Formal Verification of a Big Integer Library

Sabine Fischer
Department of Computer Science
Saarland University
Saarbruecken, Germany
sabine@wjpserver.cs.uni-sb.de

Abstract

We present formal correctness of a big integer library based on dynamically allocated lists. The package has been implemented in a restricted C-dialect. In addition to the standard operations for big integers, we provide a function which computes modulo exponentiation results using the square-and-multiply algorithm. Functional correctness has been verified using the Hoare logic environment of the interactive theorem prover Isabelle. We deal with heap modification and pointer aliasing using predicates and lemmas.

1 Introduction

Formal verification of computer programs is an active area of research. Slowly progressing to the point where it becomes an engineering problem, program verification allows us to guarantee that a specific implementation adheres to its formal specification.

This paper extends previous work [5] originally developed for the Verisoft project.[1] The main idea of the project was to verify a complete system, from hardware level up to application level, similar to the famous approach by J S. Moore [8]. One of the goals was to verify correctness of an e-mail client that provides encryption and decryption of messages based on the RSA algorithm [11]. For the implementation and verification of the RSA algorithm, a verified big integer library is one of the basic prerequisites.

Due to the limitations of the high-level programming language employed in the Verisoft project, we chose to base our big integer implementation on linked lists in order to allow big numbers of arbitrary size.

Our verification goal encompasses three vital goals: We focus on pointer-level correctness of the data structure residing on the heap, we consider 'functional' correctness of big number operations in an arithmetic sense and we deal with the problem of pointer-aliasing on the heap. Ultimately, we aim at the creation of modular, opaque specification for our library. What we do not do in this approach is checking for integer overflows or null-pointer accesses.

Concerning related work, I was unable to find much about big integer verification at all. The most relevant work I found was that of J S. Moore who proved correctness of an array-based big number addition algorithm [8] to demonstrate the success of his approach to system verification.

Considering other work on verification of heap-based data structure libraries, there are libraries for dynamically allocated strings [14] and lists [9] from the Verisoft project. Many papers on formal verification of data structure implementations focus on functional and pointer-level correctness of a particular algorithm instead of implementation correctness and useability of specification for a library.

The work on pointer assertion logic by Anders Møller and Michael I. Schwartzbach [7] focuses highly on automatization of heap aliasing and pointer-level correctness proofs for data structures but seems to neglect functional correctness properties. In their recent paper, Bor-Yuh Evan Chang and Xavier Rival [4] propose a wholistic approach that combines functional invariants for heap data structure verification with regular shape analysis techniques.

Our paper is structured in the following way. Section 2 describes the approach to program verification we applied, whereas Section 3 deals with the implementation and the HOL formalization of the big number package. In Section 4, we present predicates to deal with heap modifications caused by function calls. Based on the definitions from Section 3 and 4, we present specification for two functions of the package in Section 5. We discuss verification results in Section 6, followed by a brief summary.

2 The Approach

We have implemented the big integer package using the programming language C0 which has been specifically de-

[1] http://www.verisoft.de

veloped during the Verisoft project. The syntax and semantics of C0 greatly resemble ANSI C. Certain restrictions are made in order to keep the verification effort manageable. Most importantly, we do not consider pointer arithmetics in C0. Also, there are no side-effects in statements, no function calls in expressions and pointers may not point to local variables. Further, we require the size of arrays to be known at compile time and all variable declarations must be made at the beginning of a function body. The lack of dynamic arrays in C0 led to the choice to implement the big integer package based on the doubly-linked list library [9] that had been implemented and verified beforehand.

The framework used for verification is the Hoare logic environment by Norbert Schirmer (described in [13]) of the Isabelle/HOL theorem prover [1]. Since the imperative language model of Isabelle's Hoare logic and our programming language do not coincide, we our C0 code is translated 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 automatically 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 specification and translated program code, the verification condition generator implemented in Isabelle/HOL generates appropriate proof obligations. Since finding loop invariants in verification condition generation is not yet fully automated, most of these invariants have to be annotated manually beforehand.

We establish that the underlying doubly-linked list structure residing on the heap is maintained in the correct way. Further, we establish the connection between the list structure and an the value of the corresponding big integer using predicates. Using these predicates we are able to specify and verify arithmetic correctness properties for the big integer operations provided by the library.

