A Verification of Binary Search

Christoph Walther and Stephan Schweitzer *
Fachgebiet Programmiermethodik
Technische Universität Darmstadt

Abstract. We demonstrate the use of the \texttt{veriFun} system with a verification of the Binary Search method. We present the challenges encountered when working on this problem, illustrate the operation and performance of the system and finally discuss technical improvements as well as subjects of further research, which are not specific to our approach.

1 Introduction

We develop the \texttt{veriFun} system [1],[19], a semi-automated system for the verification of statements about programs written in a functional programming language. The motivation for this development is twofold: Since we are interested in methods for automating reasoning tasks which usually require the creativity of a human expert, we felt the need for having an experimental base of easy access which we can use to evaluate new ideas of our own and also proposals known from the literature. Such an experimental base is needed, because the value of a method ultimately can be determined only after an integration into a running system, as this is the only way to uncover

– the competing interplay with other methods being also implemented,
– the bits and pieces necessary to make a method really work, and
– the implicit assumptions of a method which may conflict with the settings of a real application.

The second reason for the development of \texttt{veriFun} originates from our experiences when teaching Formal Methods, Automated Reasoning, Semantics, Verification and subjects there like. The motivation of the students largely increase, if they can gather \textit{practical} experiences with the principles and methods taught. Students of Computer Science expect to see the computer to solve some problem instead of working at their own on small problems using pencil and paper only, thus treating the whole subject as pure theoretical exercise. Of course, powerful systems exist and may be used, e.g. NQTHM [2], ACL2 [9], PVS [11], Isabelle [12], VSE [5], KIV [13], HOL [3], only to mention a few, which are beyond in their abilities and the verification and reasoning problems they can handle as compared to a small system like \texttt{veriFun}. However, the performance of these systems also comes with the price of highly elaborated logics, complicated user

* \{chr.walther,schweitz\}@informatik.tu-darmstadt.de

Technical Report VFR 02/02
interfaces, severe installation requirements, license fees (for some of them) etc., which complicates their use for teaching mainly principles within the restricted time frame of a course (if not impossible at all). The situation greatly improves, however, when having a small, highly portable system, with an elaborated user interface and a simple base logic easy to grasp, which nevertheless allows the students to perform verification case studies for problems, e.g. Sorting, Searching, Basic Number Theory, Propositional Logic, Protocols, Matching, Unification etc., already known from other courses. Therefore Verifun has been developed as a JAVA [4] application, which the students can run after a 1 MB download on their home computer (whatever platform it may use) to work with the system whenever they like to do so.

In this paper, we demonstrate the use of Verifun with a verification of the Binary Search method.¹ We illustrate the challenges encountered when working on this problem, and - based on this analysis - discuss technical improvements to our system and also subjects of further research, which (as we feel) are not specific to our approach.

2 Binary Search

The Binary Search method is a well-known introductory example in courses about Algorithms and Data Structures [10]. Searching an array of n so-called keys for a certain key requires \( O(n) \) steps in the worst case, but needs only \( O(\log n) \) steps if the array is ordered. This is because by the ordering property, the domain of search can be halved after each array lookup, thus caring for the logarithmic complexity of the method. Fig. 1 displays a version of Binary Search, where \( |a| \) denotes the size of an array \( a \) of natural numbers with indices ranging from 0 to \( |a| - 1 \).

The correctness of this algorithm is not that obvious because of the puzzling index calculations which provide a plentiful source of programming errors. In order to specify the correctness of the algorithm, additional notions are needed: If we define “key \( \in a \) :\( \iff \exists i \in \{0,\ldots,|a| - 1\}. a[i] = \text{key} \)” and “ordered(a) :\( \iff \forall i, j \in \{0,\ldots,|a| - 1\}. i \le j \rightarrow a[i] \le a[j] \)”, then the correctness requirements for Binary Search can be stated formally as

\[
\forall \text{key:nat}, a:\text{array}. \text{BINSEARCH} (\text{key}, a) \rightarrow \text{key} \in a \tag{1}
\]

and

\[
\forall \text{key:nat}, a:\text{array}. \text{ordered}(a) \land \text{key} \in a \rightarrow \text{BINSEARCH} (\text{key}, a) \tag{2}
\]

where statement (1) formulates the soundness requirement and statement (2) is the completeness requirement of the algorithm. In addition, statement

\[
\forall \text{key:nat}, a:\text{array}. \text{BINSEARCH.steps}(\text{key}, a) \le |\log_2 |a| |
\]

¹Roughly speaking, this is a “medium size” problem, more complicated than, e.g., the verification of quicksort or mergesort, but with less effort than needed for e.g. proving the unique prime factorization theorem, the verification of heapsort or of a first-order matching algorithm.
procedure BINSEARCH(key:nat, a:array):bool <=
    var i := 0;
    var j := |a| - 1;
    while j > i and a[i+[(j-i)/2]] \neq key do
        if a[i+[(j-i)/2]] > key
            then j := i+[(j-i)/2] - 1
        else i := i+[(j-i)/2] + 1
    fi
    done;
    if j > i then return(true) fi;
    if j = i
        then return(a[i]=key)
    else return(false)
    fi;
end

**Fig. 1.** The Binary Search Algorithm

asserts that the algorithm causes only logarithmic costs (measured in terms of the array size), where $BINSEARCH.steps(key, a)$ counts the number of executions of the while-loop body in Fig. 1.

