Bachelor thesis

**Formal Verification of a Big Integer Library Written in C0**

Sabine Fischer
sabine@wjpsserver.cs.uni-sb.de

Saarland University, Computer Science Department
Institute for Computer Architecture and Parallel Computing
Prof. Dr. W. J. Paul
Contents

1 Introduction 1

2 Basics 2
   2.1 The Verification Process 2
   2.2 The C0 Programming Language 2
      2.2.1 Types in C0 3
      2.2.2 Statements 4
   2.3 Interactive Theorem Proving in Higher Order Logic 5
      2.3.1 Lambda Calculus and ML Definitions 6
      2.3.2 Higher Order Logic 11
   2.4 The Hoare Logic 16
      2.4.1 Basic Framework 17
      2.4.2 Hoare Rules 19
      2.4.3 Formalization of Heap Structures 21
      2.4.4 Heap Abstractions and the dList Predicate 22

3 The Big Integer Package 26
   3.1 Representation of Big Integers 26
   3.2 Informal Specification of the Big Integer Package 28
      3.2.1 Constant Definitions 28
      3.2.2 Public Interface 28
      3.2.3 Private Functions 31
Chapter 1

Introduction

With the increasing demand for error-free software, software verification has become an important factor. It is common knowledge that misconceptions in programming, simple typing errors and obscure bugs cause, at best, a dry laugh, or more seriously, great financial expenses.

The Verisoft\(^1\) project aims to demonstrate and further the current possibilities of formal verification. This long-term research project consists of five specific application tasks, four of them from industrial background. Computer-aided verification tools are used to guarantee correctness on all abstraction levels. Part of the academic Verisoft project is a thoroughly verified computer system, consisting of hardware, an operating system with compiler, several libraries and an exemplary user application.

This bachelor thesis is part of the implementation and verification effort of a working fundamental algorithms and data structures package in the Verisoft project. We implement and partially verify a library of big integers based on doubly linked lists which will be used mainly in the cryptography library.

In Chapter 2, we will briefly go over the basic concepts of formal verification that are needed to understand the actual thesis. We give a short overview of the C0 programming language that is used throughout the paper, introduce the basic concepts of higher order logic and Hoare logic, and consider a model for heap structures in higher order logic.

Chapter 3 deals with the actual big number package. We give a formal definition of big numbers, list the methods and utility functions contained in the package, and give specification and verification outlines for the verified functions. The proofs of formal correctness for the functions \texttt{bigIntInsertDigitFront}, \texttt{bigNumAssignInt}, and \texttt{bigIntCompare} are studied in more detail.

\(^1\)http://www.verisoft.de
Chapter 2

Basics

2.1 The Verification Process

This thesis deals with the formal verification process of a big integer package. We have implemented the package using the programming language C0 which has been specifically developed with verification in mind.

The framework used for formal verification is the Hoare logic environment (as described by Norbert Schirmer [10]) of the Isabelle/HOL theorem prover [9]. Since the imperative language model of Isabelle’s Hoare logic and our programming language C0 do not coincide, it is necessary to translate our C0 code to Isabelle’s imperative language in such a way that correctness of the Isabelle code implies correctness of our C0 code. This step is done using the C0 translation procedure developed during the Verisoft project.

We give specification in the form of Hoare triples in Isabelle’s Hoare logic environment. Using these specifications and the imperative “source code” generated from our C0 code, the verification condition generator implemented in Isabelle/HOL then generates appropriate proof obligations. However, since there is no generic algorithm to determine loop invariants in verification condition generation, these invariants have to be annotated manually.

When we obtained a number of proof obligations in higher order logic, all that is left to do is to solve them. This is done interactively using the Isabelle/HOL environment.

2.2 The C0 Programming Language

The programming language C0 has been developed in the course of the Verisoft project to be as expressive as needed while still retaining the simplicity to allow formal verification.
The syntax and semantics of C0 greatly resemble ANSI C. It is quite obvious that verifying correctness of an arbitrary C program is nearly impossible\(^1\) due to pointer arithmetics and the variety of permitted constructs. However, by introducing certain restrictions, the effort of verification is reduced to the point of manageability.

These restrictions include:

- no side-effects in statements (e.g. \(i++\))
- no function calls in expressions
- the size of arrays has to be known at compile time
- there has to be exactly one return statement at the end of a function body
- all local variable declarations must be placed at the beginning of a function body
- no pointer arithmetics
- no pointers to local variables
- no pointers to functions
- all pointers are typed, i.e. there are no void pointers

### 2.2.1 Types in C0

The language C0 offers four base types:

- **32-bit signed integers**: \(\text{int}\)
  
  Range: \([-2^{31}, \ldots, 2^{31} - 1]\)

- **32-bit unsigned integers**: \(\text{unsigned int}\)
  
  Range: \(\{0, \ldots, 2^{32} - 1\}\)

- **Boolean**: \(\text{bool}\)
  
  Range: \(\{\text{true}, \text{false}\}\)

- **8-bit signed integers**: \(\text{char}\)
  
  Range: \(\{-128, \ldots, 127\}\)

Based on a simple type, the following complex types can be constructed:

---

\(^1\)See [http://www.ioccc.org](http://www.ioccc.org) for interesting code examples.
• typed pointers
  t *a;

• arrays, for example
  t a[100];

• structs, for example
  struct s { t data }; struct s a;

2.2.2 Statements

C0 offers the following statements:

• Assignment
  l = expr;
  Assigns the result of the expression expr to the variable l. Type correctness is enforced.

• While loop
  while(cond) { statements };
  The semicolon-separated set of statements is executed until the boolean expression cond becomes false.

• Conditional
  if(cond) { statements };
  if(cond) { statements } else { statements2 };
  In case the condition cond (which has to be of type bool) is true, the series of statements statements is executed. If the else branch is specified, the statements it contains will be executed if cond is false.

• Function call
  l = f(x1,...,xn);
  The type of l has to coincide with the return value type of the function f. As well, the parameters x1,...,xn have to be of the respective types given in the declaration of f.

• Return
  return expr;
  This statement has to be the last statement in a function body. The type of expr has to match the return value type of the function containing the statement.
- **Dynamic memory allocation**

  \[ l = \text{new}(t); \]

  This statement allocates a new variable of type \( t \) on the heap, storing a pointer to it in the variable \( l \) (which therefore has to be of type \( t^* \)).

### 2.3 Interactive Theorem Proving in Higher Order Logic

Over the past decades, there have been many approaches to computer-aided theorem proving. Automated theorem proving, probably the most important subfield of automated reasoning, deals with proving mathematical theorems by a computer program. Now, there is a multitude of different logics and axiom systems in which proofs for theorems can be inferred deductively applying inference rules. However, in general, the more generic and useful the system, the harder full automation of theorem proving gets. For example, propositional logic (which is simple predicate logic without quantifiers), is still decidable while being NP-complete. The use of such a logic in practice is, however, limited. First order logic, a predicate logic with quantifiers, is no longer decidable, but still recursively enumerable. That is, if there is a proof for a given theorem it will be found eventually, given unbounded resources. If, however, there is no proof, a first order theorem prover may (if unable to disprove the theorem) fail to terminate while searching for a proof. In higher order logic, allowing quantification over functions, the situation gets even worse with respect to automation.

Even though many automated theorem provers are able to prove even some more complex theorems, there is always a theorem that is too complex to handle considering the limited amount of resources. Since, essentially, automated theorem proving is the act of searching for a derivation of the theorem by applying simple rules in a sensible way, it is still quite clear that for complex theorems the search depth can easily become too deep to prove or disprove the theorem in time.

This limitation of automated theorem provers quite naturally leads to the concept of interactive theorem proving. Instead of trying to get the program to prove the theorem all by itself, the user essentially tells the prover those proof steps the theorem prover seems to be unable to figure out. Such an interactive proof environment is provided by Isabelle, a generic theorem prover, which was developed at Cambridge University and TU Munich [1]. Isabelle supports several different logics, however we will only consider Isabelle/HOL, the higher order logic framework embedded in Isabelle.

We continue with the very basics of lambda calculus underlying higher order logic. Afterwards, we introduce Isabelle/HOL by giving the minimal set of inference rules it is based on, commenting on some, in practice, helpful rules.
2.3.1 Lambda Calculus and ML Definitions

Every logic is based on a calculus. This calculus is the language we use to formalize and represent mathematical properties and lemmata. The calculus underlying Isabelle/HOL can be described as a mix of simply typed lambda calculus and a variant of the functional language ML (metalanguage). We introduce the basic concept of lambda calculus and proceed with a brief introduction to Isabelle’s ML syntax.

Lambda expressions

Lambda calculus is built around the central concept of \( \lambda \)-expressions. A \( \lambda \)-expression is a term of the form

\[
\lambda x. M
\]

where \( M \) is a term that can contain \( x \) as a free variable. In this context, we understand a free variable to be an identifier that is not bound by a \( \lambda \)-term. Essentially, such a lambda expression \( \lambda x. M \) can be considered to describe a function that takes an argument \( x \) and returns \( M \).

Given a countably infinite set of identifiers \( I \) (e.g. \( \{a, b, c, d, \ldots, z, x_1, x_2, x_3, x_4, \ldots\} \)), the set of all \( \lambda \)-expressions can be described by the following context-free grammar:

- \( E ::= I \)
- \( E ::= (\lambda I. E) \) (Abstraction)
- \( E ::= (E E) \) (Application)

Equivalences

For \( \lambda \)-expressions, we define three equivalence relations, \( \alpha \)-equivalence, \( \beta \)-equivalence, and \( \eta \)-equivalence.

\( \alpha \)-equivalence

This relation establishes equivalence between \( \lambda \)-expressions that have the same structure, only differing in the choice of binders in the \( \lambda \)-terms. That is, we consider two terms \( \alpha \)-equivalent if we can get them to be the same by merely renaming variables bound by \( \lambda \)-expressions. We denote \( \alpha \)-equivalence as

\[
\lambda x. M \equiv_{\alpha} \lambda y. M[y/x]
\]
where $M$ is an expression that does not contain $y$ as a free variable. $M[y/x]$ denotes the expression we obtain when replacing all free occurrences of $x$ in $M$ with $y$.

**β-equivalence**

Using β-equivalence, we resolve application. When we apply a λ-expression to another, we can often simplify the resulting term. We define β-equivalence $\equiv_\beta$ such that

\[(\lambda x. M) N \equiv_\beta M[N/x]\]

When we apply $(\lambda x. M)$ to $N$, we get $M[N/x]$, the term we obtain after replacing all free occurrences of $x$ in $M$ by $N$, as a result.

**η-equivalence**

η-equivalence concerns functional extensionality, that is, we consider two functions the same if and only if they give the same result for all arguments. We denote this equivalence

\[\lambda x. F x \equiv_\eta F\]

where $F$ is a λ-expression that does not contain $x$ as a free variable.

**αβη-equivalence**

We define $\equiv_{\alpha\beta\eta}$ as the smallest equivalence relation closed under $\equiv_\alpha$, $\equiv_\beta$, and $\equiv_\eta$.

**Lambda Calculus**

It is possible to encode the natural numbers and give functions that perform any operation on natural numbers using nothing but λ-expressions. It is well-known that λ-calculus and the Turing machine are equivalent in their computing capabilities. However, going deep into technical detail here is beyond the scope of this introduction.

To increase the expressive power of λ-calculus, we introduce the common logic constants (which can all be encoded using λ-expressions):

- **True, False**
- **¬** Negation
- **∨, ∧** Disjunction and conjunction
- **→** Implication
- **∃** Existential quantification
- **∀** Universal quantification
- **=** Equality
Russel’s Paradox

Now, this representation framework is actually so expressive that it is inconsistent. Considering the computational equivalence between the Turing machine and $\lambda$-calculus, it is intuitively clear that there is some undecidable problem in $\lambda$-calculus corresponding to Turing’s halting problem.

Consider the following $\lambda$-term:

$$ R := \lambda x. \neg(x \ x) $$

We can interpret $R$ to be the characteristic function of the set of all sets that do not contain themselves:

$$ \{ x \mid x \not\in x \} $$

This is the case since every set can be uniquely described by its characteristic function.

Now, let us consider the expression $R \ R$:

$$ R \ R = (\lambda x. \neg(x \ x)) \ R \equiv_\beta \neg(R \ R) $$

and also

$$ \neg(R \ R) = \neg((\lambda x. \neg(x \ x)) \ R) \equiv_\beta \neg(R \ R) = R \ R $$

What we have here is a boolean $\lambda$-expression which is $\beta$-equivalent to its negation. That means, if $R \ R$ holds, $\neg(R \ R)$ also holds, since the expressions are equivalent, leading to a contradiction. In our set interpretation of $R$, this expresses the fact that we cannot decide whether the set containing all sets that do not contain themselves contains itself.

Simply-Typed $\lambda$-Calculus

Any logic calculus is required to be sound, that is, anything we can derive in it is actually considered valid in respect to the underlying semantics. Thus, since we do not consider True equivalent to False, we need to avoid inconsistencies like Russel’s Paradox. We can avoid these inconsistencies by using simple types for all variables and enforcing type correctness on all terms.

There are essentially two kinds of types, base types and functional types. Given a set of base types $B = \{ t_1, t_2, \ldots, t_n \}$, a simple type can be constructed in the following BNF:

- $T ::= B$
• T ::= (T → T)

For base types, we will, among others, consider the type bool of booleans (True : bool and False : bool), the type nat of natural numbers, and the type int of integers.

We assign simple types to any λ-term:

- **Typed Variables**, x : α
- **Typed Constants**, C : α
- **Application**, ((F : α → β) (M : α)) : β
- **Abstraction**, (λx : α. M : β) : α → β

where α and β are simple types.

Let us consider the term R = λx.¬(x x) from Russel’s Paradox. Assume there exists a simple type α such that R : α. Since we have an abstraction here, we know that actually, α has to be a functional type of the form (β → γ) for some simple types β and γ. We know that x : β and ¬(x x) : γ. Since ¬ is a constant of type (bool → bool), we know that γ = bool. Also, since we apply ¬ to the term (x x), we know that (x x) has to be of type bool. Since (x x) : bool is an application, we know that x has to be of some functional type δ → bool and also x has to be of type δ. Now, however, it is not possible to find a simple type δ such that δ = δ → bool. Thus, we cannot assign a type to R.

The advantage of simply typed λ-calculus is not only that we avoid inconsistence. In simply typed λ-calculus, there exists a unique normal form in respect to βη-equivalence for every typed λ-term. These normal forms are essential for term unification.

**Metalanguage**

Robin Milner and others developed the functional programming language ML in the late 1970s as metalanguage for developing proof tactics in the LCF theorem prover [5]. As a functional language it adheres to the fundamental idea of lambda calculus, that is, it implements the concept of procedural abstraction and application.

ML style syntax is used in Isabelle/HOL mainly to define new datatypes and to specify recursive functions for later use. For a basic introduction to working with Isabelle/HOL and its syntax, it is highly advisable to read the book “A Proof Assistant for Higher-Order Logic” [9].

Throughout this thesis, we mostly deal with specification, functions, and proof obligations in Isabelle/HOL, thus the definitions we give are closely related to the ML definitions used in Isabelle/HOL. We use functional style mathematical notation, i.e. instead of writing f(a, b, c) we write f a b c to denote application of the function f to the arguments a, b, and c.
Definition: List

The set $\alpha list$ of lists on a given type $\alpha$ is defined in the following way:

- The empty list $\text{nil}$ is an element of $\alpha list$
- Given an element $x : \alpha$ and a list $xs : \alpha list$, we can combine these to a new list by inserting $x$ as new head element in front of the list $xs$ using the so-called “cons”-operator $\#$:

$$x\#xs \in \alpha list$$

Functions on Lists

There are several predefined functions on lists that are used throughout the thesis.

The $@$-operator, used in the following functions, performs concatenation of two lists.

Reversing a list

The function $\text{rev} : \alpha list \to \alpha list$ takes a list $Ps$ and returns the reversed list $\text{rev} Ps$.

$$\text{rev} Ps = \begin{cases} \text{nil} & \text{if } Ps = \text{nil} \\ \text{rev } xs@ [x] & \text{if } \exists xs.\exists x. \text{Ps} = x\#xs \end{cases}$$

Applying a function to all list elements

The function $\text{map} : (\alpha \to \beta) \to \alpha list \to \beta list$ takes a function $f : \alpha \to \beta$ and a list $Ps : \alpha list$ and returns the list we obtain when applying the function $f$ to every element of $Ps$.

$$\text{map } f \ Ps = \begin{cases} \text{nil} & \text{if } Ps = \text{nil} \\ (f \ x)\#\text{map } f \ xs & \text{if } \exists xs.\exists x. \text{Ps} = x\#xs \end{cases}$$

Obtaining the set of elements of a list

The function $\text{set} : \alpha list \to \alpha set$ takes a list $Ps : \alpha list$ and returns the set of list elements of $Ps$.

$$\text{set} \ Ps = \begin{cases} \emptyset & \text{if } Ps = \text{nil} \\ \{x\} \cup (\text{set} \ xs) & \text{if } \exists xs.\exists x. \text{Ps} = x\#xs \end{cases}$$

Checking for distinctness of list elements

The function $\text{distinct} : \alpha list \to \text{bool}$ is a function that takes a list $xs : \alpha list$ and returns $\text{True}$ if and only if all elements of $xs$ are different from each other.

$$\text{distinct} \ Ps = \begin{cases} \text{True} & \text{if } Ps = \text{nil} \\ x \notin \text{set} \ xs \land \text{distinct} \ xs & \text{if } \exists xs.\exists x. \text{Ps} = x\#xs \end{cases}$$
Constants, such as the nil list, and any predefined functions, will always be typeset in sans serif style to distinguish them from bound or free variables. Variables we quantify over, such as the $x$ and $xS$ variables in the function definitions above, are typeset in true type font for better readability.

**Definition: The Hilbert Choice Operator**

Let $X$ be a set. The *Hilbert Choice Operator* for the set $X$ is a function,

$$\varepsilon : 2^X \setminus \{\emptyset\} \to X$$

which maps every non-empty subset $\emptyset \neq Y \subseteq X$ to an element $\varepsilon(Y) \in Y$.

In Isabelle/HOL, the Hilbert Choice Operator is represented by the `SOME` operator:

For a predicate $P \in \alpha \to \text{bool}$ (which can essentially be considered to be the characteristic function of a set on type $\alpha$), `SOME x. P x` denotes an element of type $\alpha$ that satisfies $P$ if there is such an element.

### 2.3.2 Higher Order Logic

Now that we have an underlying calculus, we can define a logic based on that calculus. Such a logic essentially is a sound and complete set of rules we can apply to infer valid statements in our calculus. Often, we will want to know whether a statement is true in a specific context, i.e. we want to prove $B$, assuming that $A$ holds. Therefore, we introduce a context to any given statement. If we can derive a statement in the empty context, we consider the statement valid. Such a context can be considered a list $\Delta$ of expressions of type `bool`.

**The basic rules of Isabelle/HOL**

The higher order logic implemented in Isabelle is based on a minimal set of inference rules [8]. We use the $\Rightarrow$ symbol to separate the context from the statement we try to derive. In a backward proof, we apply the rules in bottom-up fashion starting with the statement we want to prove. Often, in a given proof state, more than one rule is applicable. Most of these rules are never encountered in practice but rather applied automatically by Isabelle’s simplifier.

The rules can be read in the following way: If the part above the line holds, we can infer validity of the part below. On the right hand side of the rule, we annotate the name of the rule.

11
Assumption Rule

\[ \Delta, P \Rightarrow P \text{assumption} \]

In case we find the statement we want to prove to be part of the context, we can close the branch in our proof.

Reflexivity Rule

\[ \Delta \Rightarrow t = t \text{refl} \]

Also, when we find we are trying to prove an equality of the form \( t = t \), we consider this to be true.

Excluded Middle

\[ \Delta \Rightarrow P \lor \neg P \text{True_or_False} \]

This is a common classical rule which states that either \( P \) holds or \( \neg P \) holds, i.e. there are no other options.

Substitution Rule

\[ \Delta \Rightarrow s = t \quad \Delta \Rightarrow P \quad \Delta \Rightarrow P \text{subst}_s \]

This rule states that, when trying to prove \( P \), we can provide a term \( s \) and instead prove two things: One the one hand, we need to show that \( s = t \) holds in the same context \( \Delta \) we are trying to prove \( P \) in, and also, we need to show that \( P \) holds in \( \Delta \).

Functional Extensionality

\[ \Delta \Rightarrow \forall x. f \ x = g \ x \quad \Delta \Rightarrow \lambda x. f \ x = \lambda x. g \ x \text{ext} \]

The \( \text{ext} \)-rule deals with functional extensionality. We consider two functions equal if and only if they give the same result for all inputs.

