\, 50,\, 51,\, 52,\, 57,\, 59,\, 60,\, 61,\, 62,\, 63,\, 65,$
$\qquad\qquad 67,\, 68, 69,\, 70, 71,\, 72,\, 77,\, 78,\, 81,\, 82,\, 83,$
$\qquad\qquad 84,\, 90,\, 94,\, 95,\, 97,\, 98,\, 99,\, 100,\, 101,\, 102,$
$\qquad\qquad 103,\, 104,\, 105,\, 110,\, 111,\, 114,\, 116,\, 117,$
$\qquad\qquad 118,\, 120,\, 121,\, 123,\, 125,\, 126\}$

Appendix B: Dependencies between invariants

Let us write "φ **from** $\psi_1, \cdots, \psi_n$" to denote that φ is proved to be an invariant using that $\psi_1, \ldots, \psi_n$ hold. We write "$\varphi \;\Leftarrow\; \psi_1, \cdots, \psi_n$" to denote that predicate φ is implied by the conjunction of $\psi_1, \ldots, \psi_n$. We have verified the following "**from**" and "$\Leftarrow$" relations mechanically:

Co1 **from** fi10, Ot3, fi1
Co2 **from** de5, Ot3, de6, del, de11
Co3 **from** in5, Ot3, in6, in1, in11
Cn1 **from** Cn6, Ot3
Cn2 **from** Cn8, Ot3, del
Cn3 **from** Cn10, Ot3, in1, in5
Cn4 **from** Cn11, Ot3
No1 $\Leftarrow$ No2
No2 **from** nT1, He2, rA2, Ot3, Ha2, Ha1, rA1, rA14, rA3, nT14, rA4
He1 **from** Ha1
He2 **from** Ha3, rA5, Ha1, He1, rA2
He3, He4 **from** Ot3, rA6, rA7, mi12, rA11, rA5
He5 **from** He1
He6 **from** rA8, Ha3, mi8, nT2, rA5
Ha1 **from** true
Ha2 **from** Ha1
Ha3 **from** Ha2, Ha1, He2, He1
Ha4 $\Leftarrow$ Ha3, He3, He4
Cn5 **from** Cn6, Ot3
Cn6 **from** Cn5, Ot3
Cn7 **from** Cn8, Ot3, del
Cn8 **from** Cn7, Ot3
Cn9 **from** Cn10, Ot3, in1, in5
Cn10 **from** Cn9, Ot3, in5
Cn11 **from** Cn11, Ot3
Cu1 **from** Ot3, Ha4, rA6, rA7, nT13, nT12, Ha2, He3, He4, rA11, nT9, nT10, mi13, rA5