We aim to verify statements (1), (2) and (3) using the \texttt{VeriFun} system: Data structures are defined in \texttt{VeriFun} in a constructor-selector style discipline. The data structures \texttt{structure bool} \(<\text{false, true}\) and \texttt{structure nat} \(<\text{0, succ(pred:nat)}\) (representing natural numbers) are predefined in the system, and we use linear lists over natural numbers, defined by \texttt{structure list} \(<\text{empty, add(hd:nat, tl:list)}\), to represent arrays.

Procedures and while-loops are given by (recursively defined) functional procedures. Here we use the procedure

$$function\ element(a:list, i:nat):nat <=$$
$$\quad if\ a=empty$$
$$\quad then\ 0$$
$$\quad else\ if\ i=0\ then\ hd(a)\ else\ element(tl(a),pred(i))\ fi$$
$$\quad fi$$

for to compute the value $a[i]$ of a (non-empty) array $a$ at index $i$, and we define the tail-recursive procedure

$$function\ find(key:nat, a:list, i:nat, j:nat):bool <=$$
$$\quad if\ length(a)>j$$
$$\quad then\ if\ j>i$$
$$\quad then\ if\ element(a,plus(i,\text{half(minus}(j,i))))>key$$
$$\quad then\ find(key,a,i,pred(plus(i,\text{half(minus}(j,i))))))$$
$$\quad else\ if\ key>element(a,plus(i,\text{half(minus}(j,i))))$$

3
then find(key,a, 
    succ(plus(i,half(minus(j,i)))),j) 
  else true 
  fi 
  fi 
else if j=i then key=element(a,i) else false fi 
fi 
else false 
fi

To represent the body of the while-loop in Fig. 1. Finally, the functional version of the Binary Search algorithm is given as

\[
\text{function binsearch(key:nat, a:list):bool} <= \text{find(key,a,0,pred(length(a)))},
\]

where the definitions of the predefined procedure > and the user defined procedures \text{plus}, \text{minus}, \text{half}, \text{length}, \text{member} and \text{ordered}, which are required to define the procedure \text{find} or to formulate the correctness statements (1) and (2), are given in Appendix 6.1 or 6.2 respectively.

3 A Proof Sketch for binsearch

We briefly sketch the outline of the correctness and complexity proofs before we discuss their development with our system in Section 4.

For proving the correctness of the index calculations and also for proving the complexity statement, some properties about the >-relation must be known and some arithmetic groundwork has to be layed. The set of the ordering lemmas and of the arithmetic lemmas, which were used in the whole verification, are listed in Appendices 6.1 and 6.4.

3.1 Soundness

The soundness statement

\[
\text{lemma binsearch_is_sound} <= \text{all key:nat, a:list} \text{if(binsearch(key,a),member(key,a),true)}
\]

for \text{binsearch},\text{ cf. (1), immediately raises the proof-obligation}

\[
\text{all a:list, key:nat} \text{if(find(key,a,0,pred(length(a))),member(key,a),true)}
\]

\footnote{The ternary conditional \text{if} is the only connective for writing formulas in \text{\texttt{VerifFun}. Hence if(a,b,true) is written for a \rightarrow b, if(a,true,b) is written for a \forall b, if(a,b,false) is written for a \land b etc.}
which originates from (4) simply by the execution of the non-recursively defined procedure \texttt{binsearch}. This proof-goal has a straightforward generalization [15], viz.

\[
\text{lemma find is sound} <= \text{all } i,j,\text{key:}\text{nat, a:list} \\
\quad \text{if(find(key,a,i,j),member(key,a),true),} \\
\]

which then immediately proves the soundness statement (4) by first-order inferences only. For proving (6), some of the ordering and arithmetic lemmas and a further auxiliary lemma

\[
\text{lemma element entails member} <= \text{all } i,n:\text{nat, a:list} \\
\quad \text{if(n=element(a,i),if(length(a)>i,member(n,a),true),true)}
\]

relating \texttt{element} with \texttt{member} are required.

