{ P } S { Q }

Verifying Software with SMT and Random Testing using a Single Property Specification

Pieter Brandwijk

August 13, 2012

Master Course Software Engineering

Thesis Supervisor : Prof. dr. Jan van Eijck

Organization : Centrum Wiskunde & Informatica

Availability : Public

University of Amsterdam

Summary

This thesis investigates the combination of different methods of property-based testing on a single property specification. The idea is that by combining the strengths of different methods stronger software verification can be achieved, while a property of a program has to be specified only once. We develop a tool that combines the use of Satisfiability Modulo Theories (SMT) solvers for automatic theorem proving with random testing. SMT solving is used to prove or disprove that the property holds on small domains, random testing is used to verify the property on larger domains. When a property is disproved a counterexample is produced.

The tool uses a small imperative language to define programs and properties. To process a program into a specification that can be used by SMT solvers the program is preprocessed in three phases. First, all conditional loops are unwound up to a given bound. After this, the program is put into single assignment form, where we introduce an algorithm to assure no variable is assigned a value more than once during execution. Thirdly, the program is put into conditional normal form, where the program becomes a flat list of conditional statements, each defining the complete condition under which an assignment occurs. After this preprocessing an SMT problem can be generated that can be checked by an off-the-shelf SMT solver.

We apply the tool on several programs and find that indeed it provides stronger software verification than a single verification method could. The domain of input that can be proved efficiently by SMT solving is small but significant. When introducing a deliberate fault in a program the SMT solver shows to be very accurate in finding it, while random testing alone is highly unlikely to discover it. Furthermore, as the tool requires an explicit specification of the pre- and postconditions of a program, violations can also occur due to incorrect specifications. In these cases the tool has shown to be helpful in understanding the behavior of a program.

Preface

With this thesis I conclude two years of studies at the University of Amsterdam. I'd like to take this opportunity to thank some of the people who during this time helped me to grow as a software engineer and as a person.

First I'd like to express my thanks to my supervisor professor Jan van Eijck, who generously dedicated his time to help me learn and whose life philosophy is a great inspiration. I'd like to thank the CWI for offering me a place to do my research and also my CWI colleagues Floor Sietsma, whose pleasant company made days at the office pass away too quickly, and Bert Lisser, whose enthusiasm for maths and programming always made for good conversation.

I'd like to thank the University of Amsterdam and the teachers of the master Software Engineering for helping me to become a better software engineer. I'd also like to thank my employer Morpheus, who encouraged me and gave me all the space I needed for my studies. In particular I thank Peter-Paul Kruijsen, who advised me not to postpone my application with another year.

I thank my parents, Jan and Nel Brandwijk, for their support – not just during the last two years, but the entire twenty seven years. Also I'd like to thank the family of Dick, Anneke, Rick and Coen Verwoert, who generously offered me a place to stay on college days and who took me in as part of the family. I am very grateful for that, and will miss those Monday evenings. I thank Bart van Eijkelenburg, Silvia Pedrotti, Lennart Tange, Diana Rodriguez and Ivo Jonker for their friendship and support during the past two years.

Contents

Chapter 1

Introduction

When given a program, a set of preconditions on the program's input and a set of postconditions on its output, we have specification of the expected behavior of that program. We will call these specifications *properties*, though various names are in common use. In design-by-contract programming they are called *contracts*, which creates a metaphor for the idea that formal specifications can be used to enforce the expected behavior of software.

An elegant way to express properties is by using a Hoare triple. It has the following abstract form:

$$\{P\} \quad S \quad \{Q\}$$

Here $\{P\}$ represents a set of preconditions, S a program and $\{Q\}$ a set of postconditions. [19]

Hoare triples can be used for *property-based testing*, where we use some method to verify that the postconditions hold when the program is executed on input that conforms to the preconditions. Such a method can be dynamic, executing the program on actual input to test if the property holds, or static, where the code is analyzed to derive the validity of the property. Examples of the former are test case selection methods like random testing and heuristic t-wise selection. [16] Examples of the latter are automated theorem proving, where the computer automatically decides the validity of a logical formula, and interactive theorem proving, where the computer assists the human user in constructing a proof. [18]

Different verification methods have different strengths and weaknesses. Test case selection methods are fast, but can only disprove a property by finding a counter example – they cannot prove it. Automatic theorem provers can give proofs, but

not every proof problem is automatically decidable. Interactive provers can give very complete proofs, but require much time and expert knowledge to operate.

Because of these different strengths and weaknesses it seems obvious to try to combine different methods of verification to achieve a better result. Since we just stated that the specification contained in a Hoare triple is sufficient for property-based testing, it should be possible to define a general framework where different methods can (inter)operate on the same property. This would be very beneficial to a programmer, as he would get the benefit of multiple verification methods for the price of just one specification.

Research question

The research question of this thesis is formulated as follows:

How can multiple methods of property-based testing be implemented on a language so, that they can operate on the same property specification of a program?

We will scope this question to two concrete methods of property-based testing, namely random testing and the usage of Satisfiability Modulo Theories (SMT) solvers for automated proof. We will use a small, but Turing-complete, imperative language to define programs and properties. The language will support integers and arrays of integers. Conditional loops are represented by while statements only. Procedure calls (and thus recursion) will not be supported.

As domain of programs we assume algorithms that can be instantiated using a single list of integers, like list operations, sorting algorithms and graph algorithms. For the SMT solving we will focus only on decidable formulas, so we leave out quantified formulas. As background logics we use linear arithmetic and the theory of arrays.

Problem analysis

The challenge of this project is to combine a static and a dynamic method of software verification using a single property specification. The goal is to first prove a property on a small domain of input using SMT. Proof attempts will be done on an increasingly larger domain, until the solving process exceeds a specified time limit. After this random testing is invoked to verify the remaining domain of input.

The static analysis of SMT requires that the program be translated into a formula of first order logic. Several processing steps are needed before such a formula can be created. Most notably, programs have to be bounded. This means that conditional loops must be turned into sequential control structures to assure symbolic termination. [13] After bounding the conditional loops, the further processing of the program in to a logical formula should not change the semantics of the program.

The next challenge is to integrate the two verification methods. A control structure has to be made to govern the proof attempts with SMT solving and, after those attempts have been stopped, to invoke random testing.

Structure of the text

After this introduction, chapter 2 explains the main concepts of random testing, SMT solving and using SMT solvers to prove properties of programs. Chapter 3 discusses related work and chapter 4 explains the research method. Chapters 5 through 8 form the core of the research and are written in literate programming style, meaning that the programming code is presented along with the text. Chapter 5 introduces the grammar of our custom language. Chapter 6 explains the processing steps needed to generate SMT specifications from a program. Chapter 7 explains the verification process of the combined verification methods. Chapter 8 shows the application of the verification tool on some example programs. In chapter 9 we conclude by analyzing the results of our research and give pointers for future research. The appendices contain the remaining code of the project that was not treated in the core chapters.

Chapter 2

Background

Random Testing

Random testing is a verification method where samples of valid input are randomly generated and used to execute the program. The output of the execution is then checked against a condition. This process can be seen as a generalization of unit testing – instead of manually defining test cases, the computer automatically generates the test cases. The goal of the method is to falsify the property under test by finding a counterexample. Because of the automatic test case generation the method is very fast, and in practice the method has shown to be very effective in finding bugs. The down-side of the method is that *not* finding a counterexample is no guarantee for the correctness of the property.

A popular tool for random testing is QuickCheck [7], which was originally written for Haskell but is nowadays available for many programming languages. It is the tool we will use in this project. QuickCheck uses specifications written in the host programming language itself to define the conditions on the output of the program. The automatic generation of input is defined by an input generator, which can be customized by the programmer. When we view the tool in the context of Hoare triples we see that the program's preconditions are contained in the input generator and the postconditions are defined as programs in their own right. As an example, consider a Haskell program to reverse a list of integers;

```
reverseList :: [Int] -> [Int]
reverseList [] = []
reverseList xs = last xs : (reverseList $ init xs)
```

Executing the reverse function on a list, and then again on the resulting list should give back the original list. We can define this as the postcondition. Note that QuickCheck calls the specification of the postcondition 'property' (using the prefix `prop_`), but we use this term to refer to the specification contained in a Hoare triple.

```
prop_reverseList :: [Int] -> Bool
prop_reverseList xs = (reverseList . reverseList) xs == xs
```

As the lists in the reverse algorithm can contain any integer, we can use QuickCheck's default input generator for lists. This represents the precondition that all elements in the list are of type `Int`. QuickCheck can now verify the property by generating, by default, one hundred lists of integers, feeding them to the property function and checking that every execution returns `true`. If an execution returns `false` then the current generated list is presented to the user as a counterexample that falsifies the property.

Satisfiability Modulo Theories

SMT solvers are a type of *automated theorem provers*. [5, 8, 10] They attempt to decide the satisfiability of first order logic formulas given a certain *background theory*. The background theory is used to reduce the formula to a formula of boolean propositional logic, which can then be solved by a regular SAT solver [6].

For example, the formula

$$\varphi \equiv (x - y \leq 0) \wedge (y - z \leq 0) \wedge ((z - x \leq -1) \vee (z - x \leq -2))$$

can be reduced to a set of propositions using linear integer arithmetic as background theory. We use four propositional variables to represent the theory atoms:

$$\begin{gathered} a \text{ represents } (x - y \leq 0) \\ b \text{ represents } (y - z \leq 0) \\ c \text{ represents } (z - x \leq -1) \\ d \text{ represents } (z - x \leq -2) \end{gathered}$$

Now the SMT solver exhaustively encodes incompatible relations between the theory atoms. Given linear integer arithmetic as background theory the following negations hold in our example:

$$\begin{gathered}\neg(a \wedge b \wedge c) \\ \neg(a \wedge b \wedge d) \\ \neg(\neg a \wedge \neg b \wedge \neg c) \\ \neg(\neg a \wedge \neg b \wedge \neg d)\end{gathered}$$

Also, because of the structure of φ, it holds that:

$$a \wedge b \wedge (c \vee d)$$

Putting all these conditions together gives the following formula ψ:

$$a \wedge b \wedge (c \vee d) \wedge \neg(a \wedge b \wedge c) \wedge \neg(a \wedge b \wedge d) \wedge \neg(\neg a \wedge \neg b \wedge \neg c) \wedge \neg(\neg a \wedge \neg b \wedge \neg d)$$

With this translation we can say that φ is only satisfiable when ψ is satisfiable. Since ψ is a formula in boolean propositional logic, we can use an off-the-shelf SAT solver to determine its satisfiability – and thus the satisfiability of φ. The SAT solver will conclude that there is no assignment to a, b, c and d such that ψ becomes true, hence the formula is unsatisfiable. The SMT solver then concludes that there is also no assignment of integers to x, y and z such that φ becomes true and reports its unsatisfiability to the user.

Several standard background theories have been defined which are supported by most modern SMT solvers. Besides linear arithmetic these include uninterpreted functions, the theory of arrays, bit vectors and others. In some theories it is also possible to use quantification, although this usually means that the theory becomes undecidable in general. When an SMT solver cannot decide whether a formula is satisfiable or unsatisfiable, it reports *unknown*.

By using the background theories SMT solvers can also abstract over the binary representation of a data type. In SAT solving data types like integers are always represented by a vector of boolean variables. SAT solvers can therefore only check finite domains. SMT solvers do not have this limitation as they interpret integers using standard semantics of arithmetic.

Verifying program properties with SMT

SMT solvers can be used to check the validity of properties of programs. This can be done by formally specifying the statements of an algorithm as a logical formula. Assignments, for example, can be seen as an equivalence relation. A statement like `x := y+1;` can be formulated logically as $(x = y+1)$. A conditional statement like `if (x < 0) then y := x+3;` can be seen logically as the condition implying the assignment: $(x < 0) \Rightarrow (y = x+3)$. In this way it becomes possible to model the behavior of a program logically. Together with the pre- and postconditions on the program's input and output we can use this logical model to reason about the correctness of the program.

As an example we define a program `abs` that takes a variable `x0` as argument and assigns the absolute value of `x0` to the variable `x1`. The precondition of `abs` is that `x0` is an (arbitrary) integer. The postconditions is that `x1` is greater than 0. These three elements together are the complete specification of a property in the form of a Hoare triple.

```
Pre: x0 is an integer

abs(x0) {
  if (x0 < 0)
    then { x1 := x0 * -1 ; }
    else { x1 := x0 ; }
}

Post: x1 > 0
```

To verify that the property holds with an SMT solver we must translate the Hoare triple to a formula of first order logic. Since the program only operates on integers we can use linear integer arithmetic as background logic. Translating the precondition is trivial, as it allows any integer it is always true:

$$\upsilon \equiv true$$

The core of the `abs` program consists of a conditional statement with two clauses. The `then` clause is executed when the condition is true, the `else` clause when it is false. Logically, we can translate that into the conjunction of two implication relations:

$$\chi \equiv (x_0 < 0) \Rightarrow (x_1 = x_0 * -1) \wedge \neg(x_0 < 0) \Rightarrow (x_1 = x_0)$$

Translating the postcondition gives:

$$\psi \equiv (x > 0)$$

But we want to know if the postcondition that we have specified holds on all valid input (in this case any integer for `x0`). We can also formulate this differently and ask if there is any assignment of values to x_0 and x_1 such that the postconditions does *not* hold. Such an assignment is a *counterexample* of the property. To specify this we must *negate* the postcondition and add this to the formula. So the specification of the counterexamples of the property is the conjunction of the preconditions, the program and the negation of the postcondition $(\upsilon \wedge \chi \wedge \neg\psi)$. For our example this becomes:

$$\varphi \equiv true \wedge (x_0 < 0) \Rightarrow (x_1 = x_0 * -1) \wedge \neg(x_0 < 0) \Rightarrow (x_1 = x_0) \wedge \neg(x_1 > 0)$$

Now the question remains if there is a model for φ. If no such model exists we have proved that the property is valid. If we *can* find a model then we disproved the property by giving that model as a counterexample. Of course we use SMT solvers to answer precisely this question.