Cu2 $\Leftarrow$ Cu6, cu7, Cu3, He3, He4
Cu3 **from** rA6, rA7, nT13, nT12, mi5, mi4, Ne8, rA5
Cu4 **from** del, in1, as1, rA6, rA7, Ha2, nT13, nT12, Ne9, Cu9, Cu10, de7, in7, as5, He3, He4, mi5, mi4, Ot3, Ha4, de3, mi9, mi10, de5, rA5
Cu6 **from** Ot3, rA6, rA7, Ha2, nT13, nT12, Ha3, in3, as3, Ne23, mi5, mE6, mE7, mE10, mE3, Ne3, mi1, mi4, rA5
Cu7 **from** Ot3, rA6, rA7, Ha2, nT13, nT12, Ha3, in3, as3, in5, mi5, mE6, mE7, mE10, mE3, Ne3, mi4, de7, in7, as5, Ne22, mi9, mi10, rA5, He3, mi12, mi1, Cu9, de1 in1, as1
Cu8 **from** Cu8, Ha2, nT9, nT10, rA6, rA7, mi5, mi4, mC2, mC5, He3, He4, Cu1, Ha4, mC6, mi16, rA5
Cu9, Cu10 **from** rA6, rA7, nT13, nT12, Ha2, He3, He4, Cu1, Ha4, de3, in3, as3, mE3, mi9, mi10, mE10, mE7, rA5
Cu11, Cu12 **from** Cu9, Cu10, Cu13, Cu14, del, in1, as1, rA6, rA7, Ha2, nT13, nT12, He3, He4, Cu1, Ha4, in3, as3, mi14, mi15, de3, in10, as8, mi12, Ot2, fi5, de8, in8, as6, Cu15, de11, in11, rA5
Cu13, Cu14 **from** He3, He4, Ot2, del, in1, as1, Ot1, rA6, rA7, nT13, nT12, Ha2, Cu9, Cu10, Cu1, Ha4, de3, in3, as3, Cu11, Cu12, in10, as8, fi5, de8, in8, as6, Cu15, mi17, mi18, mi12, mi4, de11, rA5
Cu15 **from** He3, He4, rA6, rA7, nT13, nT12, Ha2, Cu1, Ha4, del, in1, as1, de3, in3, as3, fi5, de8, in8, as6, mi12, mi19, mi4, Ot2, Cu9, Cu10, Cu11, Cu12, Cu13, Cu14, rA5
Cu16 $\Leftarrow$ Cu13, Cu14, Cu15, He3, He4, Ot1
Ne1 **from** nT9, nT10, mi7
Ne2 **from** Ne5, nT3, mi8, nT9, nT10
Ne3 **from** Ne1, nT9, nT10, mi8
Ne4 **from** Ne1, nT9, nT10
Ne5 **from** Ot3, nT9, nT10, mi5
Ne6 $\Leftarrow$ Ne10, Ne24, He6, He3, He4, Cu4
Ne7 **from** Ha3, rA6, rA7, rA8, nT13, nT12, nT11, He3,

