Supporting Sequence Axiomatization on the SMT Solver Level for the Viper Project

Bachelor Thesis

Lukas Schär

supervised by
Arshavir Ter-Gabrielyan, Prof. Dr. Peter Müller
Chair of Programming Methodology
ETH Zürich

May 28, 2017

Contents

1 Introduction

1.1 Viper overview

The Viper Project is a collection of tools for software verification, aiming to allow verification of programs with persistent state, using reasoning in the style of separation logic [1]. It is based on the intermediate language Silver [2]. From there, the program is translated to the SMT[1] solver level, where the program will be checked for it's correctness. This can either be done by the verification condition generation back-end Carbon [3], or by the symbolic execution engine Silicon [4]. Carbon will translate the program further into another intermediate language, Boogie [5], which then uses the Z3 SMT solver [6] for verification. Silicon, in turn, uses Z3 directly.

1.2 Motivation

The power of an SMT solver is determined by the theories it incorporates. These include among others first order logic or linear arithmetic. Many further theories exist, and for software verification an important one is the theory of sequences. Sequences can represent connected data structures such as arrays or lists. As most meaningful programs contain such structures, it follows that Viper should incorporate such a theory. In the case of Silicon, this theory has so far been encoded via first-order axioms by Silicon itself. Z3 did not provide a theory of it's own. However, in version 4.5.0 [7] Z3 has received its own sequence theory.

Silicon's current sequence implementation can show incompleteness regarding sequences. Incompleteness means that the verifier is not able to prove the correctness of some actually correct programs. One approach to a possible solution of these weaknesses is to replace the current theory by the one of Z3.

1.3 Goal

This project modifies the existing Silicon implementation, such that it uses the new sequence theory provided by Z3. It also experimentally compares its performance and completeness to the existing approach. Baseline for this comparison was the Viper test suite, a collection of Silver files which cover multiple functional aspects of Viper. We investigated small Silver examples by hand in a first step and in a second step changed the implementation of sequences in Silicon. Lastly, we checked the new implementation with the Viper test suite and evaluated the results.

[1]Satisfiability Modulo Theory

2 Preliminaries

In this section we will introduce Viper's architecture and discuss the aspects of the software which are most important for this thesis.

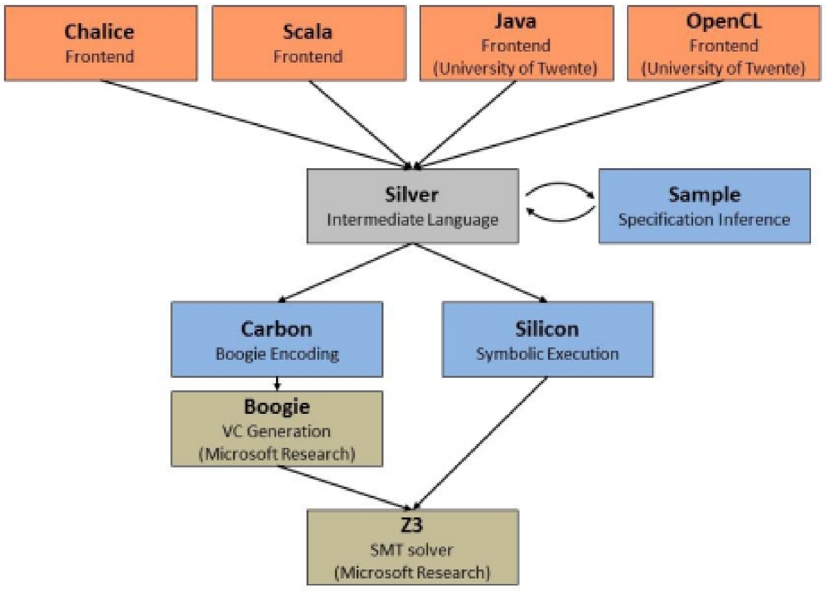

Figure 1: Overview of Viper's architecture [8]

2.1 Silver

Silver is the intermediate language of Viper. It reads very much like normal code, however it can be annotated to include contracts, meaning pre- and postconditions for methods and invariants for loops. These provide Viper with the information on what conditions should hold to consider the program correct.

In the most basic cases, Silver files are just collections of methods which should be verified. However, Silver also allows the definition of other constructs, such as predicates and functions. It is also possible to define domains, which are a way to define custom object types with fields and their own functions.

2.2 Z3

Z3 is Viper's SMT solver, based on the SMT-LIB 2.0 standard [9]. It is used in both of the possible back-ends of Viper and is effectively the heart of the verification. It uses a stack of user provided formulas and tries to find a model that satisfies all of these.

Z3 supports the definition of custom types, internally called *sorts*, and functions. A function is similar to a method in programming. It has a name, possibly several inputs of given sorts, and an output of a specific sort. They are however not implemented with statements. To determine their output, they have to be supported with assertions. These allow the user to specify what output is expected under some given conditions.

Listing 1 shows an example of such a function with a quantified assertion. In it, we define the function *foo*, which takes an integer and returns an integer. The assertion specifies that for all values of x, *foo* should return zero if x is negative, and 1 if x is non-negative. The pattern is used to determine when Z3 should actually instantiate the assertion, in this case whenever there is an expression of the form `foo x`. Without the assertion following the function declaration Z3 is free to choose any value for the function.

```
(declare-fun foo (Int) Int)
(assert (forall ((x Int)) (! (and
    (=> (< x 0) (= (foo x) 0))
    (=> (>= x 0) (= (foo x) 1)))
    :pattern((foo x))
    )))
```

Listing 1: Example SMT2 code quantified function.

2.3 Silicon

Silicon is one of Viper's back-ends, using a symbolic execution approach. To determine if assertions of a method in a Silver file hold, Silicon assumes the method's preconditions and all the knowledge it gathers by symbolically executing the statements up to the assert statement. After that it adds the negation of the assertion to the collected constraints and checks if these are now unsatisfiable. If that is the case, the assertion must hold.

To verify a program, Silicon interfaces with Z3. It generates the constraints that should hold in a method and uses Z3 to check their satisfiability. The first part of the constraints it provides to Z3 is the so called *preamble*. It sets up configuration parameters for Z3 and also produces the definitions of sorts that are not contained in Z3, for example sets and sequences. It also contains the function definitions and axioms for these types. Following this, it generates code for the functions and predicates.

Afterwards, Silicon verifies all methods in a given Viper file. When it encounters an error in a method, it stops verification of that method and continues on the next. This means, that not all errors may be found in a file, but if there are none, it is guaranteed to be true.

2.4 Sequences

So far, Silicon has used sequences which are implemented in a style analogous to the example in listing 1. The sequence type is a custom sort defined by Silicon, and the functions are custom defined functions which all have quantified assertions to define their behavior.