Giving φ to an SMT solver like Z3 [9] or Yices [11] will result in the solver reporting that the formula is in fact satisfiable. The model it provides to support this claim is $\{x_0 \mapsto 0, x_1 \mapsto 0\}$. This counterexample shows that if we input 0 into the `abs` program the postcondition will not hold, therefore we have disproved the property. Of course, if we change the postcondition to $(x \geq 0)$, the SMT solver will find no counterexample and thus proves the new property to be valid.

Chapter 3

Related work

Some work has already been done by others to combine testing and proving strategies. Dybjer et al. [12] describe a combination of testing and interactive proving techniques to improve confidence in the correctness of Haskell programs. For random testing QuickCheck is used, mostly in the context of debugging before a proof attempt. Agda/Alfa is used as proof assistant. Groce and Joshi [17] show a combination of random testing and model checking to explore the state space of nondeterministic programs.

A concrete example of using SMT solvers for software verification is given by Liu et al. [20]. They show a translation of Java programs to quantified bit-vector formulas which are then checked by an SMT solver. Their use of quantified formulas makes it possible to express SMT problems in a short way, but it opens the possibility for the solver to return false positives. Their comparison of different solvers therefore show different results.

Armando et al. [1] compare the results of using SMT solvers to using SAT solvers. They present a detailed method to preprocess unbounded programs into bounded programs that can be translated to an SMT specification. They translate all loops to sequential `if` statements, they assure that any variable is not assigned more than once during execution (single assignment form) and lastly they convert the program to conditional normal form where all assignments are guarded by the conditions under which they should occur. This method has also been used in this thesis, though we present a more efficient version of the single assignment form. Armando et al. use a programming language that contains assertions. They use the SMT solver to find models that violate an assertion, this is different from our approach were we want to know if a separately defined property holds for the program.

Chapter 4

Research method

We want to know if we can apply both random testing and SMT solving for the verification of software using the same property specifications for both methods. The obvious way to show this is by creating a tool that is capable of doing this and test it out on some programs.

The first step is to pick the right tools. As a tool for random testing we take QuickCheck [7], as this tool is in wide spread use and is being actively developed. The original QuickCheck tool is written in Haskell. This, along with Haskell's abstract data type system that is convenient for writing grammars makes Haskell a straightforward choice as the programming language for our tool.

There are several SMT solvers available that can be used freely for personal use or for research. Many of them introduce their own syntax to specify SMT problems in. However, recently the open standard SMT-LIB [10] has been published that is intended as a common language for SMT solvers. Most actively developed SMT solvers support this syntax. We will use SMT-LIB in our project so that switching between solvers becomes easy. As default SMT solver we use Z3 [9]. This solver is developed by Microsoft and represents the state-of-the-art in SMT solving.

The approach of this project is to embed a small imperative language and a language to specify properties into Haskell. To be able to use QuickCheck on this language we create an interpreter and wrapping functions to be able to test properties on programs. To turn a program and a property into an SMT problem in SMT-LIB syntax we create a preprocessing pipeline. We then create a verification function that starts by trying to prove the property on an increasingly larger input domain. The proof attempts are stopped when SMT problems start to exceed a given time

limit. After this timeout the remaining domain is tested with QuickCheck.

Our hypothesis is that this approach of combining verification methods will lead to a higher chance of bug discovery than by using a single verification method. To test this hypothesis we will implement a couple of algorithms and properties on those algorithms in our tool. In some algorithms we will deliberately introduce a violation of a property that occurs only in very specific circumstances. We will then compare the results of pure random testing to the results of combining SMT verification with random testing.

Chapter 5

Grammar

We define a simple imperative language with assignments, control structures and loops. Variables in the language are integers or one-dimensional arrays of integers. With this very small set of programming constructs we do have a Turing-complete language, so any program can be expressed in it.

The language is defined using Haskell's abstract data types. This means that we describe an abstract grammar. The structure of the language is inspired by Gordon [15] and Nielson and Nielson [21]. A concrete syntax for the language is defined in the parser module added in appendix B. The interpreter for the language is added in appendix A.

We start with the module declaration and we also define some type synonyms to enhance the readability of the grammar.

```
module Language.Colt.Grammar where

type VarName = String
type Value = Int
type Length = Int
type Index = Expr
type Cond = BExpr
```

Conditions are boolean expressions that should be true in the final state of a program. We define type synonyms for both pre- and postconditions to distinguish them later on.

```
type CondName = String
data Condition = Condition { condName :: CondName,
                             condBExpr :: BExpr }
type Precondition = Condition
type Postcondition = Condition
```

The data type for variable references comprises both regular variables and indices of arrays. An index is represented as an expression.

```
data Var = RegVar VarName
         | ArrVar VarName Index
```

Expressions can be integer constants or references to the value of a variable or an array index. They also allow for the arithmetic operations of addition, subtraction and multiplication.

```
data Expr = Const Value
          | V Var
          | Add Expr Expr
          | Subtr Expr Expr
          | Mult Expr Expr
```

Boolean expressions allow for the true and false constants, negation, disjunction and conjunction. Boolean expressions also allow for expressing the equality, less-than and greater-than relations on (integer) expressions.

```
data BExpr = BConst Bool
           | Neg BExpr
           | Or [BExpr]
           | And [BExpr]
           | ExprEq Expr Expr
           | ExprLt Expr Expr
           | ExprGt Expr Expr
           | ExprLtEq Expr Expr
           | ExprGtEq Expr Expr
```

The declaration of a regular variable consists of a name and an initial value. Arrays are declared with their name, length and a list of initial values.

```
data Declaration = RegDecl VarName Value
                 | ArrDecl VarName Length [Value]
```

The supported statements in the language are assignments, `if-then` and `if-then-else` statements, a conditional loop, an assert statement and a statement to compose other statements.

There are three kinds of assignments. The first is a regular assignment of an expression to a variable or array index. This assignment is meant to be used in the specification of programs. The other assignments are used in the preprocessing phase for generating an SMT specification from a program. The array assignment assigns one array to another. The array store assignment takes an existing array, changes the value at a given index and assigns the resulting array.

Assertions are used in the loop unwinding phase to assure during execution that the original loop condition has become false.

```
data Statement = Assign Var Expr
               | ArrAssign VarName VarName
               | Store VarName VarName Index Expr
               | Sequence [Statement]
               | If Cond Statement
               | Ite Cond Statement Statement
               | While Cond Statement
               | Assert Cond
```

A program is a combination of a list of declarations and a statement.

```
data Program = Program [Declaration] Statement
```

A Hoare triple is a list of preconditions, a program and a list of postconditions. Hoare triples represent the entities that can be verified using random testing and SMT solving. As such they are the central data type in this thesis.

```
data HTriple = HTriple {
  getPres :: [Precondition],
  getProg :: Program,
  getPosts :: [Postcondition]
}
```

Chapter 6

The Preprocessing Phase

Creating a formula of first order logic from a program boils down to specifying what assignments occur under which conditions. SMT problem instances that are generated from a program consist of only variable declarations and implication relations between conditions and equality definitions. We use the *de facto* standard SMT-LIB [10] for specifying instances of SMT problems.

To do this we pre-process a program in three phases. (1) Loop unwinding, (2) conversion to *single assignment form* and (3) conversion to *conditional normal form*. After the last phase we can generate an SMT-LIB specification. The preprocessing phases are inspired by Armando et al. [1] although we use SMT-LIB and in the second phase we introduce a more efficient algorithm to assure single assignment.

Loop unwinding

Creating an SMT problem for an algorithm requires the unwinding of loops. Essentially, SMT solvers cannot deal with loops that depend on a condition to terminate. The problem occurs because the condition is represented as a *symbolic* boolean, meaning there are two possible outcomes (`true` or `false`) and the solver will consider both paths. The `true` path always leads to another loop so this would create an infinite formula that can never be solved. The algorithm is then said to be not *symbolically terminating*. [14]

To solve this problem loops must be unwound to a set of nested `if` statements. The depth of this nesting is given by a bound k.

```
module Language.Colt.UnloopTransformer where

import Data.Map as Map hiding (map)
import Language.Colt.Grammar

type Bound = Int
```

The `unwind` function requires an integer argument for bound k. This k will determine the depth of the unwinding. Unwinding a program leaves all statements intact except for `while` loops. Loops are transformed to a series of k nested `if-then` statements. More formally, the transformation looks like this:

$$\texttt{while}(c)\ \{\mathsf{P}'\} \longrightarrow \texttt{if}\ (c)\ \{\mathsf{P}'\ \texttt{while}(c)\ \mathsf{P}'\}\}$$

The final `while` is replaced by an unwinding assertion. This assertion checks that the condition of the `while` loop has become false. If this is not the case, then a higher k-bound must be chosen for correct execution of the program. Note that the success or failure of the unwinding assertion depends on the program input.

```
unwind :: Bound -> HTriple -> HTriple
unwind k (HTriple pres prog posts) = HTriple pres prog' posts
  where prog' = unwindProg k prog

unwindProg :: Bound -> Program -> Program
unwindProg k (Program decls stmt) = Program decls stmt'
  where stmt' = unwindStmt k stmt

unwindStmt :: Bound -> Statement -> Statement
unwindStmt _ (Assign x expr) = Assign x expr
unwindStmt _ (ArrAssign arr1 arr2) = ArrAssign arr1 arr2
unwindStmt _ (Assert bExpr)  = Assert bExpr
unwindStmt k (Sequence xs)   = Sequence $ map (unwindStmt k) xs
unwindStmt k (If p s)        = If p (unwindStmt k s)
unwindStmt k (Ite p s1 s2)   = Ite p (unwindStmt k s1) (unwindStmt k s2)
unwindStmt k (While p s)     = unwindWhile k k (While p s)
  where
    unwindWhile k it (While p s)
      | it > 0 = If p (Sequence [unwindStmt k s,
```

```
                                unwindWhile k (it-1) (While p s)])
      | otherwise = Assert (Neg p)
```

We give an example of what this transformation looks like. Consider the following two programs.

$n \leftarrow 3$
$m \leftarrow 1$
while $n > 1$ **do**
    $m \leftarrow m * n$
    $n \leftarrow n - 1$
end while

$n \leftarrow 3$
$m \leftarrow 1$
if $n > 1$ **then**
    $m \leftarrow m * n$
    $n \leftarrow n - 1$
    if $n > 1$ **then**
        $m \leftarrow m * n$
        $n \leftarrow n - 1$
        if $n > 1$ **then**
            $m \leftarrow m * n$
            $n \leftarrow n - 1$
            assert $!(n > 1)$
        end if
    end if
end if

The program on the left is the original program, which computes the factorial of 3. The program on the right is the resulting program after applying the unloop transformation with a bound of three. Notice that in the unwound version, if n would be higher than 3, then the unwinding assertion in the last `if` would fail during execution. For a bound of four the maximum n is 4, and so on.

Single Assignment Form

In imperative languages the same variable can be assigned a value multiple times. Every time a new assignment occurs, the old value is simply overwritten. However, translating an imperative program to a logical formula requires that no variable is assigned (equality relation) more than once. The reason for this is that a formula like $x = a \wedge x = b$ is unsatisfiable, unless $a = b$. To capture the semantics of the imperative paradigm in the formula we must use a new variable name after each assignment, like so: $x_0 = a \wedge x_1 = b$. Now the satisfiability of this formula is

independent from a and b being equal. Expressions that use the variables must be updated to refer to the correct version of the variable.

Armando et al. [1] use an algorithm called Static Single Assignment that has its roots in compiler optimization. After each `if-then` or `if-then-else` statement conditional assignments are placed that assign the right variable based on the condition of the `if`. The downside of this technique is that many more extra variables are created than are needed for the purpose of creating SMT problems. We therefore propose another form of single assignment that we will call Guarded Single Assignment (GSA). GSA uses the condition of the `if` to assure that an assignment takes place only once during execution, this creates less overhead than SSA.

The transformation to GSA consists of the following steps:

1. Replace all assignments of the form `a[e1] := e2` with `a := store(a,e1,e2)`, where `store` is a function such that `store(a,e1,e2)` returns the array obtained from a by setting the element at position e_1 with the value of e_2;
2. Replace the occurrences of the variables that are target of assignments (say x) with new, indexed versions of the variables (say $x_0, x_1, \ldots$);
3. Replace the occurrences of variables in expressions with the appropriate versions to preserve the semantics of the original program;
4. Add guarding assignments to control structures. `if-then` statements are replaced by `if-then-else` statements. The `else` clause contains an assignment for every variable that was changed in the `then` clause. The assignment assigns to the last indexed version of a variable in the `then` clause the last indexed version of before entering the `if-then` statement. This guards that regardless of whether the condition of the `if` is true or false that the rest of the program can use the name with the highest index. A similar process has to happen for `if-then-else` statement, but now also taking into account that either clause may have assigned a variable zero, one or even multiple times.

The last step is quite complex and we will illustrate it with some examples. First we consider an `if-then` statement that contains two assignments on the same variable. On the right hand side the same program is displayed in GSA. In the `else` clause the latest version of the variable (i_2) is assigned the value from before the `if-then` statement. Note that in case the condition is false and the `else` clause is executed, the variable i_1 will never be instantiated. Variable i_2 is the version that will be used by the rest of the program.

$i \leftarrow 0$
$j \leftarrow x$
if $i < j$ **then**
    $i \leftarrow i + 1$
    $i \leftarrow i + j$
end if

$i_0 \leftarrow 0$
$j_0 \leftarrow x_0$
if $i_0 < j_0$ **then**
    $i_1 \leftarrow i_0 + 1$
    $i_2 \leftarrow i_1 + j_0$
else
    $i_2 \leftarrow 0$
end if

In the case of `if-then-else` statements for both the `then` and the `else` clause it must be checked whether or not a variable has also been assigned in the other clause. If a variable has been assigned in both clauses and has the same index number, nothing has to be done. The second possibility is that a variable was assigned in one clause, but not in the other. In this case a balancing assignment is added to the other clause which assigns the latest version of the variable to the value it had before entering the `if`. In the following example the variable i is assigned in the `then` clause, but not in the `else` clause.

