A Machine Supported Proof of the Unique Prime Factorization Theorem

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 Unique Prime Factorization Theorem. We illustrate the operation and performance of our system and present the challenges encountered when working on this problem.

1 Introduction

We develop the \texttt{VeriFun} system [1],[15], 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.

The second reason for the development of \texttt{VeriFun} origins 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 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 [3], ACL2 [9], PVS [10], Isabelle [11], VSE [8], KIV [12], HOL [6], 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 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). This 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 etc., already

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

Technical Report VFR 02/03
known from other courses. Therefore \texttt{VeriFun} has been developed as a JAVA [7] 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.

When developing an automated reasoning system, thorough case studies are inevitable. During the development of \texttt{VeriFun} we benefit to a great extend from the work of Boyer and Moore [2] when we used the system for the verification of a Tautology Checker [3], the RSA Public Key Encryption Algorithm [5], the Unsolvability of the Halting Problem [4] etc. Here we demonstrate the use of \texttt{VeriFun} with a proof of the Unique Prime Factorization Theorem, a case study which is also taken from the Boyer-Moore corpus [3].\footnote{This is a problem, more complicated than, e.g., the verification of Binary Search [16], but with less effort than needed for e.g. proving the Unsolvability of the Halting Problem [4] or the correctness of RSA [5].} We illustrate the operation and performance of our system and present the challenges encountered when working on this problem.

\section{The Prime Factorization Problem}

Let \texttt{prime} be a predicate over natural numbers which is true for all prime numbers (and false for all non-primes), let \texttt{prime.list} be a predicate over (linear) lists (of natural numbers) which is true for all lists consisting of prime numbers only, and let \texttt{prime.factors} be a function intended to compute a list of prime factors for each natural number \( > 1 \) given as input (and returning the empty list for inputs \( \leq 1 \)). Then the correctness requirements for \texttt{prime.factors} can be stated formally as

\[
\forall x:\texttt{nat}. \texttt{prime.list} (\texttt{prime.factors}(x))
\]  

and

\[
\forall x:\texttt{nat}. \quad x \neq 0 \rightarrow \texttt{prod} (\texttt{prime.factors}(x)) = x
\]  

where statement (1) formulates the soundness requirement and statement (2) is the completeness requirement of \texttt{prime.factors}. In addition, statement

\[
\forall k,l:\texttt{list}. \quad \texttt{prod}(k) = \texttt{prod}(l) \land \texttt{prime.list}(k) \land \texttt{prime.list}(l) \rightarrow k \approx l
\]  

asserts that the factorization of each natural number into a list of prime numbers is unique up to list permutation.\footnote{\texttt{prod}(k) denotes the product of all members of a list \( k \) and \( \approx \) stands for list-permutation.}

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 <= false,true and structure nat <= 0, succ(pred:nat)} (representing natural numbers) are predefined in the system,
and linear lists over natural numbers can be defined by
\textit{structure list} \leq \textit{empty}, \textit{add}\((\textit{hd:}\textit{nat}, \textit{tl:}\textit{list})\).

Algorithms are represented by (recursively defined) functional procedures. For instance, the procedure for computing a list of prime factors is defined by

\begin{verbatim}
function prime.factors\((\textit{x:}\textit{nat}):\textit{list} \leftarrow \\
  \text{if x=0} \\
  \text{then empty} \\
  \text{else if pred(x)=0} \\
    \text{then empty} \\
  \text{else if prime1(x,pred(x))} \\
    \text{then add(x,empty)} \\
  \text{else app(prime.factors(greatest.factor(x,pred(x))),} \\
    \text{prime.factors(quotient(x,} \\
    \text{greatest.factor(x,pred(x))))))} \\
  \text{fi} \\
\text{fi}
\end{verbatim}

where \textit{function prime1}\((\textit{x,y:}\textit{nat}):\textit{bool} \leftarrow \ldots \text{returns true iff y \neq 0 and x is divided by no number z} \in \{2,\ldots,y\}, \textit{function greatest.factor}\((\textit{x,y:}\textit{nat}):\textit{bool} \leftarrow \ldots \text{returns the greatest divisor z} \in \{2,\ldots,y\} \text{ of x, if} y \geq 2, \text{and returns x otherwise, function quotient}\((\textit{x,y:}\textit{nat}):\textit{bool} \leftarrow \ldots \text{computes the (downwards truncated) quotient of} x \text{ by} y, \text{if} y \neq 0, \text{and returns} x \text{ otherwise, and function app}\((\textit{x,y:}\textit{list}):\textit{list} \leftarrow \ldots \text{computes the list-concatenation, cf. Appendix 5.2.}}