Implication Introduction

\[ \Delta, P \Rightarrow Q \quad \Delta \Rightarrow P \Rightarrow Q \text{impI} \]

The implication introduction rule defines the way we go about proving implications. When we encounter an implication \( P \Rightarrow Q \), we continue by proving \( Q \) in the extended context \( \Delta, P \).
Modus Ponens

\[ \Delta \Rightarrow P \rightarrow Q \quad \Rightarrow P \rightarrow Q \]

The mp-rule (modus ponens) is a rule that, in general, requires user interaction. In the event that we are to prove a statement \( Q \) in a context \( \Delta \), we can instead prove the expression \( P \) and the implication \( P \rightarrow Q \) in the same context. Here, in general, we need to give the prover a hint by providing an appropriate term \( P \).

Equality and Equivalence

\[ \Delta \Rightarrow (P \rightarrow Q) \rightarrow (Q \rightarrow P) \rightarrow P = Q \iff \]

This rule simply states that two logically equivalent statements \( P \) and \( Q \) are actually equal with respect to equality for type \( bool \).

Hilbert Choice Introduction

\[ \Delta \Rightarrow P x \quad \Rightarrow P x \]

The rule \( \text{someI} \) reflects the defining property of the Hilbert choice operator \( \varepsilon \). It states that, when trying to show that there is an \( x \) such that \( P x \) holds, we can do so simply by providing a specific term \( x \) such that \( P a \) holds in the same context.

Derived Rules

For a complete list of rules derived from the basic rule set of Isabelle/HOL, see [8]. Here, we will just briefly comment on a few rules that were particularly helpful.

Conjunction Introduction

\[ \Delta \Rightarrow P \quad \Rightarrow \Delta \Rightarrow Q \]

Sometimes we encounter conjunctions of statements. This is where the \( \text{conjI} \) rule comes in. When we need to show that \( P \land Q \) holds in context \( \Delta \), we must show both that \( P \) holds in \( \Delta \) and that \( Q \) holds in \( \Delta \). In our proof obligations, we often find large conjunctions of statements. These usually are too complex for the simplifier to handle as a whole. We can usually see which of the statements cause trouble and thus, we can split them off using this rule, to treat them individually.
Disjunction Introduction

\[ \Delta \Rightarrow P \quad \text{disjI1} \quad \text{and} \quad \Delta \Rightarrow Q \quad \text{disjI2} \]

When trying to prove the disjunction \( P \lor Q \) in context \( \Delta \), we can decide to verify either that \( P \) holds in \( \Delta \) or \( Q \) holds in \( \Delta \). Sometimes it is wise to apply this rule by hand since we already encountered a case where automation led us into a “dead end”, i.e. a proof obligation that can no longer be fulfilled, even though a proof is possible when we choose the correct part of the disjunction.

Implication Elimination

\[ \Delta \Rightarrow P \rightarrow Q \quad \Delta \Rightarrow P \quad \Delta, Q \Rightarrow R \quad \text{impE} \]

This is a very useful rule considering the amount of implication we have to deal with in our proofs. On first glance, this rule looks more complicated than the previous ones, however, it is merely given in a general way. It states, that, when trying to prove \( R \) in context \( \Delta \), we can choose \( P \) and \( Q \) and decide to show several things instead. Usually, when we want to apply this rule, we often find our implication \( P \rightarrow Q \) to be part of our assumptions. In general, \( Q \) and \( R \) do not coincide. Thus, we need to prove that \( R \) holds in the extended context \( \Delta, Q \), i.e. \( Q \) implies \( R \). Additionally, we get a subgoal to show that \( P \) holds in \( \Delta \). Essentially, this rule is merely an extended version of the *modus ponens* rule.

Existential Quantification

The existential quantification operator \( \exists : (\alpha \rightarrow \text{bool}) \rightarrow \text{bool} \) is defined in the following way:

\[ \exists \equiv \lambda P. P(\varepsilon x. P \, x) \]

That is, it takes a predicate function \( P \) and returns true iff there is an \( x \) such that \( P \, x \) holds. In general, we abbreviate the application \( \exists(\lambda x. P \, x) \), writing \( \exists x. P \, x \) instead. The *somal*-rule yields the rule for existential quantification:

\[ \Delta \Rightarrow P \, x \quad \text{exI} \]

When we are to prove that there exists an \( x \) such that \( P \, x \) holds in context \( \Delta \), we can do so by choosing a specific \( x \) and showing that \( P \, x \) holds in \( \Delta \).
Substitution

\[ \Delta \Rightarrow t = s \quad \Delta \Rightarrow P s \]
\[ \Delta \Rightarrow P t s_{subst} \]

This is a rule that is particularly helpful in practice, since often we need to give the prover a hint in the form of the term \( s \) when trying to resolve complex statements. In particular, substitution rules are very helpful when trying to get the prover to accept equality between more complex arithmetic statements.

Negation Elimination

\[ \Delta \Rightarrow \neg P \quad \Delta \Rightarrow P \]
\[ \Delta \Rightarrow R \text{ notE} \]

During a proof, we can encounter a proof state where our assumptions contain a contradiction. This rule simply expresses that, if we can prove a contradiction in our assumptions, i.e. find a \( P \) such that we can prove \( P \) and \( \neg P \), \( R \) is also valid in the contradicting context \( \Delta \). The most obvious way to obtain such a proof state is by applying the proof by contradiction proof style.

Tactics

We interact with Isabelle’s theorem proving environment using so-called tactics. Such a tactic can essentially be considered a function that applies higher order logic rules in the attempt to resolve, simplify or modify a given subgoal. Isabelle/HOL offers many tactics, some of which are considered too low-level to be used explicitly by the user. There are higher level tactics that incorporate these low-level tactics. However, we will not go into much detail here, but instead just explain the higher level tactics and methods used in our proofs.

apply(\text{simp})

A fundamental tactic is the simplification tactic \text{simp}. Essentially said, simplification is one big part of the work the prover does for us. Term rewriting rules are applied in the hope that the proof obligation will become simpler, or, sometimes, completely eliminated. To manage this, the simplifier uses a certain set of simplification rules which are applied. We can give additional rules to the simplifier to make its application more powerful, but also more time consuming.

apply(\text{clarify})

The \text{clarify} method is another very useful tactic. \text{simp} and \text{clarify} go hand in hand to find the point in the proof where user interaction is actually necessary. As opposed to term rewriting, the clarification method applies all the obvious logical rules without splitting the current subgoal. Unsafe rules, i.e. rules that can render the goal unprovable, are not
applied. There also is another commonly used method which interleaves clarify and simp, the clarsimp method.

\textbf{apply}(\texttt{rule} \; \texttt{rule})

\textbf{apply}(\texttt{rule_tac} \; x=y \; \texttt{in} \; \texttt{rule})

This tactic allows us to apply a specific rule \texttt{rule} to the current subgoal. Using the second tactic, we can explicitely instantiate the free variables of the rule. If \( x \) is a free variable of the rule \texttt{rule}, we instantiate \( x \) with \( y \) and apply the rule to our subgoal. This tactic is particularly helpful with the application of the substitution rule \texttt{ssubst}. The free variables of \texttt{ssubst} are \( P \) and \( s \). Other rules that can be helpful to use explicitely using this tactic are \texttt{impE}, \texttt{exI} and \texttt{notE}.

\textbf{apply}(\texttt{induct_tac} \; n)

In the proofs of some lemmata, we use induction. The tactic \texttt{induct_tac} performs induction on \( n \). Note that, in this context, induction means structural induction and not necessarily induction on the natural numbers. In our proofs, we tend to perform induction on lists.

\textbf{apply}(\texttt{subgoal_tac} \; P)

The method \texttt{subgoal_tac} gives us a way to add things to our assumptions by proving them. That is, application of this method yields the current subgoal with the added assumption \( P \), and an additional subgoal that proves \( P \) under the current assumptions.

\textbf{apply}(\texttt{case_tac} \; P)

Another very useful method is case distinction on \( P \), the method \texttt{case_tac}. The method replaces the current subgoal by two subgoals, the one with added assumption \( P \), the other with added assumption \( \neg P \).

We have seen the basic structure of the logic Isabelle/HOL, introduced and explained some exemplary rules from the logic, and gave a brief introduction to Isabelle’s higher order logic tactics.

### 2.4 The Hoare Logic

Published in C. A. R. Hoare’s 1969 paper “An axiomatic basis for computer programming” [6], the concept of Hoare logic has since then been one of the main frameworks for imperative program verification. It provides a set of logical rules to reason about the correctness of computer programs in a mathematical sense. The big advantage of Hoare logic over, for example, model checking is the fact that even programs with large state spaces can be verified on all possible inputs. However, as opposed to model checking, there is no complete algorithm to prove or disprove that a certain program respects its specification.
The proofs in this thesis have been done using the Hoare logic environment of Isabelle/HOL implemented by Norbert Schirmer [10]. We lay down the basic framework and notation of Hoare logic, give an introduction to Hoare rules, show how to extend the state space to model pointers to structures on the heap, and, lastly, we will consider abstractions on heap structures using the example of the dList struct.

### 2.4.1 Basic Framework

As we want to argue formal correctness of a given program, we need a formal language to express what we expect the program to do. We can then check our formal specification against the actual program code. Fundamentally, a computer program is a series of statements which manipulate certain data in a predictable way. Now, the basic idea underlying the concept of Hoare logic is the fact that actually, all properties and consequences of program execution can be inferred deductively from the code.

We want to express that the execution of certain code modifies the program state in a specific way. Thus, we need to define what we understand to be a program state.

**Definition: Basic State space**

The *state space* $\Sigma$ of a given program without pointer-variables is the collection of

- all variables of base types,
- all individual components of structures, and
- all function parameters.

To represent the state space, we will use a *record*. Such a record contains all the variables of our state space. Let us consider a simple C0 program example:

```c
struct pair {  
  int first;  
  int second;  
};

struct pair p;  
bool x;

int combine(struct pair x) { return x.first + x.second; }

int main() { combine(p); return 0; }
```
The corresponding set $\Sigma$ of all program states can be characterized by the following record:

$$
\Sigma = \{ \ p\_first : int, \\
\ p\_second : int \\
\ x : bool \\
\ combine\_x\_first : int \\
\ combine\_x\_second : int \\
\ res\_combine : nat \\
\ res\_main : int \} 
$$

The global struct variable $p$ is flattened, resulting in the two individual fields of the struct, $p\_first$ and $p\_second$. The global variable $x$ corresponds to the variable $x$ in the state space. The combine function parameter struct variable $x$ is not only flattened, but also named according to the function to prevent name clashes, resulting in the two state space variables $combine\_x\_first$ and $combine\_x\_second$. Every function in C0 has a return value and thus, there are $res\_combine$ and $res\_main$ for their results. Now, you may notice that we match the bounded type $int$ to the unbounded type of integer numbers $int$ (and respectively, unsigned $int$ to $nat$). The inductive definition of $nat$ and the definition of $int$ using $nat$ allow us to employ standard proof techniques like induction which simply would not work on a bounded set of integers or natural numbers. This, however, leaves the proof obligation of showing that we never assign a value which is out of bounds to a variable. In our proof environment, this can be done by annotating the program with so-called guards [10]. In this work, we do not consider guards.

An individual program state $\sigma \in \Sigma$ provides an instantiation for any variable of the state space. We will use bold text to denote variables of the state space throughout the paper. Given a state $\sigma \in \Sigma$ and a program state variable $var$, we will denote the value of $var$ in state $\sigma$ by $^\sigma var$. To express changes to a program state $\sigma \in \Sigma$, we introduce the notation $\sigma[x := y]$ to describe the program state we obtain when we take $\sigma$ and simply change the value of $x$ to $y$.

**Hoare Triples**

From a state space point of view, we can interpret any program as a binary relation on program states describing the states we can reach from any given state by executing the program. We will denote that we can reach state $\tau \in \Sigma$ from state $\sigma \in \Sigma$ by executing the program $p$ by writing $\sigma \ p \ \tau$.

Now, with a formalism to express properties about program states, we can specify what effect the execution of certain program code is supposed to have on the program state. For this, we use Hoare triples.
**Definition: Hoare triple**

A *Hoare triple* is of the general form

\[ A \ p \ B \]

where \( A \) and \( B \) are subsets of the state space \( \Sigma \) of the program \( p \). \( A \) is called precondition and \( B \) is called postcondition.

**Partial Correctness**

Such a triple is called *valid* with respect to *partial correctness* if and only if, starting in a state \( \sigma \in A \), by execution of \( p \), we can only reach states \( \tau \in B \).

\[ A \ p \ B \text{ is valid} \iff (\forall \sigma, \tau \in \Sigma. \sigma \ p \tau \land \sigma \in A \implies \tau \in B) \]

Now, this notion of validity does not consider the fact that the program \( p \) could be non-terminating on some or even all inputs. Partial correctness only tells us that, if we start in a state satisfying the precondition *and* if the program terminates, we end in a state \( \tau \) in \( B \).

**Total Correctness**

A Hoare triple is called *valid* with respect to *total correctness*, if and only if for all computations starting in a state \( \sigma \) satisfying the precondition \( A \), the program \( p \) terminates, resulting in a state \( \tau \) satisfying the postcondition \( B \).

### 2.4.2 Hoare Rules

A *Hoare logic* is a logical framework in which exactly all valid Hoare triples can be derived using the *Hoare rules* it is based on. The Hoare logic we use is not actually based on C0 but on an imperative language model implemented in Isabelle/HOL [10]. A translation mechanism has been established between the language C0 and Isabelle’s imperative language in the Verisoft project. Using this mechanism, we translate our C0 code to Isabelle’s imperative language, and then verify it against our program specifications given as Hoare triples in Isabelle/HOL.

We give Hoare rules for partial correctness of C0 programs, inductively defining a simple Hoare logic. We use the Isabelle/HOL set comprehension notation \( \{s. \ Q \ s\} \) that denotes the set of all \( s \) that fulfill the predicate \( Q \). A C0 program \( p \) is a correct implementation of its specification given by the precondition \( A \) and the postcondition \( B \), if and only if we can derive the Hoare triple \( A \ p \ B \).
Assignment

\[ \vdash \{ \sigma. \sigma[a := \text{expr}] \in B \} (a = \text{expr}) B \]

Given a postcondition \( B \) and an assignment of the form \( a = \text{expr} \), we know that \( B \) holds after execution of the assignment if the state \( \sigma \) before execution fulfills \( \sigma[a := \text{expr}] \in B \). That is, if we take the state \( \sigma \) and update the value of \( a \) to \( \text{expr} \), we obtain a state that fulfills the postcondition \( B \).

Sequential Composition

\[ \vdash A \ p_1 \ B' \quad \vdash B' \ p_2 \ B \]

\[ \vdash A (p_1; p_2) B \]

Given two parts of a series of statements, \( p_1 \) and \( p_2 \), the Hoare triple \( A (p_1; p_2) B \) of the sequential composition of \( p_1 \) and \( p_2 \) is valid iff there exists a \( B' \subseteq \Sigma \) such that \( B' \) is a valid precondition for \( p_2 \) with the postcondition \( B \), and also, \( B' \) is a valid postcondition for \( p_1 \) with the precondition \( A \).

Conditional

\[ \vdash (A \cap \text{cond}) \ c_1 \ B \quad \vdash (A \cap \neg \text{cond}) \ c_2 \ B \]

\[ \vdash A (\text{if } \text{cond} \text{ then } \{c_1\} \text{ else } \{c_2\}) B \]

To show that the Hoare triple \( A (\text{if } \text{cond} \text{ then } \{c_1\} \text{ else } \{c_2\}) B \) of the conditional statement is valid, we need to show that \( A \cap \text{cond} \), the subset of states of \( A \) in which \( \text{cond} \) holds, is a sufficient precondition for \( B \) after the execution of \( c_1 \), and also, that \( A \cap \neg \text{cond} \) is a sufficient precondition for \( B \) after execution of \( c_2 \).

Rule of Consequence

\[ A \subseteq A' \quad \vdash A' \ P \ B' \quad B' \subseteq B \]

\[ \vdash A \ P \ B \]

Now, if, for a program \( P \), we derived the valid Hoare triple \( A' \ P \ B' \), it is quite obvious that, if we take a more strict precondition \( A \), and a less strict postcondition \( B \), the Hoare triple \( A \ P \ B \) will also be valid.

While Loop

\[ \vdash (I \cap \text{cond}) \ p \ I \]

\[ \vdash I (\text{while} \{\text{cond}\} \{p\}) (I \cap \neg \text{cond}) \]

Dealing with the while loop, we need to give an invariant \( I \). This invariant has to hold before execution of the while loop, before each individual execution of the while loop’s
body \( p \), and also after the loop. If we can show that \( I \cap \text{cond} \), the set of all states of \( I \) in which \( \text{cond} \) holds, is a sufficient precondition for the postcondition \( I \) after \( p \), we conclude that \( I \) is a sufficient precondition for the while loop and the postcondition \( I \cap \neg \text{cond} \).

Function Call
Due to the possibility of recursive function definitions in the complete syntax of C0, the hoare rules dealing with function calls are nontrivial. However, in this introduction to Hoare logic, we will only consider non-recursive functions, since any recursion can be expressed using while loops and variables on the heap. Therefore, we will actually only need the following Hoare rule:

\[
\frac{}{A (f_{x_1} = x_1; \ldots; f_{x_n} = x_n; p; x = \text{res}_f) B} 
\]

When we encounter a non-recursive function call of the form \( x = f(x_1, \ldots, x_n) \), we know what happens is that the input variables \( f_{x_1}, \ldots, f_{x_n} \) of the function \( f \) are assigned the values we call the function with. Then, the function body \( p \) is executed. The return statement at the end of the function body stores the result of the function call in \( \text{res}_f \). So all that is left to do is to assign this result to \( x \). Note that we have to watch out for mutually recursive function calls, too, since we have the same problem of overwriting our input variables there.

With the possibility of recursive function calls, things get quite a bit more involved. However, the functions of the big number package that have been verified during this thesis do not contain recursive function calls.

2.4.3 Formalization of Heap Structures
Since we deal with memory allocation and dynamic heap structures, we have to consider how these concepts can be represented in our verification environment.

Pointers and memory allocation
Dynamic memory allocation gives us the option to implement structures that reside on the heap, such as lists. To be able to describe these structures, we need an accurate abstraction. Pointers in Isabelle/HOL are not represented by numerical addresses, but by the abstract type \( \text{ref} \). This type is isomorphic to the natural numbers, containing a constant \( \text{Null} \) representing the null pointer. Considering the lack of pointer arithmetic in C0, it is easy to see that the type \( \text{ref} \) indeed can be used to model pointers in C0.

To obtain a fresh location on the heap, we define the function \( \text{new} : \text{ref set} \rightarrow \text{ref} \) as follows:

\[
\text{new } A \equiv \text{SOME } a. \ a \notin (\{\text{Null}\} \cup A)
\]
Given a set \( A \), the operator will return a new location \( a \), if there exists such a location, that is different from the already allocated locations in \( A \) and not the Null location. It is quite obvious that the behaviour of this constant is different from the way the new operator in C0 behaves. In our verification context, we always deal with finite sets \( A \) of allocated locations. Thus, considering there are infinitely many available locations in our model, the Hilbert Choice Operator always returns a valid new location. However, this inherently does not account for limited memory space. There actually is an approach to memory management in Isabelle/HOL via the use of a counter for unused references (which is decreased when memory is allocated and increased when it is freed again). However, this model is not quite applicable to C0 since deallocation is not handled by the programmer, but by the garbage collection algorithm. In this thesis, we do not explicitly address the fact that the new operator can indeed return the null pointer when there is no space left on the heap.

**Structs on the heap**

Obviously, a model of pointers does little good unless the value or struct a location contains can be accessed. We model the heap via functions from references to values, as it has been first introduced in the basic model by Burstall [4].

The heap can contain structured values, structs in C0. Following the model of Bornat [2], there is one heap function \( f : \text{ref} \rightarrow t \) introduced to the state space for every field \( f \) of type \( t \) of the struct.

Let us consider a simple list struct in C0:

```c
struct node {
    int value;
    struct node *next;
}
```

We add the functions \( \text{value} : \text{ref} \rightarrow \text{int} \) and \( \text{next} : \text{ref} \rightarrow \text{ref} \) to our abstract state space. Thus, given a location \( a \) which is supposed to contain a node, we can obtain the location of the next element of the list by applying the next heap function to \( a \). In case we want to obtain the value of a node at location \( a \), we simply apply the value function to \( a \) and obtain the integer value of the value field of the struct.

This model allows us to describe any struct that can be expressed in C0.

### 2.4.4 Heap Abstractions and the dList Predicate

With a model of the heap, we can express arbitrary properties about the heap state using higher order logic predicates. Handling simple structs not containing pointer fields
becomes trivial. However, many common heap datastructures, e.g. lists, consist of individual building blocks containing references to other building blocks of the same data structure. We consider how to establish the relation of such a datastructure on the heap and our abstract concept of it at the example of a list.

What makes a list a list?