$i \leftarrow 0$
$j \leftarrow x$
if $i < j$ **then**
    $i \leftarrow i + 1$
    $j \leftarrow j + i$
else
    $j \leftarrow j + 1$
end if

$i_0 \leftarrow 0$
$j_0 \leftarrow x_0$
if $i_0 < j_0$ **then**
    $i_1 \leftarrow i_0 + 1$
    $j_1 \leftarrow j_0 + i_1$
else
    $j_1 \leftarrow j_0 + 1$
    $i_1 \leftarrow 0$
end if

The third possibility is that a variable is assigned in both clauses, but more often in one clause than in the other. In this case the index numbers are out of sync. To compensate, the clause with the lower index number must add an extra assignment that assigns its own latest version of the variable to the latest version of the other clause. We show an example where the `then` clause assigns i twice and the `else` clause only once. The `else` clause therefore has to compensate by adding an extra assignment of its own i_1 to i_2. Now the rest of the program can safely use i_2, as both clauses guard that this variable exists and contains the correct value.

$i \leftarrow 0$
$j \leftarrow x$
if $i < j$ **then**
    $i \leftarrow i + 1$
    $j \leftarrow j + i$
    $i \leftarrow i - j$
else
    $j \leftarrow j + 1$
    $i \leftarrow i - 1$
end if

$i_0 \leftarrow 0$
$j_0 \leftarrow x_0$
if $i_0 < j_0$ **then**
    $i_1 \leftarrow i_0 + 1$
    $j_1 \leftarrow j_0 + i_1$
    $i_2 \leftarrow i_1 - j_1$
else
    $j_1 \leftarrow j_0 + 1$
    $i_1 \leftarrow i_0 - 1$
    $i_2 \leftarrow i_1$
end if

Some informal experiments on several programs indicate that GSA drastically decreases the amount of extra assignment statements that are needed to put a program in single assignment form. In some cases programs in GSA were almost half the size of their counterparts in SSA, while remaining semantically equivalent. Naturally, this is very beneficial to the performance of the SMT solving step, as shorter specifications require less resources to solve.

The complete code for the tranformation of programs to single assignment form can be found in appendix C.

Conditional Normal Form

The transformation to conditional normal form (CNF) flattens out nested `if` statements by guarding every assignment with a single `if-then` statement. The condition of this `if-then` is the conjunction of the conditions of all the above lying `if`s.

```
module Language.Colt.CNFTransformer where

import Language.Colt.Grammar

toCNF :: HTriple -> HTriple
toCNF (HTriple pres (Program decls stmt) posts) =
  HTriple pres (Program decls stmt') posts
    where stmt' = stmtToCNF (BConst True) stmt
```

The algorithm to transform a statement to CNF passes along the conjuncted conditions. At the beginning of the program this condition is `true` and gets expanded with every encounter of a nested `if`. An `else` clause is transformed into an `if-then` where the condition of the original `if-then-else` is negated.

```
stmtToCNF :: BExpr -> Statement -> Statement
stmtToCNF cs (Assign var value) = If cs (Assign var value)
stmtToCNF cs (ArrAssign ar1 ar2) = If cs (ArrAssign ar1 ar2)
stmtToCNF cs (Store ar2 ar1 iExpr expr) =
  If cs (Store ar2 ar1 iExpr expr)
stmtToCNF cs (Sequence xs) = Sequence $ map (stmtToCNF cs) xs
stmtToCNF cs (If c stmt) = stmtToCNF (And [cs, c]) stmt
stmtToCNF cs (Ite c thenStmt elseStmt) = Sequence [
  (stmtToCNF (And [cs, c]) thenStmt),
  (stmtToCNF (And [cs, (Neg c)]) elseStmt) ]
stmtToCNF _  (While _ _) =
  error "stmtToCNF: Cannot transform WHILE to CNF. Unwind first"
stmtToCNF cs (Assert c) = If (cs) (Assert c)
```

Again we will illustrate this transformation with an example. Consider the following two, semantically equivalent, programs.

$n \leftarrow 1$
$m \leftarrow 2$
if $n > m$ **then**
    $n \leftarrow n * 2$
    if $m > 10$ **then**
        $m \leftarrow m - 2$
    end if
else
    $n \leftarrow n - m$
end if

if $true$ **then** $n \leftarrow 1$
if $true$ **then** $m \leftarrow 2$
if $n > m$ **then** $n \leftarrow n * 2$
if $n > m$ && $m > 10$ **then** $m \leftarrow m - 2$
if $!(n > m)$ **then** $n \leftarrow n - m$

On the left is the original program, on the right the transformed version. The first two assignments are put under the condition `true`, so those assignments will aways occur. The `else` clause of the first `if` statement is transformed into an `if` statement in its own right, with its condition negated. The assignment of m under the nested `if` statement becomes a top level `if` where the condition is a conjunction of all the above conditions.

Generating SMT-LIB

Transforming a program in single assignment form and conditional normal form to an SMT-LIB specification is straightforward. All used variables and arrays must be declared and all assignments that occur under a condition (under an `if` statement) are translated to implication assertions. Beyond this, the transformation is entirely syntactical.

```
module Language.Colt.SMTLIBTransformer where

import Language.Colt.Grammar
import Language.Colt.PrettyPrinter
import Data.List
```

Expressions and boolean expressions are transformed into s-expressions that can be used directly in SMT assertions. References to indices of arrays are translated to a special `select` construct that is part of the theory of arrays.

```
varToSMTLIB :: Var -> String
varToSMTLIB (RegVar name) = name
varToSMTLIB (ArrVar ar i) =
  "(select " ++ ar ++ " " ++ exprToSMTLIB i ++ ")"

exprToSMTLIB :: Expr -> String
exprToSMTLIB (Const x)    | x < 0 = "(- " ++ show (-x) ++ ")"
                          | otherwise = show x
exprToSMTLIB (V var)      = varToSMTLIB var
exprToSMTLIB (Add x y)    = "(+ " ++ exprToSMTLIB x ++ " "
  ++ exprToSMTLIB y ++ ")"
exprToSMTLIB (Subtr x y)  = "(- " ++ exprToSMTLIB x ++ " "
  ++ exprToSMTLIB y ++ ")"
exprToSMTLIB (Mult x y)   = "(* " ++ exprToSMTLIB x ++ " "
  ++ exprToSMTLIB y ++ ")"

bExprToSMTLIB :: BExpr -> String
bExprToSMTLIB (BConst True) = "true"
bExprToSMTLIB (BConst False) = "false"
bExprToSMTLIB (Neg bexpr) = "(not " ++ bExprToSMTLIB bexpr ++ ")"
```

```
bExprToSMTLIB (Or bs) =
  "(or " ++ intercalate " " (map bExprToSMTLIB bs) ++ ")"
bExprToSMTLIB (And bs) =
  "(and " ++ intercalate " " (map bExprToSMTLIB bs) ++ ")"
bExprToSMTLIB (ExprEq e1 e2) = "(= " ++ exprToSMTLIB e1 ++ " "
  ++ exprToSMTLIB e2 ++ ")"
bExprToSMTLIB (ExprLt e1 e2) = "(< " ++ exprToSMTLIB e1 ++ " "
  ++ exprToSMTLIB e2 ++ ")"
bExprToSMTLIB (ExprGt e1 e2) = "(> " ++ exprToSMTLIB e1 ++ " "
  ++ exprToSMTLIB e2 ++ ")"
bExprToSMTLIB (ExprLtEq e1 e2) = "(<= " ++ exprToSMTLIB e1 ++ " "
  ++ exprToSMTLIB e2 ++ ")"
bExprToSMTLIB (ExprGtEq e1 e2) = "(>= " ++ exprToSMTLIB e1 ++ " "
  ++ exprToSMTLIB e2 ++ ")"
```

Because of the transformation to CNF, programs consist of only sequences and if-then statements that guard an assignment or assertion. Regular program assertions like the unwinding assertion are stripped from the SMT-LIB specification, as they would interfere with the properties later on.

```
stmtToSMTLIB :: Statement -> String
stmtToSMTLIB (If cs (Assign var expr)) =
  "(assert (=> " ++ bExprToSMTLIB cs ++ "(= "
  ++ varToSMTLIB var ++ " " ++ exprToSMTLIB expr ++ ")))\n"
stmtToSMTLIB (If cs (ArrAssign ar1 ar2)) =
  "(assert (=> " ++ bExprToSMTLIB cs
  ++ "(= " ++ ar1 ++ " " ++ ar2 ++ ")))\n"
stmtToSMTLIB (If cs (Store ar2 ar1 iExpr expr)) = "(assert (=> "
  ++ bExprToSMTLIB cs ++ "(= " ++ ar2 ++ " (store " ++ ar1 ++ " "
  ++ exprToSMTLIB iExpr ++ " " ++ exprToSMTLIB expr ++"))))\n"
stmtToSMTLIB (If cs (Assert c)) = "\n"
stmtToSMTLIB (Sequence xs) = concatMap stmtToSMTLIB xs
stmtToSMTLIB stmt = error "stmtToSMTLIB: no support for " ++ printStmt stmt
```

In SMT-LIB a declaration must explicitly state the type of a variable. The initial value is skipped in the transformation as this would constrain the proof to a single execution.

```
declsToSMTLIB :: [Declaration] -> String
```

```
declsToSMTLIB decls = concatMap writeDecl decls
  where
    writeDecl (RegDecl var _) = "(declare-const " ++ var ++ " Int)\n"
    writeDecl (ArrDecl var _ _) = "(declare-const " ++ var
      ++ " (Array Int Int))\n"
```

Properties in the form of pre- and postconditions are added to the SMT specification as assertions. Postconditions must be negated first, so we add a function for that too.

```
negateCond :: Condition -> Condition
negateCond (Condition name bExpr)= Condition name (Neg bExpr)

condToSMTLIB :: Condition -> String
condToSMTLIB cond = "; " ++ condName cond ++ "\n" ++
  "(assert " ++ bExprToSMTLIB (condBExpr cond) ++ ")\n\n"
```

Lastly the functions to get a complete SMT-LIB transformation of a program are defined. The preconditions are translated as they are. The postconditions are first negated so that when the solver returns `unsat` the postconditions is proven valid for that instance. The `generateSMTLIB` function makes sure that newlines are printed correctly.

```
toSMTLIB :: HTriple -> String
toSMTLIB (HTriple pres (Program decls stmt) posts) =
  declsToSMTLIB decls
  ++ "\n\n; Preconditions:\n" ++ concatMap condToSMTLIB pres
  ++ "\n\n; Program:\n" ++ stmtToSMTLIB stmt
  ++ "\n\n; Postconditions:\n" ++ concatMap (condToSMTLIB . negateCond) posts

generateSMTProp :: CondName -> HTriple -> String
generateSMTProp name (HTriple pres (Program decls stmt) posts) =
  declsToSMTLIB decls
  ++ "\n\n; Preconditions:\n" ++ concatMap condToSMTLIB pres
  ++ "\n\n; Program:\n" ++ stmtToSMTLIB stmt
  ++ "\n\n; Postcondition: " ++ name ++ "\n" ++ (condToSMTLIB . negateCond) post
    where
      post = head $ filter (\t -> condName t == name) posts
```

Chapter 7

Verification

The verification module contains the functionality needed to combine the SMT solving and random testing method. It first attempts to prove the properties of a program for a progressively higher loop unwinding bound. When the proof takes too long the proof attempt is canceled and QuickCheck is used to randomly test the property on the program.

We begin with the module declaration and the required imports.

```
module Language.Colt.Verification where

import Language.Colt.Grammar
import Language.Colt.Interpreter
import Language.Colt.UnloopTransformer
import Language.Colt.SAFTransformer
import Language.Colt.CNFTransformer
import Language.Colt.SMTLIBTransformer
import Language.Colt.SMTResultParser
import HSH
```

We use a timeout option for the SMT solver which is represented by an integer.

```
type Timeout = Int
```

QuickCheck properties must wrap in the interpretation of an algorithm and the checking of the property on the resulting space. Also, it should be possible to

use custom argument generators for the arguments that QuickCheck creates. We therefore introduce a type `QCProperty` to represent the execution of QuickCheck on a program.

```
type QCProperty = Int -> ([Int] -> HTriple) -> (Int -> Postcondition) -> IO ()
```

The first argument of type `Int` represents the starting bound. The function type `[Int] -> HTriple` represents functions that instantiate a Hoare triple based on a single array argument. Likewise the function `Int -> Postcondition` represent functions to instantiate postconditions based on the bound. Both the programs and the conditions in chapter 8 are written in this way. This creates a uniform method of instantiating Hoare triples that can be generalized to dynamically create QuickCheck properties.

The type of the function to instantiate a Hoare triple also determines the number and type of the arguments that QuickCheck will generate when testing a property. In this project we generate all programs using a single list of integers, hence we only need the type `[Int] -> HTriple`. It is possible to write program generators with other arguments, but in the current setup this would also require rewriting the functions for the verification process.