The sequences implemented by Z3 are based on a constructor. The constructor can take any sort sort known to Z3, be it built-in or custom defined. This creates a new sort. The functions are built in and .

3 Manual Investigation

The first step of the project was to investigate a very short and simple Viper example containing sequences. We translated it to SMT2 code and then transcribed it to the new syntax by hand. Silicon was executed with the Viper file as input to generate the Z3 code equivalent to it. This Z3 code was then manually rewritten to use the new sequence syntax of Z3, using the information from Z3's online sequence tutorial [10]. This phase was used as a first step towards understanding the new implementation, and to gather first statistics for performance and possible improvements in completeness over the existing implementation.

3.1 Selecting test cases

For the manual investigation, short examples from the Silicon and Silver issue trackers on Bitbucket were considered. This is due to the fact, that Z3 code can get relatively long and unreadable even for medium sized examples.

Eventually, we chose the method *m04simplified* from Silver issue 80 [11], found on Silver's issue tracker. It can also be found on the statistics gathering repository of this project [13]. It contains some sequence functions which could be replaced easily, and it exhibits erroneous behavior. The following list shows what functions were used:

- Declaration
- $Seq.singleton
- $Seq.length
- $Seq.contains
- $Seq.equal
- $Seq.length

3.2 Measurements

During this phase, we investigated if Z3's new sequence theory could provide the support to deliver correct results where the current version couldn't. We also compared the execution time of Z3 with both variants. To create the original version of the SMT2 code, Silicon was run with *–z3LogFile* command line argument, which allows the specification of a file which will contain all interactions of Silicon with Z3. To produce the modified file, we searched the file created in the previous step for all occurrences of the string

`"$Seq"`, which yields all places where a sequence was used. These places where then edited to produce semantically equivalent code. However, the preamble was not edited.

To check if the program is now executed correctly, we used the fact that Silicon's verification result is based on unsatisfiability, and that the method code is located at the very end of the SMT2 file. After Z3 assumes the negation of the assertion that should be checked, the check of satisfiability should return a negative result. The result of this check is emitted by Z3. Therefor, we could verify if the new implementation solves the behavior by simply running Z3 with the modified code again and checking if it returns *unsat*.

To evaluate the performance we executed Z3 several times with both the original and the modified SMT2 files as input. Because this test is only a few lines long and the verification time of one run is only very short, we chose a large number of runs to eliminate possible noise, in this case 5,000 times. The base tool for the time measurement was the Powershell *Measure-Command* cmdlet [12]. This cmdlet executes a given command and measures its execution time. The *measureManual.py* script [13] uses this cmdlet to run Z3 with both the original and modified SMT2. For each run it parses the timing output of *Measure-Command* and eventually sums them up and emits them into a csv file.

3.3 Interpretation of measurements

Running Z3 with the modified SMT2 file as input yielded the same output compared to using the original file. This means that the new Z3 sequences cannot solve this method's problem specifically. From this we concluded, that certainly not all problems will be solved by the new implementation.

The results of the performance measurements can be seen in table 1. As we can see, the difference in execution time is negligible with only 1 second over the course of 3 minutes of total execution time over 5,000 runs. This is not surprising, considering the small relative number of statements changed in the SMT2 file. In table 2 we can see the corresponding numbers.

Version	**Execution time in sec**
Original	181.4
Modified	180.1

Table 1: Total run time of 5,000 executions of Z3 with a given version of m04simplified.

Total statements	140
Changed statements	8
Changed functions	16

Table 2: Number of (changed) statements during manual investigation.

The conclusions from this phase of the project are, that there will be cases that did not verify in Silicon so far and still won't do so using the new theory. The next steps were the enabling of the new theory and automatic testing on larger examples.

4 Enabling Z3 Sequences in Viper

To support the new sequence theory, several changes to the code of Silicon had to be made. This section will describe these changes, and explain how the previously supported mathematical functions were replaced.

4.1 Replacement of the previous functions

Silicon provides 12 functions and a declaration syntax for sequences of all types. Additionally, there is the range function specific to integer sequences, which returns a half-open interval. To provide an adequate replacement, each of those needs to be implemented using the Z3 interfaces. Table 3 gives an overview of each function and how it was replaced. Note that *native* support in this table means that it is enough to replace the function name which Silicon provides with the equivalent Z3 function name to get semantically equivalent code. *Possible* support, on the other hand, means that the function can be replaced by a construct of different Z3 sequence functions to reach a semantically equivalent result.

Silicon function name	**Support by Z3**	**Notes**
\$Seq.append	native	via ”seq.++”
\$Seq.empty	native	via ”as seq.empty (Seq \$Type\$)”
\$Seq.equal	native	via ”=”
\$Seq.length	native	via ”seq.len”/aliased in Z3
\$Seq.singleton	native	via ”seq.unit”
Declaration	possible	implemented by Z3 sort alias
\$Seq.build	possible	unused in Silicon
\$Seq.contains	possible	aliased in Z3
\$Seq.drop	possible	
\$Seq.index	possible	implemented axiomatized
\$Seq.range	possible	implemented axiomatized
\$Seq.sameuntil	possible	unused in Silicon
\$Seq.take	possible	
\$Seq.update	possible	

Table 3: Overview of function support

The functions which have native support are replaced by their Z3 counterpart (which can be seen in the notes column of table 3) directly, except for *\$Seq.length*. The way it and the other non-native functions are implemented will be explained in the following passage. For each of these functions (except for *\$Seq.range*), there is either a small code example or the new axiomatization in appendix A. We also provide a small comparison of all functions following the explanations.

$Seq.length

Theoretically, *$Seq.length* is easily replaced by the function *seq.len* provided by Z3. However, using it directly with the *$Seq.range* function caused problems, which we will discuss in section 4.2. To avoid those, we rewrote the axiomatization of *$Seq.length* such that it always returns the same value as *seq.len* would.

Declaration

Although the declaration of sequences is natively supported in Z3, its syntax is problematic for Silicon. Therefor, rather than replacing the current syntax used to declare a sequence, we use Z3's feature to define a sort. The command *define-sort* allows us to use a custom sort symbol as another name for another sor in Z3 [14]. This makes Z3 use the existing declaration as an alias of its native sequences.

$Seq.build

Even though *$Seq.build* is not used currently in Silicon, it would be possible to use *seq.++* and *seq.unit* to construct a sequence generated from a sequence and a single element.

$Seq.index

This function can't be replaced directly, because Z3's equivalent, *seq.at*, does not return the value of the element at a specified position, but instead returns a sub-sequence of length one at that position. To solve this problem, *$Seq.index* is not replaced but instead it's axioms are rewritten. We introduce a new helper function, which returns the element of a length one sequence. We then let *$Seq.index* return the value of that helper function on the sub-sequence at the given position.

