Formal Correctness of Result Checking for Priority Queues

by

Ruzica Piskac

A thesis submitted in conformity with the requirements
for the degree of Master of Science
Department of Computer Science
Universität des Saarlandes

Supervisors: Prof. Dr. Harald Ganzinger†
Prof. Dr. Andreas Podelski
Dr. Hans de Nivelle

February 3, 2005

dedicated to the memory of Harald Ganzinger

Erklärung

Hiermit erkläre ich, Ruzica Piskac, an Eides statt, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen verwendet habe.

Saarbrücken, den 31.01.2005.

Acknowledgements

First and foremost, I would like to thank my first supervisor Harald Ganzinger for giving me this interesting thesis subject, for the inspiring discussions, for his support and his patience. I would like to thank him for being there until the end. I am also grateful to Uwe Waldmann, who spent lots of time giving me hints and suggestions for improvements. Then I am indebted to Andreas Podelski for being willing to act as a supervisor. I would like to thank him for his time, for his guidance and his valuable comments. I am also very grateful to my second supervisor Hans de Nivelle, for giving helpful comments, for his useful suggestions, for valuable debates and proofreading several draft versions of this thesis. Without him this thesis would not be as it is. Special thanks go to the members of AG2 at MPII for providing such a nice and productive working atmosphere. Finally, I would also like to thank my family and my friends for their constant support and encouragement that only made this thesis possible.

Abstract

We formally prove the correctness of the time super-efficient result checker for priority queues, which is implemented in LEDA [17]. A priority queue is a data structure that supports insertion, deletion and retrieval of the minimal element, relative to some order.

A result checker for priority queues is a data structure that monitors the input and output of the priority queue. Whenever the user requests a minimal element, it checks that the returned element is indeed minimal. In order to do this, the checker makes use of a *system of lower bounds*.

We have verified that, for every execution sequence in which the checker accepts the outputs, the priority queue returned the correct minimal elements. For the formal verification, we used the first-order theorem prover Saturate [25].

Contents

Chapter 1

Introduction

Everyone who has ever programmed knows that the reliability of software is a delicate issue.

One of the approaches for obtaining reliability is *formal verification*. Verifying a program means giving the formal mathematical proof of its correctness. Verification is theoretically the best approach, because it guarantees that the program will behave correct on every given input. However, in practice, verification is difficult. Verifying a program is much harder than writing a program, and it requires more skills. Small changes in the program may require a completely new correctness proof.

Another approach is *program result checking*, which was proposed by Blum [6, 7]. In program result checking, the result checking is automatized. The result checker program is run in conjunction with the original program. The checker confirms correctness of the program's output or reports an error. It does not verify whether or not the program is correct, in fact it does not look into the program code, the checker verifies that the output was correct on a given input.

There are many cases where checking a result is much easier than generating a result. The research on program result checking is very comprehensive; the papers [24][21][6][22][8] give the mathematical theory behind it and several new techniques are introduced to obtain the software reliability. Sullivan, Masson and Bright [22, 8] proposed so-called *certification-trails*. Briefly, the certification-trail method consists of two phases: in the first phase the program runs on the given input and produces an output and a trail of data. This data is chosen in such a way that in the second phase another algorithm can easily determine whether an error has occurred. Using this technique a certifier for sorting algorithm is formally proved [11].

A very important role plays checking of data structures: in [1] checking of linked data structures is described. A certification-trail method was also used for data structures [23, 10], particularly for mergable priority queues [9].

Mehlhorn et al. used the ideas of the certification-trail method and in [17,

15] a time super-efficient checker for priority queues was introduced. By a time super-efficient checker, we mean that the running times of the checker are asymptotically less than the running times of the priority queue itself. When the priority queue is incorrect and returns a non-minimal element, this will not be noticed immediately, but only at the moment when one of the smaller elements is returned. Such a checker is called an *off-line* checker.

In this thesis, we give a formal proof of the correctness of that checker. By the correctness of the checker we assume that if during the run of the priority queue the checker did not report any error, then all returned minimal elements were correct. All proofs have been formally verified using the Saturate system [25].

The Saturate system [19, 3] is a first order theorem prover based on saturation up to redundancy. The theoretical basis of the Saturate system includes the superposition calculus [4, 2] and the chaining calculus [5]. The search space is restricted by applying the rules for detecting redundancy of the clauses. Since the chaining calculus treats transitive relations efficiently, Saturate seemed to be the best prover for dealing with the total orders which are used for the priorities.

The thesis is organized as follows:

Chapter 2 introduces the basic terminology of program checking and explains the difference between checking the programs and checking the data structures;

Chapter 3 defines the abstract data type priority queues and describes the algorithm and the data structures used as a priority queue checker;

Chapter 4 gives a mathematical model that we have developed in order to establish the links between the LEDA code, priority queues and checked priority queues. In this chapter the correctness of the checker is defined and explained;

Chapter 5 contains the formal specification of priority queues and their checker in terms of first order logic. The specification is written in the simplified Saturate syntax. The theorem which validates the correctness of the checker for priority queues is formalized;

Chapter 6 describes the formal proof of the correctness theorem. The proof is done using the Saturate system and in this chapter we explain in detail all lemmas and subtheorems we had to prove in order to validate the correctness theorem. The Saturate files are not included in the thesis, but they are available

Chapter 7 gives a concluding overview and future work.

Chapter 2

Certifying programs and data structures

In this chapter we introduce the basic concepts of program checking. We define a program correctness checker and explain the difference between checking the program and checking the data structure. There are also some examples which illustrate the introduced terminology.

2.1 Certifying programs

Let f be some problem. Following [16], it is important to distinguish an *algorithm* and a *program* for f. An algorithm is defined as a description of a method for solving the problem intended for a human reader. A program is defined as a description intended for machine execution. While algorithms are described and formalized in natural language and their correctness is proved, programs are written in computer language and executed on machines. Since programs are written by human programmers, it is very likely that mistakes could occur in the program.

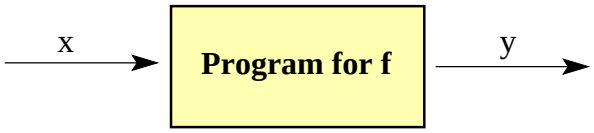

Figure 2.1: The scheme illustrates the typical behaviour of program f. The user can see input x and output y. Since he cannot be certain about the correctness of f, he would like to be able to check whether indeed $y = f(x)$

A formal verification is one way of proving the correctness of the program. In formal verification the states and the behaviour of the program are formalized in the mathematical notation and its correctness is formally proved. Such proofs assure reliability and they guarantee the correct behaviour of the program on all possible inputs. Unfortunately, in practice such proofs were constructed only for simple programs and even then it was unexpectedly difficult.

Checking is considered to be a simpler task than verification since in the checking we do not have to confirm that the given program returns the correct output on every single input, but we have to confirm that the program has returned the correct output for a given input. In [6], which is one of the first papers about checking program correctness, a program correctness checker was defined as *an algorithm for checking the output of computation.*

Often it is easier to check whether $y = f(x)$ than to calculate y. For example, if our goal is to find the roots of an equation $f(x) = 0$, then it is easier to check whether the returned value y is the root than to calculate it.

In order to make the checking easier, we require that the program should not return only the output y, but also a correctness proof (so-called *witness*). The main requirement of the introduced witness w is that it should make it easy to check whether $y = f(x)$. "Easy" means that the resource requirements of checking should be lower than the requirements of the program itself. More about that topic can be read in Section 2.2. "Easy" also means that the checker should be simple enough so that we do not need to prove its correctness, otherwise we could end up in an infinite loop of checking: first we check the correctness of the program, then the correctness of the checker, then checker's checker, ...

Although, following [18], the algorithms used for checkers should be easy and understandable such that their proof is not required, there is a footnote in that paper which states that *formal verification of simple checkers might be a realistic goal in the near future.* In this thesis we will try to give this anticipated formal proof that the priority queue checker implemented in LEDA is correct.

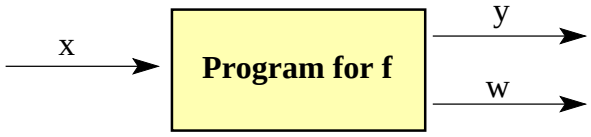

Figure 2.2: A witness w is an auxiliary output which makes checking easier

The other important requirements of the witness w is that if $y \neq f(x)$, there should not exist w such that the triple (x, y, w) passes checking.

2.1.1 A witness example

In this section we give an example of a witness which is certifying the output of a computation. The example comes from the graph theory: it is the problem of finding a maximum matching in the graph. Given an undirected graph $G = (V, E)$, a matching M in the graph G is defined as a subset of edges such that every vertex $v \in V$ is incident to at most one $e \in M$. A maximum matching in the graph G is a matching M' such that $|M'| \geq |M|$, for every matching M.

Since the maximum matching in the graph is not unique, in the case of a large graph it is quite hard to be sure whether the returned matching is indeed the maximum matching. The witness which certifies that the program has returned the maximum matching is *an odd set cover*. The whole idea of the odd set cover originates from [13, 14], but to use the odd set cover as a witness was done in LEDA ([17]).

Before we give a definition of the odd set cover, let us define *cov* and *cap*:

Definition 2.1.1. *Let $A \subset V$ such that $|A|$ is odd number.*

1. *If $|A| = 1$, i.e. A consists of a singleton set $\{v\}$ for some $v \in V$, then* $\mathrm{cov(A)} = \{e \in E \mid e \text{ incident to } v\}$ *and* $\mathrm{cap(A)} = 1$.

2. *If $|A| = 2k + 1$ and $k > 0$, then* $\mathrm{cov(A)} = \{e \in E \mid e \text{ has both endpoints in } A\}$ *and* $\mathrm{cap(A)} = k$.

Definition 2.1.2. *Let $\mathcal{A} = \{A_1, A_2, \ldots, A_k\}$ be the family of odd sized subsets of V. $\mathcal{A}$ is called* an odd set cover *for the graph G if*

$$cov(\mathcal{A}) = \bigcup_{A \in \mathcal{A}} cov(A) = E.$$

If $\mathcal{A}$ is the odd set cover for G, then the capacity of $\mathcal{A}$ is defined as:

$$cap(\mathcal{A}) = \sum_{A \in \mathcal{A}} cap(A).$$

Figure 2.3 illustrates the previously introduced definitions. Given a graph G, the maximum matching and the odd set cover are already marked on G. The whole theory behind the idea of using the odd set cover as a witness, can be easily explained. Let $\mathcal{A} = \{A_1, A_2, \ldots, A_k\}$ be the odd set cover for G. If $|A_i| = 1$, then in $cov(A_i)$ there is at most one edge which belongs to the matching M. If $|A_i| = 2k + 1$, then in $cov(A_i)$ there are at most $k = cap(A_i)$ matching edges. Therefore, we conclude that $cap(\mathcal{A}) \geq |M|$, for every matching M and every odd set cover $\mathcal{A}$.

Since $cap(\mathcal{A}) \geq |M|$ always holds, the witness that the returned matching M is indeed maximum matching is the odd set cover such that $cap(\mathcal{A}) = |M|$. The fact that such an odd set cover $\mathcal{A}$ always exists and the construction of $\mathcal{A}$ can be found in [17].

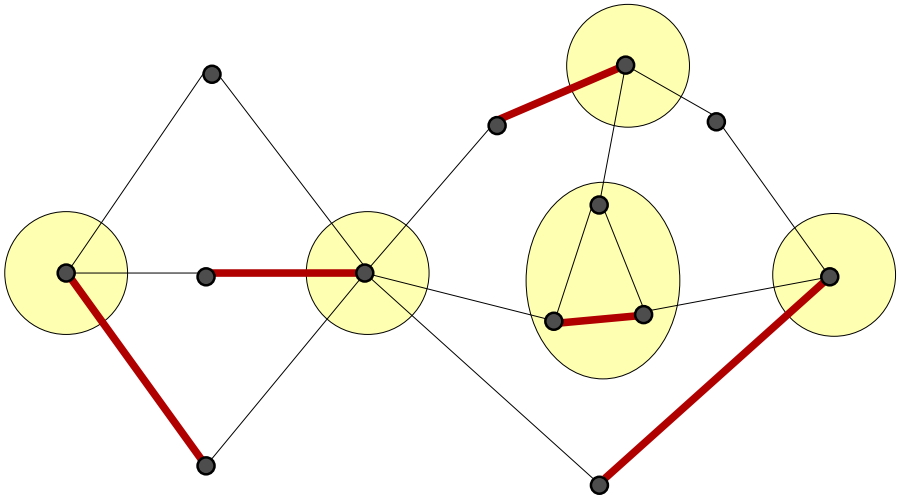

Figure 2.3: The thicker lines in the graph represent the edges which belong to maximum matching. The circled subsets of vertices form the odd set cover.

2.2 On-line and off-line checkers

Checking of data structures is a generalization of the checking concept introduced in the previous chapter.

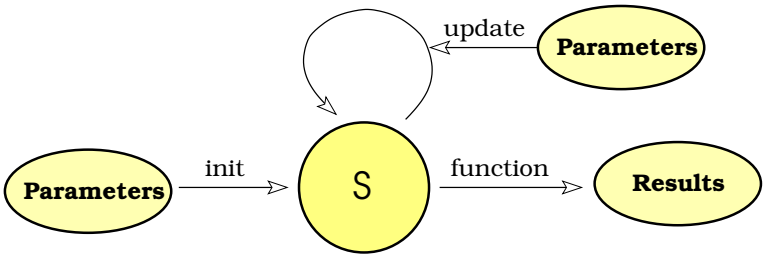

Figure 2.4: The life-cycle of a data structure S

Figure 2.4 shows the life-cycle of the data structure S. After initialization, the user may update the data structure and call functions returning results that depend on the internal state of the data structure. Each time, a function is called, the result of the function depends on the updates made so far. Using this, it is possible to construct a checker for the data structure. However, contrary

to the input to an algorithm, the sequence of updates to a data structure is not bounded in size. Therefore, the input to the function checking the data structure is also not bounded.

As was the case for program correctness checking, one can also make use of a witness. The witness is another data structure S' that is updated in parallel with the updates on the original data structure S. It should be constructed in such a way that it is easy to check whether the values returned by functions of S, are correct, using S'.

Checking data structures is distinctive from checking programs because a wrong output of a data structure does not necessarily have to be detected immediately. It is also acceptable to detect it at a later moment, after some more commands have been executed. Depending on the moment on which errors are detected, one can distinguish two types of checkers: *on-line* and *off-line checkers*. A checker is called on-line when an incorrect value is detected at the moment that is it returned. A checker is off-line, when an incorrect value will be detected eventually, but possibly at a later moment.

Often, off-line checkers are more time-efficient than on-line checkers. An example of this is the off-line checker for priority queues that we will define in Section 3.2. The checker has a complexity which is strictly less than the complexity of the priority queue. In [15], it is shown that there exists no on-line checker for priority queues with a complexity less than the complexity of the data structure itself.