The `verify` function is designed to verify one postcondition at a time on a program. Verification is done on a progressively higher bound. When a property is proven on bound k, a message is printed on the console and the `verify` function is called recursively on bound $k+1$. If a counterexample is found for a certain bound then this example is printed to the console. If an SMT instance makes use of undecidable logics it is possible that the solver returns `unknown`. This is then reported on the console. The function also takes a timeout argument. When the solver needs more time than the specified timeout to solve an SMT instance the process is stopped. After a timeout QuickCheck is invoked to randomly test the program from bound k upwards.

```
verify :: ([Int] -> HTriple) -> (Int -> Postcondition) -> QCProperty
            -> Timeout -> Bound -> IO ()
verify progGen propGen property timeout k = do
  let name = condName (propGen k)
  let dummy = take k (repeat 0)
  let smtProb = (generateSMTProp name) $ toCNF $ toGSA $ unwind k $ progGen dummy
  result <- solve timeout smtProb
  case result of
```

```
    Unsat -> do { putStrLn ("Property proved for k = " ++ show k) ;
                  verify progGen propGen property timeout (k+1) }
    Sat -> do { putStrLn ("Property disproved for k = " ++ show k) ;
                counterexample timeout smtProb }
    Unknown -> do { putStrLn ("Property unknown for k = " ++ show k) ;
                    property k progGen propGen }
    Timeout -> do { putStrLn ("Timeout on k = " ++ show k) ;
                    property k progGen propGen }
  return ()
```

Solving an SMT problem is done by writing the SMT-LIB specification to a file and passing that file as input to the Z3 SMT solver. The result given by Z3 is then parsed and returned.

```
solve :: Timeout -> String -> IO SMTResult
solve timeout smtProb = do
  writeFile "spec.smt2" (smtProb ++ "\n(check-sat)")
  let command = "z3 -T:" ++ show timeout ++ " -file:spec.smt2"
  result <- (run command) :: IO String
  return $ runResultParser result
```

To produce a counterexample when one exists the specification is re-run with the additional option to produce a model. The reason why this option is not added in the `solve` function is that when an SMT specification is `unsat` requesting a model results in the solver exiting with an error.

```
counterexample :: Timeout -> String -> IO ()
counterexample timeout smtProb = do
  writeFile "spec.smt2" ("(set-option :produce-models true)\n"
    ++ smtProb ++ "\n(check-sat)\n(get-model)\n")
  let command = "z3 -T:" ++ show timeout ++ " -file:spec.smt2"
  counter <- (run command) :: IO String
  putStrLn "Counterexample:"
  writeFile "counter.model" counter
  putStrLn counter
```

Chapter 8

Results

Reverse array algorithm

The reverse algorithm is a straightforward program that takes an array of integers and outputs an array with the same elements in reversed order. The algorithm swaps the outer elements of the array and works its way inward until it has reached the middle of the array.

```
module Language.Colt.Examples.ReverseArray where

import Language.Colt.Grammar
import Language.Colt.Verification
import Language.Colt.Parser
import Language.Colt.TestUtils

import Data.List
```

The program is specified as a macro into which arguments can be injected. All numbers that are prefixed with a hash are replaced by actual values before execution. In this way random tests can use their specific input to test the algorithm.

```
decl(N := #0);
decl(i := 0);
decl(j := 0);
```

```
decl(t := 0);
decl(array[#0] := #1);
decl(original[#0] := #1)

original := array;
N := #0;
i := 0;
j := N - 1;
while (i < j) do {
  t := array[i];
  array[i] := array[j];
  array[j] := t;
  j := j - 1;
  i := i + 1
}
```

This program macro requires two external arguments, `#0` (the length of the array) and `#1` (the array itself) to be injected before the actual program can be parsed. Also, all variables are given a value in the declarations, and some of them are then re-assigned this value at the beginning of the algorithm. This is necessary for the SMT solving process. The translation to SMT does not take over the values of the declarations, so all variables are free. However, some variables need to be static in the SMT problem as well, like the length of the array. We assure this by adding an extra assignment statement.

A complete Hoare triple can be created based on a given array. First a list of preconditions is given, but as the elements in the array can be any integer there are no preconditions. Then the program macro is parsed into a program using the array and the length derived from it. Lastly the list of postconditions is added which in this case contains the condition that the resulting array is the reversed version of the original array. The postcondition is generated based on the length of the array.

```
reverseArray :: [Int] -> HTriple
reverseArray array = HTriple
  []
  (getProgram "scripts/reverseArray.colt" [show l, show array])
  [postCond_reversed l]
    where
      l = length array
```

The 'standard' test for the `reverseArray` algorithm in QuickCheck would be to apply the algorithm twice on an array and then checking that the resulting array is equal to the original array. However, because our while language does not support procedure calls we define an alternative postcondition. It states that the first element of the original array is equal to the element at index $N-1$ of the resulting array, the second element equal to the element at index $N-2$ and so forth until all elements have been added to the conjunction. For example, when $N = 3$:

$$(original[0] = array[2]) \wedge (original[1] = array[1]) \wedge (original[2] = array[0])$$

```
postCond_reversed :: Int -> Postcondition
postCond_reversed n = Condition "postCond_reversed" (And $ conj 0 (n-1))
  where
    conj i j | j < 0 = []
          | otherwise = (ExprEq (V (ArrVar "original" (Const i)))
                                (V (ArrVar "array" (Const j))))
                        : conj (i+1) (j-1)
```

In order to be able to use QuickCheck to verify that the postcondition holds on some random tests we define a function that uses the input generator to create lists longer than the given value; in this case we start at a length of 1.

```
run_qc_reverseArray :: IO ()
run_qc_reverseArray = qc_prop_longerThan 1 reverseArray postCond_reversed
```

Calling this function runs one hundred tests successfully, indicating that the postcondition seems correct on lists of arbitrary length. Now we will verify the program by first trying to prove the postcondition on arrays of fixed length. We define a function the starts the verification at bound 1 (representing arrays with one element) and a timeout of one second. After each successful test the bound is increased with 1, representing an array with one more element. When a property is disproved by the SMT solver, the current bound and a counterexample are reported on the console. When the timeout occurs because the solving process takes too long, QuickCheck is invoked to verify longer arrays.

```
verify_reverseArray :: IO ()
verify_reverseArray = verify reverseArray postCond_reversed
  qc_prop_longerThan 1 1
```

By running this function we find that on an average laptop the postcondition can be proven with SMT on arrays with up to seventy elements before the timeout occurs. After this QuickCheck randomly tests lists of more than 70 elements. Again, the whole verification process indicates that the postcondition holds, but this time thanks to the SMT check the postcondition is actually proven to hold on small domains.

The above example shows that our method works. But the program we used it on is rather trivial, and it comes as no surprise that both verification methods ('pure' QuickCheck and the combined method of SMT plus QuickCheck) report that the postcondition holds. Our hypothesis as stated in chapter 4 claimed that the combined method should be more successful in finding faults than random testing alone. To test this we will deliberately introduce a fault in the reverse array program.

For the kinds of faults we can introduce we somewhat constrained by the background logics used by the SMT solver. In our case we can use linear arithmetic and ordering relations (equal, less-than, greater-than). The fault we introduce can occur when the (arbitrarily chosen) value 3567 is used after the fourth index in the array.

```
decl(N := #0);
decl(i := 0);
decl(j := 0);
decl(t := 0);
decl(array[#0] := #1);
decl(original[#0] := #1);
decl(index := 4);
decl(x := 3567)

index := 4;
x := 3567;
original @= array;
N := #0;
i := 0;
j := N - 1;
while (i < j) do {
  if (~(and((array[i] == x),(i > index)))) then {
    t := array[i];
    array[i] := array[j];
```

```
    array[j] := t;
    j := j - 1
  };
  i := i + 1
}
```

Because the violation of the property will only occur in very specific circumstances the chance that QuickCheck alone will find this error is negligible. However, the SMT solver should be able to find it and produce a counterexample. We create a new Hoare triple generator and functions to run pure QuickCheck and our verification tool on it.

```
reverseArrayFault :: [Int] -> HTriple
reverseArrayFault array = HTriple
  []
  (getProgram "scripts/reverseArrayFault.colt" [show l, show array])
  [postCond_reversed l]
    where
      l = length array

run_qc_reverseArrayFault :: IO ()
run_qc_reverseArrayFault = qc_prop_longerThan 1 reverseArrayFault
  postCond_reversed

verify_reverseArrayFault :: IO ()
verify_reverseArrayFault = verify reverseArrayFault postCond_reversed
  qc_prop_longerThan 1 1
```

As expected, in none of the pure QuickCheck runs was the fault discovered. The combined method, however, produces an interesting result. It proves that the postcondition holds for lists up to twelve elements before it produces a counterexample. At first glance it may have looked as if the postcondition could already be violated with five or six elements in the array. But the SMT check shows that this is not the case. In fact, some quick experiments show that by incrementing the index after which the faulty value can occur, the maximum length of lists that will never violate the property increases with two. This shows that the combined method of property-based testing can lead to a better understanding of a program's behavior.

Also, now we tested the algorithm with a fixed value for x, 3567. Using the SMT extension we can also test the property when x is free.[1] To do this we remove the assignment `x := 3567;` from the program macro. Re-running the verification function on the updated macro shows the same result; the property holds for arrays up to twelve elements. This shows that the actual value of the faulty element is irrelevant.

The above example shows that the combined method with SMT performs better on finding very rare errors than random testing alone. This finding is relevant for algorithms that run a risk of at some point producing an incorrect value, for example due to integer overflow, which corrupts the rest of the calculation. In our example we had to manually introduce the fault as we do not control conditions such as integer overflow.

Bubble sort algorithm

The previous section has shown how to use the combined verification tool on a trivial program. We will now use the tool to verify properties of a sorting algorithm. We will use bubble sort for this purpose. [3] Again we start with a module declaration.

```
module Language.Colt.Examples.BubbleSort where

import Language.Colt.Grammar
import Language.Colt.Verification
import Language.Colt.Parser
import Language.Colt.TestUtils

import Language.Colt.Interpreter

import Data.Map
import qualified Data.List as List
```

The bubble sort program is defined as follows:

[1] It should be noted that making x free can only be done for verification with SMT. QuickCheck still uses the static value assigned to x in its declaration. To do this a new Hoare triple generator type would have to be written that generates an integer and a list of integers.

```
decl(N := #0);
decl(i := 0);
decl(j := 0);
decl(t := 0);
decl(array[#0] := #1);
decl(original[#0] := #1)

original @= array;
N := #0;
j := 0;
while (j<(N-1)) do {
  i := 0;
  while (i<(N-j-1)) do {
    if (array[i]>array[i+1]) then {
      t := array[i];
      array[i] := array[i+1];
      array[i+1] := t
    };
    i := i+1
  };
  j := j+1
}
```

The Hoare triple generator directly uses the input array and its length to parse a concrete instance of the program.

```
bubbleSort :: [Int] -> HTriple
bubbleSort array = HTriple
  []
  (getProgram "scripts/bubbleSort.colt" [show l, show array])
  [postCond_ascending l, postCond_permutation l, postCond_permutation_unique l]
    where
      l = List.length array
```

Like the reverse array program, the bubble sort program poses no additional pre-conditions on the input array.

Two properties are important for the verification of sorting algorithms; the resulting array must in ascending order and it must be a permutation of the original array. [4,

22] For the ascending property we must state the postcondition that every element in the array, except the last, is smaller than or equal to its successor. For a list of three elements the logical from of that postcondition looks like this:

$$(array[0] <= array[1]) \wedge (array[1] <= array[2])$$

```
postCond_ascending :: Int -> Postcondition
postCond_ascending n = Condition "postCond_ascending" (And $ (BConst True) : (conj 0 n))
  where
    conj i 0 = []
    conj i n | i == (n-1) = []
             | otherwise =
      (ExprLtEq
        (V (ArrVar "array" (Const i)))
        (V (ArrVar "array" (Const (i+1))))
      ) : (conj (i+1) n)
```

The ascending property scales quite well; the number of equations needed in the formula is equal to $N-1$ (N being the length of the array). The permutation property is more complex. We must state that the resulting array is equal to one of the permutations of the original array. Because we are not using quantifiers in our formulas we have to state this explicitly for every property. For an array of two elements this becomes:

$$(original[0] == array[0] \wedge original[1] == array[1]) \vee$$

$$(original[0] == array[1] \wedge original[1] == array[0])$$

```
postCond_permutation :: Int -> Postcondition
postCond_permutation n = Condition "postCond_permutation"
                           (Or [ And (permEq is perm) | perm <- perms])
  where
    is = [0..(n-1)]
    perms = List.permutations is
    permEq [] [] = []
    permEq (a:as) (b:bs) = (ExprEq (V (ArrVar "original" (Const a)))
                                   (V (ArrVar "array" (Const b))))
                         : permEq as bs
```

The number of permutations of an array of length N is equal to N factorial, meaning the complexity of the permutation property will increase drastically as the arrays get longer. It can be expected that this will give performance issues to both the SMT solvers and our own interpreter when using QuickCheck.

For this reason we will define an alternative permutation property that grows less fast in complexity. However, this permutation property is only valid for arrays where every elements occurs only once. We state that every element of the original array is present in the resulting array and that the resulting array contains no elements that are not in the original array. The logical formula for an array of two elements looks like this:

$$(original[0] = result[0] \vee original[0] = result[1]) \wedge$$
$$(original[1] = result[0] \vee original[1] = result[1])$$

```
postCond_permutation_unique :: Int -> Postcondition
postCond_permutation_unique n = Condition "postCond_permutation_unique"
  (And (createConj 0))
    where
      createConj i
        | i >= n = []
        | otherwise = Or (createDisj i 0) : createConj (i+1)
      createDisj i j
        | j >= n = []
        | otherwise = ExprEq (V (ArrVar "array" (Const i)))
                             (V (ArrVar "original" (Const j)))
                      : createDisj i (j+1)
```

Since we have three properties to verify, we will write three verification functions so the properties can be checked separately. First we present the code for the verification of the ascending property. We use a timeout of two seconds and start at bound 1, meaning lists with one element.

```
verify_bubbleSort_ascending :: IO ()
verify_bubbleSort_ascending = verify bubbleSort
  postCond_ascending qc_prop_longerThan 2 1
```

Running this function on a standard laptop the SMT solver proves the property up to arrays of six elements. On seven elements the timeout occurs and QuickCheck

is invoked to randomly test lists of seven elements and longer. In none of our test runs did it find a counterexample.