He4, mi8, nT7, Ne5, Ha2, He6, rA5
Ne8 **from** Ha3, rA8, nT11, mi8, nT6, Ne5, rA5
Ne9 **from** Ha3, Ha2, Ne3, Ne5, de3, as3, rA8, rA6, rA7,
    nT8, nT11, mC2, nT4, mi8, rA5
Ne9a **from** Ha3, Ne3, rA5, de3, rA8, nT4, mi8
Ne10 **from** Ha3, Ha2, de3, rA8, nT11, Ne3, He6, mi8,
    nT8, mC2, nT2, Ne5, rA5
Ne11 **from** Ha3, Ha2, He6, nT2, nT8, rA8, nT11, mi8,
    Ne3, mC2, rA5
Ne12, Ne13 **from** Ha3, Ha2, Cu8, He6, He3, He4, Cu1, de3,
    in3, as3, rA8, rA6, rA7, nT11, nT13, nT12,
    mi12, mi16, mi5, mi4, de7, in7, as5, Ot2, del,in1,
    as1, Cu9, Cu10, Cu13, Cu14, Cu15, as9, fi5, de8, in8,
    as6, mC2, Ne3, Ot1, Ne14, Ne20, mE16, mE7, mE4,
    mE1, mE12, mE2, Ne15, Ne16, Ne17,
    Ne18, mi20, de11, in11, rA5
Ne14 **from** Ha3, Ha2, He6, He3, He4, nT2, nT8, de3, in3,
    as3, rA8, nT11, Ot2, del, in1, as1, Cu9, Cu10,
    mi8, Ne3, mC2, mE7, mE16, mE1, mE4,
    mE12, Ne17, Ne18, Cu1, rA5
Ne15, Ne16 **from** Ha3, Ha2, Cu8, He6, He3, He4, Cu1, de3,
    in3, as3, rA8, rA6, rA7, nT11, nT13, nT12, mi12, mi5,
    mi16, mi4, de7, in7, as5, Ot2, del, in1, as1, Cu9, Cu10,
    Cu13, Cu14, Cu15, as9, fi5, de8, in8, as6, mC2, Ne3, Ot1,
    Ne19, Ne20, Ne12, Ne13, mE16, mE7, mE4, mE1,
    mE12, mE10, mE2, in11, de11, rA5
Ne17, Ne18 **from** Ha3, Ha2, mi8, He6, He3, He4, Cu1, nT2,
    de3, in3, as3, rA8, rA6, rA7, nT11, nT13, nT12, de7,
    in7, as5, Ot2, del, in1, as1, Cu9, Cu10, nT8, mE2, fi5,
    de8, in8, as6, mC2, Ne3, mC11, mC6, mC12, mE7,
    mE10, mE1, Cu8, Cu15, Cu13, Cu14, Cu11, Cu12, as8,
    de11, rA5
Ne19 **from** Ha3, Ha2, He6, nT2, nT8, de3, in3, as3, rA8,
    nT11, mi8, Ne3, mE7, Ne14, mE16,
    Ot1, mE1, mE4, mE12, Ne17, Ne18, rA5
Ne20 **from** Ha3, Ha2, Cu8, He6, He3, He4, Cu1, Ha4, de3,
    in3, as3, rA8, rA6, rA7, nT11, nT13, nT12, mi12,
    mi16, mi5, mi4, Ne1, de7, in7, as5, del, in1, as1,
    Cu9, Cu10, Cu13, Cu14, Cu15, as9, fi5, de8, in8, as6,
    mC2, Ne3, Ot1, mi20, in11, rA5
Ne22 **from** Ot3, rA8, Ha2, nT11, Ha3, de3, in3, as3, mi5,
    mi4, Ne3, nT18, mE3, mi8, mE10, mE7, mE6,
    Ne5, nT5, nT2, rA5, nT8, nT12, mC2, mE2
Ne23 $\Leftarrow$ Cu6, cu7, Ne6, Ne7, He3, He4, Ne22, He6
Ne24 $\Leftarrow$ Ne27, He6
Ne25 $\Leftarrow$ Ne19, Ne17, Ne18, He6
Ne26 $\Leftarrow$ Ne17, Ne18, He6
Ne27 $\Leftarrow$ Cu16, Ne25, Ne26, Ne17, Ne18, He6
fi1, del, in1, as1 **from**
fi2 **from** fi2, Ot3
fi3 **from** fi4, Ot3, rA6, rA7, Ha2, rA5
fi4 **from** Ot3, rA6, rA7, nT13, nT12
fi5, de8, in8, as6 $\Leftarrow$ Cu2, de10, in10, as8, fi8, He3, He4
fi6 **from** Ot3, fi1, del, in1, as1, rA6, rA7, Ha2, nT13, nT12,
    mi9, mi10, Cu9, Cu10, He3, He4, Cu1,
    Ha4, fi4, in3, as3, rA5
fi7 **from** fi8, fi6, fi2, Ot3, fi1, del, in1, as1, rA6, rA7, Ha2,
    nT13, nT12, mi9, mi10, Cu9, Cu10, He3, He4,
    Cu1, Ha4, fi4, in3, as3, rA5
fi8 **from** fi4, fi7, fi2, Ot3, fi1, del, in1, as1, rA6, rA7, Ha2,
    nT13, nT12, mi9, mi10, Cu9, Cu10,