### 3.2 Completeness

As in the soundness case, the completeness statement

\[
\text{lemma binsearch is complete} <= \text{all a:list, key:nat} \\
\quad \text{if(ordered(a),if(member(key,a),binsearch(key,a),true),true)}
\]

for \texttt{binsearch}, cf. (2), immediately raises the proof-obligation

\[
\text{all a:list, key:nat} \\
\quad \text{if(ordered(a),} \\
\quad \quad \text{if(member(key,a),find(key,a,0,pred(length(a))),true),true)}
\]

which originates from (8) simply by the execution of the non-recursively defined procedure \texttt{binsearch}. However, this proof-goal lacks to have a straightforward generalization. The difficulty in proving (9) is typical for a statement involving a tail-recursive procedure stemming from the body of a loop: The variables which vary in a procedure (or in the loop-body respectively), viz. \(i\) and \(j\) in the present case, are replaced in the statement with non-variable terms representing the initialization of the loop-variables, viz. 0 and \(\text{pred(length(a))}\) here, thus preventing an induction proof upon these variables according to the recursion structure of the tail-recursive procedure, viz. \texttt{find}.

Often, a remedy to such a verification problem is to formulate an auxiliary lemma based on a (useful) loop-invariant: Let \(a[i..j]\) denote the partition of \(a\) wrt. the array-indices \(i \leq j\), i.e. \(a[i..j]\) denotes the sub-array of \(a\) with bounds \(i\) and \(j\). Then “\(\text{key} \in a \leftrightarrow \text{key} \in a[i..j]\)” is an invariant of the while-loop of Fig. 1 leading to the auxiliary lemma

\[
\text{lemma in.partition entails find} <= \text{all key,i,j:}\text{nat, a:list} \\
\quad \text{if(length(a)>j,} \\
\quad \quad \text{if(ordered(a),} \\
\quad \quad \quad \text{if(in.partition(key,a,i,j),find(key,a,i,j),true),true),true)}
\]
where \texttt{in.partition} is defined by

\begin{verbatim}
function in.partition(key:nat, a:list, i:nat, j:nat):bool <=
if a=empty
  then false
else if i=0
  then if key=hd(a)
      then true
      else if j=0
       then false
      else in.partition(key,tl(a),i,pred(j))
  fi
else if j=0
  then false
else if pred(i)>pred(j)
  then false
  else in.partition(key,tl(a),pred(i),pred(j))
fi
fi
fi
fi
\end{verbatim}

Using lemma (10) and a completeness statement about \texttt{in.partition}, viz.

\begin{verbatim}
lemma member_entails_in.partition <= all key:nat, a:list
if(member(key,a),
in.partition(key,a,0,pred(length(a))),
true)
\end{verbatim} (11)

the completeness statement (8) about \texttt{binsearch} can be proved by first-order reasoning only. However, the proof of statement (10) requires a bunch of further lemmas, viz.

\begin{verbatim}
∀key,i,j:nat,a:list.  j ≤ i ∧ key ∈ a[i..j] → i = j      (12)
∀key,i,j:nat,a:list.  j ≤ i ∧ key ∈ a[i..j] → a[i] = key    (13)
∀key,i,j:nat,a:list.  key ∈ a[i..j] → key = a[i] ∨ key ∈ a[i+1..j] (14)
∀key,i,j:nat,a:list.  key ∈ a[i..j] → key ∈ a[i..j−1] ∨ key = a[j] (15)
∀key,i,j,h:nat,a:list.
i ≤ h ≤ j ∧ key ∈ a[i..j] → key ∈ a[i..h] ∨ key ∈ a[h..j] (16)
∀key,i,j,h:nat,a:list. ordered(a) ∧ key ∈ a[i..j] ∧ a[h] > key → h > i (17)
∀key,h:nat,a:list. ordered(a) ∧ key > a[h] ∧ |a| > h → key > hd(a) (18)
\end{verbatim}
∀key, i, j, h:nat, a:list.
ordered(a) ∧ key ∈ a[i..j] ∧ key > a[h] ∧ |a| > h → j > h

and also some of the ordering and arithmetic lemmas.³

3.3 Complexity

For measuring the costs of \texttt{binsearch}, the costs of \texttt{find} have to be determined first. To this effect, the number of recursive calls in \texttt{find} (corresponding to the number of executions of the while-loop body in Fig. 1) is counted by the procedure \texttt{find.steps} and then the costs of \texttt{binsearch} are defined by the procedure \texttt{binsearch.steps}:

\begin{verbatim}
function find.steps(key:nat, a:list, i:nat, j:nat):nat <=
  if length(a)>j
    then if j>i
      then if element(a,plus(i,half(minus(j,i))))>key
         then succ(find.steps(key,a,
                       i,pred(plus(i,half(minus(j,i))))))
         else if key>element(a,plus(i,half(minus(j,i))))
               then succ(find.steps(key,a,
                                       succ(plus(i,
                                             half(minus(j,i))),j))
               else 0
      else 0
    fi
  else 0
fi
else 0
fi

function binsearch.steps(key:nat, a:list):nat <=
  find.steps(key,a,0,pred(length(a)))
\end{verbatim}

Using \texttt{binsearch.steps}, the complexity statement now is formulated as

\begin{verbatim}
lemma binsearch_log_bounded <= all a:list, key:nat
  if(binsearch.steps(key,a)>log2(length(a)),false,true)
\end{verbatim}