The first permutation property is verified with the following function:

```
verify_bubbleSort_permutation :: IO ()
verify_bubbleSort_permutation = verify bubbleSort
  postCond_permutation qc_prop_longerThan 2 1
```

This property can be proven by SMT up to arrays of five elements before the timeout occurs. Random testing arrays above five elements very quickly runs into performance problems. As a result QuickCheck has to be stopped manually due to memory shortage. The verification of this property could therefore not be entirely completed.

The second property is verified with the following function. Again we use a timeout of two seconds and we start at bound 1.

```
verify_bubbleSort_permutation_unique :: IO ()
verify_bubbleSort_permutation_unique = verify bubbleSort
  postCond_permutation_unique qc_prop_longerThan 2 1
```

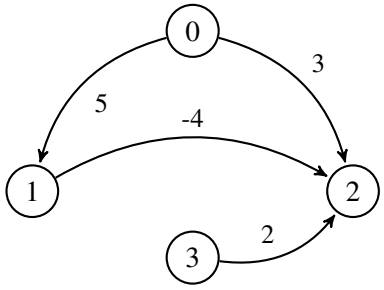
The SMT solver again proves this property for arrays up to five elements. But as the complexity of this property grows less drastically than the previous one, QuickCheck is able to perform 100 random tests on arrays longer than five elements. In none of our test runs did it find a counterexample.

The fact that the alternative permutation property was verified shows that we must take great care in understanding what a postcondition actually states. Neither the SMT solvers nor QuickCheck were configured to use only arrays with unique elements. This means the property was verified for arrays with double elements as well, though we know that in these cases it is *not* a correct check for permutation. Thus any array with double elements may give a false positive.

BellmanFord algorithm

The Bellman-Ford algorithm is a graph algorithm that computes the shortest path between nodes in a weighted graph. [2] It is similar to Dijkstra's shortest path algorithm, but it can also handle negative weights.

As an example, consider the following graph:



If we take node 0 as source node, the Bellman-Ford algorithm produces the following array to represent the shortest path from 0 to each node: [0, 5, 1, ∞] The indices of this array represent the destination nodes. So the shortest path from 0 to 0 is 0, from 0 to 1 is 5, from 0 to 2 is 1 and since there is no path from 0 to 3 this distance is infinite (∞).

We start with the module declaration and imports of other required modules.

```
module Language.Colt.Examples.BellmanFord where

import Language.Colt.Grammar
import Language.Colt.Interpreter
import Language.Colt.Verification
import Language.Colt.Parser
import Language.Colt.TestUtils

import Data.List (nub)
```

We define the Bellman-Ford program as follows and store it as `bellmanFord.colt`:

```
decl(INFINITY := 899);
decl(nodecount := #4);
decl(edgecount := #0);
decl(source := 0);
decl(Source[#0] := #1);
decl(Dest[#0] := #2);
decl(Weight[#0] := #3);
```

```
decl(distance[#4] := #5);
decl(x := 0);
decl(y := 0);
decl(i := 0);
decl(j := 0)

INFINITY := 899;
nodecount := #4;
edgecount := #0;
source := 0;
i := 0;
j := 0;
while (i < nodecount) do {
  ite (i == source) then {
    distance[i] := 0
  }
  else {
    distance[i] := INFINITY
  };
  i := i + 1
};
i := 0;
while (i < nodecount-1) do {
  while (j < edgecount) do {
    x := Dest[j];
    y := Source[j];
    if (distance[x] > distance[y] + Weight[j]) then {
      distance[x] := distance[y] + Weight[j]
    };
    j := j + 1
  };
  i := i + 1
}
```

Note that we have left the check for negative cycles out of the algorithm. This is because this check could interfere with the SMT check of the postcondition. Furthermore, we assure that there are no negative cycles in the preconditions, so this renders the check obsolete.

To parse the program macro into an executable program when given a concrete

array we use a program generator that splits up the given array into three parts of equal length (representing the arrays Source, Dest and Weight) and derives the required arguments. For this to work correctly it is required that the length of the generated array is a multiplication of 3.

```
bellmanFord :: [Int] -> HTriple
bellmanFord array = HTriple
  [preCond_oneNegative break,
   preCond_noSelfLoops break,
   preCond_weightUnderInf break,
   preCond_unionSourceDest break]
  (getProgram "scripts/bellmanFord.colt"
    [show break, show source, show dest,
     show weight, show nodeC, show dummy])
  [postCond_sourceZero break]
    where
      break = length array `div` 3
      (source,rest) = splitAt break array
      (dest,weight) = splitAt break rest
      nodeC = (length . nub) (source ++ dest)
      dummy = take nodeC (repeat 0)
```

The implementation of our Bellman-Ford algorithm imposes several preconditions on its input. For each of these we write a function that takes the length of the three arrays and produces the required precondition. First we create the precondition stating that all weights must be under $INFINITY$.

```
preCond_weightUnderInf :: Int -> Precondition
preCond_weightUnderInf n = Condition "preCond_weightUnderInf" (And $ (BConst True)
                                                                 : (conj 0 n))
  where
    conj i 0 = []
    conj i n | i == n = []
             | otherwise = (ExprLt (V (ArrVar "Weight" (Const i)))
                                   (Const 899)
                           ) : (conj (i+1) n)
```

Secondly the union of the arrays `Source` and `Dest` must contain all values $0..N-1$. This is because these values are used as indices in the array `distance`. If we use other values for `Source` and `Dest` the array `distance` will go out of bounds.

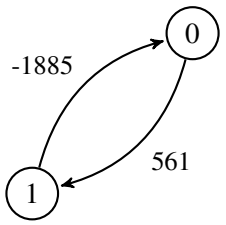
```
preCond_unionSourceDest :: Int -> Precondition
preCond_unionSourceDest 0 = Condition "preCond_unionSourceDest" (BConst True)
preCond_unionSourceDest n = Condition "preCond_unionSourceDest"
  (And $ [Or $ map (eq "Source" i) js | (i,js) <- list]
           ++ [Or $ map (eq "Dest" i) js | (i,js) <- list])
    where
      eq name i j = ExprEq (Const i) (V (ArrVar name (Const j)))
      list = zip [0..(n-1)] (repeat [0..(n-1)])
```

The next two preconditions rule out the possibility of a negative cycle. First we state that there is only one negative weight in the graph whose value is -1. All other values are 1 or larger. Secondly we rule out self loops. This makes negative cycles impossible.

```
preCond_oneNegative :: Int -> Precondition
preCond_oneNegative n = Condition "preCond_oneNegative" (Or $ (disj 0 n))
  where
    disj i n | i == n = []
             | otherwise = (And $ map (comp i) [0..n-1]) : disj (i+1) n
    comp i e | i == e = (ExprEq (V (ArrVar "Weight" (Const e))) (Const (-1)))
             | otherwise = (ExprGtEq (V (ArrVar "Weight" (Const e))) (Const 1))

preCond_noSelfLoops :: Int -> Precondition
preCond_noSelfLoops n = Condition "preCond_noSelfLoops" (And $ map notEqual [0..n-1])
  where
    notEqual i = Neg $ ExprEq (V (ArrVar "Source" (Const i)))
                              (V (ArrVar "Dest" (Const i)))
```

Actually, when first writing these preconditions we stated that there be one negative value, arbitrarily low, and that all other values be 0 or higher. The SMT solver then quickly showed that this still allows for negative cycles; producing the following counterexample:



Here it is easy to see that by following the path $0 \to 1 \to 0 \to 1 \ldots$ the path from 0 to 1 keeps getting shorter. This is invalid input that is ruled out by the current preconditions. This example shows how the SMT check can also help to understand the conditions on the input of a program.

The property that we want to verify on the Bellman-Ford program states that the distance of the source node to itself must always be 0. In logical formula:

$$distance[source] = 0$$

```
postCond_sourceZero :: Int -> Postcondition
postCond_sourceZero n = Condition "propGen_sourceZero"
                        (ExprEq (V (ArrVar "distance"
                                    (V (RegVar "source"))))
                                (Const 0))
```

For the verification function we use a custom input generator `qc_bellmanFord`. This is necessary because the algorithm requires that the nodes in the graph are numbered from $0..N-1$, where N is the number of nodes used in the algorithm. QuickCheck has no standard generator for this, so we had to make our own. The generator also assures that that is at most one negative weight in a generated graph. The complete code of input generator can be found in appendix D. The verification function starts verifying at bound 3, as arrays with under 3 elements are incomplete representations of a graph.

```
verify_bellmanFord_sourceZero = verify bellmanFord postCond_sourceZero
                                qc_bellmanFord 1 3
```

Running the verification function on a standard laptop proved the postcondition for arrays up to 30 elements. This corresponds to graphs containing 10 nodes and 10 edges. After this QuickCheck is invoked to random test arrays of more than 30 elements. In our test runs this didn't find any counterexamples either.

Chapter 9

Analysis and Conclusions

In the previous chapter we have seen the Colt tool in action using using a combined verification approach of SMT solvers and random testing. In this final chapter we will interpret the results of our verification attempts and try to answer the research question of chapter 1, namely whether our combined verification approach succeeds to work on a single property specification and whether it provides stronger verification than a single approach.

Interpretation of the results

The examples of reverse array and bubble sort algorithms show that it is possible to incorporate SMT solving for the verification of properties with very little extra configuration needed. Just like with regular QuickCheck the program and conditions are defined and a dedicated function `verify` is called to start the verification process.

This verification function did need some additional arguments besides the property defined in a Hoare triple. Two of these, the program bound and the timeout could also have been made constant values, but passing them as an argument allows for some customization. The verification function also needs a wrapped version of the property used by QuickCheck. This is due to the fact that we still had to separately define input generators to generate test cases. These input generators can be seen as the precondition guards of random testing. In the current project we did not research how the precondition definition can be used to directly generate test cases, hence we had to build custom functions for this and use them separately from the

preconditions in the property.

The reverse array and in particular the bubble sort example show how the combined approach can give added value compared to random testing. The SMT solver in these cases proves the property on a small domain. Thanks to this, we know for certain that the bubble sort program is correct for arrays of zero to seven elements, regardless of what (integer) values they contain. The scaling problem of the permutation property was more problematic for the random tests than for the SMT solvers. For the reverse array program the property was even proved to hold on arrays up to seventy elements, but this was a very trivial, and short, program. The SMT solving step succeeds to prove properties on small domains, making the whole verification process stronger. The significance of these small domains is of course up to the programmer to judge.

The example program where we introduced a fault into the reverse array algorithm shows that by integrating the SMT approach indeed more violations of the property can be found than by random testing alone. The possibility of the SMT solver to produce counterexamples when a violation can occur makes the approach very similar to that of QuickCheck. A program either passes the test, or a counterexample is produced. Additionally the SMT solver did not only show that violations are possible, but also precisely when. For the example program the solver showed that the fault only becomes relevant after twelve elements. In other words, the SMT solver showed both the domain on which the property *does* hold, as the domain on which it *does not*.

The Bellman-Ford example shows the importance of correctly defining the preconditions of a program. Moreover, we saw how the SMT solving approach can help spotting errors in these conditions, because given a bounded domain, it will find any counterexample. It is interesting to note that the program itself did not contain faults with respect to the *intended* property, but that the property specification was incomplete. This information may be just as valuable to a programmer, as it increases his understanding of the program.

Constraints imposed by using SMT solvers

The most important consequence of translating a program to an SMT formula is that the program has to be sequential. Conditional loops must be bounded by turning them in to a finite list of nested `if` statements. This bound can influence the result of the program; if chosen too low, the statements that were formerly in the

loop may not be executed enough times, which may alter the behavior of the program. If this happens during the execution of a program the user can be warned by a failing *unwinding assertion*, but for the static analysis of an SMT solver this is not possible. The example programs of this research depended one-on-one on the bound for correct execution. For example, the bubble sort algorithm a bound of two (meaning two nested `if`s) is sufficient to sort an array of two elements, a bound of three for three elements and so on. For programs that depend on the bound in another way, this will not hold.

Assertions present in a program are better not used in the SMT specification. This is due to the fact that these assertions seek a positive result (`sat`) but our approach seeks a negative (`unsat`). When combined in one SMT specification these assertions may conflict each other. In the case such a program assertion is satisfiable, it has no added value to the solver. In case it is unsatisfiable it corrupts the results of the solver, as one unsatisfiable assertion renders the whole formula unsatisfiable. Then the verification function concludes the property holds, while this may not be the case. As a consequence, all of the unwinding assertions that are placed in program when it is unwound are not present in the SMT specification and neither are the assertions in the Bellman-Ford algorithm to discover negative cycles.

Contributions and future research

We have shown that at least two methods of property-based testing can be combined effectively to verify a single specification of a property on a program. We have combined a static approach, SMT solving, with a dynamic approach, random testing. We have given a prototype implementation of this on a small imperative language. We have also introduced a more efficient way of assuring single assignment in the context of SMT solving than the traditional Static Single Assignment algorithm.

It would be very interesting to see how the combined approach to property-based testing could be integrated as an extension into a mature programming language. Also, as we stated in the introduction, the benefit of our approach is that various property-based testing methods can be used on the same property specification. SMT solving and random testing are only two of these and it would be exciting to see the integration of different methods of verification, yielding still more strongly verified software.

As a vision for the future we would hope to see a general framework for property-

based testing. In this framework verification methods can be 'plugged in' to the testing process and used directly on property specifications. With such a framework a programmer can exploit the strengths of different verification methods, while writing only a single formal specification.

Bibliography

[1] A. Armando, J. Mantovani, and L. Platania. Bounded model checking of software using smt solvers instead of sat solvers. *Int. J. Softw. Tools Technol. Transf.*, 2009.

[2] R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 1958.

[3] P. Biggar and D. Gregg. Sorting in the presence of branch prediction and caches. Technical report, 2005.

[4] A. Bradley, Z. Manna, and H. Sipma. What's decidable about arrays? *Proceedings of Verification, Model Checking and Abstract Interpretation*, 2006.

