ETH Zurich
Department of Computer Science
Chair of Programming Methodology

Induction and Termination of Functions for Automatic Program Verification

Bachelor Thesis

Author

Benjamin Fischer
fischebe@student.ethz.ch

Supervised by

Dr. Ioannis Kassios
Prof. Dr. Peter Müller

1 September 2013

Abstract

SIL is an intermediate verification language. This thesis addresses the application of the induction principle and termination of functions for automatic formal verification, realising the techniques as SIL-to-SIL transformations. The experimental evaluation was done with the verifier Silicon, which employs Symbolic Execution and relies on an SMT solver.

In the first part, SIL expressions are transformed by the induction principle, similar to the technique of another language and program verifier Dafny. A heuristic decides when to involve induction and assembles the induction variables in tuples. Alternatively, domain types are interpreted as algebraic data types, which are not expressible in SIL, to enable structural induction. With this second instance of induction, 25 of 38 selected inductive properties were verified automatically. 10 more properties can be verified by manually triggering axioms about case distinctions, leaving questions about further automation open.

The second part explores the termination of functions with variants. The developed technique guesses a variant for each function by deducing a tuple from the arguments and the folding depth of abstract predicates. The termination conditions are then generated to check whether every recursive call decreases the variant guesses along a well-founded relation. This proved successful for simple program examples.

Acknowledgements

I would like to thank all the people who have supported me to complete this thesis during the past six months. I wish to acknowledge the supervision of Dr. Ioannis Kassios. He guided me through the process with a lot of motivation, competence and determination from the first ideas for a project. The meetings with him always helped me to focus on the relevant aspects instead of getting lost in details. Prof. Dr. Peter Müller deserves special gratitude for making such exciting student projects possible.

Moreover, I am indebted to Dr. K. Rustan M. Leino for his inspiring paper on automating induction in Dafny. I wish to thank the researchers of the Semper project for their feedback. Many of their questions and comments gave me another perspective on certain things. The discussions with Dr. Alex Summers proved especially fruitful. I am grateful to Malte Schwerhoff and Stefan Heule for their development of and help with Silicon and SIL, respectively.

Contents

1 Introduction

Formal verification is a field of computer science which is about proving properties of systems like programs. The Semper Intermediate Language (SIL) is a new language to write such programs to be formally verified. This thesis has two distinct parts, both of which improve the automatic verification of SIL programs.

The first part is about induction. Induction is a crucial method of proof for well-founded structures. It is formalised as an axiom in higher-order logic. However, SMT solvers, which are employed by current verifiers for SIL programs, solely support first-order logic with equality. Thus, they cannot directly prove many interesting properties in SIL. Leino [12] successfully automated induction with an SMT solver for another verification language called Dafny. This was a major source of ideas for this thesis.

The second part is about termination of functions. SIL offers functions in the strict sense, that is, they are free from side effects. Specifications in SIL like preconditions and postconditions can call functions. These functions must terminate, otherwise the soundness of the formal verification could be compromised. None of the current verifiers for SIL programs prove termination. It is well known that the halting problem is undecidable. As we shall see, a lot of recursive functions can still be verified to terminate by a simple approach.

2 Background

As a foundation for the technical work later, we introduce the verification language in use as well as relevant mathematical background.

2.1 Semper Intermediate Language (SIL)

The Semper Intermediate Language (abbreviated to SIL) was chosen for the scope of this thesis. This choice seemed natural from the circumstances given by the research group at the time. However, it is important to notice that the concepts can be implemented for other verification languages equally well.

2.1.1 Role of SIL

SIL is an intermediate verification language. Figure 2.1 shows the infrastructure around such an intermediate language. Translators transform programs of so-called front-end languages into programs of a common intermediate language. A verifier now does the formal verification of the translated program. Any pair with a translator and a verifier can be applied, since they are independent of each other.

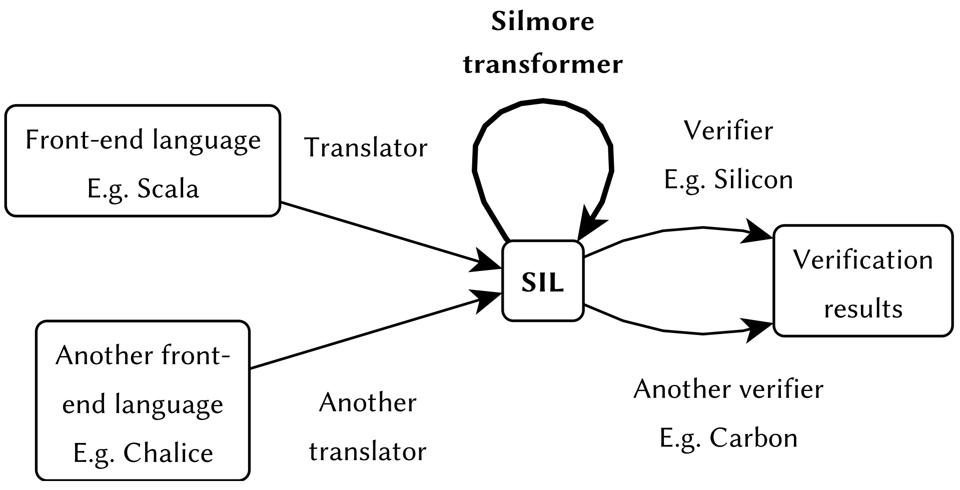
Silmore transformer
Front-end language E.g. Scala
Translator
Verifier E.g. Silicon
SIL
Verification results
Another front-end language E.g. Chalice
Another translator
Another verifier E.g. Carbon

Figure 2.1: SIL as an intermediate verification language (boxes represent data formats, while arrows correspond to transformations between these)

SIL is part of the Semper project and comes with a library [4]. The goal of this ongoing project is the automatic verification of Scala programs. Brodowsky [3] developed a translator for Scala.

Furthermore, Klauser [10] worked on a translation for Chalice, another verification language for concurrent programs [14].

Two common approaches for the construction of automatic program verifiers are Symbolic Execution (SE) and Verification Condition Generation (VCG). Kassios, Müller, and Schwerhoff [9] give an overview of the two approaches followed by a comparison of their efficiency. There currently exist two automatic verifiers for SIL programs: Silicon [15] and Carbon [7]. They implement SE and VCG, respectively. Both verifiers make use of the SMT solver Z3 [2]. We applied Silicon for all verification purposes throughout this thesis.

Besides translators and verifiers, there are transformations working on SIL programs alone. These are essentially SIL-to-SIL functions. Transformations of this third kind are especially reusable, since any other transformation can apply them independently. We shall see how both parts of this thesis can be designed as such SIL transformations, making them available for any translator or verifier. The library developed for SIL transformations is called Silmore.

2.1.2 Relevant Programming Constructs

In the following, we present programming constructs of SIL which are relevant for this thesis. A complete grammar for SIL can be found in [3]. As SIL is a rather new research language, parts of it might change.

Basic Expressions and Statements

The types *`Bool`* and *`Int`* belong to the primitive types of SIL. *`Bool`* is the type of Boolean variables with the literals *`false`* and *`true`*. *`Int`* represents arbitrary-precision integers. Table 2.1 on the next page summarises a selection of SIL operators. As usual, the implication ==> is right-associative and the Boolean operators in order of increasing binding strength are `? :`, ==>, `||`, &&, `!`. Boolean operators apply short-circuit evaluation. The scope of the quantifications *`exists`* and *`forall`* extends to the very right, a rule we also use for our mathematical notation.

Let us consider the complete SIL program of Listing 2.1. It comprises a function `factorial` and a method `factorialImperative` introduced with the corresponding keywords, which compute the factorial in a functional and imperative way, respectively. Functions and methods can have an arbitrary number of parameters.

Listing 2.1: Selection of basic SIL programming constructs

```
function factorial(count: Int): Int
  requires count >= 0
  ensures result >= 1
{ count == 0 ? 1 : count * factorial(count - 1) }

method factorialImperative(count: Int) returns (product: Int)
  requires count >= 0
  ensures product == factorial(count)
{
  var factor: Int
  product := 1
```

Table 2.1: Selected operators of SIL, where a, b are of type *Bool* and m, n of type *Int*, whereas y, z can have either type

Operators	Meaning	Example
`? :`	Conditional expression	`a ? y : z`
`==>`	Implication	`a ==> b`
`||`, `&&`	Disjunction and conjunction	`a || b`
`!`	Boolean negation	`!a`
exists	Existential quantification	*exists* `z:` *Bool* `:: a`
forall	Universal quantification	*forall* `z:` *Int* `:: a`
`==`, `!=`	Equality and inequality	`y != z`
`<`, `<=`, `>`, `>=`	Ordering	`m < n`
`+`, `-`, `*`	Addition, subtraction and multiplication	`m - n`
`\`, `%`	Integer division and modulo	`m \ n`
`+`, `-`	Plus and minus signs	`-n`

```
  factor := 0
  while (factor < count)
    invariant factor >= 0 && product == factorial(factor) && factor <= count
  {
    factor := factor + 1
    product := product * factor
  }
}
```

The body of a function in SIL is an expression, that is, it is free from side effects. Expressions can call functions, thereby enabling recursion as seen in the example. A function can specify a precondition and a postcondition by a sequence of lines with the keywords *requires* and *ensures*, respectively. Such specifications are in turn Boolean expressions, which implies that they can call functions. A postcondition of a function can refer to the evaluation of its body with the keyword *result*.

A function with specifications expresses a contract: whenever the function is called such that its precondition holds and the evaluation of its body terminates, then the postcondition holds. Satisfying this property is called *partial correctness.* If we additionally require that the function terminates, this guarantee is referred to as *total correctness.* In the case of our function `factorial`, total correctness means that for every natural number n, the call `factorial(n)` terminates and returns a nonzero natural number.