2.2.1 Running times of checkers

We distinguish checkers depending on how their complexity relates to the complexity of the program (or data structure) being checked.

Definition 2.2.1. *Let P be a program or a data structure and let C_P be a checker for P. Then C_P is*

- time-efficient *if the running time of C_P is asymptotically no more than the running time of P*
- time-inefficient *if the running time of C_P is asymptotically more than the running time of P*
- time-superefficient *if the running time of C_P is asymptotically less than the running time of P*

The same notions can be applied to space. Let us illustrate those definitions with the maximum matching example from Section 2.1.1.

In LEDA there is a function `MAX_CARD_MATCHING()` which computes a maximum matching for a given graph G. The result of `MAX_CARD_MATCHING()` is a list of edges M containing the edges in the maximum matching and an odd set cover O. The odd set cover is represented by a list of nodes and to every node is associated an integer label of the set to which the node belongs.

There is also a checking function `CHECK_MAX_CARD_MATCHING()` which takes three arguments: a graph G, a matching M and an odd set cover O, and which returns a boolean value.

For the graph $G = (V, E)$ the running time of `MAX_CARD_MATCHING()` is (almost) $O(|E| * |V|)$, while the running time of `CHECK_MAX_CARD_MATCHING()` is linear in the size of the graph. Therefore, this is an example of a time-superefficient checker.

Chapter 3

The priority queue and its checker: the LEDA implementation

In the previous chapters we have introduced the general concept of checking data structures and in this chapter we will show how this principle works for a concrete data structure, namely for priority queues. We will describe the priority queue checker implemented in LEDA [17, 15]. LEDA is a C++ class library for efficient data types and algorithms. It provides algorithmic in-depth knowledge in the field of graph and network problems, geometric computations, combinatorial optimization and other. Several algorithms in LEDA certificate the correctness of their results by providing a *witness* as an additional output of the algorithm (cf. the maximal matching problem from the paragraph 2.1.1).

3.1 The priority queue data type

A priority queue is a data type that is used in many network and graph algorithms. Typically it is used in the shortest path computation. Let us first define and describe some general properties of priority queues and later we will describe the LEDA implementation of this data structure.

Definition 3.1.1. *Let $(P, <)$ be a totally ordered set, and let I be any set. A priority queue over I using $(P, <)$ is a datatype that supports the following operations:*

- create*:* *creates empty priority queue*
- insert (i,p)*:* *inserts individual $i \in I$ with priority $p \in P$*

- delete_min: *removes an object (i,p) with minimal p*
- find_min: *returns a pair (i,p) with minimal p*
- contains(i,p): *true iff the priority queue contains $i \in I$ with associated priority $p \in P$*

Some priority queues also support the deletion of any element, not only those with minimal priority. Since the focus in priority queues is on the minimal element, we assume that operations concerning the minimal element should be implemented efficiently.

Priority queues can be implemented in different ways, but mostly they are implemented as heaps: the priority queue can be implemented as Fibonacci heap, pairing heap, redistributive heap, monotone heap, k-ary heap or binary heap, but it can also be implemented as a list or a bucket. Every mentioned type of the implementation has certain advantages. For example, with the Fibonacci heap implementation the running time of both, insertion and deletion of the minimal element, is $O(\log n)$. On the other hand, if we choose the list implementation, the running time of insertion is then $O(1)$, but the running time of deletion of the minimal element is $O(n)$. Therefore, choosing the appropriate implementation mostly depends on how the data structure is going to be used.

Although there are many implementations of priority queues, all of them provide the same user interface. This means that the user uses the same commands to work with priority queues, independently of the implementation he has chosen. The default implementation used in LEDA is the Fibonacci heap implementation. The same implementation is used in the standard library of C++.

The declaration `p_queue <P,I> Q` creates a new instance `Q` of the priority queue. The elements of `Q` are of the type `I` and their priority is of the type `P`. The priority queue `Q` will be implemented using the Fibonacci heap implementation. In order to select an implementation different from the Fibonacci heap implementation, one has to use the following declaration:

```
_p_queue <P,I,new_implementation> Q(necessary_parameters);
```

It is also possible that the user could use his own implementation. In that case the declaration looks the same, only instead of `new_implementation` one has to use the name of the class where the new priority queue implementation is designed. This new implementation could also contain some errors. This yields the need for checking the correctness of the output, but this will be explained in more details in the next section.

Let us consider the following sequence of LEDA code:

```
p_queue <int,int> Q;
Q.insert (7, 0);
Q.insert (9, 0);
Q.insert (6, 0);
int p = Q.del_min();
```

```
Q.insert (8, 0);
Q.insert (3, 0);
int p = Q.del_min();
Q.insert (4, 0);
```

Notice that all the priority queue commands are implementation-independent, i.e. independently which implementation we use, the commands for insertion and deletion are the same. This sequence demonstrates all typical priority queue commands, but there are more commands which are not presented in this segment of the code. For example, the following two commands

```
pq_item it = Q.find_min();
Q.del_item(it);
```

have the same impact as `Q.del_min();` .

In all the priority queue implementations we are working with the minimal (or the maximal) element only, but never with both of them. It is again dictated by implementation. There is no implementation which could handle both the minimal and the maximal element efficiently. Therefore, we cannot arbitrarily change the priority of an element: it can be only decreased. There are implementations that support very efficiently decrease of priorities but there are no implementations that support very efficiently decrease and increase. The information part can be changed arbitrarily.

```
Q.change_inf(it, i1);
Q.decrease_p(it, p1);
```

3.2 Checking Priority Queues

As mentioned before, there is also a possibility that the user can use his own implementation of priority queues. This implementation might be faulty, so the user wants to be sure whether the result reported by `Q.find_min()` and `Q.del_min()` is indeed the minimal element in the priority queue. The first idea how to check it would be to compare priorities of all the elements in the priority queue with the reported minimal priority. Unfortunately, this idea defeats the efficient access to the element with the minimal priority. The access to the element with the minimal priority usually takes $O(\log n)$. Therefore, our first idea is not acceptable because the checking then would be more costly ($O(n)$) than the priority queue itself. It has been proven in [15] that there is no on-line time-superefficient checker for priority queues.

For the reason mentioned above, a checker which adds only small overhead to priority queue operations has to be an off-line checker. In this section we explain the algorithm used for the checker, while in later chapters we will give the formal proof of its correctness.

The class *checked_p_queue* implements a checker for priority queues. The user interface for the *checked_p_queue* is the same as the interface for the unchecked priority queue, so the user can use checked priority queues in the

same manner as he would use unchecked priority queues, without making any changes in his code. The only difference is that the user has to declare the use of checked priority queues:

```
_p_queue<P, I, new_impl> PQ;
checked_p_queue<P,I> CPQ(PQ);
```

The priority queue checker implemented in LEDA uses the encapsulation model for checking the data structures. Sometimes in the literature it is also called "the client-checker-server model" [15, 1].

Figure 3.1 describes the behaviour of such a model. The priority queue checker is a program layer that acts as an intermediary between the user and the priority queue. The checker monitors the behaviour of the priority queue and if there was no error, it stays silent. In case of an error, the checker reports the error to the user. This means that on the user level, if the priority queue operates correctly, there is no difference between checked and unchecked priority queues. Since the checker implemented in LEDA is not an on-line checker, but an off-line, a potential error will not be reported immediately but eventually.

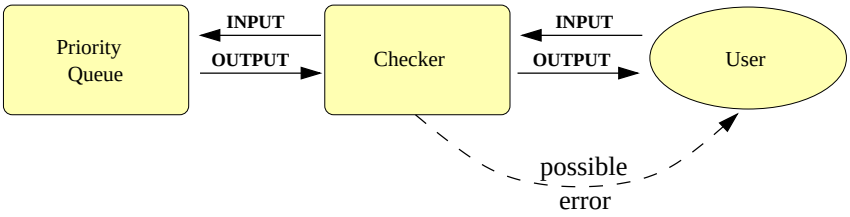

Figure 3.1: The client-checker-server model for checking priority queues

In the priority queue checker it is not necessary to consider all priority queue commands. It is sufficient to consider only the following constructors:

- `p_queue <P,I> Q;`
- `Q.insert (p, i);`
- `Q.del_item(it);`
- `P p = Q.del_min();`

since all other priority queue commands concerning priority manipulations could be synthesized by some above mentioned commands. For example, `Q.find_min()` is synthesized by a `Q.del_min()` and an insertion of the returned element. From now on, we consider that the priority queue only supports those four commands mentioned above.

The key idea of the checker is the fact that whenever we execute `Q.del_min()` all the remaining elements in the priority queue must have priority at least as

large as the reported minimum, if the implementation is correct. Therefore, to each element of the priority queue we assign a *lower bound*, which should indicate the smallest value of its priority, assuming that the implementation is not faulty.

Definition 3.2.1. *A lower bound of an element at the time t is a maximal priority reported by all* `Q.del_min()` *operations performed between the moment when the element was inserted into the priority queue and the time t.*

Every time when some element of the priority queue is accessed (and this can be done only via deletion), the checker compares the element to its lower bound. If the element is greater or equal to its lower bound, no error is reported and the checker remains silent. But, if the element is strictly smaller than its lower bound, this indicates that an error occurred during the code execution and the checker alarms the user. The resulting checker is an off-line checker. When `Q.del_min()` returns a non-minimal element, this will be noticed only when one of the smaller elements that are present in the queue is retrieved.

3.2.1 The Algorithm for the Priority Queue Checker

The lower bound of the element depends on all past `Q.del_min()` operations. Since the system of lower bounds is changing during the code execution according to the reported minimums, let us explain how the lower bound of an element evolves through the time.

The priority queue checker performes the following actions in parallel to the operations in the priority queue:

- when the element is inserted into the priority queue, its lower bound is initially set to $-\infty$, because nothing is known about it yet
- when the element is deleted from the priority queue, the checker compares the element and its lower bound and in case that the element is strictly smaller than its lower bound, reports an error
- when `Q.del_min()` returns (i, p), the checker takes a priority p and sets all lower bounds which are smaller than p to p. If the lower bound of some element is greater than p, it remains unchanged. Since `Q.del_min()` also deletes the reported element from the priority queue, the checker has to inspect whether the reported minimal element is greater than its lower bound

Let us consider a fragment of code given in Paragraph 3.1 and let us assume that the implementation is correct. After the code execution, the priority queue build from it contains four elements. Figure 3.2 represents that priority queue in a two-dimensional coordinate plane, where the x-axis corresponds to the time and the y-axis corresponds to the priority value. The horizontal line in the plane represents the lower bound.

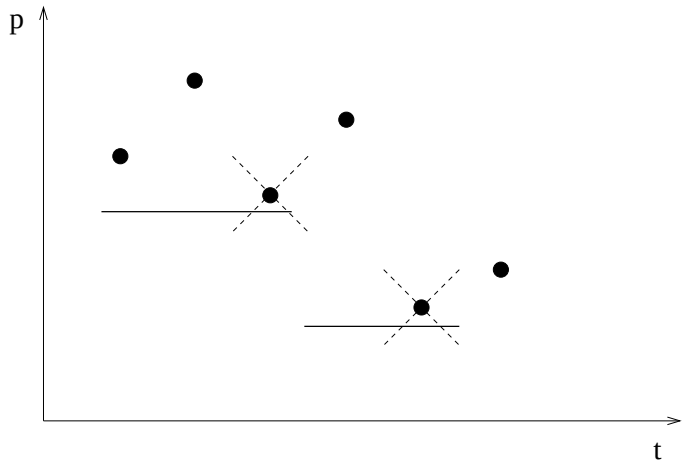

Figure 3.2: The system of lower bounds build from the code fragment given in Paragraph 3.1

3.2.2 Some simple properties of lower bounds

Let P be some LEDA code fragment concerning priority queues and let us illustrate the execution of P in a two-dimensional coordinate plane, as before. We observe that lower bounds are monotonically decreasing from left to right, i.e. if an item it_1 was inserted into the priority queue before the item it_2, then, assuming the correct implementation of priority queues, the lower bound of it_1 must be at least as large as the lower bound of it_2.

This follows from the definition of the lower bounds. Since it_2 was inserted after it_1, all the minimal elements that were used for computing the lower bound of it_2 were also used for the lower bound of it_1.

Following [17], the form of lower bounds is named *a staircase of lower bounds*. Figure 3.3 illustrates the process of updating the staircase of lower bounds. After the new minimal priority p is reported by the `Q.del_min()` command, the value of all the lower bounds which were smaller than p, is increased to p. The staircase form is preserved in the updated system of lower bounds as well.

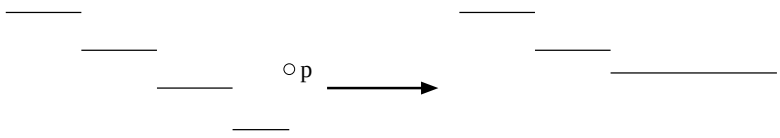

Figure 3.3: Updating the staircase of lower bounds after reporting a priority p as a minimal element

3.3 Data structures to maintain the system of lower bounds

In this section we shortly explain the data structures used for building and updating a staircase of lower bounds. The more detailed explanation, templates and running times can be found in [17], while here we give just a basic idea. Figure 3.4 illustrates the data structures which were used to build and maintain the whole system of lower bounds.

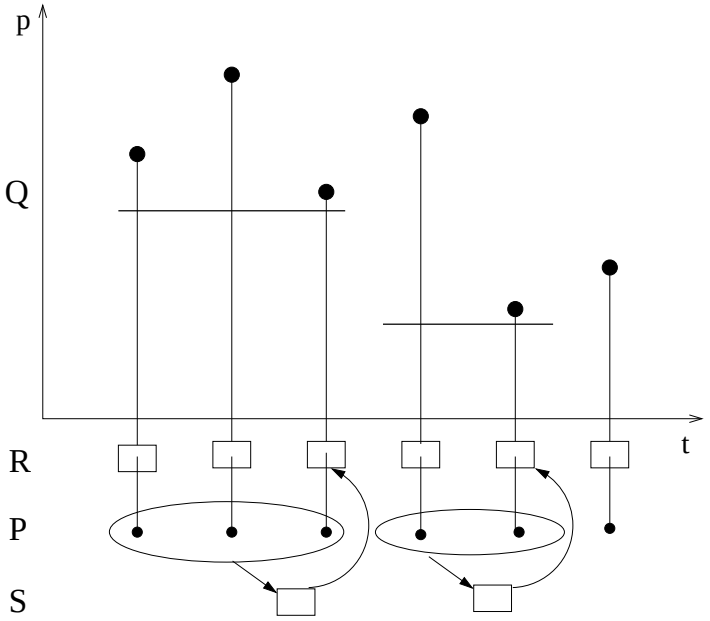

Figure 3.4: Data structures used to maintain the system of lower bounds

Let Q be a priority queue. To every element in Q we associate an element in the list R. The list R is a list of elements from Q which is linearly ordered by the time of insertion of elements. A partition P divides the list R into blocks. Let e be the element contained in P and let e_p denote the lower bound of e. If elements x and y are both contained in the same block of the partition P, then $x_p = y_p$. The blocks of P are maintained in a linearly odered list S which is odered according to the lower bounds of blocks in P.