Consider the following C0 struct:

```c
struct node {
    int value;
    struct node *next;
}
```

We declare a pointer variable `head`:

```c
struct node *head;
```

With any programming background, it is quite natural to simply start to consider this structure to represent a singly-linked list. The pointer variable `head` appears predestined to denote a list starting in the node `*head`. However, we should be aware that not every datastructure we can create on the heap using this `struct` is a list. If, for example, we create a new node on the heap and instantiate the `struct`’s fields in the following way,

```c
head = new (struct node);
head->next = head;
```

we obtain a circular datastructure which does not represent a list.

In our C0 code and thus, in our heap representation, the concept of lists is a purely abstract one. On the heap, there is merely a bunch of `node` structures. Thus, we need to establish a relation between our abstract list and the heap. As we are reasoning in the environment of Isabelle/HOL, our abstract list will be a `ref list` containing the locations we reach via the `next` heap starting from `head`, including our initial location. Still, observing that we have a `value` field in our `struct`, why do we consider the abstract list to be of type `ref list` and not of type `int list`? The reason is actually quite simple. If, for example, we want to express that two abstract lists have no list elements in common (which in turn corresponds to the lists on the heap not sharing list elements), we easily can do that using the `ref list` representation. On the other hand, if we do need the actual list of values, we can simply apply the `value` function to the elements of our abstract list. Obviously, the `ref list` representation is more flexible.

All that is left to do is to bridge the gap between the heap representation of the list and our abstract one using predicates in higher order logic. We will introduce a predicate for each abstract datastructure we want to represent with our implementation. Such a predicate
will, given the necessary heap functions, a heap location \( a \), and abstract data \( d \), yield True if that datastructure is present at location \( a \) representing the abstract value \( d \), and False otherwise.

**Paths on the heap**

Concerning lists, one fundamental property we want to check for is whether there is a path between two given references \( x \) and \( y \), connected via a heap function \( \text{next} \), yielding a list of locations \( \text{Ps} \). To establish this relation, we define the recursive function \( \text{Path} : \text{ref} \rightarrow (\text{ref} \rightarrow \text{ref}) \rightarrow \text{ref} \rightarrow \text{ref list} \rightarrow \text{bool} \) in the following way:

\[
\text{Path } x \text{ next } y \text{ Ps } = \begin{cases} 
  x = y & \text{if } \text{Ps} = \text{nil} \\
  x = p \land x \neq \text{Null} \land \text{Path}(\text{next } x) \text{ next } y \text{ ps } & \text{if } \exists p. \exists ps. \text{Ps} = p#ps
\end{cases}
\]

Given a list \( \text{Ps} \), two locations \( x \) and \( y \), and a heap function \( \text{next} \), the function returns True iff, starting at location \( x \), we obtain the list \( \text{Ps} \) by moving along the \( \text{next} \) heap until we reach \( y \). Note that \( y \) does not belong to the list \( \text{Ps} \). In case our list \( \text{Ps} \) is the empty list, the locations \( x \) and \( y \) have to coincide, since otherwise, we would get a non-empty list if we recorded all locations until we reach \( y \). In case \( \text{Ps} \) is not empty, there exists a first element \( p \) of \( \text{Ps} \) and the tail of the list \( \text{ps} \). We expect that \( x \) has to be the same as the first element \( p \) of the list and also, \( x \) cannot be the Null constant since we do not dereference the Null location. If these two constraints hold, we know that we have a path between \( x \) and \( y \) yielding the list \( \text{Ps} \), iff there is, in turn a path between \( \text{next } x \) and \( y \), yielding the list \( \text{ps} \).

**Lists are a special case of Paths**

This concept of a path on the heap is already very close to what we commonly understand to be a list on the heap. We can recognize any list on the heap using the \( \text{Path} \) function. However, we do not yet exclude the kind of circular datastructure presented before. Now, if we define the predicate \( \text{List} : \text{ref} \rightarrow (\text{ref} \rightarrow \text{ref}) \rightarrow \text{ref list} \rightarrow \text{bool} \) as follows,

\[
\text{List head next ps } = \text{Path head next Null ps}
\]

we disallow exactly these circular constructs. So, a list is actually just a path on the heap ending in the Null location.

**The dList Abstraction**

Finally, we consider the doubly-linked list (as it is used throughout our big number implementation). This kind of list is, like our singly-linked list, based on a struct representing its nodes:

\[
\text{struct node} \\
\{ \\
  \text{int value}; \\
  \text{struct node *next};
\}
\]
struct node *prev;
}

We represent and access such a list in C0 using pointers to its first and last element:

struct node *head;
struct node *last;

What we understand to be a doubly-linked list is a list that can be traversed in both directions, from the first element to the last one, following the next pointers of the nodes, but also from the last element to the first one, following the individual nodes’ prev pointers. Thus, it follows that head->prev = NULL and last->next = NULL, since otherwise, our traversal of the list would continue past what we consider to be the first and last element of the list.

As in [7], let us define the predicate dList : ref → (ref → ref) → (ref → ref) → ref → ref list → bool, which takes the locations head and last of the first and last element of the list, the heap functions next and prev, and an abstract list ls of type ref list, returning True iff the abstract list ls is represented by head and last via the heap functions:

dList head next prev last ls = List head next ls ∧ List last prev (rev ls)

So, we have the doubly linked list ls spanned between the locations head and last, if we have a List starting in head following the next heap representing the abstract list ls, and also, there is a List starting in last traversing the prev heap, resulting in the reversed list rev ls.
Chapter 3

The Big Integer Package

With this thesis, a big number package has been implemented and partially verified. This chapter deals with how we represent big numbers in C0 and HOL, lists the interface of the library we implemented, and, overall, considers the verification process of the big number package. We study abstractions used in HOL to describe big numbers and give specification and proof sketches for all covered functions. The functions have been specified and verified in Isabelle/HOL with respect to partial correctness, i.e. termination has not been proven. For each function, we outline the proof strategy that has been used. If necessary, special lemmata are given. We will not go into minute detail here, but rather give guidelines for proving formal correctness of these functions using an interactive proof assistant.

3.1 Representation of Big Integers

We represent big numbers (to a certain base \( b \in \mathbb{N}_{>1} \)) by a sign bit \( s \in \{0, 1\} \) and \( n \in \mathbb{N}_0 \) digits \( d_{n-1}, \ldots, d_0 \in \{0, 1, \ldots, b - 1\} \) with \( d_{n-1} \neq 0 \). The integer value of such an \( n \)-sized big number to base \( b \) is given as

\[
\langle s, d_{n-1}, \ldots, d_0 \rangle = (-1)^s \sum_{i=0}^{n-1} d_i \cdot b^i
\]

All 0-sized big numbers \( (n = 0) \) represent the value \( 0 \in \mathbb{Z} \).
In C0 we represent the digits $d_i$ with the following struct:

```c
struct bigint_digit
{
    struct bigint_digit *digit_next;
    struct bigint_digit *digit_prev;
    unsigned int value;
};
```

Here, `value` is the value of $d(i)$ and `digit_next` is a pointer to $d_{i+1}$ or the NULL pointer if $i = n - 1$. Accordingly, `digit_prev` is a pointer to $d_{i-1}$ or the NULL pointer if $i = 0$.

Our C0 representation of big numbers is then given by the following struct:

```c
struct bigint
{
    struct bigint_digit *first_digit;
    struct bigint_digit *last_digit;
    unsigned int size;
    bool sign;
};
```

Here, `first_digit` is a pointer to the least significant digit $d_0$, `last_digit` is a pointer to the most significant digit $d_{n-1}$, `size` is a variable that stores the number of digits, and `sign` stores the sign of the big integer (`true` for negative and `false` for positive).

Big numbers in C0 are always represented by pointer variables to `bigint` structures on the heap.

![Diagram of big number representation](image)

Figure 3.1: An example big number to base $b = 10$, representing the value $-239$

In our big number implementation in C0, we obviously have to consider big numbers to a specific base $b$.

```c
#define bigInt_base 65536u
```
We defined the constant `bigInt_base` in such a way that the multiplication of any big number digits will not cause an integer overflow, i.e. the result will fit into an unsigned integer variable.

### 3.2 Informal Specification of the Big Integer Package

We list the interface and additional functions belonging to the big number package. During the implementation effort of the package, we added several utility functions to reduce complexity of individual functions.

#### 3.2.1 Constant Definitions

The following constants are used throughout the fundamental algorithms and datastructures package of which the big number package is a part.

```c
#define NO_ERROR 0
#define ERROR_OUT_OF_MEMORY -1
#define ERROR_INVALID -3
#define ERROR_DIV_BY_ZERO -5
#define EQUAL 0
#define LESS 1
#define GREATER 2
```

#### 3.2.2 Public Interface

The public interface of the big number package contains the following functions:

- **Creation of a new big number**
  ```c```
  ```c
  struct bigint *bigIntNew();
  ```
  The function creates a new big integer whose value equals zero, if there is enough memory space left. Return values:
  - **NULL**: The function ran out of memory.
  - **otherwise**: The function returns a pointer to the newly created big integer.
• **Assignment: Integer to big number**

\[
\text{int bigNumAssignInt(struct bigint *bignum, int integer);}\\
\]

The function assigns the value of the integer `integer` to the big number `bignum`. Return values:

- `NO_ERROR`: The function finished successfully.
- `ERROR_OUT_OF_MEMORY`: The function ran out of memory.

• **Assignment: Big number to big number**

\[
\text{int bigNumAssignBigint(struct bigint *fstbignum,}\\
\text{ struct bigint *sndbignum);}\\
\]

The function assigns the value of the big integer `sndbignum` to the big integer `fstbignum`. Return values:

- `NO_ERROR`: The function finished successfully.
- `ERROR_OUT_OF_MEMORY`: The function ran out of memory.

• **Addition of two big numbers**

\[
\text{int bigIntAdd(struct bigint *op1, struct bigint *op2,}\\
\text{ struct bigint *result);}\\
\]

The function adds two big numbers `op1` and `op2` and assigns the result to the big number `result`. Return values:

- `NO_ERROR`: The function finished successfully.
- `ERROR_OUT_OF_MEMORY`: The function ran out of memory.

• **Unary minus**

\[
\text{int bigIntMinus(struct bigint *num);}\\
\]

The function inverts the sign bit of the big number pointed to by `num`. Return value:

- `NO_ERROR`: The function finished successfully.

• **Multiplication of two big numbers**

\[
\text{int bigIntMul(struct bigint *op1, struct bigint *op2,}\\
\text{ struct bigint *result);}\\
\]

The function multiplies two big numbers `op1` and `op2` assigning the result to the big number pointed to by `result`. Return values:

- `NO_ERROR`: The function finished successfully.
- ERROR_OUT_OF_MEMORY: The function ran out of memory.

- **Division of two big numbers**

  ```c
  int bigIntDiv (struct bigint *dividend, struct bigint *divisor, struct bigint *result);
  ```

  The function divides the big number `dividend` by `divisor`, rounds the quotient down to the next integer and assigns the result to the big number pointed to by `result`. Return values:

  - **NO_ERROR**: The function finished successfully.
  - **ERROR_OUT_OF_MEMORY**: The function ran out of memory.

- **Modulo function**

  ```c
  int bigIntMod (struct bigint *dividend, struct bigint *divisor, struct bigint *result);
  ```

  The function computes the remainder of dividing `dividend` by `divisor` and assigns it to the big number pointed to by `result`. Return values:

  - **NO_ERROR**: The function finished successfully.
  - **ERROR_OUT_OF_MEMORY**: The function ran out of memory.

- **Exponentiation modulo m**

  ```c
  int bigIntModExp (struct bigint *base, struct bigint *exponent, struct bigint *m, struct bigint *result);
  ```

  The function computes \(\text{base}^{\text{exponent}} \mod m\) and assigns the result to the big number pointed to by `result`. Return values:

  - **NO_ERROR**: The function finished successfully.
  - **ERROR_OUT_OF_MEMORY**: The function ran out of memory.

- **Comparing the values of two big numbers**

  ```c
  int bigIntCompare (struct bigint *first, struct bigint *second);
  ```

  The function compares the values of the two big numbers `first` and `second`. Return values:

  - **EQUAL**: The values of the two big numbers are the same.
  - **LESS**: The value of `first` is smaller than the value of `second`.
  - **GREATER**: The value of `first` is greater than the value of `second`.
3.2.3 Private Functions

To reduce the complexity of individual functions, several utility functions (which explicitly do not belong to the public interface of the package) have been implemented.

- Clearing a big number

```c
int bigNumClear(struct bigint *bignum);
```

This function sets the value of `bignum` to zero. Return values:

- NO_ERROR: The function finished successfully.

- Digit Insertion: Front

```c
int bigIntInsertDigitFront(struct bigint *bignum,
unsigned int digit);
```

This function inserts a new digit with the value of `digit` in front of the list belonging to `bignum`, that is, `digit` will be inserted as new most significant digit. This function assumes that `bignum` is a pointer to a valid big number. Return values:

- NO_ERROR: The new digit has successfully been created and inserted.
- ERROR_OUT_OF_MEMORY: The function ran out of memory.

- Digit Insertion: Back

```c
int bigIntInsertDigitBack(struct bigint *bignum,
unsigned int digit);
```

This function inserts a new digit with the value of `digit` at the end of the list belonging to `bignum`, that is, `digit` will be inserted as new least significant digit, effectively increasing the significance of the previous digits. Return values:

- NO_ERROR: The new digit has successfully been created and inserted.
- ERROR_OUT_OF_MEMORY: The function ran out of memory.

- In-place Addition of Big Numbers

```c
int bigIntAddLocal(struct bigint *a, struct bigint *b);
```

This function adds the absolute values of the big numbers `a` and `b`, storing the result in `a` while keeping the sign bit of `a`. This efficient in-place addition helps to speed up multiplication. Return values:

- NO_ERROR: The function finished successfully.
- ERROR_OUT_OF_MEMORY: The function ran out of memory.
• **Multiplication with a single digit**

```c
int bigIntDigitMul(struct bigint *bignum, unsigned int digit,
                    struct bigint *product);
```

This function multiplies the big number `bignum` and the digit `digit`, storing the result in `product`. Return values:

- **NO_ERROR**: The function finished successfully.
- **ERROR_OUT_OF_MEMORY**: The function ran out of memory.

• **Absolute value comparison**

```c
int bigIntAbsCompare(struct bigint *first, struct bigint *second);
```

The function compares the absolute values of the two big numbers `first` and `second`. Return values:

- **ERROR_INVALID**: This error is returned if `first` or `second` are the NULL pointer.
- **EQUAL**: The absolute values of the two big numbers are the same.
- **LESS**: The absolute value of `first` is smaller than the value of `second`.
- **GREATER**: The absolute value of `first` is greater than the value of `second`.

• **Division by a single digit**

```c
int bigIntDigitDiv(struct bigint *a, unsigned int b,
                    struct bigint *quotient, struct bigint *remainder);
```

This function divides the big number represented by `a` by the single digit `b`, storing the quotient in the big number `quotient` and the remainder in `remainder`. This function is used in the main division routine. Return values:

- **NO_ERROR**: The function finished successfully.
- **ERROR_OUT_OF_MEMORY**: The function ran out of memory.

• **Dividing three digits by two**

```c
unsigned int bigIntDiv3DigitsBy2(unsigned int a1, unsigned int a2,
                                 unsigned int a3, unsigned int b1, unsigned int b2);
```

This function takes five unsigned integer values, `a1`, `a2`, `a3`, `b1`, and `b2`. These are essentially interpreted as the two big numbers `< 0, a1, a2, a3 >` and `< 0, b1, b2 >`. The function returns the unsigned integer quotient:

```c
q = ⌊< 0, a1, a2, a3 > / < 0, b1, b2 > ⌋
```

iff all digits are in range and `< 0, b1, b2 >` is normalized (i.e. `b1 ≥ bigInt_base`). The quotient is used in the quotient estimate of the division routine. This function has been taken from
the research report on fast recursive division by Christoph Burnikel and Joachim Ziegler [3] which provides a good overview on the division quotient estimation procedure.

- **Subtraction removing only one leading zero**

  ```c
  int bigIntSubLocal(struct bigint *a, struct bigint *b);
  ```

  This function subtracts the big number represented by b from the big number a, preserving all but one leading zeroes of the result which is assigned to a. Obviously that means, a will not necessarily represent a big number but a big number with leading zeroes. This kind of subtraction is needed in the big number division function. Return values:

  - **NO_ERROR**: The function finished successfully.

- **Division and modulo routine**

  ```c
  int bigIntDivHelp(struct bigint *a, struct bigint *b,
                    struct bigint *quotient, struct bigint *remainder);
  ```

  This function simultaneously computes the quotient and remainder that occur when dividing a by b using the simple school division method and the quotient estimate from [3]. The division and modulo functions of the interface both call this function. If quotient or remainder are the NULL pointer, the respective result is not calculated. Return values:

  - **NO_ERROR**: The function finished successfully.
  - **ERROR_OUT_OF_MEMORY**: The function ran out of memory.

- **Removing leading zeroes**

  ```c
  int bigIntRemoveLeadingZeros(struct bigint *a)
  ```

  This function removes all leading zeroes of a, turning it from a big number with leading zeroes to a big number. The function always returns **NO_ERROR**.

### 3.3 Big Number Formalization in HOL

In this section, we will introduce the fundamental heap abstraction used to describe big numbers. Also, we will discuss a few helpful predicates that describe properties that very often need to be specified and proven. At last, we present a select few lemmas of general importance throughout the thesis.
3.3.1 Heap Abstractions

To give a formal definition of big numbers in higher order logic, we need to define several functions. Let us recall the struct used to define a big number in C0:

```
struct bigint
{
    struct bigint_digit *first_digit;
    struct bigint_digit *last_digit;
    unsigned int size;
    bool sign;
};
```

So, the fields `first_digit`, `last_digit`, `size`, and `sign` of the big number have to be represented in HOL. For this task, we have the following heap functions of our state space:

- `first_digit : ref → ref`
- `last_digit : ref → ref`
- `size : ref → nat`
- `sign : ref → bool`

These heap functions take a heap location and return the corresponding fields of the `bigint` struct residing at that location. We use the type `ref` to denote pointers in C0.

However, we can clearly see that these heap functions do not give a complete description of big numbers. So far, we know nothing about the contents of the digits referenced by `first_digit` and `last_digit`, nor do we know about the list structure on the heap. The struct used to represent the digits,