Methods can specify preconditions and postconditions, too. In contrast to functions, methods can compute a tuple of results indicated by the keyword *returns*. This tuple is possibly empty, because methods may solely live from their side effects. Method bodies can first declare a number of local variables with *var* followed by a sequence of statements. Possible SIL statements include variable assignments with `:=`, method calls and conditional statements with the keywords *if*, *elsif* and *else* as expected. A loop is introduced with the keyword *while*. Loops can express an invariant with the keyword *invariant*. A loop invariant should hold before the entry of the loop and after each iteration.

Inhaling, Exhaling and Inhale-Exhale Expressions

It is the objective of formal verification to prove that a program is correct in a certain sense. For example, Listing 2.1 on page 8 can be verified by Silicon. Two fundamental notions for the model of SIL programs are inhaling and exhaling. These respective terms refer to assuming and asserting Boolean expressions during the verification. In fact, there exist the keywords *inhale* and *exhale* in SIL.

To see this concept in use, think symbolically of a function `foo` as in

```
function foo(…): T
  requires pre
  ensures post
{ b }
```

with a precondition `pre`, a postcondition `post` and a body `b` of type `T`. Formally verifying partial correctness of this function can work as follows. The verifier assumes `pre`, processes `b` and finally has `post` as a proof obligation. Thus, this could be translated into

```
inhale pre
// Process function body b
exhale post
```

When the verifier encounters a call to `foo`, this works the other way round. `pre` is exhaled before the call and `post` is inhaled after the call.

A new construct in SIL are the so-called *inhale-exhale expressions*. An inhale-exhale expression has the syntax

```
[i, e]
```

where `i` and `e` are Boolean expressions. The semantics is that when `[i, e]` is inhaled, `i` is inhaled, while when `[i, e]` is exhaled, `e` is exhaled. In other words, `[i, e]` is replaced by `i` or `e` for inhaling or exhaling, respectively. As a possible justification, an inhale-exhale expression can be employed if `i` and `e` are equivalent.

Let us clarify inhale-exhale expressions in the context of formal verification with the example of a function `bar` as in

```
function bar(…): T
  requires [pre_in, pre_ex]
  ensures [post_in, post_ex]
{ b }
```

For partial correctness of `bar`, the verifier generates

```
inhale pre_in
// Process function body b
exhale post_ex
```

From the perspective of calling bar, the verifier translates the call into

```
exhale pre_ex
inhale post_in
```

Domain Types

Besides the primitive types, SIL programs can define domain types. Domain types are a general concept to introduce own theories for formal verification. For instance, they allow the definition of structures like sets, lists or natural numbers. A domain type is a collection of function signatures accompanied by axioms in first-order logic with equality. Since domain functions neither have specifications nor bodies, the axioms usually specify their behaviour. In addition, a domain type can have type variables to abstract from the concrete types of its domain functions.

Listing 2.2 defines a domain type for binary trees with the keys in their leaves. The keyword *domain* precedes the name Tree of the domain type together with the type variable A for the type of the keys. Keys can be wrapped into leaves with the domain function leaf. In the recursive nature of trees, the domain function node shows that an inner node has a left and a right subtree. The first axiom leafOrNode guarantees that any tree is equal to an application of leaf to some key or node to some pair of trees. The domain function key unwraps the key of a leaf again, as stated by the second axiom keyLeaf. Applying key to an inner node is undefined. The remaining axioms define equality on trees by matching patterns for a pair of trees.

Listing 2.2: Domain type representing a binary tree with the keys in its leaves

```
domain Tree[A] {
  function leaf(key: A): Tree[A]

  function node(left: Tree[A], right: Tree[A]): Tree[A]

  function key(tree: Tree[A]): A

  axiom leafOrNode {
    forall t: Tree[A] :: (exists k: A :: t == leaf(k)) ||
      exists l: Tree[A], r: Tree[A] :: t == node(l, r)
  }

  axiom keyLeaf {
    forall k: A :: key(leaf(k)) == k
  }

  axiom equalLeafLeaf {
    forall l: A, r: A :: (leaf(l) == leaf(r)) == (l == r)
  }

  axiom equalLeafNode {
    forall k: A, l: Tree[A], r: Tree[A] :: leaf(k) != node(l, r)
  }

  axiom equalNodeLeaf {
    forall l: Tree[A], r: Tree[A], k: A :: node(l, r) != leaf(k)
  }

  axiom equalNodeNode {
    forall lL: Tree[A], lR: Tree[A], rL: Tree[A], rR: Tree[A] ::
      (node(lL, lR) == node(rL, rR)) == (lL == rL && lR == rR)
  }
}
```

An important thing to note is that domain types may describe infinite structures. For example, infinite trees conform to the domain type of Listing 2.2 on the previous page. Even though infinite trees cannot be constructed by a finite number of applications of the domain functions `leaf` and `node`, they can be deconstructed into a left and right subtree, thereby obeying the rules of this domain type. A way to think about domain types is as a set of objects restricted by rules. To further restrict the set to finite trees, one could additionally provide an axiom asking for the smallest such set. However, an axiom like this cannot be expressed in first-order logic.

Fields, Permissions, Predicates and Unfolding Expressions

SIL offers another primitive type *`Ref`* for a reference to a collection of memory locations in the heap. Such memory locations are instances of the fields of a program, which are declared like variables on the top level of the program. For example, Listing 2.3 on the following page has the fields `next` and `item`. Since the field `next` is in turn a reference, this recursively defines a linked list of integers. References may be *`null`*. Variables of type *`Ref`* are passed to a method by reference, which introduces the possibility of side effects on the referenced memory locations. This is one way of aliasing.

Methods and functions in SIL need to provide an upper bound on the part of the heap they modify. This specification problem is an instance of the so-called *frame problem* and is crucial for the modular verification of programs. The central concept of permissions helps to solve this frame problem. Permissions are a way to express access rights on heap locations like read or write access. SIL bases its permission model on Chalice. Leino, Müller, and Smans [14] give a tutorial about Chalice, including an informal guide to permissions. A speciality of SIL is that it supplies constructs for permissions at the level of an intermediate language.

Predicates abstract from permissions. In Listing 2.3 on the next page, the predicate `valid` is a wrapper for write access to the first item of the given list, its reference to the remaining list and recursively to the remaining list if it exists. Note that `r.p()` for a reference `r` and a predicate `p` represents a call `p(r)`. The function `sum` recursively adds the items of a linked list. For this reason, it needs to have read access to the complete list. So its precondition gives it just enough permissions by the keyword *wildcard*. Functions always implicitly give back the permissions they require, thus, there is no postcondition ensuring this.

In the body of the function `sum`, we apply the keyword *unfolding* to exchange the predicate `valid` of `list` for its permissions on the `next` reference, the current `item` and the predicate `valid` of the `next` list. Such unfolding expressions are can be thought of as unwrapping one layer of abstraction in order to reveal memory locations or more layers of abstraction. Unfolding expressions only unfold permissions temporarily for their evaluation, leaving the predicates folded otherwise. For this reason, there is no need for a folding expression. What exists are *fold* and *unfold* statements covered in [14].

Listing 2.3: Linked list with a predicate `valid`

```
var next: Ref

var item: Int

predicate valid(list: Ref)
{ acc(list.item, write) && acc(list.next, write) &&
    (list.next != null ==> acc(list.next.valid(), write)) }

function sum(list: Ref): Int
  requires acc(list.valid(), wildcard)
{ unfolding acc(list.valid(), wildcard) in
    list.next == null ? list.item : list.item + sum(list.next) }
```

What will be relevant later is that when evaluating an expression, the number of successfully executed unfolding expressions is always finite. More precisely, we are given an arbitrary unfolding expression `u` as

```
unfolding p in e
```

for expressions `p` and `e`. We assume that the evaluation of `p` terminates, so that the unfolding eventually succeeds. Prior to evaluating `u`, only a finite number $n \in \mathbb{N}$ of foldings can be executed. As a result, the evaluation of `u` can execute at most n unfolding expressions, since expressions cannot produce new foldings.

2.2 Mathematical Background

We now lay the relevant foundations of mathematics. What connects the two parts of induction and termination of functions is the notion of well-foundedness.

2.2.1 Well-Foundedness

Definition 1. A binary relation $\prec$ on a class C is *well-founded* precisely if any nonempty subset of C has a minimal element with respect to $\prec$, that is,

$$\forall S \subseteq C \bullet S \neq \emptyset \to \exists m \in S \bullet \forall e \in S \bullet \neg(e \prec m).$$

We reserve the symbol $\prec$ solely for well-founded relations. Due to the irreflexivity of $\prec$, some texts refer to it as a strictly well-founded relation. [8]

Equivalently, a relation $\prec$ on a class C is well-founded if and only if there exists no infinite sequence e_0, e_1, e_2, ... with $e_n \in C$ such that $e_{n+1} \prec e_n$ for each $n \in \mathbb{N}$. Intuitively, every decreasing chain constructed with $\prec$ eventually terminates when reaching a minimal element.

Well-founded relations can be defined on various sets.