Defined in such a manner, the checker can easily maintain this whole system

of data structures. While the maintenance for the insertion and deletion of an element is fairly simple, the maintenance of the `Q.del_min()` command requires more manipulations with the data structures. Let e^* be the priority returned as the output of the `Q.del_min()`, performed on the priority queue Q. The checker deletes e^* from R and P and checks whether $e_p^* \leq e^*$. All blocks in P whose "canonical" item is smaller than e^* are united into the new block with the canonical element e^*.

Chapter 4

A mathematical model for the priority queue checker

We are going to show that the algorithm used for the priority queue checker works correctly, i.e. if the checker did not report any mistake, then no error occurred during the code execution. In order to prove its correctness, we have developed a mathematical model whose description is given in this chapter. In the next chapter the model will be formalized in first order logic and using Saturate syntax, but in this chapter we only establish a link between the LEDA code, priority queues and checked priority queues.

4.1 Modeling LEDA code

The priority queue data structure is defined as a collection of items which are built of two parts: the priority part and the information part. Let (p, i) be such an item: then p is the priority from a linearly ordered data type P and i is information of type I. I can be any data type. Since all priority queue operations concern only the priority part, we model priority queues simply as a collection of items from a linearly ordered data type *Element*. Erasing the information part of the data structure does not lead to a loss of any functionality as the information part is not significant for priority queue outputs.

With this restriction and with the previously introduced reduction of all the priority queue commands to insertion, deletion and deletion of an minimal element only, we gain the following definition of LEDA code concerning priority queues:

Definition 4.1.1. *Priority queue LEDA code is a word from the alphabet*

$$\Sigma = \{\mathsf{ins}(e), \mathsf{del}(e), \mathsf{dmin} \mid e \in \mathit{Element}\}$$

Figure 4.1 demonstrates the transformation of a fragment of LEDA code P into the corresponding word from the alphabet Σ. From now on, whenever we talk about LEDA code, we refer to the word w which represents that specific code fragment.

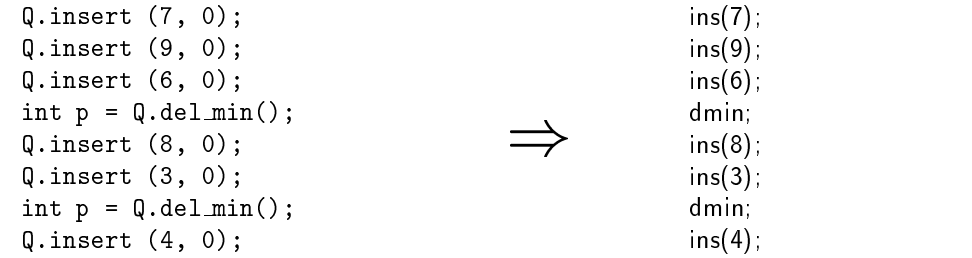

Figure 4.1: Translation of LEDA code into a word from the alphabet Σ

Every word from the alphabet Σ can be seen as a list whose domain is set Σ. We assume the following inductive definition of lists with domain D:

Definition 4.1.2. *The empty list* nil *is a list with domain D. If l is a list with domain D, then for every $d \in D$,* cons*(l,d) is also a list with domain D. The function* cons*(l,d) appends the element d at the end of the list l.*

The definition of lists which we are using differs from the standard definition, since in the standard definition *cons* puts the newly added element at the beginning of the list. The reasons for our approach are historical. Our earliest models and specification involved only priority queues and they were modeled with appending the element at the end of the list. Later on, as we have introduced a system of lower bounds and words over Σ, it was more natural to model them in the same way as we have modeled priority queues. Since we already had a developed model for priority queues, it was easier to redefine lists than to rebuild the already existing models.

Not only the words from the alphabet Σ are modeled as lists, but we also recognize a system of lower bounds as a list.

Let "." represent the concatenation of two words. Let w be a word from the alphabet Σ and $\sigma \in \Sigma$ be a letter: $w.\sigma$ denotes then the shortcut for $w.\{\sigma\}$, i.e. since words are modeled as lists, $w.\sigma$ is a shortcut for $\mathit{cons}(w, \sigma)$. The empty word ε corresponds to the empty list nil.

We assume that the following standard list operations are defined on every word w: concatenation, prefix and suffix.

4.2 Modeling (un)-checked priority queues

We define priority queues also inductively, in a similar way as we defined lists, only the constructors for priority queues are called *empty* and *insert*. Their behaviour is identical to the behaviour of *nil* and *cons*.

Checked priority queues are modeled as lists. We use a list over $Element \times (Element \cup \{-\infty\})$ to represent a system of lower bounds. If (e_1, e_2) occurs in a system of lower bounds, this means that e_1 has associated lower bound e_2.

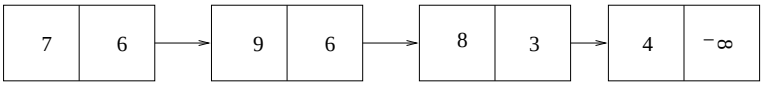

Figure 4.2: Model of a checked priority queue built from the code shown on figure 4.1

Although this model is much simpler than the whole system of data structures which is used to maintain the system of lower bounds (cf. figures 3.4 and 4.2), it is already powerful enough to specify and describe all the desired properties of the checker. Let us recall that our goal is not to verify the correctness of the data structure but to check its output.

4.3 Characterization of priority queues

In the next two sections we will establish a link between the LEDA code, priority queues and checked priority queues. Figure 4.3 shows the dependencies between them.

Let *Queues* be the set of all priority queues defined over the linearly ordered set *Element*. On the set *Queueus* we define the following operations: *insert*, *delete* and *del_min*. We assume that the implementations of *insert*/*delete* are correct, but we assume nothing about the correctness of *del_min*. In order to distinguish the implementation-dependent *del_min* command from the correct *del_min* command, we use the name *del_min_impl*. We only assume that *del_min_impl* is going to delete some element from the queue and return its value.

Let P be a code fragment using priority queues. In order to construct the priority queue Q out of P, we define a function *ex* (which stands for *execute*). Function *ex* treats the implementation of priority queues as a black box: the structure and inner states of Q are not visible to *ex*. It has two arguments: a code fragment w and a priority queue q. The result should be the new priority queue q'.

The function *del_min_impl* deletes the minimal element, but also return its

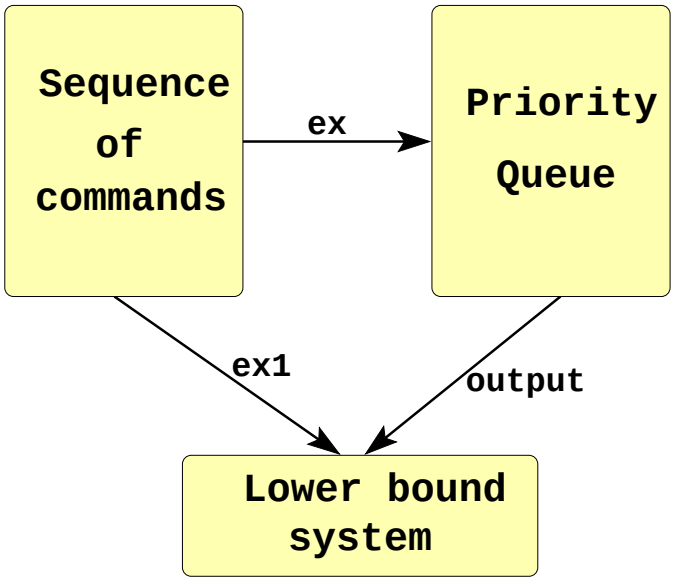

Figure 4.3: Connection between code, priority queues and checked priority queues

value. In order to deal with this case, the function ex will return an ordered pair $\langle q', p\rangle$, where q' is the modified priority queue and $p \in Element$ is the priority value. We need an additional object $* \notin Element$ (denoting `undefined`) which is going to be used as the auxiliary output for *insert* and *delete*. Since the result of the function ex is an ordered pair, we are going to use projection functions: $\langle q, e\rangle.Q = q$ and $\langle q, e\rangle.E = e$.

The function ex is defined as follows:

$$
\begin{aligned}
&ex(\varepsilon,\ q) \ =\ \langle empty,\ *\rangle\\
&ex(w.\mathsf{ins}(e),\ q) \ =\ \langle insert(ex(w,q).Q,\ e),\ *\rangle\\
&ex(w.\mathsf{del}(e),\ q) \ =\ \langle delete(ex(w,q).Q,\ e),\ *\rangle\\
&ex(w.\mathsf{dmin},\ q) \ =\ del\_min\_impl(ex(w,q).Q)
\end{aligned}
$$

4.4 Characterization of lists of lower bounds

In this section we define a function $ex1$ that constructs the system of lower bounds resulting from a code fragment $w \in \Sigma^*$.

The lower bound system is represented by a list and the operations defined on the system of lower bounds should be all those operations that are used in the algorithm for the checker:

- appending of an element to the end of the list

- deletion of an element from the list
- upraising of the system of lower bounds regarding the reported minimal element

The function $ex1$ has two arguments: a code fragment w and a lower bounds list l. The result is the new and modified list l'. Since the operations on the system of lower bounds do not return any values , there is no need that $ex1$ returns a pair. The execution of the function $ex1$ corresponds to the algorithm for the priority queue checker, defined in Section 3.2.1.

$$
\begin{aligned}
&ex1(\varepsilon,\ l) \ =\ nil\\
&ex1(w.\mathsf{ins}(e),\ l) \ =\ cons(ex1(w,l),\ (e,-\infty))\\
&ex1(w.\mathsf{del}(e),\ l) \ =\ remove(e,\ ex1(w,l))\\
&ex1(w.\mathsf{dmin},\ l) \ =\ adjust(e^*,\ remove(e^*,\ ex1(w,l))),\\
&\qquad \text{where } e^* = del\_min\_impl(ex(w,empty)).E
\end{aligned}
$$

The function $adjust$ has two parameters: an element e and a list l and its result is the new list l'. It assigns the value e to all lower bounds in l which are smaller than e. It is actually the function which upraises the system of lower bounds.

The value e^* which is passed to the $adjust$ function during code execution, is the value which was returned by the del_min_impl operation. Thus, the system of lower bounds is also modeled implementation-dependently.

4.5 Characterization of the correctness of the checker

In previous sections we have described the models for priority queues and a system of lower bounds and in this section we informally explain what it means to prove the correctness of checker. The formal statement of the checker correctness can be read in Paragraph 6.1 (Theorem 6.1.1).

The implementations of priority queues are not verified, but checked, i.e. we do not analyze the code of the implementation but we test whether it returns the correct output. What we prove is the following: If during code execution the checker did not report any error, then every returned output was correct. In other words, the element returned by every dmin operation in this code sequence was indeed the minimal element.

Let w be a code fragment using priority queues. If during the execution of w the checker did not report any error, then w is called an *error-free* code sequence. Error-freeness of w can be easily characterized as: w is error-free if and only if at any moment where an element e was deleted from the priority queue, e was greater or equal than its lower bound.

It is not enough to have an error-free code sequence in order to conclude that every dmin command returned the minimal element. The reason for this is that

lower bounds are only checked when the element is retrieved. In other words, if dmin returns a non-minimal element, then some elements in the lower bound system will have an incorrect lower bound. However this will be detected only when these elements are retrieved. The code sequence is called *complete* if after the code execution the resulting q contains no elements. Using these notions, we can give the correctness statement:

For every code fragment w that is complete and error-free the output of every dmin command was the minimal element.

The formal proof will follow in the next chapters.

Chapter 5

A formal model

In the previous chapter we have described models for priority queues and their checker informally. In this chapter we will give an axiomatized specification of their behaviour written in terms of first order logic and using a simplified Saturate syntax [19, 25]. The Saturate System is a theorem prover for first-order logic, primarily based on saturation. Its main focus is on the efficient treatment of transitive relations by term rewrite techniques [3, 4] and on the restriction of the search space by applying techniques for detecting redundancy of clauses. The Saturate System is written in Prolog and therefore its syntax is similar to the syntax of Prolog: variables begin by a capital letter, function symbols start with lower-case letter and other similarities.

5.1 The specification of a priority queue and its checker

The whole specification of a priority queue and its checker consists of several smaller modules, where every module can be seen as one conceptual component of the specification. The hierarchy of the modules is shown in figure 5.1. Arrows represent module dependencies. To give a complete description of the specification, we will use a Bottom-up approach.

5.1.1 Specification of total order

A priority queue is modeled as a collection of elements from the totally ordered set *Elements*. Therefore, the first module we have written is the one that describes a totally ordered set. A total order is a relation $<$ defined on the set *Element*, with the following properties: $<$ is irreflexive, transitive and total. We also define the relation $\leq$ as a reflexive closure of the relation $<$. Moreover, in the set *Element* we have specified two special elements: $-\infty$ and ∞. Those

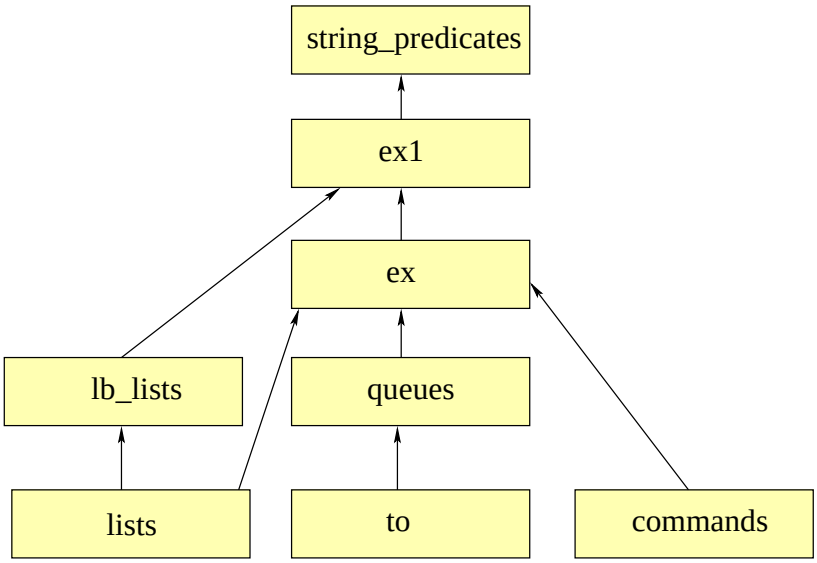

Figure 5.1: The hierarchy of modules used in the specification of priority queues and their checkers

constants represent the minimal and the maximal element in the set *Element*. Those requirements on a totally ordered set are formalized in terms of the first-order logic.