$Seq.update

This function is supposed to return a new sequence with the same contents as the sequence provided as argument, with the exception of the specified element, which should be changed. There is no Z3 function that can provide this functionality. The closest is *seq.replace*, but this replaces the first occurrence of a specified element. However, the *seq.extract* function can be used in conjunction with *seq.++* and *seq.unit* to construct a new sequence by concatenating the parts before and after the element that should be updated together with the length one sequence of the new value at the updated position.

$Seq.contains

Even though there is a function *seq.contains* in the Z3 syntax, it behaves slightly different to the one of Silicon. In the original version of Silicon, *$Seq.contains* checks if a single element is contained in a sequence. The Z3 implementation checks if a given sequence is a sub-sequence of another. Another issue is that *$Seq.contains* is also used in the axiomatization of *$Seq.range*, which leads to problems similar to the one we faced when replacing *$Seq.length*. To solve both issues, we again used an axiomatized approach: We didn't change the function itself, but instead replaced the axiomatization.

$Seq.take and $Seq.drop

There are no functions to create a sequence based on another sequence's beginning or ending. It is nevertheless possible to do achieve the same result with *seq.extract*. The function *seq.extract* allows to create a subsequence of a given sequence, starting from a given offset and including a specified number of elements. To replace *$Seq.take*, we can extract as many elements as we want to take, starting from position zero. If instead we want to drop, we extract from the offset given, and take as many elements that are following it, which is the length of sequence minus the offset. However, there appears to be non-deterministic behavior when using extract with some values. This will be discussed in section 4.2.

$Seq.sameuntil

This function is currently not used in Silicon, but it would still be possible to implement it using the function *seq.prefixof* together with *seq.extract*. They allow us to create a shortened sequence and check if the result is the prefix of another sequence.

$Seq.range

For this function, there is no simple replacement in the Z3 syntax and we had to reuse the axiomatized approach. We didn't change the function's representation in Z3 code and the axiomatization has stayed the same. Because we didn't change the code to use the functions for *length* or *contains*, the only sequence functions used in the axioms, no adjustments had to be made to the axioms. Since we added the alias for the declaration of a sequence, the return type did not have to be changed as well. Because of this, this function's code has seen no changes, but still returns a Z3 native sequence instead of the one that Silicon had provided so far.

Example

Listings 2 and 3 show an example of semantically equivalent code. As one can see, not all functions change, which is because of the reasons mentioned in the previous passages.

```
(declare-sort $Seq<Int>)
; ...  Many lines of preamble
(define-const s1 $Seq<Int>)
(assert ($Seq.equal s1 ($Seq.range 0 3)))
(assert (= ($Seq.length s) 3))
```

```
(assert (= ($Seq.index s 0) 0))
(assert ($Seq.equal
     s1 ($Seq.append s1 ($Seq.empty<Int>))))
(assert ($Seq.contains s1 1))
(assert ($Seq.equal s1 ($Seq.take 3)))
(assert ($Seq.equal ($Seq.drop s1 2) ($Seq.singleton 2)))
(assert ($Seq.equal s1 ($Seq.update s 1 1)))
```

Listing 2: Example smt2 code with old sequences

```
(define-sort $Seq<Int> () (Seq Int))
; ...  Many lines of preamble
(define-const s1 $Seq<Int>)
(assert (= s1 ($Seq.range 0 3)))
(assert (= ($Seq.length s) 3))
(assert (= ($Seq.index s 0) 0))
(assert (= s1 (seq.++ s1 (as seq.empty (Seq Int)))))
(assert ($Seq.contains s 1))
(assert (= s1 (seq.extract s 0 3)))
(assert (= (seq.extract s 2 (- ($Seq.length s1) 2)) (seq.unit 2)))
(assert (= s1
    (seq.++
        (seq.extract s 0 1)
        (seq.unit 1)
        (seq.extract s 1 (- ($Seq.length s1) 1))
    )))
```

Listing 3: Example SMT2 code with new sequences

4.2 Issues

During the implementation, two main issues came to light. The first was the problematic use of Z3's sequence methods in forall quantified assertions.

```
(assert (forall ((min Int) (max Int) (v Int) ) (! (and
(=> ($Seq.contains ($Seq.range min max) v) (and
   (<= min v)
   (< v max)))
   (=> (and
   (<= min v)
   (< v max)) ($Seq.contains ($Seq.range min max) v)))
   :pattern ( ($Seq.contains ($Seq.range min max) v))
   )))
```

Listing 4: Old contains axiom for $Seq.range

Consider listing 4. It is one of three axioms that describe the *$Seq.range* function and it states that all integers between the minimum and maximum value are contained in the range, and all values contained in the range have to be between the minimum and maximum. It would be easy to replace all three occurrences of *$Seq.contains* with the new *seq.contains* method, however when doing so, Silicon is not able to verify the simple example of listing 5. By introducing wrapper functions for used in the range function (c.f. Appendix A), this behavior was eliminated.

```
method m (s : Seq[Int])
    requires s == [0..3)
    ensures 2 in s
{}
```

Listing 5: Example for irregular behavior when using new functions

The second issue was the indeterminate behavior of *seq.extract* when extracting from offsets that are negative or larger than the length of the sequence we want to extract from; the same happens also if the number of elements we want to extract is out of these bounds. Listing 6 shows an example where this behavior can be observed.

```
(declare-const s (Seq Int))
(declare-const t (Seq Int))
(declare-const u (Seq Int))
(declare-const i Int)
(assert (= s (seq.++ (seq.unit 1) (seq.unit 2) (seq.unit 3))))
(assert (= t (seq.extract s 0 i)))
(assert (= u (seq.extract s i (- 3 i))))
(assert (not (= s (seq.++ t u))))
(check-sat)
(get-model)
```

Listing 6: Example for irregular behavior of seq.extract

It defines three integer sequences and an integer which will be used as an index for the extraction. We define s to be the sequence of consecutive integers one, two and three and the sequences t and u are extracts from said sequence, one starting from zero and taking i elements, the other starting at element i and extracting as many as are left in the sequence.

We then want to prove that the concatenation of those extracts is equal to the original sequence. For that we assume that s and the concatenation of t and u are not equal and we expect that Z3 is not able to find a model that satisfies these constraints. However, Z3 will output a model in which the index i is negative and the sequences t and u are sequences of random numbers. This is something that could not be solved during this project. It's consequences will be discussed in section 6.

4.3 Code changes

This section will outline which parts of Silicon's code base had to be changed in order to enable the new sequence theory. We will briefly list what problems had to be solved in the source classes and resource files of Silicon and Silver to implement the new sequence interface. The actual changes can be found on the Bitbucket repositories of this project [15–17].