- The strict inequality $<$ on the natural numbers $\mathbb{N}$ is well-founded, since any decreasing chain $n_0 > n_1 > n_2 > ...$ is limited by zero and hence finite.
- Leino [12] suggests a well-founded relation on the integers $\mathbb{Z}$ as

  $$k \prec n \leftrightarrow k < n \wedge b \leq n \tag{2.1}$$

  for an arbitrary $b \in \mathbb{Z}$.
- A well-founded relation on ordered pairs is possible. Let the ordered pairs be a subset of $S_0 \times S_1$ with well-founded relations $\prec_0$ on S_0 and $\prec_1$ on S_1. For an equivalence relation $=$ on S_0, a well-founded relation is

  $$(l_0, l_1) \prec (h_0, h_1) \leftrightarrow l_0 \prec_0 h_0 \vee (l_0 = h_0 \wedge l_1 \prec_1 h_1). \tag{2.2}$$

  If $\prec_0$ and $\prec_1$ are partial orders, then this relation $\prec$ is known as a lexicographical order.
- Tuples of arbitrary lengths can be represented recursively by ordered pairs. For example, an alternative representation of the triple (a, b, c) is the ordered pair $(a, (b, c))$. Therefore, a well-founded relation on n-tuples can be reduced to a well-founded relation on ordered pairs.
- Given a well-founded relation $\prec_0$ on a set S_0, we can define a well-founded relation on a set S_1 by applying a function $f : S_1 \to S_0$ as in

  $$l \prec h \leftrightarrow f(l) \prec_0 f(h). \tag{2.3}$$

- Finally, algebraic data types allow well-founded relations. An algebraic data type defines a set A by a set of n data constructors

  $$\{c_i : S_{i,0} \times S_{i,1} \times ... \times S_{i,a_i-1} \to A \mid i \in \{0, 1, ..., n-1\}\}.$$

  For every i, the data constructor c_i is a function of arity a_i with codomain A. A is now the smallest set such that

  $$\forall i \in \{0, ..., n-1\} \bullet \forall t \bullet t \in S_{i,0} \times ... \times S_{i,a_i-1} \to c_i(t) \in A.$$

  Recursive structures are possible as soon as $S_{i,k} = A$ for some i and k.

  Based on an idea by Leino [12], for a set A produced by an algebraic data type, we define an application of one of its data constructors to succeed each of its arguments of the same type A along a well-founded relation. More formally, the relation $\prec$ on A is characterised by

$$\forall i \bullet \forall e_0 \in S_{i,0}, ..., e_{a_i-1} \in S_{i,a_i-1} \bullet \forall k \bullet S_{i,k} = A \to e_k \prec c_i(e_0, ..., e_{a_i-1}). \quad (2.4)$$

Take the algebraic data type List as an example. It has the data constructors

$$\{\text{nil} : \text{List}, \quad \text{cons} : S \times \text{List} \to \text{List}\}$$

to create the empty list and prepend an element of the set S to another list, respectively. According to our template, the well-founded relation $\prec$ on List is defined by

$$\forall e_0 \in S, e_1 \in \text{List} \bullet e_1 \prec \text{cons}(e_0, e_1).$$

2.2.2 Noetherian Induction

Definition 2. The axiom of *Noetherian induction* states that for any well-founded relation $\prec$ on a set S and an arbitrary predicate P on S, the formulae

$$\forall n \bullet P(n) \quad (2.5)$$

and

$$\forall n \bullet (\forall k \bullet k \prec n \to P(k)) \to P(n) \quad (2.6)$$

are equivalent when quantifying over S.

The axiom of Noetherian induction is stated in second-order logic, since it quantifies over predicates P. As SMT solvers just employ first-order logic with equality, an idea by Leino [12] is to translate between the formulae of Equation (2.5) and Equation (2.6) as part of a translation to an intermediate verification language. To put this fruitful idea in a nutshell, Equation (2.5) is used for assumptions and Equation (2.6) for assertions. An SMT solver often seems to have a better chance to prove Equation (2.6).

If a set A is characterised by an algebraic data type, we can make use of the well-founded relation $\prec$ defined by Equation (2.4) to instantiate the rule of Noetherian induction for A. This kind of induction on algebraic data types is referred to as structural induction.

Definition 3. Let the set A be defined by an algebraic data type with n data constructors

$$\{c_i : S_{i,0} \times S_{i,1} \times ... \times S_{i,a_i-1} \to A \mid i \in \{0, ..., n-1\}\}.$$

The rule of *structural induction* states that for an arbitrary predicate P on A, the formulae

$$\forall n \bullet P(n)$$

and

$$\forall i \bullet \forall e_0 \in S_{i,0}, ..., e_{a_i-1} \in S_{i,a_i-1} \bullet (\forall k \bullet S_{i,k} = A \to P(e_k)) \to P(c_i(e_0, ..., e_{a_i-1}))$$

are equivalent.

Let us come back to our example with the algebraic data type List from Section 2.2.1 on page 13. In this case, the rule of structural induction states that for a predicate P on List, the formula $\forall n \bullet P(n)$ is equivalent to

$$P(\text{nil}) \land \forall e_0 \in S, e_1 \in \text{List} \bullet P(e_1) \to P(\text{cons}(e_0, e_1)). \quad (2.7)$$

2.2.3 Recursive Functions with Variants

We introduce important terms to see later how these play together for the termination of recursive functions. Definitions 4, 8 and 9 on the current page and on the following page are not really official, but rather creations by the author.

Call Graphs and Strongly Connected Components

Definition 4. A set F of functions is *closed* precisely if for any $f \in F$, if a call to t appears in the definition of f, then $t \in F$.

Definition 5. The *call graph* of a closed set of functions V is a directed multigraph $G = (V, A)$. A is the multiset with an arc (f, t) for every call to function t in the definition of function f.

Definition 6. Let f be an arbitrary function and (V, A) a call graph with $f \in V$. f is called

- *directly recursive* if $(f, f) \in A$,
- *indirectly recursive* if f is part of a cycle with at least two functions,
- *recursive* if f is directly or indirectly recursive

and vice versa.

Definition 7. The *strongly connected components* of a directed multigraph G are the subgraphs G_0, G_1, ..., G_{n-1} of G such that n is minimal and for any i, all vertices of G_i can reach each other.

When every strongly connected component of a directed graph G is contracted to one vertex, the resulting directed multigraph is called the condensation of G. The condensation is acyclic; otherwise, the original strongly connected components would not be maximal. An example of a graph and its condensation is shown in Figure 2.2.

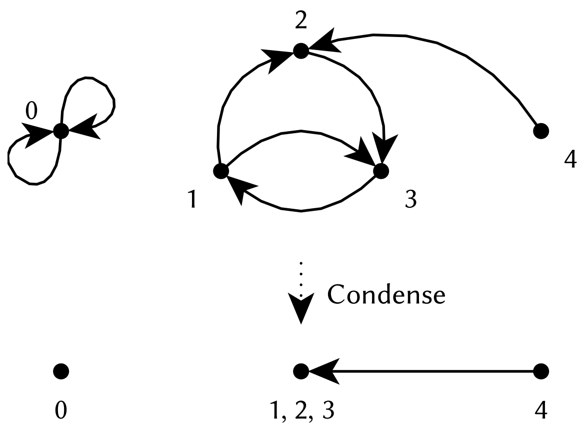

Figure 2.2: Directed multigraph with its condensation, which is acyclic

Evaluation Trees and Variants

Definition 8. The *evaluation tree of a function call* $f(a_0, ..., a_{n-1})$ is a rooted tree $T = (V, A)$. The set of vertices V is made of all function calls encountered during the evaluation of the root $f(a_0, ..., a_{n-1})$. The set of arcs A is the set of all pairs (p, c) of function calls, where the parent p directly calls the child c. T can be infinite.

As an example, let us consider the Fibonacci numbers calculated by the SIL function `fibonacci` of Listing 2.4. The evaluation tree of the function call `fibonacci(3)` is shown in Figure 2.3.

Listing 2.4: Fibonacci numbers in SIL

```
function fibonacci(n: Int): Int
  requires n >= 0
{ n <= 1 ? n : fibonacci(n - 1) + fibonacci(n - 2) }
```

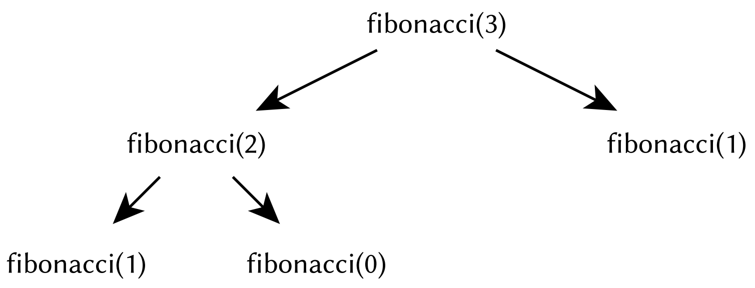

Figure 2.3: Evaluation tree of the function call `fibonacci(3)` for Listing 2.4

An evaluation of a function call terminates if and only if the evaluation tree of this function call is finite. The tree must be finite if every path from the root of an evaluation tree can be interpreted as a decreasing chain with respect to a well-founded relation.

Definition 9. Let C be the functions of a strongly connected component of a call graph. A *variant of the strongly connected component* C is a function v mapping calls to functions of C to elements of a set S with a well-founded relation $\prec$ such that the following holds: any evaluation tree (V, A) is such that for every arc $(p, c) \in A$ where both p and c denote calls to functions of C, we have $v(c) \prec v(p)$. If v is partially applied to a function $f \in C$, we call the resulting function v_f a *variant of the function* f.

In the example of Listing 2.4, a variant of the function `fibonacci` is the current argument with the well-founded relation $<$ on the natural numbers $\mathbb{N}$. The reason is that the recursive calls have $n - 1$ or $n - 2$ as arguments, respectively, both of which are smaller than n and still in $\mathbb{N}$.