5.1.2 Specification of priority queues

We assume that a priority queue is defined by the following operations: *insert*, *delete*, *del_min* and *del_min_impl*. *del_min* and *del_min_impl* are different in the sense that *del_min* is always correct and always removes the minimal element from the queue, whereas *del_min_impl* is an implementation-dependent operation and the only thing we know about *del_min_impl* is that it removes some element from the queue. A priority queue containing no elements is represented by a constant *empty*.

functions:

$-\infty : Element$

$\infty : Element$

predicates:

$<: Element \times Element$

$\leq: Element \times Element$

axioms:

$X < Y \ \wedge \ Y < Z \Rightarrow X < Z$

$\neg X < X$

$X < Y \ \vee \ Y < X \ \vee \ X = Y$

$-\infty \leq X$

$X \leq \infty$

$X \leq X$

$X < Y \Rightarrow X \leq Y$

$X \leq Y \Rightarrow X = Y \ \vee \ X < Y$

Figure 5.2: *to* module

uses: *to* module

functions:

empty: $Queues$
insert: $Queues \times Element \rightarrow Queues$
delete: $Queues \times Element \rightarrow Queues$
del_min: $Queues \rightarrow QE\_pair$
del_min_impl: $Queues \rightarrow QE\_pair$
queue: $QE\_pair \rightarrow Queues$
elem: $QE\_pair \rightarrow Queues$

predicates:

contains: $Queues \times Element$

axioms:

$empty \neq insert(Q, E)$
$(insert(Q_1, E_1) = insert(Q_2, E_2)) \Leftrightarrow (Q_1 = Q_2 \ \wedge \ E_1 = E_2)$

$\neg contains(empty, E)$
$contains(insert(Q, E_1), E_2) \Leftrightarrow (contains(Q, E_2) \ \vee \ E_1 = E_2)$

$queue(Q, E) = Q$
$elem(Q, E) = E$

$delete(empty, E) = empty$
$delete(insert(Q, E), E) = Q$
$E_1 \neq E_2 \Rightarrow delete(insert(Q, E_1), E_2) = insert(delete(Q, E_2), E_1)$

$del\_min(empty) = (empty, -\infty)$
$del\_min(insert(empty, E)) = (empty, E)$
$E \leq elem(del\_min(Q)) \ \wedge \ Q \neq empty \ \Rightarrow \ del\_min(insert(Q, E)) = (Q, E)$
$elem(del\_min(Q)) < E \ \wedge \ Q \neq empty \Rightarrow$
$\quad del\_min(insert(Q, E)) = \big(insert(queue(del\_min(Q)), E), \ elem(del\_min(Q))\big)$

$del\_min\_impl(empty) = (empty, -\infty)$
$queue(del\_min\_impl(Q)) = delete(Q, elem(del\_min\_impl(Q)))$
$Q \neq empty \Rightarrow contains(Q, elem(del\_min\_impl(Q)))$

Figure 5.3: *queues* module

We have restricted possible priority queue models to "term generated" models only. This means that only priority queues we will investigate are those that are results of applying the above-mentioned queue operations on *empty*. For example, one such a queue is $delete(insert(insert(empty, E_1), E_2), E_1)$. Moreover, since we axiomatize all queue operations using only *empty* and *insert*, the further restriction would be that we do not investigate all term generated models, but exclusively those that are modeled using only *empty* and *insert*. Thus, the previous example would be represented with the following priority queue: $insert(empty, E_2)$.

From now on, we only consider those priority queues that are modeled inductively: *empty* is a priority queue. If Q is a priority queue, then for every element E, $insert(Q, E)$ is also a priority queue. The inverse case also holds: every priority queue Q we investigate is either *empty* or there exists a priority queue Q' and an element E such that $Q = insert(Q', E)$. The reason behind this definition and this restriction lies in the fact that later every lemma about some queue property can be proved using inductive reasoning and moreover it reduces the induction step to the *insert* case only.

Since we were using *empty* and *insert* as constructors for other queue operations and since we also want to reduce the queue induction step to the *insert* case only, we have to prove that the result of applying *delete*, *del_min* or *del_min_impl* to a term of the form *empty*/*insert* is again a term of the same form. More detailed explanations and proofs can be found in Paragraph 6.3

5.1.3 Specification of lists

A list is defined in a fairly similar way as priority queues. A list has two constructors: *nil* and *cons*, analogously to *empty* and *insert*. Again we have assumed the completeness of such a definition (cf. 6.3) and we have expressed all list operations through *nil* and *cons*. Although in most of the standard interpretations, *cons* puts the element at the beginning of a list, in our specification we assume that *cons* appends an element to the end of the list. In this module we just give the general specification for lists, without describing operations that are specific for lower bounds lists. Complete specification for lists in terms of first-order logic can be found in [20]. Lists are used to represent a sequence of commands as well as a system of lower bounds.

---

functions:

nil: $Lists$

cons: $Lists \times Domain \rightarrow Lists$

predicates:

prefix: $Lists \times Lists$

includes: $Lists \times Domain$

axioms:

$nil \neq cons(L, E)$
$(cons(L_1, E_1) = cons(L_2, E_2)) \Leftrightarrow (L_1 = L_2 \ \wedge \ E_1 = E_2)$

$prefix(L, L)$
$prefix(L_1, L_2) \Rightarrow prefix(L_1, cons(L_2, E))$
$prefix(L, nil) \Rightarrow L = nil$
$prefix(L_1, cons(L_2, E)) \Rightarrow \big(L_1 = cons(L_2, E) \ \vee \ prefix(L_1, L_2)\big)$

$\neg includes(nil, E)$
$includes(cons(L, E_1), E_2) \Leftrightarrow (includes(L, E_2) \ \vee \ E_1 = E_2)$

Figure 5.4: *lists* module

5.1.4 Specification of lower bounds lists

A system of lower bounds (LBS) is modeled as a list. We can formally define it in the following way: $LBS = (Element \times Element)^*$. All operations on lists defined before are inherited here. We had to slightly modify some of lists operations as here every list item consists of two elements: the underlying priority queue element itself and its lower bound. The function *remove* deletes an element from the list, while the function *adjust* lifts up the whole system of lower bounds.

---

uses: *lists* module

functions:

remove: $Element \times LBS \rightarrow LBS$
adjust: $Element \times LBS \rightarrow LBS$

axioms:

$\big((E_1, E_2) = (E_3, E_4)\big) \Leftrightarrow (E_1 = E_3 \ \wedge \ E_2 = E_4)$

$remove(E, nil) = nil$
$remove(E_1, cons(L, (E_1, E_2))) = L$
$E_1 \neq E_2 \Rightarrow remove(E_1, cons(L, (E_2, E_3))) = cons(remove(E_1, L), (E_2, E_3))$

$adjust(E, nil) = nil$
$E_3 < E_1 \Rightarrow adjust(E_1, cons(L, (E_2, E_3))) = cons(adjust(E_1, L), (E_2, E_1))$
$E_1 \leq E_3 \Rightarrow adjust(E_1, cons(L, (E_2, E_3))) = cons(adjust(E_1, L), (E_2, E_3))$

Figure 5.5: *lb_lists* module

5.1.5 Specification of commands

As we have already mentioned, a sequence of commands is also built as a list. But before we will go into the description of how to execute one such a sequence and build up the priority queue and the system of lower bounds, we have to emphasize that all those commands (letters from the alphabet Σ) are different. We have already done a similar thing when we included axioms for equality of queues and for equality of lists into the specification. The reason for that is again the automated theorem prover. Although our human way of thinking clearly distinguishes between lists nil and $insert(L, E)$, they can be interpreted to be equivalent in some models and for that reason we had to implicitly introduce (in)equality in the set of axioms.

---

axioms:

$dmin \neq del(E)$
$dmin \neq ins(E)$

$ins(E1) \neq del(E2)$

$ins(E1) = ins(E2) \Leftrightarrow E1 = E2$
$del(E1) = del(E2) \Leftrightarrow E1 = E2$

Figure 5.6: *commands* module

5.1.6 Specification of the function *ex*

---

uses: *queues, lists, commands* modules

functions:

ex: $\Sigma^* \times Queues \rightarrow Queues \times (Element \cup \{ok\})$

axioms:

$ex(nil, Q) = (empty, ok)$
$ex(cons(S, ins(E)), Q) = (insert(queue(ex(S, Q)), E), ok)$
$ex(cons(S, del(E)), Q) = (delete(queue(ex(S, Q)), E), ok)$
$ex(cons(S, dmin), Q) = del\_min\_impl(queue(ex(S, Q)))$

Figure 5.7: *ex* module

The function ex is used to build a priority queue out of a sequence of com-

mands. One could naively assume that the type of the function *ex* should be $ex : \Sigma^* \times Queues \rightarrow Queues$. But, the queue operation *del_min_impl* returns two values: a modified queue and an element, which means that we cannot use the naive specification. For that reason we use the constant *ok* as a supplementary output in other queue operations.

5.1.7 Specification of the function *ex1*

The function *ex1* is used to build a system of lower bounds. A system of lower bounds cannot be built only from a sequence of commands, but it also needs the output of an appropriate priority queue. This is because the system of lower bounds is maintained by the output of the *del_min* operations. The axioms for the function *ex1* fully describe the algorithm for the priority queue checker.

---

uses: *lb_lists, ex* modules

functions:

ex1: $\Sigma^* \times LBS \rightarrow LBS$

axioms:

$ex1(nil, L) = nil$

$ex1(cons(S, ins(E)), L) = cons(ex1(S, L), (E, -\infty))$

$ex1(cons(S, del(E)), L) = remove(E, ex1(S, L))$

$ex1(cons(S, dmin), L) = adjust(e^*, remove(e^*, ex1(S, L)))$,
where $e^* = elem(ex(cons(S, dmin), empty))$

Figure 5.8: *ex1* module

Chapter 6

Automatic Verification

After the formal specification of the system of lower bounds, we give a proof of the correctness of the checker. This is also proved in the Saturate, but the Saturate proofs are not included in this chapter. The proofs can be found in [27], but we give detailed proof schemes for all major theorems. While describing those schemes, we will mention difficulties we have faced during proving and working with the automated theorem prover, and we will also describe how we worked out those difficulties.

6.1 Specification of the correctness of the checker

With the specification of the function $ex1$ we have almost finished the description of figure 5.1. The purpose of the introduced specification is to help us state a theorem which will prove the correctness of the priority queue checker implemented in LEDA. Let us recall that we need to confirm that, if during the code execution the checker did not report any error, then there was no error at all, i.e. every element returned by a *dmin* operation is indeed the minimal element.

In order to formalize and prove that theorem, we have to introduce some additional predicates. A sequence of commands s is called *error_free* if the checker did not report any error, i.e. every time when some element is accessed, its lower bound is less than or equal to the element itself. As an element in the priority queue can be accessed only via deletion, we always have to compare the element which we are about to delete and its lower bound.

A sequence of commands is called *complete* if $ex(s, empty) = (empty, *)$, where $*$ can be arbitrary element of $Element \cup \{ok\}$.

A sequence of commands is described as *correct* if every element returned

uses: *ex1* module

predicates:
complete: Σ^*
error_free: Σ^*
correct: Σ^*

axioms:
$complete(S) \Leftrightarrow ex(S, empty) = (empty, *)$

$error\_free\ (nil)$
$error\_free\ (cons(S, ins(E))) \Leftrightarrow error\_free\ (S)$
$error\_free\ (cons(S, del(E))) \Leftrightarrow$
$(error\_free\ (S) \wedge \forall E_1\ includes(ex1(S, nil), (E, E_1)) \Rightarrow E_1 \leq E)$
$error\_free\ (cons(S, dmin)) \Leftrightarrow$
$(error\_free\ (S) \wedge \forall E\ includes(ex1(S, nil), (e^*, E)) \Rightarrow E \leq e^*)$,
where $e^* = elem(ex(cons(S, dmin), empty))$

$correct(S) \Leftrightarrow$
$\forall S_1 \forall S_2 \big((prefix(S_1, S) \wedge S_1 = cons(S_2, dmin)) \Rightarrow$
$elem(del\_min\_impl(Q)) = elem(del\_min(Q))\big)$,
where $Q = queue(ex(S_2, empty))$

Figure 6.1: The first part of the *string_predicates* module

by a *del_min_impl* command was indeed the minimal element in the queue.

Let S be a sequence of priority queue commands and let us assume that the checker did not report any error ($error\_free(S)$). We also have to assure that every element which was inserted into the queue was at some point accessed, otherwise there could be an element left in the queue whose lower bound is greater than the element itself ($complete(S)$). Those two facts are united in the formula $complete(S) \wedge correct(S)$. Since no error was reported and every element which was inserted into the queue was at some point accessed, our goal is to conclude that then the code sequence was executed properly, according to the specification of priority queues, and every element returned by a *del_min_impl* command was the minimal element in the queue. This is described with the predicate $correct(S)$. In other words, we need to prove that the following formula holds for every S, $S \in \Sigma^*$:

$$(complete(S) \wedge error\_free(S)) \Rightarrow correct(S) \tag{6.1}$$

The formula (6.1) proves that the possibly incorrect implementation-dependent

output of *del_min* operation is going to be reported by the priority queue checker. To prove (6.1), we could try structural induction on S. But, clearly, it would not work, as $complete(cons(S, comm))$ does not imply $complete(S)$ and we cannot apply the induction hypothesis. This means that we have to state a more general theorem and then prove the formula (6.1) as that generalized theorem's corollary.

In the generalized theorem we keep the predicate $error\_free$ as it does not have any significant influence on the induction hypothesis, but the predicate *complete* has to be replaced. The predicate *complete* requires that all commands from S are executed and that the priority queue built out of S is *empty*, i.e. it does not contain any element. If we take an arbitrary error-free sequence S, not necessarily the complete one, the priority queue built out of S might contain some elements. Let us take the following error-free sequence: *ins(1); ins(2); dmin;* and let us assume that the *dmin* command has returned the element 2. Then no mistake is reported although, the implementation is clearly faulty. In order to assure that a generalized version of the predicate *complete* will hold, we require that all elements left in the queue built out of S must be greater or equal to their lower bounds. This corresponds to the queue operation check() defined and used in [15].

We add two more predicate symbols to the specification of string predicates. The predicate symbol *preconditions* unites the operation check() and the error-freeness of a sequence, while the predicate symbol *both_properties* is only a shortcut.