4.3.1 Silicon classes

Silicon's classes contain the logic on how to emit the SMT2 code. The modifications to the code include the changes to actually emit the new functions, and adjustments to the order in which collections are defined in the SMT2 file. This was necessary to avoid problems with the aliasing used for the sequence declarations. Additionally to the listed changes, we also renamed the unit test classes (apart from SiliconTests.scala), such that they would not be executed during testing.

TermToSMTLib2Converter.scala : Responsible for generating Z3 code. The changes here are necessary to actually emit the new function syntax.

Sequences.scala : Responsible for Z3 code specific to sequences, such as loading axiom templates and declaring the sort. The changes here replace the definition of the Silicon generated sequences with an alias command, which maps the current syntax to the Z3 native sequences.

DefaultMasterVerifier.scala : Responsible for the general flow of verifying a file. The changes in this class cause sequences to be defined after other types. This is necessary because aliasing will not work if we for example want to define sequences of sets, and sets are not defined yet.

Note that it is not a problem to define, for example, a set of sequences before defining the sequence. This is because Silicon's sets are not based on a constructor like Z3's sequences. If we define the type `Set<$Seq<Int>>`, to Z3 this is just one long name of a sort. It does not realize that these are nested types for Silicon. The problems only start when `$Seq<Int>` is used as a sort for a function and it is not yet defined. However, the first usage of this type happens when the axioms are defined. At this point, all sorts have already been defined.

SiliconTests.scala : Unit test which runs the Viper test suite. The changes here lead to the reduced size of the test suite. Now, instead of verifying all test cases contained in Silver's resource folder, the unit test will only verify cases which are located in the *sequencesuite* directory.

4.3.2 Silicon resources

Silicon's resources contain the template files from which the axioms of the sequence functions are generated. As we have seen in section 4.1, it is not possible to remove all existing axioms. The files in this category had to be changed, so that they emit the new axioms.

4.3.3 Silver classes

Silver was only slightly modified to allow collection of timing statistics for the different phases of execution without having to rely on the execution of unit tests.

4.3.4 Silver resources

No files were modified, however several test cases were duplicated and the new folder *sequencesuite* was created. This allows the testing of a test suite reduced to the relevant cases (c.f section 5.3).

5 Experimental Procedure

This section describes how the new Z3 sequence theory was compared to the axiomatized approach provided by Silicon with the Viper test suite. The results of the measurements will be presented and analyzed in section 6.

5.1 Testing system

5.1.1 Hardware

CPU Intel(R) Core(TM) i7-5500U @2.4GHz

RAM 8.00 GB

OS Windows 7 Professional (64 Bit)

5.1.2 Software

Silicon (reference) forked at commit 51a0afd, changed up to commit c881666 [15]

Silicon (new version) forked at commit 51a0afd, changed up to commit ed76d51 [16]

Silver forked at commit e5aebac, changed up to commit fcc9e0e [17]

Z3 version 4.5.0 [18]

Python version 3.6.1 [19]

Powershell version 2.0 [20]

IntelliJ IDEA version 2017.1 [21]

Nailgun version 9.2.0 SNAPSHOT [22]

5.2 Selecting test cases

The Viper project provides a large number of test cases which cover multiple aspects of the verifier. In this section we discuss the way it was used to test the new sequence implementation in Silicon.

Not all test cases are useful for comparing the new implementation to the original. Obviously only those that contain sequences may show significant differences. For all other cases the two back-ends should produce the same output. To select the interesting cases, IntelliJ IDEA's project search function was used. All Silver test files in Silver's and Silicon's test resource folder were searched for the string "`Seq[`". This returns the list of all files in which a sequence was defined.

Note that this may not include every case containing sequences. Listing 7 shows that it is possible for a test to contain sequences without explicit declarations. However, we still obtained a wide selection of cases.

```
method m(x : Int)
    requires x < 0
    ensures ! (x in [0..10))
{}
```

Listing 7: Silver example containing sequences without explicit declaration.

We removed some cases from the resulting list, since they are expected to fail at the type checker level, and therefore no actual verification would take place during Silicon's execution. We also removed a case that fails in the current version of Silicon. Additionally, several cases are ignored or canceled during the testing in the used reference version of Silicon. These cases were excluded as well. This leaves us with 166 cases to test with. As an addition to that, a small extra case was written to test the update function, because barely any of the other cases contained applications of it. The selected test cases can be found in the *AllTests* directory on the same repository where the measurement scripts are located [13].

Test name	**# lines**	**# methods**
0071.sil	207	5
0120a.sil	82	1
binarySearchSeq.sil	28	1
foralls.sil	40	6
issue_0142.sil	37	3
list_insert.sil	108	1
RingBufferRd.sil	133	6
test_list.sil	56	2
tree_delete_min_no_assert.sil	76	1
update.sil	20	1

Table 4: First set of tests.

From this list, we selected a small subset of tests which we investigated in more detail. Table 4 shows which cases these are, as well as some information on them. This smaller set allowed for a very focused evaluation based on different factors, such as number of methods and used sequence functions. The files are also located on the repository mentioned in the previous paragraph in the directory *SampleSet*. Note that they are also contained in the *AllTests* directory.

5.3 Measurements

We were mainly interested in the timing statistics of the actual verification phase of Silicon, however we also compared the preceding phases, which are parsing, type-checking and translation. Note that translation is the name of the phase, but that name is chosen a bit isn't ideal, as this phase is only used to filter the provided program based on command-line arguments. Additionally, we are interested in the pure execution time of Z3, since the verification time of Silicon includes interactions between Z3 and Silicon. The phase timings are put out by Silicon when executed, the execution time of Z3 is gathered by running it separately. Lastly, we wanted to investigate the memory usage of Z3 when using the new version, and compare it to the original version's memory footprint.

To collect this data, we used the script *nailgunTest.py* [13]. It gathers the timings in nanoseconds for each version of Silicon separately. It launches a nailgun server that is supported with a fat jar file of the chosen version of Silicon. This creates a JVM that stays open until aborted. The script then runs the main class of Silicon with arguments set to produce named SMT2 files and to only utilize one CPU core. We only use one core to eliminate indeterminacies that could emerge from parallel execution. This also assures that all methods are verified by one instance of Z3 and therefor all logging information is passed into one file, which can later be verified by Z3 directly. We also pass an argument to set Z3's timeout to a value of several seconds. This makes sure that the execution will eventually terminate, should something go wrong with the verification.

Using nailgun eliminates the startup time of the JVM. Because the JVM is kept running, the dynamic compiler will tune the optimization during the multiple executions of a test. Therefor, we run each test 30 times to eliminate the noise produced by the optimization. After this, we run another 10 runs, whose output will be written to files. From there the script will later collect and average the timings.

