Encoding hashes in an SMT-solver using Z3

Sky Sarah van Grieken

Techincal University of Eindhoven, 5612 AZ Eindhoven, The Netherlands
s.s.v.grieken@student.tue.nl

Abstract. Model checking is an important way to check if a program is working correctly. Models can contain constraints that need to be solved. The constraint solving software SMTreesolver currently can only solve equality constraints on hashes. In this paper we research how to encode hashes in Z3 such that other types of constraints can also be applied. We came up with three possible hash encodings in Z3: using sequences, using declare-function and using arrays. We conclude that from these three ways, using arrays in Z3 is the best way to encode hashes.

1 Introduction

Testing software is important, but writing tests is a lot of work. Axini automates this process by creating a Software Under Test model, which is used to automatically generate tests for testing software. This is called model-based testing. One way of model-based testing is to use transition-based systems. In such a system, transitions between states describe the actions that a system can take. Those transitions can hold constraints. Currently in Axini, GProlog is used to solve these constraints. An alternative for using GProlog, is using satisfiability modulo theories solvers (SMT-solvers). To test if this is more efficient (in some cases), Florine de Geus made an SMT-solver using Z3 for Axini [1]. This first version of the solver converts all non-numerical types to integers before solving the constraints on them. Z3 is capable of handeling all sorts of data types, so the logical next step is to add one of the types of constraints: hash functionality.

In this report we research what the possible ways are to encode hashes in Z3, which ways are the best, and how to implement this in the current SMT-solver.

2 Preliminaries

In this section, the required knowledge to understand this work is explained.

2.1 Z3

Boolean satisfiability problems (SAT) are problems about deciding if there exists a solution for a boolean formula. Satisfiability Modulo Theories problems (SMT) extend these problems with arithmatic, quantifiers, arrays and more first-order theories. Z3 can solve these SMT problems [2].

2.2 Hash

In this paper we see a hash as a list of mappings from one type of value to another type of value. This first type of value is the 'key', as input for a hash, and after input the hash returns the corresponding 'value'. The type of these keys and values can be of any kind, also hashes themselves. If a value type is a hash, the complete hash is called a nested hash. A hash can have more than one element. An example of a nested hash with two elements is:

```
1 h = {"a" => {1 => 1.0, 2 => 2.0}, "b" => {3 => 3.0}}
```

Let us define the term 'end-value' of a hash as follows: the value of a hash after processing keys in the hash until the value type is not a hash. In the example above, the end-values are 1.0, 2.0 and 3.0.

3 Ideal Hash Constraints

Before we research how to encode a hash in Z3, it is important to know what types of constraints we want to be able to put on those hashes. In this way, we can immediately know when researching the possible encodings if they will be a success.

Value-constraint We want to be able to put constraints on the values of hashes. For example: the constraint $h[\text{"}a\text{"}] > 1$, would give a hash $h = \{\text{"}a\text{"} => 2\}$ or any other number greater than 1 instead of 2.

Length-constraint Constraining the total number of elements in a hash is also important. This can be expressed as the length of a hash. For example: the constraint $length(h) > 0$ would return a hash with at least one element in it. Note that it is important for Z3 to know what the type of this hash is, otherwise it cannot create elements.

Total-constraint Finally, putting constraints on the complete hashes themselves can be significant. These kind of constraints would be of the equality type. For example: the constraint $h \neq expected\_hash$ will ensure that h is not the same as $expected\_hash$.

4 Hash Encodings in Z3

Since there is no datatype for hashes in Z3, we must find a ways to encode hashes with the existing datatypes. We investigate three possible methods to encode hashes in Z3.

4.1 Sequences

The first method for encoding hashes in Z3 is using sequences. A sequence in Z3 is similar to how an array would be described by programmers: a number of elements with the same type are stored together and can be accessed by integers starting from 0. Because of this indexing, the length of the sequence can easily be obtained by a build-in function in Z3. To store a hash in a sequence, we use the keys of the hash for the integer index, and the values of the hash for the content of the sequence. There is a drawback to using this index: the key of the hash has to be of type integer, otherwise it is not possible to access the values in the sequence. A solution is to convert the key in a codec to an integer and use this integer in the sequence in Z3. After getting a solution from Z3, the integer needs to be converted back to the original type with the codec.