3 Induction

The axiom of Noetherian induction from Definition 2 on page 15 gives us reason to translate an expression like

$$\forall n \bullet P(n)$$

into an inhale-exhale expression

$$[\forall n \bullet P(n),\ \forall n \bullet (\forall k \bullet k \prec n \rightarrow P(k)) \rightarrow P(n)] \tag{3.1}$$

for a predicate P. This is still abstract and leaves many questions about the implementation unanswered. A central aspect is the decision for a well-founded relation $\prec$ on the universe quantified over. In the following, we show the technical work for two possibilities.

Another important question arises in the general case of a universal quantification

$$\forall n_0, n_1, ... \bullet P(n_0, n_1, ...)$$

with an arbitrary number of variables over possibly different universes. On which variables do we do induction? If we select no variable at all, then we suppress the translation. If we select exactly one variable n_i, we can reduce this to

$$\forall n_i \bullet P'(n_i)$$

for the predicate

$$P'(n_i) = \forall n_0, ..., n_{i-1}, n_{i+1}, ... \bullet P(n_0, n_1, ...).$$

Lastly, if we decide to select multiple variables, what well-founded relation on them would make sense?

3.1 Induction on Tuples

We first consider a way to do induction on an arbitrary number of variables by assembling them in a tuple.

3.1.1 Well-founded Relation on Tuples

Recall the well-founded relation on tuples based on Equation (2.2) on page 14. As elements of the tuples, we permit objects of arbitrary sets S as long as there is a function $f : S \rightarrow \mathbb{Z}$ mapping this set to integers. Then the well-founded relation applied to such tuples is the well-founded relation on the integer tuples constructed by mapping the objects to integers with their respective function f. That is, we reduce the well-founded relations for each S by Equation (2.3) on page 14 to the one on integers of Equation (2.1) on page 14.

3.1.2 Induction on Integers

Let us have a look at an example with induction on a single variable of type `Int`. We would like to verify that the sum of the first n odd natural numbers equals n^2, that is,

$$\sum_{k=1}^{n}(2k-1) = n^2$$

for all $n \in \mathbb{N}$. The SIL program of Listing 3.1 shows a function `sumOdd` to add the first n odd natural numbers. We then use this function to express the proof obligation with a method `checkClosedFormSumOdd`. Methods with empty bodies like this are helpful to prove formulae solely about functions. However, Silicon is not able to verify this program.

Listing 3.1: Original SIL program to prove $\sum_{k=1}^{n}(2k-1) = n^2$, not verified by Silicon

```
function sumOdd(n: Int): Int
  requires n >= 0
{ n == 0 ? 0 : (2 * n - 1 + sumOdd(n - 1)) }

method checkClosedFormSumOdd()  // Not verified
  ensures forall n: Int :: n >= 0 ==> sumOdd(n) == (n * n)
{ }
```

This is where Silmore comes into play. Applying the induction transformation for tuples to Listing 3.1 results in the program of Listing 3.2. The universal quantification is transformed into an inhale-exhale expression with the pattern of Equation (3.1) on the preceding page. We recognise the predicate $P(n)$ as

```
n >= 0 ==> sumOdd(n) == (n * n)
```

and the well-founded relation $\prec$ on integers as

```
n_1 < n_0 && 0 <= n_0
```

with $b = 0$ in Equation (2.1) on page 14. The transformed program is successfully verified by Silicon.

Listing 3.2: Transformation of Listing 3.1 with Silmore, verified by Silicon

```
function sumOdd(n: Int): Int
  requires n >= 0
{ n == 0 ? 0 : (2 * n - 1 + sumOdd(n - 1)) }

method checkClosedFormSumOdd()  // Verified
  ensures [forall n: Int :: n >= 0 ==> sumOdd(n) == (n * n),  // P(n)
    forall n_0: Int ::
      (forall n_1: Int :: n_1 < n_0 && 0 <= n_0 ==>  // n_1 ≺ n_0
        n_1 >= 0 ==> sumOdd(n_1) == (n_1 * n_1)) ==>  // P(n_1)
          n_0 >= 0 ==> sumOdd(n_0) == (n_0 * n_0)]  // P(n_0)
{ }
```

3.1.3 Mapping Domain Types to Integers

Variables of a domain type can be used as elements of tuples to do induction on if a function mapping the domain type to integers is provided. By default, Silmore automatically chooses the first domain function in the definition of a domain type D which maps D to integers. For instance, for the domain type `List[A]` of Listing 3.3 offers the function `length` as a mapping to integers.

Listing 3.3: Domain type `List[A]` without axioms about equality of lists

```
domain List[A] {
  function nil(type: A): List[A]

  function cons(element: A, list: List[A]): List[A]

  function length(list: List[A]): Int

  function concatenate(left: List[A], right: List[A]): List[A]

  axiom nilOrCons {
    forall l: List[A] ::
      (exists t: A :: l == nil(t)) || exists e: A, r: List[A] :: l == cons(e, r)
  }

  // Axioms about equality of lists omitted

  axiom lengthNil {
    forall t: A :: length(nil(t)) == 0
  }

  axiom lengthCons {
    forall e: A, xs: List[A] :: length(cons(e, xs)) == (length(xs) + 1)
  }

  axiom concatenateNilList {
    forall t: A, ys: List[A] :: concatenate(nil(t), ys) == ys
  }

  axiom concatenateConsList {
    forall x: A, xs: List[A], ys: List[A] ::
      concatenate(cons(x, xs), ys) == cons(x, concatenate(xs, ys))
  }
}
```

For the next example, we define the domain type `List[A]` for lists with elements of type `A` as in Listing 3.3. The domain functions `nil` and `cons` construct the empty list and a list with an element prepended, respectively. Note that `nil` only has a parameter for the sake of determining the type for the type variable `A` when constructing the empty list. `length` computes the size of the list and finally, `concatenate` joins two lists.

The property to be verified is that the concatenation of lists is associative as formalised by Listing 3.4. For properties which should hold for any type for type variables of a domain type, we introduce the domain type `Any`. `Any` has no restrictions and can thus be replaced by any other type.

Listing 3.4: Original property, not verified by Silicon

```
domain Any {
}

method checkConcatenationAssociative()  // Not verified
  ensures forall l: List[Any] ::  // P(l)
    forall m: List[Any], r: List[Any] ::
      concatenate(concatenate(l, m), r) == concatenate(l, concatenate(m, r))
{ }
```

Let $P(l)$ be the predicate

```
forall m: List[Any], r: List[Any] ::
  concatenate(concatenate(l, m), r) == concatenate(l, concatenate(m, r))
```

to improve the readability of the results. The transformed program is shown in Listing 3.5. The domain function `length` is applied to lists for a well-founded relation. However, the property is still not verified by Silicon. For the induction hypothesis to be of use for the verifier, it needs to know that the `length` of a `List` is always natural, namely

```
0 <= length(l_0)
```

which in turn calls for induction. Even if we give this to the verifier as an axiom, the property cannot be verified.

Listing 3.5: Transformation of Listing 3.4 with integer mapping `length` of `List[A]`, still not verified by Silicon

```
method checkConcatenationAssociative()  // Not verified
  ensures [forall l: List[Any] :: P(l),
    forall l_0: List[Any] :: (forall l_1: List[Any] ::
      length(l_1) < length(l_0) && 0 <= length(l_0) ==> P(l_1)) ==> P(l_0)]
{ }
```

3.1.4 Selecting Induction Variables

Induction is not always of benefit to a verifier, but might instead just reduce its efficiency. Furthermore, one could prefer not to do induction on all variables of a universal quantification, but make a selection. Therefore, it is worth thinking about an approach to automatically select the induction variables.

Given $\forall n \bullet P(n)$, Leino [12] proposes to select the variable n for induction if $P(n)$ has a call to a recursive function with n as part of an argument. Silmore applies this heuristic to expressions like

$$\forall n_0, n_1, ... \bullet P(n_0, n_1, ...)$$

in order to compose the tuple of variables n_i to do induction on. In the example of Listing 3.1 on page 19, the function `sumOdd` of is recursive, thereby triggering induction for the universal quantification of `checkClosedFormSumOdd`.

Silmore allows to customise the rule to select induction variables. Besides the default heuristic, there exists an induction transformation which forces induction on all variables. Solely the outermost universal quantification is considered, but not possibly nested ones. Forcing induction may fulfil its purpose for expressions without function calls. Another use case are domain functions, since they do not have bodies to be examined for recursion. For instance, induction is forced to arrive at Listing 3.5 on the previous page.

3.2 Structural Induction on Domain Types

Algebraic data types are a very expressive modelling construct. They intrinsically come with a well-founded relation and enable structural induction by Definition 3 on page 15.

3.2.1 Representing Algebraic Data Types by Domain Types

SIL does not natively offer algebraic data types. However, the generality of domain types make it possible to emulate algebraic data types to a certain extent. Basically, an algebraic data type A has two ingredients. Firstly, there is a set of functions known as the data constructors. And secondly, the set defined by A is the smallest possible when repeatedly applying these data constructors.

In order to emulate A with a domain type D, we can introduce a domain function for every data constructor of A and supply axioms about their behaviour. What cannot be expressed in SIL is the second ingredient asking for the smallest set obeying the restrictions of D. Whenever a tool applies structural induction to D, it is responsible to ensure this guarantee, otherwise the transformation might not be sound. For instance, care must be taken with infinite structures.

For structural induction on a domain type D representing an algebraic data type, a number of domain functions of D are designated as the data constructors. In Silmore, this designation can either be done manually or automatically. If D should be automatically interpreted as an algebraic data type, then the longest prefix of domain functions of D with a codomain D is designated as the data constructors. For example, the domain type `Tree[A]` of Listing 2.2 on page 11 has the data constructors `leaf` and `node` by default.

