JOHANNES KEPLER
UNIVERSITY LINZ

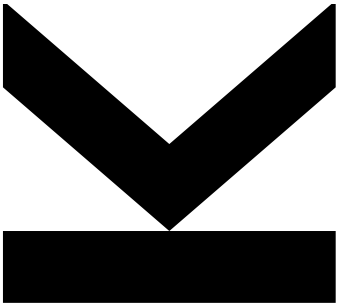
Submitted by
Franz-Xaver Reichl BSc

Submitted at
Research Institute for Symbolic Computation

Supervisor
A.Univ.-Prof. DI Dr. Wolfgang Schreiner

April 2020

The Integration of SMT Solvers into the RISCAL Model Checker



Master Thesis
to obtain the academic degree of
Diplom-Ingenieur
in the Master's Program
Computermathematik

JOHANNES KEPLER UNIVERSITY LINZ
Altenbergerstraße 69
4040 Linz, Österreich
www.jku.at
DVR 0093696

Abstract

In this thesis we present an alternative approach to check specifications from a substantial subset of the RISC Algorithm Language (RISCAL). The main goal for this new approach is to speed up checks done in RISCAL. For this purpose we developed a translation from the RISCAL language to the SMT-LIB language (using the QF_UFBV logic). The realisation of this translation in particular required to solve the problems of eliminating RISCAL's quantifiers and of encoding RISCAL's types and operations. We formally described core components of this translation, proved some aspects of their correctness, and implemented it in the programming language Java. We tested the implementation together with the SMT solvers Boolector, CVC4, Yices and Z3, on several real world RISCAL specifications. Additionally, we evaluated the performance of our approach by systematic benchmarks and compared it with that of the original RISCAL checking mechanism. Finally, we analysed the tests in order to determine patterns in specifications that could possibly have a negative influence on the performance of the presented method. The tests showed that among the four used SMT solvers, the solver Yices delivered, for almost all, tests the best results. Additionally, the tests indicated that the translation is indeed a viable alternative to RISCAL's current checking method, especially when used together with Yices. So the translation used with Yices was substantially faster than RISCAL in approximately 75% of the test cases. Nevertheless, the tests also illustrated that RISCAL could still check certain test cases substantially faster. Thus, the translation cannot fully replace RISCAL's current checking methods.

Zusammenfassung

In dieser Masterarbeit stellen wir eine alternative Vorgehensweise vor, um Spezifikationen eines wesentlichen Teils der RISC Algorithm Language (RISCAL) zu überprüfen. Das primäre Ziel dieser neuen Vorgehensweise ist es, die von RISCAL ausgeführten Überprüfungen zu beschleunigen. Zu diesem Zweck wurde eine Übersetzung der RISCAL Sprache in die SMT-LIB Sprache (unter Verwendung der QF_UFBV Logik) entwickelt. Für diese Übersetzung war es insbesondere notwendig, die Quantoren aus RISCAL zu entfernen und Typen und Operation aus RISCAL zu kodieren. Wesentliche Teile dieser Übersetzung wurden formal beschrieben, und für ausgewählte Komponenten davon die Korrektheit formal bewiesen. Des Weiteren wurde die Übersetzung in der Programmiersprache Java implementiert. Im Folgenden wurde diese Implementierung mit den SMT Beweisern Boolector, CVC4, Yices und Z3 getestet. Wir evaluierten die Geschwindigkeit unseres Ansatzes durch systematische Experimente und verglichen ihn mit der des ursprünglichen Überprüfungs-Mechanismus von RISCAL. Schlussendlich wurden die Tests analysiert, um Muster in den Spezifikationen festzustellen, die möglicherweise einen negativen Einfluss auf die Leistung der präsentierten Methode haben. Die Tests zeigten, dass von den vier benutzten Beweisern, Yices für fast alle Tests die besten Ergebnisse lieferte. Darüber hinaus zeigten die Tests, dass die Übersetzung, vor allem wenn sie gemeinsam mit Yices genutzt wird, eine brauchbare Alternative zu RISCALs derzeitiger Überprüfungsmethode darstellt. Dies kann daran festgestellt werden, dass die Übersetzung mit Yices in ca. 75% der Fälle wesentlich schneller war als RISCAL. Nichtsdestotrotz zeigen die Tests auch, dass für einige Tests RISCAL schneller war. Aus diesem Grund kann die Übersetzung die derzeitige Überprüfungsmethode nicht vollständig ersetzen.

Contents

1 Introduction

Over the past few decades computer programs became almost omnipresent, and today's life, as we know it, would not be possible without them. Whereas this development facilitated our daily life, this also caused a great dependency on the programs we use. This dependency implies a huge vulnerability of the whole society to software errors. Thus, more than ever, it is crucial that programs work flawlessly. Unfortunately traditionally applied techniques, like the testing of software, are not always enough to ensure this – as testing can only show the presence of errors and not their absence [17]. To overcome the restrictions of techniques like testing, a possible solution is to additionally make use of *Formal Verification* [33]. Formal Verification is a technique based on mathematics that shall ensure that systems like programs, work correctly (i.e. they shall work as they are supposed to do).

In order to apply formal verification, it is first of all necessary to generate a mathematical model of the system of interest (program/algorithm/...). We can then formally specify certain properties that this system shall have. The development of a system can already benefit from this kind of formalisation – as the necessary precise descriptions can reveal problems which would perhaps stay hidden in ambiguous system descriptions given in natural language. But this is not the only advantage of such a formalisation of a system. The formal nature of the model and of the properties allows to check whether the model actually satisfies the given properties or not. Obviously this means that we do not show that the actual system (e.g. a program) has the specified properties but that the model has. In order to perform this check we can apply the technique of *Model Checking* [10]. When applying model checking we examine all execution paths that are permitted by the model of the system. If we can find an execution that falsifies the specified properties, we know that the properties do not hold in the given model. Otherwise, we can conclude that they hold.

For the practical application of model checking, there are various different techniques/tools. In particular there are model checking tools that are based on SMT (satisfiability modulo theory) solvers [4]. SMT solvers are programs that can decide whether certain kinds of formulae from first order logic are satisfiable with respect to a theory — i.e. they decide whether there is an interpretation from the respective theory that makes the formula true. As the validity and the satisfiability of a formula are closely related — a formula is valid iff its negation is unsatisfiable — SMT solvers can also be used to determine whether a formula is valid with respect to a theory.

Today SMT solvers are tools of rising interest, for several areas of applications (including model checking) [14]. This development was supported by the SMT-LIB [2] initiative which developed a common input and output language for SMT solvers.

This language is used today by several of the major SMT solvers like Boolector [31], CVC4 [5], Yices [18] or Z3 [30].

In this thesis we will work with the RISC Algorithm Language (RISCAL) [34, 36]. RISCAL provides a specification language bundled with a software system. It aims to model algorithms — this means the focus lies not on concrete programs, but rather on more abstract ones. For this purpose RISCAL, on the one hand, provides the capability of describing algorithms and on the other hand, we can use RISCAL to specify these algorithms on the basis of first order logic. Moreover, RISCAL provides an extensive collection of built-in operations and data types (e.g. sets and maps). In RISCAL all data types are of finite size. This makes it possible for the RISCAL system to fully automatically analyse whether an algorithm satisfies its specification or not. For this purpose RISCAL applies model checking – RISCAL evaluates a given algorithm for every possible input [37].

While RISCAL provides a very rich language its checking mechanism is relatively basic, as RISCAL makes use of a brute force evaluation. As a consequence of this, checks performed by RISCAL can be quite time-consuming. Thus, the aim of this thesis is to elaborate an alternative checking mechanism for RISCAL that is, at least for certain cases, more efficient than the current one. For this purpose we want to make use of SMT solvers, where as an underlying logic the SMT-LIB logic of *Quantifier Free Formulae with Fixed Size Bit Vectors and Uninterpreted Functions* (QF_UFBV) shall be used[1].

This means that the RISCAL system shall be extended such that it supports the application of SMT solvers. The advantage of this procedure is that we can make use of the highly optimised decision procedures, provided by the SMT solvers instead of the rather basic checking provided by RISCAL. However, while the SMT solvers provide highly optimised decision procedures, their logics are quite limited compared to RISCAL. This makes the integration of SMT solvers in the RISCAL system a challenging task.

In order to achieve the goal of the thesis, we developed a translation for a major subset of the RISCAL language[2]. This idea of a translation is related to [29] where a translation of TLA$^+$ enables the application of SMT solvers. Our translation involves the formalisation of core components and an implementation in the programming language Java within the RISCAL system. To check a specification we translate the negation of the property that shall be checked and its dependencies to SMT-LIB. An SMT solver is then applied to this translation. If the solver reports that the

[1] Among other factors, the choice for a logic with fixed-size bit vectors, was motivated by the consideration that bit vectors are well-suited for the encoding of RISCAL's types and operations. Additionally, there is a good support for the bit vector logics by different SMT solvers. The usage of uninterpreted functions is necessary, as we need them for the encoding of certain components of the RISCAL language. In particular, we want to make use of the SMT solver Boolector that is being developed at JKU. As a substantial number of SMT solvers do not support quantified bit vectors (and also Boolector does not support quantified bit vectors with uninterpreted functions), we use the quantifier-free fragment of the logic.

[2] The only unsupported RISCAL concept is that of recursive types; also their translation would be possible but involves major additional efforts and was therefore not considered in this thesis.

translation is unsatisfiable, we know that the analysed specification holds.

The challenges for this translation were, as already mentioned earlier, the substantial differences between the RISCAL language and the SMT-LIB language. In particular the following problems had to be solved:

- Formulae in RISCAL may be quantified, whereas no quantifiers may occur in the translations.
- In RISCAL a nondeterministic choice can be used.
- Most importantly the available data types and operations substantially vary between RISCAL and SMT-LIB.

To solve these problems we made use of known techniques/encodings, like skolemisation for the elimination of existential quantifiers or the encoding of sets by bit vectors. Although we used known ideas as a basis for several encodings, the necessary adaptations were still nontrivial — for example: if we have two bit vectors that represent sets from the same universe; the union of these sets can be represented in a straight forward way by a bitwise or. But we will also see that the case has to be treated, where we have bit vectors that represent sets with different universes. In this instance the standard encoding is no longer applicable.

Finally, we tested the implementation of the translation. These tests illustrated that substantial speedups can be achieved by the methods developed in this thesis. But they also showed that there are still cases where the current checking mechanism of RISCAL is still preferable.

The remaining part of this thesis is structured in the following way: In Chapter 2 we will give a brief overview over related topics. In Chapter 3 we will present a *many sorted logic*, this logic will serve as a formal basis for the following chapters. In Chapter 4 we will discuss how to eliminate quantifiers from formulae given in the logic from the previous chapter. In Chapter 5 we will discuss core concepts of the encoding of RISCAL's data types and operations. In Chapter 6 we will combine the results from chapters 4 and 5. In Chapter 7 we will discuss some major differences between the theoretical model from the previous chapters and the implementation of the translation. In Chapter 8 we will examine practical tests of the implementation of the translation. In Chapter 9 we will sum up the results of the thesis and discuss possible further work that can be done on this topic.

The electronic form of this thesis is available at `https://www.risc.jku.at/research/formal/software/RISCAL/papers/thesis-Reichl.tgz`. Additional to the thesis itself this archive contains all the appendices for this thesis. The archive contains:

- The thesis in the form of a PDF.
- An adapted version of RISCAL for testing the translation.
- The scripts, for running the tests.

- The tested RISCAL specifications.
- The generated translations.
- An adapted version of the parallel command [35] that can be used to run the tests in parallel.

Work on this thesis was supported by the Johannes Kepler University, Linz, Institute of Technology(LIT) project LOGTECHEDU.

2 State of the Art

2.1 RISCAL

The RISC Algorithm Language (RISCAL) [36, 37] is a specification language that is equipped with a software environment. RISCAL is being developed by Wolfgang Schreiner at the RISC institute at the Johannes Kepler University Linz. The RISCAL software is implemented in Java and it is freely available at [34].

The aim of RISCAL is to provide support for the specification and verification of programs. By *programs* we actually do not mean concrete implementations, but rather conceptional procedures, i.e. algorithms. For this purpose, RISCAL provides a substantial collection of data types, like integers, arrays and sets, and operations, like the arithmetic operations for the integers or the union and the intersection for sets. As the focus of RISCAL lies on rather abstract procedures, the RISCAL language also contains components that can usually not be found in actual programming languages. RISCAL, for example, allows nondeterministic choices or implicitly defined functions. Moreover, the RISCAL language is capable of describing algorithms formally by means of commonly used programming instructions, like conditionals and loops. For stating properties, like in the specification of an algorithm, RISCAL builds upon first order logic.

The components outlined above can then be used to state certain mathematical theories, e.g. the theory for the greatest common divisor (gcd). This in general involves the definition of functions and predicates. Furthermore, we can state theorems in order to check that certain properties are fulfilled by the theory. Such theories can then be used for the specification of an algorithm. For example, if we have a theory for the gcd with a function representing the gcd, we can use this function in the specification of the Euclidean algorithm. In order to specify an algorithm, we can give pre- and postconditions. Additionally, we can also give annotations for algorithms, such as loop invariants or termination measures for loops.

In order to decide whether a given algorithm is correct or to check whether a theorem holds in a given theory, the RISCAL software system can be used. As RISCAL was developed with the idea of easy usability in mind, RISCAL performs these checks fully automatically. This would not be possible without any restrictions[1]. Hence, RISCAL restricts all variables (both in algorithms as in formulae) to types of

[1] Without any restrictions, checking the validity of formulae in RISCAL would be undecidable. This can be justified by the following consideration: Assume that we do not apply any restrictions and assume that checking the validity of formulae in RISCAL is decidable. Then we could use RISCAL to decide whether an diophantine equation has a solution. But this Problem is undecidable [28].

finite size. This, for example means that in RISCAL we cannot define the gcd for all possible natural number arguments (where at least one is not zero). Instead, we have to define it for integer intervals. This restriction allows RISCAL to check all given algorithms/theorems by means of an evaluation, for all suitable variable assignments (model checking). In conclusion this means that with RISCAL we actually can not necessarily validate the correctness of an algorithm. Still, RISCAL gives valuable results – e.g. if RISCAL reports that an algorithm is not correct, we can avoid starting a correctness proof that is doomed to fail.

Additionally, RISCAL is capable of automatically generating *verification conditions* (e.g. the condition that the termination measure for a loop is actually decreased). As these conditions are formulae, like user defined theorems, RISCAL can also automatically analyse them.

To finish this brief section about RISCAL, we will give in Listing 2.1 a short sample RISCAL-specification, which shall illustrate the RISCAL language. The given sample specification is a slightly modified and shortened version of the gcd specification which is shipped with the RISCAL system [34].

In the beginning of Listing 2.1 we can see how we are able to set the size of types. By the use of $\mathbb{N}[N]$ we restrict the natural numbers to numbers smaller-equal than N. We can set the value of N in the RISCAL system. Subsequently, this specification shows how RISCAL can be used to specify theories. Here we first define the gcd of two natural numbers. Then we want to prove certain well-known properties about the gcd. Last but not least, we can also see how to use RISCAL to describe and specify algorithms. In the example above the euclidean algorithm is given.

2.2 SMT

In logic, the *satisfiability* of a formula means that there is some interpretation that makes the formula true. The problem of *Satisfiability Modulo Theories* (SMT) is strongly related to the problem of satisfiability. The difference is that, while we are considering arbitrary interpretations in the case of satisfiability, we are only considering specific interpretations in the case of SMT. For this purpose, we are considering *Theories* – i.e. classes (in the set-theoretical sense) of interpretations. We now only consider a first order logic formula as satisfiable modulo a theory, iff there is an interpretation from the theory that satisfies the formula. For practical applications (additional to the semantic restriction discussed above) syntactic restrictions are also of interest. This, for example means that we may only consider formulae without any quantifiers [4].

There is a substantial number of different theories that are actually used in practice ([2] gives several examples of such theories). To give a better picture of SMT, we will briefly discuss two theories, which are of particular interest:

- Our first example is the theory of *Linear Integer Arithmetic*. As the name already suggests, we fix the domain of integers with its usual meaning. Additionally, functions, like addition and subtraction and predicates like equality and

Listing 2.1: RISCAL example

```
val N: ℕ;
type nat = ℕ[N];

pred divides(m:nat,n:nat) ⇔ ∃p:nat. m·p = n;

fun gcd(m:nat,n:nat): nat
  requires m ≠ 0 ∨ n ≠ 0;
= choose result:nat with
    divides(result,m) ∧ divides(result,n) ∧
    ¬∃r:nat. divides(r,m) ∧ divides(r,n) ∧ r > result;

theorem gcd0(m:nat) ⇔ m≠0 ⇒ gcd(m,0) = m;
theorem gcd1(m:nat,n:nat) ⇔ m ≠ 0 ∨ n ≠ 0
⇒ gcd(m,n) = gcd(n,m);
theorem gcd2(m:nat,n:nat) ⇔ 1 ≤ n ∧ n ≤ m
⇒ gcd(m,n) = gcd(m%n,n);

proc gcdp(m:nat,n:nat): nat
  requires m≠0 ∨ n≠0;
  ensures result = gcd(m,n);
{
  var a:nat := m;
  var b:nat := n;
  while a > 0 ∧ b > 0 do
    invariant a ≠ 0 ∨ b ≠ 0;
    invariant gcd(m,n)=gcd(a,b);
    decreases a+b;
  {
    if a > b then
      a := a%b;
    else
      b := b%a;
  }
  return if a = 0 then b else a;
}
```

the different inequalities, get their expected semantic meaning. As mentioned before, we could also restrict the considered formulae to formulae without quantifiers. This fragment of Linear Integer Arithmetic is called *Quantifier Free Linear Integer Arithmetic*.

- Our second example is the theory of *Fixed Size Bit Vectors*. In contrast to the previous example, here we fix several domains. So we have, for each positive integer n, a domain for bit vectors of length n. Again, several operations are defined for these domains. This includes addition, subtraction and multiplication (these operations correspond to the arithmetic operations in usual programming languages; thus for example overflows have to be considered). Moreover, bitwise shifts and extractions of bits are supported.