---

uses: *ex1* module

predicates:
  preconditions: Σ^*
  both_properties: Σ^*

axioms:
  $preconditions(S) \Leftrightarrow$
    $\big(error\_free(S) \ \wedge \ \forall E_1 \ \forall E_2 \ (includes(ex1(S, nil), (E_1, E_2)) \Rightarrow E_2 \leq E_1\big)$

  $both\_properties(S) \Leftrightarrow (preconditions(S) \ \wedge \ correct(S))$

Figure 6.2: The second part of the *string_predicates* module

Now we can state the main theorem we need to prove in order to show that the algorithm used in LEDA works correctly:

Theorem 6.1.1. (The Correctness Theorem) *Let $S \in \Sigma^*$ be a sequence of*

commands on a priority queue. Then

$$\forall S\big(preconditions(S) \Rightarrow correct(S)\big) \tag{6.2}$$

6.2 The proof of the correctness of the checker

The rest of this thesis will be dedicated to the proof of theorem 6.1.1, since it was our goal to prove the correctness of the checker and all the previously introduced specification developed in order to be able to state and prove theorem 6.1.1.

Figure 6.3 represents the proof scheme for the formulas (6.1) and (6.2). Both formulas were proved using The Saturate System. The complete specification and complete proofs for every theorem from figure 6.3 can be found in [27]. In parallel with reading the next sections concerning the automated proof of the theorem, the reader could also execute the Saturate files.

Names in the figure represent the formulas (for example, the name *corollary* represents the formula (6.1)), while arrows in the figure represent "interaction".

Let us briefly explain what do we have in mind when we use the term "interaction". Sometimes in the proof of a theorem T we were using already proved theorems T_1, T_2,... In that case theorem T_i is asserted as a valid formula in the module containing theorem T and the theorem prover treats T_i as an input. When in figure 6.3 an arrow points from the theorem T_1 to the theorem T_2, the theorem T_1 was already proved and it is used to prove the theorem T_2.

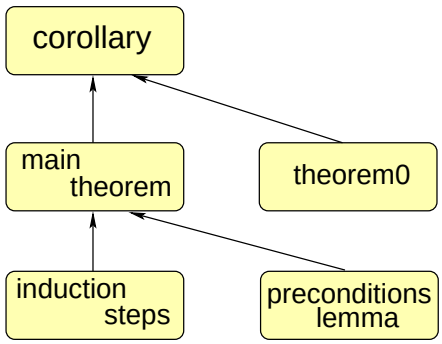

Figure 6.3: The scheme for the proof of the main theorem and its corollary

A new predicate symbol *main_theorem* was introduced only as a shortcut for the formula (6.2):

$$main\_theorem(S) \Leftrightarrow (preconditions(S) \Rightarrow correct(S))$$

Therefore our goal is to show that $main\_theorem(S)$ holds for every S, $S \in \Sigma^*$. Because of inductive definition of the sequences, this fact was proved by structural induction: first we have confirmed that $main\_theorem(nil)$ holds and then we proved induction step: if $main\_theorem(S)$ holds, then also, for every $comm \in \Sigma$, $main\_theorem(cons(S, comm))$ holds. To prove the induction step, we have been using two theorems that have already been proved: $general\_induction\_step$ and $preconditions\_lemma$ (cf.figure 6.3).

The $general\_induction\_step$ was the very first theorem we proved in the Saturate System and it states that the formula (6.3) holds for every $S \in \Sigma^*$ and for every $comm \in \Sigma$:

$$(both\_properties(S) \ \wedge \ preconditions(cons(S, comm))) \Rightarrow correct(cons(S, comm)) \tag{6.3}$$

In the $preconditions\_lemma$ we have proved that

$$preconditions(cons(S, comm)) \Rightarrow preconditions(S) \tag{6.4}$$

for every $S \in \Sigma^*$ and for every $comm \in \Sigma$.

More detailed proofs of the formulas (6.3) and (6.4) are discussed in paragraphs 6.2.1 and 6.2.2, while *theorem0* is described in Paragraph 6.2.3. *theorem0* says that every element which is contained in the priority queue is also included in the lower bound system:

$$contains(queue(ex(S, empty)), E) \Leftrightarrow \exists E_1 \ includes(ex1(S, nil), (E, E_1)) \tag{6.5}$$

Both, *theorem0* and $main\_theorem$ are used to prove *corollary* which represents the formula (6.1). Its name comes from the fact that *corollary* is just a special case of theorem (6.2).

$$(complete(S) \ \wedge \ error\_free(S)) \Rightarrow correct(S)$$

As a reminder, the interpretation of *corollary* is: let $S \in \Sigma^*$ be some sequence of commands on priority queues. After executing S on *empty* the result is *empty* again (the predicate *complete*). This means that every element inserted into a queue was accessed at some point, compared to its lower bound and found greater or equal (predicate $error\_free$). In that case every result of a *dmin* operation was the minimal element (predicate *correct*).

But, after reading the *corollary* file, one can notice that we did not include the entire $string\_predicates$ module. The reason for this is strictly technical. The whole theory of the $string\_predicates$ module was too large for our theorem prover, the maximal number of auxiliary propositions was exceeded and the Saturate System would terminate without finding the proof.

Because of that, we have included four axioms only instead of the complete theory. It is logically right to use only some subset of a theory rather than the whole theory since if $T' \subset T$ and $T' \models F$ then also $T \models F$. In some other theorem prover one could try to include the entire $string\_predicates$ module and then try to prove the formula (6.1). Depending on the system's resources it could lead to a result.

6.2.1 The General Induction Theorem

Figure 6.5, similarly to figure 6.3, represents the proof scheme for the formula (6.3). Before we go deeper into the proof scheme, let us give some intuition behind that formula. Our main goal was to prove theorem (6.2) and we tried to prove it by induction. We applied the typical scheme used in the case of an implication (figure 6.4).

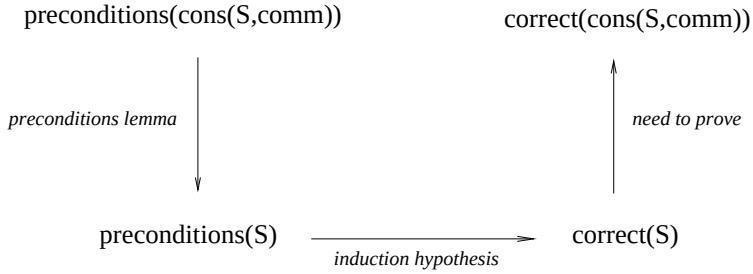

Figure 6.4: The scheme for an induction proof of the main theorem

Our inductive reasoning in the proof for formula (6.3) was the following: let us assume that $preconditions(cons(S, comm))$ holds. Then there are two possibilities: $preconditions(S)$ holds true or it does not. In the case if $preconditions(S)$ holds, we can apply the induction hypothesis and we can conclude that $correct(S)$ also holds, i.e $preconditions(S)$ and $correct(S)$ both hold. This then means that $both\_properties(S)$ holds and then in order to prove the main theorem, we only need to prove that $correct(cons(S, comm))$ holds. As a reminder, here is formula (6.3):

$$(\, both\_properties(S) \ \wedge \ preconditions(cons(S, comm)) \,) \Rightarrow correct(cons(S, comm))$$

The fact that the other possibility ($preconditions(S)$ does not hold) cannot happen was proved in the $preconditions\_lemma$.

We named formula (6.3) "induction steps" or "the general induction theorem", although it was not proved by induction; we have proved that it holds by doing a case analysis for every single $comm \in \Sigma$. The name "induction steps" is used because it actually represents the step of a structural induction. For $comm \in \{ins(E), del(E)\}$ the formula (6.3) was proved smoothly and without any interaction, but to prove it for $comm = dmin$ we had to use the previously proved *lemma1* (6.6) and *lemma2* (6.7) (cf. figure 6.5).

$$\begin{array}{l} preconditions(cons(S, dmin)) \Rightarrow \\ \quad \forall E \, \big(contains(queue(ex(S, empty)), E) \Rightarrow \\ \qquad elem(del\_min\_impl(queue(ex(S, empty)))) \leq E \big) \end{array} \tag{6.6}$$

$$\begin{aligned}\forall E\ \big(&contains(queue(ex(S, empty)), E) \Rightarrow \\ &elem(del\_min\_impl(queue(ex(S, empty)))) \leq E\big) \ \Rightarrow \\ &elem(del\_min(queue(ex(S, empty)))) = \\ &= elem(del\_min\_impl(queue(ex(S, empty))))\end{aligned} \tag{6.7}$$

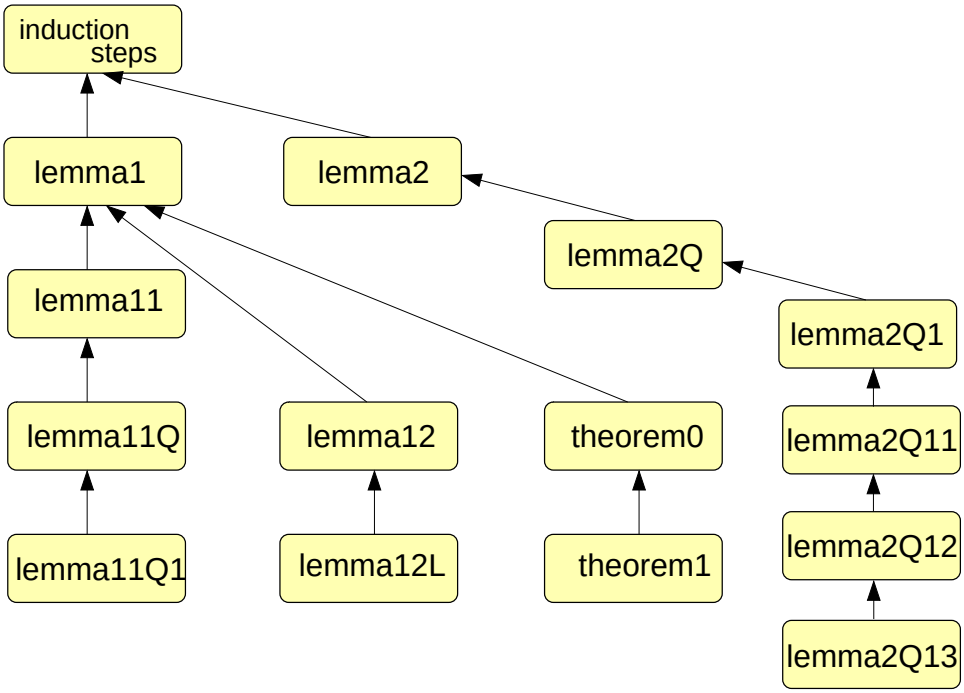

Figure 6.5: The scheme for the proof of the general induction theorem

Figure 6.5 also shows the main idea which we were using during the entire proving process. Usually our task was to prove that a given formula is a logical consequence of the module *string_predicates*. If we were not able to prove the formula directly, we tried to simulate the pen-and-paper proof. As a human being we can immediately conclude some facts which cannot be so directly concluded in machine reasoning. Those facts, which are usually some simpler formulas compared to the one we have to prove, are then inserted as valid formulas in the module containing the formula we want to justify. The inserted formulas also have to be proved and we repeat the process until we have found formulas which we are able to verify. Usually in that process we would move down from the *string_predicates* level to simpler levels: *queues*, *lb_lists*, ... If in the name of a lemma a letter Q occurs, then this lemma proves a formula which is a logical consequence of the module *queues*. If the letter L occurs, we are dealing with *lb_lists* module.

Let us demonstrate the described technique with the examples. *lemma2* (the formula (6.7)) was proved using the formula (6.8) (*lemma2Q*), which holds true for every priority queue Q. The formula (6.8) is a logical consequence of the module *queues*.

$$\begin{array}{l}\forall E\ \big(contains(Q,E) \Rightarrow elem(del\_min\_impl(Q)) \leq E\big) \\ \Rightarrow elem(del\_min(Q)) = elem(del\_min\_impl(Q)\end{array} \tag{6.8}$$

The formula (6.6) (*lemmma1*) was proved with the help of three new theorems (*lemma11* (6.9), *lemma12* (6.10) and *theorem0*).

$$\begin{array}{l}\forall E\ \big(contains(queue(ex(S,empty)),E) \Rightarrow \\ \quad (contains(queue(ex(cons(S,dmin),empty)),E) \vee \\ \quad\quad elem(del\_min\_impl(queue(ex(S,empty)))) = E)\big)\end{array} \tag{6.9}$$

$$\begin{array}{l}\forall E_1 \forall E_2\ \ \big(includes(ex1(cons(S,dmin),nil),(E_1,E_2)) \Rightarrow \\ \quad\quad elem(del\_min\_impl(queue(ex(S,empty)))) \leq E_2\big)\end{array} \tag{6.10}$$

lemma11 and *lemma12* were proved analogously to *lemma2*, by proving some simpler formulas from smaller modules, while *theorem0* is just a corollary of *theorem1*, which is described in paragraph 6.2.3. *lemma11* was verified using the corresponding formula (6.11) from the module *queues* and for *lemma12* we used the similar formula (6.12) from *lb_lists* module.

$$\begin{array}{l}\forall E\ \big(contains(Q,E)\ \ \Rightarrow\ \ (contains(queue(del\_min\_impl(Q)),E)\ \vee \\ \quad\quad\quad\quad elem(del\_min\_impl(Q)) = E)\big)\end{array} \tag{6.11}$$

$$\forall E\ \forall E_1\ \forall E_2\ \big(includes(adjust(E,L),(E_1,E_2)) \Rightarrow E \leq E_2\big) \tag{6.12}$$

Proving formulas in *queues* or *lb_lists* modules is much simpler than proving formulas in the *string_predicates* module. First of all, *queues* and *lb_lists* are less complex than *string_predicates* so there is a smaller chance that we will have to reduce the size of the module. The other reason which is also of practical nature, is structural induction. To prove that some formula holds for every string S we need to prove that it holds for $S = nil$ and we have to consider the induction step for every $comm \in \Sigma$. In order to prove that some formula holds for every priority queue Q, we have to confirm that it holds for $Q = empty$ and the induction step consists of only one case. The same reasoning holds for the module *lb_lists*.