3 The Big Integer Package

Definition: A *big number* (to base $b \in \mathbb{N}$) is represented by a sign bit $s \in \{0,1\}$ and $n \in \mathbb{N}$ 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 by

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

In C0 we represent big numbers using the structs in Fig. 1. The digit structure contains a `value`-field for the value of the digit as well as pointers `next` and `prev` to the next and previous digits. The big number struct contains pointers `first` and `last` to the first and last digit of the list, as well as fields for size (which is the number of digits) and sign.

```
struct digit
{
  struct digit *next;
  struct digit *prev;
  unsigned int value;
};
```

```
struct bigint
{
  struct digit *first;
  struct digit *last;
  unsigned int size;
  bool sign;
};
```

Figure 1: The structs for big number digits and big numbers in C0

Interface of the Library The interface provides functions to create a new big number, to assign an integer to a big number, to assign the value of a big number to another big number, to invert the sign of a big number, and to compare the value of two big numbers. Concerning arithmetic operations, the big number package offers addition and subtraction, as well as multiplication, division, and modulo routines. Additionally, there is a function to compute modulo exponentiation results, as needed in the RSA algorithm.

To reduce complexity of individual functions, several utility functions have been implemented. There are functions for digit insertion (at the front or back), for in-place addition and subtraction, for multiplication and division of big numbers by a single digit, for removal of leading zeros, and for quotient estimation during the division routine.

The Heap Model In the Isabelle/HOL model, we use the abstract type of locations *ref* to model pointers to the heap. This type is isomorphic to the natural numbers and includes a constant Null. The program state contains a list, called the *allocation list*, used to keep track of allocated locations. The new operator in the HOL model always returns a location that is not in the allocation list. We do not consider a limited memory size. For a system with garbage collection (which is what we consider), it appears more sensible to do a separate analysis on memory consumption instead.

Generally speaking, we follow the model of Bornat [2] (which is based on the earlier model of Burstall [3]). In this model, dereferencing of pointers is done by using *heap functions*, i.e. functions that map locations to values. This model abstracts away the low-level model of a byte-addressed heap with update semantics and is justified by the mapping of structure fields to memory by the compiler [6] which deals with inter-type aliasing. Thus, the model by design provides separation between the heap functions (e.g. accesses to different fields of a struct cannot modify the same memory cell).

Formalization in HOL To give a formal definition of the heap representation of big numbers in our model, we introduce a heap function for every field of a struct. To represent the `bigint` structure, we introduce heap functions

$$\mathit{first} : \mathit{ref} \to \mathit{ref} \quad \mathit{size} : \mathit{ref} \to \mathit{nat}$$
$$\mathit{last} : \mathit{ref} \to \mathit{ref} \quad \mathit{sign} : \mathit{ref} \to \mathit{bool}$$

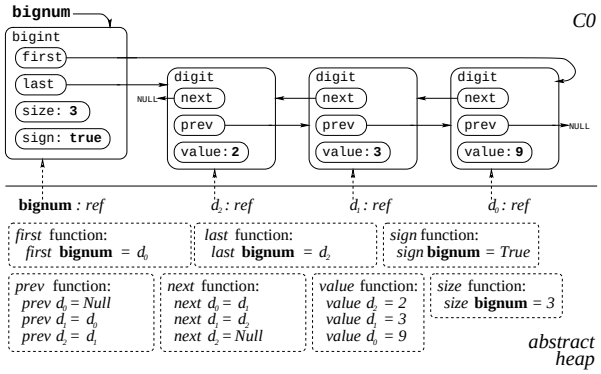

Figure 2: Abstract heap and C0 representations of an example big number representing the value -239 when interpreted to base 10.

These functions take a heap location and return the corresponding fields of the big number structure residing at that location. For the `bigint` struct, we obtain the functions

$next : ref \to ref$ $\quad value : ref \to nat$

$prev : ref \to ref$

For this structure, we use the dList-predicate (from [9]):

dList(**a**, *next*, *prev*, **b**, **ls**)

The predicate asserts for locations **a** and **b** that there is a doubly linked list on the heap between **a** and **b**, represented by the list of locations **ls**.