³It is interesting to see that the lemmas (12) - (15) refer to the bounds of the array, known as a source of frequent programming errors. Lemma (16) states that each legal index \(h\) separates a partition into a pair of covering sub-partitions, and lemmas (17) - (19) relate the ordering of array elements with the ordering of the array indices.
where the definition of \( \log_2 \) (computing the binary logarithm truncated downwards) is given in Appendix 6.2.\(^4\) Also here, the system replaces the call of `binsearch.steps` raising after some simplification steps the proof-obligation
\[
\text{all } a:\text{list}, \text{key}:\text{nat} \\
\text{if}(\text{find.steps}(\text{key},a,0,\text{length}(t1(a))) > \log_2(\text{succ(\text{length}(t1(a))))) \\
\text{false,} \\
\text{true}) .
\]
This subgoal now is straightforwardly generalized to a corresponding lemma about `find.steps`, viz.
\[
\text{lemma find.steps_log_bounded }\leq\text{ all } i,j,\text{key}:\text{nat}, \text{a:list} \\
\text{if}(\text{find.steps}(\text{key},a,i,j) > \log_2(\text{succ(\text{minus}(j,i))}), \\
\text{false,} \\
\text{true}) ,
\]
then proving statement (20) by first-order reasoning only. However, the proof of statement (22) requires induction and two additional lemmas, viz.
\[
\text{lemma complexity_lemma#1 }\leq\text{ all } i,j:\text{nat} \\
\text{if}(\log_2(\text{succ(\text{minus}(\text{pred(plus(i,\text{half(\text{minus}(j,i))})},i))}) \\
> \text{pred(\log_2(\text{succ(\text{minus}(j,i)))))),} \\
\text{false,} \\
\text{true})
\]
and
\[
\text{lemma complexity_lemma#2 }\leq\text{ all } i,j:\text{nat} \\
\text{if}(\log_2(\text{succ(\text{minus}(\text{pred(j)},\text{plus(i,\text{half(\text{minus}(j,i))})}))}) \\
> \text{pred(\log_2(\text{succ(\text{minus}(j,i)))))),} \\
\text{false,} \\
\text{true})
\]
and also some of the ordering and arithmetic lemmas.

4 Verifying binsearch with \texttt{VeriFun}

We now report on the efforts required to guide \texttt{VeriFun} to the verification of Binary Search. First, we briefly illustrate how to use the system before we consider the actual case. We then analyze the system's behaviour and finally discuss problems which either are specific to our system or constitute a general challenge for computer supported verification.

\(^4\) We use \texttt{if}(\texttt{x}>\texttt{y},\texttt{false},\texttt{true}), i.e. \( x \not\leq y \) instead of the more readable \( x \leq y \) because \( \leq \) and several lemmas about \( \leq \) (like transitivity, irreflexivity etc.) are predefined in \texttt{VeriFun} cf. Appendix 6.1, thus saving the additional definition of \( \leq \) and formulation and proving of lemmas about \( \leq \).
4.1 About \texttt{VeriFun}

\texttt{VeriFun} is a semi-automated system for the verification of statements about programs written in a functional programming language, cf. \cite{1,19}. In a typical session with the system, a user

- defines a (functional) program by stipulating the data structures and the procedures of the program using \texttt{VeriFun}'s language editor,
- defines statements about the data structures and procedures of the program using \texttt{VeriFun}'s language editor,
- verifies these statements and the termination of the procedures using \texttt{VeriFun}'s proof editor.

\texttt{VeriFun} consists of several fully-automated routines for theorem proving and for the formation of hypotheses to support verification. It is designed as an interactive system, where, however, the automated routines substitute the human expert in striving for a proof until they fail. In such a case, the user may step in to guide the system for a continuation of the proof.

When called to prove a statement, the system computes a proof-tree. An interaction, which may be required when the construction of the proof-tree gets stuck, is to instruct the system to prune some unwanted branches of the proof-tree (if necessary), and then

- to perform a case analysis,
- to use an instance of a lemma or an induction hypothesis,
- to unfold a procedure call,
- to apply an equation,
- to use an induction axiom, or
- to insert, to move or to delete some hypotheses.\footnote{In \texttt{VeriFun}, the nodes of a proof-tree consist of sequents, and hypotheses can be inserted into the antecedent of a sequent, can be deleted from the antecedent or moved to the succedent. The insertion of a hypothesis implements a case analysis, and the deletion of a hypothesis corresponds to a generalization step, cf. \cite{15}. The move of a hypothesis preserves equivalence and is sometimes needed (for technical reasons) to enable a subsequent induction.}

In addition, it may be necessary to formulate (and to prove) an auxiliary lemma (sometimes after providing a new definition) in order to continue with the actual proof task.