[5] R. Bruttomesso. Satisfiability modulo theories - seminar, September 2011.

[6] K. Claessen, N. Een, M. Sheeran, and N. Sörensson. Sat-solving in practice, 2008.

[7] K. Claessen and J. Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *ICFP*, 2000.

[8] L. De Moura and N. Bjørner. Satisfiability modulo theories: introduction and applications. *Commun. ACM*, 54(9), September 2011.

[9] L. De Moura and N. Bjørner. Z3 - a tutorial. Technical report, Microsoft, 2012.

[10] L. De Moura and N. Bjørner. Z3 - guide. Technical report, Microsoft, 2012.

[11] B. Dutertre and L. De Moura. The yices smt solver. 2006.

[12] P. Dybjer, Q. Haiyan, and M. Takeyama. Verifying haskell programs by combining testing and proving. In *Proceedings of the Third International Conference on Quality Software*. IEEE Computer Society, 2003.

[13] L. Erkok. Data.sbv.examples.codegeneration.gcd, 2012.

[14] L. Erkök and J. Matthews. Pragmatic equivalence and safety checking in Cryptol. In *Programming Languages meets Program Verification*. ACM Press, 2009.

[15] M. Gordon. *Programming Language Theory and its Implementation*. Prentice Hall, 1998.

[16] M. Grindal, J. Offutt, and S. Andler. Combination testing strategies: A survey. *Software Testing, Verification, and Reliability*, 15, 2005.

[17] A. Groce and R. Joshi. Random testing and model checking: building a common framework for nondeterministic exploration. In *Proceedings of the 2008 international workshop on dynamic analysis*, 2008.

[18] J. Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 2009.

[19] C. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10), 1969.

[20] T. Liu, M. Nagel, and M. Taghdiri. Bounded program verification using an smt solver: A case study. In *5th International Conference on Software Testing, Verification and Validation (ICST)*, April 2012.

[21] H. Nielson and F. Nielson. *Semantics with applications: a formal introduction*. John Wiley & Sons, Inc., 1992.

[22] N. Suzuki and D. Jefferson. Verification decidability of presburger array programs. *J. ACM*, 1980.

Appendix A

Interpreter

The interpreter for the while language of the Colt tool uses Haskell's State monad. The state contains seperate spaces for regular variables and array variables. A space is a mapping from a variable name to respectively a value or a list of values.

```
module Language.Colt.Interpreter where

import Control.Monad.State
import qualified Data.List as List
import Data.Map as Map
import Language.Colt.Grammar
import Language.Colt.PrettyPrinter

type VarSpace = Map VarName Value
type ArrVarSpace = Map VarName [Value]
type ProgramState = (VarSpace, ArrVarSpace)
type Action = State ProgramState ()
type Intension a = State ProgramState a

-- | Return the same value list, but with the new value replacing the old
--   value at the given index
updateList :: [Value] -> Int -> Value -> [Value]
updateList list i v = list'
  where
    iList = zip (take (length list) [0..]) list
    check (index,val) = if index == i then v else val
```

```
    list' = List.map check iList

showState :: ProgramState -> IO ()
showState (vSpace, aSpace) = do
  showSpace vSpace
  showArrSpace aSpace

showSpace :: VarSpace -> IO ()
showSpace space =
  putStrLn $ concatMap (\(k,v) -> "\n  " ++ k ++ " -> " ++ show v)
           $ Map.assocs space

showArrSpace :: ArrVarSpace -> IO ()
showArrSpace space =
  putStrLn $ concatMap (\(k,v) -> "\n  " ++ k ++ " -> " ++ show v)
           $ Map.assocs space

emptyState :: ProgramState
emptyState = (Map.empty, Map.empty)

skip :: Action
skip = return ()

readValue :: Var -> Intension Value
readValue (RegVar name) = do
  (vSpace,_) <- get
  return $ vSpace ! name
readValue (ArrVar name expr) = do
  (_,aSpace) <- get
  index <- evalExpr expr
  let aList = aSpace ! name
  return $ aList !! index

storeValue :: Var -> Expr -> Action
storeValue (RegVar name) expr = do
  (vSpace, aSpace) <- get
  exprVal <- evalExpr expr
  put $ (insert name exprVal vSpace, aSpace)
storeValue (ArrVar name iExpr) expr = do
  (vSpace, aSpace) <- get
```

```
  exprVal <- evalExpr expr
  index <- evalExpr iExpr
  let curArray = aSpace ! name
  let newArray = updateList curArray index exprVal
  put $ (vSpace, insert name newArray aSpace)

-- | Used to check a property on the state after execution of the algorithm
checkCondition :: ProgramState -> Condition -> Bool
checkCondition state cond = evalState
  (do condResult <- evalBExpr $ condBExpr cond
      return condResult) state

-- | Interprets the given program on the empty space. The final space is returned.
interpret :: Program -> ProgramState
interpret (Program decls stmt) = execState
  (do interpretDecls decls
      interpretStmt stmt) emptyState

-- | For program interpretation the declarations are seen as inital assignments of
--   values to variables, or lists of values to array variables.
interpretDecls :: [Declaration] -> Action
interpretDecls [] = return ()
interpretDecls (RegDecl name x:decls) = do
  (vSpace, aSpace) <- get
  put $ (insert name x vSpace, aSpace)
  interpretDecls decls
interpretDecls (ArrDecl name length vs:decls) = do
  (vSpace, aSpace) <- get -- Add check for length of value list
  put $ (vSpace, insert name vs aSpace)
  interpretDecls decls

interpretStmt :: Statement -> Action
interpretStmt (Assign var expr) = do
  storeValue var expr

interpretStmt (ArrAssign ar1 ar2) = do
  (vSpace, aSpace) <- get
  let valList = (aSpace ! ar2)
  put (vSpace, insert ar1 valList aSpace)
```

```
interpretStmt (Store ar1 ar2 iExpr valExpr) = do
  (vSpace, aSpace) <- get
  index <- evalExpr iExpr
  value <- evalExpr valExpr
  let valList = (aSpace ! ar2)
  let valList' = updateList valList index value
  put (vSpace, insert ar1 valList' aSpace)

interpretStmt (Sequence []) = skip
interpretStmt (Sequence (x:xs)) = do
  interpretStmt x
  interpretStmt (Sequence xs)
interpretStmt (If p stmt) = do
  pval <- evalBExpr p
  if pval then interpretStmt stmt else skip
interpretStmt (Ite p stmt1 stmt2) = do
  pval <- evalBExpr p
  if pval then interpretStmt stmt1 else interpretStmt stmt2
interpretStmt (While p stmt) = do
  pval <- evalBExpr p
  if pval then interpretStmt stmt >>
               interpretStmt (While p stmt) else skip
interpretStmt (Assert p) = do
  pval <- evalBExpr p
  if pval then skip
          else error ("Failed assertion: " ++ printBExpr p)

evalExpr :: Expr -> Intension Value
evalExpr (Const x) = return x
evalExpr (V var) = do readValue var
evalExpr (Add x y) = do
  xval <- evalExpr x
  yval <- evalExpr y
  return $ xval + yval
evalExpr (Subtr x y) = do
  xval <- evalExpr x
  yval <- evalExpr y
  return $ xval - yval
evalExpr (Mult x y) = do
  xval <- evalExpr x
```

```
  yval <- evalExpr y
  return $ xval * yval

evalBExpr :: BExpr -> Intension Bool
evalBExpr (BConst x) = return x
evalBExpr (Neg x) = do
  xval <- evalBExpr x
  return $ not xval
evalBExpr (Or ps) = do
  ps' <- mapM evalBExpr ps
  return (or ps')
evalBExpr (And ps) = do
  ps' <- mapM evalBExpr ps
  return (and ps')
evalBExpr (ExprEq lhs rhs) = do
  lhsval <- evalExpr lhs
  rhsval <- evalExpr rhs
  return $ lhsval == rhsval
evalBExpr (ExprLt lhs rhs) = do
  lhsval <- evalExpr lhs
  rhsval <- evalExpr rhs
  return $ lhsval < rhsval
evalBExpr (ExprGt lhs rhs) = do
  lhsval <- evalExpr lhs
  rhsval <- evalExpr rhs
  return $ lhsval > rhsval
evalBExpr (ExprLtEq lhs rhs) = do
  lhsval <- evalExpr lhs
  rhsval <- evalExpr rhs
  return $ lhsval <= rhsval
evalBExpr (ExprGtEq lhs rhs) = do
  lhsval <- evalExpr lhs
  rhsval <- evalExpr rhs
  return $ lhsval >= rhsval
```

Appendix B

Parser

```
module Language.Colt.Parser where

import Language.Colt.Grammar
import Language.Colt.Interpreter

import Text.Parsec
import Text.Parsec.String
import Text.Parsec.Expr
import Text.Parsec.Token
import Text.Parsec.Language

import Data.String.Utils
import System.IO.Unsafe

def = emptyDef{ commentStart = "{-"
              , commentEnd = "-}"
              , identStart = letter
              , identLetter = alphaNum
              , opStart = oneOf "~&=<>:+-*"
              , opLetter = oneOf "~&=<>:+-*"
              , reservedOpNames = ["~", "<", ">", "==",
                     "<=", ">=", ":=", "+", "-", "*", "or", "and"]
              , reservedNames = ["true", "false", "ite",
                     "if", "then", "else", "while", "do", "decl", "assert"]
              }
```

```
TokenParser{ parens = m_parens
           , identifier = m_identifier
           , reservedOp = m_reservedOp
           , reserved = m_reserved
           , integer = m_integer
           , semiSep = m_semiSep
           , semiSep1 = m_semiSep1
           , braces = m_braces
           , brackets = m_brackets
           , whiteSpace = m_whiteSpace } = makeTokenParser def

parseBExpr :: Parser BExpr
parseBExpr = do
  try $ m_parens parseBExprPure
  <|> parseBExprPure

parseBExprPure :: Parser BExpr
parseBExprPure = do
  try parseBConst
  <|> try parseNeg
  <|> try parseOr
  <|> try parseAnd
  <|> try parseExprEq
  <|> try parseExprLt
  <|> try parseExprGt
  <|> try parseExprLtEq
  <|> parseExprGtEq

parseBConst :: Parser BExpr
parseBConst = do
  (m_reserved "True" >> return (BConst True))
  <|> (m_reserved "False" >> return (BConst True))

parseNeg :: Parser BExpr
parseNeg = do
  m_reservedOp "~"
  bExpr <- parseBExpr
  return $ Neg bExpr
```

```
parseOr :: Parser BExpr
parseOr = do
  m_reservedOp "or"
  terms <- m_parens $ sepBy parseBExpr (char ',')
  return $ Or terms

parseAnd :: Parser BExpr
parseAnd = do
  m_reservedOp "and"
  terms <- m_parens $ sepBy parseBExpr (char ',')
  return $ And terms

parseExprEq :: Parser BExpr
parseExprEq = do
  lhs <- parseExpr
  m_reservedOp "=="
  rhs <- parseExpr
  return $ ExprEq lhs rhs

parseExprLt :: Parser BExpr
parseExprLt = do
  lhs <- parseExpr
  m_reservedOp "<"
  rhs <- parseExpr
  return $ ExprLt lhs rhs

parseExprGt :: Parser BExpr
parseExprGt = do
  lhs <- parseExpr
  m_reservedOp ">"
  rhs <- parseExpr
  return $ ExprGt lhs rhs

parseExprLtEq :: Parser BExpr
parseExprLtEq = do
  lhs <- parseExpr
  m_reservedOp "<="
  rhs <- parseExpr
  return $ ExprLtEq lhs rhs
```

```
parseExprGtEq :: Parser BExpr
parseExprGtEq = do
  lhs <- parseExpr
  m_reservedOp ">="
  rhs <- parseExpr
  return $ ExprGtEq lhs rhs

parseExpr :: Parser Expr
parseExpr = buildExpressionParser table term <?> "Expression"
table = [ [Infix (m_reservedOp "+" >> return Add) AssocLeft]
        , [Infix (m_reservedOp "-" >> return Subtr) AssocLeft]
        , [Infix (m_reservedOp "*" >> return Mult) AssocLeft]
        ]

parseInteger :: Parser Int
parseInteger = try parseNegInt <|> parsePosInt

parsePosInt :: Parser Int
parsePosInt = do
  value <- many1 digit
  return $ read value

parseNegInt :: Parser Int
parseNegInt = do
  char '-'
  value <- many1 digit
  return $ (read value) * (-1)

term :: Parser Expr
term = m_parens parseExpr
       <|> varRef
       <|> constant

constant :: Parser Expr
constant = do
  int <- parseInteger
  return $ Const int

varRef :: Parser Expr
varRef = do
```

```
  ref <- try arrVar <|> regVar
  return (V ref)

regVar :: Parser Var
regVar = do
  name <- m_identifier
  return (RegVar name)

arrVar :: Parser Var
arrVar = do
  name <- m_identifier
  index <- m_brackets parseExpr
  return (ArrVar name index)



parseStmt :: Parser Statement
parseStmt = fmap Sequence (m_semiSep1 stmt1)
  where
      stmt1 = try parseAssStmt
              <|> try parseIfStmt
              <|> try parseIteStmt
              <|> try parseWhileStmt
              <|> parseAssertStmt

parseAssertStmt :: Parser Statement
parseAssertStmt = do
  m_reserved "assert"
  bExpr <- m_parens parseBExpr
  return $ Assert bExpr

parseAssStmt :: Parser Statement
parseAssStmt = try parseArrAssStmt
               <|> parseRegAssStmt

parseArrAssStmt :: Parser Statement
parseArrAssStmt = do
  a2 <- m_identifier
  m_reservedOp "@="
  a1 <- m_identifier
```

```
  return $ ArrAssign a2 a1

parseRegAssStmt :: Parser Statement
parseRegAssStmt = do
  ref <- try arrVar <|> regVar
  m_reservedOp ":="
  expr <- parseExpr
  return $ Assign ref expr

parseIfStmt :: Parser Statement
parseIfStmt = do
  m_reserved "if"
  c <- parseBExpr
  m_reserved "then"
  char '{'
  spaces
  thenStmt <- parseStmt
  spaces
  char '}'
  return $ If c thenStmt

parseIteStmt :: Parser Statement
parseIteStmt = do
  m_reserved "ite"
  c <- parseBExpr
  m_reserved "then"
  char '{'
  spaces
  thenStmt <- parseStmt
  spaces
  char '}'
  spaces
  m_reserved "else"
  char '{'
  spaces
  elseStmt <- parseStmt
  spaces
  char '}'
  return $ Ite c thenStmt elseStmt
```

```
parseWhileStmt :: Parser Statement
parseWhileStmt = do
  m_reserved "while"
  spaces
  c <- parseBExpr
  spaces
  m_reserved "do"
  spaces
  char '{'
  spaces
  stmt <- parseStmt
  spaces
  char '}'
  return $ While c stmt




parseDecls :: Parser [Declaration]
parseDecls = do
  m_semiSep parseDecl

parseDecl :: Parser Declaration
parseDecl = do
  m_reserved "decl"
  varDecl <- m_parens $ try parseArrDecl <|> parseRegDecl
  return varDecl

parseArrDecl :: Parser Declaration
parseArrDecl = do
  name <- m_identifier
  length <- m_brackets $ parseInteger
  m_reservedOp ":="
  values <- m_brackets $ sepBy parseInteger (char ',')
  return $ ArrDecl name length values

parseRegDecl :: Parser Declaration
parseRegDecl = do
  name <- m_identifier
  m_reservedOp ":="
```

```
  value <- parseInteger
  return $ RegDecl name value


parseProgram :: Parser Program
parseProgram = do
  decls <- parseDecls
  spaces
  stmt <- parseStmt
  return $ Program decls stmt

type Arg = String

getProgram :: FilePath -> [Arg] -> Program
getProgram loc args =  unsafePerformIO $ do
  macro <- readFile loc
  let progStr = applyMacro args 0 macro
  let program = case (parse parseProgram "" progStr) of
                  Left err -> error "parse error"
                  Right prog -> prog
  return $ program

applyMacro :: [Arg] -> Int -> String -> String
applyMacro args i str = result
  where
    (arg:rest) = args
    str' = replace ("#" ++ show i) (head args) str
    result = if (null rest)
               then str'
               else (applyMacro rest (i+1) str')


run :: Show a => Parser a -> String -> IO ()
run p input =
  case parse p "" input of
    Left err ->
      do putStr "parse error at "
         print err
    Right x -> print x
```