There are multiple ways to store an item in a sequence in Z3. The first way is to use `(= (seq.nth a b) c)`, the other way is to use `(= (seq.at a b) (seq.unit c))`. In both cases, `a` is a sequence of type `x`, `b` is the position in the sequence, and `c` of type `x` is the variable needed to store in the sequence. In the first case, `seq.nth` returns the variable stored at that position. In the second case, `seq.at` returns a sub-sequence of length 1. Therefore `c` must be converted to a sequence of length 1, which is done by `seq.unit`.

For the example of a hash encoded by a sequence, we use `seq.nth` since it is shorter than `seq.at`. Let `h` be the following hash: `h = {'a' => {True => 3.0, False => 11.0} }`. Using sequences in Z3, first the keys need to be stored in a codec and to be transposed. For this hash, the codec would be:

```
{:string => {'a' => 0},
:boolean => {True => 0, False => 1}}
```

Then, the encoding for `h` is:

```
(declare-const h (Seq (Seq (Seq Real))))
(declare-const i (Seq (Seq Real)))
(declare-const j (Seq Real))

(assert (= (seq.nth h 0) i))
(assert (= (seq.nth i 0) j))
(assert (= (seq.nth j 0) 3.0))
(assert (= (seq.nth j 1) 11.0))
```

All the three ideal hash constraints are possible using the sequence encoding. The only, but major, drawback is that for non-numerical keys in a hash a codec has to be used. This means that Z3 cannot put any constraints over these keys anymore. Only equality constraints over these types would still be possible. Therefore using sequences as encoding for hashes is not a good idea.

4.2 Declare-function

The second method for encoding hashes in Z3 is using declare-function. Declare-function is used in Z3 to map one or more input values to one or more output

values. To declare a function in Z3, first the type of the function is declared with `(declare-fun a (b) c)`, where `a` is the name of the function, `b` is the set of input parameter types, and `c` is the output type. Then, after declaration, one can assert constraints on this function, such as `(= (a b) d)`. This means that function `a` after input of `b`, is equal to `d`. In terms of hashes, this would be $a[b] = d$.

When trying to declare a nested hash, the output type of a first level hash should be the complete type of the second level hash. Only, SMT-lib does not allow for higher-order functions. A solution for this problem is when declaring nested hashes to use internally new hashes, and remember that one output of one specific hash should be the input of another specific hash. This approach creates a problem when trying to define the values of a nested hash where the deeper nested hashes have more than one element. In that case, the first-order hash maps a value to a (second-order) hash with multiple elements. But since we define the output type of the first-order hash as the input type of the second-order hash, we must map the input value of the first-order hash to the input value of the second-order hash. But because the second-order hash has multiple elements, this is impossible. On the bright side, it is possible to set constraints on the non-hash values, by directly using the output of a hash as input of the next-order hash.

Let `h` be the following hash: `h = {'a' => {True => 3.0} }`. Using `declare-fun` in Z3, this would be encoded as:

```
(declare-fun h (String) Bool)
(declare-fun h2 (Bool) Real)

(assert (= (h "a") true))
(assert (= (h2 (h "a")) 3.0))
```

The only ideal constraint we can fullfill with the declare-function encoding, is the **value-constraint**. The other two ideal constraints are not possible, because in Z3, there is no functionality for getting the length of a declared function, nor is it possible to compare two declared functions with each other. On top of that, it is difficult to use nested hashes and hashes with more than one element. Therefore using declade-function as encoding for hashes is not a good idea.

4.3 Arrays

The final method to encode hashes is using arrays. An array in Z3 is different from the normally used array by programmers. In Z3, arrays are infinite, and the index of the content can be any datatype, not necessarily integers starting from 0. Since the arrays are infinite, the only way to know how many elements are in an array would be to keep track of the number of elements that are put in the array, there is no build-in `length(array)` function in Z3.