We now want to give a basic example that illustrates the difference of *satisfiability* and *satisfiability modulo a theory* of a formula.

$$2 + 2 < 4$$

In the usual mathematical understanding this formula obviously does not hold. Still, the formula is satisfiable – if the $<$ symbol is interpreted with the $\geq$ predicate and the other symbols get their usual meaning, the formula gets true. If we now instead consider the theory of integers, the formula is unsatisfiable modulo this theory – as all symbols get the expected meaning.

The above example already gives a reason why satisfiability modulo theories can be of interest. So we are sometimes only interested in interpretations that assign a specific meaning to certain symbols. Additionally, proving that a first order logic formula is satisfiable (respectively valid) is generally not a simple task. It is challenging, as proving the satisfiability of a formula is *undecidable* and proving the validity of a formula is only *semi-decidable*. Here, *undecidable* means that there can be no procedure that determines, whether an arbitrary first order logic formula is satisfiable or not; *semi-decidable* means that there can be a procedure that detects the validity of arbitrary valid formulae, but this procedure may not terminate for invalid formulae [8]. In contrast to this, for several theories, the SMT problem is *decidable* — thus there is a procedure that can determine, for an arbitrary formula, whether it is satisfiable or not. Therefore, validity modulo such theories is also decidable. Examples for decidable theories are the theories of fixed size bit vectors or the theory of integers (and thus also the quantifier free fragments of these theories) [23]. But not all theories that would be of interest are decidable. Thus, for example the theory of *Peano Arithmetic* (that represents the natural numbers with the Peano Axioms) is undecidable [8]. For other theories, restrictions have to be applied, in order to make the checking of satisfiability decidable. For example, the theory of arrays is undecidable, but its quantifier free fragment is decidable [8].

To finish this brief section on SMT, we want to point out that currently, SMT is a topic of high interest. Hence, there are various different SMT solvers for several different theories – for example Boolector [31], Z3 [30], Yices [18] or CVC4 [5].

Additionally, such solvers are used for different practical applications, like for example software verification (as can be seen in the OpenJML tool which makes use of SMT solvers [12]).

2.3 SMT-LIB

The Satisfiability Modulo Theory Library (SMT-LIB) initiative was established in 2002 by Silvio Ranise and Cesare Tinelli. It is aimed at facilitating research and development in the field of SMT [2, 3]. To pursue this aim, the SMT-Lib initiative has set up three goals. These goals are, to a large extent, achieved today:

- A standardised description of theories used by SMT solvers. Today, a large variety of such theories, respectively fragments of these theories, is supported by SMT-LIB, like the already mentioned linear integer arithmetic and the theory of fixed size bit vectors and several more.
- A standardised input and output language for SMT solvers. Currently, several solvers accept the SMT-LIB language, like: Boolector, CVC4, Yices and Z3. Below two short examples for this language will be given.
- A large library of benchmarks for SMT solvers.

Closely related to SMT-LIB is the Satisfiability Modulo Theories Competition (SMT-COMP). SMT-COMP is an annual competition of SMT solvers that has been taking place since 2005 [39]. The large number of solvers that support SMT-LIB and the large number of participants at the SMT-COMP indicate the good acceptance of SMT-LIB.

Finally, we will give two sample solver inputs in the SMT-LIB language. In the first example we will make use of the SMT-LIB quantifier-free linear integer arithmetic logic:

```
(declare-fun x () Int)
(declare-fun y () Int)
(declare-fun z () Int)
(assert (<= x y))
(assert (<= y z))
(assert (not (<= x z)))
(check-sat)
```

This example can be interpreted in the following way: We declare three functions x, y, z with no parameters, i.e. constant. As these functions are uninterpreted, they can have an arbitrary value. We want to show transitivity of the less equal relation. To do this, we assume that $x \leq y$ and $y \leq z$. We do this by asserting this property. As we use an SMT solver, we can only determine, if a formula is satisfiable, respectively unsatisfiable. Therefore, we have to consider the negation of the assertion we want to prove. If we apply this example to an SMT solver that supports the logic, we will get

as a result that our input is unsatisfiable. Therefore, the desired result is proven. In our second example we make use of the logic of quantifier-free fixed size bit vectors:

```
(declare-fun x() (_ BitVec 4))
(define-fun y() (_ BitVec 4)#b0001)
(assert (not (bvule x (bvadd x y))))
(check-sat)
```

In this example we want to show that by adding the bit vector 0001, which represents the number 1, to some bit vector of length 4, the sum will be greater than the original bit vector. Similarly, as before, we have to consider the negation of the assertion we want to show. Of course a SMT solver will tell us that this input is satisfiable and thus that the assertion we want to show is not valid (if we add 1111 and 0001 we get 0000 which is less than 1111).

2.4 SMT Solvers

Today, there is a large selection of different SMT solvers for different theories. For this thesis we decided to use the the following SMT solvers:

Boolector [31] Boolector is an SMT solver which is being developed by Prof. Armin Biere at the Institute for Formal Models and Verification at JKU and Aina Niemetz and Mathias Preiner at Stanford University. Boolector is an SMT solver dedicated to fixed-size bit vectors – it supports the SMT-LIB theory of Bit Vectors (BV) and also the quantifier-free fragments of BV with/without arrays and/or uninterpreted functions. Boolector supports different input languages including SMT-LIB version 2 and its own language BTOR. Furthermore, it provides APIs for the programming languages C and Python [7]. Boolector won the second place of the QF_UFBV (Single Query Track) at SMT-COMP 2019 [39]. Boolector is freely available at [7].

CVC4 [5] CVC4 is an SMT solver which is being developed under the lead of Clark Barrett at Stanford University and Cesare Tinelli at University of Iowa. The solver supports multiple logics, including the logic QF_UFBV. Additionally, CVC4 supports different input languages including SMT-LIB 2. CVC4 provides APIs for the programming languages C++ and Java [13]. CVC4 is freely available at [13].

Yices [18] Yices is an SMT solver which is being developed at SRI International's Computer Science Laboratory. Yices supports several logics, including for example the SMT-LIB logics QF_UFBV or QF_LIA. Yices supports, both SMT-LIB version 1 and 2 and an input language specific to Yices. Yices provides a C API and bindings for Python Go and OCaml [42]. The SMT solver won the QF_UFBV (Single Query Track) at SMT-COMP 2019 [39]. Yices is freely available at [42].

Z3 [30] Z3 is an SMT solver which has been developed by Nikolaj Bjørner and others at Microsoft Research. The solver supports various logics, including QF_UFBV . Z3 supports different input languages, including SMT-LIB. It moreover provides APIs for seeral programming languages including C++, Java and Python. Z3 is freely available at [43].

2.5 Application of SMT Solvers

The approach to integrate SMT solvers into programs associated to a specification language is by no means a new one. Actually, this thesis is also motivated by the fact that this is already done. In this section we will give some examples of specification languages where SMT solvers were integrated into the associated software. Note that these specification languages (significantly) differ from RISCAL. Additionally, we will briefly discuss the role of SMT solvers, respectively of SAT (satisfiability) solvers in *Bounded Model Checking* [11, 6].

JML [26] The Java Modelling Language (JML) is a specification language which allows to specify Java programs by annotating the Java source files. Specifications in JML can be checked with the OpenJML tool [12]. In order to do this, OpenJML translates the JML specifications into SMT-LIB and then applies an SMT solver, such as Z3 or Yices, for checking.

TLA$^+$ [25] TLA$^+$ is a specification language based on temporal linear logic, which is especially used to model concurrent systems. In [29] and [40] a transformation of TLA$^+$ specifications into SMT-LIB is presented. With the help of this transformation, the usage of an SMT solver in the TLA$^+$ proof systems gets possible.

Event-B [1] Event-B is a specification language that is characterised by refinements of models i.e. refining abstract models to more concrete models. The papers [15] and [16] present a translation of Event-B proof obligations into an SMT-LIB language.

Alloy [22] Last but not least, Alloy is a specification language based on a relational logic. In [20] it is shown that SMT solvers can be applied to Alloy specifications. Again this is done with a translation to SMT-LIB. Also in [19] SMT solvers are applied, here the solver Yices is used. Unlike to the previous example, a Lisp-like language, that is accepted by Yices and not the SMT-LIB language is used for a translation.

Last but not least we will briefly discuss bounded model checking. As we already discussed in the context of model checking, also in bounded model checking we analyse execution paths of a model in order to determine whether the model has some property. But unlike to before, the lengths of the considered execution paths are bounded in bounded model checking. In order to analyse the execution, a formula

is generated from the bounded execution paths and the property of interest. This formula shall be satisfiable iff there is an execution path of the given length that violates the property of interest. In order to check the satisfiability of such formulae there are on the one hand techniques based on SAT solving [11, 6] and on the other hand techniques based on SMT solving [4].

We want to point out that the way we apply SMT solvers in this thesis differs from the application of SMT, respectively of SAT solvers, for bounded model checking. We use RISCAL in order to generate verification conditions. These conditions are translated to SMT-LIB and then they are analysed by means of an SMT solver. Thus, unlike to bounded model checking we do not apply a certain bound on the length of execution paths.

3 The Logical Framework

In this chapter we will build up a logical framework, which we will use as a basis for the modelling of the translation from the RISCAL language to the SMT-LIB language. To achieve this we will present a many-sorted first order logic and several notions for this logic. Last but not least we will prove certain properties for this framework.

3.1 Many-Sorted First Order Logic

In this section a many-sorted version of first order logic is defined.

The logic presented in this section is based to a large extent on the many-sorted logic presented in [27]. The only major difference is that in contrast to the original version we do not require an untyped equality in the logic. Both RISCAL and SMT-LIB do not support such an equality [36, 2]. As the purpose of the logic we use here is to model a translation from RISCAL to SMT-LIB, we also do not need an untyped equality in the logic.

3.1.1 Syntax

First we define the language $\mathcal{L}$ of well-formed expressions from the many sorted logic. We start with defining the signature of the logic.

Definition 1. Let $C \coloneqq \{\neg_{\mathcal{L}}, \wedge_{\mathcal{L}}, \vee_{\mathcal{L}}, \Rightarrow_{\mathcal{L}}, \Leftrightarrow_{\mathcal{L}}\}$. We denote the elements of C as *connectives*. Moreover let *OpSyms* be a set of symbols such that $C \subseteq \mathit{OpSyms}$. We denote these symbols as *operation symbols*. Then we define a *signature* Σ as a pair $\Sigma = (\mathit{Sort}, \mathit{Func})$ such that:

- *Sort* is a set of natural numbers such that $0 \in \mathit{Sort}$. We denote the elements of this set as the *sorts* of the signature.
- *Func* is a function that maps operation symbols to finite tuples of sorts.

  $$\mathit{Func} : \mathit{OpSyms} \to \bigcup_{n \in \mathbb{N}^*} \mathit{Sort}^n$$

  We denote the tuple that is assigned to an operation symbol as its *arity*.
- We require that the connectives have the following arities:

  $$\mathit{Func}(\neg_{\mathcal{L}}) = (0,0) \quad \mathit{Func}(\wedge_{\mathcal{L}}) = (0,0,0) \quad \mathit{Func}(\vee_{\mathcal{L}}) = (0,0,0)$$
  $$\mathit{Func}(\Rightarrow_{\mathcal{L}}) = (0,0,0) \quad \mathit{Func}(\Leftrightarrow_{\mathcal{L}}) = (0,0,0)$$

- Let $o \in OpSyms \setminus C$ and $n \in \mathbb{N}$ such that $Func(o) \in Sort^n$. Then we require that for all $i \in \mathbb{N}^*$, with $i \neq 1$ and $i \leq n$ that:

$$Func(o)_i \neq 0.$$ •

Remark In the following let Σ be a signature.

- To refer to the components of Σ, we write $\Sigma.Sort$ and $\Sigma.Func$.
- We denote the signature's set of operation symbols as Op_Σ.

Definition 2. Let Σ and Ξ be signatures. We say Σ is *contained* in Ξ, written as $\Sigma \subseteq \Xi$, iff the following conditions hold:

- $\Sigma.Sort \subseteq \Xi.Sort$
- $Op_\Sigma \subseteq Op_\Xi$
- $\forall f \in Op_\Sigma : \Sigma.Func(f) = \Xi.Func(f)$ •

Next we can define the set of expressions $\mathcal{L}$, over a signature Σ. In the following we assume that for each $i \in \Sigma.Sort \setminus \{0\}$ a set $\mathcal{V}_i$, containing variables of sort i, is defined. These sets shall be pairwise disjoint. For simplicity, we assume that the elements of these sets are annotated with the sort of the variables e.g. v^i for an element of $\mathcal{V}_i$. Furthermore $\mathcal{V}$ shall denote the set of all variables, i.e., $\mathcal{V} = \bigcup_{i \in \Sigma.Sort \setminus \{0\}} \mathcal{V}_i$.

Definition 3. Let Σ be a signature. Then for $i \in \Sigma.Sort$ the set of *expressions of sort i over the signature* Σ (denoted as Exp_i^Σ) is defined as:

- Let $i \neq o$ and let $v^i \in \mathcal{V}_i$. Then $v^i \in Exp_i^\Sigma$.
- Let $f \in OpSyms, n \in \mathbb{N}, \Sigma.Func(f) = (i, i_1, \ldots, i_n)$ and $e_1 \in Exp_{i_1}^\Sigma, \ldots, e_n \in Exp_{i_n}^\Sigma$. Then $f(e_1, \ldots, e_n) \in Exp_i^\Sigma$.
- Let $e \in Exp_0$, $j \in \Sigma.Sort \setminus \{0\}$ and $v^j \in \mathcal{V}_j$. Then $\exists_\mathcal{L} v^j e \in Exp_0^\Sigma$ and $\forall_\mathcal{L} v^j e \in Exp_0^\Sigma$.
- There are no other elements of Exp_i^Σ.

We denote by $\mathcal{L}^\Sigma$ the set of all expressions, i.e., $\mathcal{L}^\Sigma = \bigcup_{i \in \Sigma.Sort} Exp_i^\Sigma$. If the used signature is clear from the context, respectively if the used signature is not of importance, we will skip the super script and denote the set of expressions by $\mathcal{L}$. •

Remark

- For an expression e, we denote its sort with $sort(e)$.
- Let Σ be a signature and f be an operation symbol from Op_Σ. Moreover let $n \in \mathbb{N}$ such that $\Sigma.Func(f) \in (\Sigma.Sort)^{n+1}$. Then we define the number of arguments of f, $args(\Sigma, f)$, as:

$$args(\Sigma, f) \coloneqq n$$

- Let Σ be a signature and f be an operation symbol from Op_Σ. Moreover let $0 \leq i \leq args(\Sigma, f)$. Then we define $arg(\Sigma, f, i)$ as the sort of $\Sigma.Func(f)_{i+1}$. For $i = 0$ this is the sort of the function itself and for $i \neq 0$ it is the sort of the i^{th} function argument.
- Let $s \in \Sigma.Sort$. If $s \neq 0$, then we denote s as a *non-boolean sort*.
- We call every $e \in Exp_0$ a *formula* and every $e \in \mathcal{L} \setminus Exp_0$ a *term*. Later, when we deal with the semantic of this logic, we will see that formulae are associated to boolean values and terms to non-boolean values.
- We call every $f \in Op_\Sigma$ with $\Sigma.Func(f) \in \Sigma.Sort$ a *constant symbol*.
- We call every $f \in Op_\Sigma$ with $\Sigma.Func(f)_1 \neq 0$ a *function symbol*.
- We call every $f \in Op_\Sigma$ with $f \notin \{\neg_\mathcal{L}, \wedge_\mathcal{L}, \vee_\mathcal{L}, \Rightarrow_\mathcal{L}, \Leftrightarrow_\mathcal{L}\}$ and $Func(f)_1 = 0$ a *predicate symbol*.
- For the connectives we use infix notation instead of prefix notation.
- Unlike to the usual presentation of first order logic, here the logical connectives are also operation symbols.

3.1.2 Semantics

Now we can proceed with giving a semantics to the language defined above. Firstly, we introduce structures and variable assignments, in order to define interpretations of expressions from the language $\mathcal{L}$.

Definition 4. Let Σ be a signature. Then we define a *structure* $\mathcal{A}$ over the signature Σ as a pair $\mathcal{A} = ((\mathcal{A}_i)_{i \in \Sigma.Sort}, (f^\mathcal{A})_{f \in OpSyms})$ such that:

- $(\mathcal{A}_i)_{i \in \Sigma.Sort}$ is a family of non-empty sets where $\mathcal{A}_0 = \{\mathbb{T}, \mathbb{F}\}$. We call $\mathcal{A}_i$ the *domain* or the *universe* of sort i.
- $(f^\mathcal{A})_{f \in OpSyms}$ is a family of functions such that for each operation symbol f we have:

$$\text{If } args(\Sigma, f) = 0 \text{ then } f^\mathcal{A} \in \mathcal{A}_{arg(\Sigma,f,0)},$$
$$\text{else } f^\mathcal{A} : \bigtimes_{i=1}^{args(\Sigma,f)} \mathcal{A}_{arg(\Sigma,f,i)} \to \mathcal{A}_{arg(\Sigma,f,0)}.$$

- The connectives shall get a fixed meaning.

$$\begin{array}{lll} \neg_\mathcal{L}^\mathcal{A} : \mathcal{A}_0 \to \mathcal{A}_0 & \wedge_\mathcal{L}^\mathcal{A} : \mathcal{A}_0 \times \mathcal{A}_0 \to \mathcal{A}_0 & \vee_\mathcal{L}^\mathcal{A} : \mathcal{A}_0 \times \mathcal{A}_0 \to \mathcal{A}_0 \\ \mathbb{T} \mapsto \mathbb{F} & (\mathbb{T}, \mathbb{T}) \mapsto \mathbb{T} & (\mathbb{T}, \mathbb{T}) \mapsto \mathbb{T} \\ \mathbb{F} \mapsto \mathbb{T} & (\mathbb{T}, \mathbb{F}) \mapsto \mathbb{F} & (\mathbb{T}, \mathbb{F}) \mapsto \mathbb{T} \\ & (\mathbb{F}, \mathbb{T}) \mapsto \mathbb{F} & (\mathbb{F}, \mathbb{T}) \mapsto \mathbb{T} \\ & (\mathbb{F}, \mathbb{F}) \mapsto \mathbb{F} & (\mathbb{F}, \mathbb{F}) \mapsto \mathbb{F} \end{array}$$

The interpretation of $\Rightarrow_{\mathcal{L}}$ and $\Leftrightarrow_{\mathcal{L}}$ is defined similarly. •

Remark Let Σ be a signature, $i \in \Sigma.Sort$ and $\mathcal{A}$ a structure over Σ. Then we denote the universe of sort i with respect to $\mathcal{A}$, by $\mathcal{A}_i$.