6.2.2 Preconditions Lemma

The *preconditions_lemma* was the last theorem we have proved, probably because it looked as a fairly simple theorem to prove. $preconditions(S)$ symbolizes

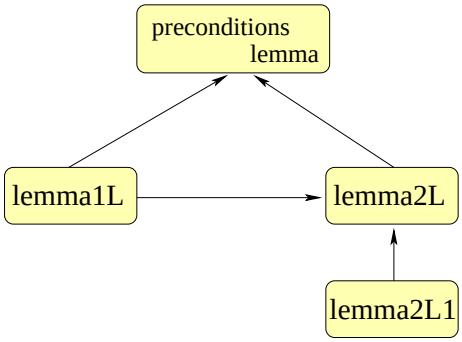

Figure 6.6: The proof scheme of the preconditions lemma

that whenever we were deleting an element E from the queue built out of S, E was greater or equal than its lower bound and all elements that are left in the queue were also greater or equal to their lower bound. $preconditions\_lemma$ was used to finalize the reasoning in the induction step of the theorem 6.1.1. It states that

$$preconditions(cons(S, comm)) \Rightarrow preconditions(S)$$

The proof for the $preconditions\_lemma$ was again done by a case analysis for every $comm \in \Sigma$. The proof scheme is not very complicated (cf. figure 6.6). For $comm = ins(E)$ the proof was done smoothly and without the need to include some already proved theorems, while for $comm \in \{del(E), dmin\}$ we used $lemma1L$ (the formula (6.13)) and $lemma2L$ (the formula (6.14)).

$$\begin{aligned} &\forall E_1\ \forall E_2\ \big(includes(L, (E_1, E_2)) \Rightarrow \\ &\qquad \forall E\ (includes(remove(E, L), (E_1, E_2))\ \vee\ E = E_1)\big) \end{aligned} \tag{6.13}$$

$$\begin{aligned} &\forall E_1\ \forall E_2\ \big(includes(L, (E_1, E_2)) \Rightarrow\ \forall E \\ &\quad (\exists E_3\ (includes(adjust(E, remove(E, L)), (E_1, E_3)) \wedge E_2 \leq E_3) \\ &\quad \vee\ E = E_1)\big) \end{aligned} \tag{6.14}$$

To verify $lemma2L$ we used $lemma1L$ and $lemma2L1$. Both, $lemma2L1$ and $lemma1L$ were proved directly, by structural induction. In order to prove that some $lemmaL$ holds true for every list L, first we have to prove that $lemmaL(nil)$ holds true and then we have to prove that the following formula is also valid:

$$\forall L\ \forall E_1\ \forall E_2\big(lemmaL(L) \Rightarrow lemmaL(cons(L, (E_1, E_2)))\big).$$

6.2.3 Theorem1

theorem1 is probably the most complex theorem we have proved in this thesis. The only purpose of *theorem1* consists in proving *theorem0*. Actually, we tried initially to prove *theorem0* only, but we did not succeed. The reason for our failure lies in the fact that queues and lists are modeled as multisets, so clearly, we also had to introduce natural numbers in order to count the number of occurrences of some element.

Let us remind that *theorem0* states that for every element contained in an original priority queue, that element is also a member of a checked priority queue and vice versa.

$$contains(queue(ex(S, empty)), E) \Leftrightarrow \exists E_1\ includes(ex1(S, nil), (E, E_1))$$

In our attempt of proving *theorem0* we were descending from the initial formula to simpler ones in the manner described before and at one point the only assumption that was left to verify was that formula (6.15) holds true for every priority queue Q and for every list L:

$$\begin{aligned} &\forall E(contains(Q, E) \Leftrightarrow includes(L, E)) \Rightarrow \\ &\quad \forall E_1 \forall E_2(contains(delete(Q, E_1), E_2) \Leftrightarrow includes(remove(E_1, L), E_2)) \end{aligned} \tag{6.15}$$

Although it looks like a correct formula at first sight, we must not forget that formula (6.15) should hold true for every Q and for every L. Let Q consist of two copies of an element E and let L only have one copy of the same element E. Then the premise of formula (6.15) evidently holds true, but the conclusion does not (after deleting E, L is an empty list, while Q still contains the element).

Since queues and lists are modeled as multisets, sooner or later we would faced a problem similar to the one with formula (6.15). In the end we had to introduce natural numbers.

In the *theorem0* none of the predicates introduced in the *string_predicates* module did occur, so the module $ex1$ should already be sufficient to verify *theorem0*. But $ex1$ does not cover the counting of element occurrences and we had to enrich our specification which we were using earlier.

theorem1, which is more general than *theorem0*, should be a logical consequence of the module $ex1$ and the module that describes the counting of element occurrences.

6.2.4 Theorem1: specification of natural numbers

In order to describe figure 6.7 we will start with the specification for natural numbers (module *nat_numbers*). The complete specification for natural numbers, including comparison and some simple operations (+,-,*,div, mod...) can be found in [20] but here it would be overkill to include all that in our module. We need natural numbers only for counting. In the module *nat_numbers* we only have introduced the definition and the equality of natural numbers and

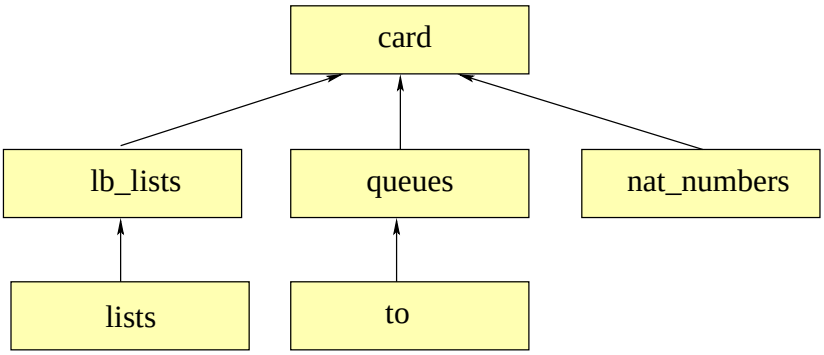

Figure 6.7: The hierarchy of modules used for the proof of the theorem1

some simple axioms. The definition is again inductive, similar to the previous definitions of priority queues and lower bound lists. Constructors for natural numbers are 0 and the function symbol s. s stands for successor.

---

functions:

$0 : number$
$p : number \rightarrow number$
$s : number \rightarrow number$

axioms:

$p(s(N)) = N$
$s(p(N)) = N$

$N \neq s(N)$

$N_1 = N_2 \Leftrightarrow s(N_1) = s(N_2)$

Figure 6.8: *nat_numbers* module

6.2.5 Theorem1: specification of the multiplicity

The module *card* is used for counting occurrences of some element: $cardQ(Q, E)$ represents the multiplicity of the element E in the priority queue Q, while $cardL(L, E)$ represents the multiplicity of the element E in the list L. The first two sets of axioms in this module describe how to calculate $cardQ$ and $cardL$, while the last set states that $cardQ$ and $cardL$ cannot be negative numbers. Actually, we formulated only that they cannot be -1, but it was already sufficient for proving all properties we needed of multiplicities of elements.

functions:

$cardQ : Queues \times Element \rightarrow number$

$cardL : Lists \times Element \rightarrow number$

axioms:

$cardQ(empty, E) = 0$

$cardQ(insert(Q, E), E) = s(cardQ(Q, E))$

$E_1 \neq E_2 \Rightarrow cardQ(insert(Q, E_1), E_2) = cardQ(Q, E_2)$

$cardL(nil, E) = 0$

$cardL(cons(L, lb(E, E1)), E) = s(cardL(L, E))$

$E_1 \neq E_3 \Rightarrow cardL(cons(L, lb(E1, E2)), E3) = cardL(L, E3)$

$s(cardL(L, E)) \neq 0$

$s(cardQ(Q, E)) \neq 0$

Figure 6.9: *nat_numbers* module

6.2.6 Theorem1: the formalization and the proof

At this point, the specification and the axioms we have developed are expressive enough for formalizing and proving *theorem1*. *theorem1* should be a logical consequence of modules $ex1$ and *card* and its immediate consequence should be *theorem0*. In *theorem1* we do not only claim that the element contained in the queue is also contained in the checked priority queue, but we also claim that the number of occurrences of any element in the priority queue is equal to the number of occurrences in the checked priority queue.

$$\forall E\big(\ contains(queue(ex(S, empty)), E) \Leftrightarrow \exists E_1, includes(ex1(S, nil), (E, E_1))$$
$$\wedge\ \ cardQ(queue(ex(S, empty)), E) = cardL(ex1(S, nil), E)\big) \tag{6.16}$$

Notice that *theorem0* is enclosed in *theorem1* and with having *theorem1* as a valid theorem, it is easy to confirm the validity of *theorem0*. One can see that the proof of validity of formula (6.16) completes the proof of theorem 6.1.1 (The Correctness Theorem). As we have already mentioned, *theorem1* was probably the most complicated theorem we have proved and figure 6.10 represents only one part of the proof, while figure 6.11 represents the second part of the proof.

We have verified the validity of *theorem1* by structural induction. The complete formalization of every theorem and every lemma used in the proof can be read in [27], but here we are going to explain all the main ideas that we were using in the proof process.

A new predicate symbol $theorem1$ was introduced as an abbreviation for: $theorem1(S)$ holds true iff formula (6.16) holds for S. Therefore, we had to ver-

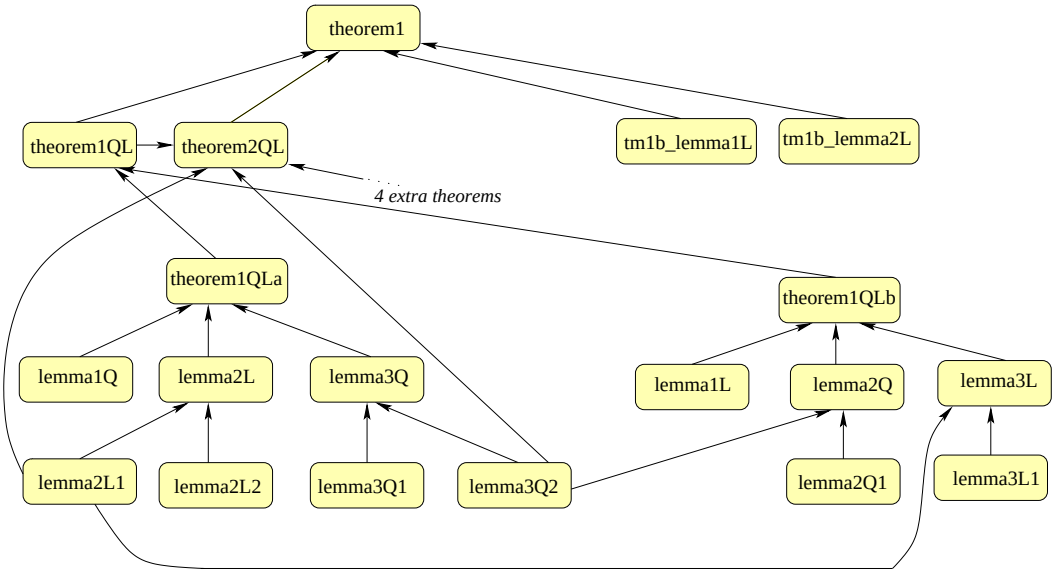

Figure 6.10: The proof scheme for theorem1

ify that $theorem1(S)$ holds true for every $S \in \Sigma^*$. The base case, $theorem1(nil)$ holds, was proved smoothly and easily, it was an immediate consequence of our input axioms. The induction step was more complicated: we had to prove that the following three formulas hold true for every $S \in \Sigma^*$:

$$theorem1(S) \quad \Rightarrow \quad \forall E \;\; theorem1\big(cons(S, ins(E))\big) \tag{6.17}$$

$$theorem1(S) \quad \Rightarrow \quad \forall E \;\; theorem1\big(cons(S, del(E))\big) \tag{6.18}$$

$$theorem1(S) \quad \Rightarrow \quad theorem1\big(cons(S, dmin)\big) \tag{6.19}$$

Formula (6.17) was again proved smoothly, although the theorem prover was using here a little bit more reasoning than in the base case for $S = nil$.

On the other hand in the proof of formula (6.19) we had to include several simpler lemmas as valid facts: $tm1b\_lemma1L$ (formula (6.20)) and $tm1b\_lemma2L$ (formula (6.21)). Those lemmas confirm that during the operation *adjust* performed on the lower bound list, no element was erased from the list: every element which was in the list before the *adjust* command was performed, stays in the list and its multiplicity also did not change.

$$\forall E \forall E_1 \Big(\exists E_2 \; includes(L, (E, E_2))) \Leftrightarrow \exists E_3 \; includes(adjust(E_1, L), (E, E3)) \Big) \tag{6.20}$$

$$\forall E \forall E_1 \;\; cardL(L, E) = cardL(adjust(E_1, L), E)) \tag{6.21}$$

Not only that we had to include those two lemmas to get the proof of formula (6.19), but we also have included formula (6.18) as a valid formula. We can prove formula (6.16) in that way, since in the proof of formula (6.18) we did not use formula (6.19) and we do not have a cycle. With those three formulas included and using the fact from the priority queue specification that *del_min_impl* operation deletes some element from the queue, the theorem prover has found the proof for the formula (6.19). Both lemmas, $tm1b\_lemma1L$ and $tm1b\_lemma2L$, were verified directly using structural induction.

At this point only formula (6.18) is left to be verified. As we did not succeed in the direct proof, we have assumed the following formulas are valid:

$$\begin{aligned} &\forall E\ \big(contains(Q,E) \Leftrightarrow \exists E_1\ includes(L,(E,E_1)) \\ &\quad \wedge\ cardQ(Q,E) = cardL(L,E)\big) \Rightarrow \\ &\forall E\ \forall E_1\ \big(contains(delete(Q,E_1),E) \Leftrightarrow \exists E_2\ includes(remove(E_1,L),(E,E_2))\big) \end{aligned} \tag{6.22}$$

$$\begin{aligned} &\forall E\ \big(contains(Q,E) \Leftrightarrow \exists E_1\ includes(L,(E,E_1)) \\ &\quad \wedge\ cardQ(Q,E) = cardL(L,E)\big) \Rightarrow \\ &\forall E\ \forall E_1\ \big(cardQ(delete(Q,E_1),E) = cardL(remove(E_1,L),E)\big) \end{aligned} \tag{6.23}$$

One closer look at the above formulas shows that although they characterize the behaviour of queues and lists, they are actually verifying formula (6.18), only we have moved from $ex1$ module to *card* module. Unfortunately, this is again too large for our theorem prover and once again we have to choose formulas from the whole theory that are sufficient for the proof of formula (6.18). In order to prove formula (6.18), having statements of formulas (6.22) and (6.23), we only necessitate axioms that define the execution of a priority queue command $del(E)$:

$$\begin{aligned} ex1(cons(S,del(E)),L) &= remove(E,ex1(S,L)) \\ queue(ex(cons(S,del(E)),Q)) &= delete(queue(ex(S,Q)),E) \end{aligned}$$

These two axioms, together with formulas (6.22) and (6.23) included as valid, entail formula (6.18). Of course, in some other theorem provers one might not need to reduce the size of the initial theory, it depends all on system resources and abilities of the theorem prover.

Formula (6.22), which we called $theorem1QL$ (cf. figure 6.10), states the following: let Q be a priority queue and let L be a list and let every element contained in Q be also contained in L with the same multiplicity and vice versa.Then, after removing any element from Q and removing the same element from L, every element contained in Q is also contained in L and vice versa. Note that here we do not count occurrences of elements after deletion: it is done in $theorem2QL$ (formula (6.23)). At this point we have descended from the top most level of figure 6.10 to the next level and $theorem1QL$ and $theorem1QL$ are left to be proved.