Applying the *value* function to the list elements of **ls**, we obtain a *nat list* of actual digit values. Using the following function, we convert the list of digit values to the corresponding natural number:

$$\begin{aligned}&\mathsf{list2nat} : nat\ list \to nat\\ &\mathsf{list2nat}\ \mathsf{nil} = 0\\ &\mathsf{list2nat}\ (x\#xs) = x + \mathsf{bigInt\_base} \cdot (\mathsf{list2nat}\ xs)\end{aligned}$$

The function takes a list of natural numbers beginning with the least significant digit. It returns the sum of the products of the digits and the corresponding powers of the big integer base bigInt_base: $\Sigma_{i=0}^{n-1} d_i \cdot \mathsf{bigInt\_base}^i$.

We define the predicate that accepts any big number:

$$\begin{aligned}&\mathsf{BigNumber} : ref \times (ref \to ref) \times (ref \to ref) \times (ref \to nat)\\ &\qquad \times (ref \to bool) \times (ref \to ref) \times (ref \to ref)\\ &\qquad \times (ref \to nat) \times ref\ list \times int \times ref\ list \to bool\\ &\mathsf{BigNumber}(\mathbf{bn}, first, last, size, sign, next, prev, value, \mathbf{ls}, \mathbf{num}, alloc)\\ &\quad = (\mathbf{bn} \neq \mathsf{Null} \wedge \mathbf{bn} \in \mathsf{set}\ alloc \wedge \mathsf{set}\ \mathbf{ls} \subseteq \mathsf{set}\ alloc\\ &\qquad\qquad \wedge \mathsf{dList}\ (first\ \mathbf{bn})\ next\ prev\ (last\ \mathbf{bn})\ \mathbf{ls}\\ &\qquad\qquad \wedge \mathsf{length}\ \mathbf{ls} = size\ \mathbf{bn}\\ &\qquad\qquad \wedge (\forall x \in \mathsf{set}\ \mathbf{ls}.\ (value\ x) < \mathsf{bigInt\_base})\\ &\qquad\qquad \wedge \mathsf{bool2int}\ (sign\ \mathbf{bn}) \cdot \mathsf{int}(\mathsf{list2nat}\ (\mathsf{map}\ value\ \mathbf{ls})) = \mathbf{num}\\ &\qquad\qquad \wedge (first\ \mathbf{bn} \neq \mathsf{Null}) = (last\ \mathbf{bn} \neq \mathsf{Null})\\ &\qquad\qquad \wedge (last\ \mathbf{bn} \neq \mathsf{Null} \to value\ (last\ \mathbf{bn}) \neq 0)\end{aligned}$$

Here, bool2int is a function that maps True to -1 and False to 1 and bigInt_base is a constant that denotes the big integer base we consider. This predicate takes variables, **bn**, **ls**, and **num**, and all relevant heap functions. It yields True if and only if there is a big number with digit list **ls** and value **num** at location **bn**.

We do not consider any structure residing at the Null location to be a big number. Additionally, we require that our big number location as well as the corresponding list are allocated. The dList predicate states that there is a bidirectional path resulting in the list **ls** via the *next* and *prev* functions between the first and last digit of **bn**. Another property is that the *size* field of our big number contains the length of the list **ls**. Also, all digits of the big number contain values less than the chosen base. We have to establish equality between the value associated with the list **ls** and the integer **num**. Thus, **ls** is converted to a list of natural numbers (by mapping the *value* function on **ls**) which is in turn converted to a natural number via the function list2nat. Multiplying with the sign of **bn**, we obtain the value of our big number. Additionally, it is helpful to know that either both *first* and *last* 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.

Note that this predicate should be treated as opaque by other modules that call to our library. To talk about the value of a big integer, other modules should neither refer to the list nor to the digit structure, using the value **num** instead.