To gather the timing information of Z3's execution time, the script runs the Powershell *Measure-Command* cmdlet, which measures in microseconds. For each case, the SMT2 file generated earlier in the run is executed 10 times and the runtime measured and averaged. The results are compiled and then written to a csv file.

For measuring the completeness of the implementation, sbt's *test* task [23] was used, which runs Silicon's unit tests. The unit test *SiliconTests.scala* verifies all Viper files located in Silver's test resource directory *sequencesuite*. These files can be annotated to inform the test suite that certain errors are expected. The unit test then classifies a case as successful if it satisfies these annotations, or if it verifies as correct if there are no annotations. Otherwise the case is considered a failure. The script *sbtTest.py* runs sbt and parses the output. It produces a csv file containing the results of running sbt test for both versions of Silicon.

To collect the information on memory usage, we used Z3's built in statistics tool. The command `(get-info :all-statistics)` from Z3 will print out a set of statistics, which also contain the maximum amount of memory used in megabytes. To gather these statistics, we used the *memoryTest.py* [13] script. It runs a test case through Silicon to create the SMT2 code, which is saved to a file. The script then edits the file to include the statistics command at the end of execution. Z3 is then executed with the edited file as input, and the out put is parsed to extract the maximum memory usage. The script runs 10 executions and averages their memory. Afterwards it emits them in a csv file.

6 Interpretation of Experimental Results

In this section we present the results of the experiments of section 5 and interpret their meaning. We also discuss possible reasons for some of the results.

For this discussion, we distribute the tested cases in three different groups, based on the correctness of their verification. The first group contains all test cases for which the result of the verification matches between both versions of Silicon. The second group contains the cases for which the result of the verification did no match. The last group consists of cases, for which Z3 runs into non-termination when it is queried by the new version of Silicon. The distribution of these cases can be seen in table 5.

Status	**# of cases**	**% of cases**
Matching	150	89.8
Mismatching	12	7.2
Non-terminating	5	3.0
Total	167	100

Table 5: Number of cases in each category of verification status.

6.1 Matching results

From the 167 cases we evaluated, a total of 150 returned the same result in the new version of Silicon as in the reference version. Included in those are the 10 tests we chose to investigate in more detail. We will first look into the results of these tests. The overall results will be considered afterwards.

Sample set

Pre-verification time

Phase	Geometric mean of relative time
Parsing	105.22%
Type-checking	99.13%
Translation	99.77%

Table 6: Geometric means of relative time of the new version of Silicon compared to the original version for different phases (sample set).

The evaluation of the three phases preceding verification revealed, that the elapsed time for each phase is very similar in both tested versions of Silicon. The relevant data can be seen in table 6. These results suggest, that the pre-verification times are not affected by the changes of the sequence implementation.

Verification time

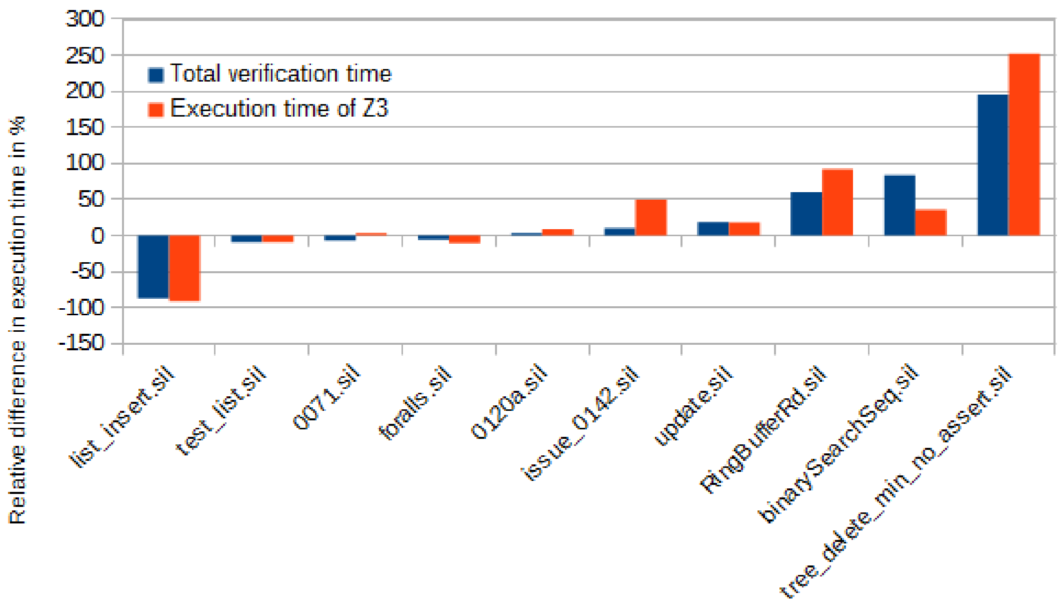

Figure 2: S-curve: Relative difference in verification time of the new version of Silicon compared to the reference implementation (sample set).

Figure 2 shows how the verification time of the new version of Silicon compares to the original version. It also shows the relative execution time of Z3 when it is executed independently with an SMT2 file generated by the new version of Silicon. The geometric mean of the relative verification time of Silicon is 101.79%, the geometric mean of the relative pure verification time of Z3 is 103.08%.

These values suggest that the performance of the new version is very similar to the reference version. However, there are some cases which show remarkable differences. We investigated the source code of the two most extreme cases, *list_insert.sil* and *tree_delete_min_no_assert.sil*, in an attempt to suggest possible reasons for this behavior.

Both cases contain very similar applications of sequence functions. The biggest difference is the complexity of the used predicates and magic wands. It is possible, that the sequence functions interact poorly with these magic wands.