3 Solving the Prime Factorization Problem with \textit{\texttt{VeriFun}}

We report on the efforts required to guide \texttt{VeriFun} to the verification of the correctness statements (1) and (2) for \textit{prime.factors} and also to the proof of the uniqueness theorem (3). We start with a brief illustration how to use the system before we consider the actual case and analyze the system’s behavior.

3.1 About \textit{\texttt{VeriFun}}

\texttt{VeriFun} is a semi-automated system for the verification of statements about programs written in a functional programming language, cf. [1],[15]. 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,
- to insert, move or delete some hypothesis in the sequent of a proof-node.\footnote{In \texttt{verifun}, the nodes of a prooftree 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 \textit{case analysis}, and the deletion of a hypothesis corresponds to a \textit{generalization} step, cf. [13]. The movement 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 \textit{termination function}. Using this hint, the system computes \textit{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 \textit{well-founded} relation). In order to ease such termination proofs, the system holds a set of \textit{predefined lemmas} about $>$, cf. Appendix 5.1.

Having proved the termination of a functional procedure, the system may generate additional (terminating) procedures and (verified) lemmas about them. These \textit{system-generated} procedures and lemmas are used by \texttt{verifun}'s automated termination analysis [14], but are also useful for proofs not related to termination. The system-generated procedures and lemmas which were actually used in this case study are listed in Appendix 5.3.

For proving the statements of the Unique Prime Factorization Problem, a considerable amount of lemmas about the involved arithmetic functions are obviously also needed. Here we used \texttt{verifun}'s \textit{Import}-feature which allows to import lemmas together with their proofs from a file. In this case study, we started with an import from our Arithmetic-Library of all lemmas about the procedures \texttt{plus, times, minus, min(imum), quotient, remainder, gcd} and \texttt{prod}. See [1] for the 55 arithmetic lemmas which were actually needed when developing the proofs. The auxiliary lemmas which are specific for this case study are listed in Appendix 5.4.

3.2 Termination

\texttt{verifun} 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 [14] 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 \texttt{plus} or \texttt{delete}, which we consider as trivial termination problems. Among the procedures of this case study, cf. Appendix 5.2, only \texttt{quotient, remainder, gcd, perm} and \texttt{prime.factors} cause non-trivial termination problems. The system proves the termination of \texttt{quotient, remainder, gcd} and \texttt{perm} automatically,
but fails to succeed for \texttt{prime.factors}. Obviously, $Ax : \text{nat}. \ x$ is a termination function for \texttt{prime.factors}, and after providing this hint, the system generates two termination hypotheses for \texttt{prime.factors} which are automatically verified using two auxiliary lemmas.\footnote{In particular, a so-called \textit{boundedness-lemma}, viz. "$\forall x, y : \text{nat}. \ x \neq 0 \land y \neq 0 \land y \neq 1 \rightarrow x > \text{quotient}(x, y)$", is needed for \texttt{quotient}. Using an alternative definition of \texttt{quotient} (where the argument $\text{minus}(x, y)$ of \texttt{quotient}'s recursive call is replaced by $\text{minus}(\text{pred}(x), \text{pred}(y))$) this boundedness-lemma is speculated (and verified) by the system, and \texttt{VeriFun} recognizes the termination of \texttt{prime.factors} without any hints. However, \texttt{quotient}'s alternative definition costs much more user interactions when verifying theorems about \texttt{quotient}, \texttt{remainder} and \texttt{gcd} than necessitated by the original definition (and saved for the termination requirement of \texttt{prime.factors}).}