For proving the termination of a recursively defined procedure, it may be required to tell the system a useful termination function. Using this hint, the system computes termination hypotheses for the procedure which then must be verified like any other given statement. In a user guided termination proof, the termination hypotheses are based on the predefined procedure \(>\) (which is “assumed” to compute a well-founded relation). In order to ease such termination proofs, the system holds a set of predefined lemmas about \(>\), cf. Appendix 6.1.
It is therefore useful (albeit not required) not to use other “usual” orderings on natural numbers for a verification problem, e.g. $<$ or $\geq$, thus saving the formulation and proving of lemmas about these orderings. This also motivates why our formulation of Binary Search is based on $>$. Having proved the termination of a functional procedure, the system may generate additional (terminating) procedures and (verified) lemmas about them. These system-generated procedures and lemmas are used by vériFun’s automated termination analysis [16], but are also useful for proofs not related to termination. Appendix 6.3 lists the system-generated procedures and lemmas which were actually used in the verification of Binary Search.

For proving the correctness and complexity statements for Binary Search, some lemmas about the arithmetic functions are obviously also needed. In particular, monotonicity and estimation statements are required to justify the soundness of the index calculations and to prove the loop-invariant. Here we used vériFun’s Import-feature which allows to import lemmas together with their proofs from a file. While working with the system on several verification problems, we set up certain libraries, e.g., for Arithmetic and for Linear Lists, which we use when we start to work on a new problem. Usually we begin the work with an import from the libraries of all lemmas stating properties about the procedures in our verification problem. This guarantees that all proven lemmas are available when needed, where vériFun’s lemma-filter takes care that system performance is not spoiled by a vast amount of irrelevant lemmas. A library is updated by those lemmas (not too specific to a certain verification problem) which were needed in the actual verification, but are not member of a library. For the verification of Binary Search, we started with an import from our Arithmetic-library of all lemmas about plus, minus, half and log2. Appendix 6.4 lists the arithmetic lemmas which were actually needed when developing the proofs for Binary Search.

4.2 Termination

vériFun demands that the termination of each procedure which is called in a statement is verified before a proof of the statement can be started. Therefore, the system’s automated termination analysis [16] is activated immediately after the definition of a recursively defined procedure. The termination analysis recognizes termination based on (nested) structural recursion in particular, as e.g. for minus and half, which we consider as trivial termination problems. In the Binary Search example, only log2, find and find.steps cause non-trivial termination problems. The system proves the termination of log2, but fails to succeed for find and for find.steps. Obviously, minus(j,i) is a termination function for find, and providing this hint, the system generates the termination hypotheses (25) and (26), but fails to verify each of them. We therefore demand to ignore all hypotheses except $j > i$ in (25) and in (26), causing the system to compute the generalized termination hypotheses (27) and (28), which both have an automatic (induction) proof using some of the predefined, system-
all a:list, key,i,j:nat
if(length(a)>j,
 if(element(a,plus(i,half(minus(j,i))))>key,
   if(j>i,
     minus(j,i)>minus(pred(plus(i,half(minus(j,i)))),i),
     true),
   true),
 true)
(25)

all a:list, key,i,j:nat
if(length(a)>j,
 if(key>element(a,plus(i,half(minus(j,i))))),
 if(element(a,plus(i,half(minus(j,i))))>key,
   true,
   if(j>i,
     minus(j,i)>minus(pred(j),plus(i,half(minus(j,i)))),
     true)),
 true),
 true)
(26)

all i,j:nat
if(j>i,
 minus(j,i)>minus(pred(plus(i,half(minus(j,i)))),i),
 true)
(27)

all i,j:nat
if(j>i,
 minus(j,i)>minus(pred(j),plus(i,half(minus(j,i)))),
 true)
(28)

generated and arithmetic lemmas.\(^6\) As \texttt{find} and \texttt{find.steps} share the same recursion structure, we use \texttt{veriFun}'s \texttt{Create Lemma}-command to make (27) and (28) available for subsequent use. Now the verification of \texttt{find.steps}' termination only requires to provide the termination function \texttt{minus}(j,i), leaving the remaining work to the system.

4.3 Soundness, Completeness and Complexity

Proving the soundness of \texttt{binsearch} (4) does not require much effort: The system immediately comes up with the proof-goal (5), which motivates us to formulate lemma (6). However, the proof of this lemma also gets stuck, but it is immediately obvious from the failure that another lemma, viz. (7), (having a

\(^6\) The motivation for these generalizations is not only to support the proofs, but to identify the hypotheses irrelevant for the procedure's termination. As the system generates induction axioms from the body of the terminating procedures, disregarding irrelevant hypotheses results in induction axioms stronger than would be obtained otherwise, see [15] for details.
straightforward induction proof) is needed. Given the proof of (7), (6) and in turn (4) are easily proved.

The completeness statement is considerably harder to verify: The proof of (8) gets stuck with the proof-obligation (9), and the task here is to develop a lemma useful for proving this subgoal. However, different from the soundness (and complexity) case, the required lemma cannot be formulated with the given notions. Instead a new concept, viz. \textit{partition}, has to be invented which allows to demand the success of \textit{find}(key, a, i, j) only for keys \textit{key} satisfying \textit{key} \in a[i..j].