He3, He4, Cu1, Ha4, in3, as3, rA5
fi9 $\Leftarrow$ Cu1, Ha4, Cu9, Cu10, Cu11, Cu12, fi8, fi3, fi4, fi5,
    de8, in8, as6, He3, He4
fi10 **from** fi9, Ot3
fi11, de12, in12, as10 **from** Ot3, nT9, nT10, mi9, mi10, Cu8,
    fi4, de3, in3, as3, fi3, de2, in2, as2
de2 **from** de3, Ot3, rA6, rA7, Ha2, rA5
de3 **from** Ot3, rA6, rA7, nT13, nT12
de4, in4, as4 **from** Ot3
de5 **from** Ot3
de6 **from** Ot3, de1, de11
de7, in7, as5 $\Leftarrow$ de3, in3, as3, Cu1, Ha4, de13, in13, as11
de9 **from** Ot3, del, in1, as1, rA6, rA7, Ha2, nT13, nT12,
    mi9, mi10, Cu9, Cu10, de3, de7, in7, as5, rA5
de10 **from** de3, de9, Ot3, del, in1, as1, rA6, rA7, Ha2, nT13,
    nT12, mi9, mi10, Cu9, Cu10, de7, in7, as5, He3,
    He4, rA5
de11 $\Leftarrow$ de10, de2, de3, He3, He4, Cu1, Ha4, Cu9, Cu10,
    Cu11, Cu12, fi5, de8, in8, as6
de13, in13, as11 $\Leftarrow$ Ax2, de2, de3, de4, in2, in3, in4, as2,
    as3, as4
in2 **from** in3, Ot3, rA6, rA7, Ha2, rA5
in3 **from** Ot3, rA6, rA7, nT13, nT12
in5 **from** Ot3
in6 **from** Ot3, in1, in11
in9 **from** Ot3, del, in1, as1, rA6, rA7, Ha2, nT13, nT12, mi9,
    mi10, Cu9, Cu10, He3, He4, in3, de7, in7, as5, rA5
in10 **from** in9, fi2, Ot3, del, in1, as1, rA6, rA7, Ha2, nT13,
    nT12, mi9, mi10, Cu9, Cu10, He3, He4, in3, de7, in7,
    as5, rA5
in11 $\Leftarrow$ in10, in2, in3, Cu1, Ha4, Cu9, Cu10, Cu11, Cu12,
    fi5, de8, in8, as6
as2 **from** as3, He3, He4, Ot3, rA6, rA7, Ha2, rA5
as3 **from** Ot3, rA6, rA7, nT13, nT12
as7 **from** Ot3, del, in1, as1, rA6, rA7, Ha2, nT13, nT12, mi9,
    mi10, Cu9, Cu10, as3, de7, in7, as5, rA5
as8 **from** as7, Ot3, del, in1, as1, rA6, rA7, Ha2, nT13, nT12,
    mi9, mi10, Cu9, Cu10, He3, He4, as3, de7, in7,
    as5, rA5
as9 $\Leftarrow$ as8, as2, as3, He3, He4, Cu1, Ha4, Cu9, Cu10, Cu11,
    Cu12, fi5, de8, in8, as6
rA1 **from** Ha2
rA2 **from** Ot3
rA3 **from** Ot3, rA9, He2, He1, rA2, rA13
rA4 **from** Ot3, nT14
rA5 **from** Ot3, rA1, rA2, Ha3, He2
rA6, rA7 **from** Ot3, nT13, nT12, nT14, rA11, mi4, bu2, bu3,
    Ha3, mi6, Ha2, He3, He4, He2, rA2
rA8 **from** Ot3, bu4, nT14, mi6, Ne2, mi5
rA9 **from** Ot3, Ha2, nT14, He1, He2
rA10 **from** Ot3
rA11 **from** Ot3, nT13, nT12, mi2
rA12 **from** Ot3, nT9, nT10
rA13 **from** Ot3, rA5
rA14 **from** Ot3, rA4, He1, rA2
nT1 **from** Ot3, pr5, Ha3, nT14, nT16, Ha2
nT2 **from** Ot3, nT14, Ha3, rA5
nT3 **from** Ot3, nT9, nT10
nT4 **from** Ot3, Ha3, de3, nT13, nT12, nT15, rA5
nT5 **from** Ot3, Ha3, in3, as3, nT13, nT12, nT15, nT18,
    mE3, mi4, rA5