There are two ways to store an item in an array in Z3: using `select` and using `store`. The function `select` uses two parameters to select the value stored in array `a` at index `x`. The function `store` uses three parameters to store the

value `v` in array `a` at index `x`. Both the values and the indexes can be of any type, which makes array encoding a good candidate for hash encodings.

Let `h` be the following hash: `h = {'a' => {True => 3.0} }`. Using arrays in Z3, the encoding for `h` is:

```
(declare-const h (Array String (Array Bool Real)))
(assert (= 3.0 (select (select h "a") true)))
```

The only ideal constraint that is not possible to assert in Z3 with array encodings is the **length-constraint**. There is no build-in function for getting the length of an array in Z3. The other two ideal constraints are possible and there are no problems with nested hashes or multiple elements per hash. This makes that arrays are good to be used as encoding for hashes.

Arrays in Z3 are known to be slow and not the right datatype to use if only few elements are stored in them. This might be a problem, but the other two possible hash encodings are worse, so that is why we use the arrays in the implementation.

5 Implementation

For the implementation, we used the already existing code for the SMT-solver: the SMTreesolver made by Florine de Geus. This SMTreesolver, written in the programming language Ruby, functions as a translator for the constraint expressions to Z3 and from the Z3 solution back to Ruby values. In the SMTreesolver, there were only three classes for creating elements used in the constraints: Constant, Variable and Constant_Array. The hash type we want to implement can contain multiple constants or variables of any type, so we needed to create a new class: 'HashMap'. When creating a HashMap element, the class needs to know the name of the hash, the keys of the hash (in array form) and the type of the hash. These three values are given as parameters when creating a HashMap element.

At creation of a HashMap, the hash is already bound by a certain constraint. This constraint specifies the specific key(s) it needs to put in the (nested) hash to come to the position of the end-value. At this point, the end-value might not be known. Therefore we choose to store the hash in a different way. The known keys are store in an array, in the order in which to put them in the hash.

In other languages, the name of the hash might not be relevant, but with the Array encoding, Z3 needs to know the name of the hash. That is why this is a requirement why the name should be a string.

At the creation of a HashMap, the three parameters are checked if they are correct: are the keys from the same type as given in the hash type, are the types themselves valid, and is the hash name a string. If no error is raised, another function in the HashMap can be called to create a pointer to Z3 where the actual hash is stored. When this pointer is called, the HashMap uses the array encoding to translate the HashMap parameter values to a Z3 array. The elements that are stored in the hash are then created with the corresponding Constant or Variable classes.

When the SMTreesolver is translating the complete expressions to Z3 and finds a solution, multiple functions are called at the Constant, Variable and HashMap, to get or compute certain values over these class elements. During this process, since the HashMap has a complete type structure, we had to adapt these functions to be able to process the hash type.

Currently, the most basic expressions with hashes in them can be calculated in the SMTreesolver: not-nested hashes with multiple elements and constraints on the end-values on them, nested hashes with single elements and constraints on the end-values on them. The other types of constraints still are needed to be implemented correctly in the SMTreesolver.

6 Conclusion

We can conclude that from the three possible hash encodings in Z3, the array encoding is the most fitting. The array encoding can handle the hash type structure and is able to contain multiple elements. The other two hash encodings cannot cope with the multiple elements, nested hashes or keys with other datatypes than integers, so they are not fitting. After implementing the array encoding, the most basic expressions containing hash constraints are solvable in the SMTreesolver.

7 Future work

In the future, the work on implementing the hash encoding in the SMTreesolver can be continued to enable the more difficult types of constraints on the hashes. Furthermore, other types of complex data structures can be implemented in the SMTreesolver such as Arrays with variables in them, Structs and Lists.

References

1. F. de Geus, "On the use of smt solvers in model-based testing," 2020.
2. L. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems* (C. R. Ramakrishnan and J. Rehof, eds.), (Berlin, Heidelberg), pp. 337–340, Springer Berlin Heidelberg, 2008.