Generally, the challenge here is to spot the missing concept, of course, and, once it is found, to represent it in such a way that subsequent proofs are supported. The problem is, however, that (except in trivial cases) it seems hard to distinguish a good representation from a bad one a priori. We were trapped by this problem in the following way: It seemed straightforward to us to define a procedure \texttt{function partition(a:list,i:nat,j:nat):list <= ... computing the sublist of list a between the indices i and j, and then to prove the lemma

\begin{verbatim}
lemma member\_partition\_entails\_find <=
all key,i,j:nat, a:list
  if(length(a)>j),
    if(ordered(a),
      if(member(key,partition(a,i,j)),find(key,a,i,j),true),
      true),
    true).
\end{verbatim}

However, we failed in proving (29). New subgoals were created involving the creation of new lemmas, which in turn raised new subgoals etc., until we gave up. An analysis of this tedious and frustrating effort revealed that the problem is the use of \texttt{partition}. Roughly speaking, this procedure (using \texttt{tl}) strips of the list elements in \texttt{a} between positions 0 and \(i-1\) from the beginning of \(a\) and cuts of (using a procedure \texttt{butlast}) the list elements between \(|a| - 1\) and \(j + 1\) working from the end of \(a\) towards the beginning. The problem which comes with this approach now is that by the presence of \texttt{butlast}, the use of induction hypotheses and lemmas are spoiled so that the proofs do not get through. This problem immediately disappears if we use \textit{key} \in a[i..j], cf. \texttt{function in\_partition}, instead of \texttt{a[i..j]} as the new concept to express the lemma required to continue with the proof of (8).

Using \texttt{in\_partition}, lemma (10) can be formulated and a proof attempt can be started. Now the situation develops similar to the soundness case: Proofs get stuck, the system’s outcome is analyzed, a new lemma is spotted as the result of the analysis, a proof of the new lemma is started which either requires further lemmas (and so on ...) or gets through so that the proof of the statement calling for the proved lemma can be resumed.\footnote{The way proofs are developed with the system does not differ in principle from the way proofs are developed using pencil and paper. It is good practice to continue with the proof obligation which is most in doubt: If it seems obvious that the spotted}
the lemmas (12)....(19) were invented and proved, were (16) is the central one. Each of these lemmas evolved with less imagination from the analysis of the systems outcome when it gets stuck in proving another lemma. The proofs use some of the predefined and system-generated lemmas, and the proof of (10) also uses some of the arithmetic lemmas. Finally, the completeness statement (8) is proved by some first-order reasoning steps using (10) and (11).

Also the proof of the complexity statement requires more work than the soundness proof, but considerably less effort than needed in the completeness case. The proof of (20) gets stuck with the proof-obligation (21), and the missing lemma, viz. (22), is easy to spot. Again, the induction proof of (22) gets stuck and it is immediately obvious from the result and the induction hypotheses that the lemmas (23) and (24) will do the job. Although these lemmas look considerably nasty and one may expect a lot of tedious proof steps, to our surprise both lemmas had a straightforward induction proof (using some of the predefined and arithmetic lemmas). After (23) and (24) being proved, the system immediately comes up with a proof of (22), and then the proof of (20) is routine.

4.4 Analysis

The verification of Binary Search is characteristic for the way proofs are developed with VeriFun (which, as we suspect, does not differ in principle from the way when using other systems). User interactions are required from time to time to recover from the weakness of the first-order theorem prover, which, in particular, is responsible for proving the base and step cases of an induction. Such a theorem prover needs to terminate, because the system frequently deals with proof-obligations which are not valid but can be proved by induction only. Consequently, a sound theorem prover must be incomplete. E.g., the proof of (6) gets stuck with a proof-obligation which is true but invalid, and therefore a lemma, viz. (7), must be formulated and proved to finish the proof of (6).

In addition, a reasonable compromise must be made between theorem-proving performance and system efficiency (measured both in memory and computing time), thus providing a further source of incompleteness. To yield an acceptable answer time, VeriFun uses several kinds of resource restrictions. When searching for a proof, the system’s first-order theorem prover considers verified lemmas (and induction hypotheses). To test the applicability of such a lemma, the prover calls itself recursively, however with additional restricted resources not to waste too much time for deciding whether a lemma supports the proof under consideration. Consequently, the prover sometimes overlooks a useful lemma and then must be told to use it. E.g. when proving (6), the system must be called to use lemma will do the job, one should continue first with proving the lemma. On the other hand, if the lemma’s proof seems easy but its use is in doubt, it is advantageous to verify first that the lemma is useful indeed. VeriFun supports both ways of proof development at the user’s wish.