nT6 **from** Ot3, nT13, nT12, nT14, Ha3, rA5
nT7 **from** Ot3, nT13, nT12, nT15, rA6, rA7, Ha2, mi9,
    mi10, nT14, Ha3, nT16, rA5
nT8 **from** Ot3, de3, in3, as3, nT13, nT12, nT15, nT18,
    mE3, mi4, Ha3, mC2, nT16, nT2, Ha2, rA5
nT9, nT10 **from** Ot3, pr2, pr3, nT18
nT11 **from** Ot3, pr4, nT16, mi8
nT13, nT12 $\Leftarrow$ nT9, nT10, Ha3, He3, He4
nT14 **from** Ot3, nT9, nT10, nT18, nT16, pr7
nT15 $\Leftarrow$ nT14, Ha3, nT2
nT16 **from** Ot3, pr8
nT17 **from** Ot3, mi5, pr4, nT11, mi10
nT18 **from** Ot3, pr9, mi5, nT11
mi1 **from** Ot3, mi9, mi10, mi10
mi2 **from** Ot3, Ne4
mi3 **from** Ot3, fi11, de12, in12, as10, nT9, nT10, Ne5
mi4 **from** Ot3, mi9, mi10, mi3
mi5 **from** Ot3, nT9, nT10, Ne5, mi10, mi4
mi6 **from** Ot3, mi5, bu6, rA8, mi9, mi10, bu4, mi4
mi7 **from** Ot3, mi2, mi7, mi4, nT18, Ne2, mi10, nT17, mi3
mi8 **from** Ot3, mi10, Ne2, mi3
mi9, mi10 **from** Ot3, He3, He4, nT9, nT10, nT18, Ne3, Ha3,
    mi3, nT17, mi10, He2, mi4, mi12, mi6, He6
mi11 **from** Ot3, nT18, mi9, mi6, mi6
mi12 **from** Ot3, rA8, nT2, He6, mi9, mi5, mi3, Ha3, mi4, rA5
mi12 **from** Ot3, mi12, nT18, mi6, Ha3, mi4, rA5
mi13 **from** Ot3, rA6, rA7, Ha2, nT13, nT12, He3, He4, mi9,
    mi10, mC9, rA5
mi14, mi15 $\Leftarrow$ Ne12, Ne13, mi5, Cu15, mi13, Ot2, He3,
    He4, Ne17, Ne18, Cu8, He6, He5, mi4, Ot1
mi16 $\Leftarrow$ Ne11, mi5, mi4
mi17, mi18 $\Leftarrow$ Ne15, Ne16, mi5, Cu15, mi13, Ot2, He3,
    He4, Ne17, Ne18, Cu8, He6, He5, mi4
mi19 $\Leftarrow$ Ne20, mi5, Cu15, mi13, Ot2, He3, He4
mi20 **from** Ha3, Ha2, Cu8, He6, He3, He4, Cu1, Ha4, de3,
    in3, as3, rA8, rA6, rA7, nT11, nT13, nT12, mi5, mi4,
    de7, in7, as5, Ot2, del, in1, as1, Cu9, Cu10, Cu13,
    Cu14, Cu15, as9, fi5, de8, in8, as6, mC6, Ne3, Ot3,
    mC11, mi13, mi9, mi10, mC2, mE3, mE10, mE7,
    mC12, mE1, mE13, Ne17, Ne18, mE2, mE4, Ot1, mE6,
    Ne10, in11, rA5
mC1 **from** Ot3, mi6, mi11, nT18
mC2 **from** Ot3, rA6, rA7, nT13, nT12, mC2
mC3 **from** Ot3, mC3, nT13, nT12, rA6, rA7, Ha2, rA5
mC4 **from** Ot3, mC4, mC2, mC3, He3, He4, rA6, rA7, Ha2,
    rA5
mC5 **from** Ot3
mC6 **from** Ot3, rA6, rA7, Ha2, nT13, nT12, mC2, rA5
mC7 **from** Ot3, rA6, rA7, Ha2, nT13, nT12, mC2, rA5
mC8 **from** Ot3, rA6, rA7, Ha2, nT13, nT12, He3, He4, mC7,
    rA5
mC9 **from** Ot3, rA6, rA7, Ha2, nT13, nT12, He3, He4, mi9,
    mi10, He5, mC7, mC8, rA5
mC10 **from** Ot3, rA6, rA7, Ha2, nT13, nT12, mC2, del, in1,
    as1, mi6, Ha3, mi4, nT18, mE15, mC11, mi5, rA5
mC11 **from** Ot3, rA6, rA7, Ha2, nT13, nT12, mC2, rA5
mC12 **from** Ot3, rA6, rA7, mC2, mC11, Cu9, Cu10, de7,
    in7, as5, mi9, mC6
mE1 **from** Ot3
mE2 **from** Ot3
mE3 **from** mC1, Ot3, mi6, nT18