Appendix C

SAFTransformer

```
module Language.Colt.SAFTransformer (
  toGSA
) where

import Control.Monad.State
import Data.Map as Map hiding (map)
import qualified Data.List as List
import Language.Colt.Grammar
```

The transformation to single assignment appends a numeric suffix to all variables. This suffix is updated in every assignment. To keep track of variables and their suffixes we use Haskell's state monad. The state contains two spaces that map variable names to integers; one space for regular variables and one for arrays.

```
type VarNameSpace = Map VarName Int
type SuffixState = (VarNameSpace, VarNameSpace)
```

References to variables or indices of arrays in the original program are replaced by constructing their new name out of the original name plus suffix. The current suffix is extracted from the state.

```
varToGSA :: Var -> State SuffixState Var
varToGSA (RegVar var) = do
  (vSpace, _) <- get
```

```
  let seqNum = (vSpace ! var)
  let var' = var ++ show seqNum
  return (RegVar var')
varToGSA (ArrVar var iExpr) = do
  (_, aSpace) <- get
  let seqNum = (aSpace ! var)
  let var' = var ++ show seqNum
  iExpr' <- exprToGSA iExpr
  return (ArrVar var' iExpr')
```

Expressions and boolean expressions just push the transformation downwards. Only the variable names have to be updated.

```
exprToGSA :: Expr -> State SuffixState Expr
exprToGSA (Const val) = return (Const val)
exprToGSA (V var) = do
  var' <- varToGSA var
  return (V var')
exprToGSA (Add x y)   = do
  xval <- exprToGSA x
  yval <- exprToGSA y
  return $ (Add xval yval)
exprToGSA (Subtr x y) = do
  xval <- exprToGSA x
  yval <- exprToGSA y
  return $ (Subtr xval yval)
exprToGSA (Mult x y)  = do
  xval <- exprToGSA x
  yval <- exprToGSA y
  return $ (Mult xval yval)

bExprToGSA :: BExpr -> State SuffixState BExpr
bExprToGSA (BConst bool) = return (BConst bool)
bExprToGSA (Neg bexpr) = do
  bexpr' <- bExprToGSA bexpr
  return (Neg bexpr')
bExprToGSA (Or ps) = do
  ps' <- mapM bExprToGSA ps
  return (Or ps')
```

```
bExprToGSA (And ps) = do
  ps' <- mapM bExprToGSA ps
  return (And ps')
bExprToGSA (ExprEq lhs rhs) = do
  lhs' <- exprToGSA lhs
  rhs' <- exprToGSA rhs
  return (ExprEq lhs' rhs')
bExprToGSA (ExprLt lhs rhs) = do
  lhs' <- exprToGSA lhs
  rhs' <- exprToGSA rhs
  return (ExprLt lhs' rhs')
bExprToGSA (ExprGt lhs rhs) = do
  lhs' <- exprToGSA lhs
  rhs' <- exprToGSA rhs
  return (ExprGt lhs' rhs')
bExprToGSA (ExprLtEq lhs rhs) = do
  lhs' <- exprToGSA lhs
  rhs' <- exprToGSA rhs
  return (ExprLtEq lhs' rhs')
bExprToGSA (ExprGtEq lhs rhs) = do
  lhs' <- exprToGSA lhs
  rhs' <- exprToGSA rhs
  return (ExprGtEq lhs' rhs')
```

The transformation of statements is more complex. We will describe the process for each statement separately.

Assignments to regular variables first transform the expression. This assures that the variable references point to the right versions. Then the suffix number is taken from the state and incremented with 1. This will be the new suffix for the variable under assignment. The new suffix is put in the space and the assignment statement with updated variable and expression is returned.

Assignments to indices of arrays work the same, with the addition that also the expression representing the index is transformed before the incrementation of the suffix. Also, for array assignments the Store statement is returned which assigns one array to another, but with the value at index i changed to expression e.

```
stmtToGSA :: Statement -> State SuffixState Statement
stmtToGSA (Assign (RegVar name) expr) = do
```

```
  (vSpace, aSpace) <- get
  expr' <- exprToGSA expr
  let curNum = (vSpace ! name)
  let seqNum = curNum + 1
  put $ (insert name seqNum vSpace, aSpace)
  let name' = name ++ show seqNum
  return (Assign (RegVar name') expr')
stmtToGSA (Assign (ArrVar name iExpr) expr) = do
  (vSpace, aSpace) <- get
  expr' <- exprToGSA expr
  iExpr' <- exprToGSA iExpr
  let curNum = (aSpace ! name)
  let seqNum = curNum + 1
  put $ (vSpace, insert name seqNum aSpace)
  let oldName = name ++ show curNum
  let name' = name ++ show seqNum
  return (Store name' oldName iExpr' expr')
```

Array assingments are a shortcut to copying arrays. Instead of having to assign every value at index i of the first array to index i in the second array this statement handles them all at once. The array to which is assingned does recieve and increased suffix.

```
stmtToGSA (ArrAssign arrName1 arrName2) = do
  (vSpace, aSpace) <- get
  let arrNum2 = aSpace ! arrName2
  let arrName2' = arrName2 ++ show arrNum2
  let curNum = (aSpace ! arrName1)
  let seqNum = curNum + 1
  put $ (vSpace, insert arrName1 seqNum aSpace)
  let arrName1' = arrName1 ++ show seqNum
  return (ArrAssign  arrName1' arrName2')
```

Sequences apply the transformation to all statements under it and return a new sequence with the transformed statements.

```
stmtToGSA (Sequence []) = return (Sequence [])
stmtToGSA (Sequence (x:xs)) = do
```

```
  x' <- stmtToGSA x
  rest <- stmtToGSA (Sequence xs)
  let getStmts (Sequence stmts) = stmts
  let restList = getStmts rest
  return (Sequence (x':restList))
```

All `if-then` statements are transformed into `if-then-else` statements. This is to overcome the problem of indices going out of sync. The new `else` clause will contain an assignment of every variable that was (once or more) assigned in the `then` clause. The new assignments refer back to the name plus suffix as it was before entering the `if`. To achieve this we take the following steps:

1. Store the state as it was upon entering the if-then statement;
2. Transform the condition;
3. Then transform the `then` clause, new assignment may have been done that updated the state;
4. Store the current version of the state;
5. Compare the current state with the original state to see which variables have a lower suffix number in the original state. Those must be added to the `else` clause;
6. Create the assignments by assigning the name plus suffix from the original state to the name plus suffix from the current state. This is done separately for regular variables and arrays, because arrays must use the `store` assignment;
7. Create a sequence of the new assignment statements for variables and arrays to represent the new `else` clause;
8. Return an `if-then-else` statement with the transformed condition, `then` clause and newly created `else` clause.

```
stmtToGSA (If p stmt) = do
  (orgVSpace, orgASpace) <- get
  p' <- bExprToGSA p
  stmt' <- stmtToGSA stmt
  (curVSpace, curASpace) <- get
  let orgVarLowerIndices = lowerIndices orgVSpace curVSpace
```

```
  let orgArrLowerIndices = lowerIndices orgASpace curASpace
  let elseVarAssmts = gsaVarAssmts orgVSpace curVSpace orgVarLowerIndices
  let elseArrAssmts = gsaArrAssmts orgASpace curASpace orgArrLowerIndices
  let elseStmt = Sequence (elseVarAssmts ++ elseArrAssmts)
  return (Ite p' stmt' elseStmt)
```

Transforming `if-then-else` statements is a bit more complex than `if-then` statements because it is possible that variables that were assigned in one clause were not assigned in the other or that they were assigned in both clauses, but one has a higher suffix (meaning it was assigned more than once). There are four cases:

1. Variables that were assigned in `else`, but not in `then`. Of these variables the highest name plus suffix from `else` must be added to the `then` clause. They are assigned the highest name plus suffix from the original state;
2. Variables that were assigned in `then` and in `else`, but have a higher index in `else`. (multiple assignments on the same variable). In this case the highest name plus suffix from the `else` clause must be added to the `then` clause. They are assigned the highest name plus suffix from the (original) `then` clause;
3. Same as (1), but now for the `else` clause;
4. Same as (2), but now for the `else` clause;

The algorithm to achieve this transformation is similar to the transformation of `if-then`, but more elaborate.

```
stmtToGSA (Ite p thenStmt elseStmt) = do
  (orgVSpace, orgASpace) <- get
  p' <- bExprToGSA p
  thenStmt' <- stmtToGSA thenStmt
```

The state resulting from the transformation of the `then` clause is stored and the changed variables and arrays are calculated by comparing it to the original state.

```
  (thenVSpace, thenASpace) <- get
  let thenChangedVars = higherIndices thenVSpace orgVSpace
  let thenChangedArrs = higherIndices thenASpace orgASpace
```

Now the current state is replaced by the original state from before entering the `if-then-else`. Because of this the transformation of the `else` clause will start with the same names plus suffixes as the `then` clause did.

```
put (orgVSpace, orgASpace)
elseStmt' <- stmtToGSA elseStmt
(elseVSpace, elseASpace) <- get
let elseChangedVars = higherIndices elseVSpace orgVSpace
let elseChangedArrs = higherIndices elseASpace orgASpace
```

For both the `then` and the `else` clause the variables and array names are stored that were changed less often in one clause then in the other. Also the variables and array names are stored that were changed in one clause, but not in the other.

```
let thenLessChangedVars = lowerIndices thenVSpace elseVSpace
let thenLessChangedArrs = lowerIndices thenASpace elseASpace
let elseLessChangedVars = lowerIndices elseVSpace thenVSpace
let elseLessChangedArrs = lowerIndices elseASpace thenASpace
let thenNotChangedVars = elseChangedVars List.\\ thenChangedVars
let thenNotChangedArrs = elseChangedArrs List.\\ thenChangedArrs
let elseNotChangedVars = thenChangedVars List.\\ elseChangedVars
let elseNotChangedArrs = thenChangedArrs List.\\ elseChangedArrs
```

Assignments for the variables and arrays that must be added to the clauses are created. Assignments to variables that were not before in the clause refer back to the original state. Variables that were assigned before refer back to their highest name plus suffix from the same clause. This assures that after the `if-then-else` statement all variable and array names are synchronized.

```
let thenVarAssmts1 = gsaVarAssmts orgVSpace elseVSpace thenNotChangedVars
let thenArrAssmts1 = gsaArrAssmts orgASpace elseASpace thenNotChangedArrs
let thenVarAssmts2 = gsaVarAssmts thenVSpace elseVSpace
                       (thenLessChangedVars List.\\ thenNotChangedVars)
let thenArrAssmts2 = gsaArrAssmts thenASpace elseASpace
                       (thenLessChangedArrs List.\\ thenNotChangedArrs)
let elseVarAssmts1 = gsaVarAssmts orgVSpace thenVSpace elseNotChangedVars
let elseArrAssmts1 = gsaArrAssmts orgASpace thenASpace elseNotChangedArrs
let elseVarAssmts2 = gsaVarAssmts elseVSpace thenVSpace
```

```
                        (elseLessChangedVars List.\\ elseNotChangedVars)
  let elseArrAssmts2 = gsaArrAssmts elseASpace thenASpace
                        (elseLessChangedArrs List.\\ elseNotChangedArrs)
```