### 3.3 Soundness, Completeness and Uniqueness

The soundness and the completeness statement have a non-interactive proof, if the required auxiliary lemmas are provided. The same holds for the uniqueness statement, except that the system must be instructed to use a specific case analysis. In the soundness case, only one auxiliary lemma, viz. \texttt{prime.list.append}, is needed which also has a non-interactive proof. The completeness statement needs three auxiliary lemmas, viz. \texttt{greatest.factor.divides}, \texttt{greatest.factor.is.zero} and \texttt{greatest.factor.is.one}. The first lemma requires a certain induction and also a certain case-analysis, but no interaction is needed for the other two lemmas. In the uniqueness case, the auxiliary lemmas \texttt{prime.list.delete}, \texttt{prod_prime.list.not_one} and \texttt{prime.member} are needed. The first lemma needs no interaction, \texttt{prod_prime.list.not_one} needs one \texttt{Use-Lemma} hint wrt. an arithmetic lemma, and \texttt{prime.member} is proved by first-order reasoning, where the system must be told to use the auxiliary lemma \texttt{prime.list.prod} and to apply a certain equation. The lemma \texttt{prime.list.prod} expresses the key fact for proving uniqueness and its proof requires the most effort: The auxiliary lemmas \texttt{prime1.basic} and \texttt{prime.key} must be used, which in turn require \texttt{hack1} and \texttt{prime.gcd}, which in turn needs \texttt{prime1.basic}. To guide the system to a proof of \texttt{prime.list.prod} and its auxiliary lemmas, the system has to be told to use a certain induction once, 11 times to use an instance of a lemma, 8 times to apply an equation, 4 times to perform a case analysis, and 2 times to unfold a procedure. All proofs of this case study can be obtained from [1].

Chapter XIX of [3] provides the original presentation of the problem and the development of its solution, and [2] presents the proofs. We followed the proof outline presented there when proving the three main statements (1), (2) and (3) using \texttt{VeriFun}. This eased our work considerably, because on such a problem the main burden of a system user consists in the speculation of useful lemmas when the proof of the statement under consideration gets stuck. Some of the auxiliary lemmas needed here, like \texttt{greatest.factor.is_zero} or \texttt{greatest.factor.divides}, are quite obvious and easy to speculate, because they remain as some unsolved part of a proof obligation, while others, e.g.
\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|c|c|}
\hline
         & Sound & Complete & Unique & Arithmetic & Termination & \(\Sigma\) \\
\hline
Proof Obligations & 2 & 4 & 9 & 55 & 16 & 86 \\
\hline
Insert Lemma & 1 & 3 & 8 & – & 2 & 14 \\
\hline
Prooftree Edits & – & 2 & 27 & 23 & – & 50 \\
\hline
Case Analysis & – & 1 & 5 & 2 & – & 8 \\
Use Lemma & – & – & 11 & 1 & – & 12 \\
Unfold Procedure & – & – & 2 & 5 & – & 7 \\
\hline
Apply Equation & – & – & 8 & 8 & – & 16 \\
Induction & – & 1 & 1 & 2 & – & 4 \\
Hypotheses & – & – & – & 3 & – & 3 \\
\hline
none & 2 & 3 & 1 & 45 & 15 & 66 \\
\hline
\end{tabular}
\caption{User Interactions for Unique Prime Factorization case study}
\end{table}

\texttt{prime1.basic} or \texttt{hack1}, look strange and we suspect that their formulation was not that obvious (as the names chosen by authors may indicate).

\subsection{Analysis}

The verification of the statements in this case study is characteristic for the way proofs are developed with \texttt{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.

Table 1 illustrates the efforts when verifying the statements of this case study with \texttt{VeriFun} (system version 2.5.6).\footnote{Predefined and system-generated procedures and lemmas are not considered in this table as they are given “for free”.} The row \textit{Proof Obligations} displays the overall number of statements proved, separated into the subtasks “Soundness”, “Completeness”, “Uniqueness”, “Arithmetic” and “Termination”. E.g., 4 lemmas had to be proved for verifying the completeness statement. The 55 lemmas about the arithmetic functions are listed separately in the \textit{Arithmetic}-column. The whole case study uses 16 recursively defined procedures (including those for arithmetic, cf. Appendix 5.2) for which the system also had to prove termination.

Below, the \textit{Insert Lemma} - row displays the number of the auxiliary lemmas which had to be created to support the proof of each of the main statements respectively, e.g. 3 for \textit{Completeness}. In the \textit{Termination}-column, the number of termination functions which we had to submit to the system are counted in addition.