Remark We use the following notation for altering functions. Let f be a function and A, B two sets such that $f : A \to B$. If $a \in A, b \in B$, then $f[a \mapsto b]$ denotes a function such that:

- $\forall x \in A : x \neq a \Rightarrow f[a \mapsto b](x) = f(x)$
- $f[a \mapsto b](a) = b$

Definition 5. A variable assignment is a function M which maps variables of sort i to the universe of sort i, i.e.:

$$M : \bigcup_{i \in \Sigma.Sort} \mathcal{V}_i \to \bigcup_{i \in \Sigma.Sort} \mathcal{A}_i \qquad \text{s.t.} \qquad M(\mathcal{V}_i) \subseteq \mathcal{A}_i \quad \bullet$$

With these prerequisites we can define the notion of interpretations, i.e., we can give the semantics of the language $\mathcal{L}$.

Definition 6. Let Σ be a signature. We define an *interpretation* as a pair $\langle \mathcal{A}, M \rangle$ where $\mathcal{A}$ is a structure over Σ and M is a variable assignment.

We now define the application of an interpretation I to expressions from $\mathcal{L}^{\Sigma}$. Subsequently let $f \in Op_{\Sigma}$, $n \in \mathbb{N}$ and $e_1, \cdots, e_n$ be expressions from $\mathcal{L}^{\Sigma}$ such that $e_1, \cdots, e_n$ fit to the arity of f.

- $I(x^i) = M(x^i)$
- $I(f(e_1, \ldots, e_n)) = f^{\mathcal{A}}(I(e_1), \ldots, I(e_n))$
- $I(\exists_{\mathcal{L}} x^i e) = \mathbb{T}$ iff $\{a \in \mathcal{A}_i \mid \langle \mathcal{A}, M[x^i \mapsto a] \rangle(e) = \mathbb{T}\} \neq \emptyset$
- $I(\forall_{\mathcal{L}} x^i e) = \mathbb{T}$ iff $\{a \in \mathcal{A}_i \mid \langle \mathcal{A}, M[x^i \mapsto a] \rangle(e) = \mathbb{T}\} = \mathcal{A}_i$ •

Remark We want to point out that the way the logic is defined, implies that:

- The only operations that may take formulae as arguments are the connectives.
- Quantifications with variables over the boolean values are not allowed.

3.2 Conventions for the Logic

In this section we will define several functions and introduce some notions on the logic described in the previous section.

To start, we introduce some conventions for the names of meta-variables. In the following let Σ be some fixed signature. For the subsequent conventions the underlying signature shall be Σ.

- F, F_1, F_2 shall denote formulae.
- t, t_1, t_2 shall denote terms.
- $e, e_1, \cdots, e_n$ shall denote expressions.
- x^i, x^j, y^i, y^j shall denote variables.
- f shall denote a function symbol and P a predicate symbol.
- o shall denote an arbitrary operation symbol, ρ an operation symbol that is not a connective and c shall denote a connective.
- $\mathcal{A}$ and $\mathcal{B}$ shall denote structures.
- M shall denote a variable assignment.
- I shall denote an interpretation.

Initially we will give some definitions which will enable us to retrieve certain syntactic aspects of formulae. First of all we will define a function which retrieves the free variables of a formula.

Definition 7. The set of *free variables* of an expression e, $\mathit{free}(e)$, is a set of variable symbols defined as follows:

$$\begin{aligned}
\mathit{free} &: \mathcal{L} \to \mathbb{P}(\mathcal{V}) \\
\mathit{free}(x^i) &\coloneqq \{x^i\} \\
\mathit{free}(o(e_1, \ldots, e_n)) &\coloneqq \bigcup_{i=1}^{n} \mathit{free}(e_i) \\
\mathit{free}(\exists_{\mathcal{L}} x^i F) &\coloneqq \mathit{free}(F) \setminus \{x^i\} \\
\mathit{free}(\forall_{\mathcal{L}} x^i F) &\coloneqq \mathit{free}(F) \setminus \{x^i\}
\end{aligned}$$

•

Definition 8. Let F be a formula. F is a *closed formula* iff $\mathit{free}(F) = \emptyset$. •

Similar as for the free variables, we also define a function which retrieves all the bound variables from a formula.

Definition 9. The set of *bound variables* of a formula F, $\mathit{bound}(F)$, is a set of variable symbols defined as follows:

$$\begin{aligned}
\mathit{bound} &: \mathit{Exp}_0 \to \mathbb{P}(\mathcal{V}) \\
\mathit{bound}(P(t_1, \ldots, t_n)) &\coloneqq \emptyset \\
\mathit{bound}(\neg_{\mathcal{L}} F) &\coloneqq \mathit{bound}(F) \\
\mathit{bound}(c(F_1, F_2) &\coloneqq \mathit{bound}(F_1) \cup \mathit{bound}(F_2) \\
\mathit{bound}(\exists_{\mathcal{L}} x^i F) &\coloneqq \{x^i\} \cup \mathit{bound}(F) \\
\mathit{bound}(\forall_{\mathcal{L}} x^i F) &\coloneqq \{x^i\} \cup \mathit{bound}(F)
\end{aligned}$$

•

In the following we define a function which retrieves all the operation symbols that are no connectives from a given formula.

Definition 10. We define the function *getOpSyms* as:

$$
\begin{aligned}
getOpSyms : \mathcal{L} &\to \mathbb{P}(Op_\Sigma) \\
getOpSyms(x^i) &\coloneqq \emptyset \\
getOpSyms(\neg_{\mathcal{L}} F) &\coloneqq getOpSyms(F) \\
getOpSyms(c(F_1, F_2) &\coloneqq getOpSyms(F_1) \cup getOpSyms(F_2) \\
getOpSyms(\rho(e_1, \ldots, e_n)) &\coloneqq \{\rho\} \cup \bigcup_{i=1}^{n} getOpSyms(e_i) \\
getOpSyms(\exists_{\mathcal{L}} x^i F) &\coloneqq getOpSyms(F) \\
getOpSyms(\forall_{\mathcal{L}} x^i F) &\coloneqq getOpSyms(F)
\end{aligned}
$$

•

Our next goal is to introduce a notion that enables us to deal with the individual parts of a formula i.e. the sub-formulae of a formula.

To do this we will define the set of *positions* of a formula. The set of positions of formulae and the way we define sub-formulae are based on the notion of positions of terms presented in [24].

Definition 11. The set of *positions* of a formula F, $Pos(F)$, is a set of strings consisting of the numbers 1 and 2 defined as follows:

$$
\begin{aligned}
Pos : Exp_0 &\to \mathbb{P}(\{1,2\}^*) \\
Pos(P(t_1, \cdots t_n)) &\coloneqq \{\varepsilon\} \\
Pos(\neg_{\mathcal{L}} F) &\coloneqq \{1p \mid p \in Pos(F)\} \cup \{\varepsilon\} \\
Pos(c(F_1, F_2)) &\coloneqq \{ip \mid i \in \{1,2\}, p \in Pos(F_i)\} \cup \{\varepsilon\} \\
Pos(\exists_{\mathcal{L}} x^i F) &\coloneqq \{1p \mid p \in Pos(F)\} \cup \{\varepsilon\} \\
Pos(\forall_{\mathcal{L}} x^i F) &\coloneqq \{1p \mid p \in Pos(F)\} \cup \{\varepsilon\}
\end{aligned}
$$

Here ε denotes the empty string. •

With the positions of a formula we can now define a notion for retrieving all sub-formulae of a given formula.

Definition 12. Let F be a formula, $p \in Pos(F)$ and $i \in \{1,2\}$. We define the *sub-formula of F at position p*, $F\langle p \rangle$, as follows:

$$
\begin{aligned}
\cdot\langle\cdot\rangle : Exp_0 \times \{1,2\}^* &\to Exp_0 \\
F\langle\varepsilon\rangle &\coloneqq F \\
(\exists_{\mathcal{L}} x^i F_1)\langle 1p\rangle &\coloneqq F_1\langle p\rangle \\
(\forall_{\mathcal{L}} x^i F_1)\langle 1p\rangle &\coloneqq F_1\langle p\rangle \\
o(F_1, F_2)\langle ip\rangle &\coloneqq F_i\langle p\rangle \\
\neg_{\mathcal{L}} F_1\langle 1p\rangle &\coloneqq F_1\langle p\rangle
\end{aligned}
$$

•

Subsequently, we will define syntactic substitutions in formulae.

Definition 13. Let i be a non-boolean sort and t be a term of sort i. We define the *substitution*, $subs(e, x^i, t)$, of the variable x^i by the term t in an expression e as:

$$
\begin{aligned}
subs : \mathcal{L} \times (\bigcup_{s \in \Sigma.Sort} \mathcal{V}_s \times Exp_s) &\to \mathcal{L} \\
subs(x^i, x^i, t) &\coloneqq t \\
y^j \neq x^i \Rightarrow subs(y^j, x^i, t) &\coloneqq y^j \\
subs(o(e_1, \ldots, e_n), x^i, t) &\coloneqq o(subs(e_1, x^i, t), \ldots, subs(e_n, x^i, t)) \\
subs(\forall_{\mathcal{L}} x^i F, x^i, t) &\coloneqq \forall_{\mathcal{L}} x^i F \\
y^j \neq x^i \Rightarrow subs(\forall_{\mathcal{L}} y^j F, x^i, t) &\coloneqq \forall_{\mathcal{L}} y^j subs(F, x^i, t) \\
subs(\exists_{\mathcal{L}} x^i F, x^i, t) &\coloneqq \exists_{\mathcal{L}} x^i F \\
y^j \neq x^i \Rightarrow subs(\exists_{\mathcal{L}} y^j F, x^i, t) &\coloneqq \exists_{\mathcal{L}} y^j subs(F, x^i, t)
\end{aligned}
$$

•

Remark

- Obviously the results of such substitutions are again well-formed formulae.
- Unlike to the usual definition, here only a single variable can be replaced by some expression.

Last but not least, we will now define some notions concerning the semantics of formulae.

Definition 14.

- If we have $I(F) = \mathbb{T}$, then we denote this by $I \vDash F$.
- If we have $I(F) = \mathbb{F}$, then we denote this by $I \nvDash F$. •

Definition 15. Two formulae F_1 and F_2 are *semantically equivalent*, written as $F_1 \equiv F_2$, iff for each interpretation I:

$$I \vDash F_1 \Leftrightarrow I \vDash F_2$$ •

Remark It immediately follows that $F_1 \equiv F_2$ iff, for each interpretation I, we have that $I(F_1) = I(F_2)$.

Definition 16. We say that a formula F_1 entails a formula F_2, written as $F_1 \vDash F_2$ iff for each interpretation I such that $I \vDash F_1$ also $I \vDash F_2$ holds. •

Definition 17. Let F be a closed formula. F is *satisfiable* iff there is an interpretation I such that $I \vDash F$. F is *valid* iff for each interpretation I, $I \vDash F$ holds. •

Definition 18. Two closed formulae F_1 and F_2 are *equisatisfiable* iff F_1 is satisfiable iff F_2 is satisfiable •

3.3 Satisfiability Modulo Theories

In this section we will deal with the satisfiability modulo theory problem with respect to the many-sorted first order logic presented above. This section is based on [4].

Definition 19. Let Σ and Ξ be signatures such that $\Sigma \subseteq \Xi$ and let $\mathcal{A}$ be a structure of the signature Ξ. Then we denote the structure, which is the result of restricting $\mathcal{A}$ to symbols respectively sorts in Σ, with $\mathcal{A}^{\Sigma}$. •

Definition 20. Let Σ be a signature and $\mathcal{C}$ be a class of structures of the signature Σ. Then we denote the pair $\langle \Sigma, \mathcal{C} \rangle$ as a theory (see [4]). Let T be a theory. Then we denote its signature with Σ_T. •

Definition 21. Let T be a theory over the signature Σ. Then we define Op_T as the set of operation symbols from Σ, i.e. $Op_T \coloneqq Op_{\Sigma}$. •

Definition 22. Let T be a theory over the signature Σ, whose class of interpretations we denote with $\mathcal{C}$, let Ξ be a signature such that $\Sigma \subseteq \Xi$, let $\langle \mathcal{A}, M \rangle$ be an interpretation over the signature Ξ and let $\mathcal{A}^{\Sigma} \in \mathcal{C}$. Then we denote $\langle \mathcal{A}, M \rangle$ as a T-interpretation. Such a structure $\mathcal{A}$ we denote as a T-structure. •

Definition 23. Let T be a theory over the signature Σ, let Ξ be a signature such that $\Sigma \subseteq \Xi$ and let F be a closed formula of the signature Ξ. Then we say that F is T-satisfiable or satisfiable with respect to the theory T if there is a T-interpretation I such that $I \vDash F$. Respectively we say that F is T-valid or valid with respect to the theory T if for all T-interpretations I, $I \vDash F$ holds (see [4]). •

Similarly, we can also introduce the entailment for T-interpretations.

Definition 24. Let T be a theory over the signature Σ, let Ξ be a signature such that $\Sigma \subseteq \Xi$ and let F_1 and F_2 be closed formulae of the signature Ξ. Then F_1 T-entails F_2, written as $F_1 \vDash_T F_2$ iff for each T-interpretation I such that $I \vDash F_1$ also $I \vDash F_2$ holds. •

Definition 25. Let T be a theory over the signature Σ, let Ξ be a signature such that $\Sigma \subseteq \Xi$ and let F_1 and F_2 be closed formulae of the signature Ξ. Then the formulae F_1 and F_2 are *T-equisatisfiable* or *equisatisfiable with respect to the theory* T iff F_1 is T-satisfiable iff F_2 is T-satisfiable. •

Example 26. Assume we have a signature Σ_I such that $\Sigma_I.Sort = \{0, 1\}$. Moreover assume that $2, <$ are operation symbols in Op_{Σ} such that $\Sigma_I.Func(2) = 1$ and $\Sigma_I.Func(<) = (0, 1, 1)$. Furthermore let $\mathcal{C}_I$ be the class of all interpretations that interpret the sort 1 with $\mathbb{Z}$ and which assign to the symbol 2 the actual number 2 and to the symbol $<$ the actual predicate $<$ (note that in this case the class consist only of a single interpretation). Now let us consider the formula $f(2) < f(2)$ over a suitable signature, with the same sorts as Σ_I. This formula is obviously satisfiable, e.g. we could interpret the sort 1 with $\mathbb{Z}$, the symbol f with the identity function and the symbol $<$ with the greater-equals predicate. In contrast to this, the formula is unsatisfiable modulo the theory $\langle \Sigma_I, \mathcal{C}_I \rangle$, as no number can be less than itself.

Remark By considering T-interpretations instead of general interpretations we effectively impose semantic restrictions. As we mentioned at the beginning of the thesis, syntactic restrictions are also of importance. Such restrictions can be imposed by allowing only certain signatures, for which T-interpretations may be considered. For example, we could only consider signatures with the same sorts as in the respective theory. Moreover, as we already mentioned, we can restrict the considered expressions to expressions without quantifiers. Besides this there are also other restrictions, but we will not go any deeper here. Examples for such restrictions can be seen in [2]. We will not formally elaborate such restrictions.

Remark In the above definition of a many-sorted logic we required that there is a sort 0 in a signature. Furthermore, we also required that the set $\{\neg_{\mathcal{L}}, \wedge_{\mathcal{L}}, \vee_{\mathcal{L}}, \Rightarrow_{\mathcal{L}}, \Leftrightarrow_{\mathcal{L}}\}$ is a subset of the set of operation symbols of the signature. Last but not least we required that the sort 0 is interpreted with the domain of boolean values and that the operation symbols in the set mentioned before get their expected semantic meaning. We could also define this logic by means of theories: We define a signature Σ containing only the sort 0 and the function symbols $\{\neg_{\mathcal{L}}, \wedge_{\mathcal{L}}, \vee_{\mathcal{L}}, \Rightarrow_{\mathcal{L}}, \Leftrightarrow_{\mathcal{L}}\}$. Let $\mathcal{C}$ be the class of all structures such that the domain of boolean values is assigned to the sort 0 and where the function symbols get their expected meaning. This means we found a theory $T = \langle \Sigma, \mathcal{C} \rangle$ such that satisfiability respectively validity of a formula F in our many-sorted logic is equivalent to satisfiability / validity with respect to the theory T in a logic without the restrictions we use in the presented logic.

3.4 Properties of this Logic

In this section we will prove several properties about the logical framework we presented above. Most of the results presented in this section are well-known from "usual" first order logic.

Remark In this section we will use meta-variables like in the previous sections, e.g., F denotes a formula, t a term and so on.

The proofs in this section and in the subsequent sections are mainly proofs by induction. To be more precisely this means that we first prove the respective property for terms and then for formulae. For both cases we will usually use structural induction. But for some proofs we will need a stronger induction hypothesis — we have proofs where we need the hypothesis that a property holds for all sub-formulae instead of only the direct sub-formulae of a formula. In this case we use the more general *Noetherian Induction* (see [41]). As a perquisite for this induction we need a relation $>_{\mathcal{L}}$ on the formulae from $\mathcal{L}$. For formulae F_1 and F_2 we define $>_{\mathcal{L}}$ such that $F_1 >_{\mathcal{L}} F_2$ iff

$$|Pos(F_1)| > |Pos(F_2)|.$$

We can see that there cannot be an infinite sequence $F_1, F_2, \cdots$ of formulae such that $F_1 >_{\mathcal{L}} F_2 >_{\mathcal{L}} \cdots$. This means that we can use this relation for Noetherian induction.