We noticed that in the conclusion part of $theorem1QL$ there is an equivalence symbol, so in order to prove it, we introduced two simpler formulas: $theorem1QLa$ and $theorem1QLb$ (formulas (6.24) and (6.25)) and we replaced an equivalence symbol by two implication symbols.

$$\begin{aligned}&\forall E\,\big(contains(Q,E) \Leftrightarrow \exists E_1\; includes(L,(E,E_1))\\ &\qquad \wedge\, cardQ(Q,E) = cardL(L,E)\big) \Rightarrow\end{aligned} \tag{6.24}$$

$$\forall E\,\forall E_1\,\big(contains(delete(Q,E_1),E) \Rightarrow \exists E_2\; includes(remove(E_1,L),(E,E_2))\big)$$

$$\begin{aligned}&\forall E\,\big(contains(Q,E) \Leftrightarrow \exists E_1\; includes(L,(E,E_1))\\ &\qquad \wedge\, cardQ(Q,E) = cardL(L,E)\big) \Rightarrow\end{aligned} \tag{6.25}$$

$$\forall E\,\forall E_1\,\big(\exists E_2\; includes(remove(E_1,L),(E,E_2)) \Rightarrow contains(delete(Q,E_1),E)\big)$$

With these two formulas we gain a nice and simple proof of $theorem1QL$, only we got two new theorems whose validity has to be checked and verified. But, $theorem1QLa$ and $theorem1QLb$ are fairly similar and their only difference is that they have switched places of queues and lists. This means that if we find a proof for one of them, we can only exchange places of queues and lists and the proof for the other theorem is also found. For this reason we shall only explain the proof procedure for $theorem1QLa$.

$theorem1QLa$ was proved using three simple formulas: $lemma1Q$, $lemma2L$ and $lemma3Q$. Let Q be some priority queue and let Q' be the priority queue which is the result of deleting some element E from Q. Then, every element contained in Q' was also contained in Q. This simple fact was proved in $lemma1Q$ (formula (6.26)).

$$\forall E_1 \forall E_2\,\big(contains(delete(Q,E_2),E_1) \Rightarrow contains(Q,E_1)\big) \tag{6.26}$$

$lemma2L$ (formula (6.27)) is the lemma that describes one simple property of lists. Let L be some list and let L' be the list that is the result of removing some element E from L. If L' does not contain the element E_1, then E_1 was also not contained in L or E_1 is exactly the element we have deleted from L and it was contained in L in the only one copy.

$$\begin{aligned}&\forall E_1 \forall E_2 \big(\forall E_3\; \neg includes(remove(E_2,L),(E_1,E_3)) \Rightarrow\\ &\quad \forall E_3 \neg includes(L,(E_1,E_3)) \;\vee\; (E_2 = E_1 \wedge cardL(L,E_1) = s(0))\big)\end{aligned} \tag{6.27}$$

The last lemma we have used for the proof of $theorem1QLa$ was $lemma3Q$ (formula (6.28)). It states a simple fact about queues: if a priority queue Q contains exactly one copy of an element E, then after deleting E from Q the new queue does not contain E.

$$\forall E\ \big((contains(Q,E) \wedge cardQ(Q,E) = s(0)) \Rightarrow \neg contains(delete(Q,E),E)\big) \tag{6.28}$$

Those three lemmas were proved almost directly (cf. figure 6.10 and [27]), either with the help of even simpler lemmas or by induction. It completes the proof of $theorem1QLa$. Rewriting those formulas by replacing queues and lists, we also get the proof for $theorem1QLb$.

Probably now it is the right moment to explain how one can find the suitable simpler formulas that are needed for completion of the proof of the original formula.

Initially we always try to verify the given formula, but it might happen that the theorem prover cannot find the proof. Sometimes the run of the prover does not terminate. Although it is a bit tricky to claim that it does not terminate, because we can only claim for sure that it did not terminate in some finite time, we cannot state accurately that it will never terminate. But, in practice if the run takes too long, usually we conclude that it does not terminate. Again, it is hard to define what it is "too long". The case of an infinite run is treated in the same way as the case of termination without finding a proof.

Sometimes the run terminates abnormally due to the insufficiency of system resources. In that case, we try to find axioms that are sufficient for the proof of the formula and only include them instead of the whole module. We have seen this case in most of our proofs.

On the other hand the run can terminate without finding a proof. In that case we have to concentrate on the output of theorem prover and construct a model in which our initial formula is interpreted as false. During that phase we can detect gaps in the axiomatization which cause that the given formula is not entailed by the set of axioms. That problem can be easily solved by adding a new axiom to the set of already existing axioms and then we try to verify that the given formula is entailed by the new set of axioms. A typical omission of axioms in the specification were axioms for the base cases. For example, we have specified axioms for the behaviour of the del_min command, but we have not included the axiom for $del\_min(empty)$.

Another thing that we have to keep in our mind is that the theorem prover just cannot conclude some facts which human beings could during a pen-and-paper proof. Unfortunately, those facts are usually essential for finding a proof and in that case we just add those facts as valid formulas.

In this section we will demonstrate by an example how one can find simpler formulas needed for the proof. We consider $theorem1QLa$ (formula (6.24)). In the previous section we have introduced simpler formulas and here we are going to describe how did we find them.

The attempt of verifying $theorem1QLa$ by induction failed. We know that the theorem prover verifies that some set of axioms S entails formula F, $S \models F$, by checking whether $S \cup \neg F$ is unsatisfiable. Therefore in the first place we

have to negate formula (6.24). Formula (6.24) is an implication: by negating it, we can conclude that the premise holds true (formula (6.29)), whereas the conclusion is false (i.e. formula (6.30) holds true).

$$\forall E\big(contains(Q,E) \Leftrightarrow \exists E_1\ includes(L,(E,E_1)) \wedge cardQ(Q,E) = cardL(L,E)\big) \tag{6.29}$$

$$\exists E\ \exists E_1\big(contains(delete(Q,E_1),E) \wedge \forall E_2\ \neg includes(remove(E_1,L),(E,E_2))\big) \tag{6.30}$$

From the atom $contains(delete(Q,E_1),E)$ we cannot directly, as a consequence of the axiom set, conclude anything since we know nothing about the structure of Q. But we need to raise the contradiction so we will try to contradict formula (6.29). We could try to establish the link between $contains(Q,E)$ and $contains(delete(Q,E_1),E)$ and after we could do the similar thing with lists. Therefore, we came up with $lemma1Q$ and we derived that $contains(Q,E)$ holds.

The formula $\forall E_2\ \neg includes(remove(E_1,L),(E,E_2))$, the second part of formula (6.30), is also not useful for finding a contradiction with the current set of axioms, since here again we cannot determine any new formulas. But once again we use our human reasoning and we formulate $lemma2L$: either $\forall E_2\ \neg includes(L,(E,E_2))$ holds or $E = E_1\ \wedge\ cardL(L,E_1) = s(0)$. The first possibility $\forall E_2\ \neg includes(L,(E,E_2))$ together with $lemma1Q$ clearly contradicts the formula (6.29) and we conclude that only the second possibility is true: $E = E_1\ \wedge\ cardL(L,E_1) = s(0)$.

Let us see what we have gained from the negation of $theorem1QLa$. If we include $lemma1Q$ and $lemma2L$ as valid formulas, the theorem prover can conclude that there exists an elements E such that $contains(delete(Q,E),E)$ and $contains(Q,E)$ and $cardQ(Q,E) = s(0)$. Immediately we see that those three facts are contradictory, but still the theorem prover cannot derive $\bot$. But after the inclusion of $lemma3Q$ as a valid formula, clearly we were able to derive $\bot$.

If we return to figure 6.10, we can see that the only theorem left to verify is $theorem2QL$. $theorem2QL$ is used in the proof of *theorem1*, namely in the induction step for the case of the command $del(E)$. While $theorem1QL$ only implies what happens with membership in the queue/list after the deletion of an element, $theorem2QL$ states what effects the deletion of an element has on multiplicities of elements. Just as a reminder, $theorem2QL$ says:

$$\begin{aligned}&\forall E\ \big(contains(Q,E) \Leftrightarrow \exists E_1\ includes(L,(E,E_1))\\ &\qquad \wedge\ cardQ(Q,E) = cardL(L,E)\big) \Rightarrow\\ &\forall E\ \forall E_1\ \big(cardQ(delete(Q,E_1),E) = cardL(remove(E_1,L),E)\big)\end{aligned}$$

One should notice that with the proof of $theorem2QL$ the whole proof of

theorem1 is finished. Since $theorem2QL$ is a fairly complex formula, we did not manage again to prove it without auxiliary lemmas. The proof scheme for $theorem2QL$ is shown on figure 6.11.

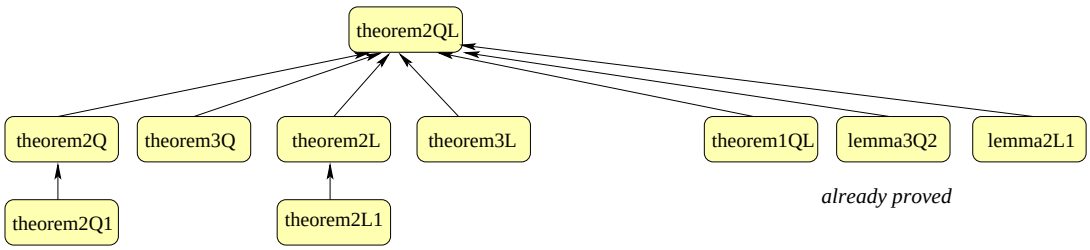

Figure 6.11: The proof scheme for theorem2QL (*theorem1*)

In the proof process of $theorem2QL$, after we have noticed that we have to use auxiliary lemmas, we focused on the formulation of $theorem2QL$ in order to find some simpler formulas that can be used as a valid input. $theorem2QL$ is an implication and again we negate it. From the negation of its conclusion we could not derive any new knowledge, so we tried to prove it in a roundabout way.

The premise of $theorem2QL$ holds true and after applying $theorem1QL$, we derive the following fact:

$$\forall E\, \forall E_1\, \big(contains(delete(Q, E_1), E) \Leftrightarrow \exists E_2\; includes(remove(E_1, L), (E, E_2))\big)$$

In order to prove $theorem2QL$, we try to establish a link between that fact and the multiplicity of elements. If the element is not contained in the queue/list then its multiplicity in that queue/list is 0. This simple observation was proved in $lemma3Q2$ and $lemma2L1$ (formulas (6.31) and (6.32)).

$$\forall E\big(cardQ(Q, E) = 0 \Leftrightarrow \neg contains(Q, E)\big) \tag{6.31}$$

$$\forall E\big(cardL(L, E) = 0 \Leftrightarrow \forall E_1\, \neg includes(L, (E, E_1))\big) \tag{6.32}$$

Since with these two lemmas we have proved $theorem2QL$ for the case when an element is not contained in the queue/list after the deletion, from now on, we shall consider only the case when the element is contained. Let Q be some priority queue, let E_1 be some element and let Q' be the queue which results from deleting E_1 from Q. Then for every element $E \in Q'$, either the multiplicity of E in Q' did not change with respect to the multiplicity of E in Q, or E was exactly the element deleted from Q and in that case the multiplicity of E in Q' was decreased by 1. $theorem2Q$ (formula (6.33)) and $theorem3Q$ (formula (6.34)) were used to express those facts:

$$\forall E \forall E_1 \big(contains(delete(Q, E_1), E) \Rightarrow \\ (cardQ(delete(Q, E_1), E) = cardQ(Q, E) \ \vee \ E = E_1)\big) \tag{6.33}$$

$$\forall E \big(contains(delete(Q, E), E) \Rightarrow \\ cardQ(delete(Q, E), E) = p(cardQ(Q, E))\big) \tag{6.34}$$

Similar formulas ((6.35) and (6.36)) were proved about lists.

$$\forall E \forall E_1 \big(\exists E_2 \ includes(remove(E_1, L), (E, E_2)) \Rightarrow \\ (cardL(remove(E_1, L), E) = cardL(L, E) \ \vee \ E = E_1)\big) \tag{6.35}$$

$$\forall E \big(\exists E_1 \ includes(remove(E, L), (E, E1)) \Rightarrow \\ cardL(remove(E, L), E) = p(cardL(L, E))\big) \tag{6.36}$$

All those formulas were sufficient for the theorem prover to verify $theorem2QL$ after we have included them into the $theorem2QL$ module. Their correctness was verified either directly, by induction, or they were proved by introducing even simpler lemmas.

At this point we have finished completely the proof of the initial theorem which verifies the correctness of the priority queue checker:

$$(complete(S) \ \wedge \ correct(S)) \Rightarrow safe(S)$$

All the missing lemmas and more detailed proofs can be found in [27].

6.3 Completeness of definitions

When we were defining priority queues, *empty* was defined as a priority queue, and for every already defined priority queue Q and for every $E \in Element$ $insert(Q, E)$ was also defined as a priority queue. This definition is also considered as a complete definition in the sense that we restrict the priority queues that we have investigated only on those priority queues that have the "empty/insert" form. This means that every priority queue Q for us is either $Q = empty$ or there exists a priority queue Q' and an element E such that $Q = insert(Q', E)$. Again recursively, Q' has to have the same "empty/insert" form. With this restriction on the priority queues which we study, we gain an inductive definition of priority queues, which helps us in the proofs of all the lemmas about priority queues. Because of the inductive definition, we can use structural induction.

In the specification we were using *empty* and *insert* as constructors, i.e. the behaviour of every priority queue operation was described through *empty* and *insert*. In order to apply structural induction for proving priority queue properties, we have to verify first that the result of the priority queue operations *delete*, *del_min* and *del_min_impl* performed on priority queues of the "empty/insert"

form is again a priority queue of the same form. Without verifying that fact the process of proving priority queue lemmas gets more complex, since we have to verify the lemma then not only for *insert*, but also for all other priority queue operations.

Let $form$ be a predicate that describes that the priority queue Q has the "empty/insert" form. Notice that if $form(Q)$ holds, then $form(insert(Q, E))$ also holds for every $E \in Element$.

Theorem 6.3.1. (Completeness of the *delete* operation) *Let Q be a priority queue such that* form(Q) *holds. Then, also* form(delete(Q,E)) *holds for every $E \in Element$.*

Proof. The axioms for the *delete* operation are:

$$delete(empty, E) = empty$$
$$delete(insert(Q, E), E) = Q$$
$$E_1 \neq E_2 \Rightarrow delete(insert(Q, E_1), E_2) = insert(delete(Q, E_2), E_1)$$

We shall prove theorem 6.3.1 by structural induction. If $Q = empty$, then $delete(Q, E) = empty$ and *form(delete(Q,E))* holds. Let us assume that *form(Q)* and *form(delete(Q,E))* hold for every E and let $Q_1 = insert(Q, E_1)$. We need to verify that *form(delete(Q_1, E_2))* also holds for every E_2. If $E_1 = E_2$, we apply the second axiom and then clearly *form(delete(Q_1, E_2))* holds. If $E_1 \neq E_2$, we apply the last axiom first and after that the induction hypothesis. The remark about $form(insert(Q, E))$ confirms that *form(delete(Q_1, E_2))* holds true.