3.2.2 Transformation of Structural Induction

Let us come back to the example about the associativity of the concatenation of lists. We wish to verify the property of Listing 3.4 on the preceding page for finite lists. For this purpose, we interpret the domain type `List[A]` of Listing 3.3 on page 20 as an algebraic data type. By default, the data constructors are given by the domain functions `nil` and `cons` as desired. Given that $P(l)$ again denotes the predicate

```
forall m: List[Any], r: List[Any] ::
  concatenate(concatenate(l, m), r) == concatenate(l, concatenate(m, r))
```

the result of applying structural induction is shown in Listing 3.6. It has the form of Equation (2.7) on page 15 apart from an additional universal quantification for the type of `nil` and a more local universal quantification of the prepended element. Silicon verifies this transformed property.

Listing 3.6: Transformation of Listing 3.4 on page 21 with data constructors `nil` and `cons` of `List[A]`, now verified by Silicon

```
method checkConcatenationAssociative()  // Verified
  ensures [forall l: List[Any] :: P(l),
    (forall t_1: Any :: P(nil(t_1))) &&
      forall l_1: List[Any] :: P(l_1) ==> forall e_1: Any :: P(cons(e_1, l_1))]
{ }
```

At the moment, there is no heuristic to select the induction variables for structural induction. If there is a choice between multiple variables, induction on each of them is attempted individually, connecting the resulting formulae by disjunctions. The verification of Listing 3.7 exemplifies this approach. The domain type `Natural` represents a common algebraic data type for the natural numbers. Its data constructors are the domain functions `zero` for the first natural number and `successor` to increment a natural number by one. Natural numbers can be added by `plus` and subtracted by `minus`. Only once transformed with structural induction into Listing 3.8, the property is verified by Silicon. Induction on the variable `n` can be verified, but not on the other variable `m` quantified over.

Listing 3.7: Property number 7 from [5] for a domain type `Natural` representing an algebraic data type for the natural numbers with the data constructors `zero` and `successor`

```
method property07()  // Not verified
  ensures forall n: Natural, m: Natural :: minus(plus(n, m), n) == m
{ }
```

Listing 3.8: Transformation of Listing 3.7 with induction tried on either variable `n` and `m` in turn, verified by Silicon

```
method property07()  // Verified
  ensures [forall n: Natural, m: Natural :: minus(plus(n, m), n) == m,
    // Structural induction on 'n': verified
    (forall m_3: Natural :: minus(plus(zero(), m_3), zero()) == m_3) &&
      (forall n_0: Natural ::
        (forall m_4: Natural :: minus(plus(n_0, m_4), n_0) == m_4) ==>
          forall m_5: Natural ::
            minus(plus(successor(n_0), m_5), successor(n_0)) == m_5)
    ||
    // Structural induction on 'm': not verified
    (forall n_1: Natural :: minus(plus(n_1, zero()), n_1) == zero()) &&
      (forall n_5: Natural ::
        (forall n_6: Natural :: minus(plus(n_6, n_5), n_6) == n_5) ==>
          (forall n_7: Natural ::
            minus(plus(n_7, successor(n_5)), n_7) == successor(n_5)))]
{ }
```

4 Termination of Functions

The second part of this thesis is about generating conditions for the termination of functions. The terminology of Section 2.2.3 on page 16 serves as a base.

4.1 Termination with Variants

First of all, let us prove that the existence of variants of functions ensures their termination. We begin with the termination of a single function and then generalise to multiple functions in two steps.

Lemma 1. *If the function f does not call other functions and there exists a variant of f, then the evaluation of each call to f terminates.*

Proof. Let v be a variant of f with a well-founded relation $\prec$. Furthermore, $f(a_0, ..., a_{n-1})$ is an arbitrary call to f with the evaluation tree T. In order to arrive at a contradiction, we assume that the call $f(a_0, ..., a_{n-1})$ does not terminate. This implies that T has an infinite path c_0, c_1, c_2, ... originating from its root. By the definition of a variant of a function, we have $v(c_{n+1}) \prec v(c_n)$ for each $n \in \mathbb{N}$. Thus, we can construct an infinite decreasing chain $v(c_0)$, $v(c_1)$, $v(c_2)$, ... with respect to $\prec$. However, this contradicts our assumption that $\prec$ is a well-founded relation. $\square$

Lemma 2. *Let F be a closed set of functions such that no $f \in F$ is indirectly recursive. If every $f \in F$ has a variant, then the evaluation of any function call to an $f \in F$ terminates.*

Proof. Let F be an arbitrary closed set of functions with variants and no indirectly recursive function. We prove termination by induction on the cardinality of F. Clearly, if F is empty, all functions terminate. Otherwise, F is not empty. As our induction hypothesis, we assume that all closed proper subsets of F solely have terminating functions.

Let us choose an arbitrary $f \in F$. Lemma 1 implies that the evaluation of any call to f either terminates or eventually calls another function. For each such call to another function $t \in F$, we define the set F' to be F without the functions reaching f in the call graph of F. Since there is no indirect recursion, the call to t never reaches any of the functions $F \setminus F'$ and can thus be evaluated with the functions F' alone. As F' is closed and $f \notin F'$, we conclude by virtue of the induction hypothesis that the call to t terminates. $\square$

Now we move to the general case of arbitrary recursion. The idea is that a strongly connected component of a call graph can essentially be translated into a single function which is not indirectly recursive and has a variant. Intuitively, variants justify the condensation of a call graph as an abstraction from the evaluation of its functions.

Lemma 3. *If each strongly connected component of a call graph with the functions F has a variant, then the evaluation of any function call to an $f \in F$ terminates.*

Proof. Let G be an arbitrary call graph with n strongly connected components G_0, ..., G_{n-1}. We translate the functions of every G_i into a single function with an analogous behaviour. We have any $G_i = (V, A)$ with k functions V and a variant v. If $k \geq 2$, we construct a function $f : I \to O$ as follows. Both I and O are algebraic data types with k data constructors, one for each function of V. f pattern matches on its input to evaluate the definition of the corresponding function in V. For the output, f applies the data constructor corresponding to the desired function. All function calls within G_i are replaced by direct recursion of f with essentially the same variant v. This way, we can reduce this case to the case of directly recursive functions of Lemma 2 on the preceding page. □

4.2 Variant Guessing and Generating Termination Conditions

How do we find variants? As stated by Definition 9 on page 17, the value of a variant must decrease along a well-founded relation for each function call within the same strongly connected component of a call graph. We cannot be sure that a mapping is indeed a variant until this condition is proved. Once we have a variant for every function of a closed set, Lemma 3 guarantees the termination of these functions.

Definition 10. A *variant guess* is a function which is conjectured to be a variant.

Silmore makes a variant guess for a function f as a mapping from the current arguments of a call to f to a tuple. Arguments of types which supply a mapping to integers can be part of the tuple, while the other arguments are omitted. The user can define which types are chosen by providing a mapping to integers. Besides arguments of type `Int`, domain types with an integer mapping can be chosen as documented in Section 3.1.3 on page 20.

The well-founded relation $\prec$ applied for the generated tuples has the same form as the one in Section 3.1.1 on page 18. For every call within the same strongly connected component, a condition is generated that the variant guesses applied to the respective arguments decrease along the well-founded relation $\prec$. The function calls checked for termination are only those part of the bodies of SIL functions. Function calls in specifications like preconditions and postconditions, on the other hand, do not matter for the evaluation of a function and are therefore not considered.

For an arbitrary function f, we always assume the partial correctness of f. For any checked function call of f, the path condition in the body of f leading to this call is assumed for the generated condition. Otherwise, well-formedness might be compromised. For instance, if a certain call performs a modulo operation, the divisor must not be zero, which may be given as part of the path condition. Optionally, the precondition of the callee instantiated with the current arguments may be assumed. Eventually, all the checks for f are collected in a single termination condition. A method with an empty body is created with this termination condition as the postcondition. This checking method has the same parameters and preconditions as f to establish the context of f.

Let us demonstrate this transformation with some examples. We would like to verify that the directly recursive function `fibonacci` of Listing 4.1 terminates. For this purpose, we transform the SIL program to generate the termination check of the method `check_fibonacci`. The variant guess is a tuple with the parameter `n` as its single element. If `n <= 1`, then there is no recursive call. Otherwise, there are two recursive calls with the variant guess instantiated to `n - 1` and `n - 2`, respectively. Both of these instances are checked to be smaller than `n` along the well-founded relation on integers. Furthermore, the precondition of `fibonacci` is assumed for these two recursive calls to show how this works, although Silicon verifies the check anyway.

Listing 4.1: Directly recursive function with its termination check, verified by Silicon

```
function fibonacci(n: Int): Int
  requires n >= 0
{ n <= 1 ? n : fibonacci(n - 1) + fibonacci(n - 2) }

// Check for termination of function 'fibonacci'.
// Variant guess: (n)
method check_fibonacci(n: Int)  // Verified
  requires n >= 0
  ensures n <= 1 ||
    (n - 1 >= 0 ==> n - 1 < n && 0 <= n) &&  // n - 1 < n
      (n - 2 >= 0 ==> n - 2 < n && 0 <= n)  // n - 2 < n
{ }
```

The next example is the Ackermann-Péter function of Listing 4.2. It is directly recursive and has two *Int* parameters, not both of which are made smaller in every recursive call. For this reason, the variant cannot simply choose a single argument. The variant guess made is the ordered pair of both arguments of a call, which is checked to decrease for each of the three recursive calls. This time, the well-founded relation for ordered pairs as in Equation (2.2) on page 14 is employed and the precondition is not assumed. Silicon successfully verifies this program.