Thus, to prove that a property P holds for every formula we can prove:

$$\forall F_1(\forall F_2 : F_1 >_{\mathcal{L}} F_2 \Rightarrow P(F_2)) \Rightarrow P(F_1)$$

This is now exactly what we need as here we can conclude from the induction hypothesis that a property holds for all sub-formulae and not only for the direct sub-formulae. Subsequently, we will not explicitly state the used proof technique.

Lemma 27. *Let F_1 and F_2 be formulae and let $F_1 \equiv F_2$. Then we have:*

- *F_1 and F_2 are equisatisfiable.*
- *Let T be a theory with an appropriate signature then F_1 and F_2 are equisatisfiable with respect to the theory T.*

Proof. We will only prove the first assertion of this lemma, the second one can be proved similarly. Assume $F_1 \equiv F_2$ and F_1 is satisfiable. Then there is some interpretation I such that $I \vDash F_1$. Since $F_1 \equiv F_2$ we get $I \vDash F_2$. So also F_2 is satisfiable. The other direction can be proved analogously. □

Lemma 28. *Equisatisfiability is transitive.*

Proof. Obvious. □

Lemma 29. *Let T be a theory then equisatisfiability with respect to this theory is transitive.*

Proof. Obvious. □

Lemma 30. *Let F_1 and F_2 be closed formulae. Then the following assertions are equivalent*

- *$F_1 \vDash F_2$*
- *$F_1 \Rightarrow_{\mathcal{L}} F_2$ is valid*
- *$F_1 \wedge_{\mathcal{L}} \neg_{\mathcal{L}} F_2$ is unsatisfiable*

Proof. As $(F_1 \Rightarrow_{\mathcal{L}} F_2) \equiv \neg_{\mathcal{L}}(F_1 \wedge_{\mathcal{L}} \neg_{\mathcal{L}} F_2)$ it is obvious that the second and the third assertions are equivalent.

"1 $\Rightarrow$ 2": Let I be an interpretation. Assume that $F_1 \vDash F_2$. We show that $I \vDash F_1 \Rightarrow_{\mathcal{L}} F_2$. If $I \nvDash F_1$ then $I \vDash F_1 \Rightarrow_{\mathcal{L}} F_2$. If $I \vDash F_1$ we get by the assumption that $I \vDash F_2$, that $I \vDash F_1 \Rightarrow_{\mathcal{L}} F_2$.

"2 $\Rightarrow$ 1": Let I be an interpretation. We assume 2 and that $I \vDash F_1$ and show $I \vDash F_2$. Assuming that $I \nvDash F_2$ yields a contradiction to 2. Therefore, we have $F_1 \vDash F_2$. □

Similarly, we also have

Lemma 31. *Let T be a theory and let F_1 and F_2 be closed formulae. Then the following assertions are equivalent.*

- $F_1 \vDash_T F_2$
- $F_1 \Rightarrow_{\mathcal{L}} F_2$ *is T-valid*
- $F_1 \wedge_{\mathcal{L}} \neg_{\mathcal{L}} F_2$ *is T-unsatisfiable*

Proof. Can be proved analogously to Lemma 30. □

Remark As in the usual first order logic also in the logic, which we use here, De Morgan's laws hold. Although we will use them, we will not give a proof.

Lemma 32.

(i) $F \equiv F'$ *iff* $\neg_{\mathcal{L}} F \equiv \neg_{\mathcal{L}} F'$

(ii) $F_1 \equiv F_1'$ *and* $F_2 \equiv F_2'$ *implies* $F_1 \wedge_{\mathcal{L}} F_2 \equiv F_1' \wedge_{\mathcal{L}} F_2'$

(iii) $F_1 \equiv F_1'$ *and* $F_2 \equiv F_2'$ *implies* $F_1 \vee_{\mathcal{L}} F_2 \equiv F_1' \vee_{\mathcal{L}} F_2'$

(iv) $F_1 \equiv F_1'$ *and* $F_2 \equiv F_2'$ *implies* $F_1 \Rightarrow_{\mathcal{L}} F_2 \equiv F_1' \Rightarrow_{\mathcal{L}} F_2'$

(v) $F_1 \equiv F_1'$ *and* $F_2 \equiv F_2'$ *implies* $F_1 \Leftrightarrow_{\mathcal{L}} F_2 \equiv F_1' \Leftrightarrow_{\mathcal{L}} F_2'$

Proof.

(i) We prove the direction $F \equiv F'$ implies $\neg_{\mathcal{L}} F \equiv \neg_{\mathcal{L}} F'$. The other direction can be proved similarly. Let I be an interpretation such that $I \vDash \neg_{\mathcal{L}} F$ and $F \equiv F'$. So $I \nvDash F$. Since $F \equiv F'$ we get $I \nvDash F'$ and therefore $I \vDash \neg_{\mathcal{L}} F'$. Similarly we can show that $I \vDash \neg_{\mathcal{L}} F'$ implies $I \vDash \neg_{\mathcal{L}} F$

(ii) Let I be an interpretation such that $I \vDash F_1 \wedge_{\mathcal{L}} F_2$ and $F \equiv F'$. This implies $I \vDash F_1$ and $I \vDash F_2$. By the hypothesis we get $I \vDash F_1'$ and $I \vDash F_2'$. So we get $I \vDash F_1' \wedge_{\mathcal{L}} F_2'$. The other direction can be showed analogously.

(iii-v) Can be proved similarly like (ii). □

Lemma 33.

(i) $F \equiv F'$ *implies* $\exists_{\mathcal{L}} x^i F \equiv \exists_{\mathcal{L}} x^i F'$

(ii) $F \equiv F'$ *implies* $\forall_{\mathcal{L}} x^i F \equiv \forall_{\mathcal{L}} x^i F'$

Proof. (i) Assume $F \equiv F'$. If $\langle \mathcal{A}, M \rangle \vDash \exists_{\mathcal{L}} x^i F$ then

$$\{y \in \mathcal{A}_i \mid \langle \mathcal{A}, M[x^i \mapsto y] \rangle \vDash F\} \neq \emptyset.$$

Since this set is not empty we can choose an element of this set.

$$z \in \{y \in \mathcal{A}_i \mid \langle \mathcal{A}, M[x^i \mapsto y] \rangle \vDash F\}$$

So $\langle \mathcal{A}, M[x^i \mapsto z] \rangle \vDash F$. By the assumption we get $\langle \mathcal{A}, M[x^i \mapsto z] \rangle \vDash F'$. Thus we have that:

$$\{y \in \mathcal{A}_i \mid \langle \mathcal{A}, M[x^i \mapsto y] \rangle \vDash F'\} \neq \emptyset.$$

The direction $\langle \mathcal{A}, M \rangle \vDash \exists_{\mathcal{L}} x^i F'$ can be proved analogously. (ii) can be proved similarly as (i) □

Lemma 34. *Let i be a non-boolean sort, F be a formula and t a term of sort i. Then we have:*

(i) $Pos(F) = Pos(subs(F, x^i, t)$

(ii) $\forall p \in Pos(F) : subs(F, x^i, t)\langle p \rangle = subs(F\langle p \rangle, x^i, t)$

Proof. In both steps we have to distinguish between the possible shapes of F.
(i):

- *Case:* $F = P(t_1, \ldots, t_n)$
  In this case we have $Pos(F) = \{\varepsilon\}$. As we know that

  $$subs(F, x^i, t) = P(subs(t_1, x^i, t), \cdots, subs(t_2, x^i, t))$$

  we also get

  $$Pos(subs(F, x^i, t)) = \{\varepsilon\}$$

- *Case:* $F = F_1 \wedge_{\mathcal{L}} F_2$
  Our hypothesis is:

  $$Pos(F_1) = Pos(subs(F_1, x^i, t))$$
  $$Pos(F_2) = Pos(subs(F_2, x^i, t))$$

  From the definition of positions and the hypothesis we can conclude

  $$\begin{aligned} Pos(F) &= \{1p \mid p \in Pos(F_1)\} \cup \{2p \mid p \in Pos(F_2)\} \cup \{\varepsilon\} \\ &= \{1p \mid p \in Pos(subs(F_1, x^i, t))\} \cup \{2p \mid p \in Pos(subs(F_1, x^i, t))\} \cup \{\varepsilon\} \\ &= Pos(subs(F, x^i, t)) \end{aligned}$$

All the other cases can be proved similar like the above ones.
With this result we can prove (ii):

- *Case:* $F = P(t_1, \ldots, t_n)$
  In this case we have $Pos(F) = \{\varepsilon\}$.

  $$subs(F, x^i, t)\langle \varepsilon \rangle = subs(F, x^i, t) = subs(F\langle \varepsilon \rangle, x^i, t)$$

- *Case:* $F = F_1 \wedge_{\mathcal{L}} F_2$
  By the hypothesis we get:

$$\forall p \in Pos(F_1) : subs(F,x^i,t)\langle p\rangle = subs(F_1\langle p\rangle, x^i, t)$$
$$\forall p \in Pos(F_2) : subs(F_2,x^i,t)\langle p\rangle = subs(F_2\langle p\rangle, x^i, t)$$

  Similar like in the previous case we can show that the assertion holds for ε. So let us fix a $p \in Pos(F) \setminus \{\varepsilon\}$ and prove the assertion for F. Either $p = 1p_1$ or $p = 2p_2$, where $p_1 \in Pos(F_1)$ and $p_2 \in Pos(F_2)$, must hold. W.l.o.g. $p = 1p_1$

$$subs(F\langle p\rangle, x^i, t) = subs(F_1\langle p_1\rangle, x^i, t)$$
$$= subs(F_1, x^i, t)\langle p_1\rangle = subs(F, x^i, t)\langle 1p_1\rangle$$

All the other cases can be proved similarly like the above ones. □

Remark Earlier we pointed out that for some proofs we apply *Noetherian Induction* with respect to the cardinality of the set of positions. In combination with the above lemma this means, that if we have a formula, for which the induction hypothesis holds, then the hypothesis still holds if we apply a substitution to this formula.

Lemma 35. *Let T be a term and $I_1 = \langle \mathcal{A}, M_1\rangle$ and $I_2 = \langle \mathcal{A}, M_2\rangle$ be two interpretations such that $\forall x^i \in free(T) : M_1(x^i) = M_2(x^i)$. Then we have $I_1(T) = I_2(T)$.*

Proof. This proof is based on a similar proof in [21].

We assume that I_1 and I_2 are interpretations such that the premise of the lemma is fulfilled. We have to distinguish two cases, either T is a variable or a function symbol applied to terms.

- *Case:* T is a variable – $T = x^i$
  Then $x^i \in free(T)$. This implies $M_1(x^i) = M_2(x^i)$ therefore $I_1(T) = I_2(T)$

- *Case:* T is a function symbol applied to terms – $T = f(t_1, \ldots, t_n)$
  By the hypothesis the assertion holds for the t_i.

$$free(T) = \bigcup_{i=1}^{n} free(t_i)$$
$$\text{Thus } \underset{i}{\forall} 1 \leq i \leq n : free(t_i) \subseteq free(T)$$
$$\text{Thus } \underset{i}{\forall} 1 \leq i \leq n : I_1(t_i) = I_2(t_i) \qquad \text{(follows by the hypothesis)}$$

  Now we can use the definition of interpretations of function symbols applied to terms and get the result.

$$I_1(f(t_1, \ldots, t_n)) = f^{\mathcal{A}}(I_1(t_1), \ldots, I_1(t_n)) = f^{\mathcal{A}}(I_2(t_1), \ldots, I_2(t_n))$$
$$= I_2(f(t_1, \ldots, t_n))$$

  Thus, we have that: $I_1(T) = I_2(T)$. □

Lemma 36. *Let F be a formula and $I_1 = \langle \mathcal{A}, M_1 \rangle$ and $I_2 = \langle \mathcal{A}, M_2 \rangle$ be two interpretations such that $\forall x^i \in \mathit{free}(F) : M_1(x^i) = M_2(x^i)$. Then we have $I_1(F) = I_2(F)$.*

Proof. This proof is based on a similar proof in [21].

We assume that I_1 and I_2 are interpretations such that the premise of the lemma is fulfilled. We have to distinguish between the possible shapes of F

- *Case:* F is a predicate symbol applied to terms – $F = P(t_1, \ldots, t_n)$
  We know $\mathit{free}(F) = \bigcup_{i=1}^{n} \mathit{free}(t_i)$. With Lemma 35 we get that $\underset{i}{\forall} 1 \leq i \leq n : I_1(t_i) = I_2(t_i)$ Now we use the definition of interpretations of predicate symbols applied to terms.

  $$\begin{aligned} I_1(P(t_1, \ldots, t_n)) &= P^{\mathcal{A}}(I_1(t_1), \ldots, I_1(t_n)) = P^{\mathcal{A}}(I_2(t_1), \ldots, I_2(t_n)) \\ &= I_2(P(t_1, \ldots, t_n)) \end{aligned}$$

  Thus, we have that: $I_1(F) = I_2(F)$.

- *Case:* $F = \neg_{\mathcal{L}} F_1$
  Since $\mathit{free}(\neg_{\mathcal{L}} F_1) = \mathit{free}(F_1)$ we get that $I_1(F_1) = I_2(F_1)$. From this we can easily conclude that $I_1(F) = I_2(F)$

- *Case:* $F = F_1 \wedge_{\mathcal{L}} F_2$
  Since $\mathit{free}(F_1 \wedge_{\mathcal{L}} F_2) = \mathit{free}(F_1) \cup \mathit{free}(F_2)$ we can conclude that $I_1(F_1) = I_2(F_1)$ and $I_1(F_2) = I_2(F_2)$. From this we can easily conclude that $I_1(F) = I_2(F)$

- *Case:* $F = \forall_{\mathcal{L}} x^i F_1$
  We know $\mathit{free}(F) = \mathit{free}(F_1) \setminus \{x^i\}$. We know that:

  $$\begin{aligned} I_1 \vDash F \Leftrightarrow \{y \in \mathcal{A}_i \mid \langle \mathcal{A}, M_1[x^i \mapsto y] \rangle \vDash F_1\} = \mathcal{A}_i \\ I_2 \vDash F \Leftrightarrow \{y \in \mathcal{A}_i \mid \langle \mathcal{A}, M_2[x^i \mapsto y] \rangle \vDash F_1\} = \mathcal{A}_i \end{aligned}$$

  Fix $y \in \mathcal{A}_i$ then by our assumption on I_1 and I_2 we get:

  $$\forall z^k \in \mathit{free}(F_1) : M_1[x^i \mapsto y](z^k) = M_2[x^i \mapsto y](z^k).$$

  We can now use the hypothesis to conclude that:

  $$\langle \mathcal{A}, M_1[x^i \mapsto y] \rangle(F_1) = \langle \mathcal{A}, M_2[x^i \mapsto y] \rangle(F_2).$$

  Since y was arbitrary this holds for each $y \in \mathcal{A}_i$. This means

  $$\{y \in \mathcal{A}_i \mid \langle \mathcal{A}, M_1[x^i \mapsto y] \rangle \vDash F_1\} = \{y \in \mathcal{A}_i \mid \langle \mathcal{A}, M_2[x^i \mapsto y] \rangle \vDash F_1\}.$$

  This implies $I_1(F) = I_2(F)$

- *Case:* Remaining cases
  All the other cases can be proved similarly like the above ones. □

Lemma 37. *Let T be a term and let $\mathcal{A}$ and $\mathcal{B}$ be two structures such that*

$$\forall i \in \Sigma.Sort : \mathcal{A}_i = \mathcal{B}_i$$
$$\forall f \in getOpSyms(T) : f^{\mathcal{A}} = f^{\mathcal{B}}$$

Furthermore let M be some variable assignment. I_1, I_2 shall denote the interpretations $\langle \mathcal{A}, M \rangle$ and $\langle \mathcal{B}, M \rangle$. Then we have:

$$I_1(T) = I_2(T)$$

Proof. We assume that I_1 and I_2 are interpretations such that the premise of the lemma is fulfilled. We have to distinguish two cases:

- *Case:* $T = x^i$
  $I_1(x^i) = M(x^i) = I_2(x^i)$
- *Case:* $T = f(t_1, \ldots, t_n)$
  We know $\underset{i}{\forall} 1 \leq i \leq n : getOpSyms(t_i) \subseteq getOpSyms(T)$. So we get $\underset{i}{\forall} 1 \leq i \leq n : I_1(t_i) = I_2(t_i)$. From this we can use conclude the wished result.

  $$\begin{aligned} I_1(f(t_1, \ldots, t_n)) &= f^{\mathcal{A}}(I_1(t_1), \ldots, I_1(t_n)) = f^{\mathcal{A}}(I_2(t_1), \ldots, I_2(t_n)) \\ &= I_2(f(t_1, \ldots, t_n)) \end{aligned}$$

  Thus, we have that: $I_1(T) = I_2(T)$. □

Lemma 38. *Let F be a formula and let $\mathcal{A}$ and $\mathcal{B}$ be two structures such that*

$$\forall i \in \Sigma.Sort : \mathcal{A}_i = \mathcal{B}_i$$
$$\forall f \in getOpSyms(F) : f^{\mathcal{A}} = f^{\mathcal{B}}$$

Furthermore let M be some variable assignment. I_1, I_2 shall denote the interpretations $\langle \mathcal{A}, M \rangle$ and $\langle \mathcal{B}, M \rangle$. Then we have:

$$I_1(F) = I_2(F)$$

Proof. We assume that I_1 and I_2 are interpretations such that the premise of the lemma is fulfilled. We have to distinguish between the possible shapes of F

- *Case:* $F = P(t_1, \ldots, t_n)$
  Since $getOpSyms(F) = \{P\} \cup (\bigcup_{i=1}^{n} getOpSyms(t_i))$ and because of Lemma 37 we get $\underset{i}{\forall} 1 \leq i \leq n : I_1(t_i) = I_2(t_i)$. Similar as in previous proofs we can now use the definition of interpretations of predicate symbols applied to terms and get the wished result.
- *Case:* $F = \neg_{\mathcal{L}} F_1$
  Since $getOpSyms(\neg_{\mathcal{L}} F1) = getOpSyms(F1)$ we can conclude that $I_1(F_1) = I_2(F_1)$ From this $I_1(\neg_{\mathcal{L}} F_1) = I_2(\neg_{\mathcal{L}} F_1)$ can be concluded.