□

Theorem 6.3.2. (Completeness of the *del_min* operation) *Let Q be a priority queue such that* form(Q) *holds. Then, also* form(queue(del_min(Q)) *holds.*

Proof. The proof is just a copy of the proof of theorem 6.3.1: we use structural induction and axioms of the *del_min* operation. □

Theorem 6.3.3. (Completeness of the operation *del_min_impl*) *Let Q be a priority queue such that* form(Q) *holds true. Then,* form(queue(del_min_impl(Q)) *also holds.*

Proof. The specification of the operation *del_min_impl* contains the following axiom

$$queue(del\_min\_impl(Q)) = delete(Q, elem(del\_min\_impl(Q)))$$

Theorem 6.3.1 confirms completeness of the *delete* operation. □

Lists are defined in the same way as queues, only constructors have different names here: *nil* and *cons*. They behave in the same way as *empty* and *insert*. Also, all operations on lists are expressed through these two constructors. Let us recall that we were using *cons* in the same way as *insert*: instead of conventionally putting the element at the beginning of the list, our *cons* appends the

element to the end. Therefore, proving that the result of every list operation on a list of the form "nil/cons" is again a list of the form "nil/cons", is identical to proving these properties in the case of queues.

The list operations we still have to investigate are *remove* and *adjust*. The list operation *remove* has analogous axioms to the queue operation *delete*:

$$remove(E, nil) = nil$$
$$remove(E1, cons(L, (E1, E2))) = L$$
$$E1 \neq E2 \Rightarrow remove(E1, cons(L, (E2, E3))) = cons(remove(E1, L), (E2, E3))$$

Inductive reasoning, similar to the one in the case of queues, verifies that the result of the *remove* operation performed on the list of the form "nil/cons" is again a list of the same form.

The axioms for the list operation *adjust* confirm that in order to prove the same statement for the operation *adjust*, we can apply the following inductive reasoning:

$$adjust(E, nil) = nil$$
$$E3 < E1 \Rightarrow adjust(E1, cons(L, (E2, E3))) = cons(adjust(E1, L), (E2, E1))$$
$$E1 \leq E3 \Rightarrow adjust(E1, cons(L, (E2, E3))) = cons(adjust(E1, L), (E2, E3))$$

We have to use here one more argument, namely the totality of $\leq$.

6.4 The "create" command

The specification presented in this chapter is not the same specification and the same model that we have initially developed. Initially, the alphabet Σ also contained the letter create.

$$\Sigma = \{\mathsf{create}, \mathsf{ins}(e), \mathsf{del}(e), \mathsf{dmin} \mid e \in E\}.$$

In [27] there are two models available for download. One consists in the model we have presented in this chapter, whereas the other one is the older version which includes the create command. Although those two models are almost the same, still there are some differences. In this paragraph we will describe two main differences in the specification of those two models.

Our first assumption was that every code fragment has to start with a create command. create is used to build a new instance of *empty*. This model was more appropriate for real world examples, since every new instance of a priority queue is initialized with the command `p_queue <P,I> Q;` where P is the data type of the priority part and I is the data type of the item part of the priority queue. As create was an element of the alphabet Σ, the first big difference was that the specification for the functions ex and $ex1$ had to change.

In the model presented earlier in this chapter, we assumed that code execution starts with an empty priority queue, which changes then according to the program's code. Since create was an element in the alphabet, we have to assume in the former model that independently with which priority queue we start, the create command creates an empty queue. The old specification for the functions ex and $ex1$ was as follows:

$$\begin{array}{ll} ex(nil, Q) = (Q, ok) & ex1(nil, L) = L \\ ex(cons(S, create), Q) = (empty, ok) & ex(cons(S, create), L) = nil \\ \ldots & \ldots \end{array}$$

The other commands did not change. Although initially this model was more suitable for real world problems, we faced another difficulty: the command create could now occur on any place in a code sequence, not necessarily only at the beginning. The execution functions ex and $ex1$ treat that problem efficiently: if there are more create commands in the code sequence S, only the create command that occurred last in S creates a new empty queue; the whole priority queue built from the prefix that came before the last create command is simply erased.

The second big difference between the model with included create command and the model without it, lies in the predicate $error\_free$. Let us recall that the predicate $error\_free$ denotes the fact that whenever we have accessed an element during code execution, this element was greater than or equal than its lower bound. The first attempt to specify $error\_free(cons(S, create))$ was to assume that it always holds. The reason for that is the fact that $ex(cons(S, create), Q) = (empty, ok)$, so since the whole priority queue $Q' = queue(ex(S, Q))$ is erased, one could expect that also then the possible errors of Q' will be erased. Unfortunately, this reasoning is not suitable for theorem proving since in such a case we cannot even prove that $error\_free(cons(S, comm)) \Rightarrow error\_free(S)$, which is essential for proving the $preconditions\_lemma$, which is again fundamental for the $main\_theorem$.

Thus, in a second attempt to specify $error\_free(cons(S, create))$ we have considered the axiom $error\_free(cons(S, create)) \Leftrightarrow error\_free(S)$. This was still too weak to prove $preconditions\_lemma$, although this rule keeps track of passed events. Let us consider the following code sequence S

$$S \;=\; \mathsf{create; ins(1); ins(2); dmin; create;}$$

and let us assume that the result returned by the dmin command is 2. Let us define $S_1 =$ create; ins(1); ins(2); dmin; Then, $S = cons(S_1, create)$. Note that $error\_free(S_1)$ holds true, so therefore $error\_free(S)$ also holds true. Since the empty priority queue does not contain any element, $preconditions(S)$ holds as well. But $preconditions(S_1)$ does not hold since the priority queue built out of S_1 contains 1 and its lower bound is 2. Thus $preconditions(cons(S_1, create))$ does not imply $preconditions(S)$, so S_1 violates the $preconditions\_lemma$ and our axiom for $error\_free(cons(S, create))$ was again not strong enough.

In order to be able to prove the $preconditions\_lemma$, we had to introduce

an even stronger axiom for $error\_free(cons(S, create))$, and the final axiom that we have tried was $error\_free(cons(S, create)) \Leftrightarrow preconditions(S)$ This axiom was sufficient to prove that $preconditions\_lemma$ also holds for the create command.

There are no more big differences between the model without the create command and the model with it. Although in the beginning we have proved the correctness theorem in the model with the create command, we decided at the end to remove it from the model. There are several reasons to do that: the specification becomes more natural and understandable and the proofs become shorter and faster. The model without the create command corresponds to the real world situations more than the model with the create command, since in the model without it, the create command cannot occur anywhere in a code sequence.

Even though in the end we have completely erased the create command, implicitly it is still there: we have always assumed that every code sequence starts with create which builds *empty*. The code sequence is then executed with *empty* forwarded as a parameter to the function ex.

Chapter 7

Conclusion and Future Work

The goal of this work was to two-fold: the first goal was to formally verify the algorithm used for the priority queue checker. The second goal was to determine what can be the role of saturation-based theorem proving in verification.

The first goal was fulfilled successfully; we managed to develop the specification and using this specification, we were able to prove the correctness theorem.

As for the second goal, we have learnt that saturation-based theorem proving is useful for verification, but still Saturate needed a lot of guidance. For example, in Figure 6.10, it can be seen that 22 lemmas were necessary in order to prove *theorem*1. The reason for such a big number of auxiliary lemmas lies in the inductive definition of priority queues and lists. Even fairly simple properties of priority queues could not be proved directly. In order to prove them we had to use even simpler lemmas.

Also, it is important to mention that one part of the automatic verification was done by hand: since Saturate is a first-order theorem prover and induction is not a first-order property, all lemmas and theorems had to be proved in two steps. First we verified that the lemma holds for the base case and then we had to prove validity of the induction step. The inductive definition causes that we cannot prove directly that some formula holds for every priority queue. We could do that only when we include as valid some simpler formulas, but at the end those simpler formulas had to be verified again in two steps.

One of the possible future directions of this work is to do the same verification in an interactive verification system, for example PVS [26], and compare the number of interaction needed. It would also be interesting to compare the Saturate system to other theorem provers, to see whether the initial assumption that the chaining calculus is the best strategy, is true.

Another direction of future research would be the generalization to other

data structures. The most natural candidate are *multi-dimensional priority queues* [12]:

Definition 7.0.1. *Let $(P, <_i)$ be a totally ordered set for $i \in \{1, \ldots, n\}$, and let I be any set. A multi-dimensional priority queue over I using $(P, <_1, \ldots <_n)$ is a datatype that supports the following operations:*

- create*:* *creates empty priority queue*
- insert (i,p)*:* *inserts individual $i \in I$ with priority $p \in P$*
- delete_min(j)*:* *removes an object (i, p) which has a minimal p with respect to $<_j$, $j \in \{1, \ldots, n\}$*
- find_min(j)*:* *returns a pair (i, p) which has a minimal p with respect to $<_j$, $j \in \{1, \ldots, n\}$*
- contains(i,p)*:* *true iff the priority queue contains $i \in I$ with associated priority $p \in P$*

Briefly, in this case the checker would be implemented using several lower bounds, for each of orders $<_i$ there should be a new lower bound defined in the same manner as for standard priority queues. Thus, in the system of lower bounds every element $e \in I \times P$ would be represented with the tuple $(e, lb_1, \ldots, lb_n) \in (I \times P) \times P \times \ldots \times P$. The priority lb_i is representing the lower bound of the element e with respect to $<_i$.

Bibliography

[1] Nancy M. Amato and Michael C. Loui. Checking linked data structures. In *Proceedings of The 24th International Symposium on Fault Tolerant Computing (FTCS-24)*, pages 164–73, 1994.

[2] Leo Bachmair and Harald Ganzinger. Completion of first-order clauses with equality by strict superposition (extended abstract). In S. Kaplan and M. Okada, editors, *Conditional and Typed Rewriting Systems, 2nd International Workshop*, volume 516 of *Lecture Notes in Computer Science*, pages 162–180, 1991.

[3] Leo Bachmair and Harald Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *Journal of Logic and Computation*, 4(3):217–247, 1994.

[4] Leo Bachmair and Harald Ganzinger. Rewrite techniques for transitive relations. In *Proceedings of the 9th IEEE Symposium on Logic in Computer Science*, pages 384–393. IEEE Computer Society Press, 1994.

[5] Leo Bachmair and Harald Ganzinger. Ordered chaining calculi for first-order theories of transitive relations. *Journal of the ACM*, 45(6):1007–1049, 1998.

[6] Manuel Blum and Sampath Kannan. Designing programs that check their work. In *Proceedings of the 21st annual ACM symposium on Theory of computing*, pages 86–97. ACM Press, 1989.

[7] Manuel Blum, Michael Luby, and Ronitt Rubinfeld. Self-testing/correcting with applications to numerical problems. *Journal of Computer and System Sciences*, 47(3):549–595, 1993.

[8] Jonathan D. Bright. *Checking and Certifying Computational Results*. PhD thesis, John Hopkins University, 1994.

[9] Jonathan D. Bright and Gregory F. Sullivan. Checking mergeable priority queues. In *Digest of the 24th Symposium on Fault-Tolerant Computing*, pages 144–153. IEEE Computer Society Press, 1994.

[10] Jonathan D. Bright and Gregory F. Sullivan. On-line error monitoring for several data structures. In *Digest of the 25th Symposium on Fault-Tolerant Computing*, pages 392–401. IEEE Computer Society Press, 1995.

[11] Jonathan D. Bright, Gregory F. Sullivan, and Gerald M. Masson. A formally verified sorting certifier. *IEEE Transactions on Computers*, 46(12):1304–1312, December 1997.

[12] Yuzheng Ding and Mark A. Weiss. The k-d heap: An efficient multi-dimensional priority queue (extended abstract). In *Workshop on Algorithms and Data Structures*, volume 709 of *Lecture Notes in Computer Science*, pages 303–313, 1993.

[13] Jack Edmonds. Maximum matching and polyhedron with 0,1 - vertices. *Journal of Research of the National Bureau of Standards*, 69B:125–130, 1965.

[14] Jack Edmonds. Paths, trees and flowers. *Canadian Journal on Mathematics*, 17:449–467, 1965.

[15] Ulrich Finkler and Kurt Mehlhorn. Checking priority queues. In *Proceedings of the 10th annual ACM-SIAM symposium on Discrete algorithms (SODA'99)*, pages 901–902. Society for Industrial and Applied Mathematics, 1999.

[16] Kurt Mehlhorn and Stefan Näher. From algorithms to working programs on the use of program checking in leda. In L. Brim, J. Gruska, and J. Zlatuska, editors, *MFCS*, volume 1450 of *Lecture Notes in Computer Science*, pages 84–93, 1998.

[17] Kurt Mehlhorn and Stefan Näher. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

[18] Kurt Mehlhorn, Stefan Näher, Michael Seel, Raimund Seidel, Thomas Schilz, Stefan Schirra, and Christian Uhrig. Checking geometric programs or verification of geometric structures. *Computational Geometry*, 12(1-2):85–103, 1999.

[19] Pilar Nivela and Robert Nieuwenhuis. Practical results on the saturation of full first-order clauses: Experiments with the saturate system. (system description). In C. Kirchner, editor, *5th International Conference on Rewriting Techniques and Applications (RTA)*, volume 690 of *Lecture Notes in Computer Science*, 1993.

[20] Peter Padawitz. *Computing in Horn clause theories*. Springer-Verlag New York, 1988.

[21] Ronitt A. Rubinfeld. *A mathematical theory of self-checking, self-testing and self-correcting programs*. PhD thesis, University of California at Berkeley, 1991.

[22] Gregory F. Sullivan and Gerald M. Masson. Using certification trails to achieve software fault tolerance. In *Digest of the 20th Symposium on Fault-Tolerant Computing*, pages 423–431. IEEE Computer Society Press, 1990.

[23] Gregory F. Sullivan and Gerald M. Masson. Certification trails for data structures. In *Digest of the 21st Symposium on Fault-Tolerant Computing*, pages 240–247. IEEE Computer Society Press, 1991.

[24] Hal Wasserman and Manuel Blum. Software reliability via run-time result-checking. *Journal of the ACM*, 44(6):826–849, 1997.

[25] `http://www.mpi-sb.mpg.de/SATURATE/`.

[26] `http://pvs.csl.sri.com/`.

[27] `http://www.mpi-sb.mpg.de/~rpiskac/queues/`.