```
struct bigint_digit
{
    struct bigint_digit *digit_next;
    struct bigint_digit *digit_prev;
    unsigned int value;
};
```

yields the remaining heap functions

- `digit_next : ref → ref`
- `digit_prev : ref → ref`
• value : ref → nat

needed to represent the fields of the bigint_digit struct. This is a simple doubly-linked list structure, thus we will be able to utilize the dList predicate we introduced in Section 2.4.4 to define our big number predicate.

![Diagram of bigint_digit](image)

Figure 3.2: An example illustrating the connection between abstract heap and C0 heap representation of big numbers.

What we need is a predicate that returns True iff there is a big number with a certain list and value at a given location bn. With our dList predicate we establish a ref list of the actual list of digits on the heap represented by the first_digit and last_digit of our big number and the digit_next and digit_prev heap functions. Clearly, we can apply the value function to the elements of the list we obtained, obtaining a nat list of actual values of the digits. To establish equality between a big number and an integer value, we need two functions, dealing with the value of the digit list and the value of the sign:

- **Converting nat list to the according natural number**

  list2nat : nat list → nat

  list2nat nil = 0
  list2nat (x#xs) = x + BigInt_base · (list2nat xs)
This function takes a list of natural numbers ordered according to digit significance starting with the least significant digit as head element. It returns the sum of the products of the individual digits and powers of $\text{bigInt\_base}$ according to their significance:

$$\sum_{i=0}^{n-1} d_i \cdot \text{bigInt\_base}^i$$

- **Converting the boolean sign to the appropriate integer value**

  \[
  \text{bool2int} : \text{bool} \rightarrow \text{int} \\
  \text{bool2int} \ \text{False} = 1 \\
  \text{bool2int} \ \text{True} = -1
  \]

  For convenience, we define another predicate:

  - **The value of the digit is in the allowed range**

    \[
    \text{in\_range\_pos} : \text{nat} \rightarrow \text{bool} \\
    \text{in\_range\_pos} \ x \equiv x < \text{bigInt\_base}
    \]

Finally, we define the predicate that accepts any valid big number:

\[
\text{BigNumber} : \text{ref} \rightarrow (\text{ref} \rightarrow \text{ref}) \rightarrow (\text{ref} \rightarrow \text{ref}) \rightarrow (\text{ref} \rightarrow \text{nat}) \rightarrow (\text{ref} \rightarrow \text{bool}) \rightarrow (\text{ref} \rightarrow \text{ref}) \rightarrow (\text{ref} \rightarrow \text{ref}) \rightarrow (\text{ref} \rightarrow \text{nat}) \rightarrow \text{ref list} \rightarrow \text{int} \rightarrow \text{bool}
\]

\[
\text{BigNumber} \ \text{bn} \ \text{first\_digit} \ \text{last\_digit} \ \text{size} \ \text{sign} \ \text{digit\_next} \ \text{digit\_prev} \ \text{value} \ \text{ls} \ \text{num} \equiv
(\text{bn} \neq \text{Null} \land \\text{dList} (\text{first\_digit} \ \text{bn}) \ \text{digit\_next} \ \text{digit\_prev} (\text{last\_digit} \ \text{bn}) \ \text{ls} \land
\ \land \ \text{length} \ \text{ls} = \text{size} \ \text{bn} \land
\ \land (\forall x \in \text{set} \ \text{ls}. \ \text{in\_range\_pos} (\text{value} \ x)) \land
\ \land \ \text{bool2int} (\text{sign} \ \text{bn}) \cdot \text{int} (\text{list2nat} (\text{map} (\lambda y. \text{value} \ y) \ \text{ls})) = \text{num} \land
\ \land (\text{first\_digit} \ \text{bn} \neq \text{Null}) = (\text{last\_digit} \ \text{bn} \neq \text{Null}) \land
\ \land (\text{last\_digit} \ \text{bn} \neq \text{Null} \rightarrow \text{value} \ (\text{last\_digit} \ \text{bn}) \neq 0)
\]

This predicate takes three variables, $\text{bn}$, $\text{ls}$, and $\text{num}$, and all heap functions of relevance. The predicate yields $\text{True}$ if and only if there is a big number with the list of references $\text{ls}$ and the value $\text{num}$ at the location $\text{bn}$.

So, obviously, we do not consider any structure residing at the Null location a big number. The $\text{dList}$ predicate tells us whether there is a bidirectional path resulting in the list $\text{ls}$ via the $\text{digit\_next}$ and $\text{digit\_prev}$ functions between our first and last digits. Another property that needs to be guaranteed is that the size field of our big number contains the actual length of the list $\text{ls}$. Also, all digits of the big number have to be in range, i.e. be smaller than the chosen base. Then, obviously, we have to establish equality between the value associated with the list $\text{ls}$ and the integer $\text{num}$. Thus, $\text{ls}$ is converted to a list of natural numbers (by mapping the $\text{value}$ function on $\text{ls}$) which in turn converted to a natural number via the function $\text{list2nat}$. Multiplying with the sign of $\text{bn}$, we obtain the value
of our big number. Additionally, it is helpful to know that either both first_digit and last_digit of bn are equal to Null, or both are not. At last, we expect that, if there is at least one digit, the value of the most significant digit of bn is greater than zero. This constraint prevents leading zeroes.

Still, to specify and verify the correctness of several functions, we need another predicate to describe big numbers with leading zeroes. These are used in the division routine. Obviously, this predicate is obtained by simply removing the last constraint of our BigNumber predicate:

BigNumberLZ : ref → (ref → ref) → (ref → ref) → (ref → nat) → (ref → bool) → (ref → ref) → (ref → ref) → (ref → nat) → ref list → int → bool

BigNumberLZ bn first_digit last_digit size sign digit_next digit_prev value ls num = (bn ≠ Null) ∧ dList (first_digit bn) digit_next digit_prev (last_digit bn) ls ∧ length ls = size bn ∧ (∀ x ∈ set ls. in_range_pos (value x)) ∧ bool2int (sign bn)-int(list2nat (map (λ y. value y) ls)) = num ∧ (first_digit bn ≠ Null) = (last_digit bn ≠ Null)

3.3.2 Helpful Predicates

With an abstraction of big numbers, we can express many properties about big numbers in a uniform way. However, there are several other issues that can be addressed in a standardized way. With big numbers, and heap structures in general, often, we want to express that big number structures on the heap apart from the one at our specific location have not been changed at all.

Therefore, we introduce an additional predicate:

- Other big numbers have not been changed

OtherBigIntsUnchanged : ref list → (ref → bool) → (ref → bool) → (ref → nat) → (ref → ref) → (ref → ref) → (ref → ref) → (ref → ref) → (ref → ref) → bool

OtherBigIntsUnchanged bns signb signa sizeb sizea first_digitb first_digita last_digitb last_digita = (∀ bn:ref. bn ∉ set bns → (signa bn = signb bn ∧ sizea bn = sizeb bn ∧ first_digita bn = first_digitb bn ∧ last_digita bn = last_digitb bn))

Given a list of references bns supposed to be locations of big numbers, we want to express that only the structs at these locations may have been changed. To express this notion of before and after, this predicate requires two sets of heap functions, one from the state
before and one from the one after. Obviously, if these heap functions coincide on all big numbers not contained in our list, we guarantee that at most the big numbers at the locations of our list have been changed.

However, even though the heap functions associated with the bigint structure have not been modified, the value of our big number may have changed due to manipulations to the list structure associated with \textit{first_digit} and \textit{last_digit}. Therefore, we want to express that all digits except some specific exceptions have not been changed.

- Other lists have not been changed

\begin{align*}
\text{OtherListsUnchanged} : & \text{ref list} \rightarrow (\text{ref} \rightarrow \text{ref}) \rightarrow (\text{ref} \rightarrow \text{ref}) \rightarrow (\text{ref} \rightarrow \text{ref}) \rightarrow (\text{ref} \rightarrow \text{ref}) \rightarrow (\text{ref} \rightarrow \text{ref}) \rightarrow (\text{ref} \rightarrow \text{nat}) \rightarrow (\text{ref} \rightarrow \text{nat}) \rightarrow (\text{ref list}) \rightarrow \text{bool} \\
\text{OtherListsUnchanged} & \text{liselems nexta nextb preva prevb valuea valueb alloc} \\
= (\forall \text{el} : \text{ref}. \text{el} \notin \text{set liselems} \land \text{el} \in \text{set alloc} \rightarrow (\text{nexta} \text{ el} = \text{nextb} \text{ el} \\
\land \text{preva} \text{ el} = \text{prevb} \text{ el} \\
\land \text{valuea} \text{ el} = \text{valueb} \text{ el}))
\end{align*}

Given a list of locations \text{liselems} of big number digits, we state that the digit structures at all allocated locations differing from those contained in the list have not been changed. Again, we need two digit heap states between which the relation needs to be expressed. The careful reader might notice that here, in our list element preservation predicate, we require that all individual digits are allocated, whereas we do not consider allocation in the predicate for the bigint \textit{struct}. Since we can only modify the values of heap functions on specific references, it is easy to see that it is not actually necessary to require allocation. However, we do not need to consider preservation of not allocated structures, thus we can safely add this constraint here.

Often it is necessary to prove that two locations are not equal. In particular, we often need to prove that a newly allocated location is different from a particular, previously allocated one. This is one of the main reasons we keep a list of allocated references \text{alloc}. It will prove helpful to have a predicate that describes the allocation changes between two particular states and their respective allocation list:

- Allocation of new references

\begin{align*}
\text{NewAlloc} : & \text{ref set} \rightarrow \text{ref list} \rightarrow \text{ref list} \rightarrow \text{bool} \\
\text{NewAlloc} & \text{newelems oldalloc newalloc} \\
= \text{newelems} \cup \text{set oldalloc} = \text{set newalloc} \land \text{newelems} \cap \text{set oldalloc} = \emptyset
\end{align*}

The predicate takes a set of supposedly newly allocated elements \text{newelems}, the list of allocated references of the initial state \text{oldalloc}, and the new list of allocated references
newalloc of the final state. Given these, we express that the new elements and the previously allocated references are allocated in the final state. We could have expressed this notion of allocated elements staying allocated using the subset property

\[(\texttt{newelems} \cup \texttt{set oldalloc}) \subseteq \texttt{set newalloc}\]. However, this subset property is not a good choice since we implicitly allow the possibility of newalloc containing additional references we know nothing about. In some proofs, this lack of precision caused unforeseen problems which led to this final definition of the NewAlloc predicate. The other property we guarantee via the predicate is that the elements of \texttt{newelems} are all fresh, not previously allocated references.

These three predicates helped considerably to make individual proof states easier to read and understand while also reducing the complexity of expressions given to the Isabelle/HOL simplifier.

### 3.3.3 Generic Big Number Lemmata

We present three lemmata that are used in almost every proof of the thesis.

#### Preservation of Big Number Lists on the Heap

**Lemma** OtherListsUnchanged_intersect_empty:

**Shows** \(\text{(sizea bn = sizeb bn) \land \text{set Ls} \cap \text{set ls} = \emptyset \land \text{set ls} \subseteq \text{set alloc}}\) \land \text{OtherListsUnchanged Ls d_next d_nexta d_prev d_preva value valuea alloc} \land \text{BigNumber bn first_digit last_digit sizea sign d_next d_prev value ls num)} \rightarrow \text{BigNumber bn first_digit last_digit sizeb sign d_nexta d_preva valuea ls num}\)

This lemma allows us to show that the big number at location bn has not been changed even though there were list changes which did not affect it. Assuming the existence of a big number with list ls and value num at the location bn, we additionally require that the size of bn has not been changed, that the list ls of our big number bn contains only allocated elements, and that no element of ls has been changed, i.e. the intersection of the list of possibly changed list elements Ls and ls is empty.

The proof of this lemma in Isabelle/HOL is lengthy compared to those of other lemmata. While there are no overly complex statements, we have to deal with many trivialities that cannot be automatically resolved. These result from the fact that we need to completely expand the BigNumber definition down to the Path abstraction level. We have to individually consider all parts of the big number definition concerning our updated heap functions d_preva, d_nexta, and valuea.
Preservation of the Big Number Structure on the Heap

lemma OtherBigIntsUnchanged_bn_notin_list:
  shows \((bn \not\in \text{set } xs \land \text{OtherBigIntsUnchanged } xs \text{ signb signa sizeb sizea firstb firsta lastb lasta} \land \text{BigNumber } bn \text{ firstb lastb sizeb signb d_next d_prev value ls num}) \rightarrow \text{BigNumber } bn \text{ firsta lasta sizea signa d_next d_prev value ls num}\)

This is a simple but useful lemma concerning the preservation of big numbers in respect to the bigint struct’s heap functions. Given a list \(xs\) of locations representing big numbers which may have changed and a big number location \(bn\) with corresponding value \(num\) and list \(ls\), we know that our big number at location \(bn\) has not been changed as long as \(bn\) is not one of the locations contained in \(xs\). The lemma is trivially proven by giving the definitions of our predicates to the automated theorem prover.

The list2nat-value of appended Lists

lemma list2nat_split:
  shows \(\text{list2nat } (xs@ys) = \text{list2nat } xs + \text{bigInt_base length } xs \cdot \text{list2nat } ys\)

We show that given two lists \(xs\) and \(ys\), the value of applying list2nat to their concatenation is the same as evaluating the right hand expression. This is a particularly important lemma in many proofs considering that we often go through a big number list using a pointer, splitting the list at that location. That leaves us with three parts of the list, the part before the location, the location itself, and the remaining part of the list after that location. This lemma can be proven by induction on \(xs\) and application of the commutativity and distributivity rules for natural numbers.

3.4 bigIntNew

Creation of a new big number

struct bigint *bigIntNew();

The function creates a new big integer whose value equals zero, if there is enough memory space left. Return values:

- NULL: The function ran out of memory.
- otherwise: The function returns a pointer to the newly created big integer.
## 3.4.1 Implementation

First, we call the `new` operator asking for a pointer to a new big number. In case, `new` does not return `NULL` but an actual pointer, we set first and last digit of our big number to `NULL`, yielding a big number with an empty list. Afterwards we set `sign` and `size` of our new big number and return the pointer.

```c
struct bigint* bigIntNew()
{
    struct bigint *bignum;
    bignum = new (struct bigint);
    if (bignum != NULL)
    {
        bignum->first_digit = NULL;
        bignum->last_digit = NULL;
        bignum->sign = false;
        bignum->size = 0u;
    }
    return bignum;
}
```

Listing 3.1: Source code of `bigIntNew`

## 3.4.2 Specification

This function modifies `first_digit`, `last_digit`, `size`, `bignum`, `sign`, and `alloc`.

Let $\sigma \in \Sigma$ be the program state before execution of the function.

There is no precondition for this function, we specify this function for the most general case. We claim that, after execution of

$$\text{bignum} = \text{bigIntNew}()$$

the following postcondition holds:

$$\text{Postcondition of bigIntNew}$$

$$\begin{align*}
\text{BigNumber } & \text{bignum } \text{first_digit } \text{last_digit } \text{size } \text{sign } \text{digit_next } \text{digit_prev } \text{value} \text{ nil } \text{0} \\
\wedge & \text{NewAlloc } \{ \text{bignum} \}^\sigma \text{ alloc alloc} \\
\wedge & \text{OtherBigIntsUnchanged } [\text{bignum}] \text{ sign }^\sigma \text{ size }^\sigma \text{ first_digit }^\sigma \text{ last_digit }^\sigma \text{ last_digit }^\sigma
\end{align*}$$

Since we do not consider memory allocation here, the returned reference `bignum` is not the Null constant and we can expect that there is a zero-valued big number with empty list at that location. Since we know that the function changes the heaps `first_digit`, `last_digit`,
size, and sign, we need to express that all big numbers not residing at the location bignum remain unchanged. Since this function allocates memory the alloc list, we use the NewAlloc predicate to express that the location bignum has been newly allocated. None of the list heap functions have been changed, thus we do not need to use the OtherListsUnchanged predicate since the modifies lemma already covers the fact that the lists stay the same.

3.4.3 Verification

This function is trivially verified by expanding the definitions of BigNumber and OtherBigIntsUnchanged.

3.5 bigNumClear

Clearing a big number

int bigNumClear(struct bigint *bignum);

This function sets the value of bignum to zero. Return values:

- NO_ERROR: The function finished successfully.

This is a utility function used in many other functions.

3.5.1 Implementation

First we check if bignum is the NULL pointer. If it is, we return the ERROR_INVALID error. If not, we clear the size and then set the two list pointers first_digit and last_digit to NULL, turning bignum to a zero-valued big number.

```c
1 int bigNumClear(struct bigint *bignum)
2 {
3   int retval;
4   bignum->size = 0u;
5   bignum->first_digit = NULL;
6   bignum->last_digit = NULL;
7   retval = NO_ERROR;
8 }
```
return retval;
}

Listing 3.2: Source Code of bigNumClear

3.5.2 Specification

This function modifies \texttt{retval}, \texttt{size}, \texttt{first_digit}, and \texttt{last_digit}.

Let \( \sigma \in \Sigma \) be the state before execution of the function.

<table>
<thead>
<tr>
<th>Precondition of \texttt{bigNumClear}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{bignum} \neq \texttt{Null}</td>
</tr>
</tbody>
</table>

If the precondition is fulfilled in state \( \sigma \), we claim that after execution of

\( \texttt{retval} = \text{bigNumClear}(\texttt{bignum}) \)

the following postcondition holds:

<table>
<thead>
<tr>
<th>Postcondition of \texttt{bigNumClear}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{retval} = \texttt{NO_ERROR} &amp; \texttt{size}(\sigma \texttt{bignum}) = 0 &amp; \texttt{first_digit}(\sigma \texttt{bignum}) = \texttt{Null} &amp; \texttt{last_digit}(\sigma \texttt{bignum}) = \texttt{Null} &amp; \texttt{OtherBigIntsUnchanged}(\sigma \texttt{bignum}) &amp; \texttt{sign} &amp; \texttt{sign} &amp; \texttt{size} &amp; \texttt{size} &amp; \texttt{first_digit} &amp; \texttt{first_digit} &amp; \texttt{last_digit} &amp; \texttt{last_digit}</td>
</tr>
</tbody>
</table>

Our postcondition concerning the big number at location \( \sigma \texttt{bignum} \) is essentially equivalent to the predicate application

\( \text{BigNumber} \ \sigma \texttt{bignum} \ \sigma \texttt{first_digit} \ \sigma \texttt{last_digit} \ \sigma \texttt{size} \ \sigma \texttt{sign} \ \sigma \texttt{size} \ \sigma \texttt{first_digit} \ \sigma \texttt{first_digit} \ \sigma \texttt{last_digit} \ \sigma \texttt{last_digit} \ \sigma \texttt{nil} \ \sigma \texttt{0} \)

in the sense that Isabelle’s automated theorem prover can solve this equivalence by itself. However, as this is a utility function it is often more useful to have the definition already expanded and simplified in the postcondition.

3.5.3 Verification

This function is trivially verified applying the definition of \texttt{OtherBigIntsUnchanged}.
3.6 bigIntMinus

Unary minus

int bigIntMinus(struct bigint *num);

The function inverts the sign bit of the big number pointed to by num.

• NO_ERROR: The function finished successfully.

3.6.1 Implementation

In case num is not the NULL pointer, we simply invert the sign field of the struct.

```c
int bigIntMinus(struct bigint *num)
{
    int retval;
    retval = NO_ERROR;
    num->sign = !num->sign;
    return retval;
}
```

Listing 3.3: Source Code of bigIntMinus

3.6.2 Specification

This function modifies retval and sign.

Let num ∈ int, and let Ls ∈ ref list and let σ ∈ Σ be the state before execution of the function.

Precondition of bigIntMinus

<table>
<thead>
<tr>
<th>BigNumber bignum</th>
<th>first_digit</th>
<th>last_digit</th>
<th>size</th>
<th>sign</th>
<th>digit_next</th>
<th>digit_prev</th>
<th>value</th>
<th>Ls</th>
<th>num</th>
</tr>
</thead>
</table>

We claim that, if the precondition was fulfilled in the initial state σ, after execution of

retval = bigIntMinus(bignum),

the following postcondition will hold:

44
Postcondition of \texttt{bigIntMinus}

\begin{verbatim}
retval = NO_ERROR ∧ BigNumber bignum first_digit last_digit size sign digit_next digit_prev value Ls (-num) ∧ OtherBigIntsUnchanged [bignum] sign σ size σ first_digit σ first_digit last_digit σ last_digit
\end{verbatim}

The expected behavior of this function is to return NO\_ERROR, changing the sign of our big number at location \texttt{bignum}. We guarantee that no other big numbers were changed during the procedure.

3.6.3 Verification

This function is verified with a simple case split on the \textit{sign} of \texttt{bignum} and giving the constant definitions to the automated theorem prover.

3.7 \texttt{bigIntCompare}

Comparing the values of two big numbers

\begin{verbatim}
int bigIntCompare(struct bigint *first, struct bigint *second);
\end{verbatim}

The function compares the values of the two big numbers \texttt{first} and \texttt{second}. Return values:

- \texttt{EQUAL}: The values of the two big numbers are the same.
- \texttt{LESS}: The value of \texttt{first} is smaller than the value of \texttt{second}.
- \texttt{GREATER}: The value of \texttt{first} is greater than the value of \texttt{second}.

3.7.1 Implementation

The algorithm works as follows:

- check for NULL pointers
- if both \texttt{a} and \texttt{b} represent the value zero, return \texttt{EQUAL}
- otherwise: at least one of the numbers is non-zero, if their sign is different return the appropriate return value
• otherwise: their sign is the same, we start to go through both lists at the same time
starting from the most significant digit, comparing the current digits until (a) we
encounter a pair of different digits or (b) the lists end