New statements are created for the `then` and `else` clauses. They are composed of the transformed original clauses and the extra assignments that are needed for synchronization of names plus suffix combinations.

```
  let thenStmt'' = Sequence [ thenStmt', Sequence (thenVarAssmts1 ++ thenVarAssmts2
                              ++ thenArrAssmts1 ++ thenArrAssmts2) ]
  let elseStmt'' = Sequence [ elseStmt', Sequence (elseVarAssmts1 ++ elseVarAssmts2
                              ++ elseArrAssmts1 ++ elseArrAssmts2) ]
```

Because of the replacement of the state earlier in the process variables or arrays that were assigned more often in the `then` clause than in the `else` clause still have an incorrect suffix number in the current state. Therefore the correct suffix numbers are copied from the `then`-state to the current state. Now the current state is up to date. Finally a new `if-then-else` statement is returned with the transformed condition and clauses plus the added assignments.

```
  copyVarIndices thenVSpace (thenChangedVars List.\\ elseChangedVars)
  copyArrIndices thenASpace (thenChangedArrs List.\\ elseChangedArrs)
  return (Ite p' thenStmt'' elseStmt'')
```

The final two statements are straightforward. `while` loops are not accepted, as all loops are expected to be unwound. `assert` statements only have to transform their boolean expression.

```
stmtToGSA (While p stmt) = error "stmtToGSA: Do not use GSA on WHILE loop, unwind first"

stmtToGSA (Assert p) = do
  p' <- bExprToGSA p
  return (Assert p')
```

The transformation of statements uses some auxiliary functions which are defined here.

```
lowerIndices :: VarNameSpace -> VarNameSpace -> [VarName]
lowerIndices a b = Map.keys $ Map.filterWithKey comparePair a
  where comparePair key val = val < (b ! key)

higherIndices :: VarNameSpace -> VarNameSpace -> [VarName]
higherIndices a b = Map.keys $ Map.filterWithKey comparePair a
  where comparePair key val = val > (b ! key)

copyVarIndices :: VarNameSpace -> [VarName] -> State SuffixState ()
copyVarIndices _ [] = return ()
copyVarIndices vSpace' (var:vs) = do
  (vSpace, aSpace) <- get
  let i = (vSpace' ! var)
  put $ (insert var i vSpace, aSpace)
  copyVarIndices vSpace' vs

copyArrIndices :: VarNameSpace -> [VarName] -> State SuffixState ()
copyArrIndices _ [] = return ()
copyArrIndices aSpace' (var:vs) = do
  (vSpace, aSpace) <- get
  let i = (aSpace' ! var)
  put $ (vSpace, insert var i aSpace)
  copyArrIndices aSpace' vs

gsaVarAssmts :: VarNameSpace -> VarNameSpace -> [VarName] -> [Statement]
gsaVarAssmts _ _ [] = []
gsaVarAssmts oldSpace newSpace (var:vs) = assmt : (gsaVarAssmts oldSpace newSpace vs)
  where
    oldIndex = oldSpace ! var
    oldVar = var ++ show oldIndex
    newIndex = newSpace ! var
    newVar = var ++ show newIndex
    assmt = Assign (RegVar newVar) (V (RegVar oldVar))

gsaArrAssmts :: VarNameSpace -> VarNameSpace -> [VarName] -> [Statement]
gsaArrAssmts _ _ [] = []
gsaArrAssmts oldSpace newSpace (var:vs) = assmt : (gsaArrAssmts oldSpace newSpace vs)
  where
    oldIndex = oldSpace ! var
    oldVar = var ++ show oldIndex
```

```
    newIndex = newSpace ! var
    newVar = var ++ show newIndex
    assmt = (ArrAssign newVar oldVar)
```

After the transformation to single assignment form the declarations of the program must be updated. For each name plus suffix combination a new declaration must be created.

```
updateDecls :: [Declaration] -> State SuffixState [Declaration]
updateDecls []        = return []
updateDecls (decl:xs) = do
  newDecls <- updateDecl decl
  rest <- updateDecls xs
  return $ newDecls ++ rest

updateDecl :: Declaration -> State SuffixState [Declaration]
updateDecl (RegDecl name value) = do
  (vSpace, aSpace) <- get
  let lastSeqNum = vSpace ! name
  let newDecls = createRegDecls name value lastSeqNum
  return newDecls
updateDecl (ArrDecl name length vs) = do
  (vSpace, aSpace) <- get
  let lastSeqNum = aSpace ! name
  let newDecls = createArrDecls name length vs lastSeqNum
  return newDecls
```

An original variable declaration is turned into n variable declarations where the sequential number $1..n$ is appended to the original name. The original value is only preserved in the first declaration. Sequential declarations use the default value 0.

```
createRegDecls :: VarName -> Value -> Int -> [Declaration]
createRegDecls name value 0   = [RegDecl (name ++ "0") value]
createRegDecls name value seq = createRegDecls name value (seq - 1)
                                ++ [RegDecl (name ++ show seq) 0]

createArrDecls :: VarName -> Int -> [Value] -> Int -> [Declaration]
createArrDecls name length vs 0   = [ArrDecl (name ++ "0") length vs]
```

```
createArrDecls name length vs seq = createArrDecls name length vs (seq - 1)
               ++ [ArrDecl (name ++ show seq) length (take length $ repeat 0)]
```

Lastly the functions are defined to run the transformation. An empty state is initialized with all variables and array names pointing to a suffix 1. The function `toGSA` takes a loop unwound program and returns a semantically equivalent program in single assignment form.

```
emptyState :: SuffixState
emptyState = (Map.empty, Map.empty)

initState :: [Declaration] -> State SuffixState ()
initState [] = return ()
initState ((RegDecl name _):decls) = do
  (vSpace, aSpace) <- get
  put $ (insert name 0 vSpace, aSpace)
  initState decls
initState ((ArrDecl name _ _):decls) = do
  (vSpace, aSpace) <- get
  put $ (vSpace, insert name 0 aSpace)
  initState decls

tripleToGSA :: HTriple -> State SuffixState HTriple
tripleToGSA (HTriple pres (Program decls stmt) posts) = do
  initState decls
  pres' <- mapM (condToGSA) pres
  stmt' <- stmtToGSA stmt
  decls' <- updateDecls decls
  posts' <- mapM (condToGSA) posts
  return (HTriple pres' (Program decls' stmt') posts')
```

Properties are transformed to single assingment by updating their boolean expression.

```
condToGSA :: Condition -> State SuffixState Condition
condToGSA cond = do
  bExpr' <- bExprToGSA (condBExpr cond)
  return (Condition (condName cond) bExpr')
```

```
toGSA :: HTriple -> HTriple
toGSA prog = evalState (tripleToGSA prog) emptyState
```

Appendix D

TestUtils

```
module Language.Colt.TestUtils where

import Language.Colt.Grammar
import Language.Colt.Interpreter
import Language.Colt.UnloopTransformer
import Language.Colt.SAFTransformer
import Language.Colt.CNFTransformer
import Language.Colt.SMTLIBTransformer
import Language.Colt.Verification

import Control.Monad
import System.Random
import System.IO.Unsafe
import Test.QuickCheck
```

The calling of QuickCheck is wrapped in a function that allows for the use of custom input generators and test conditions. Such a function takes a length, a program generator and a property generator. The length is used to specify the minimum length of the lists that QuickCheck generates. The random list is then injected in both the program generator and the property generator to create a complete program instance that can be checked.

The default function generates lists longer than l, but adds no additional constraints. It uses a custom generator `listLongerThan` for QuickCheck to make sure all generated lists are above length l. First it generates a random integer y where $0 < y < 100$

and then it creates a list of $l + y$ random integers.

```
qc_prop_longerThan :: QCProperty
qc_prop_longerThan l progGen propGen = quickCheck $ forAll (listLongerThan l)
  $ (\xs -> checkCondition (interpret $ getProg $ progGen xs) (propGen (length xs)))


listLongerThan :: Int -> Gen [Int]
listLongerThan l = do
  y <- smallNumber
  replicateM (l+y) arbitrary

smallNumber :: Gen Int
smallNumber = fmap ((`mod` 100) . abs) arbitrary
```

The QuickCheck property generator for the BellmanFord algorithm must assure that the generated list can be split up un thee parts of equal length and the first two of these parts contain only (and all) numbers in the range $0..N$.

```
qc_bellmanFord :: QCProperty
qc_bellmanFord l progGen propGen = quickCheck $ forAll (multThree l) $
  (\xs -> checkCondition (interpret $ getProg $ progGen xs) (propGen (length xs)))
```

Again a custom generator was used to comply to these preconditions. It that takes integer N (between 0 and 100, for practicality), creates a list of 0..N and scrambles the result for the `Source` and `Dest` arrays. Lastly it generates a list of N arbitrary elements. The three lists are then concatenated and returned. As a consequence of this generator every node is exactly once the source of an edge and once the destination.

```
multThree :: Int -> Gen [Int]
multThree l = do
  n <- smallNumber
  let source = scramble [0..l+n]
  let destination = scramble [0..l+n]
  weight <- replicateM (l+n+1) smallNumber
  let index = (fst . pick) weight
  let weight' = replace (-1) index weight
```

```
  let weight'' = if (length weight > 6) then weight' else weight
  return $ source ++ destination ++ weight''

scramble :: [a] -> [a]
scramble [] = []
scramble xs = x : scramble xs'
  where
    (x,xs') = pickRemove xs

pickRemove :: [a] -> (a,[a])
pickRemove xs = (x,xs')
  where
    (n,x) = pick xs
    xs' = take n xs ++ drop (n+1) xs

pick :: [a] -> (Int,a)
pick [] = error "pick from empty list"
pick [x] = (0,x)
pick xs = (n, xs !! n)
  where n = randomNat (length xs)

randomNat :: Int -> Int
randomNat n = unsafePerformIO $
               getStdRandom (randomR (0,n-1))

replace :: a -> Int -> [a] -> [a]
replace elem index xs = map check (zip [0..] xs)
  where
    check (i,v) | i == index = elem
                | otherwise = v
```

Appendix E

PrettyPrinter

```
module Language.Colt.PrettyPrinter where

import Language.Colt.Grammar
import Data.List

printVar :: Var -> String
printVar (RegVar name) = name
printVar (ArrVar name expr) = name ++ "[" ++ printExpr expr ++ "]"

printExpr :: Expr -> String
printExpr (Const x)   = show x
printExpr (V var)     = printVar var
printExpr (Add x y)   = printExpr x ++ " + " ++ printExpr y
printExpr (Subtr x y) = printExpr x ++ " - " ++ printExpr y
printExpr (Mult x y)  = printExpr x ++ " * " ++ printExpr y

printBExpr :: BExpr -> String
printBExpr (BConst p)       = show p
printBExpr (Neg p)          = "~" ++ printBExpr p
printBExpr (Or ps)          = "or(" ++ intercalate "," (map printBExpr ps) ++ ")"
printBExpr (And ps)         = "and(" ++ intercalate "," (map printBExpr ps) ++ ")"
printBExpr (ExprEq lhs rhs) = "(" ++ printExpr lhs ++ " == " ++ printExpr rhs ++ ")"
printBExpr (ExprLt lhs rhs) = "(" ++ printExpr lhs ++ " < " ++ printExpr rhs ++ ")"
printBExpr (ExprGt lhs rhs) = "(" ++ printExpr lhs ++ " > " ++ printExpr rhs ++ ")"
printBExpr (ExprLtEq lhs rhs) = "(" ++ printExpr lhs ++ " <= " ++ printExpr rhs ++ ")"
```

```
printBExpr (ExprGtEq lhs rhs) = "(" ++ printExpr lhs ++ " >= " ++ printExpr rhs ++ ")"

printDecl :: Declaration -> String
printDecl (RegDecl name val) = "decl (" ++ name ++ " := " ++ show val ++ ")\n"
printDecl (ArrDecl name l vs) = "decl (" ++ name ++ "[" ++ show l ++ "] := "
                              ++ show vs ++ ")\n"

printStmt :: Statement -> String
printStmt stmt = printStmt' 0 stmt

printStmt' :: Int -> Statement -> String
printStmt' i (Assign var expr) = indent i ++ printVar var ++ " := " ++ printExpr expr
  ++ "\n"
printStmt' i (Store ar2 ar1 iExpr expr) = indent i ++ ar2 ++ " := (store "
  ++ ar1 ++ " " ++ printExpr iExpr ++ " " ++ printExpr expr ++ ")\n"
printStmt' i (ArrAssign ar2 ar1) = indent i ++ ar2 ++ " := " ++ ar1 ++ "\n"
printStmt' i (Sequence xs) = concatMap (printStmt' i) xs
printStmt' i (If p s) = indent i ++ "if (" ++ printBExpr p ++ ") then {\n"
  ++ printStmt' (i+1) s ++ indent i ++ "}\n"
printStmt' i (Ite p s1 s2) = indent i ++ "ite (" ++ printBExpr p ++ ") then {\n"
  ++ printStmt' (i+1) s1 ++ indent i ++ "}\n" ++ indent i ++ "else {\n"
  ++ printStmt' (i+1) s2 ++ indent i ++ "}\n"
printStmt' i (While p s) = indent i ++ "while (" ++ printBExpr p ++ ") do {\n"
  ++ printStmt' (i+1) s ++ indent i ++ "}\n"
printStmt' i (Assert p) = indent i ++ "assert (" ++ printBExpr p ++ ")\n"

indent :: Int -> String
indent i = concat $ take i (repeat "  ")

printProp :: Condition -> String
printProp (Condition name bExpr) = name ++ ":\n" ++ printBExpr bExpr

printProg :: Program -> String
printProg (Program decls stmt) = concatMap printDecl decls ++ "\n"
    ++ printStmt stmt

printHTriple :: HTriple -> String
printHTriple (HTriple pres prog posts) =
    "Preconditions:\n\n" ++ intercalate "\n" (map printProp pres)
    ++ printProg prog ++ "Postconditions:\n\n" ++ intercalate "\n" (map printProp posts)
```