4 Heap Modification

When dealing with multiple data structures on the heap, it is extremely important to know that data structures we did not modify are still intact on the modified heap. This is not always trivial due to possible pointer aliasing on the heap. There have been several approaches to deal with the aliasing problem, in particular, separation logic [10], parametric shape analysis [12] and pointer assertion logic [7]. There actually is an implementation of separation logic in Isabelle/HOL [15] which uses a byte-addressable memory model that allows pointer arithmetics. However, due to the different memory model and the explicit decision not to use pointer arithmetics, it was decided not to integrate a similar separation logic framework into our working environment during the project. In particular, in the absence of pointer arithmetics, it was unclear whether the extra-effort required to employ separation logic would actually bring sufficient advantages. Instead, we use a very basic and straight-forward approach, including information about the unmodified part of the heap in function specification.

Frame Predicates A *frame predicate* is a predicate that specifies which locations potentially had their values modified during an heap update. It takes a list of heap locations, as well as a certain number of heap pairs, and is true if the

values of all locations that are not contained in the list remain unmodified between the individual heap pairs.

We determined that this minimalistic definition, while expressing the necessary properties, is not the best choice in practice. Such a predicate requires the system to track all heap locations which were modified, even for temporary structures that we would like to discard after a function call. In reality, we only need to talk about allocated heap locations. Thus, we consider the allocation list associated with the older of the two heap states in our frame predicate.

For verification, we use two frame predicates, one for the digit structure and one for the big number structure:

BigIntsUnchanged : *ref list* × (*ref* → *bool*) × (*ref* → *bool*)
× (*ref* → *nat*) × (*ref* → *nat*) × (*ref* → *ref*) × (*ref* → *ref*)
× (*ref* → *ref*) × (*ref* → *ref*) × *ref list* → *bool*
BigIntsUnchanged(**bns**,*sign*,*sign'*,*size*,*size'*,*first*,*first'*,*last*,*last'*,*alloc*)
= (∀ bn:*ref*. bn ∉ set **bns** ∧ bn ∈ set *alloc* → (*sign* bn = *sign'* bn
∧ *size* bn = *size'* bn
∧ *first* bn = *first'* bn
∧ *last* bn = *last'* bn))

The frame predicate for the `bigint` struct considers a list of locations **bns**, pairs of heap functions belonging to the struct and an allocation list. It is true if no big number structures were modified apart from those in **bns**.

ListsUnchanged : *ref list* × (*ref* → *ref*) × (*ref* → *ref*) × (*ref* → *ref*)
× (*ref* → *ref*) × (*ref* → *nat*) × (*ref* → *nat*) × *ref list* → *bool*
ListsUnchanged(**els**, *next*, *next'*, *prev*, *prev'*, *value*, *value'*, *alloc*)
= (∀ el:*ref*. el ∉ set **els** ∧ el ∈ set *alloc* → (*next* el = *next'* el
∧ *prev* el = *prev'* el
∧ *value* el = *value'* el))

The predicate for the `bigint` struct is built in an analogous way. The only difference is that there is a list of digit locations associated with a single big number, and, thus, using this predicate we tend to specify complete big number lists.

Allocation Information Another specification issue is that we need to keep track of elements which were allocated during a function call. This information is particularly imporant when we want to show that a newly allocated data structure differs from those already allocated before execution of the function. We introduce a predicate that enables us to say for a number of heap locations at a time that they were allocated between two individual allocation states:

NewAlloc : *ref set* × *ref list* × *ref list* → *bool*
NewAlloc(**xs**, *oldalloc*, *newalloc*)
= (**xs** ∩ set *oldalloc* = ∅ ∧ (**xs** ∪ set *oldalloc*) ⊆ set *newalloc*)

The predicate takes a set of locations **xs** and two allocation lists. We say that a set of elements **xs** has been allocated between two allocation states iff the elements from **xs** were not contained in the old allocation list but are contained in the new one. Additionally, we always keep the previously allocated elements in our allocation list.

Frame Lemmas Based on the predicates we introduced, we are able to show lemmas to structure our proofs.

Definition: A *frame lemma* states for a data structure, a corresponding frame predicate and the fact that the data structure is not contained in the modified heap frame, that the data structure is still present on the modified heap.