```c
int bigIntCompare(struct bigint *a, struct bigint *b)
{
    int result;
    struct bigint_digit *current_a, *current_b;

    result = EQUAL;

    if(a->first_digit == NULL && b->first_digit == NULL)
    { result = EQUAL; } // their values are both zero
    else // at least one of the numbers is non-zero
    {
        if(a->sign != b->sign) // their signs differ
        {
            if(a->sign)
            { result = LESS; }
            else
            { result = GREATER; }
        }
        else // they are of the same sign
        {
            if(a->size != b->size) // their size differs
            {
                if(!a->sign)
                {
                    if(b->size < a->size)
                    { result = GREATER; }
                    else
                    { result = LESS; }
                }
                else
                {
                    if(a->size < b->size)
                    { result = GREATER; }
                    else
                    { result = LESS; }
                }
            }
            else // they have the same sign and size,
            { // thus, we have to go through the lists
```
current_a = a->last_digit;
current_b = b->last_digit;

if(current_a != NULL && current_b != NULL)
{
    if(current_b->value < current_a->value)
        { result = GREATER; }
    else
        {
            if(current_a->value < current_b->value)
                { result = LESS; }
        }

while(current_a->digit_prev != NULL
    && current_b->digit_prev != NULL
    && result == EQUAL)
{
    current_a = current_a->digit_prev;
    current_b = current_b->digit_prev;

    if(current_b->value < current_a->value)
        { result = GREATER; }
    else
        {
            if(current_a->value < current_b->value)
                { result = LESS; }
        }
}
if(a->sign)
{
    if(result == GREATER)
        { result = LESS; }
    else
        {
            if(result == LESS)
                { result = GREATER; }
        }
}
return result;

Listing 3.4: Source Code of bigIntCompare
3.7.2 Specification

This function modifies result, current_a, and current_b.

Let Ls1, Ls2 ∈ ref list and let num1, num2 ∈ int. Let σ ∈ Σ be the program state before execution of the function.

Precondition of bigIntCompare

BigNumber a first_digit last_digit size sign digit_next digit_prev value Ls1 num1
∧ BigNumber b first_digit last_digit size sign digit_next digit_prev value Ls2 num2

Function call: result = bigIntCompare(a, b);

Postcondition of bigIntCompare

result = GREATER ∧ (num2 < num1)
∨ result = LESS ∧ (num1 < num2)
∨ result = EQUAL ∧ (num2 = num1)

The formal specification of this function is simple. The reason for this is that this function does not modify the heap state in any way. The only variables that are changed are the temporary variables current_a and current_b and the return value result. Thus, we do not need to argue about the heap state in the postcondition.

It is quite clear that with some programming discipline we can make sure that this function is only called when there actually are valid big numbers present at locations a and b with respective values num1 and num2. After execution of the function we claim that the return value result reflects the relation between the big numbers a and b. That is, result is GREATER if the value num1 of a is greater than the value num2 of b, result is LESS if num1 is less than num2, and result is EQUAL if the values are equal.

3.7.3 Verification

This is the first function of the big number package including a while loop, and thus, formal verification of this function becomes quite a bit more involved. However, we do not alter the heap state in this function, resulting in a comparatively simple invariant for the while loop of this function. We will treat this function in more detail than the following functions, considering it is the first function containing a while loop.

Since this function contains one while loop, invoking the verification condition generator yields three subgoals. The first subgoal deals with the part of the function before the while loop. The second subgoal considers the while loop, and the last subgoal deals with the part after leaving the while loop.

We first present and explain the invariant used in our proof. Afterwards, we consider the three individual subgoals.
The Invariant

Invariant for the while loop in `bigIntCompare`

\( (\exists Lsa \ Lsb \ Lsa1 \ Lsb1. Ls1 = Lsa[@[current_a]@Lsa \land Ls2 = Lsb[@[current_b]@Lsb \land \\
\text{length } Lsa = \text{length } Lsb \land \text{length } Lsb1 = \text{length } Lsa1 \land \\
\text{map value } Lsa = \text{map value } Lsb) \land \\
\text{BigNumber } a \ \text{first_digit last_digit size sign digit_next digit_prev value } Ls1 \ \text{num1} \land \\
\text{BigNumber } b \ \text{first_digit last_digit size sign digit_next digit_prev value } Ls2 \ \text{num2} \land \\
\text{sign } a = \text{sign } b \land \\
((\text{result } = \text{LESS} \land \text{value current_a } < \text{value current_b}) \lor \\
(\text{result } = \text{GREATER} \land \text{value current_b } < \text{value current_a}) \lor \\
(\text{result } = \text{EQUAL} \land \text{value current_a } = \text{value current_b})) \)

In our while loop we simultaneously advance our `current_a` and `current_b` pointers along the `digit_prev` heap. We also know that, in case we reach the loop, our lists have the same length. Thus, we know that we can split the lists `Ls1` and `Ls2` in the following way:

\[
Ls1 = Lsa[@[current_a]@Lsa \quad \text{and} \quad Ls2 = Lsb[@[current_b]@Lsb}
\]

with lists `Lsa`, `Lsa1`, `Lsb`, and `Lsb1` such that `length Lsa = length Lsb` and `length Lsa1 = length Lsb1`. Also, we know that the values of the elements of the lists `Lsa` and `Lsb` must be equal, since we already passed these elements without leaving the loop.

Another thing we need to maintain through the while loop is our information about the big numbers at locations `a` and `b`, since, after the loop, we need this information to establish the validity of the postcondition. Additionally, we know that we only reach the loop in case the sign of our big numbers is equal. This, also, is needed later, when leaving the while loop.

At last, we want to establish that, before processing the next step of the loop, either, the current digits are the same and the `result` is `EQUAL`, or they differ and `result` reflects this in the correct way.

The first subgoal

The first subgoal deals with the program text before we encounter the while loop. As we can see clearly by looking at the source code, we do not always enter the while loop. We only reach the while loop if

- both numbers are non-zero,
- the signs of the numbers are the same, and
- the lengths of the lists of the numbers are the same.
In case we do not reach the while loop, we were able to decide the relation between the
two numbers using simple means, and we have to prove that the postcondition holds.
The proof structure here is closely related to the if-statement structure in the program
text. The case that both the locations $a$ and $b$ contain a zero-valued big number is solved
automatically. Thus, we consider the case where at least one of these numbers is non-zero. The if-statement in line 15 of the program text was translated to a case split by the
verification condition generator. We will now consider the case $\text{sign } a \neq \text{sign } b$.

Before we can proceed in our proof, we need a small lemma. What we need to know is,
that, in case we have a big number with a non-empty list, the value of that big number is
non-zero. In our BigNumber predicate we had the constraint that $(\text{last_digit } bn) \neq \text{Null} \rightarrow \text{value (last_digit } bn) \neq 0$. Thus, we know that the value of the most significant digit is always non-zero, if there is such a digit. Now we want to prove that list2nat applied to the
list of such a big number returns a strictly positive value:

```
lemma list2nat_Positive_rule:
  shows Ls=xs@[x] ∧ 0 < f x ⇒ 0 < list2nat (map f Ls)
```

The lemma is proven using induction on $xs$. With this lemma we can prove that, if we
have two non-empty big numbers with different signs, their values differ.

Let us return to the proof. We are in the case that at least one of our big numbers $a$ and
$b$ is non-zero and their signs differ. We perform a case-distinction on $\text{last_digit } a$(which
essentially is a case distinction on the fact whether the big number at $a$ is zero or not):

**Case: last_digit $a$ = Null:**
In this case, we know that the big number represented by $b$ must be non-zero, thus
$\text{last_digit } b \neq \text{Null}$. We are in a situation where we can expand the BigNumber defini-
tion of $b$ and use our lemma to prove that the value of $b$ is non-zero, validity of the
postcondition follows trivially by transitivity rules.

**Case: last_digit $a$ $\neq$ Null:**
This case is resolved in essentially the same way as the previous one. The only difference
is, that here, we know that the number represented by $a$ is non-zero.

As we are done with the case where the signs of the numbers differ, we continue with the
case where the signs are the same and at least one of the numbers is non-zero. Again, the
structure of if-statements provide us with a natural case distinction on the lengths of the
lists. We will now consider the case where the lengths of the lists $Ls1$ of $a$ and $Ls2$ of $b$
differ.

For this part of the proof, we need another lemma. Since we know that the most signifi-
cant digit of a non-zero big number is always non-zero, we need to prove that, for two big
numbers with the same sign, the one with the shorter list is the smaller one. To manage
this, we need to show upper and lower bounds for the list2nat values of big number lists.
Since we know that the most significant digit of a big number is non-zero, there is a trivial
lower bound for a big number list of length \( n \), namely the constant \( \text{bigInt}_\text{base}^{n-1} \). This fact can be recognized by the theorem proving environment in an automated fashion applying the list2nat_split rule. The upper bound for the value of a big number list, however, cannot be inferred this easily. Thus, we prove the following lemma:

**lemma** list2nat_upper_bound:

**shows**  
(\( \forall z \in \text{set } xs \). in_range_pos z) \( \rightarrow \) list2nat xs \( \leq \) bigInt_base^{length xs - 1}

This lemma is proven by induction on \( xs \) and applying transitivity and distributivity rules. Again, there are two symmetric cases:

**Case:** \( \text{size } b < \text{size } a \):

In this case, we know that, since the length of \( a \)'s list is greater than that of \( b \)'s list, \( a \) is definitely a non-zero big number, thus \( \text{last_digit } a \neq \text{Null} \). Since the most significant digit of \( a \) is not the Null location, using a dList lemma, we know that there exists a list \( ps \) such that \( Ls1 = ps@[\text{last_digit } a] \). Thus, invoking list2nat_split, we know that the list2nat value of \( Ls1 \) is greater than or equal to \( \text{bigInt}_\text{base}^{\text{length } Ls1-1} \). Applying our upper bound lemma, we know that:

\[
\text{list2nat } (\text{map value } Ls2) \leq \text{bigInt}_\text{base}^{\text{length } (\text{map value } Ls2) - 1} < \text{bigInt}_\text{base}^{\text{length } Ls2} \leq \text{bigInt}_\text{base}^{\text{length } Ls1-1} \leq \text{value}(\text{last_digit } a \cdot \text{bigInt}_\text{base}^{\text{length } Ls1-1} \leq \text{list2nat } (\text{map value } Ls1)
\]

Having established this, the postcondition follows automatically.

**Case:** \( \text{size } a < \text{size } b \):

We conduct the proof in an analogous way.

The next case we have to consider is the one where the signs of the big numbers are the same, and, additionally, the lengths of their lists are equal. We are in the case that we reach the while loop, thus we need to prove that the invariant holds under these conditions. First off, we know from our precondition that we have big numbers at locations \( a \) and \( b \) with lists \( Ls1 \) and \( Ls2 \) and values \( \text{num1} \) and \( \text{num2} \). Thus, this part of the invariant is trivially true. The part of the invariant about the sign of \( a \) and \( b \) also is trivial. At last, the part about the \textbf{result} value is again proven easily by automated means. This leaves us with the core part of the invariant, the fact that we can split the lists of our big numbers in the specified way. Proving an existential statement is done by giving an appropriate instantiation. Since \( \text{current}_a \) and \( \text{current}_b \) point to the most significant digit of the respective number, we know that both \( Lsa \) and \( Lsb \) have to be instantiated with the nil list which also reduces the map value \( Lsa = \text{map value } Lsb \) condition to triviality. Using the \textbf{BigNumber} and, most important, dList definitions and the fact that the lists are non-empty since the numbers are non-zero, we know that lists \( Lsa1 \) and \( Lsb1 \) exist, such that \( Ls1 = Lsa1@[\text{current}_a] \) and \( Ls2 = Lsb1@[\text{current}_b] \). Since we know that we only reach
the loop if the length of the lists is the same, it is trivially true that length \( Lsa_1 = Lsb_1 \).

The second subgoal, proving the invariant

Here, we have to show that the invariant is preserved under execution of the body of the while loop. Essentially we can interpret this as the invariant being both precondition and postcondition for the while loop’s body. However, there is another property belonging to our precondition here, and that is, that the condition of the while loop yields \( \text{True} \), since otherwise we would not continue with the while loop. In general, we will use the symbol \( \tau \) to refer to the state after execution of the while loop and use non-annotated variables for the state before execution.

In the body of the while loop, we advance our current list pointer, \( \text{current}_a \) and \( \text{current}_b \), along the \( \text{digit}_{\text{prev}} \) heap and set the result variable according to the values residing at these new locations. Looking at the program text, we see that \( \tau_{\text{current}_a} = \text{digit}_{\text{prev}} \text{current}_a \) and \( \tau_{\text{current}_b} = \text{digit}_{\text{prev}} \text{current}_b \). Thus, also from the program text, it follows easily that the postcondition for the \( \tau \) variable holds.

Since \( \text{digit}_{\text{prev}} \text{current}_a \neq \text{Null} \) and \( \text{digit}_{\text{prev}} \text{current}_b \neq \text{Null} \), we can prove that there exist lists \( ps \) and \( psa \) such that:

\[
Lsa_1 = ps[\text{digit}_{\text{prev}} \text{current}_a] \quad \text{and} \quad Lsb_1 = psa[\text{digit}_{\text{prev}} \text{current}_b]
\]

Thus, we can split the lists

\[
Ls_1 = \underbrace{ps@[\text{digit}_{\text{prev}} \text{current}_a]}_{Lsa_1'}@\underbrace{\text{current}_a}@Lsa \quad \text{and} \quad Ls_2 = \underbrace{psa@[\text{digit}_{\text{prev}} \text{current}_b]}_{Lsb_1'}@\underbrace{\text{current}_b}@Lsb
\]

in the desired way. This completes the proof of the second subgoal.
The third subgoal

When we leave the while loop, we know that both the invariant and the negation of the
while loop condition hold. Thus, there exist lists $Lsa, Lsa1, Lsb$, and $Lsb1$ such that:

\[
\begin{align*}
Ls1 &= \text{map value } Lsa1[@current_a] \oplus Lsa \\
Ls2 &= \text{map value } Lsb1[@current_b] \oplus Lsb
\end{align*}
\]

with $\text{length } Lsa = \text{length } Lsb$, $\text{length } Lsa1 = \text{length } Lsb1$, and most notably, $\text{map value } Lsa = \text{map value } Lsb$. Also, we know that the result reflects the relation between $current_a$ and $current_b$.

What we need to prove is that result also reflects the relation between the values $num1$ and $num2$ of $a$ and $b$. Considering the different possibilities for result and the signs of $a$ and $b$, the proof involves a fair amount of case distinction. The case $\text{result} = \text{EQUAL}$ is rather trivial. Concerning the other cases, let us study one representative case instead of discussing them all. Assume that we are in the case that $\text{result} = \text{LESS}$, the sign of both numbers is positive, and thus, the value of $current_a$ is less than that of $current_b$. What we have to show is that $\text{num1} < \text{num2}$ holds. Since the signs are positive, we know that

\[
\begin{align*}
\text{num1} &= \text{int(list2nat(map value Ls1))} \\
\text{num2} &= \text{int(list2nat(map value Ls2))}
\end{align*}
\]

Applying the list2nat_split rule, we obtain that

\[
\begin{align*}
\text{num1} &= \text{int(list2nat(map value Ls1) + (bigInt_base}^{\text{length } Lsa1}(\text{value } current_a + \text{bigInt_base}^{\text{length } Lsa1, \text{value } Lsa}))} \\
\text{num2} &= \text{int(list2nat(map value Ls1) + (bigInt_base}^{\text{length } Lsb1}(\text{value } current_b + \text{bigInt_base}^{\text{length } Lsb1, \text{value } Lsb}))}
\end{align*}
\]

Since $\text{length } Lsa1 = \text{length } Lsb1$, the inequality $\text{num1} < \text{num2}$ simplifies to

\[
\text{list2nat(map value Ls1) + bigInt_base}^{\text{length } Lsa1, \text{value } current_a} < \text{list2nat(map value Ls1) + bigInt_base}^{\text{length } Lsa1, \text{value } current_b}
\]

Applying our upper bound lemma and the fact that $\text{value } current_a < \text{value } current_b$, we get

53
\[ \text{num1} = \text{list2nat} \left( \text{map} \ value \ \text{Lsa1} \right) + \text{bigint}_\text{base}^\text{length} \text{Lsa1}\cdot \text{value current_a} \]
\[ \leq \text{bigint}_\text{base}^\text{length} \text{Lsa1} - 1 \]
\[ < \text{bigint}_\text{base}^\text{length} \text{Lsa1} + \text{bigint}_\text{base}^\text{length} \text{Lsa1}\cdot \text{value current_a} \]
\[ = \text{bigint}_\text{base}^\text{length} \text{Lsa1}\cdot (\text{value current_a} + 1) \]
\[ \leq \text{bigint}_\text{base}^\text{length} \text{Lsa1}\cdot \text{value current_b} \]
\[ \leq \text{list2nat} \left( \text{map} \ value \ \text{Lsb1} \right) + \text{bigint}_\text{base}^\text{length} \text{Lsa1}\cdot \text{value current_b} \]
\[ = \text{num2} \]

We have shown that \( \text{num1} < \text{num2} \) which completes our proof.

\[ \square \]

3.8 \textbf{bigIntInsertDigitFront}

Digit Insertion: Front

\[
\text{int bigIntInsertDigitFront(struct bigint *bignum, unsigned int digit);}\]

This function inserts a new digit with the value of \( \text{digit} \) in front of the list belonging to \( \text{bignum} \), that is, \( \text{digit} \) will be inserted as new most significant digit. This function assumes that \( \text{bignum} \) is a pointer to a valid big number. Return values:

- **NO_ERROR**: The new digit has successfully been created and inserted.
- **ERROR_OUT_OF_MEMORY**: The function ran out of memory.

3.8.1 Implementation

We create a new \text{bigint_digit} and store the pointer to it in the temporary variable \text{new_digit}.

If memory allocation was successful, and thus, \text{new_digit} is not the NULL pointer, we set the value of the new digit to \text{digit}. Then, in case there already is a non-empty list, we simply insert the \text{new_digit} after the \text{last_digit} of our big number using the \text{dlist_InsertAfter} function, set the \text{last_digit} pointer of \text{bignum} to our new digit, and also increment the size of our big number. If, however, there is merely the empty list present, we insert our \text{new_digit} as the only digit by making the \text{first_digit} pointers of \text{bignum} point to it and setting the size of \text{bignum} to 1.

If memory allocation failed, and thus, \text{new_digit} is the NULL pointer, we return the \text{ERROR_OUT_OF_MEMORY} error.
int bigIntInsertDigitFront(struct bigint *bignum, unsigned int digit) {
    int retval;
    struct bigint_digit *new_digit;
    retval = NO_ERROR;

    new_digit = new(struct bigint_digit);
    if(new_digit != NULL) {
        new_digit->value = digit;

        if(bignum->last_digit != NULL) {
            retval = dlist_InsertAfter(bignum->last_digit, new_digit);
            bignum->size = bignum->size + 1u;
            bignum->last_digit = new_digit;
        } else {
            bignum->last_digit = new_digit;
            bignum->first_digit = new_digit;
            new_digit->digit_next = NULL;
            new_digit->digit_prev = NULL;
            bignum->size = 1u;
        }
    } else {
        retval = ERROR_OUT_OF_MEMORY;
    }
    return retval;
}

Listing 3.5: Source Code of bigIntInsertDigitFront

3.8.2 Specification

Even though this function does not contain a while loop, the formal verification of this function is still more complicated than that of the previous functions without loops. The complexity shows mainly in the formal specification of the function. This is due to the fact that we change the heap state in a nontrivial way. Inserting a digit as most significant digit to a big number is a fundamental operation used in many functions throughout our big number package. Thus, we already had the chance to put the formal specifi-
tion of this function to effective use in the proofs of other functions, namely the functions bigNumAssignInt, bigNumAssignBigInt, bigIntAddLocal, and bigIntDigitMul. Another point where some complexity enters the proof is the use of the function dList_InsertAfter.

This function modifies retval, digit_next, digit_prev, first_digit, last_digit, value, size, and alloc.

Let \( Ls \in \text{ref list} \) and \( \text{num} \in \text{int} \). Let \( \sigma \in \Sigma \) be the program state before execution of the function.

**Precondition of bigIntInsertDigitFront**

\[
\begin{align*}
\text{set } Ls & \subseteq \text{set alloc} \\
\land & \, \, \text{digit} < \text{bigInt_base} \\
\land & \, \, \text{BigNumberLZ } \text{bignum} \, \text{first_digit last_digit size sign digit_next digit_prev value} \, \text{Ls num}
\end{align*}
\]

**Function call:** \( \text{retval} = \text{bigIntInsertDigitFront}(\text{bignum, digit}); \)

**Postcondition of bigIntInsertDigitFront**

\[
\begin{align*}
\text{(retval = NO_ERROR} & \land \\
\exists x. \text{value} \, x = \text{digit} & \land \text{NewAlloc } \{x\} \, \sigma \text{alloc alloc} \\
\land & \, \, (0 < \text{digit}) \rightarrow \text{BigNumber } \text{bignum} \, \text{first_digit last_digit size sign digit_next digit_prev value} \\
& \, \, (Ls[@x]) \, (\text{num} + \text{bool2int(} \sigma \, \text{sign bignum})\, \text{-int(} \text{value} \, x\text{)}\, (\text{int(} \text{bigInt_base(} \sigma \, \text{size } \text{bignum})-1))) \\
\land & \, \, (0 = \text{digit}) \rightarrow \text{BigNumberLZ } \text{bignum} \, \text{first_digit last_digit size sign digit_next digit_prev value} \\
& \, \, (Ls[@x]) \, (\text{num} + \text{bool2int(} \sigma \, \text{sign bignum})\, \text{-int(} \text{value} \, x\text{)}\, (\text{int(} \text{bigInt_base(} \sigma \, \text{size } \text{bignum})-1))) \\
\land & \, \, (\forall \, \text{el:ref. el} \neq x \rightarrow (\text{value} \, \text{el} = \sigma \, \text{value} \, \text{el}) \\
\land & \, \, \text{el} \notin (\text{set Ls} \cup \{x\}) \rightarrow (\text{digit_next} \, \text{el} = \sigma \, \text{digit_next} \, \text{el} \\
& \, \, \land \, \text{digit_prev} \, \text{el} = \sigma \, \text{digit_prev} \, \text{el})) \\
\land & \, \, \text{OtherBigIntsUnchanged } \{\text{bignum}\} \, \sigma \, \text{sign sign size size } \sigma \, \text{first_digit first_digit last_digit last_digit}
\end{align*}
\]

Our precondition contains three fundamental requirements that enable us to insert a digit in front of a big number. Most importantly, we require that there is a big number (possibly with leading zeroes in front) at location bignum with corresponding integer value num and list Ls. It is also necessary to require that digit, the digit to be inserted, is in range. That is, the value of digit is smaller than our chosen bigInt_base. Another thing we require is that Ls contains only allocated elements. We need to know this so we can prove that the new digit we allocate is different from the ones already contained in the list Ls.

Since we do not consider the possibility that we may run out of memory, we always are successful allocating a new digit. Thus, since the function finished successfully, we need to express all changes to the heap state in enough detail to make our specification worthwhile. We state that there exists a location x such that:

56
• the value of $x$ equals $\text{digit}$
• $x$ has been newly allocated
• the value of all digits except for $x$ stays the same
• the $\text{digit\_next}$ and $\text{digit\_prev}$ functions remain unchanged on all digits that are different from $x$ and are not contained in the list $\text{Ls}$
• and, most importantly, we obtain a big number (or, depending on the value of $\text{digit}$, a big number with leading zeroes) with list $\text{Ls}@[x]$ at location $\text{bignum}$. The value of this big number is

$$\text{num} + \text{bool2int}(\text{sign bignum})\cdot \text{int value x}\cdot \text{int bigInt\_base(size bignum)−1}$$

That is, we inserted a new digit $x$ with the value of $\text{digit}$ in front of the former most significant digit.

At last, we need to express that all big numbers with the exception of $\text{bignum}$ have not been modified at all. This is, as always, done using the $\text{OtherBigIntsUnchanged}$ predicate.

### 3.8.3 Verification

First off, we need to give the formal specification of the $\text{dList\_InsertAfter}$ function, taken from the $\text{dList}$ package. The function takes two arguments, $\text{here}$, the location after which the second argument will be inserted, and $\text{newelem}$, the location of the node that will be inserted.

The function modifies $\text{digit\_next}$, $\text{digit\_prev}$, and $\text{retval}$.

Let $\text{Ps} \in \text{ref list}$ and let $\sigma \in \Sigma$ be the program state before execution of the function.

**Precondition of $\text{dList\_InsertAfter}$**

$$\exists \text{last1 head. dList head} \text{ digit\_next digit\_prev last1 Ps}$$

Function call: $\text{retval} = \text{dList\_InsertAfter(here, newelem)}$;

**Postcondition:**

**Postcondition of $\text{dList\_InsertAfter}$**

57
(\texttt{newelem} \notin \texttt{set} \texttt{Ps} \land \texttt{here} \neq \texttt{Null} \land \texttt{here} \in \texttt{set} \texttt{Ps} \rightarrow

(\texttt{newelem} \neq \texttt{Null} \rightarrow

(\exists \texttt{last1 head. dList head digit\_next digit\_prev last1 (insertAfter newelem here \texttt{Ps}))

\land (\forall x. x \notin \texttt{set}\texttt{Ps} \cup \texttt{newelem} \rightarrow \texttt{digit\_next} x = \sigma \texttt{digit\_next} x)

\land (\forall x. x \notin \texttt{set}\texttt{Ps} \cup \texttt{newelem} \rightarrow \texttt{digit\_prev} x = \sigma \texttt{digit\_prev} x)

\land \texttt{retval} = \texttt{NO\_ERROR})

\land (\texttt{newelem} = \texttt{Null} \rightarrow

(\exists \texttt{last1 head. dList head digit\_next digit\_prev last1 \texttt{Ps})

\land (\texttt{digit\_next} = \sigma \texttt{digit\_next})

\land (\texttt{digit\_prev} = \sigma \texttt{digit\_prev})

\land \texttt{retval} = \texttt{NO\_ERROR}))

\land (\texttt{here} = \texttt{Null} \rightarrow

(\exists \texttt{last1 head. dList head digit\_next digit\_prev last1 \texttt{Ps})

\land \texttt{digit\_next} = \sigma \texttt{digit\_next}

\land \texttt{digit\_prev} = \sigma \texttt{digit\_prev}

\land \texttt{retval} = \texttt{ERROR\_INVALID})

It is expected that before execution of the function, there exist locations \texttt{last1} and \texttt{head}, such that there is a doubly linked list present on the heap starting in \texttt{head}, ending in \texttt{last1}, connected via \texttt{digit\_next} and \texttt{digit\_prev}, representing the list \texttt{Ps}.

If the location \texttt{here} we gave to the function was the \texttt{Null} location, the function will terminate returning the \texttt{ERROR\_INVALID} error without having made any changes to the heap. In case \texttt{here} is not the \texttt{Null} location and, also, \texttt{here} is an element of the list \texttt{Ps} while, however, \texttt{newelem} explicitely is not part of \texttt{Ps}, there are two cases:

In case \texttt{newelem} is not the \texttt{Null} location, the function terminates after inserting \texttt{newelem} into the list \texttt{Ps} after location \texttt{here}, while preserving the heap on all other locations, returning \texttt{NO\_ERROR}.

If, however, \texttt{newelem} turns out to be the \texttt{Null} location, nothing is modified and still, we return \texttt{NO\_ERROR}.

Since there is no while loop in \texttt{bigIntInsertDigitFront}, we have merely a single subgoal to prove.

\textbf{Solving the only subgoal}

Applying the simplification and clarification methods, we end up at the case split introduced by line 12 of the program text. The case \texttt{last\_digit bignum} = \texttt{Null} can be solved easily by the prover by giving the appropriate constant definitions to the simplifier.

In case \texttt{last\_digit bignum} \neq \texttt{Null}, we call the function \texttt{dList\_InsertAfter} and thus, we are obliged to verify that the precondition of that function is fulfilled. We instantiate
Ps with the list \( Ls \) which belongs to the big number with leading zeroes \( a \). Additionally, head is instantiated with first_digit \( bignum \) and, accordingly, last1 is instantiated with last_digit \( bignum \). The precondition of \( \text{dList}\_\text{InsertAfter} \) is resolved automatically by giving the BigNumberLZ definition to the simplifier.

Since last_digit \( bignum \) \( \neq \) Null, the postcondition of \( \text{dList}\_\text{InsertAfter} \) (which we can now use in our assumptions) simplifies to the first subcase of the first case. That is, if the newly created list element new \( (\text{set alloc}) \) is not part of the list \( Ls \), while last_digit \( bignum \) is, then we know that there is a dList on the updated heap where the new digit has been inserted after last_digit \( bignum \). Additionally, we know that all list elements not containing to the updated list remain unmodified and that \( \text{retval} = \text{NO\_ERROR} \) holds.

What remains to prove is that there exists an \( x \) with the properties required in our postcondition. This \( x \) is the newly allocated element new \( (\text{set alloc}) \) we gave to the \( \text{dList}\_\text{InsertAfter} \) function. All conditions except for the BigNumber and BigNumberLZ conditions are trivial, and thus, automatically resolved. We are left to prove

- \( (0 < \text{digit} \rightarrow \) BigNumber \( bignum \) first_digit \\
  (last_digit(\( bignum := \text{new (set alloc)}) \)) \\
  (size(\( bignum := \text{Suc (size bignum)}) \)) \text{sign digit_nexta digit_preva} \\
  (value(new (\text{set alloc}) := \text{digit}) \) \( (Ls@[\text{new (set alloc)}) \)) \\
  (\text{num + bool2int (sign bignum)-int digit-int (bigInt_base}^\text{size bignum})), \) and

- \( (\text{digit} = 0 \rightarrow \) BigNumberLZ \( bignum \) first_digit \\
  (last_digit(\( bignum := \text{new (set alloc)}) \)) \\
  (size(\( bignum := \text{Suc (size bignum)}) \)) \text{sign digit_nexta digit_preva} \\
  (value(new (\text{set alloc}) := \text{0}) \) \( (Ls@[\text{new (set alloc)}) \)) \text{num} \)

where digit_nexta and digit_preva are the according heap functions after execution of \( \text{dList}\_\text{InsertAfter} \). We use the function update notation \( f(x := y) \), denoting the function that gives the same result \( f \) does, except for the input \( x \) it yields \( y \) for.

We are in a situation where we have to prove two very similar cases. Thus, to reduce the workload, we introduce a subgoal which solves both of the constraints:

\[
\text{BigNumberLZ bignum first_digit} \\
(last_digit(bignum := \text{new (set alloc)})) \\
(size(bignum := \text{Suc (size bignum)}) \text{sign digit_nexta digit_preva}) \\
(value(new (\text{set alloc}) := \text{digit}) \) (\( Ls@[\text{new (set alloc)}) \)) \text{num}) \\
(\text{num +bool2int (sign bignum)-int digit-int (bigInt_base}^\text{size bignum})).
\]

We are left to prove our subgoal in the current context:
Studying our assumptions, we can see that there is some work to be done to establish the fact that we have a `BigNumberLZ` at location `bignum` with the updated heap functions and the new list `Ls@[new (set alloc)]`. First off, we need to expand our `BigNumberLZ` definitions. A small part of the resulting conditions can be resolved by the `clarsimp` method. The remaining conditions deal with the list `Ls@[new (set alloc)]` on the new heap and the equality between the `list2nat`-value of this list and the value we expect in our postcondition for `bignum`.

Since we know that `Ls` is non-empty, we know that there exists a list `ps` such that

\[ Ls = ps@[last_digit bignum] \]

We know that, either `ps = nil` and `last_digit bignum = first_digit bignum`, or `ps \neq nil` and there exists another list `psa` such that

\[ ps = first_digit bignum \# psa \]

and there exists a path

\[ Path \left( digit_next (first_digit bignum) \right) digit_next (last_digit bignum) psa \]

on the heap.

**Case: ps = nil \land last_digit bignum = first_digit bignum**

In this case, we know that `Ls = [last_digit bignum]`, and thus, applying the `insertAfter` and `dList` definitions, establishing the existence of the list `Ls@[new (set alloc)]` becomes trivial. What is left to prove is the equality of

\[
\text{bool2int}(\text{sign } bignum) - (\text{int} (\text{value} (first_digit bignum)) + \text{int}(\text{bigInt_base}^{\text{size } bignum} \cdot \text{digit})) \\
\text{and} \\
\text{num} + \text{bool2int}(\text{sign } bignum) \cdot \text{int} \cdot \text{int}(\text{bigInt_base}^{\text{size } bignum})
\]

The first expression is the one resulting from our `BigNumberLZ` definition by applying the `list2nat_split` lemma. The second one is the value we expect `bignum` to have in our postcondition. As we know that `Ls = [last_digit bignum]`, we obtain that `num = \text{bool2int}(\text{sign bignum}) \cdot \text{int}(\text{value} (last_digit bignum))`. Applying the distributivity laws for integer numbers, the equality is solved.
Case: \( ps \neq \text{nil} \land (\exists \text{psa}. ps = \text{first_digit bignum} \# \text{psa} \land \text{Path (digit_next (first_digit bignum)) digit_next (last_digit bignum) psa}) \)

In this case, we know that \( Ls = \text{first_digit bignum} \# \text{psa} \@ [\text{last_digit bignum}] \).

Again, we have the integer equality between the \text{list2nat} value present for \text{bignum} and the value we require in the postcondition. The only difference now is that the expressions are more complex. Applying the \text{list2nat_split} rule to \( Ls@[\text{last_digit bignum}] \) now yields four terms instead of merely two. With added term complexity, we have to apply more individual integer arithmetic properties by hand, resulting in a lengthy but not difficult proof.

However, contrary to the first case, the result of the \text{insertAfter} function application is nontrivial. Thus, proving that we have the required list on the updated heap becomes a nontrivial task. The main points are to prove that there are paths

\[
\text{Path (digit_preva (digit_preva (new (set alloc)))) (digit_preva (first_digit bignum)) (rev psa)}
\]

and

\[
\text{Path (digit_nexta (first_digit bignum)) digit_nexta (last_digit bignum) psa}
\]

on the updated heap. We need to use our assumption that there is a \text{dList} on the updated heap with list \( (\text{insertAfter (new (set alloc)) (last_digit bignum) Ls}) \). Expanding the definition of \text{insertAfter}, we obtain that we have the list

\[
(\text{if first_digit bignum} \neq \text{last_digit bignum}
\text{then first_digit bignum} \# \text{takeWhile (\lambda u. u \neq last_digit bignum)} (\text{psa}@[\text{last_digit bignum}])
\text{else nil}) \@ [\text{last_digit bignum}, \text{new (set alloc)}]
\]

on our updated heap. As we know that \( \text{first_digit bignum} \neq \text{last_digit bignum} \), the expression simplifies further:

\[
\text{first_digit bignum} \# \text{takeWhile (\lambda u. u \neq last_digit bignum)} (\text{psa}@[\text{last_digit bignum}]) \@ [\text{last_digit bignum}, \text{new (set alloc)}]
\]

\text{takeWhile} is a function that takes a condition \( P \) and a list \( xs \), and returns, starting from the head element, the part of the list we visit until the condition becomes \text{False}. As we know that all the location of a list are distinct, we know that \( \text{last_digit bignum} \) is not part of the list \( \text{psa} \). Thus, our application of \text{takeWhile} simply returns the list \( \text{psa} \). That is, we know that on our updated heap, we have the doubly linked list

\[
\text{first_digit bignum} \# \text{psa}@[\text{last_digit bignum}, \text{new (set alloc)}]
\]
Since the path properties we need to prove deal with the sublist $psa$ of this $dList$, we can see that the respective paths are present on the heap in the specified form.

This concludes the proof of $\text{bigIntInsertDigitFront}$.

\[ \square \]

### 3.9 bigNumAssignInt

**Assignment: Integer to big number**