- *Case:* $F = \forall_{\mathcal{L}} x^i F_1$
  Because $getOpSyms(F) = getOpSyms(F_1)$ we get that

  $$\forall y \in \mathcal{A}_i : \langle \mathcal{A}_1, M[x^i \mapsto y] \rangle (F_1) = \langle \mathcal{A}_2, M[x^i \mapsto y] \rangle (F_1)$$

  This implies $I_1(F) = I_2(F)$

The remaining cases can be proved similar. □

Lemma 39. *Let i be a non-boolean sort, T be a term, t a term of sort i and x^i be a variable. Then we have:*

$$getOpSyms(subs(T, x^i, t)) \subseteq getOpSyms(T) \cup getOpSyms(t)$$

Proof. We have to distinguish two cases:

- *Case:* T is a variable
  If $T = x^i$ then $subs(T, x^i, t) = t$. So the assertion holds. If $T \neq x^i$ then $subs(T, x^i, t) = T$. Also in this case the assertion obviously holds.

- *Case:* $T = f(t_1, \ldots, t_n)$

  $$subs(T, x^i, t) = f(subs(t_1, x^i, t), \ldots, subs(t_n, x^i, t))$$
  $$\text{Thus } getOpSyms(subs(T, x^i, t)) = \{f\} \cup \bigcup_{i=1}^{n} getOpSyms(subs(t_i, x^i, t))$$

  By the hypothesis we can conclude that:

  $$\begin{aligned} &\subseteq \{f\} \cup \bigcup_{i=1}^{n} (getOpSyms(t_i) \cup getOpSyms(t)) \\ &= getOpSyms(f(t_1, \ldots, t_n)) \cup getOpSyms(t) \end{aligned}$$

□

Lemma 40. *Let i be a non-boolean sort, F be a formula, t a term of sort i and x^i a variable. Then we have:*

$$getOpSyms(subs(F, x^i, t)) \subseteq getOpSyms(F) \cup getOpSyms(t)$$

Proof. We have to distinguish between the possible shapes of F:

- *Case:* $F = P(t_1, \ldots, t_n)$
  Can be proved as the second case of Lemma 39. The only difference is that we have to use Lemma 39 instead of the hypothesis.

- *Case:* $F = \forall_{\mathcal{L}} y^j F_1$
  If $x^i = y^j$ then $subs(\forall_{\mathcal{L}} x^i F_1, x^i, t) = \forall_{\mathcal{L}} x^i F_1$. So the assertion holds.
  If $x^i \neq y^j$ then

$$subs(\forall_{\mathcal{L}} y^j F_1, x^i, t) = \forall_{\mathcal{L}} y^j subs(F_1, x^i, t)$$
$$\text{Thus } getOpSyms(subs(\forall_{\mathcal{L}} y^j F_1, x^i, t)) = getOpSyms(subs(F_1, x^i, t))$$

  By the hypothesis we can conclude that:

$$getOpSyms(subs(F_1, x^i, t)) \subseteq getOpSyms(F_1) \cup getOpSyms(t)$$

  Since $getOpSyms(F_1) = getOpSyms(F)$ this is the wished result.

The remaining cases can be proved in a similar way. □

Lemma 41. *Let i be a non-boolean sort, F be a variable, t a term of sort i and x^i a variable. Then we have:*

$$bound(F) = bound(subs(F, x^i, t))$$

Proof. We have to distinguish between the possible shapes of F:

- *Case:* $F = P(t_1, \ldots, t_n)$
  Since $bound(F) = \emptyset$ we get the assertion as:

$$\begin{aligned} & bound(subs(F, x^i, t)) \\ =& bound(P(subs(t_1, x^i, t), \ldots, subs(t_n, x^i, t))) = \emptyset \end{aligned}$$

- *Case:* $F = F_1 \wedge_{\mathcal{L}} F_2$
  By using the definition of *bound* we get:

$$\begin{aligned} bound(F) &= bound(F_1) \cup bound(F_2) \\ bound(subs(F, x^i, t)) &= bound(subs(F_1, x^i, t) \wedge_{\mathcal{L}} subs(F_2, x^i, t)) \\ &= bound(subs(F_1, x^i, t)) \cup bound(subs(F_2, x^i, t)) \end{aligned}$$

  By the hypothesis we can conclude:

$$= bound(F_1) \cup bound(F_2) = bound(F)$$

- *Case:* $F = \forall_{\mathcal{L}} y^j F_1$
  If $y^j = x^i$ then $subs(F, x^i, t) = F$ so the assertion holds. If $y^j \neq x^i$ we can conclude by the definition of *bound* and by the hypothesis that:

$$bound(subs(F, x^i, t)) = bound(F_1) \cup \{y^j\} = bound(F)$$

The remaining cases can be proved similar. □

Lemma 42. *Let i be a non-boolean sort, T be a term, t be a term of sort i and x^i a variable. Then we have:*

$$\begin{aligned} x^i \in \mathit{free}(T) &\Rightarrow \mathit{free}(\mathit{subs}(T, x^i, t)) \\ &= (\mathit{free}(T) \setminus \{x^i\}) \cup \mathit{free}(t) \\ x^i \notin \mathit{free}(T) &\Rightarrow \mathit{free}(\mathit{subs}(T, x^i, t)) = \mathit{free}(T) \end{aligned}$$

Proof. We have to distinguish two cases:

- *Case:* T is a variable
  If $T = x^i$ then obviously $x^i \in \mathit{free}(T)$. Furthermore:

  $$\begin{aligned} \mathit{free}(\mathit{subs}(T, x^i, t)) &= \mathit{free}(t) \\ \mathit{free}(T) \setminus \{x^i\} &= \emptyset \end{aligned}$$

  From this we immediately get the result. If $T \neq x^i$ then $x^i \notin \mathit{free}(T)$. Additionally $\mathit{subs}(T, x^i, t) = T$. This implies the result.

- *Case:* $T = f(t_1, \ldots, t_n)$
  Assume the assertion holds for the terms t_j.

  $$\begin{aligned} &\mathit{subs}(T, x^i, t) = f(\mathit{subs}(t_1, x^i, t), ..., \mathit{subs}(t_n, x^i, t)) \\ &\text{Thus } \mathit{free}(\mathit{subs}(T, x^i, t)) = \bigcup_{j=1}^{n} \mathit{free}(\mathit{subs}(t_j, x^i, t)) \end{aligned}$$

  If $x^i \notin \mathit{free}(T)$ then $\forall j : x^i \notin \mathit{free}(t_j)$ So by the assumption:

  $$\bigcup_{j=1}^{n} \mathit{free}(\mathit{subs}(t_j, x^i, t)) = \bigcup_{j=1}^{n} \mathit{free}(t_j) = \mathit{free}(T)$$

  If $x^i \in \mathit{free}(T)$ then $\exists j : x^i \in \mathit{free}(t_j)$ So by the assumption:

  $$\bigcup_{j=1}^{n} \mathit{free}(\mathit{subs}(t_j, x^i, t)) = \bigcup_{j \in J_1} \mathit{free}(t_j) \cup \bigcup_{j \in J_2} (\mathit{free}(t_j) \setminus \{x^i\} \cup \mathit{free}(t))$$

  Where J_1 is the set of all indices j such that $x^i \notin \mathit{free}(t_j)$ and J_2 the set of all indices such that $x^i \in \mathit{free}(t_j)$

  $$\begin{aligned} &= (\bigcup_{j=1}^{n} \mathit{free}(t_j)) \setminus \{x^i\} \cup \mathit{free}(t) \\ &= \mathit{free}(T) \setminus \{x^i\} \cup \mathit{free}(t) \end{aligned}$$

  □

Lemma 43. *Let i be a non-boolean sort, F be a formula, t be a term of sort i and x^i a variable. If $\mathit{free}(t) \cap \mathit{bound}(F) = \emptyset$ then we have:*

$$\begin{aligned} x^i \in \mathit{free}(F) &\Rightarrow \mathit{free}(\mathit{subs}(F, x^i, t)) \\ &= (\mathit{free}(F) \setminus \{x^i\}) \cup \mathit{free}(t) \\ x^i \notin \mathit{free}(T) &\Rightarrow \mathit{free}(\mathit{subs}(T, x^i, t)) = \mathit{free}(T) \end{aligned}$$

Proof. We have to distinguish between the possible shapes of F:

- *Case:* $F = P(t_1, \ldots, t_n)$
  Can be proved similar to the second case in Lemma 42. The difference is that instead of the hypothesis we use Lemma 42.

- *Case:* $F = F_1 \wedge_{\mathcal{L}} F_2$
  If the premise of the lemma is satisfied for F, it is obviously also satisfied for F_1 and F_2. So by the hypothesis we can use the conclusion of the lemma.

  $$\begin{aligned} &\mathit{free}(\mathit{subs}(F_1 \wedge_{\mathcal{L}} F_2, x^i, t)) \\ &= \mathit{free}(\mathit{subs}(F_1, x^i, t)) \cup \mathit{free}(\mathit{subs}(F_2, x^i, t)) \end{aligned}$$

  By the hypothesis and since $x^i \in \mathit{free}(F)$ iff $x^i \in \mathit{free}(F_1)$ or $x^i \in \mathit{free}(F_2)$ we get the result.

- *Case:* $\forall_{\mathcal{L}} y^j F_1$
  Obviously if the premise of the lemma holds for F it also holds for F_1. So by the hypothesis we can use the conclusion of the lemma for F_1. If $x^i = y^j$ then $\mathit{subs}(F, x^i, t) = F$ so the assertion holds. If $x^i \neq y^j$ then:

  $$\mathit{free}(\mathit{subs}(F, x^i, t)) = \mathit{free}(\mathit{subs}(F_1, x^i, t)) \setminus \{y^j\}$$

  If $x^i \in \mathit{free}(F_1)$ then by the hypothesis:

  $$= ((\mathit{free}(F_1) \setminus \{x^i\}) \cup \mathit{free}(t)) \setminus \{y^j\} = \mathit{free}(F) \setminus \{x^i\} \cup \mathit{free}(t)$$

  If $x^i \notin \mathit{free}(F_1)$ then by the hypothesis:

  $$= \mathit{free}(F_1) \setminus \{y^j\} = \mathit{free}(F)$$

  Since $x^i \neq y^j$ we get that $x^i \in \mathit{free}(F)$ iff $x^i \in \mathit{free}(F_1)$. This implies the assertion.

The other cases can be proved similar. □

Lemma 44. *Let i be a non-boolean sort, F be a formula, t be a term of sort i and x^i a variable. Then we have:*

$$\mathit{free}(\mathit{subs}(F, x^i, t)) \subseteq (\mathit{free}(F) \setminus \{x^i\}) \cup \mathit{free}(t)$$

Proof. Similar as the proof of Lemma 43 but easier. □

Lemma 45. *Let i be a non-boolean sort, F be a formula, t be a term of sort i and x^i a variable. Then we have:*

$$\mathit{free}(F) \setminus \{x^i\} \subseteq \mathit{free}(\mathit{subs}(F, x^i, t))$$

Proof. We have to distinguish between the possible shapes of F:

- *Case:* $F = P(t_1, \ldots, t_n)$
  Since $\mathit{bound}(F) = \emptyset$ we can use Lemma 43 and get the result.

- *Case:* $F = F_1 \wedge_{\mathcal{L}} F_2$
  By the hypothesis we have:

$$\mathit{free}(F_1) \setminus \{x^i\} \subseteq \mathit{free}(\mathit{subs}(F_1, x^i, t))$$
$$\mathit{free}(F_2) \setminus \{x^i\} \subseteq \mathit{free}(\mathit{subs}(F_2, x^i, t))$$

  We also know that $\mathit{free}(F_1 \wedge_{\mathcal{L}} F_2) = \mathit{free}(F_1) \cup \mathit{free}(F_2)$. Thus we have:

$$\begin{aligned}\mathit{free}(F) \setminus \{x^i\} &= (\mathit{free}(F_1) \setminus \{x^i\}) \cup (\mathit{free}(F_2) \setminus \{x^i\}) \\ &\subseteq \mathit{free}(\mathit{subs}(F_1, x^i, t)) \cup \mathit{free}(\mathit{subs}(F_2, x^i, t)) \\ &= \mathit{free}(\mathit{subs}(F, x^i, t))\end{aligned}$$

- *Case:* $F = \forall_{\mathcal{L}} y^j F_1$
  If $x^i = y^j$ then we get, similar to previous proofs, the result immediately. If $x^i \neq y^j$ then we have:

$$\begin{aligned}\mathit{free}(F) \setminus \{x^i\} &= (\mathit{free}(F_1) \setminus \{y^j\}) \setminus \{x^i\} \\ &\subseteq \mathit{free}(\mathit{subs}(F_1, x^i, t) \setminus \{y^j\} \\ &= \mathit{free}(\mathit{subs}(F, x^i, t))\end{aligned}$$

The other cases can be proved similar. □

Lemma 46. *Let i be a non-boolean sort, T be a term, t be a term of sort i, let x^i be a variable and let $I = \langle \mathcal{A}, M \rangle$ be an interpretation. Then we have:*

$$I(\mathit{subs}(T, x^i, t) = \hat{I}(T)$$

where $\hat{I} = \langle \mathcal{A}, M[x^i \mapsto I(t)] \rangle$.

Proof. This proof is based on a similar proof in [21].
We have to distinguish two cases:

- *Case:* $T = y^j$
  If $x^i = y^j$ then:

$$\mathit{subs}(T, x^i, t) = t \text{ and } \hat{I}(T) = I(t)$$
$$\text{Thus } I(\mathit{subs}(T, x^i, t)) = \hat{I}(T)$$

  If $x^i \neq y^j$ then:

$$\mathit{subs}(T, x^i, t) = T \text{ and } \hat{I}(T) = I(T)$$
$$\text{Thus } I(\mathit{subs}(T, x^i, t)) = \hat{I}(T)$$

- *Case:* $T = f(t_1, \ldots, t_n)$

$$\begin{aligned} I(subs(T, x^i, t)) &= I(f(subs(t_1, x^i, t), \ldots, subs(t_n, x^i, t))) \\ &= f^{\mathcal{A}}(I(subs(t_1, x^i, t)), \ldots, I(subs(t_n, x^i, t))) \end{aligned}$$

  By the hypothesis we get:

$$= f^{\mathcal{A}}(\hat{I}(t_1), \ldots, \hat{I}(t_n)) = \hat{I}(f(t_1, \ldots, t_n)) \qquad \square$$

Theorem 47. *Let i be a non-boolean sort, F be a formula, x^i a variable and t a term of sort i, such that $free(t) \cap bound(F) = \emptyset$. Then we have:*

$$I(subs(F, x^i, t) = \hat{I}(F)$$

where $\hat{I} = \langle \mathcal{A}, M[x^i \mapsto I(t)] \rangle$.

Proof. This proof is based on a proof in [21]. In the following we assume that the premise of the theorem is satisfied. We have to distinguish between the possible shapes of F:

- *Case:* $F = P(t_1, \ldots, t_n)$
  Can be proved similar to the second part of Lemma 46.

- *Case:* $F = F_1 \wedge_{\mathcal{L}} F_2$
  Since the premise of the theorem is fulfilled we get that:

$$\begin{aligned} free(t) \cap bound(F_1) &= \emptyset \\ free(t) \cap bound(F_2) &= \emptyset \end{aligned}$$

  This implies that by the hypothesis we can use the conclusion of the theorem for the subcases. We also know that:

$$subs(F_1 \wedge_{\mathcal{L}} F_2, x^i, t) = subs(F_1, x^i, t) \wedge_{\mathcal{L}} subs(F_2, x^i, t)$$

  therefore if $I \vDash subs(F_1 \wedge_{\mathcal{L}} F_2, x^i, t)$ then:

$$I \vDash subs(F_1, x^i, t) \text{ and } I \vDash subs(F_2, x^i, t)$$

  By the assumption we get:

$$\begin{aligned} \hat{I} \vDash F_1 \text{ and } \hat{I} \vDash F_2 \\ \text{Thus } \hat{I} \vDash F \end{aligned}$$

  If $I \nvDash subs(F_1 \wedge_{\mathcal{L}} F_2, x^i, t)$ we can reason similarly.