Memory usage

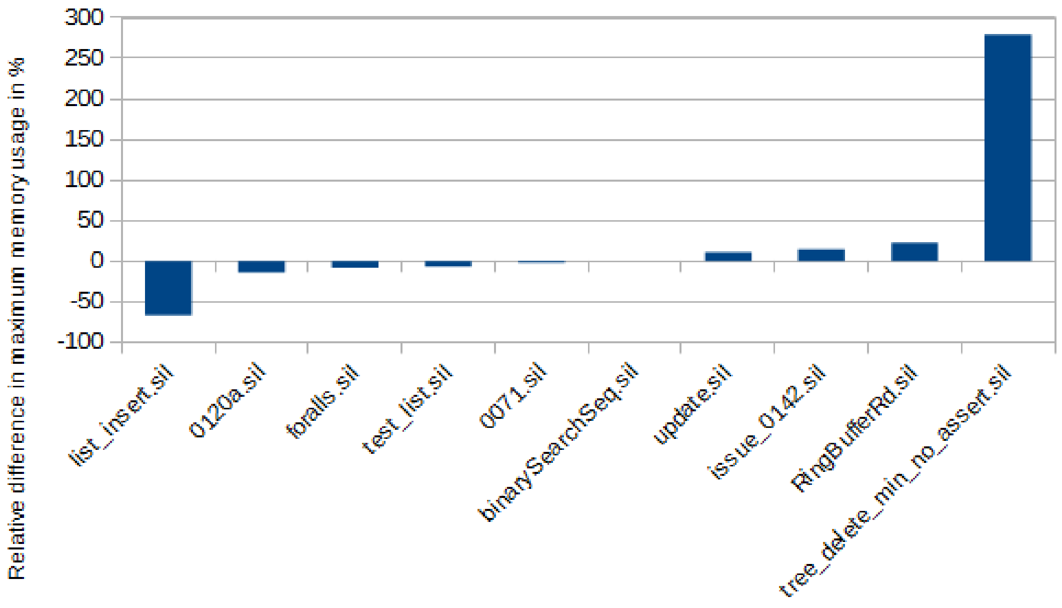

Figure 3: S-curve: Relative difference of maximum memory usage of Z3 when verifying a test with the new version of Silicon compared to the reference implementation (sample set).

The investigations of the memory usage of Z3 show overall very similar values for both versions. The geometric mean of the relative amount of memory used in the new version is 103.54%. Nevertheless, we see the same outliers we observed during the evaluation of the verification time.

Phase	Geometric mean of relative time
Parsing	103.30%
Type-checking	98.43%
Translation	97.41%

Table 7: Geometric means of relative time of the new version of Silicon compared to the original version for different phases (all matching tests).

All tests

Pre-verification time

The measurements over all matching test cases were very similar to the sample set. As we can see in table 7, the geometric mean for each phase is very close to 100%. This suggests, that the results from the sample set are not arbitrary, but actually relevant, and that these phases were actually not affected by the changes.

Verification time

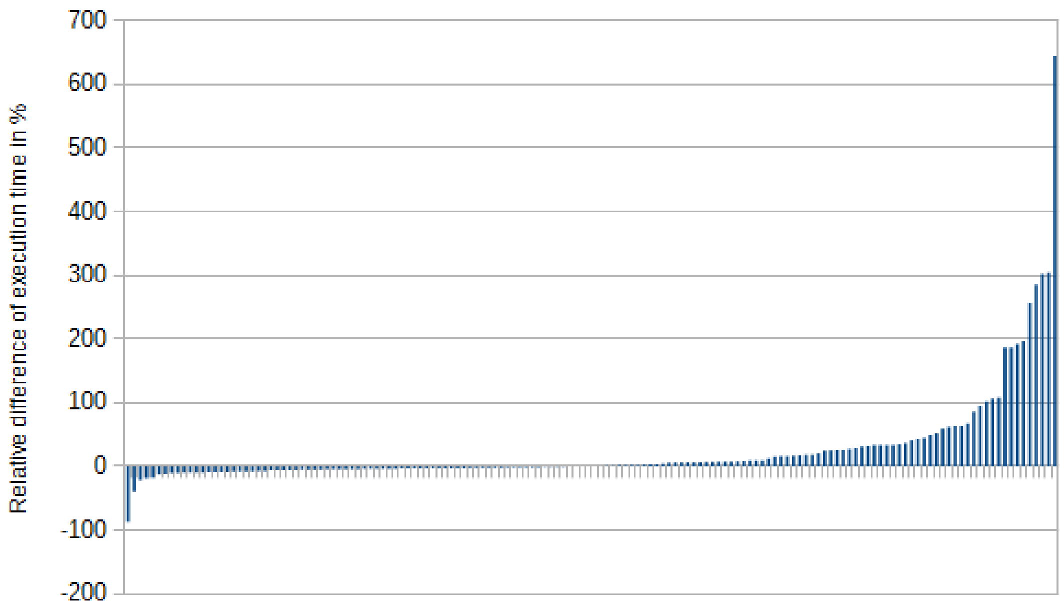

Figure 4: S-curve: Relative difference in total verification time of the new version of Silicon compared to the reference version (all matching tests).

Figure 4 shows the relative difference in verification time from the new version of Silicon compared to the original. As we can see, there are again large outliers in both directions. To compare the overall performance of the new system, we consider the geometric mean of the relative execution time of the modified version of Silicon, which in this case is 114.07%.

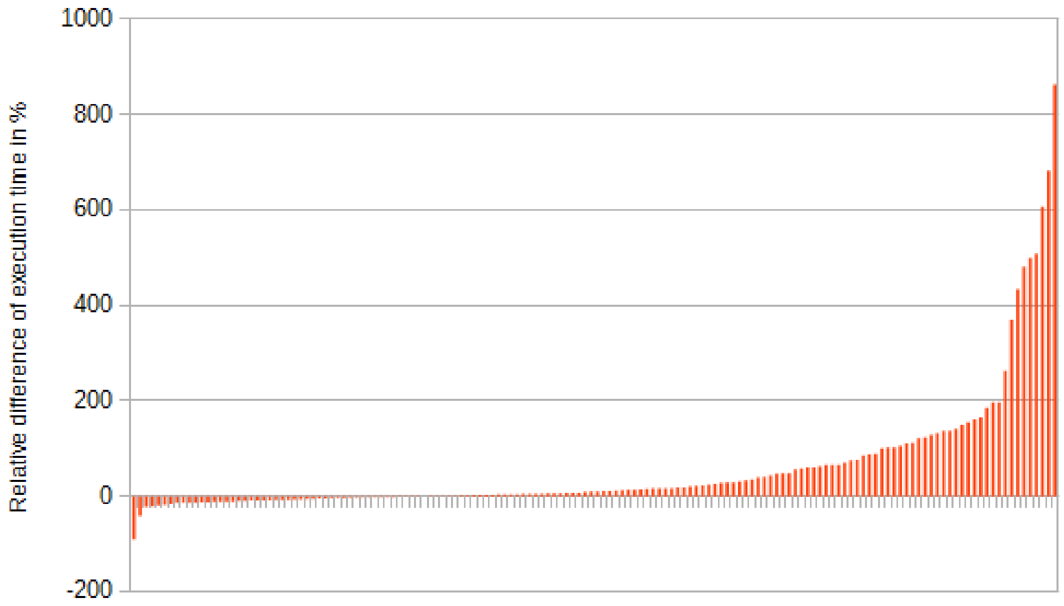

Figure 5: S-curve: Relative difference in runtime of Z3 when verifying with the new version of Silicon compared to the old (all matching tests).