Listing 4.2: Directly recursive function with an ordered pair as a variant, verified by Silicon

```
function ackermannPeter(l: Int, r: Int): Int
  requires l >= 0 && r >= 0
  ensures result >= 0
{ l == 0 ? r + 1 : r == 0 ? ackermannPeter(l - 1, 1) :
    ackermannPeter(l - 1, ackermannPeter(l, r - 1)) }

// Check for termination of function 'ackermannPeter'.
// Variant guess: (l, r)
method check_ackermannPeter(l: Int, r: Int)  // Verified
  requires l >= 0 && r >= 0
  ensures l == 0 || (r == 0 ?
    // (l - 1, 1) < (l, r)
    l - 1 < l && 0 <= l || l - 1 == l && (1 < r && 0 <= r) :
      // (l - 1, ackermannPeter(l, r - 1)) < (l, r)
      (l - 1 < l && 0 <= l || l - 1 == l &&
        (ackermannPeter(l, r - 1) < r && 0 <= r)) &&
          // (l, r - 1) < (l, r)
          (l < l && 0 <= l || l == l && (r - 1 < r && 0 <= r)))
{ }
```

The program of Listing 4.3 features the indirectly recursive functions `isEven` and `isOdd` to determine the parity of a natural number. By calling each other, they form a strongly connected component and give rise to two termination checks. Deriving a variant guess from the respective parameter `n`, the program is verified by Silicon.

Listing 4.3: Indirectly recursive functions with their termination checks, verified by Silicon

```
function isEven(n: Int): Bool
  requires n >= 0
{ n == 0 ? true : !isOdd(n - 1) }

function isOdd(n: Int): Bool
  requires n >= 0
{ n == 0 ? false : !isEven(n - 1) }

// Check for termination of function 'isEven'.
// Variant guess: (n)
method check_isEven(n: Int)  // Verified
  requires n >= 0
  ensures n == 0 || n - 1 < n && 0 <= n  // n - 1 < n
{ }

// Check for termination of function 'isOdd'.
// Variant guess: (n)
method check_isOdd(n: Int)  // Verified
  requires n >= 0
  ensures n == 0 || n - 1 < n && 0 <= n  // n - 1 < n
{ }
```

4.3 Folding Depth as Part of Variant Guesses

We now make use of unfolding expressions to enhance the completeness of the termination checking. Let `u` be an arbitrary unfolding expression

```
unfolding p in e
```

for expressions `p` and `e`. Under the assumption that the evaluation of `p` terminates, the evaluation of `u` always encounters a finite number $n \in \mathbb{N}$ of unfolding expressions, because the preceding program execution cannot fold infinitely many times. This is explained in Section 2.1.2 on page 12.

As the current folding depth n cannot be increased solely by evaluating a function, it does not hurt to always include n in a variant guess. Silmore implicitly takes n as the first tuple element of any variant guess. Since n is decremented by one in the expression `e`, all the function calls as part of `e` automatically have a decreased variant guess along the well-founded relation on tuples. Thus, it is enough to check the expression `p` for termination. The idea to prove the termination of functions by using the folding depth comes from the verification language Chalice [13].

Listing 4.4 exemplifies this approach. The program shows a linked list recursively built from references where the last cell assigns *`null`* to `next`. The predicate `valid` expresses access permissions to the reference to the tail `next` of the current linked list and its predicate `valid` unless it is the last cell. The directly recursive function `length` unfolds this predicate to calculate the size of a linked list. The only recursive call happens within the second part of the unfolding expression and hence has a smaller folding depth. When Silmore transforms this program to check termination, it does not generate a termination condition, since it recognises the unfolding expression around the recursive call.

Listing 4.4: Recursive function with an unfolding expression decreasing its variant

```
var next: Ref

predicate valid(list: Ref)
{ acc(list.next, write) &&
    (list.next != null ==> acc(list.next.valid(), write)) }

function length(list: Ref): Int
  requires acc(list.valid(), wildcard)
  ensures result >= 1
{ unfolding acc(list.valid(), wildcard) in
    list.next == null ? 1 : 1 + length(list.next) }
```

5 Library Guide

The following is a guide to the basic usage of the library Silmore. The programming language in use is Scala. Please refer to the readme file for more information such as installation instructions, usage as a standalone program and notes about the development like testing.

5.1 Basic Usage Example

Let us give a basic, yet complete example to introduce the usage of the library. Listing 5.1 shows how to parse a SIL program, create a program transformer and transform the program with it. Associating the program text with a filename, the parsing applies the SIL library to generate a program AST called `program`. For simplicity, we omit any error handling. Next, a well-founded relation `wellFounded` for integer tuples is created with domain types of the `program` mapped to integers if possible. The program transformer is then instantiated with this well-founded relation to check the functions of the `program` for termination. Functions which might not terminate are reported.

Listing 5.1: Complete example for the basic usage of the Silmore library

```scala
import java.nio.file.Paths
import semper.silmore.transformers.{
  TerminationTransformer,
  WellFoundedIntegerRelation
}
import semper.silmore.utility.ProgramProcessor

/* Parse program (associated with filename). No error handling here. */
val programText = "function a(): Bool { a() }"
val associatedFile = Paths.get("Program.sil")
val program = ProgramProcessor.parse(programText, associatedFile).right.get

/* Create termination checker with default well-founded relation. */
val wellFounded = WellFoundedIntegerRelation.defaultFromProgram(program)
val checker = new TerminationTransformer(wellFounded)

/* Add termination checks for functions of program. */
val (transformedProgram, uncheckedFunctions) = checker.addChecks(program)()

/* Report functions without generated termination check. */
println("No check for: " + uncheckedFunctions.map(_.name).mkString(" "))
```

5.2 Well-Founded Relations

Well-founded relations are of central importance for induction as well as termination of functions. The following hierarchy exists in the library:

```
trait WellFoundedRelation
trait WellFoundedTupleRelation extends WellFoundedRelation
class WellFoundedIntegerRelation(PartialFunction[Exp, Exp], Int)
  extends WellFoundedTupleRelation
```

The trait `WellFoundedRelation` essentially guarantees a function `isSmaller` to generate a SIL expression checking whether one element is smaller than another along the defined well-founded relation. This is extended by the trait `WellFoundedTupleRelation` to a well-founded relation on tuples. The class `WellFoundedIntegerRelation` implements a well-founded relation on tuples by mapping the elements to integers. For this purpose, the constructor takes a `PartialFunction[Exp, Exp]` which decides to map expressions satisfying certain conditions, for instance when they are of a domain type with an integer mapping. The `Int` defines the b in Equation (2.1) on page 14.

The companion object of `WellFoundedIntegerRelation` features some helpers. A collection of partial functions `PartialFunction[Exp, Exp]` is available to map expressions of type `Bool` and `Int` to integers. What is more, domain types can be mapped to integers, either implicitly by automatically trying to find integer mappings or explicitly by supplying a map `Map[Domain, Exp => Exp]` defining the mappings for a selection of domain types. For example, the following program defines two well-founded relations, both of which choose expressions of type `Int` and domain types of the `program` with an integer mapping. The well-founded relation `extendedWellFounded` additionally maps Boolean expressions to integers:

```
import semper.silmore.transformers.WellFoundedIntegerRelation

val program = …

val defaultWellFounded = WellFoundedIntegerRelation.defaultFromProgram(program)

val partialFunction = WellFoundedIntegerRelation.
  mapDomainsInteger(program.domains).
    orElse(WellFoundedIntegerRelation.mapBoolean)
val extendedWellFounded = new WellFoundedIntegerRelation(partialFunction)
```

5.3 Induction

The trait `InductionTransformer` is a common ancestor for induction. It offers the method

```
def transform(Program): Program
```

to transform the universal quantifications of the specifications of a program by induction. It does not specify the well-founded relation in use or how to select the induction variables.

Furthermore, the abstract method `expressionTransformer` generates a function to transform single universal quantifications instead of entire programs.

5.3.1 Induction on Tuples

The two classes

```
class DefaultInductionTransformer(WellFoundedRelation)
class ForcedInductionTransformer(WellFoundedRelation)
```

do the induction transformation on tuples as described in Section 3.1.1 on page 18. Both take a well-founded relation to instantiate the rule of Noetherian induction. The heuristic to select the induction variables by Leino [12] is integrated into the `DefaultInductionTransformer`. For example, it transforms Listing 3.1 on page 19 into Listing 3.2 on page 19. Forcing induction is the business of `ForcedInductionTransformer`. For instance, it transforms the property of Listing 3.4 on page 21 into Listing 3.5 on page 21.

5.3.2 Structural Induction on Domain Types

For structural induction, the class

```
class StructuralInductionTransformer(Map[Domain, AlgebraicDataType])
```

works as shown in Section 3.2 on page 22. It takes a map defining which domain types should be interpreted as which algebraic data types. With its help, the example of Listing 3.4 on page 21 is transformed into Listing 3.6 on page 23.

The companion object of the class `AlgebraicDataType` assists in constructing algebraic data types from domain types. To this end, it comes with the methods

```
def defaultDomainMapping(Seq[Domain]): Map[Domain, AlgebraicDataType]
def domainAlgebraicDataType(Seq[DomainFunc]): AlgebraicDataType
```

The method `defaultDomainMapping` automatically interprets the given domain types as algebraic data types. For every domain type, its longest prefix of domain functions with a codomain of the domain type itself is designated as the set of data constructors. If the data constructors of a domain type should be specified manually, then the other method `domainAlgebraicDataType` can construct an algebraic data type from domain functions. Care must be taken to make sure that such a structure inferred from a domain type really is an algebraic data type. Otherwise, the soundness of structural induction is endangered. More details can be found in Section 3.2.1 on page 22.