The \textit{Prooftree Edits} - row counts the number of prooftree edits required to guide the system to success even if the required lemmas are available, subsequently separated into the different activities. E.g., 2 user interventions were required for the completeness case, one to tell the system a useful induction and...
the other to introduce a case analysis. The Hypotheses-rules had to be called 3 times for Arithmetic, because the commutativity of plus and of times was proved by nested inductions. Finally, the last row gives the number of proof-obligations which went through the system with no user guidance at all.

4 Concluding Remarks

The number of required user interactions measured in terms of prooftree edits gives a fair account of a system’s automatization degree. Approx. 78% of all proof obligations went through the system with no user guidance, indicating a high degree of automatization. On the contrary, (caused by the uniqueness proof) almost 0.6 prooftree edits are required for each proof-obligation in this case study, suspecting a low degree of mechanization. However, considering the latter value one has to take into account the complexity of the domain of discourse: Different to the domains of e.g. Matching, Sorting, Searching etc., statements from number theory quite often require a great amount of user interactions, cf. e.g. [5] and [4], an observation also made by others.\footnote{Personal communication with J. S. Moore, Feb. 2002} We therefore feel that the system shows a good performance here, a judgement which is also based on a further analysis of the proofs computed by the system, cf. [1]. For instance, when proving the step case of the completeness statement (2) for prime.factors, the system computed a proof of 1034 steps using 16 instances of the 2 induction hypotheses and 30 instances of 12 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 to be told to use a certain induction only 4 times (for the 70 statements) and has to be called only 16 times in this equality intensive domain to apply a certain equation.

5 Appendix

5.1 Predefined Procedures and Lemmas

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

\[
\begin{align*}
\forall x,y: \text{nat}. \ x = y \rightarrow x \neq y & \quad \forall x,y: \text{nat}. \ x = y \rightarrow x^{\prime} \neq y \\
\forall x,y: \text{nat}. \ x > y \land y > z \rightarrow x > z & \quad \forall x,y: \text{nat}. \ x > y \rightarrow y \neq 0 \\
\forall x,y: \text{nat}. \ x \neq y \land y \neq z \rightarrow x \neq z & \quad \forall x,y: \text{nat}. \ x > y \rightarrow y \neq x \\
\forall x,y: \text{nat}. \ x > y \lor x > y \land x = y & \quad \forall x,y: \text{nat}. \ x^{\prime} > y \rightarrow x \neq 0 \\
\forall x,y: \text{nat}. \ x \neq 0 \land x = y \rightarrow x > y^{\prime}1 & \quad \forall x,y: \text{nat}. \ x \neq 0 \land y = 0 \rightarrow x > y \\
\forall x,y: \text{nat}. x > y \rightarrow x > x^{\prime}1 & \quad \forall x,y: \text{nat}. x^{\prime} > y \rightarrow x \neq 0 \\
\forall x,y: \text{nat}. x > y^{\prime}1 \rightarrow (x > y \lor x = y) & \quad \forall x,y: \text{nat}. x = y \rightarrow 1+x > y \footnote{We write \(1+\) for the successor function and \(-1\) for the predecessor.}
\end{align*}
\]
5.2 Procedures

function plus(x:nat, y:nat):nat <=
if x=0 then y else succ(plus(pred(x),y)) fi

function times(x:nat, y:nat):nat <=
if x=0 then 0 else plus(times(pred(x),y),y) fi

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 min(x,y:nat):nat <=
if x=0
then 0
else if y=0 then 0 else succ(min(pred(x),pred(y))) fi
fi

function quotient(x,y:nat):nat <=
if y=0
then x
else if y>x
then 0
else if x=0 then 0 else succ(quotient(minus(x,y),y)) fi
fi
fi

function remainder(x,y:nat):nat <=
if x=0
then 0
else if y=0
then y
else if y>x then x else remainder(minus(x,y),y) fi
fi

function gcd(x,y:nat):nat <=
if x=0
then y
else if y=0
then x
else if x>y
then gcd(minus(x,y),y)
else gcd(x,minus(y,x))
fi
fi
fi
function delete(n:nat, k:list):list <=
  if k=empty
    then empty
  else if n=hd(k) then tl(k) else add(hd(k),delete(n,tl(k))) fi
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 app(k:list, l:list):list <=
  if k=empty then l else add(hd(k),app(tl(k),l)) fi