Figure 5 shows the relative differences in runtime for all matching cases. We once again get a similar curve form. The geometric mean of the relative execution time is 131.89%. This is significantly higher than the value we got for the total verification phase time in the previous paragraph.

This difference can be explained by the interaction time between Silicon and Z3. The measurements showed that the difference between the total verification time of Silicon and the pure Z3 verification time is usually several 100 milliseconds for both versions of Silicon. This happens no matter how long Z3 takes to verify. For very short tests, this time can be quite significant and the pure execution time becomes insignificant, which leads to similar relative times, which means less relative difference.

Memory usage

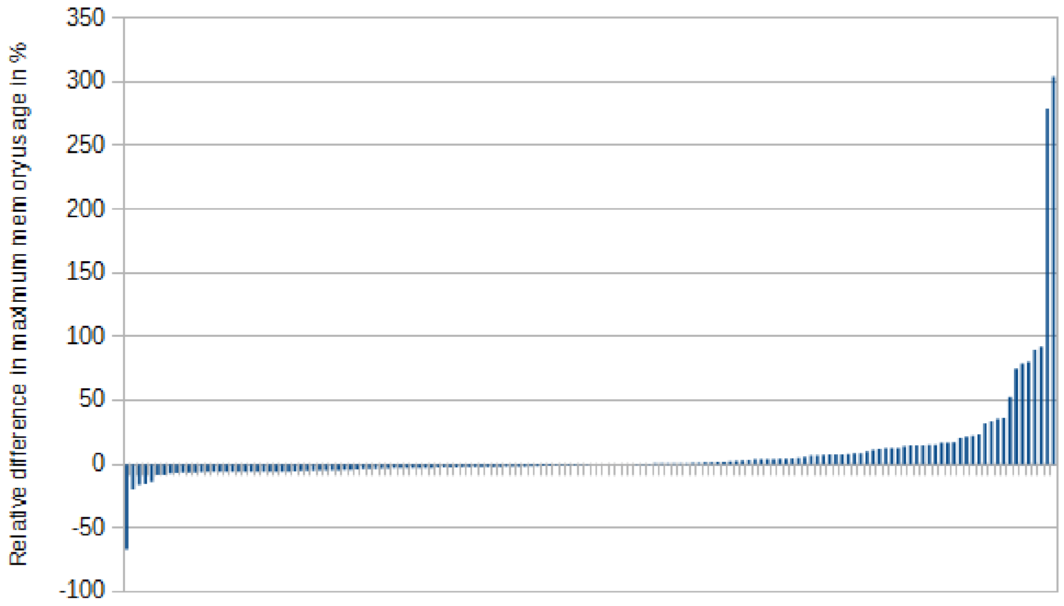

Figure 6: S-curve: Relative difference of maximum memory usage of Z3 when verifying a test with the new version of Silicon compared to the reference implementation (All matching tests).

The investigation of the memory usage over all matching tests show a behavior comparable to the behavior of the verification time. We get the same outliers and overall a very similar curve form. The outliers don't go as high as the times, though. The geometric mean of the relative memory usage is 104.53%. This suggests that overall, the memory usage is not largely different between both versions of Silicon.

6.2 Mismatching results

The results for 12 cases were not the same in the modified version of Silicon compared to the reference version. For these cases we omit the performance analysis. We do this because Silicon stops verification of a method in case of an assertion it can't verify. This means that the parts that actually get verified could be much shorter in the failing case, especially for large methods. Instead we investigate the cases to find possible reasons why they failed.

The failing cases are:

- 0076.sil
- arrays_quickselect_rec.sil
- arrays_quickselect_rec_index-shifting.sil
- issue_0060.sil
- issue_0176.sil
- linked-list-predicates.sil
- linked-list-qp-append.sil
- linkedlists.sil
- test2.sil
- testTreeWand.sil
- testTreeWandE1.sil
- tree_delete_min_heuristics.sil

A reappearing reason of failure is the inability to preserve loop invariants or guarantee them on entry. In each of these cases, there is a index function involved, which could mean that the new implementation of it is not perfect yet.

In the case *linked-list-predicates.sil* multiple postconditions including take and drop operations fail. The same is true for the case *linkedlists.sil.* It is possible that the implementation of these functions somehow causes problems. This could also be related to the issues we faced in section 4.2.

Another reason of failure seems to be the insufficiency of access permission. It is possible that the new implementation of sequences interacts badly with the implementation of permissions.

6.3 Non-termination

The cases in this category behave normal when using the original Silicon sequence implementation, however they don't terminate when running *sbt test* with the new sequence implementation. There are also some cases that only sporadically show the non-terminating behavior, or don't show it at all if they are not run through *sbt test* but verified by Silicon directly. This non termination hints at a matching loop, a recursive instantiation of the same axioms, somewhere during the execution. These cases were also not considered for performance evaluation, we instead investigated the cases and tried to find possible reasons for the non-termination. The tests in this category are the following:

- issue_0124.sil
- issue_0139.sil
- testTreeRecursive.sil
- tree_delete_min(2).sil
- tree_delete_min.sil

We could not find a distinct feature that is used in these cases, which other cases of the test suite don't use. However, they all have quantified assertions or recursive functions, which could be a possible cause.

Looking at the cases which only sometimes terminate showed, that the SMT2 code of a non-terminating case only differs slightly from the terminating case. The case *tree_delete_min.sil* terminated when using Silicon directly and restricting it to one core. When verified with *sbt test*, the only method of the test was assigned to the fourth core of the test system. In this case, Silicon uses different suffixes for the variables it uses in Z3. For example, the variable *x@0@01* was now called *x@0@04*. This was the only difference in the code.

We can only speculate why this small change leads to non-termination. It could be that the different name of the variable changes the heuristics of Z3 and this makes it not avoid the matching loop. The influence of variable names on execution time in Z3 has been shown before [24], perhaps the detection of matching loops is affected similarly.

7 Conclusion

We modified Silicon and replaced all used sequence functions with equivalent functions or combinations thereof in the new Z3 syntax. We also investigated the new implementation with respect on performance and completeness.

We investigated the time consumption of the different phases of Silicon and compared how they change in the new implementation. As expected, the measured differences in the pre-verification phases were very minor, and can probably be attributed to noise. This assumption is strengthened by the fact that no code concerning these phases was changed.

For the actual verification times, the experiments have shown outliers in both directions. There are cases that are verified significantly faster when using the modified version of Silicon. However, there are also many cases for which the new solution is much slower than the existing implementation. We have also seen that the overall performance seems to be slower, since the geometric mean of the relative execution time of the new version compared to the reference version is significantly over 100%. The average memory usage stayed about the same for both versions of Silicon.