We have a frame lemma for each of our frame predicates.

lemma BigIntsUnchanged_bn_notin_list:
(**bn** ∉ set **xs**
∧ BigIntsUnchanged **xs** *sign sign' size size' first first' last last' alloc*
∧ BigNumber **bn** *first last size sign next prev value* **ls num** *alloc*)
→ BigNumber **bn** *first' last' size' sign' next prev value* **ls num** *alloc*

lemma ListsUnchanged_intersect_empty:
(set **Ls** ∩ set **ls** = ∅
∧ ListsUnchanged **Ls** *next next' prev prev' value value' alloc*
∧ BigNumber **bn** *first last size sign next prev value* **ls num** *alloc*)
→ BigNumber **bn** *first last size sign next' prev' value'* **ls num** *alloc*

The frame lemma for the digit structure is more difficult to prove than that for the big number structure, because we show preservation of the list **ls** after the heap update.

What is the Benefit? Using this predicate framework we have created modular specification for our library functions. With the frame lemmas, it is possible to deal with heap modification without having to expand the definition of our predicates, i.e. the predicates can be treated as opaque by another module that uses our library.

To apply a frame lemma, we just need to show that our structure is different from all those that were modified, i.e. we show heap separation for these structures. The NewAlloc-predicate is used to show that structures that were allocated sequentially are different from each other.

5 Specification Examples

In the following, we use bold fonts to represent variables from the state space, italic fonts for heap functions and the allocation list and true-type fonts for values we quantify over. We use the notation $^{\sigma}\mathbf{a}$ to denote the value of the variable **a** in program state σ, in order to save space.

A specification always includes functional and pointer-level correctness as well as modified heap state information

5.1 Multiplication of two big integers

The function takes big integer locations **a**, **b** and **product** and stores the result of the multiplication of **a**'s value by **b**'s value in **product**.

Let `Ls1, Ls2` ∈ *ref list* and `num1, num2` ∈ *int*. Let $\sigma \in \Sigma$ be the program state before execution of the function.

---

b ≠ **product** ∧ **a** ≠ **product**
∧ **product** ≠ Null ∧ **product** ∈ set *alloc*
∧ BigNumber(**a**, *first*, *last*, *size*, *sign*, *next*, *prev*, *value*, `Ls1`, `num1`, *alloc*)
∧ BigNumber(**b**, *first*, *last*, *size*, *sign*, *next*, *prev*, *value*, `Ls2`, `num2`, *alloc*)

Function call: **retval** = bigIntMul(**a**,**b**,**product**)

(∃ Ls. **retval** = NO_ERROR
∧ BigNumber($^{\sigma}$**product**, *first*, *last*, *size*, *sign*, *next*, *prev*, *value*, Ls, (num1·num2), *alloc*)
∧ NewAlloc(set Ls, $^{\sigma}$*alloc*, *alloc*)
∧ ListsUnchanged([], $^{\sigma}$*next*, *next*, $^{\sigma}$*prev*, *prev*, $^{\sigma}$*value*, *value*, $^{\sigma}$*alloc*)
∧ BigIntsUnchanged([$^{\sigma}$**product**], $^{\sigma}$*sign*, *sign*, $^{\sigma}$*size*, *size*, $^{\sigma}$*first*, *first*, $^{\sigma}$*last*, *last*, $^{\sigma}$*alloc*))

In our precondition, we require that the heap locations **a** and **b** are different from the location **product** intended to store the multiplication result. Also, we expect that **product** is allocated and not the Null location. At last, we require that the heap locations **a** and **b** represent big numbers with the respective values num1 and num2.

After execution of the function, we claim that **retval** contains NO_ERROR and there is a new list Ls that belongs to **product**. At location **product**, we have a big number containing the value num1·num2. Also, the new list Ls was allocated during the function call. At last, no previously allocated list elements were modified and no previously allocated big numbers apart from **product** were changed.

5.2 Modulo Exponentiation

The function takes big integer locations **base**, **exp**, **n** and **remainder**. It computes the exponentiation result of **base** to the power of **exp** modulo **n**, storing the result in **remainder**.

Let Ls1, Ls2, Lsn ∈ *ref list* and b, e, nn ∈ *int*. Let $\sigma \in \Sigma$ be the program state before execution of the function.