```c
int bigNumAssignInt(struct bigint *bignum, int integer);
```

The function assigns the value of the integer $integer$ to the big number $bignum$. Return values:

- **NO_ERROR**: The function finished successfully.
- **ERROR_OUT_OF_MEMORY**: The function ran out of memory.

#### 3.9.1 Implementation

In our implementation, $\text{bigInt_base}^2 - 1$ equals the maximum integer value. Thus, one integer value corresponds to at most two big number digits.

First off, we check that $bignum$ is not the NULL pointer. Then, we use the function $\text{bigNumClear}$ to set the value of our big number to zero. In case $integer == 0$, we are done.

We do a simple case split on the sign of our non-zero integer $integer$.

Given a positive integer $integer$, we extract the most significant digit of the corresponding big number by dividing $integer$ by $\text{bigInt_base}$, storing the result in tmp. We obtain the least significant digit by computing $integer - tmp*\text{bigInt_base}$, which we promptly give to the function $\text{bigIntInsertDigitFront}$, inserting it as most significant digit in our empty big number. Afterwards, in case tmp is non-zero, we again use the function $\text{bigIntInsertDigitFront}$ to insert the most significant digit $tmp$ in front of the big number $bignum$. Also, we set the sign of $bignum$ accordingly.

For the negative case, things are implemented in an analogous way.
```c
int bigNumAssignInt(struct bigint *bignum, int integer)
{
    unsigned int tmp;
    int retval;

    retval = NO_ERROR;
    retval = bigNumClear(bignum);

    if(integer != 0)
    {
        if(integer < 0)
        {
            tmp = unsigned(-integer)/ bigInt_base;
            retval = bigIntInsertDigitFront(bignum, unsigned(-integer) - tmp*bigInt_base);

            if(retval == NO_ERROR && tmp != 0u)
            {
                retval = bigIntInsertDigitFront(bignum, tmp);
            }

            bignum->sign = true;
        }
        else
        {
            tmp = unsigned(integer)/ bigInt_base;

            retval = bigIntInsertDigitFront(bignum, unsigned(integer) - tmp*bigInt_base);

            if(retval == NO_ERROR && tmp != 0u)
            {
                retval = bigIntInsertDigitFront(bignum, tmp);
            }

            bignum->sign = false;
        }
    }

    return retval;
}
```
3.9.2 Specification

This function modifies \texttt{retval}, \texttt{size}, \texttt{first_digit}, \texttt{last_digit}, \texttt{value}, \texttt{digit_next}, \texttt{digit_prev}, \texttt{sign}, \texttt{tmp}, and \texttt{alloc}.

Let $\sigma \in \Sigma$ be the program state before execution of the function.

\begin{Verbatim}
Precondition of \texttt{bigNumAssignInt}
\end{Verbatim}

\begin{itemize}
\item \texttt{nat(abs(integer)) < bigInt_base·bigInt_base} \land \texttt{bignum \neq Null}
\end{itemize}

\begin{Verbatim}
Function call: \texttt{retval = bigNumAssignInt(bignum, integer)};
\end{Verbatim}

\begin{Verbatim}
Postcondition of \texttt{bigNumAssignInt}
\end{Verbatim}

\begin{itemize}
\item \texttt{NO_ERROR} \land
\item \texttt{(\exists Ls. BigNumber bignum first_digit last_digit size sign digit_next digit_prev value Ls integer} \land \texttt{NewAlloc (set Ls) \sigma alloc alloc}
\item \texttt{OtherListsUnchanged Ls digit_next \sigma digit_next digit_prev \sigma digit_prev value \sigma value \sigma alloc}
\item \texttt{OtherBigIntsUnchanged [bignum] \sigma sign sign \sigma size size \sigma first_digit first_digit \sigma last_digit last_digit)}
\end{itemize}

We do not require many preconditions for this function. There are merely two requirements,

- the location \texttt{bignum} is different from the Null location, and
- the absolute value of \texttt{integer} is smaller than \texttt{bigInt_base}^2.

In our idealized environment with unbounded memory space, the function always returns \texttt{NO_ERROR}, we state that there exists a list \texttt{Ls}, such that:

- The elements of \texttt{Ls} have been newly allocated
- The heap functions on all list elements with the exception of those contained in \texttt{Ls} remain unchanged
- There is a big number with the value \texttt{integer} and the list \texttt{Ls} at location \texttt{bignum}

Additionally, we state that no big numbers apart from \texttt{bignum} have been modified.
3.9.3 Verification

This, again, is a function without a while loop and, thus, it only yields a single subgoal when invoking the verification condition generator. One of the main points in proving correctness of this function lies in re-establishing equality between \texttt{integer} and the value of the big number we obtain when inserting our computed digits using the function \texttt{bigIntInsertDigitFront}. The case split on the sign of \texttt{integer} introduces two symmetric cases. We will, in the following only consider the first case, \texttt{integer} < 0.

To verify formal correctness of this function, several lemmata are required. One particularly important lemma is the \texttt{mult_div_cancel} property of the division and modulo operator:

\begin{verbatim}
lemma mult_div_cancel:
  (n:nat) · (m div n) = m - (m mod n)
\end{verbatim}

This lemma is part of the higher order theory libraries of the Isabelle/HOL package. It is not only applied in the proofs of some of the following lemmata, but also at several points during the formal correctness proof. We know that dividing a natural number \texttt{m} by another number \texttt{n}, multiplying the result again by \texttt{n} yields the same as subtracting \texttt{m mod n} from \texttt{m}.

\begin{verbatim}
lemma a_sub_a_div_mod_less:
  0 < n → (a:nat) - (a div n) · n < n
\end{verbatim}

Lines 13 and 14 of the program text yield the expression

\[
\text{nat}(-\text{integer}) - \text{tmp} \cdot \text{bigInt_base} = \text{nat}(-\text{integer}) - (\text{nat}(-\text{integer}) \div \text{bigInt_base}) \cdot \text{bigInt_base}
\]

which is given to the function \texttt{bigIntInsertDigitFront}. As we can see, the left hand side of the inequality in this lemma is of the same form as the expression we insert to our big number. In the precondition of \texttt{bigIntInsertDigitFront}, we have to prove that the value of the digit we insert is smaller than \texttt{bigInt_base}. This, we obtain simply by applying this lemma.

\begin{verbatim}
lemma n_div_a_less_a:
  ((n:nat) < a² ∧ 0 < a) → n div a < a
\end{verbatim}

This lemma is useful for proving that, in the second call to \texttt{bigIntInsertDigitFront} (line 17), the value of the digit we insert is smaller than \texttt{bigInt_base}. Note that, in our precondition for \texttt{bigNumAssignInt}, we require that \texttt{integer} < \texttt{bigInt_base²}, and thus, the lemma is applicable.

Concerning the proof that the value of the big number we obtain after applying the function actually has the same values as the variable \texttt{integer}, let us consider the list2nat-value

65
of the list that has been created using \texttt{bigIntInsertDigitFront}. There are, again, two cases. Either \texttt{tmp = 0} and we inserted only one digit to our fresh big number, or \texttt{tmp \neq 0} and we inserted two digits.

**Case: tmp = 0**
In case we inserted only one digit, we know that we inserted a digit with value \texttt{-integer} and set the sign of \texttt{bignum} to \texttt{True}. Thus, it is easy to see that the value associated with \texttt{bignum} is the value of \texttt{integer}.

**Case: tmp \neq 0**
In this case, two digits have been inserted to \texttt{bignum}. Let us refer to the least significant digit by \texttt{x} and to the most significant digit by \texttt{xa}. From the postconditions of the \texttt{bigIntInsertDigitFront} calls, we know that there is a big number at location \texttt{bignum} with list \texttt{[x]@[xa]} and value

\[
\texttt{value x + bigInt\_base \cdot value xa}
\]

Inserting the specific values of these digits we inserted yields

\[
\texttt{nat(-integer) - (nat(-integer) \div bigInt\_base) \cdot bigInt\_base + bigInt\_base \cdot (nat(-integer) \div bigInt\_base)}
\]

Here, it is important to note, that \texttt{(nat(-integer) \div bigInt\_base) \cdot bigInt\_base \leq nat(-integer)}. Since we are dealing with natural numbers, the last two terms can only cancel if the result of the subtraction beforehand had a well-defined result in the natural numbers. As these terms cancel, we obtain that the \texttt{list2nat} value associated with the list is \texttt{nat(-integer)} and thus, the value of \texttt{bignum} is \texttt{integer}.

\[
\square
\]

### 3.10 \texttt{bigIntAddLocal}

**In-place Addition of Big Numbers**

\[
\text{int bigIntAddLocal(struct bigint *a, struct bigint *b)};
\]

This function adds the absolute values of the big numbers \texttt{a} and \texttt{b}, storing the result in \texttt{a} while keeping the sign bit of \texttt{a}. This efficient in-place addition helps to speed up multiplication. Return values:

- \texttt{NO\_ERROR}: The function finished successfully.
- \texttt{ERROR\_OUT\_OF\_MEMORY}: The function ran out of memory.
3.10.1 Implementation

First off, we set our temporary variable \texttt{carry}, used to store the carry of the addition operation, to zero, and initialize our list iteration pointers \texttt{current_a} and \texttt{current_b} by setting them to point to the respective big number’s least significant digit.

We begin going through both lists at the same time along the \texttt{digit\_next} pointers.

- At every step, we add the values of the current digits, storing the result in \texttt{tmp}. We extract the new \texttt{carry} from \texttt{tmp} by dividing \texttt{tmp} by \texttt{bigInt\_base}. Afterwards, we change the value of \texttt{current_a} to \texttt{tmp - carry*bigInt\_base}.

At some point, one of the lists, or both, will end, resulting in one or both of \texttt{current_a} and \texttt{current_b} being the \texttt{NULL} pointer.

In case \texttt{current_a} is not the \texttt{NULL} pointer, we follow the list of the big number \texttt{a} along the \texttt{digit\_next} pointers of the digits as long as the \texttt{carry} is still non-zero.

- During every step, we add the \texttt{carry} to the value of \texttt{current_a}, storing the result in \texttt{tmp}. We extract the new \texttt{carry} from \texttt{tmp} by dividing \texttt{tmp} by \texttt{bigInt\_base}. Afterwards, we change the value of \texttt{current_a} to \texttt{tmp - carry*bigInt\_base}.

In case, \texttt{current_b} is not the \texttt{NULL} pointer, we follow the list of the big number \texttt{b} along the \texttt{digit\_next} pointers, inserting new digits to \texttt{a} as we go along.

- In every step, we add \texttt{carry} to the value of \texttt{current_b}, storing the result in \texttt{tmp}. We extract the new \texttt{carry} from \texttt{tmp} by dividing \texttt{tmp} by \texttt{bigInt\_base}. Afterwards, we insert a new digit with the value of \texttt{tmp - carry*bigInt\_base} to \texttt{a} using the function \texttt{bigInt\_InsertDigit\_Front}.

Note that depending on the length of the lists, either, none of the last two loops is executed, or at most one of them is actually applicable.

At last, in case we leave the while loops with a non-zero carry, we insert the \texttt{carry} as most significant digit to \texttt{a} using the function \texttt{bigInt\_InsertDigit\_Front}.