- *Case:* $F = \forall_{\mathcal{L}} y^j F_1$
  Since the premise of the theorem is satisfied it is also satisfied for the sub-case. So similar as in the previous case we get that we can use the conclusion of the theorem for the sub-case. If $x^i = y^j$ then $subs(\forall_{\mathcal{L}} x^i F_1, x^i, t) = \forall_{\mathcal{L}} x^i F_1$. Obviously $I(\forall_{\mathcal{L}} x^i F_1) = \hat{I}(\forall_{\mathcal{L}} x^i F_1)$. If $x^i \neq y^j$ then we know:

$$I \vDash \forall_{\mathcal{L}} y^j subs(F_1, x^i, t)$$
$$\Leftrightarrow \{y \in \mathcal{A}_j \mid \langle \mathcal{A}, M[y^j \mapsto y] \rangle \vDash subs(F_1, x^i, t)\} = \mathcal{A}_j$$
$$\hat{I} \vDash \forall_{\mathcal{L}} y^j F_1 \Leftrightarrow \{y \in \mathcal{A}_j \mid \langle \mathcal{A}, M[x^i \mapsto I(t), y^j \mapsto y] \rangle \vDash F_1\} = \mathcal{A}_j$$

  We fix $y \in \mathcal{A}_j$. By the hypothesis we get:

$$\langle \mathcal{A}, M[y^j \mapsto y] \rangle (subs(F_1, x^i, t)) = \langle \mathcal{A}, M[y^j \mapsto y, x^i \mapsto \langle \mathcal{A}, M[y^j \mapsto y] \rangle (t)] \rangle (F_1)$$

  Since $free(t) \cap bound(F) = \emptyset$ we get by Lemma 35 respectively Lemma 36:

$$\langle \mathcal{A}, M[y^j \mapsto y] \rangle (t) = \langle \mathcal{A}, M \rangle (t) = I(t)$$
$$\text{Thus } \langle \mathcal{A}, M[y^j \mapsto y] \rangle (subs(F_1, x^i, t)) = \langle \mathcal{A}, M[y^j \mapsto y, x^i \mapsto I(t)] \rangle (F_1)$$
$$\text{Thus } \{y \in \mathcal{A}_j \mid \langle \mathcal{A}, M[y^j \mapsto y] \rangle \vDash subs(F_1, x^i, t)\}$$
$$= \{y \in \mathcal{A}_j \mid \langle \mathcal{A}, M[x^i \mapsto I(t), y^j \mapsto y] \rangle \vDash F_1\}$$

  But if these sets are equal we get

$$I(subs(\forall_{\mathcal{L}} y^j F_1, x^i, t)) = \hat{I}(\forall_{\mathcal{L}} y^j F_1)$$

The other cases can be proven similar. □

Theorem 48. *Let Σ be a signature, F be a formula, $n \in \mathbb{N}$, $s, s_1, \cdots, s_n$ sorts in Σ, $x^s, z_1^{s_1}, \cdots z_n^{s_n}$ variables such that $\{z_1^{s_1}, \cdots z_n^{s_n}\} = free(\exists_{\mathcal{L}} x^s F)$. Moreover let $free(\exists_{\mathcal{L}} x^s F) \cap bound(F) = \emptyset$ and let f be a operation symbol such that $f \notin getOpSyms(F)$ and that the arity of f fits to $(s, s_1, \cdots, s_n)$. Then, for any structure $\mathcal{A}$ over Σ and any variable assignment M there is a structure $\mathcal{B}$ over Σ such that we have:*

$$\forall i \in Sort : \mathcal{A}_i = \mathcal{B}_i$$
$$\forall g \in Op_\Sigma \setminus \{f\} : g^{\mathcal{A}} = g^{\mathcal{B}}$$
$$\langle \mathcal{A}, M \rangle (\exists_{\mathcal{L}} x^s F) = \langle \mathcal{B}, M \rangle (subs(F, x^s, f(z_1, \ldots, z_n)))$$

Proof. The proof is inspired by a similar proof in [21]. Let $\mathcal{A}$ and M be arbitrary but fixed. We will give a structure $\mathcal{B}$ that fulfils the required properties. Thus let $\mathcal{B}$ be a structure over Σ that used the same universes as $\mathcal{A}$ and that assigns all operation symbols except f, to the same function as $\mathcal{A}$ does. We differentiate between two cases: If $\langle \mathcal{A}, M \rangle \vDash \exists_{\mathcal{L}} x^s F$ then there is a $y \in \mathcal{A}_s$ such that $\langle \mathcal{A}, M[x^s \mapsto y] \rangle \vDash F$. $\hat{x}$

shall denote such a value. Otherwise, if $\langle \mathcal{A}, M\rangle \not\models \exists_{\mathcal{L}} x^s F$ we set $\hat{x}$ to an arbitrary value from $\mathcal{A}_s$. For the interpretation of f we make the following requirement:

$$f^{\mathcal{B}}(\langle \mathcal{B}, M\rangle(z_1^{s_1}), \ldots, \langle \mathcal{B}, M\rangle(z_n^{s_n})) = \hat{x}.$$

For any other argument $f^{\mathcal{B}}$ shall be equal to $f^{\mathcal{A}}$. Now we have to show that $\mathcal{B}$ fulfils also the third property.

$$\begin{aligned}
&\langle \mathcal{B}, M\rangle(subs(F, x^s, f(z_1^{s_1}, \ldots, z_n^{s_n}))) \\
\text{By Theorem 47 } &= \langle \mathcal{B}, M[x^i \mapsto \langle \mathcal{B}, M\rangle(f(z_1^{s_1}, \ldots, z_n^{s_n})\rangle(F) \\
&= \langle \mathcal{B}, M[x^i \mapsto f^{\mathcal{B}}(\langle \mathcal{B}, M\rangle(z_1^{s_1}), \ldots, \langle \mathcal{B}, M\rangle(z_n^{s_n})\rangle(F) \\
\text{By Lemma 38 } &= \langle \mathcal{A}, M[x^s \mapsto f^{\mathcal{B}}(\langle \mathcal{B}, M\rangle(z_1^{s_1}), \ldots, \langle \mathcal{B}, M\rangle(z_n^{s_n})\rangle(F) \\
&= \langle \mathcal{A}, M[x^i \mapsto \hat{x}]\rangle(F)
\end{aligned}$$

The way we defined $\hat{x}$ implies that:

$$\langle \mathcal{A}, M\rangle \models (\exists_{\mathcal{L}} x^s F) \Leftrightarrow \langle \mathcal{A}, M[x^s \mapsto \hat{x}]\rangle \models F$$

This implies that

$$\langle \mathcal{A}, M\rangle(\exists_{\mathcal{L}} x^s F) = \langle \mathcal{B}, M\rangle(subs(F, x^s, f(z_1^{s_1}, \cdots, z_n^{s_n}))) \qquad \square$$

Remark This procedure of removing existential quantifiers is called *skolemisation*.

4 The Elimination of Quantifiers

In this chapter we will describe a procedure UNQUANTIFY that removes quantifiers from formulae from $\mathcal{L}$. We will show that for an appropriate theory T this procedure preserves satisfiability, i.e. a formula F is satisfiable modulo T iff UNQUANTIFY(F) is satisfiable modulo T.

In this chapter we will restrict the considered theories in the following way. Let Σ be the signature of $\mathcal{L}$. Then we assume that T is a theory such that for each interpretation $\langle \mathcal{A}, M \rangle \in T$ we have:

$$\forall s \in \Sigma.Sort : |\mathcal{A}_s| < \infty$$

Moreover we assume that we have a function

$$Constants : \Sigma.Sort \to \mathcal{P}(Op_\Sigma)$$

such that we have for each interpretation $\langle \mathcal{A}, M \rangle \in T$:

- $\forall s \in \Sigma.Sort : |Constants(s)| = |\mathcal{A}_s|$
- $\forall s \in \Sigma.Sort\ \forall v \in \mathcal{A}_s\ \exists c \in Constants(s)\ \forall I \in T : I(c) = v$

This means for each semantic value of a sort we can find a syntactic constant.

4.1 The Algorithm and its Correctness

In this section we will discuss the overall structure of the unquantification algorithm UNQUANTIFY. In order to do this we will use several sub-algorithms that will be introduced in the following sections. Moreover, we will prove the correctness of UNQUANTIFY under the assumption that the sub-algorithms are correct.

Description of Algorithm 1 In this algorithm we want to use skolemisation in order to remove existential quantifiers. The problem is that we cannot easily apply skolemisation to arbitrary formulae. Thus first we apply some preparatory steps. This we do in the first three steps, where we transform the initial formula to another formula, to which we can apply skolemisation. The first two steps deal with the problem of negated quantifiers. The problem with negated quantifiers is that we cannot skolemise negated existential quantifiers, but we can skolemise negated universal quantifiers. First we remove implications and equivalences. Then we can push negations inward. For both steps we use the logical equivalences for the involved

connectives. Besides the negated quantifiers we also have to deal with a second problem. If we apply skolemisation we substitute an existentially quantified variable by an unused function symbol applied to all free variables. It is now possible that one of these free variables is bounded somewhere in the formulae. We illustrate this with an example:

$$\forall_{\mathcal{L}} x^i (\exists_{\mathcal{L}} y^i (\forall_{\mathcal{L}} x^i P(x^i, y^i)) \wedge_{\mathcal{L}} Q(x^i, y^i))$$

If we skolemise the existential quantifier by substituting y^i with $f(x^i)$ we get:

$$\forall_{\mathcal{L}} x^i (\forall_{\mathcal{L}} x^i P(x^i, f(x^i))) \wedge_{\mathcal{L}} Q(x^i, f(x^i))$$

We can see that the variable x^i got bound in a sub-formula. To avoid this problem we rename the quantified variables in such a way that for each quantifier, the quantified variable does not get quantified again in a sub-formula. This renaming is done in the third step. In the fourth step we apply skolemisation to remove the existential quantifiers from all sub-formulae. The second argument of SKOL gives operation symbols that may not be used as skolem functions. Therefore, we pass the operation symbols part of the formulae and the operation symbols part of the theory T. Last but not least, we can remove the universal quantifiers by expanding them with constant symbols.

Algorithm 1 Unquantification of formulae

Input: A formula F_0
Require: $\mathit{free}(F_0) = \emptyset$
Output: A formula F_{out}
Ensure: $\mathit{free}(F_{out}) = \emptyset$.
Ensure: F_0 and F_{out} are equisatisfiable with respect to the theory T
Ensure: $\nexists p \in Pos(F_{out}), x^i \in \mathcal{V}, F' \in \mathcal{L} : F_{out}\langle p \rangle = \forall_{\mathcal{L}} x^i F'$
Ensure: $\nexists p \in Pos(F_{out}), x^i \in \mathcal{V}, F' \in \mathcal{L} : F_{out}\langle p \rangle = \exists_{\mathcal{L}} x^i F'$

```
1:  function UNQUANTIFY(F_0)
2:      ▷ Remove the implications and equivalences from F_0.
3:      F_1 ←REMIMPEQU(F_0)
4:      ▷ Push the negations in F_1 inward.
5:      F_2 ←SINKNEG(F_1)
6:      ▷ Rename the quantified variables in F_2.
7:      F_3 ←UNIQVARS(F_2, ∅, bound(F_2))
8:      ▷ Skolemise the existential quantifiers in F_3.
9:      (F_4, FS) ←SKOL(F_3, getOpSyms(F_3) ∪ Op_T)
10:     ▷ Expand the universal quantifiers in F_4.
11:     F_5 ←EXPALL(F_4)
12:     return F_5
13: end function
```

The sub-algorithms and their specifications, will be introduced in 4.2. The proofs for their correctness can be found in 4.3.

Theorem 49. *The preconditions of all called functions in* Unquantify *are satisfied.*

Proof. We assume that the postconditions of the called functions hold, if their preconditions are satisfied. Then we immediately get the conclusion of the theorem by comparing the pre- and postconditions of the respective functions. □

Theorem 50. Unquantify *terminates.*

Proof. All the called sub-procedures terminate, so we can conclude that the procedure itself terminates. □

Theorem 51. *Let F be a closed formula. Then* Unquantify(F) *satisfies its post-conditions.*

Proof. From the postcondition of ExpAll we can conclude that Unquantify(F) does not contain any quantifiers. As in the precondition we require that F_0 is a closed formula we can conclude from the specifications of the sub-procedure that $F_1, \cdots, F_5$ are also closed formulae. This means that the resulting formula is closed.

We still have to show that F_0 and F_5 are equisatisfiable with respect to the theory T. From the postcondition of RemImpEqu we get that $F_0 \equiv F_1$. From the postcondition of SinkNeg we get that $F_1 \equiv F_2$. From the postcondition of UniQVars we get that $F_2 \equiv F_3$. Transitivity of semantic equivalence implies that $F_0 \equiv F_3$. We have seen that semantic equivalence implies equisatisfiability (Lemma 27). From this we can conclude that F_0 and F_3 are T-equisatisfiable. From the postcondition of Skol we get that F_3 and F_4 are T-equisatisfiable. Equisatisfiability with respect to a theory is a transitive relation. By transitivity, we can conclude that F_0 is T-equisatisfiale to F_4. By the postcondition of ExpAll we get that F_4 is T-equisatisfiale to F_5. Thus F_0 is T-equisatisfiale to F_5 □

Besides the above algorithm we can also consider the alternative unquantification algorithm $\text{ExpAll}_{\text{Expand}}$. In this algorithm we eliminate both existantial and universal quantifiers by means of quantifier expansion. For this algorithm, we need a procedure ExpExists which eliminates existential quantifiers by expansion. Additionally, we also have to define a generalised version of ExpAll. This generalised procedure must allow input formulae which may contain implications and equivalences and negations in arbitrary positions. As these procedures can be defined similarly to the procedures used in Unquantify we will not define them.

The proofs for the correctness for the algorithm and for the sub-algorithms work similar to the proofs necessary for Unquantify, thus we will not present them.

4.2 The Sub-Algorithms

In this section we will introduce the sub-algorithms, which we used in Algorithm 1. The proofs for their correctness can be found in 4.3.

First we will describe a procedure to remove implications and equivalences from an arbitrary formula.

Algorithm 2 Unquantification of formulae without skolemisation

Input: A formula F_0
Require: $\mathit{free}(F_0) = \emptyset$
Output: A formula F_{out}
Ensure: $\mathit{free}(F_{out}) = \emptyset$.
Ensure: $F_0 \equiv F_{out}$
Ensure: $\nexists p \in Pos(F_{out}), x^i \in \mathcal{V}, F' \in \mathcal{L} : F_{out}\langle p\rangle = \forall_{\mathcal{L}} x^i F'$
Ensure: $\nexists p \in Pos(F_{out}), x^i \in \mathcal{V}, F' \in \mathcal{L} : F_{out}\langle p\rangle = \exists_{\mathcal{L}} x^i F'$
1: **function** UNQUANTIFY$_{\text{EXPAND}}(F_0)$
2: ▷ Expand the existential quantifiers in F_0.
3: $F_1 \leftarrow$EXPEXISTS(F_0)
4: ▷ Expand the universal quantifiers in F_1.
5: $F_2 \leftarrow$EXPALL$_{\text{EXPAND}}(F_1)$
6: **return** F_2
7: **end function**

Description of Algorithm 3 Since a formula from $\mathcal{L}$ can have various shapes, we have to consider all of them. This is done by the select case command. In the cases of implication and equivalence, we use the properties of these connectives to remove them and apply the function to the sub-cases. In all the other cases, implication and equivalence can only occur in the proper sub-formulae. Therefore, we just apply the procedure to the sub-formulae.

Next we define a procedure to push negations inward. This means that after this step, negations shall only be bound to predicates.

Description of Algorithm 4 As in the previous algorithm, we deal with the various cases of formulae by a select case command. Because of the precondition we do not have to deal with implications and equivalences. In all the cases without negations, we can simply apply the procedure to the sub-formulae. In the other cases, we make use of the properties of the logical connectives to push the negations a level inwards.

The procedure we define in following will rename quantified variables. After the renaming for every quantifier, the quantified variable shall not be used for another quantification inside of the quantified expression.

Description of Algorithm 5 Similar as in the previous case we distinguish between the possible shapes of formulae. The cases which do not contain any quantifiers work straight forward, so we will only deal with the cases where a quantifier occurs. For both existential and universal quantifier there are two cases: either the quantified variable was already used ($x^i \in V_1$) or it was not ($x^i \notin V_1$). In the first case we substitute, in the quantified formula, the used variable by a fresh variable. Then we proceed with the next level of the formula by applying the procedure to the

Algorithm 3

Input: A formula F
Output: A formula F_{out}
Ensure: $F \equiv F_{out}$ (Theorem 53)
Ensure: $\mathit{free}(F) = \emptyset \Rightarrow \mathit{free}(F_{out}) = \emptyset$ (Corollary 55)
Ensure: $\nexists p \in \mathit{Pos}(F_{out})\ \exists F_1, F_2 \in \mathcal{L} : F_{out}\langle p \rangle = F_1 \Rightarrow_{\mathcal{L}} F_2$ (Lemma 56)
Ensure: $\nexists p \in \mathit{Pos}(F_{out})\ \exists F_1, F_2 \in \mathcal{L} : F_{out}\langle p \rangle = F_1 \Leftrightarrow_{\mathcal{L}} F_2$ (Lemma 57)

1: **function** RemImpEqu(F)
2: **select case in** F
3: **Case** $F = F_1 \Rightarrow_{\mathcal{L}} F_2$
4: **return** $\neg_{\mathcal{L}}$RemImpEqu(F_1) $\vee_{\mathcal{L}}$ RemImpEqu(F_2)
5: **Case** $F = F_1 \Leftrightarrow_{\mathcal{L}} F_2$
6: **return** ($\neg_{\mathcal{L}}$RemImpEqu(F_1) $\vee_{\mathcal{L}}$ RemImpEqu(F_2))$\wedge_{\mathcal{L}}$ ($\neg_{\mathcal{L}}$RemImpEqu(F_2) $\vee_{\mathcal{L}}$ RemImpEqu(F_1))
7: **Case** $F = F_1 \wedge_{\mathcal{L}} F_2$
8: **return** RemImpEqu(F_1) $\wedge_{\mathcal{L}}$ RemImpEqu(F_2)
9: **Case** $F = F_1 \vee_{\mathcal{L}} F_2$
10: **return** RemImpEqu(F_1) $\vee_{\mathcal{L}}$ RemImpEqu(F_2)
11: **Case** $F = \neg_{\mathcal{L}} F_1$
12: **return** $\neg_{\mathcal{L}}$RemImpEqu(F_1)
13: **Case** $F = \exists_{\mathcal{L}} x^i F_1$
14: **return** $\exists_{\mathcal{L}} x^i$RemImpEqu($F_1$)
15: **Case** $F = \forall_{\mathcal{L}} x^i F_1$
16: **return** $\forall_{\mathcal{L}} x^i$RemImpEqu($F_1$)
17: **Case** $F = P(t_1, \ldots, t_n)$
18: **return** F
19: **end select**
20: **end function**

Algorithm 4

Input: A formula F
Require: $\nexists p \in Pos(F)\ \exists F_1, F_2 \in \mathcal{L} : F\langle p \rangle = F_1 \Rightarrow_{\mathcal{L}} F_2$
Require: $\nexists p \in Pos(F)\ \exists F_1, F_2 \in \mathcal{L} : F\langle p \rangle = F_1 \Leftrightarrow_{\mathcal{L}} F_2$
Output: A formula F_{out}
Ensure: $F \equiv F_{out}$ (Theorem 60)
Ensure: $free(F) = \emptyset \Rightarrow free(F_{out}) = \emptyset$ (Corollary 62)
Ensure: $\nexists p \in Pos(F_{out})\ \exists F_1, F_2 \in \mathcal{L} : F_{out}\langle p \rangle = F_1 \Rightarrow_{\mathcal{L}} F_2$ (Lemma 64)
Ensure: $\nexists p \in Pos(F_{out})\ \exists F_1, F_2 \in \mathcal{L} : F_{out}\langle p \rangle = F_1 \Leftrightarrow_{\mathcal{L}} F_2$ (Lemma 65)
Ensure: $\forall p \in Pos(F_{out})\ \forall F' \in \mathcal{L} : F_{out}\langle p \rangle = \neg_{\mathcal{L}} F' \Rightarrow F' = P(t_1, \ldots, t_n)$ (Lemma 63)