function perm(k:list, l:list):bool <=
  if k=empty
    then l=empty
  else if member(hd(k),l)
    then perm(tl(k),delete(hd(k),l))
    else false fi
  fi

function prod(k:list):nat <=
  if k=empty then 1 else times(hd(k),prod(tl(k))) fi

function greatest.factor(x:nat,y:nat):nat <=
  if y=0
    then x
  else if pred(y)=0
    then x
    else if remainder(x,y)=0
      then min(x,y)
      else greatest.factor(x,pred(y))
    fi
  fi

function prime1(x:nat,y:nat):bool <=
  if y=0
    then false
  else if pred(y)=0
    then true

\footnote{This definition differs from the original in \cite{Baeten92} by the result term \texttt{min(x,y)} instead of \texttt{y}. Obviously, our definition and the original one compute the same function if \(x \neq 0\) (which is the relevant case). We need this modification to support the automated termination analysis \cite{Baeten93}, as otherwise the system does not generate the lemmas (5) - (8) of Appendix 5.3, and then would fail to prove the termination of \texttt{prime.factors}.}
else if remainder(x, y)=0
  then false
  else prime1(x, pred(y))
fi
fi

function prime(x:nat):bool <=
if x=0
  then false
else if pred(x)=0 then false else prime1(x, pred(x)) fi
fi

function prime.list(l:list):bool <=
if l=empty
  then true
else if prime(hd(l)) then prime.list(tl(l)) else false fi
fi

5.3 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 min$1$(x:nat, y:nat):bool <=
if x=0
  then false
else if y=0 then true else min$1$(pred(x), pred(y)) fi
fi

function min$2$(x:nat, y:nat):bool <=
if x=0
  then y=succ(pred(y))
else if y=0 then false else min$2$(pred(x), pred(y)) fi
fi

function greatest.factor$1$(x:nat, y:nat):bool <=
if y=0
  then false
else if pred(y)=0
  then false
  else if remainder(x, y)=0
    then min$1$(x, y)
    else greatest.factor$1$(x, pred(y))
  fi
fi
fi
5.4 Auxiliary Lemmas

lemma greatest.factor_divides <= all x,y:nat
   if(y=0,true,remainder(x,greatest.factor(x,y))=0)

lemma greatest.factor_is_zero <= all x,y:nat
   if(greatest.factor(x,y)=0,x=0,true)

lemma greatest.factor_is_one <= all x,y:nat
   if(pred(greatest.factor(x,y))=0,pred(x)=0,true)

lemma prime.list_append <= all k,l:list
   if(prime.list(k),
      if(prime.list(1),
        prime.list(app(k,1)),
        if(prime.list(app(k,1)),false,true)),
      if(prime.list(app(k,1)),false,true))

lemma prime1.basic <= all x,y,z:nat
   if(prime1(x,plus(y,z)),
      if(remainder(x,z)=0,
         if(z=x,true,pred(z)=0),
         true),
      true)

lemma hack1 <= all n,x,y,z:nat
   if(times(x,y)=times(z,n),
      if(x=gcd(times(x,n),times(x,y)),remainder(x,n)=0,true),
      true)

lemma prime1.gcd <= all n,x:nat
   if(prime1(n,pred(n)),
      if(remainder(n)=0,true,pred(gcd(n,x))=0),
      true)

\* All variables are universally quantified over nat. gf abbreviates greatest.factor.
lemma prime_key <= all n,x,y,z:nat
  if(prime(n),
    if(times(x,y)=times(z,n),
      if(remainder(x,n)=0,true,remainder(y,n)=0),
      true),
    true)

lemma prime_list_prod <= all n:nat, k:list
  if(remainder(prod(k),n)=0,
    if(prime(n),if(prime_list(k),member(n,k),true),true),
    true)

lemma prime_list_delete <= all n:nat, k:list
  if(prime_list(k),prime_list(delete(n,k),true))

lemma prod_prime_list_not_one <= all k:list
  if(prod(k)=1,
    if(prime_list(k),k=empty,true),true),
  true)

lemma prime_member <= all k,l:list, n:nat
  if(times(n,prod(l))=prod(k),
    if(prime(n),
      if(prime_list(k),member(n,k),true),
      true),
    true)

References