The following example shows how these modules work together:

```
import semper.silmore.transformers.{
  AlgebraicDataType,
  StructuralInductionTransformer
}

val program = …

val domainMapping = AlgebraicDataType.defaultDomainMapping(program.domains)
val transformer = new StructuralInductionTransformer(domainMapping)
val transformedProgram = transformer.transform(program)
```

5.4 Termination of Functions

For checking the termination of functions, the class

```
class TerminationTransformer(WellFoundedRelation, Boolean)
```

takes a well-founded relation to generate the termination conditions. Moreover, a `Boolean` says whether the preconditions of a called function should be assumed for the termination checks. For instance, Listing 4.1 on page 27 assumes preconditions, while Listings 4.2 and 4.3 on page 27 and on page 28 do not.

Termination can be checked by

```
def addChecks(Program)(Seq[Function]): (Program, Seq[Function])
```

The first argument is the program to which the termination checks are added as methods. The functions to be checked for termination are supplied as the second argument. The returned pair comprises the transformed program and the functions (from the ones given as a second argument) which still remain without a termination check. Unchecked functions occur if Silmore already knows that the variant guess is wrong in case the termination condition does obviously not hold. This is a chain-of-responsibility pattern for the functions to be checked. For unchecked functions, there are function calls for which the applied variant guesses do not decrease along the desired well-founded relation. These problematic calls are added in the form of a comment to the generated program.

6 Evaluation

As an ultimate examination, the technical work is now evaluated. We first concentrate on the question whether the transformations behave as specified. Afterwards, experiments about the formal verification of transformed programs are reported. The verifier in use is always Silicon by Schwerhoff [15]. The programs for all the experiments can be found as part of the Silmore test suite.

6.1 Test Suite

Silmore comes with a test suite of currently 121 own tests. Most of these tests basically have a pair of SIL programs, namely the original input program and the expected output program. The original program is transformed by a designated program transformer and the transformed program is compared to the expected program. Measuring the code coverage with a tool by Koponen [11], the tests achieve a statement coverage of about 97 % of the Silmore code.

In addition to these own tests, Silmore uses the programs given by the tests of the SIL library [4]. Silmore applies every program transformer to each of these programs. The purpose is to check whether no transformation crashes on various program constructs, ignoring the results of a transformation. At the moment, this gives rise to additional 57 tests per program transformer.

6.2 Induction

Bundy, Dixon, and Johansson [5] assembled 87 properties to evaluate their system IsaPlanner for inductive proofs. IsaPlanner can automatically verify 47 of these properties, and of these, Dafny by Leino [12] can verify 45. Of these 45 properties, we selected 38 properties for the experiments. Not all properties can be translated into SIL, since some involve higher-order functions, a construct not expressible in SIL. Whenever a type is arbitrary in a property, we employ the domain type `Any` with an empty body as in Listing 3.4 on page 21.

6.2.1 Automatically Verified Properties

Table 6.1 on the following page lists the properties verified by the corresponding tools. Without applying any induction transformation by Silmore, Silicon alone is able to verify 10 of the 38 properties. When applying the structural induction by Silmore, a total of 25 properties are verified automatically. No manual intervention in the verification like selecting the induction variables is done at all. For instance, property number 7 shown in Listing 3.7 on page 23 is transformed into Listing 3.7 on page 23 and then verified.

Table 6.1: Formal verification of 38 properties taken from [5]

Tool	Indices of the properties verified in addition to the preceding tool	Totally verified of all 38
Silicon alone	4, 5, 11, 13, 16, 28, 38, 39, 40, 42	10
Silicon with Silmore	2, 3, 6, 7, 8, 10, 15, 17, 18, 21, 26, 27, 29, 30, 37	25
Silicon with Silmore and manual CAT	1, 19, 22, 23, 24, 25, 31, 32, 33, 34	35
Dafny [12]	9	36
IsaPlanner [5]	20, 47	38

6.2.2 Cases Axiom Triggering (CAT)

Let us now see how more properties can be verified by a manual supplement. Property 1 is not automatically verified. Its original form is given by Listing 6.1 with the domain types `Natural` and `List` as seen earlier. The property states that for an arbitrary list, the concatenation of a prefix of any length and the list without this prefix equals the same list again. If $P(xs)$ is the predicate

```
forall n: Natural :: concatenate(take(n, xs), drop(n, xs)) == xs
```

then the transformed property is shown in Listing 6.2. For simplicity, we only consider induction on the variable `xs`.

Listing 6.1: Property number 1 from [5] for the domain types `Natural` and `List` representing algebraic data types, not verified by Silicon

```
method property01()  // Not verified
  ensures forall n: Natural, xs: List[Natural] ::
    concatenate(take(n, xs), drop(n, xs)) == xs
{ }
```

Listing 6.2: Transformation of Listing 6.1 with structural induction just on variable `xs` for simplicity, still not verified by Silicon

```
method property01()  // Not verified
  ensures [forall n: Natural, xs: List[Natural] ::  // forall xs :: P(xs)
    concatenate(take(n, xs), drop(n, xs)) == xs,
      (forall t_0: Natural :: P(nil(t_0))) && forall l_0: List[Natural] ::
        P(l_0) ==> forall e_0: Natural :: P(cons(e_0, l_0))]
{ }
```

Why can property 1 still not be verified? An axiom about every algebraic data type is that each of its elements is equal to an application of one of its data constructors for suitable arguments. For any domain type representing an algebraic data type in the experiments, an axiom for such a case distinction is already provided. However, the verifier seems not always to be able to trigger these axioms when needed. For instance, the axiom for a case distinction of the domain type `Natural` is

```
axiom zeroOrSuccessor {
  forall n: Natural :: n == zero() || exists p: Natural :: n == successor(p)
}
```

We call the process of reminding the verifier of the case distinction of an algebraic data type *cases axiom triggering*, abbreviated to CAT. An idea by Leino [12] is to integrate CAT into the rule of structural induction by replicating the case distinction. The kind of CAT we now do is to manually supply calls to domain functions which represent a case distinction.

For property 1, we extend the domain type `Natural` with the domain function and the axiom

```
function isZeroOrSuccessor(n: Natural): Bool

axiom isZeroOrSuccessorNatural {
  forall n: Natural :: isZeroOrSuccessor(n) ==
    (n == zero() || exists p: Natural :: n == successor(p))
}
```

such that calls to `isZeroOrSuccessor` trigger this axiom for the case distinction. In the same way, we add the pieces

```
function isNilOrCons(l: List[A]): Bool

axiom isNilOrConsList {
  forall l: List[A] :: isNilOrCons(l) == ((exists t: A :: l == nil(t)) ||
    exists e: A, r: List[A] :: l == cons(e, r))
}
```

to the domain type `List[A]`. Manual CAT is then done in Listing 6.3. The program reminds the verifier of the case distinctions, thereby indeed being verified by Silicon. There are three instances of manual CAT, none of which seems to be dispensable. The program fails to verify when removing the first or second manual CAT. If the third manual CAT is omitted, then the verification with Silicon does not terminate within half an hour.

Listing 6.3: Manual CAT for Listing 6.2 on the previous page at different places, now verified by Silicon

```
method property01()  // Verified
  ensures [forall n: Natural, xs: List[Natural] ::  // forall xs :: P(xs)
    concatenate(take(n, xs), drop(n, xs)) == xs,

      (forall t_0: Natural :: P(nil(t_0))) && forall l_0: List[Natural] ::
        isNilOrCons(l_0) ==>  // First manual CAT
          P(l_0) ==> forall e_0: Natural ::
            isZeroOrSuccessor(e_0) ==>  // Second manual CAT
              forall n_5: Natural ::
                isZeroOrSuccessor(n_5) ==>  // Third manual CAT
```

```
                  concatenate(take(n_5, cons(e_0, l_0)),
                    drop(n_5, cons(e_0, l_0))) == cons(e_0, l_0)]
{ }
```

The arguments of the calls for these manual CAT instances have different roles. In the first two calls, the arguments are variables introduced by the rule of structural induction, namely `l_0` and `e_0`. The third call for manual CAT happens inside the predicate of induction and takes the variable `n_5` of that predicate. This demonstrates that CAT may be irregular and therefore not easy to automate. By manual CAT, as many as 10 more properties can be verified.

6.2.3 Difficult Properties

Property 9 is verified by Dafny, but Silicon with Silmore and even manual CAT fails. As seen in Listing 6.4, this property has a universal quantification with three variables. Silmore tries induction on each of them, which results in the SMT solver Z3 running out of memory during the verification. When we manually select the variable `j` to do induction on and employ CAT, this property can be verified as well.

Listing 6.4: Property number 9 from [5] with three variables quantified over

```
method property09()  // Not verified
  ensures forall i: Natural, j: Natural, k: Natural ::
    minus(minus(i, j), k) == minus(i, plus(j, k))
{ }
```

As proposed by Leino [12], property 47 can be verified once property 23 is invoked by a method call. The remaining property 20 could not be verified.

6.3 Termination of Functions

To evaluate the generation of termination conditions, 8 programs which do not terminate and 9 other programs which do terminate were created. The programs were then transformed with Silmore for the termination checks and attempted to verify with Silicon.

6.3.1 Nonterminating Programs

Nonterminating programs were created by injecting mistakes into terminating programs to prevent their termination. Such mistakes can be forgotten preconditions, missing base cases in the definition of a function or recursive calls which do not decrease the variant guess along a well-founded relation. The program of Listing 6.5 contains a terminating function `factorial` and a pathological modification showing ways to prevent its termination. Of all 8 nonterminating programs, none are verified as desired.