1: **function** SinkNeg(F)
2: **select case in** F
3: **Case** $F = F_1 \wedge_{\mathcal{L}} F_2$
4: **return** SinkNeg(F_1) $\wedge_{\mathcal{L}}$ SinkNeg(F_2)
5: **Case** $F = \neg_{\mathcal{L}}(F_1 \wedge_{\mathcal{L}} F_2)$
6: **return** SinkNeg($\neg_{\mathcal{L}} F_1$) $\vee_{\mathcal{L}}$ SinkNeg($\neg_{\mathcal{L}} F_2$)
7: **Case** $F = F_1 \vee_{\mathcal{L}} F_2$
8: **return** SinkNeg(F_1) $\vee_{\mathcal{L}}$ SinkNeg(F_2)
9: **Case** $F = \neg_{\mathcal{L}}(F_1 \vee_{\mathcal{L}} F_2)$
10: **return** SinkNeg($\neg_{\mathcal{L}} F_1$) $\wedge_{\mathcal{L}}$ SinkNeg($\neg_{\mathcal{L}} F_2$)
11: **Case** $F = \neg_{\mathcal{L}}(\neg_{\mathcal{L}} F_1)$
12: **return** SinkNeg(F_1)
13: **Case** $F = \exists_{\mathcal{L}} x^i F_1$
14: **return** $\exists_{\mathcal{L}} x^i$SinkNeg($F_1$)
15: **Case** $F = \neg_{\mathcal{L}}(\exists_{\mathcal{L}} x^i F_1)$
16: **return** $\forall_{\mathcal{L}} x^i$SinkNeg($\neg_{\mathcal{L}} F_1$)
17: **Case** $F = \forall_{\mathcal{L}} x^i F_1$
18: **return** $\forall_{\mathcal{L}} x^i$SinkNeg($F_1$)
19: **Case** $F = \neg_{\mathcal{L}}(\forall_{\mathcal{L}} x^i F_1)$
20: **return** $\exists_{\mathcal{L}} x^i$SinkNeg($\neg_{\mathcal{L}} F_1$)
21: **Case** $F = P(t_1, \ldots, t_n)$
22: **return** F
23: **Case** $F = \neg_{\mathcal{L}} P(t_1, \ldots, t_n)$
24: **return** F
25: **end select**
26: **end function**

Algorithm 5

Input: A formula F, two sets of variables V_1 and V_2
Require: $\nexists p \in Pos(F)\ \exists F_1, F_2 \in \mathcal{L} : F\langle p\rangle = F_1 \Rightarrow_{\mathcal{L}} F_2$
Require: $\nexists p \in Pos(F)\ \exists F_1, F_2 \in \mathcal{L} : F\langle p\rangle = F_1 \Leftrightarrow_{\mathcal{L}} F_2$
Require: $\forall p \in Pos(F)\ \forall F' \in \mathcal{L} : F\langle p\rangle = \neg_{\mathcal{L}} F' \Rightarrow F' = P(t_1, \ldots, t_n)$
Require: $free(F) \subseteq V_1$
Require: $bound(F) \subseteq V_2$
Output: A formula F_{out}
Ensure: $F \equiv F_{out}$ (Theorem 77)
Ensure: $V_1 = \emptyset \Rightarrow free(F_{out}) = \emptyset$ (Corollary 71)
Ensure: $\nexists p \in Pos(F_{out})\ \exists F_1, F_2 \in \mathcal{L} : F_{out}\langle p\rangle = F_1 \Rightarrow_{\mathcal{L}} F_2$ (Lemma 75)
Ensure: $\nexists p \in Pos(F_{out})\ \exists F_1, F_2 \in \mathcal{L} : F_{out}\langle p\rangle = F_1 \Leftrightarrow_{\mathcal{L}} F_2$ (Lemma 76)
Ensure: $\forall p \in Pos(F_{out})\ \forall F' \in \mathcal{L} : F_{out}\langle p\rangle = \neg_{\mathcal{L}} F' \Rightarrow F' = P(t_1, \ldots, t_n)$ (Lemma 74)
Ensure: $\forall p \in Pos(F_{out}) : bound(F_{out}\langle p\rangle) \cap free(F_{out}\langle p\rangle) = \emptyset$ (Corollary 73)

1: **function** UniQVars(F, V_1, V_2)
2: **select case in** F
3: **Case** $F = F_1 \wedge_{\mathcal{L}} F_2$
4: **return** UniQVars(F_1, V_1, V_2)$\wedge_{\mathcal{L}}$ UniQVars(F_1, V_1, V_2)
5: **Case** $F = F_1 \vee_{\mathcal{L}} F_2$
6: **return** UniQVars(F_1, V_1, V_2)$\vee_{\mathcal{L}}$ UniQVars(F_1, V_1, V_2)
7: **Case** $F = \neg_{\mathcal{L}} F_1$
8: **return** F
9: **Case** $F = \exists_{\mathcal{L}} x^i F_1 \wedge x^i \notin V_1$
10: **return** $\exists_{\mathcal{L}} x^i$UniQVars($F_1, V_1 \cup \{x^i\}, V_2$)
11: **Case** $F = \exists_{\mathcal{L}} x^i F_1 \wedge x^i \in V_1$
12: $y^i \leftarrow$ choose $\mathcal{V}_i \setminus (V_1 \cup V_2)$
13: **return** $\exists_{\mathcal{L}} y^i$UniQVars($subs(F_1, x^i, y^i), V_1 \cup \{y^i\}, V_2$)
14: **Case** $F = \forall_{\mathcal{L}} x^i F_1 \wedge x^i \notin V_1$
15: **return** $\forall_{\mathcal{L}} x^i$UniQVars($F_1, V_1 \cup \{x^i\}, V_2$)
16: **Case** $F = \forall_{\mathcal{L}} x^i F_1 \wedge x^i \in V_1$
17: $y^i \leftarrow$ choose $\mathcal{V}_i \setminus (V_1 \cup V_2)$
18: **return** $\forall_{\mathcal{L}} y^i$UniQVars($subs(F_1, x^i, y^i), V_1 \cup \{y^i\}, V_2$)
19: **Case** $F = P(t_1, \ldots, t_n)$
20: **return** F
21: **end select**
22: **end function**

substituted formula. In the other case we do not have to change the variable, so we can simply apply the procedure to the sub-formula. In this algorithm we use the sets V_1 and V_2 to ensure that we actually use a fresh variable, when we make the substitution.

In the following we will define a procedure that removes the existential quantifiers by skolemising them.

Description of Algorithm 6 As before we distinct between the possible shapes of a formula. In the following we will describe two of these cases. The cases that are not described are either straight forward or similar to cases we actually described.

- $F = F_1 \wedge_{\mathcal{L}} F_2$ For skolemisation we must use fresh operation symbols, when we make the substitutions. This means that we have to make sure that no operation symbol is introduced in both sub-cases. We start with skolemising the first sub-formula. The second component, of the result contains all operation symbols from the original formula and also all the new ones. So we use this set of operation symbols as the second argument for the second sub-case. This ensures that only fresh operation symbols are introduced.

- $F = \exists_{\mathcal{L}} x^i F_1$ We first generate a sequence *vars*, which consist of the free variables of F. We then choose a new operation symbol f (the way we choose it guarantees that it is fresh). For simplicity, we assume that we have a signature such that $\mathit{Func}(f)$ fits to the sorts of the free variables. This means that $\mathit{Func}(f)_1 = i$ and the remaining parts of $\mathit{Func}(f)$ fit to the sorts of *vars*. Therefore we are allowed to substitute the quantified variable by $f(\mathit{vars})$. Last but not least, we go to the next level of the formula by applying the procedure to the substituted formula.

Finally we will now define a procedure which removes the universal quantifiers, by expanding them. This means that in all universal quantifiers we will substitute the quantified formula with certain constants. These substituted formulae we will then conjoin.

Description of Algorithm 7 Similar to the previous sub-algorithms again we have to differentiate between the possible shapes of the formula. We will only deal with the case where a universal quantifier occurs, the others are straight forward. In this case we first expand the quantified formula. In the expanded formula we then substitute the quantified variable with the constant symbols that are associated to the sort of the variable. Finally we return the conjunction of these substituted formulae.

Algorithm 6 Skolemisation

Input: A formula F and a set of operation symbols FS
Require: $\nexists p \in Pos(F)\ \exists F_1, F_2 \in \mathcal{L} : F\langle p\rangle = F_1 \Rightarrow_{\mathcal{L}} F_2$
Require: $\nexists p \in Pos(F)\ \exists F_1, F_2 \in \mathcal{L} : F\langle p\rangle = F_1 \Leftrightarrow_{\mathcal{L}} F_2$
Require: $\forall p \in Pos(F)\ \forall F' \in \mathcal{L} : F\langle p\rangle = \neg_{\mathcal{L}} F' \Rightarrow F' = P(t_1, \ldots, t_n)$
Require: $\forall p \in Pos(F) : bound(F\langle p\rangle) \cap free(F\langle p\rangle) = \emptyset$
Require: $getOpSyms(F) \subseteq FS$
Output: A formula F_{out} and a set of operation symbols FS_{out}
Ensure: $free(F) = \emptyset \Rightarrow free(F_{out}) = \emptyset$ (Corollary 89)
Ensure: For any variable assignment M we have that there is a T-structure $\mathcal{A}$ such that $\langle \mathcal{A}, M\rangle \vDash F$ iff there is a T-structure $\mathcal{B}$ such that $\langle \mathcal{B}, M\rangle \vDash F_{out}$ (Theorem 86)
Ensure: $\nexists p \in Pos(F_{out})\ \exists F_1, F_2 \in \mathcal{L} : F_{out}\langle p\rangle = F_1 \Rightarrow_{\mathcal{L}} F_2$ (Lemma 91)
Ensure: $\nexists p \in Pos(F_{out})\ \exists F_1, F_2 \in \mathcal{L} : F_{out}\langle p\rangle = F_1 \Leftrightarrow_{\mathcal{L}} F_2$ (Lemma 92)
Ensure: $\forall p \in Pos(F_{out})\ \forall F' \in \mathcal{L} : F_{out}\langle p\rangle = \neg_{\mathcal{L}} F' \Rightarrow F' = P(t_1, \ldots, t_n)$ (Lemma 90)
Ensure: $\nexists p \in Pos(F_{out})\ \exists x^i \in \mathcal{V}\ \exists F' \in \mathcal{L} : F_{out}\langle p\rangle = \exists_{\mathcal{L}} x^i F'$ (Lemma 93)
Ensure: $getOpSyms(F_{out}) \subseteq FS_{out}$ (Lemma 81)

```
1:  function SKOL(F, FS)
2:      select case in F
3:          Case F = F_1 ∧_L F_2
4:              ψ ← SKOL(F_1, FS)
5:              χ ← SKOL(F_2, ψ_2)
6:              return (ψ_1 ∧_L χ_1, χ_2)
7:          Case F = F_1 ∨_L F_2
8:              ψ ← SKOL(F_1, FS)
9:              χ ← SKOL(F_2, ψ_2)
10:             return (ψ_1 ∨_L χ_1, χ_2)
11:         Case F = ¬_L F_1
12:             return (F, FS)
13:         Case F = ∃_L x^i F_1
14:             S ← free(F)
15:             vars ← [ ]
16:             while S ≠ ∅ do
17:                 v ← choose S
18:                 vars ← [v, vars]
19:                 S ← S \ {v}
20:             end while
21:             f ← choose(OS \ FS)
22:             return SKOL(subs(F_1, x^i, f(vars)), FS ∪ {f})
```

Algorithm 6 Continuation

```
23:         Case F = ∀_L x^i F_1
24:             ψ ← Skol(F_1, FS)
25:             return (∀_L x^i ψ_1, ψ_2)
26:         Case F = P(t_1, ..., t_n)
27:             return F
28:     end select
29: end function
```

Algorithm 7 Expansion of universal quantifiers

Input: A formula F
Require: $\nexists p \in Pos(F)\ \exists F_1, F_2 \in \mathcal{L} : F\langle p\rangle = F_1 \Rightarrow_{\mathcal{L}} F_2$
Require: $\nexists p \in Pos(F)\ \exists F_1, F_2 \in \mathcal{L} : F\langle p\rangle = F_1 \Leftrightarrow_{\mathcal{L}} F_2$
Require: $\forall p \in Pos(F)\ \forall F' \in \mathcal{L} : F\langle p\rangle = \neg_{\mathcal{L}} F' \Rightarrow F' = P(t_1, \ldots, t_n)$
Require: $\nexists p \in Pos(F)\ \exists x^i \in \mathcal{V}\ \exists F' \in \mathcal{L} : F\langle p\rangle = \exists_{\mathcal{L}} x^i F'$
Output: A formula F_{out}
Ensure: $free(F) = \emptyset \Rightarrow free(F_{out}) = \emptyset$ (Lemma 96)
Ensure: For any T-interpretation I we have that: $I(F) = I(F_{out})$ (Theorem 98)
Ensure: $\nexists p \in Pos(F_{out})\ \exists x^i \in \mathcal{V}\ \exists F' \in \mathcal{L} : F_{out}\langle p\rangle = \forall_{\mathcal{L}} x^i F'$ (Lemma 97)
Ensure: $\nexists p \in Pos(F_{out})\ \exists x^i \in \mathcal{V}\ \exists F' \in \mathcal{L} : F_{out}\langle p\rangle = \exists_{\mathcal{L}} x^i F'$ (Lemma 97)

```
1: function ExpAll(F)
2:     select case in F
3:         Case F = F_1 ∧_L F_2
4:             return ExpAll(F_1) ∧_L ExpAll(F_2)
5:         Case F = F_1 ∨_L F_2
6:             return ExpAll(F_1) ∨_L ExpAll(F_2)
7:         Case F = ¬_L F_1
8:             return F
9:         Case F = ∀_L x^i F_1
10:            F_sub ← ExpAll(F_1)
11:            F_Res ← ⋀_{c ∈ Constants(i)}^{L} subs(F_sub, x^i, c)
12:            return F_Res
13:        Case F = P(t_1, ..., t_n)
14:            return F
15:    end select
16: end function
```

4.3 The Correctness of the Subalgorithms

In this section, we use the auxiliary results introduced above to actually prove the correctness of the various subalgorithms.

4.3.1 RemImpEqu

First we show that the algorithm terminates, then that its postconditions hold.

Theorem 52. RemImpEqu *terminates.*

Proof. In order to prove the termination we give a termination measure. We define this measure as:

$$T(F) \coloneqq |Pos(F)|$$

We now show, if in RemImpEqu(F), RemImpEqu(F') with some formula F' is called then $T(F') < T(F)$ holds. We prove this by considering all possible shapes of F.

- *Case:* $F = P(t_1, \ldots, t_n)$
  In this case there is no recursive call of RemImpEqu.

- *Case:* $F = F_1 \wedge_{\mathcal{L}} F_2$
  From the definition of positions it immediately follows that:

$$Pos(F_1 \wedge_{\mathcal{L}} F_2) = \{1p \mid p \in Pos(F_1)\} \cup \{2p \mid p \in Pos(F_2)\} \cup \{\varepsilon\}$$
$$\text{But this impies: } |F| = |Pos(F_1)| + |Pos(F_2)| + |\{\varepsilon\}|$$
$$\text{Thus } T(F_1) < T(F) \wedge T(F_2) < T(F)$$

- *Case:* $F = \exists_{\mathcal{L}} x^i F_1$
  We proceed similarly as in the previous case.

$$Pos(\exists_{\mathcal{L}} x^i F_1) = \{1p \mid p \in Pos(F_1)\} \cup \{\varepsilon\}$$
$$\Rightarrow |Pos(\exists_{\mathcal{L}} x^i F_1)| = |Pos(F_1)| + 1$$
$$\text{Thus } T(F_1) < T(F)$$

The other cases can be proven similar. Since the cardinality of sets is bounded from below, also T is bounded from below. So RemImpEqu terminates. □

Theorem 53. *Let F be a formula. Then $F \equiv$* RemImpEqu(F).

Proof. We prove this by considering all possible shapes of F.

- *Case:* $F = P(t_1, \ldots, t_n)$
  RemImpEqu(F) $= F \Rightarrow$ RemImpEqu(F) $\equiv F$

- *Case:* $F = F_1 \Rightarrow_{\mathcal{L}} F_2$
  The hypothesis implies: $\textsc{RemImpEqu}(F_1) \equiv F_1$ and $\textsc{RemImpEqu}(F_2) \equiv F_2$.

$$\textsc{RemImpEqu}(F) = \neg_{\mathcal{L}} \textsc{RemImpEqu}(F_1) \vee_{\mathcal{L}} \textsc{RemImpEqu}(F_2)$$
$$\text{By the assumption and Lemma 32: } \equiv \neg_{\mathcal{L}} F_1 \vee_{\mathcal{L}} F_2 \equiv F$$

The other cases can be proved similarly. □

Lemma 54. *Let F be a formula. Then $\mathit{free}(F) = \mathit{free}(\textsc{RemImpEqu}(F))$.*

Proof. We prove this by considering all possible shapes of F.