```c
1  int bigIntAddLocal(struct bigint *a, struct bigint *b)
2  {
3    unsigned int carry;
4    unsigned int tmp;
5    struct bigint_digit *current_a, *current_b;
6    int retval;
7    retval = NO_ERROR;
```
carry = 0u;
current_a = a->first_digit;
current_b = b->first_digit;

while(current_a != NULL && current_b != NULL)
{
    tmp = carry + current_a->value + current_b->value;
carry = tmp / bigInt_base;
tmp = tmp - carry*bigInt_base;
current_a->value = tmp;
current_a = current_a->digit_next;
current_b = current_b->digit_next;
}

while(current_a != NULL && carry != 0u)
{
    tmp = carry + current_a->value;
carry = tmp / bigInt_base;
tmp = tmp - carry*bigInt_base;
current_a->value = tmp;
current_a = current_a->digit_next;
}

while(current_b != NULL && retval == NO_ERROR)
{
    tmp = carry + current_b->value;
carry = tmp / bigInt_base;
tmp = tmp - carry*bigInt_base;
retval = bigIntInsertDigitFront(a, tmp);
current_b = current_b->digit_next;
}

if(carry != 0u && retval == NO_ERROR)
{
    retval = bigIntInsertDigitFront(a, carry);
}
return retval;

Listing 3.7: Source Code of bigIntAddLocal

3.10.2 Specification

This function modifies retval, size, first_digit, last_digit, value, digit_next, digit_prev, and alloc.
Let $num1, num2 \in \text{int}$, let $Ls1, Ls2 \in \text{ref list}$, and let $\sigma \in \Sigma$ be the program state before execution of the function.

**Precondition of bigIntAddLocal**

\[
\begin{align*}
\text{BigNumber } a \text{ first_digit last_digit size sign digit_next digit_prev value } Ls1 & \text{ num1} \\
\land b \neq a & \land ((\text{set } Ls1) \cap (\text{set } Ls2)) = \emptyset \\
\land \text{BigNumber } b \text{ first_digit last_digit size sign digit_next digit_prev value } Ls2 & \text{ num2} \\
\land \text{set } Ls1 & \subseteq \text{set alloc} \land \text{set } Ls2 \subseteq \text{set alloc}
\end{align*}
\]

Function call: \text{retval} = \text{bigIntAddLocal(a,b)};

**Postcondition of bigIntAddLocal**

\[
\begin{align*}
\text{retval} & = \text{NO_ERROR} \\
\land (\exists \text{Ls. BigNumber } a \text{ first_digit last_digit size sign digit_next digit_prev value } (Ls1@Ls)) \\
\land (\lceil \text{num1} \rceil + \lceil \text{num2} \rceil) \cdot (\text{bool2int (sign a)}) & \\
\land \text{NewAlloc (set Ls)} & \text{ alloc alloc} \\
\land \text{OtherListsUnchanged (Ls1@Ls)} & \text{ digit_next } \sigma \text{digit_next digit_prev } \sigma \text{digit_prev value } \sigma \text{alloc} \\
\land \text{OtherBigIntsUnchanged [a]} & \text{ sign } \sigma \text{size size } \sigma \text{first_digit first_digit } \sigma \text{last_digit last_digit}
\end{align*}
\]

In our precondition we require that there are big numbers residing at the locations $a$ and $b$ with the respective lists $Ls1$ and $Ls2$ and the respective values $num1$ and $num2$. Also, we will only consider function calls where the locations $a$ and $b$ actually differ and thus, their lists are entirely different. Additionally, we require that all considered list elements are already allocated.

After execution of the function, we know that, either, we ran out of memory or there was no error and there exists a list $Ls$ such that several things hold:

- There is a big number at location $a$ with corresponding list $Ls1@Ls$ and value $(\lceil \text{num1} \rceil + \lceil \text{num2} \rceil) \cdot (\text{bool2int (sign a)})$. That is, we added the absolute value associated with the big number at location $b$ to the absolute value (represented by the list $Ls1$) residing at $a$.

- The elements of $Ls$ have been newly allocated.

- Any list elements not contained in the lists $Ls$ and $Ls1$ remain unchanged.

Also, we guarantee that no big numbers other than the one at location $a$ have been modified.
3.10.3 Verification

This function, while not being overly complicated, is hard to formally verify. Here, we have three while loops that each require an invariant containing detailed information about the current heap state and the changes to it.

The proof of formal correctness for this function is the longest one we produced. The arithmetic involved is to be considered trivial on paper, however, in our higher order representation of natural and integer numbers, we have to pay close attention to detail, often applying fundamental arithmetic properties by hand since the prover inherently has problems dealing with arithmetic expressions.

We have seven subgoals to consider. Three of them deal with invariant preservation in the loops, the rest consider what happens before, between, and after the while loops. Luckily, most of them are rather trivial and the prover can solve them almost by itself. However, we will not go into detail about the individual subgoals, considering we will mainly find lengthy technical details there. The main part of the proof lies in finding the correct invariants.

Let us simply introduce and explain the invariants.

The first Invariant

<table>
<thead>
<tr>
<th>Invariant for the first while loop in bigIntAddLocal</th>
</tr>
</thead>
<tbody>
<tr>
<td>( (current_a = \text{Null} \land current_b = \text{Null} \land length\ Ls1 = length\ Ls2 \land (\text{carry} \neq 0) \lor (last_digit\ a \neq \text{Null} \rightarrow value\ (last_digit\ a) \neq 0)) \land \text{BigNumberLZ\ a first_digit last_digit size sign digit_next digit_prev value Ls1 (bool2int (\text{sign} a) \cdot (abs(num1) + int(list2nat (map value Ls2)) - int(carry \cdot (\text{bigint}_base length\ Ls1))))}) \land \text{OtherListsUnchanged\ Ls1 digit_next digit_next digit_prev digit_prev value value alloc} \lor )</td>
</tr>
<tr>
<td>( (current_a = \text{Null} \land current_b = \text{Null} \land (\exists LsLb1 LsLb2. Ls1 = LsLb1 @ [current_a] @ LsLb2 \land length\ LsLb1 = length\ Ls1 \land \text{BigNumberLZ\ a first_digit last_digit size sign digit_next digit_prev value Ls1 (bool2int (\text{sign} a) \cdot (abs(num1) + int(list2nat (map value Ls2)) - int(carry \cdot (\text{bigint}_base length\ Ls1))))}) \land \text{OtherListsUnchanged\ Ls1 digit_next digit_next digit_prev digit_prev value value alloc} \lor )</td>
</tr>
<tr>
<td>( (current_a \neq \text{Null} \land current_b = \text{Null} \land (\exists LsLb1 LsLb2. Ls1 = LsLb1 @ [current_a] @ LsLb2 \land length\ LsLb1 = length\ Ls2 \land \text{BigNumberLZ\ a first_digit last_digit size sign digit_next digit_prev value Ls1 (bool2int (\text{sign} a) \cdot (abs(num1) + int(list2nat (map value Ls2)) - int(carry \cdot (\text{bigint}_base length\ Ls2))))}) \land \text{OtherListsUnchanged\ LsLb1 digit_next digit_next digit_prev digit_prev value value alloc} \lor )</td>
</tr>
<tr>
<td>( (current_a \neq \text{Null} \land current_b \neq \text{Null} \land (\exists LsLb1 LsLb2. Ls1 = LsLb1 @ [current_a] @ LsLb2 \land LsLb1 = length\ LsLb1 \land \text{BigNumberLZ\ a first_digit last_digit size sign digit_next digit_prev value Ls1 (bool2int (\text{sign} a) \cdot (abs(num1) + int(list2nat (map value Ls2)) - int(carry \cdot (\text{bigint}_base length\ Ls1))))}) \land \text{OtherListsUnchanged\ LsLb1 digit_next digit_next digit_prev digit_prev value value alloc} \lor )</td>
</tr>
</tbody>
</table>
Our first invariant, like all our invariants for this function consists of two parts:

- The first part deals with the state of our list iteration pointers and their position (if applicable) in our lists \( Ls1 \) and \( Ls2 \) and the value of the big number \( a \).
- The second part establishes some properties needed to prove invariance, and also, maintains useful information through the loop.

In the first part of this invariant, we perform a complete case analysis on \( current_a \) and \( current_b \). If one of the first three cases occurs, we know that we will not perform another loop iteration since the while loop condition yields false. We specify each individual case in detail, since, when leaving the while loop, in case one of the list iteration pointers is not the Null location, we need to remember its position in the corresponding list for the invariant of the next while loop. We know that, before any loop iteration, \( a \) is a \( \text{BigNumberLZ} \), a big number with leading zeroes. Depending on the state of \( current_a \) and \( current_b \), the integer value of \( a \) is expressed in the appropriate way. Note that we subtract the carry from the sum of the \( \text{list2nat} \) value of \( Ls1 \) and the value of the part of the list \( Ls2 \) we already passed (in other words, \( a \) is a temporary result where the addition carry has not yet been added).

In case both \( current_a = \text{Null} \) and \( current_b = \text{Null} \), both lists had the same length and we finished adding the values of the digits of \( b \) to those of \( a \). Note the additional condition that we either have a non-zero carry or that the most significant digit of \( a \) is non-zero,

\[
\text{carry} \neq 0 \lor (\text{last_digit} a \neq \text{Null} \rightarrow \text{value} (\text{last_digit} a) \neq 0)
\]

If we leave out this condition, we later on lack the facts to prove that after execution of \( \text{bigIntAddLocal} \) there is a \( \text{BigNumber} \) present at location \( a \).

In the second case, \( current_a = \text{Null} \) and \( current_b \neq \text{Null} \), we know that we processed the complete list \( Ls1 \), but there is still a part of \( Ls2 \) left to be added to \( a \). This is the job of the third while loop. Thus, we need to remember the position of \( current_b \) in the list \( Ls2 \).
The invariant of the second while loop will then pass this information on to the invariant of the third loop.

In case \( \text{current}_a \neq \text{Null} \) and \( \text{current}_b = \text{Null} \), we find that the list \( Ls_2 \) of \( b \) was shorter than the list \( Ls_1 \) of \( a \). We still have to process possible rippling of the carry through the remaining list of \( a \). This is done by the second while loop. Thus, we express the position information of \( \text{current}_a \) in this condition so that we can make use of it in the invariant of the second while loop.

In the last case, \( \text{current}_a \neq \text{Null} \) and \( \text{current}_b \neq \text{Null} \), we find both our \( \text{current}_a \) and \( \text{current}_b \) list iterators to be somewhere in the middle of the lists \( Ls_1 \) and \( Ls_2 \) in such a way that both iterators already passed parts of the lists of the same length. That is, we are still in the process of adding the values of the corresponding digits, storing the results in the digit list of \( a \). This is our actual invariant condition in the sense that processing of the loop will continue.

Additionally, independent of the former case distinction, there are a few facts we want to maintain throughout the loop. We want to remember that \( b \) contains a big number with list \( Ls_2 \) and value \( \text{num2} \), the locations \( a \) and \( b \) are still different, and that we do not allocate new list elements in respect to the state \( \sigma_{\text{alloc}} \) of allocated locations on the heap before execution of the function. Also, we know that the big number heap functions remain unchanged on all inputs. Another thing we want to remember is, that if \( a \) was non-zero in state \( \sigma \) before execution of the function, the value of the most significant digit was non-zero at that time. We need this property in the case that the list of \( b \) is shorter than that of \( a \). At all times, we need to guarantee that the value of the carry is in range, i.e. less than \( \text{bigint}_\text{base} \). Also, we state that our lists \( Ls_1 \) and \( Ls_2 \) that are required to consist of allocated elements in the precondition stay allocated and their sets of elements are still disjoint.

The second Invariant

Invariant for the second while loop in \( \text{BigIntAddLocal} \)

\[
\begin{align*}
& (\text{current}_a \neq \text{Null} \land \text{current}_b = \text{Null}) \\
& \quad \land (\exists Lsa1 Lsa2. Ls1 = Lsa1[\text{current}_a]@Lsa2) \\
& \quad \land \text{BigIntLZ a first_digit last_digit size sign digit_next digit_prev value Ls1} \\
& \quad \quad (\text{bool2int}(\sigma_{\text{sign a}}) \cdot (\text{abs(num1)} + \text{int(list2nat(map value Ls2)})) \\
& \quad \quad \text{int(carry} \cdot \text{bigint}_\text{base}^\text{length(Ls1)})) \\
& \quad \land \text{OtherListsUnchanged Ls1 digit_next digit_prev value Ls1} \\
& \quad \lor (\text{current}_a = \text{Null} \land \text{current}_b = \text{Null}) \\
& \quad \land \text{BigIntLZ a first_digit last_digit size sign digit_next digit_prev value Ls1} \\
& \quad \quad (\text{bool2int}(\sigma_{\text{sign a}}) \cdot (\text{abs(num1)} + \text{int(list2nat(map value Ls2)})) \\
& \quad \quad \text{int(carry} \cdot \text{bigint}_\text{base}^\text{length(Ls1)})) \\
& \quad \land (\text{carry} \neq 0 \lor (\text{last_digit a} \neq \text{Null} \rightarrow \text{value (last_digit a)} \neq 0)) \\
& \quad \land \text{OtherListsUnchanged Ls1 digit_next digit_prev value alloc Ls1} \\
& \lor \end{align*}
\]
With the second while loop, we consider the case that the list of \(a\) is longer than that of \(b\), and thus, \(current\_a \neq Null\). We only need to continue with the loop as long as \(carry\) is non-zero, since all we do here is to allow the carry to ripple through the remaining part of \(a\)’s digit list. This is our while loop condition.

The second invariant is quite similar to the first one. The second part of it is almost identical to that of the first invariant. Our case distinction in the first part, however, consists of one case less than in the first invariant. This is because, after leaving the first while loop, \(current\_a \neq Null \land current\_b \neq Null\) can no longer hold. Thus, we consider the remaining three cases.

The first case, \(current\_a \neq Null \land current\_b = Null\), is the one where execution of the while loop continues, as long as \(carry \neq 0\). Thus, we express what we know about the value of the big number with leading zeroes at location \(a\), which is, that we only need to add the carry at the next digit position to this partial result to get a final one. Other than that and the fact that \(current\_a\) is part of the list \(Ls1\), we only need to maintain that no other digits were modified apart from those contained in \(Lsa1\).

The second case, \(current\_a = Null \land current\_b = Null\), can either occur directly after leaving the first while loop, or it can be reached by executing the second while loop until we reach the end of the list of \(a\), starting from the first case. What we need to know is that we now either have a non-zero carry, or our most significant digit of \(a\) is already non-zero. Other than that, we merely need our standard properties to hold.

Our third case, \(current\_a = Null \land current\_b \neq Null\), can occur only after leaving the first while loop. It serves only to provide the necessary information about the list split of \(b\) by \(current\_b\) to the third while loop which deals with the case that the list of \(b\) is longer than that of \(a\).
The third Invariant

Invariant for the third while loop in \texttt{bigIntAddLocal}

\[
\begin{align*}
& (\text{current}_b \neq \text{Null} \land \text{current}_a = \text{Null} \\
& \land (\exists \text{Lsb}_1 \text{Lsb}_2 \text{Lsa}. \text{NewAlloc}(\text{set Lsa})^{\sigma} \text{alloc alloc} \land \text{length}(\text{Lsa} @ \text{Lsa}) = \text{length} \text{Lsb}_1 \\
& \land \text{Lsa}_2 = \text{Lsb}_1 @ [\text{current}_b] @ \text{Lsb}_2 \\
& \land \text{BigNumber}Z \text{a first_digit last_digit size sign digit_next digit_prev value} (\text{Lsa} @ \text{Lsa})) \\
& \quad \text{(bool2int} (\sigma \text{sign}) (\text{abs} \text{num1}) + \text{int} \text{list2nat (map value} \text{Lsb}_1)) \\
& \quad \text{- int} (\text{carry} \cdot \text{bigInt_base}^{\text{length} \text{Lsb}_1})) \\
& \land \text{OtherListsUnchanged} (\text{Lsa} @ \text{Ls}_1) \text{digit_next} ^{\sigma} \text{digit_next digit_prev} ^{\sigma} \text{digit_prev value} ^{\sigma} \text{value alloc})
\end{align*}
\]

\[
\begin{align*}
& \lor (\text{current}_b = \text{Null} \\
& \land (\text{carry} \neq 0 \lor (\text{last_digit} \text{a} \neq \text{Null} \rightarrow \text{value} (\text{last_digit} \text{a}) \neq 0)) \\
& \land (\exists \text{Lsa}. \text{NewAlloc}(\text{set Lsa})^{\sigma} \text{alloc alloc} \land (\text{current}_a \neq \text{Null} \rightarrow \text{carry} = 0) \\
& \land \text{BigNumber}Z \text{a first_digit last_digit size sign digit_next digit_prev value} (\text{Lsa} @ \text{Lsa})) \\
& \quad \text{(bool2int} (\sigma \text{sign}) (\text{abs} \text{num1}) + \text{int} \text{list2nat (map value} \text{Lsa}_2)) \\
& \quad \text{- int} (\text{carry} \cdot \text{bigInt_base}^{\text{length} \text{Lsa}_1})) \\
& \land \text{OtherListsUnchanged} (\text{Lsa} @ \text{Lsa}) \text{digit_next} ^{\sigma} \text{digit_next digit_prev} ^{\sigma} \text{digit_prev value} ^{\sigma} \text{value alloc})
\end{align*}
\]

\[
\begin{align*}
& \land \text{BigNumber} \text{b first_digit last_digit size sign digit_next digit_prev value} \text{Ls}_2 \text{num2} \\
& \land \text{retval} = 0 \land \text{b} \neq \text{a} \\
& \land \text{OtherBigIntsUnchanged [a]}^{\sigma} \text{sign} \text{sign} \text{size size} ^{\sigma} \text{first_digit first_digit} ^{\sigma} \text{last_digit last_digit} \\
& \land (\sigma \text{last_digit} \text{a} \neq \text{Null} \rightarrow (\sigma \text{value} \sigma \text{last_digit} \text{a}) \neq 0)) \\
& \land \text{carry} \cdot \text{bigInt_base} \land \text{set Ls}_1 \subseteq \text{set alloc} \\
& \land \text{set Ls}_2 \subseteq \text{set alloc} \land \text{sign a} = \sigma \text{sign a} \land \text{set Ls}_1 \cap \text{set Ls}_2 = \emptyset
\end{align*}
\]

We see that in our third invariant, there are merely two cases left to consider. However, on first glance, the second case will look odd. Why do we consider \text{current}_b = \text{Null} instead of \text{current}_b = \text{Null} \land \text{current}_a = \text{Null}? The reason for this lies in the while loop condition of the second while loop, \text{current}_a \neq \text{Null} \land \text{carry} \neq 0. We know that, when leaving the second while loop, the negation of its while loop condition holds, that is, \text{current}_a = \text{Null} \lor \text{carry} = 0. Thus, we cannot be sure that \text{current}_a = \text{Null} in the case that the condition of the third while loop is not (or no longer) applicable.

In case we continue loop iteration, \text{current}_b \neq \text{Null} \land \text{current}_a = \text{Null}, we need to express a property different from those in the previous while loops. Since the list of \text{a} was shorter than that of \text{b}, we need to insert additional digits to \text{a} using the function \text{bigIntInsertDigitFront}. Thus, we know that there exists a list \text{Lsa} which has been appended to the front of \text{Ls}_1. Also, we know, that our position of \text{current}_b in \text{b}'s list corresponds to the length of the list currently belonging to \text{a}. Other than this, the remaining conditions of this case are essentially the same as always.

In the second case, \text{current}_b = \text{Null}, we know that we are done adding the lists. We need the only condition distinguishing a big number from a big number with leading zeroes to hold:
\[ \text{carry} \neq 0 \lor (last\_digit\ a \neq \text{Null} \rightarrow \text{value}(last\_digit\ a) \neq 0) \]

Additionally, we express that there may be a list \( L_{sa} \) of digits inserted to the front of \( a \), such that we only need to insert the carry in front of \( a \). This works both for the case that the third while loop was applicable and the case that we left the second while loop with a final result, since in that case, \( L_{sa} = \text{null} \).

After execution of this loop, the carry will, in case it is non-zero, be inserted as most significant digit to \( a \) using the function \( \text{BigIntInsertDigitFront} \). Thus, we can see that, after execution of the function we obtain the desired result at location \( a \).

### 3.11 bigIntDigitMul

**Multiplication with a single digit**

\[
\text{int bigIntDigitMul(struct bigint } \ast \text{bignum, unsigned int } \text{digit,} \\
\quad \text{struct bigint } \ast \text{product);} \\
\]

This function multiplies the big number \( \text{bignum} \) and the digit \( \text{digit} \), storing the result in \( \text{product} \). Return values:

- NO_ERROR: The function finished successfully.
- ERROR_OUT_OF_MEMORY: The function ran out of memory.

#### 3.11.1 Implementation

Multiplication of a big number \( \text{bignum} \) with a single digit \( \text{digit} \) is actually quite simple. Since the value of a given big number digit is always smaller than \( \text{bigInt}\_\text{base} \), we know that the value of the product of two big numbers is always less than \( \text{bigInt}\_\text{base}^2 - 1 \). Thus, we get a digit we insert into the result big number \( \text{product} \) and a carry.

After we checked for NULL pointers, we set the value of \( \text{product} \) to zero using the function \( \text{bigNumClear} \). Also, we set the sign of \( \text{product} \) to that of \( \text{bignum} \).

In case neither \( \text{digit} \) or the value of \( \text{bignum} \) are zero, we begin going through the list belonging to \( \text{bignum} \) using our list iteration pointer \( \text{current} \), starting with the least significant digit.

In every step, we multiply the value of our current digit \( \text{current} \) by \( \text{digit} \) and afterwards add the carry \( \text{carry} \), storing the result in \( \text{tmp} \). We extract the new carry from
tmp by dividing tmp by bigInt_base. Afterwards, we insert a new most significant digit with the value of tmp - carry*bigInt_base to the result big number product using the function bigIntInsertDigitFront.

In case carry is non-zero when we leave the while loop, we insert the carry as most significant digit to product using bigIntInsertDigitFront.