We also saw that the new implementation shows weaknesses regarding completeness. There are some cases which do not verify as expected or do not even terminate with the new version of Silicon. The reasons for this behavior have not been fully uncovered, however, we have discussed some of the possible reasons. The further investigation of these problems could be the entry point for future work.

Acknowledgments

At last, I want to thank all people who supported me during this thesis. I want to thank especially Arshavir Ter-Gabrielyan for his input while supervising the project.

8 References

[1] Viper, `http://www.pm.inf.ethz.ch/research/viper.html`, May 2017

[2] Silver on Bitbucket, `https://bitbucket.org/viperproject/silver`, May 2017

[3] Carbon on Bitbucket, `https://bitbucket.org/viperproject/carbon`, May 2017

[4] Silicon on Bitbucket, `https://bitbucket.org/viperproject/silicon`, May 2017

[5] Boogie, `https://www.microsoft.com/en-us/research/project/boogie-an\-intermediate-verification-language`, May 2017

[6] Z3 on Github, `https://github.com/Z3Prover/z3`, May 2017

[7] Z3 Release Notes, `https://github.com/Z3Prover/z3/blob/master/RELEASE_NOTES`, May 2017

[8] Viper's architecture, `http://www.pm.inf.ethz.ch/research/viper/_jcr_content/par/fullwidthimage_0/image.imageformat.fullwidth.1838601289.png`, May 2017

[9] Z3 - Guide, Introduction, `http://rise4fun.com/Z3/tutorialcontent/guide#h21`, May 2017

[10] Z3 - Sequences, `http://rise4fun.com/Z3/tutorial/sequences`, May 2017

[11] Silver issue 80 on Bitbucket, `https://bitbucket.org/viperproject/silver/issues/80/`, May 2017

[12] Using the Measure-Command Cmdlet, `https://technet.microsoft.com/en-us/library/ee176899.aspx`, May 2017

[13] Scripts to gather statistics for Silicon on Bitbucket, `https://bitbucket.org/schaerl/siliconstatsgathering`, May 2017

[14] Z3 - guide, `http://rise4fun.com/Z3/tutorial`, May 2017

[15] Fork of reference version of Silicon on Bitbucket, `https://bitbucket.org/schaerl/siliconreference`, May 2017

[16] Fork of modified version of Silicon on Bitbucket, `https://bitbucket.org/schaerl/siliconnewz3sequences`, May 2017

[17] Fork of Silver on Bitbucket, `https://bitbucket.org/schaerl/silverreference`, May 2017

[18] Releases of Z3 on GitHub, `https://github.com/Z3Prover/z3/releases`, May 2017

[19] Python, `https://www.python.org/`, May 2017

[20] Powershell, `https://msdn.microsoft.com/en-us/powershell/`, May 2017

[21] IntelliJ IDEA, `https://www.jetbrains.com/idea/`, May 2017

[22] Nailgun, `http://www.martiansoftware.com/nailgun/`, May 2017

[23] sbt documentation: Testing, `http://www.scala-sbt.org/0.12.4/docs/Detailed-Topics/Testing.html`, May 2017

[24] Z3 issue 909 on GitHub, `https://github.com/Z3Prover/z3/issues/909`, May 2017

A Z3 examples

For the following sections we assume `s1` and `s2` to be sequences of integers and `x` to be an integer.

If the axiomatization is shown, it will be a generic example where the type of the sequence is called `$Type$`. To get the code for integers, for example, all occurrences of `$Type$` have to be replaced with `Int`.

Declaration

The new way to declare a sequence of a type looks as follows:

```
(declare-const s1 (Seq $Type$))
```

Since we can't change to this syntax due to problematic interactions with other code Silicon produces, we change the definition of the custom sort; instead of declaring a new sort, we make it an alias for Z3's native sequences:

Old definition

```
(declare-sort $Seq<$Type$>)
```

New Definition

```
(define-sort $Seq<$Type$> () (Seq $Type$))
```

$Seq.build

In this example we want to assert that s2 equals s1 concatenated with x.

Original code

```
(assert ($Seq.equal s2 ($Seq.build s1 x)))
```

New code

```
(assert (= s2 (seq.++ s1 (seq.unit x))))
```

$Seq.index

Since the code emitted for this function has not changed, the new helper function and axiomatization will be listed.

Helper function

```
(declare-fun $Seq.get ((Seq $Type$)) $Type$)
```

New axiomatization

```
(assert (forall ((xs (Seq $Type$)) (i Int)) (!
    (=
        ($Seq.index xs i)
        ($Seq.get (seq.at xs i)))
    :pattern (($Seq.index xs i))
     )))
assert (forall ((xs (Seq $Type$))) (!
    (and
        (= (seq.unit ($Seq.get xs)) xs)
        (seq.contains xs (seq.unit ($Seq.get xs)) ))
    :pattern (($Seq.get xs))
  )))
```

$Seq.update

In the following example we assume that s1 and s2 are integer sequences and x is a integer. We want to assert, that s2 equals s1 except for position x, which should be updated to 5.

Original code

```
(assert ($Seq.equal s2 ($Seq.update s1 x 5)))
```

New code

```
(assert (= s2
    (seq.++
        (seq.extract s1 0 x)
        (seq.unit 5)
        (seq.extract s (+ x 1) (- (seq.len s) x))
 )))
```

Note that we use the possibility of concatenating multiple sequences at once of Z3's append function.

$Seq.contains

Since the emitted code for this function has not changed, the new axiomatization will be listed.

New axiomatization

```
(assert (forall ((xs (Seq $Type$)) (x $Type$)) (!
    (=
        ($Seq.contains xs x)
        (seq.contains xs (seq.unit x)))
    :pattern (($Seq.contains xs x))
  )))
```

$Seq.take

In this example we want to assert that s2 is equal to the first x elements of s1.

Original code

```
(assert ($Seq.equal s2 ($Seq.take s1 x)))
```

New code

```
(assert (= s2 (seq.extract s1 0 x)))
```

$Seq.drop

In this example we want to assert that s2 is equal to s1 dropping the first x elements.

Original code

```
(assert ($Seq.equal s2 ($Seq.drop s1 x)))
```

New code

```
(assert (= s2 (seq.extract s1 x (- (seq.len s1) x))))
```

$Seq.sameuntil

In this example we want to assert that s1 and s2 are equal up to position x.

Original code

```
(assert ($Seq.sameuntil s1 s2 x))
```

New code

```
(assert (seq.prefixof (seq.extract s1 0 x) s2))
```

B Declaration of originality

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Supporting Sequence Axiomatization on the SMT Solver Level for the Viper Project

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Schär	Lukas

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Vordemwald, 28.05.2017	[illegible]

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.