In fact, it costs far more time TEXing one of the formulas than the system needs to prove the log2-boundedness of binsearch.
Table 1. User Interactions for Binary Search

<table>
<thead>
<tr>
<th>Proof-Obligations</th>
<th>Sound</th>
<th>Complete</th>
<th>Complexity</th>
<th>Arithmetic</th>
<th>Termination</th>
<th>Σ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert Lemma</td>
<td>3</td>
<td>11</td>
<td>4</td>
<td>13</td>
<td>11</td>
<td>42</td>
</tr>
<tr>
<td>Insert Function</td>
<td>2</td>
<td>10</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>20</td>
</tr>
</tbody>
</table>

| Proof-Tree Edits  | 1      | 3         | 3          | 4          | 4           | 14 |
| Case Analysis     |        |           |            |            |             |    |
| Use Lemma         | 1      | 3         | 1          | —          | —           | 5  |
| Unfold Procedure  |        |           | 2          | 1          | —           | 3  |

| Apply Equation    | —      |           | —          | —          | —           |    |
| Induction         | —      |           | —          | —          | —           |    |
| Hypotheses        | —      |           | —          | —          | —           |    |

| none              | 2      | 8         | 1          | 11         | 9           | 31 |

Table 1. User Interactions for Binary Search

(7) albeit available, because it fails to prove \( j > i \rightarrow j > i + \lfloor (j - i) / 2 \rfloor \) when operating under additional restrictions, but easily proves this subgoal when instructed to use (7), because then theorem proving proceeds with the resources initially given to the system.

Table 1 illustrates the efforts when verifying Binary Search with VeriFun.\textsuperscript{9} The row Proof-Obligations displays the overall number of statements proved in this example, separated into the subtasks “Soundness”, “Completeness”, “Complexity”, “Arithmetic” and “Termination”. E.g., 11 lemmas had to be proved for verifying the completeness statement. The 13 lemmas about the arithmetic functions (collected in Appendix 6.4), which were used for Binary Search, are listed separately in the Arithmetic-column. The whole case study uses 11 recursively defined procedures (including those for arithmetic, cf. Appendix 6.2) for which the system also had to prove termination.

Below, the Insert Lemma - row displays the number of lemmas we had to create to support the proof of each of the main statements respectively, e.g. 10 for Completeness, and the Insert Function - row shows the number of additional definitions required for formulating these lemmas, e.g. 1 for Completeness. In the Termination-column, the number of termination functions which we had to submit to the system are counted in addition. The Insert Lemma - row gives an account of the system’s inability to spot the “right” lemma by itself, which is rather large for VeriFun as (except for the termination analysis) no methods for lemma speculation (and generalization, in particular) are implemented in the used system version 2.5.6.

The Proof-Tree Edits - row counts the number of proof-tree edits required to guide the system to success even if the required lemmas are available, subsequently separated into the different activities. E.g., 3 user interventions were re-

\textsuperscript{9} Predefined and system-generated procedures and lemmas are not considered in this table as they are given “for free”.

14
quired for the completeness case. The Hypotheses-rules had to be called 4 times for Termination to provide the generalizations of the termination hypotheses for find, and had been called 2 times for Arithmetic, because the commutativity of plus was proved by 2 nested inductions. Finally, the last row gives the number of proof-obligations which went through the system with no user guidance at all.

5 Concluding Remarks

The number of required user interactions measured in terms of additional definitions and proof-tree edits gives a fair account of a system’s automatization degree. Since approx. 1/3 proof-tree edit is required for each proof-obligation in this case study, the first-order prover shows a good performance here. This judgement is based on a further analysis of the proofs computed by the system, cf. [1]. For instance, after being told to use (16) in the proof of (10), the system continued (and succeeded) without any further intervention computing a proof of 127 steps using the 2 induction hypotheses and 20 different instances of 13 lemmas. We consider this degree of automatization as an important feature, because it relieves a user to reason which lemmas to consider and how to apply them, which is not obvious for a large lemma set and a long proof. Also the heuristic for choosing induction axioms and the implemented equality reasoning behaves well here, as the system had not to be called to use a certain induction or to apply a certain equation.

Nevertheless, we intend to improve some of the values in Table 1: The need for the insertion of a lemma can be decreased by extending the system with some state-of-the-art technology for lemma speculation, e.g. [6],[7],[8],[14],[17],[18], and based on experiences gained from further case studies - the need of proof-tree edits can be decreased by a finer tuning of the theorem prover wrt. the performance vs. efficiency tradeoff. Also the steady increase of hardware performance may improve system performance: \textsc{c{e}r{f}i}{\textsc{p}u}{\textsc{n}}’s resource restrictions are controlled by certain resource parameters. This eases to benefit from an increase of hardware performance, as only the resource parameters need to be adjusted when moving to a more powerful platform.

However, a problem which seems far more serious is the problem of concept formation, i.e. the invention of new concepts which are required to formulate

\footnotesize
\textsuperscript{10} The Use Lemma row also counts the use of induction hypotheses. When distinguishing proof-tree edits between Case Analysis, Use Lemma and Unfold Procedure, one should bear in mind that very often the desired effect can be obtained by either of the proof-rules. For instance, a cases analysis may enable the system to recognize a useful lemma or to unfold a procedure call, a required case analysis may implicitly evolve from the use of a lemma, etc.