Listing 6.5: Factorial function and a nonterminating modification

```
// Variant guess: (n)
function factorial(n: Int): Int
```

```
  requires n >= 0
{ n == 0 ? 1 : n * factorial(n - 1) }

// Variant guess: (n)
function f(n: Int): Int
{ n * f(n + 1) }
```

6.3.2 Terminating Programs

Of the 9 terminating programs, the termination of 7 programs can be verified automatically. One of the recursive functions with a termination check which is not verified is shown in Listing 6.6. Extending the domain type of Listing 2.2 on page 11, the type `Tree[Int]` represents a binary search tree with the integer keys in the leaves. In order to find the minimum of such a tree, the function `minimumKey` recursively visits the left subtree until a leaf is reached. This gives rise to a termination check, which applies the integer mapping `height` for the length of the longest path in a tree from the root to a leaf.

Listing 6.6: Terminating function with an integer mapping of a domain type as a variant guess, not verified by Silicon

```
function minimumKey(tree: Tree[Int]): Int
{ isLeaf(tree) ? key(tree) : minimumKey(leftChild(tree)) }

// Check for termination of function 'minimumKey'.
// Variant guess: (tree)
method check_minimumKey(tree: Tree[Int])  // Not verified
  ensures isLeaf(tree) ||
    height(leftChild(tree)) < height(tree) && 0 <= height(tree)
{ }
```

The termination condition can be stated as

```
forall t: Tree[Int] :: isLeaf(t) ||
  height(leftChild(t)) < height(t) && 0 <= height(t)
```

where we introduced a universal quantification. The idea is to apply structural induction. For this purpose, we treat `Tree[A]` as an algebraic data type and try to verify the transformed expression. Only when additionally employing manual CAT, Silicon verifies the property. This example demonstrates how a composition of the termination and induction transformations might prove useful.

The other program which terminates but is not verified automatically is Listing 6.7 on the following page. It calculates the greatest common divisor of two natural numbers (not both of which are zero) in a recursive manner. The variant guess is the tuple (l, r) of the two arguments l and r. However, this is not a variant. If $l < r$, the recursive call just exchanges the arguments and instantiates the variant guess with (r, l), which is not smaller than (l, r) along the well-founded relation in use. For a function call with arguments l and r, possible variants are simply r or the tuple (r, l). Our approach of automatically determining a variant guess reaches a limit.

Listing 6.7: Terminating function with a variant guess which is not a variant

```
// Variant guess: (l, r)
function gcd(l: Int, r: Int): Int
  requires l >= 0 && r >= 0
  requires l != 0 || r != 0
{ r == 0 ? l : gcd(r, l % r) }
```

7 Discussion

Let us now summarise the most important work done about automating induction for formal verification. Finally, we discuss how this thesis can be improved and what new questions could be explored.

7.1 Related Work

Bundy, Dixon, and Johansson [5] integrated an analysis of pattern-matching for inductively defined data types into a heuristic called rippling [1]. This was used to extend the system IsaPlanner to automatically prove theorems with case statements by induction. These authors collected 87 properties for the evaluation, of which their extension of IsaPlanner automatically proves 47.

Drossopoulou, Eisenbach, and Sonnex [6] developed the tool Zeno for the automatic verification of functional programs. It uses an alternative technique to rippling. Of the 87 properties from [5], Zeno can prove 84 theorems, while of the remaining 5 properties, 2 are false and only 3 left unproven.

The work by Leino [12] is closely related to this thesis and serves as a major reference. When translating the source language Dafny into the intermediate verification language Boogie, a simple heuristic determines whether to apply induction to a property and if so, on which variables. The program is then verified by an SMT solver, namely Z3 [2]. Compared to IsaPlanner and Zeno, this approach is straightforward and effective. Of the 47 properties proved by IsaPlanner [5], Dafny is able to automatically verify 45.

7.2 Future Work

The transformation for structural induction by Silmore benefits from manual CAT. Therefore, it would be desirable to automate CAT in a way that significantly enhances the completeness of verification and does not poison efficiency because of too many case distinctions. Furthermore, a heuristic to select induction variables for structural induction is missing. Trying induction on all variables individually may consume too much memory as seen for Listing 6.4 on page 37.

What concerns the termination of functions, the well-founded relation on algebraic data types of Equation (2.4) on page 15 is not employed. It might allow to automatically verify termination conditions for functions as in Listing 6.6 on page 38, since it seems more direct than the approach with mappings to integers. In addition, a heuristic to make a better variant guess for examples like Listing 6.7 on the preceding page would help. Similar to trying induction on each variable in turn, multiple variant guesses could be tried to see whether one of them verifies.

Based on the lessons learned during this thesis, SIL and the tools working with it like Silicon could be further strengthened. For instance, one could think about offering algebraic data types as a programming construct of SIL.

8 Conclusion

In conclusion, diverse forms of the induction principle and generation of termination conditions for functions have been enabled in SIL. The structural induction is not yet as complete as it could be, since case distinctions coming with algebraic data types are not always triggered automatically. Nevertheless, many interesting properties stating something about all instances of a structure can now be verified. By cooperating with the Silmore library, the completeness of Silicon was significantly improved, both in terms of partial correctness and termination.

The transformations are realised as independent SIL-to-SIL functions, allowing them to be applied by any SIL program verifier. Not only that, arbitrary translators of front-end languages for SIL can make use of the library, hopefully with more translators to come. This in turn should further motivate the work on SIL, thereby developing the automatic verification of concurrent programs in languages such as Scala.

Bibliography

[1] David Basin et al. *Rippling: Meta-Level Guidance for Mathematical Reasoning.* Vol. 56. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2005. ISBN: 978-0521834490.

[2] Nikolaj Bjørner and Leonardo de Moura. “Z3: An Efficient SMT Solver”. In: *Tools and Algorithms for the Construction and Analysis of Systems.* Ed. by C. R. Ramakrishnan and Jakob Rehof. Vol. 4963. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, 337–340. ISBN: 978-3-540-78799-0. DOI: `10.1007/978-3-540-78800-3_24`. URL: `http://research.microsoft.com/projects/z3/z3.pdf`.

[3] Bernhard Brodowsky. “Translating Scala to SIL”. Master thesis. ETH Zurich, 2013. URL: `http://www.pm.inf.ethz.ch/education/theses/student_docs/Bernhard_Brodowsky/MA_report`.

[4] Bernhard Brodowsky et al. *SIL*. Version 0.1-SNAPSHOT. Aug. 10, 2013. URL: `https://bitbucket.org/semperproject/sil`.

[5] Alan Bundy, Lucas Dixon, and Moa Johansson. “Case-Analysis for Rippling and Inductive Proof”. In: *Interactive Theorem Proving.* Ed. by Matt Kaufmann and Lawrence C. Paulson. Vol. 6172. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, 291–306. ISBN: 978-3-642-14051-8. DOI: `10.1007/978-3-642-14052-5_21`. URL: `http://dream.inf.ed.ac.uk/projects/isaplanner/papers/case-split-rippling-2010.pdf`.

[6] Sophia Drossopoulou, Susan Eisenbach, and William Sonnex. *Zeno: A tool for the automatic verification of algebraic properties of functional programs.* Tech. rep. Imperial College London, 2011. URL: `http://pubs.doc.ic.ac.uk/zeno/`.

[7] Stefan Heule. *Carbon.* 2013. URL: `https://bitbucket.org/stefanheule/carbon`.

[8] Winfried Just and Martin Weese. *Discovering Modern Set Theory. The Basics.* Vol. I. Graduate Studies in Mathematics 8. American Mathematical Society, 1996. ISBN: 978-0821802663.

[9] Ioannis T. Kassios, Peter Müller, and Malte Schwerhoff. “Comparing Verification Condition Generation with Symbolic Execution: an Experience Report”. In: *Proceedings of the 4th international conference on Verified Software: theories, tools, experiments.* Ed. by Rajeev Joshi, Peter Müller, and Andreas Podelski. 2012. URL: `http://pm.inf.ethz.ch/publications/getpdf.php?bibname=Own&id=KassiosMuellerSchwerhoff12.pdf`.

[10] Christian Klauser. “Translating Chalice into SIL”. Bachelor thesis. ETH Zurich, 2012. URL: `http://www.pm.inf.ethz.ch/education/theses/student_docs/Christian_Klauser/BA_Report_Christian_Klauser`.

[11] Mikko Koponen. *scct. Scala Code Coverage Tool.* Version 0.2-SNAPSHOT. Dec. 9, 2012. URL: `http://mtkopone.github.io/scct/`.

[12] K. Rustan M. Leino. "Automating Induction with an SMT Solver". In: *Proceedings of the 13th international conference on Verification, Model Checking, and Abstract Interpretation.* Ed. by Viktor Kuncak and Andrey Rybalchenko. 2012. URL: `http://research.microsoft.com/en-us/um/people/leino/papers/krml218.pdf`.

[13] K. Rustan M. Leino and Peter Müller. "A Basis for Verifying Multi-Threaded Programs". In: *European Symposium on Programming (ESOP).* Ed. by Giuseppe Castagna. Vol. 5502. Lecture Notes in Computer Science. Springer-Verlag, 2009, 378–393. URL: `http://pm.inf.ethz.ch/publications/getpdf.php?bibname=Own&id=LeinoMueller09.pdf`.

[14] K. Rustan M. Leino, Peter Müller, and Jan Smans. "Verification of Concurrent Programs with Chalice". In: *Foundations of Security Analysis and Design V.* Ed. by Alessandro Aldini, Gilles Barthe, and Roberto Gorrieri. Vol. 5705. Lecture Notes in Computer Science. Springer-Verlag, 2009, 195–222. URL: `http://pm.inf.ethz.ch/publications/getpdf.php?bibname=Own&id=LeinoMuellerSmans09.pdf`.

[15] Malte Schwerhoff. *Silicon.* Version 0.1-SNAPSHOT. Aug. 10, 2013. URL: `https://bitbucket.org/semperproject/silicon`.