BigNumber(**base**, *first*, *last*, *size*, *sign*, *next*, *prev*, *value*, Ls1, b, *alloc*)
∧ BigNumber(**exp**, *first*, *last*, *size*, *sign*, *next*, *prev*, *value*, Ls2, e, *alloc*)
∧ BigNumber(**n**, *first*, *last*, *size*, *sign*, *next*, *prev*, *value*, Lsn, nn, *alloc*)
∧ distinct [**remainder**, **exp**, **n**, **base**]
∧ 0 ≤ b ∧ 0 ≤ e ∧ 0 < nn
∧ **remainder** ≠ Null ∧ **remainder** ∈ set *alloc*

---

rval = bigIntModExp(**base**,**exp**,**n**,**remainder**)

(∃ Ls. **rval** = NO_ERROR
∧ NewAlloc(set Ls, $^{\sigma}$*alloc*, *alloc*)
∧ BigNumber(**remainder**, *first*, *last*, *size*, *sign*, *next*, *prev*, *value*, Ls, ($b^{nat\ e}$ mod nn), *alloc*)
∧ ListsUnchanged([], $^{\sigma}$*next*, *next*, $^{\sigma}$*prev*, *prev*, $^{\sigma}$*value*, *value*, $^{\sigma}$*alloc*)
∧ BigIntsUnchanged([**remainder**], $^{\sigma}$*sign*, *sign*, $^{\sigma}$*size*, *size*, $^{\sigma}$*first*, *first*, $^{\sigma}$*last*, *last*, $^{\sigma}$*alloc*))

We expect that there are big numbers at the mutually-distinct locations **base**, **exp** and **n**. Additionally, the values of **base** and **exp** are not negative and the value of **n** is positive while **remainder** is allocated and not the Null location.

After execution of the function, we have a new list Ls for **remainder** which represents the desired modulo exponentiation result. No allocated list elements were changed and no big numbers apart from **remainder** were modified.

6 Verification Results

Correctness of all functions of the big number package has successfully been verified with respect to partial correctness (i.e., termination has not been proven).

Observations Apart from finding good specification and invariants for the individual functions, the main difficulty we encountered during verification was formulating the correct frame predicates and lemmas. In the beginning we tried to use frame predicates that do not consider allocation. As we proceeded to verify the first function that uses temporary result variables (in our case, this was the multiplication routine) the problem with our predicates became obvious. When we came up with the new frame predicates that include allocation information, verification of the remaining functions became considerably easier.

Even though the employed heap model already resolves a significant amount of pointer aliasing conditions, the use of frame lemmas and predicates can and does take up the majority of space in our proofs (estimated 75 percent of the commands in the proof of the division and modulo routine DivMod deal with frame conditions or facts derived using them). On the one hand this depends on the number of function calls that are made in a function we want to verify correctness for. On the other hand it is very likely that we should have created additional lemmas that aid in frame lemma application early on.

In general, we noticed that verification is straightforward when the correct specification and invariant condition have been found. Usefulness of specification, however, can only be determined with full certainity when we have to use it during verification of another function. For most of the implemented functions we were able to confirm that our specification is effective for use in other functions.

Some Statistics In Fig. 3 we compare the proof effort for a few functions of the big integer package. We consider the functions InsertDigitBack (inserts a digit as least significant digit of a big integer), Mul (multiplication of two big integers), Div3DigitsBy2 (division of a three digit number by a two digit number, needed in quotient digit estimation), getQuotientDigit (calculates the first digit of the quotient of two big integers), DivMod (calculates both quotient and remainder of a big integer division) and ModExp (calculates the modulo result of the exponentiation result of two big integers).

In case of InsertDigitBack, most of the 121 proof lines deal with modification of the heap list belonging to its big number argument as well as the mathematical properties of digit insertion.

The proof of Div3DigitsBy2 is pretty much exclusively a mathematical one dealing with inequalities. However, these inequalities contain multiplication and division operations, thus, automatic tactics for Presburger arithmetic are not applicable. This led to a very detailed proof involving frequent application of low-level arithmetic rules by the user.