\textsuperscript{11} \textsc{c{e}r{f}i}{\textsc{p}u}{\textsc{n}}’s equality reasoning is based on conditional term rewriting, where the orientation of the equations are computed by the system.

\textsuperscript{12} Since the right tuning of the resource parameters require a deep insight into the system’s architecture and operation, these parameters are not at a users disposal, but are fixed for each system version.
a necessary lemma. E.g. in the Binary Search verification, some new concept, viz. \texttt{in.partition}, is required to express the success of \texttt{find}, cf. (10). The invention of new, \textit{usefully represented}, concepts require much more creativity than needed to formulate a lemma with the notions given, or to tell the prover what to do next. We believe that this constitutes the main obstacle for a wider use of theorem-proving based verification systems, and major improvements in this situation will only come with some computer support also for this problem.

6 Appendix

6.1 Predefined Procedures and Lemmas

\begin{verbatim}
function >>(x:nat, y:nat):bool <=
  if x=0 then false else if y=0 then true else pred(x)\geq pred(y) fi fi

\forall x,y:nat. x = y \rightarrow x \neq y
\forall x,y,z:nat. x \geq y \wedge y \geq z \rightarrow x \geq z
\forall x,y,z:nat. x \neq y \wedge y \neq z \rightarrow x \neq z
\forall x,y:nat. x \geq y \vee y \geq x \vee x = y
\forall x,y:nat. x\neq 0 \wedge x = y \rightarrow x \geq y-1
\forall x,y:nat. x \geq y \rightarrow x \geq y-1
\forall x,y:nat. x \geq y \rightarrow (x \geq y \vee x = y)
\end{verbatim}

6.2 Auxiliary Procedures

\begin{verbatim}
function minus(x:nat, y:nat):nat <=
  if x=0
    then 0
  else if y=0 then x else minus(pred(x),pred(y)) fi
fi

function half(x:nat):nat <=
  if x=0
    then 0
  else if pred(x)=0 then 0 else succ(half(pred(pred(x)))) fi
fi

function log2(x:nat):nat <=
  if x=0
    then 0
  else if pred(x)=0
    then 0
  else succ(log2(succ(half(pred(pred(x))))))
fi
fi

\end{verbatim}

\footnote{We write \(1^+\) for the successor function and \(\sim 1\) for the predecessor.}
function plus(x:nat, y:nat):nat <=
if x=0 then y else succ(plus(pred(x),y)) fi

function member(n:nat, k:list):bool <=
if k=empty then false
else if n=hd(k) then true else member(n,tl(k)) fi
fi

function length(k:list):nat <=
if k=empty then 0 else succ(length(tl(k))) fi

function ordered(k:list):bool <=
if k=empty then true
else if tl(k)=empty then true
else if hd(k)>hd(tl(k)) then false else ordered(tl(k)) fi
else if hd(k)>hd(tl(k)) then false else ordered(tl(k)) fi
fi

6.3 System-Generated Procedures and Lemmas

function minus$1(x:nat, y:nat):bool <=
if x=0 then false
else if y=0 then false else true fi
fi

function half$1(x:nat):bool <=
if x=0 then false else true fi

∀ x, y : nat. x ≮ (x − y) → (x − y) = x
∀ x : nat. x ≮ \lfloor x/2 \rfloor → \lfloor x/2 \rfloor = x

∀ x, y : nat. ¬ minus$1(x, y) → (x − y) = x
∀ x : nat. ¬ half$1(x) → \lfloor x/2 \rfloor = x

∀ x : nat. half$1(x) → x > \lfloor x/2 \rfloor

6.4 Arithmetic Lemmas

∀ x_1, x_2 : nat. x_1 = x_2 → x_1 − x_2 = 0
∀ x, y, z : nat. x > y → x > y + z

∀ x, y : nat. x ≠ 0 → x + y > x − 1
∀ x, y : nat. x + y = y + x

∀ x, y, z : nat. x > y → \lfloor x/2 \rfloor ≠ \lfloor y/2 \rfloor

∀ x : nat. \lfloor x/2 \rfloor ≠ x − 1
∀ x : nat. \lfloor x/2 \rfloor − 1 ≠ \lfloor (x−1)/2 \rfloor

∀ x, y : nat. x ≠ y ↔ x − y = 0
∀ x, y, z : nat. x > y → x > x + (y + z)
∀ x, y, z : nat. (x + y) + z = x + (y + z)
∀ x, y : nat. x ≠ y → \lfloor \log_2(x) \rfloor ≠ \lfloor \log_2(y) \rfloor

∀ x, y : nat. x > y → x + \lfloor (x − y)/2 \rfloor ≠ x − 1
∀ x, y : nat. (x − y) − 1 = (x − 1) − y

∀ x, y : nat. x ≠ y → \lfloor x/2 \rfloor ≠ \lfloor (x−1)/2 \rfloor
∀ x : nat. \lfloor (x−1)/2 \rfloor ≠ x − 1
∀ x : nat. (x−1) − \lfloor x/2 \rfloor ≠ \lfloor (x−1)/2 \rfloor
References