```c
int bigIntDigitMul(struct bigint *bignum, unsigned int digit,
                    struct bigint *product)
{
    unsigned int carry;
    struct bigint_digit *current;
    unsigned int tmp;
    int retval;

    retval = NO_ERROR;
    carry = 0u;
    retval = bigNumClear(product);
    product->sign = bignum->sign;

    if(digit != 0u && bignum->first_digit != NULL)
    {
        current = bignum->first_digit;

        while(current != NULL && retval == NO_ERROR)
        {
            tmp = current->value * digit + carry;
            carry = tmp / bigInt_base;
            tmp = tmp - carry*bigInt_base;

            retval = bigIntInsertDigitFront(product, tmp);
            current = current->digit_next;
        }
    }
    if(carry != 0u && retval == NO_ERROR)
    {
        retval = bigIntInsertDigitFront(product, carry);
    }
    return retval;
}
```

Listing 3.8: Source Code of bigIntDigitMul
3.11.2 Specification

This function modifies retval, size, first_digit, last_digit, value, digit_next, digit_prev, sign, tmp, and carry.

Let Ls ∈ ref list, and let num ∈ int. Let σ ∈ Σ be the program state before execution of the function.

**Precondition of bigIntDigitMul**

\[
\begin{align*}
\text{bignum} \neq \text{product} \land \text{product} \neq \text{Null} \land \text{set Ls} \subseteq \text{set alloc} \land \text{digit} < \text{bigInt_base} \\
\land \BigNumber \text{bignum} \text{first_digit last_digit size sign digit_next digit_prev value Ls num}
\end{align*}
\]

\[\text{retval} = \text{bigIntDigitMul(bignum, digit, product)};\]

**Postcondition of bigIntDigitMul**

\[
\begin{align*}
\text{retval} = \text{NO_ERROR} \\
\land (\exists \text{Ls1}. \BigNumber \text{product} \text{first_digit last_digit size sign digit_next digit_prev value Ls1} \\
\quad (\text{num·int(digit)})) \\
\land \text{NewAlloc (set Ls1)\sigma alloc alloc} \\
\land \text{OtherListsUnchanged Ls1 digit_next \sigma digit_next digit_prev \sigma digit_prev value \sigma value \sigma alloc} \\
\land \text{OtherBigIntsUnchanged [product] \sigma sign sign \sigma size size \sigma first_digit first_digit \sigma last_digit last_digit}
\end{align*}
\]

In our precondition, there are several properties that are required to hold before execution of the function:

- There is a big number residing at location bignum with respective list Ls and value num.
- The list Ls consists only of allocated elements.
- product is a non-zero location different from bignum.
- The natural number digit we multiply by is smaller than bigInt_base.

If these hold, we guarantee that after execution the function, there exists a list Ls1 such that:

- There is a big number at location product with list Ls1 and value num·int(digit).
- The list elements of Ls1 have been newly allocated.
- All list elements apart from those in the list Ls1 remain unchanged.

Additionally, we state that only the big number product has been modified.
3.11.3 Verification

The specification of this function is rather brief and straightforward. The invariant however, as we have to keep track of the heap state, is lengthy. Still, as there is merely one while loop, there is only a single invariant, and thus, the actual proof turns out to be much shorter than that of bigIntAddLocal. However, as in bigIntAddLocal, we have to deal with proving correctness of our carry-splitting mechanism which involves lengthy application of integer arithmetic axioms and lemmata.

For this function, we will merely present and explain the invariant.

The Invariant

\[
\begin{align*}
& \text{Invariant for the while loop in bigIntDigitMul} \\
& \text{(current = Null) } \\
& \land \exists Ls1. 0 < \text{length } Ls1 \\
& \land (\text{carry} \neq 0 \lor \text{carry} = 0 \land \text{value (last_digit product) } \neq 0) \\
& \land \text{BigIntLZ product first_digit last_digit size sign digit_next digit_prev value } Ls1 \\
& \quad (\text{num-int(digit)} - \text{bool2int (sign product)} \cdot \text{int(bigInt_base size product)} \cdot \text{int(carry)}) \\
& \land \text{NewAlloc (set } Ls1) \land \text{alloc alloc} \\
& \land \text{OtherListsUnchanged Ls1 digit_next alloc alloc digit_prev value value alloc } \\
& \lor \text{current } \neq \text{Null} \\
& \land \exists Ls2 Ls1 Ls Ls1 = Ls2 @ \text{[current]} @ Ls2 \land \text{length } Ls1 = \text{length } Ls \\
& \land \text{BigIntLZ product first_digit last_digit size sign digit_next digit_prev value } Lsa \\
& \quad ((\text{bool2int (sign product)}) - \text{int(list2nat(map value Ls1) digit)} \\
& \quad \quad \quad \quad - \text{int(carry)} \cdot \text{int(bigInt_base length Lsa)})) \\
& \land \text{NewAlloc (set } Ls) \land \text{alloc alloc} \\
& \land \text{OtherListsUnchanged Lsa digit_next alloc alloc digit_prev value value alloc} \\
& \land \text{retval } = \text{NO_ERROR} \land \text{sign product } = \text{sign bignum} \land \text{carry } < \text{bigInt base} \\
& \land \text{OtherBigIntsUnchanged product sign sign size size first_digit first_digit last_digit last_digit } \\
& \land \text{bignum } \neq \text{product} \land \text{set } Ls \subseteq \text{set alloc digit } < \text{bigInt base} \land 0 < \text{digit} \\
& \land \text{BigIntLZ product first_digit last_digit size sign digit_next digit_prev value } Ls \text{ num} \\
\end{align*}
\]

Like the invariants of bigIntAddLocal, the invariant for this function’s while loop is split in general part and a case-specific one. However, since we only check for the end of a single list, Ls, there are merely two cases to distinguish here.

In the first case, we reached the end of the list and, thus, current = Null. We leave the while loop knowing that there exists a list Ls1 of newly allocated references Ls1, such that there is a big number with leading zeroes present at location product with list Ls1 whose value equals the following arithmetic expression:

\[
\text{num-int(digit)} - \text{bool2int (sign product)} \cdot \text{int(bigInt_base size product)} \cdot \text{int(carry)}
\]
That is, product contains a temporary result of the multiplication of the value of bignum by digit, in such a way that we only need to insert the carry in front of product to obtain the final result. We know that the list Ls1 is not the empty list, since we only enter the while loop in case we will get a non-zero result for our multiplication. Since we are leaving the while loop, we need that at location a, there is a big number, instead of a big number with leading zeroes. Thus, we require the following additional condition:

\[ \text{carry} \neq 0 \lor \text{carry} = 0 \land \text{value (last_digit product)} \neq 0 \]

We can now prove that, if these conditions hold when leaving the while loop, the post-condition holds after inserting carry as most significant digit of product.

The other case, current \neq \text{Null} is actually the more interesting one, since it deals with how we build the result list at location product during our while loop. Since we go through the list Ls of bignum using current, we express the property that current is part of Ls in such a way that there is a part of the list before and after current. As, in every step of the while loop, we insert a digit to product, the list Lsa of product, which is being created during the while loop, has the same length as the list we already passed with current. We state that, before the loop iteration, the big number at location product has the value

\[
\text{bool2int (sign product)-int(list2nat(map value Ls1)-digit) - int(carry)-int(bigInt_base^{length Lsa})}
\]

That is, product contains a temporary product of the value of the part of the list of bignum we already passed and digit. However, we still have to add the carry at the position in front of the list Lsa of the big number product. Establishing invariance of this property again involves lengthy application of integer properties which are, on paper, considered trivial. We can see that, in case Ls2 = nil, in the next step we will reach the case current = \text{Null} and the conditions there will be fulfilled.

### 3.12 bigNumAssignBigInt

**Assignment: Big number to big number**

```
int bigNumAssignBigint(struct bigint *fstbignum, struct bigint *sndbignum);
```

The function assigns the value of the big integer sndbignum to the big integer fstbignum. Return values:

- **NO_ERROR:** The function finished successfully.
- **ERROR_OUT_OF_MEMORY:** The function ran out of memory.
3.12.1 Implementation

First off, we set the value of fstbignum to zero, giving it an empty list, using the function bigNumClear.

In case sndbignum is not the NULL pointer, we begin to copy the digits from sndbignum to fstbignum. We go through the list of sndbignum using current as list iteration pointer starting with the least significant digit.

In every step, we simply insert the value of the current digit of sndbignum as most significant digit to fstbignum using the function bigIntInsertDigitFront.

At last, we set the sign of fstbignum to that of sndbignum and we are done.

```c
int bigNumAssignBigInt(struct bigint *fstbignum, struct bigint *sndbignum)
{
    struct bigint_digit *current;
    int retval;
    retval = NO_ERROR;
    retval = bigNumClear(fstbignum);

    if (retval == NO_ERROR)
    {
        current = sndbignum->first_digit;
        if (current!= NULL)
        {
            while (current->digit_next != NULL && retval == NO_ERROR)
            {
                retval = bigIntInsertDigitFront(fstbignum,current->value);
                current = current->digit_next;
            }
            if (retval == NO_ERROR)
            {
                retval = bigIntInsertDigitFront(fstbignum,current->value);
            }
        }
        fstbignum->sign = sndbignum->sign;
    }
    return retval;
}
```
3.12.2 Specification

This function modifies \texttt{retval}, \texttt{digit\_next}, \texttt{digit\_prev}, \texttt{first\_digit}, \texttt{last\_digit}, \texttt{value}, \texttt{size}, \texttt{alloc}, and \texttt{sign}.

Let \( Ls \in \text{ref list} \) and let \( num \in \text{int} \). Let \( \sigma \in \Sigma \) be the program state before execution of the function.

**Precondition of \texttt{bigNumAssignBigInt}**

\[
\text{set } Ls \subseteq \text{set } alloc \land \text{sndbignum} \neq \text{fstbignum} \land \text{fstbignum} \neq \text{Null} \\
\land \text{BigNumber} \text{sndbignum} \text{ first\_digit} \text{ last\_digit} \text{ size} \text{ sign} \text{ digit\_next} \text{ digit\_prev} \text{ value} \text{ Ls} \text{ num}
\]

Function call: \( \text{retval} = \text{bigNumAssignBigInt}(\text{fstbignum}, \text{sndbignum}); \)

**Postcondition of \texttt{bigNumAssignBigInt}**

\[
\text{retval} = \text{NO\_ERROR} \\
\land (\exists Ls1. \text{BigNumber} \text{fstbignum} \text{ first\_digit} \text{ last\_digit} \text{ size} \text{ sign} \text{ digit\_next} \text{ digit\_prev} \text{ value} \text{ Ls1} \text{ num} \\
\land \text{OtherListsUnchanged} \text{Ls1} \text{ digit\_next} \text{digit\_next} \text{digit\_prev} \text{digit\_prev} \text{value} \text{value} \text{alloc} \\
\land \text{NewAlloc} (\text{set} \text{Ls1}) \text{alloc} \text{alloc}) \\
\land \text{OtherBigIntsUnchanged} [\text{fstbignum}] \text{sign} \text{sign} \text{size} \text{size} \text{first\_digit} \text{first\_digit} \text{last\_digit} \text{last\_digit}
\]

In our precondition we require that:

- There is a big number at location \texttt{sndbignum} with list \texttt{Ls} and value \texttt{num}.
- The elements of the list \texttt{Ls} are all allocated.
- The location \texttt{fstbignum} is not the \texttt{Null} location and different from \texttt{sndbignum}.

If these hold before execution of the function, we state in our postcondition that there exists a list \texttt{Ls1} such that:

- There is a big number at location \texttt{fstbignum} with list \texttt{Ls1} and value \texttt{num}. That is, we assigned the value \texttt{num} of the big number \texttt{sndbignum} to the big number \texttt{fstbignum}.
- The elements of \texttt{Ls1} have been newly allocated.
- All list elements apart from the ones contained in \texttt{Ls1} remain unchanged.

At last, we state that all big numbers other than \texttt{fstbignum} have not been modified.
3.12.3 Verification

To conclude, we present the last function whose formal correctness has been verified during our thesis. `bigNumAssignBigInt` is a comparatively simple function considering we do not need to deal with nontrivial equality of arithmetic expressions, but merely need to keep track of the changes to the heap state and the newly allocated locations. Thus, the invariant is quite short compared to those of the previous functions.

**The Invariant**

---

**Invariant for the while loop in `bigNumAssignBigInt`**

\[
(\exists Ls1, Ls2, Lsa. \text{Ls} = Ls1 @ [\text{current}] @ Ls2 \land \text{length Lsa} = \text{length Ls1} \\
\land \text{NewAlloc}(\text{set Lsa} \circ \text{alloc alloc} \land \text{set Lsa} \cap \text{set Ls} = \emptyset) \land \text{OtherListsUnchanged}(\text{Ls1, Ls2, Lsa}) \\
\land \text{BigNumberLZ}(\text{fstbignum first_digit last_digit size sign digit_next digit_prev value Lsa}) \\
\land \text{map value Lsa} = \text{map value Ls1} \\
\land \text{sndbignum} \neq \text{fstbignum} \land \text{set Ls} \subseteq \text{set alloc} \land \text{reval} = \text{NO_ERROR} \\
\land \text{OtherBigIntsUnchanged}[\text{fstbignum}] \land \text{sign sign size size first_digit first_digit last_digit last_digit} \\
\land \text{fstbignum} \neq \text{Null} \land \text{BigNumber sndbignum first_digit last_digit size sign digit_next digit_prev value Ls num})
\]

---

This invariant does not contain a case distinction on our list iteration pointer. This is the case since we already leave the while loop when we know that the digit we currently point to is the last one. This digit is then processed immediately after the while loop. So we know that, in, before, and after our while loop, `current` will be different from the `Null` location. As we go through the list `Ls` of `sndbignum`, inserting digits to `fstbignum`, we need to express that there is a part of `Ls` that `current` already passed (`Ls1`) and a part that is yet to be processed (`Ls2`). In every loop step, we know that there is a newly created list `Lsa` for `fstbignum` such that `Lsa` has the same length as the list `Ls1` we already copied. We know that the values of these lists digits are identical.

To maintain invariance, we establish a few more properties not dealing with the newly created list or list traversal. We preserve the fact that `sndbignum`, the big number which is being copied to the location `fstbignum`, stays the same. Also, we remember that the locations we are copying from and copying to still differ. The elements of the list belonging to `sndbignum`, `Ls`, stay allocated and no big numbers apart from `fstbignum` are modified.

We leave the loop when `Ls2 = \text{nil}`, since then, `next_digit current` will be the `Null` location. Thus, we know that all but the `current` digit of `sndbignum` have been copied to the big number at location `fstbignum`. Since we insert this missing digit immediately after the while loop, it is clear that the postcondition will hold afterwards.

82
Chapter 4

Summary

In this thesis, we gave a short overview on the programming language C0, higher order logic, and the Hoare logic environment of Isabelle/HOL. We briefly explained abstraction of heap structures using heap functions.

We have implemented and tested a big number package using the programming language C0. For several functions of this package, we have proven partial correctness without guards. Some of these functions contain loops and memory allocation. The work has been done with the aid of the Hoare logic environment of the interactive theorem prover Isabelle. To reach the goal of formal verification, higher order logic predicates have been defined and used to abstract complexity of individual proof parts. Specifications and, as necessary, invariants have been given and briefly explained for all covered functions.

In particular detail, we considered the proofs of formal correctness for the functions bigIntCompare (compares the values of two big numbers), bigIntInsertDigitFront (inserts a new most significant digit to a big number), and bigNumAssignInt (assigns an integer value to a big number). We gave lemmata of relevance and commented in detail on the subgoals and specific issues of the individual proofs.

However, this work covers only a part of the verification effort for the big number package. Termination and correctness with guards has not yet been proven. The correctness of several functions is left to verify in the course of the Verisoft project.
Bibliography