The function ModExp implements the simple square-

function	code	spec	inv	proof
`InsertDigitBack`	26	7	-	121
`Mul`	29	9	24	1313
`Div3DigitsBy2`	25	10	-	1283
`getQuotientDigit`	37	13	-	1629
`DivMod`	57	14	32	3008
`ModExp`	48	11	24	2191

Figure 3: Number of lines of program code, specification, invariant, and proof lines in Isabelle for a selection of functions.

and-multiply algorithm for fast exponentiation. Even though the algorithm is simple, the proof is very large. Actually, a large part of our proof consists of frame lemma application (with a depth of 8 calls before the while loop and a maximum depth of 6 heap updates during the while loop) since the function consists mainly of calls to other functions from the package. Still, correctness of this function was straightforward to verify and posed no particular problem. Verification of `ModExp` took us less than a week.

The main division routine `DivMod` was the greatest challenge we encountered in our work. We have a significant amount of function calls (7 function calls before the while loop, 6 calls in the while loop and three function calls after the while loop for the longest path of execution of the function) and some arithmetic properties related to big integer division which could not automatically be established (e.g., those dealing with operand normalization in division). The invariants were so large that the proof actions and undo information for them could not be stored in memory by the proof assistant. We solved this problem by splitting the proof into two parts, but this suggests that our efforts are pushing the limits of state-of-the-art higher-order logic tool support.

Conclusions This project gave some insight on the real complexity of heap data structure verification. One thing that should be noted is that the concept of frame predicates and lemmas can be generalized to any heap data structure.

Without any kind of support for heap aliasing conditions by the prover, different heap states accumulate quickly to the point where the sheer number of formulas slows down the prover or forces the user to apply specific rules 'by-hand'. In our eyes, the problem could be easily solved if the prover was able to group formulas concerned with heap updates after each function call. Additionally, all information that deals with distinctness of heap locations (e.g., disjointness of lists) needs to be aggregated to a normalized form. Then, using the frame lemmas and a few lemmas for the allocation predicate, the prover should easily be able to transfer a data structure from oldest heap to newest one, if it has not been modified.

7 Summary

We presented formal verification of a heap-based big integer library using the Hoare logic environment of Isabelle/HOL. For this task, we introduced predicates and lemmas to manage information about changes to the allocated heap during function calls. Specification of our functions was effective for those functions which call other functions from the big integer package.

References

[1] Homepage of the Isabelle theorem proving environment. http://isabelle.in.tum.de/.

[2] R. Bornat. Proving pointer programs in Hoare Logic. In R. Backhouse and J. Oliveira, editors, *Mathematics of Program Construction (MPC)*, volume 1837 of *LNCS*, pages 102–126. Springer, 2000.

[3] R. Burstall. Some techniques for proving correctness of programs which alter data structures. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 7, pages 23–50. Edinburgh University Press, 1972.

[4] B.-Y. E. Chang and X. Rival. Relational inductive shape analysis. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 247–260. ACM, 2008.

[5] S. Fischer. Formal Verification of a Big Integer Library Including Division. Master's thesis, Saarland University, 2007.

[6] D. Leinenbach, W. Paul, and E. Petrova. Towards the formal verification of a C0 compiler: Code generation and implementation correctness. In *3rd International Conference on Software Engineering and Formal Methods (SEFM)*, 2005.

[7] A. Møller and M. Schwartzbach. The pointer assertion logic engine, 2000.

[8] J. S. Moore. A mechanically verified language implementation. *Journal of Automated Reasoning*, 5(4):461–492, 1989.

[9] V. G. Nguiekom. Verifikation von doppelt verketteten Listen auf Pointerebene. Diplomarbeit, Saarland University, 2005.

[10] J. Reynolds. Separation logic: a logic for shared mutable data structures, 2002.

[11] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.

[12] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3–valued logic. In *Symposium on Principles of Programming Languages*, pages 105–118, 1999.

[13] N. Schirmer. Verification of Sequential Imperative Programs in Isabelle/HOL. Ph.D. thesis, Technical University of Munich, 2006.

[14] A. Starostin. Formal Verification of a C-Library for Strings. Master's Thesis, Saarland University, 2006.

[15] H. Tuch, G. Klein, and M. Norrish. Types, bytes, and separation logic. In M. Hofmann and M. Felleisen, editors, *Proc. 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07)*, pages 97–108, 2007.