mE4 **from** Ot3, mE1
mE5 **from** Ot3, mE3, Ha3, mi6, mi4, nT18, Ha2, rA5
mE6 **from** Ot3
mE7 **from** Ot3, Ha2, Ha3, mi6, mi4, mE3, rA5
mE8 **from** Ot3, Ha3, mi6, mi4, nT18, Ha2, mE3, rA5
mE9 **from** Cu1, Ha4, Ot3, Ha2, Ha3, mi6, mi4, mE3, mC2,
    mC10, mE1, mC1, del, in1, as1, mi13, mi12, mC6,
    mE2, rA5
mE10 **from** del, in1, as1, mE3, mi6, Ot3, Ha2, Ha3, mi4,
    mE11, mE9, mE7, rA5
mE11 $\Leftarrow$ mE10, mi13, mE16, mi16, mi5, mE3, Ne12, Ne13,
    mC12, mE2, mE1, mE4, mC6, mE12, mi12, Cu13,
    Cu14, He3, He4, mi4
mE12 $\Leftarrow$ Ne23, Ne22, mE16, He6, Ne8
mE13 **from** Ot3, Ha2, mE14, del, in1, as1, Ha3, mi6, mi4,
    mE3, rA5
mE14 **from** Ot3, Ha2, del, in1, as1, Ha3, mi6, mi4, nT18,
    mE3, mE2, rA5
mE15 **from** Ot3, mE1, Ha2, del, in1, as1, Ha3, mi6, mi4,
    nT18, mE3, mE2,
    mE7, mE14, mE4, rA5
mE16 **from** Ha3, Ha2, mE3, del, in1, as1, mi6, mE2, mE4,
    mE1, mE7, mi4,
    Ot3, mE14, mE13, rA5
pr1 **from** Ot3, rA11, rA10, nT9, nT10, Ne5, mi2, mi4, mi8,
    mi5
pr2, pr3 **from** pr1, Ot3, rA11, mi1
pr4 $\Leftarrow$ pr1
pr5 $\Leftarrow$ pr6, pr1, bu1
pr6 **from** Ot3, Ha2, nT9, nT10, nT14, nT16, He2, rA2, pr1,
    bu1, pr10, rA9, He1, rA4
pr7, pr8, pr9 $\Leftarrow$ pr1, mi4
pr10 **from** Ot3, pr1, nT9, nT10, nT14, nT17
bu1 **from** Ot3, rA11, rA10, nT9, nT10, Ne5, mi2, mi8,
    mi5, bu5
bu2, bu3 $\Leftarrow$ bu1, Ot3, rA10
bu4 $\Leftarrow$ bu1
bu5 **from** Ot3, nT9, nT10, nT16, nT18, pr1, bu1
bu6 $\Leftarrow$ bu1, mi4
Ot1 **from** del, in1, as1
Ot2 **from** del, in1, as1
Ot3 **from** true
Ot4 **from** Ot3

References

1. Anderson JH, Ramamurthy S, Jeffay K: Real-time comput-
ing with lock-free shared objects. ACM Trans Comput Syst
15(2):134–165 (1997)
2. Attiya H, Bar-Noy A, Dolev D, Peleg D, Reischuk R: Renaming
in an asynchronous environment. J ACM 37:524–548 (1990)
3. Bar-Noy A, Dolev D: Shared-memory vs. message-passing in
an asynchronous distributed environment. In: Proc. 8th ACM
Symp. on principles of distributed computing, 1989, pp 307–
318
4. Barnes G: A method for implementing lock-free shared-data
structures. In: Proceedings of the fifth annual ACM sympo-
sium on Parallel algorithms and architectures. ACM Press, 1993,
pp 261–270

5. Cassez F, Jard C, Rozoy B, Dermot M: Modeling and verification of parallel processes (2000) 4th Summer School, MOVEP 2000, Nantes, France

6. Gao H, Groote JF, Hesselink WH: PVS verification. `http://www.cs.rug.nl/~wim/mechver hashtable` (2003)

7. Groote JF, Hesselink WH, Mauw S, Vermeulen R: An algorithm for the asynchronous write-all problem based on process collision. Distr Comput 14:75–81 (2001)

8. Harbison SP: Modula-3. Prentice Hall, New Yersey 1992

9. Herlihy M: Wait–free synchronization. ACM Trans Program Lang Syst 13:124–149 (1991)

10. Herlihy M: A methodology for implementing highly concurrent data objects. ACM Trans Program Lang Syst 15:745–770 (1993)

11. Herlihy M, Wing J: Linearizability: a correctness condition for concurrent objects. ACM Trans Program Lang Syst 12:463–492 (1990)

12. Herlihy MP, Moss JEB: Lock-free garbage collection for multiprocessors. IEEE Transactions on Parallel and Distributed Systems 3:304–311 (1992)

13. Hesselink WH: Wait–free linearization with a mechanical proof. Distr Comput 9:21–36 (1995)

14. Hesselink WH: Bounded delay for a free address. Acta Inf 33:233–254 (1996)

15. Hesselink WH, Groote JF: Wait-free concurrent memory management by Create, and Read until Deletion (CaRuD). Distr Comput 14:31–39 (2001)

16. Kanellakis PC, Shvartsman AA: Fault-tolerant parallel computation. Kluwer Academic Publishers, Dordrecht 1997

17. Knuth DE: The Art of Computer Programming. Part 3, Sorting and searching. Addison-Wesley 1973

18. LaMarca A: A performance evaluation of lock-free synchronization protocols. In: Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing. ACM Press 1994, pp 130–140

19. Lamport L: The temporal logic of actions. ACM Trans Program Lang Syst 16:872–923 (1994)

20. Lynch NA: Distributed Algorithms. Morgan Kaufman, San Francisco 1996

21. Manna Z, Pnueli A: The Temporal Logic of Reactive and Concurrent Systems: Specification. Springer, New York 1992

22. Michael MM: High performance dynamic lock-free hash tables and list-based sets. In: Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures. ACM Press 2002, pp 73–82

23. Owre S, Shankar N, Rushby JM, Stringer-Calvert DWJ: PVS Version 2.4, System Guide, Prover Guide, PVS Language Reference (2001) `http://pvs.csl.sri.com`

24. Rajwar R, Goodman JR: Transactional lock-free execution of lock-based programs. In: Tenth international conference on architectural support for programming languages and operating systems on Proceedings of the 10th international conference on architectural support for programming languages and operating systems (ASPLOS-X). ACM Press 2002, pp 5–17

25. Shalev O, Shavit N: Split-ordered lists: lock-free extensible hash tables. In: Proceedings of the twenty-second annual symposium on Principles of distributed computing. ACM Press 2003, pp 102–111

26. Sundell H, Tsigas P: Scalable and lock-free concurrent dictionaries. In: Proceedings of the 2004 ACM symposium on Applied computing. ACM Press 2004, pp 1438–1445

27. Valois JD: Implementing lock-free queues. In: Proceedings of the seventh international conference on Parallel and Distributed Computing Systems, 1994, pp 64–69

28. Valois JD: Lock-free linked lists using compare-and-swap. In: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing. ACM Press 1995, pp 214–222. See also Valois JD: ERRATA. Lock-free linked lists using compare-and-swap. Unpublished manuscript (1995)

29. Wirth N: Algorithms + Data Structures = Programs. Prentice Hall 1976