- *Case:* $F = P(t_1, \ldots, t_n)$
  $\textsc{RemImpEqu}(F) = F \Rightarrow \mathit{free}(F) = \mathit{free}(\textsc{RemImpEqu}(F))$
- *Case:* $F = F_1 \Rightarrow_{\mathcal{L}} F_2$
  By using the properties of *free* we get:

$$\mathit{free}(\textsc{RemImpEqu}(F))$$
$$= \mathit{free}(\textsc{RemImpEqu}(F_1)) \cup \mathit{free}(\textsc{RemImpEqu}(F_2))$$

  By the hypothesis we get:

$$\mathit{free}(\textsc{RemImpEqu}(F)) = \mathit{free}(F_1) \cup \mathit{free}(F_2) = \mathit{free}(F)$$

- *Case:* $\forall_{\mathcal{L}} x^i F_1$

$$\mathit{free}(\textsc{RemImpEqu}(F)) = \mathit{free}(\textsc{RemImpEqu}(F_1)) \setminus \{x^i\}$$
$$= \mathit{free}(F_1) \setminus \{x^i\} = \mathit{free}(F)$$

The other cases can be proved quite similarly. □

Corollary 55. *Let F be a formula. Then we have:*

$$\mathit{free}(F) = \emptyset \Rightarrow \mathit{free}(\textsc{RemImpEqu}(F)) = \emptyset$$

Proof. Follows directly from Lemma 54. □

Lemma 56. *Let F be a formula. Then we have:*

$$\nexists p \in \mathit{Pos}(\textsc{RemImpEqu}(F)), F_1, F_2 \in \mathcal{L} : \textsc{RemImpEqu}(F)\langle p \rangle = F_1 \Rightarrow_{\mathcal{L}} F_2$$

Proof. We prove this by considering all possible shapes of F.

- *Case:* $F = P(t_1, \ldots, t_n)$
  In this case we have $\mathit{Pos}(F) = \{\varepsilon\}$. So we have to show that the sub-formula at position ε is no implication.

$$\textsc{RemImpEqu}(F)\langle \varepsilon \rangle = \textsc{RemImpEqu}(F)$$
$$= P(t_1, \ldots, t_n) \neq F_1 \Rightarrow_{\mathcal{L}} F_2 \text{ for any formulae } F_1, F_2$$

- *Case:* $F = F_1 \Rightarrow_{\mathcal{L}} F_2$
  To begin we show that the sub-formula at position ε is no implication.

$$\begin{aligned} &\textsc{RemImpEqu}(F)\langle\varepsilon\rangle = \textsc{RemImpEqu}(F) \\ &= \neg_{\mathcal{L}}\textsc{RemImpEqu}(F_1) \vee_{\mathcal{L}} \textsc{RemImpEqu}(F_2) \\ &\neq F_1 \Rightarrow_{\mathcal{L}} F_2 \text{ for any formulae } F_1, F_2. \end{aligned}$$

  Now fix $p \in Pos(\textsc{RemImpEqu}(F)) \setminus \{\varepsilon\}$. We either have $p = 1p_1$ or $p = 2p_2$, where $p_1 \in Pos(\textsc{RemImpEqu}(F))$ and $p_2 \in Pos(\textsc{RemImpEqu}(F))$ .Let $p = 1p_1$, the other case can be proved similar but it is easier.

$$\textsc{RemImpEqu}(F)\langle p\rangle = \neg_{\mathcal{L}}\textsc{RemImpEqu}(F_1)\langle p_1\rangle$$

  By the hypothesis this formula does not contain any implications.

The other cases can be proved similarly. □

Lemma 57. *Let F be a formula. Then we have:*

$$\nexists p \in Pos(\textsc{RemImpEqu}(F)) : \exists F_1, F_2 : \textsc{RemImpEqu}(F)\langle p\rangle = F_1 \Leftrightarrow_{\mathcal{L}} F_2$$

Proof. Can be proved analogously like Lemma 56, we just have to consider $\Leftrightarrow_{\mathcal{L}}$ instead of $\Rightarrow_{\mathcal{L}}$. □

4.3.2 SinkNeg

We first show that the preconditions of the function hold in every recursive call. Then we show that the algorithm terminates. Finally, we show that the postconditions hold.

Theorem 58. *Let F be a formula such F fulfils the preconditions of* SinkNeg. *Then also the preconditions for all recursive calls are fulfilled.*

Proof. We only prove that this holds for the first precondition. The second one can be handled analogously. In the following we assume that the precondition holds for F. We prove this theorem by considering all possible shapes of F.

- *Case:* $F = P(t_1, \ldots, t_n)$
  No recursive call of SinkNeg.

- *Case:* $F = F_1 \wedge_{\mathcal{L}} F_2$
  We make a proof by contradiction. We assume that for one of the recursive calls the precondition is not fulfilled. W.l.o.g. we assume that the precondition is not fulfilled for F_1. Because of the assumption we know that:

$$\exists p \in Pos(F_1)\ \exists \hat{F}, \tilde{F} : F_1\langle p\rangle = \hat{F} \Rightarrow_{\mathcal{L}} \tilde{F}$$

  Let $\hat{p}$ be such a position then:

$$\exists \hat{F}, \tilde{F} : F\langle 1\hat{p}\rangle = \hat{F} \Rightarrow_{\mathcal{L}} \tilde{F}$$

  But this is a contradiction to the assumption that F satisfies the precondition.

- *Case:* $F = \neg_{\mathcal{L}}(F_1 \wedge_{\mathcal{L}} F_2)$
  Assume there are formulae $\hat{F}, \tilde{F}$ and a position $p \in Pos(F_1)$ such that $F_1\langle p\rangle = \hat{F} \Rightarrow_{\mathcal{L}} \tilde{F}$ (obviously $p \neq \varepsilon$). But then $11p \in Pos(F)$ and $F\langle 11p\rangle = \hat{F} \Rightarrow_{\mathcal{L}} \tilde{F}$. This is a contradiction to the assumption that F satisfies the precondition.

The other cases can be proved similarly. □

Theorem 59. SinkNeg *terminates.*

Proof. Can be proved analogously to Theorem 52. □

Theorem 60. *Let F be a formula that satisfied the precondition of* SinkNeg. *Then we have:*

$$F \equiv \textsc{SinkNeg}(F)$$

Proof. To prove this lemma we consider all the possible shapes of F. In the following, we assume that the assertion holds for the sub-cases.

- *Case:* $F = P(t_1, \ldots, t_n)$
  $\textsc{SinkNeg}(F) = F \Rightarrow \textsc{SinkNeg}(F) \equiv F$. This proves the assertion for this case.

- *Case:* $F = F_1 \wedge_{\mathcal{L}} F_2$
  $\textsc{SinkNeg}(F) = \textsc{SinkNeg}(F_1) \wedge_{\mathcal{L}} \textsc{SinkNeg}(F_2) \equiv F_1 \wedge_{\mathcal{L}} F_2 = F$

- *Case:* $F = \neg_{\mathcal{L}}(F_1 \wedge_{\mathcal{L}} F_2)$
  Obviously $|Pos(\neg_{\mathcal{L}} F_1)| < |Pos(F)|$ so we can use the induction hypothesis. We can conclude that:

$$\begin{aligned}\textsc{SinkNeg}(F) = \textsc{SinkNeg}(\neg_{\mathcal{L}} F_1) \vee_{\mathcal{L}} \textsc{SinkNeg}(\neg_{\mathcal{L}} F_2) \equiv \\ \equiv \neg_{\mathcal{L}} F_1 \vee_{\mathcal{L}} \neg_{\mathcal{L}} F_2 \equiv F\end{aligned}$$

- *Case:* $F = \neg_{\mathcal{L}}(\exists_{\mathcal{L}} x^i F_1)$
  By the hypothesis we get $\textsc{SinkNeg}(\neg_{\mathcal{L}} F_1) \equiv F_1$. Also in this logic the well-known property $\neg_{\mathcal{L}}(\exists_{\mathcal{L}} x^i F_1) \equiv (\forall_{\mathcal{L}} x^i \neg_{\mathcal{L}} F_1)$ holds. So it follows that

$$\textsc{SinkNeg}(F) \equiv F.$$

All other cases can be proved similarly. □

Lemma 61. *Let F be a formula fulfilling the precondition of* SinkNeg. *Then we have:*

$$\mathit{free}(F) = \mathit{free}(\textsc{SinkNeg}(F)).$$

Proof. This lemma can be proved similarly as Lemma 54 □

Corollary 62. *Let F be a formula fulfilling the precondition of* SinkNeg. *Then we have:*

$$\mathit{free}(F) = \emptyset \Rightarrow \mathit{free}(\textsc{SinkNeg}(F)) = \emptyset$$

Proof. Follows directly from Lemma 61. □

Remark In the next lemma we will make use of the following meta-predicate:

$$Post(F) \Leftrightarrow (\forall \hat{F}, p \in Pos(\textsc{SinkNeg}(F)) : (\textsc{SinkNeg}(F)\langle p\rangle = \neg_{\mathcal{L}}\hat{F}) \Rightarrow \\ \Rightarrow \exists P, t_1, \cdots, t_n : \hat{F} = P(t_1, \ldots, t_n))$$

Lemma 63. *Let F be a formula fulfilling the precondition of* SinkNeg. *Then we have $Post(F)$.*

Proof. To prove this lemma we consider all the possible shapes of F. We assume that F fulfils the premise of the lemma and that the conclusion of the lemma holds for all the sub-cases.

- *Case:* $F = P(t_1, \ldots, t_n)$
  Obvious.
- *Case:* $F = F_1 \wedge_{\mathcal{L}} F_2$
  Obviously there is no formula $\hat{F}$ such that $\textsc{SinkNeg}(F)\langle\varepsilon\rangle = \neg_{\mathcal{L}}\hat{F}$. Let us now assume that:

  $$\exists \hat{F}, p \in Pos(\textsc{SinkNeg}(F)) \setminus \{\varepsilon\} : \textsc{SinkNeg}(F)\langle p\rangle = \neg_{\mathcal{L}}\hat{F}$$

  Let be p be such a position and $\hat{F}$ such a formula which is not the application of a predicate. Then either $p = 1p_1$ or $p = 2p_2$. W.l.o.g. we assume that $p = 1p_1$. But since $p_1 \in \textsc{SinkNeg}(F_1)$ this is a contradiction to the hypothesis as $\textsc{SinkNeg}(F_1)\langle p_1\rangle = \neg_{\mathcal{L}}\hat{F}$.
- *Case:* $F = \neg_{\mathcal{L}}(F_1 \wedge_{\mathcal{L}} F_2)$
  Assume $p \in Pos(\textsc{SinkNeg}(F))$ and $\textsc{SinkNeg}(F)\langle p\rangle = \neg_{\mathcal{L}}F'$ where $F' \neq P(t_1, \ldots, t_n)$. Obviously $p \neq \varepsilon$. So either $p = 1p_1$ or $p = 2p_2$, where

  $$p_1 \in Pos(\textsc{SinkNeg}(\neg_{\mathcal{L}}F_1)) \wedge p_2 \in Pos(\textsc{SinkNeg}(\neg_{\mathcal{L}}F_2)).$$

  W.l.o.g. let $p = 1p_1$. But then we get a contradiction to our hypothesis since

  $$\textsc{SinkNeg}(F)\langle p_1\rangle = \neg_{\mathcal{L}}F'.$$

The other cases can be proved similarly. □

Lemma 64. *Let F be a formula. Then we have:*

$$\nexists p \in Pos(\textsc{SinkNeg}(F)), F_1, F_2 : \textsc{SinkNeg}(F)\langle p\rangle = F_1 \Rightarrow_{\mathcal{L}} F_2$$

Proof. The proof is similar to Lemma 56, so we will not give a proof of this lemma here. □

Lemma 65. *Let F be a formula. Then we have:*

$$\nexists p \in Pos(\textsc{SinkNeg}(F)), F_1, F_2 : \textsc{SinkNeg}(F)\langle p\rangle = F_1 \Leftrightarrow_{\mathcal{L}} F_2$$

Proof. For a similar reason as in Lemma 64 we will not give a proof here. □

4.3.3 UniQVars

Initially we show that the preconditions of the function hold in every recursive call. Then we show that the algorithm terminates. After this we show that the postconditions hold.

Theorem 66. *Let F be a formula and let V_1 and V_2 be sets of variables such that F, V_1 and V_2 fulfil the preconditions of* UniQVars*. Then also the preconditions for all recursive calls are fulfilled.*

Proof. We show this for one precondition after the other. For each precondition we prove this by considering all possible shapes of F. We start with the third one. We do not give a proof for the first two preconditions here, because on the one hand the proofs work similar as the proof for the third one and on the other hand we had the same precondition with a related proof for SinkNeg. In the following we assume that the formula F satisfies the preconditions.

- *Case:* $F = \exists_{\mathcal{L}} x^i F_1$ and $x^i \notin V_1$
  Assume there is a $p \in Pos(F_1)$ and a formula $\hat{F} \neq P(t_1, \ldots, t_n)$ such that $F_1\langle p\rangle = \neg_{\mathcal{L}}\hat{F}$. Then there is also $\hat{p} \in Pos(F)$ such that $F\langle\hat{p}\rangle = \neg_{\mathcal{L}}\hat{F}$ namely $\hat{p} = 1p$. But this is a contradiction to our assumption that F satisfies the precondition.

- *Case:* $F = \exists_{\mathcal{L}} x^i F_1$ and $x^i \in V_1$
  Assume there is $p \in Pos(subs(F_1, x^i, y^i))$ and a formula $\hat{F} \neq P(t_1, \ldots, t_n)$ such that $subs(F_1, x^i, y^i)\langle p\rangle = \neg_{\mathcal{L}}\hat{F}$. By Lemma 34 we get that
  $$subs(F_1\langle p\rangle, x^i, y^i) = \neg_{\mathcal{L}}\hat{F}.$$
  By the definition of *subs* we get that $F_1\langle p\rangle = \neg_{\mathcal{L}}\tilde{F}$ for some formula $\tilde{F}$. Obviously $\tilde{F}$ is also not a predicate applied to terms. Since $F\langle 1p\rangle = F_1$ we therefore get a contradiction to the assumption that F satisfies the precondition.

The other cases can be proved similar. We now prove the assertion for the fourth precondition.

- *Case:* $F = F_1 \wedge_{\mathcal{L}} F_2$
  We know that $free(F) = free(F_1) \cup free(F_2)$. Thus, we get that $free(F_1) \subseteq free(F)$ and $free(F_2) \subseteq free(F)$. Therefore $free(F_1) \subseteq V_1$ and $free(F_2) \subseteq V_1$.

- *Case:* $F = \exists_{\mathcal{L}} x^i F_1$ and $x^i \notin V_1$
  Since $free(F) = free(F_1) \setminus \{x^i\}$ we know that $free(F_1) \setminus \{x^i\} \subseteq V_1$ and therefore $free(F_1) \subseteq V_1 \cup \{x^i\}$.

- *Case:* $F = \exists_{\mathcal{L}} x^i F_1$ and $x^i \in V_1$
  By Lemma 40 we know that
  $$free(subs(F_1, x^i, y^i)) \subseteq (free(F_1) \setminus \{x^i\}) \cup \{y^i\}$$
  Similar as in the previous case we get: $free(F_1) \setminus \{x^i\} \subseteq V_1$ If we combine this we get that $free(subs(F_1, x^i, y^i)) \subseteq V_1 \cup \{y^i\}$

The other cases can be proved similarly. Last but not least we now prove the assertion for the last precondition.

- *Case:* $F = F_1 \wedge_{\mathcal{L}} F_2$
  $bound(F) = bound(F_1) \cup bound(F_2)$ So we get $bound(F_1) \subseteq bound(F)$ and $bound(F_2) \subseteq bound(F)$. So $bound(F_1) \subseteq V_2$ and $bound(F_2) \subseteq V_2$
- *Case:* $F = \exists_{\mathcal{L}} x^i F_1$ and $x^i \notin V_1$
  We have that $bound(F) = bound(F_1) \cup \{x^i\}$, this implies $bound(F_1) \subseteq bound(F)$. So we have $bound(F_1) \subseteq V_2$.
- *Case:* $F = \exists_{\mathcal{L}} x^i F_1$ and $x^i \in V_1$
  By Lemma 41 we have that $bound(F_1) = bound(subs(F_1, x^i, y^i))$ As in the previous case we have that $bound(F_1) \subseteq bound(F)$. So we have $bound(F_1) \subseteq V_2$.

The other cases can be proved similarly. □

Theorem 67. UniQVars *terminates.*

Proof. Can be proved analogously to Theorem 52 □

Lemma 68. *Let F be a formula and let V_1 and V_2 be sets of variables such that F, V_1 and V_2 fulfil the preconditions of* UniQVars*. Then we have:*

$$bound(\textsc{UniQVars}(F, V_1, V_2)) \cap V_1 = \emptyset$$

Proof. The base case holds since: If $|Pos(F)| = 1$ then $F = P(t_1, \ldots, t_m)$. So we have that $bound(F) = \emptyset$. So $bound(\textsc{UniQVars}(F, V_1, V_2)) \cap V_1 = \emptyset$. For the inductive step we have to distinct between the possible shapes of F.

- *Case:* $F = F_1 \wedge_{\mathcal{L}} F_2$

$$\begin{gathered} bound(\textsc{UniQVars}(F, V_1, V_2)) = \\ = bound(\textsc{UniQVars}(F_1, V_1, V_2) \wedge_{\mathcal{L}} \textsc{UniQVars}(F_2, V_1, V_2)) = \\ = bound(\textsc{UniQVars}(F_1, V_1, V_2) \cup bound(\textsc{UniQVars}(F_2, V_1, V_2)) \\ \Rightarrow bound(\textsc{UniQVars}(F, V_1, V_2)) \cap V_1 = \\ = (bound(\textsc{UniQVars}(F_1, V_1, V_2) \cap V_1) \cup \\ \cup (bound(\textsc{UniQVars}(F_2, V_1, V_2)) \cap V_1) = \\ \text{By the hypothesis: } = \emptyset \cup \emptyset = \emptyset \end{gathered}$$

- *Case:* $F = \exists_{\mathcal{L}} x^i F_1$ and $x^i \notin V_1$

$$\begin{gathered} bound(\textsc{UniQVars}(F, V_1, V_2)) = \\ = bound(\textsc{UniQVars}(F_1, V_1 \cup \{x^i\}, V_2) \cup \{x^i\} \\ \text{Thus } bound(\textsc{UniQVars}(F, V_1, V_2)) \cap V_1 = \\ = (bound(\textsc{UniQVars}(F_1, V_1 \cup \{x^i\}, V_2) \cup \{x^i\}) \cap V_1 = \\ = (bound(\textsc{UniQVars}(F_1, V_1 \cup \{x^i\}, V_2) \cap V_1) \cup (\{x^i\} \cap V_1) = \\ \text{By the hypothesis: } = \emptyset \cup \emptyset = \emptyset \end{gathered}$$

- *Case:* $F = \exists_{\mathcal{L}} x^i F_1$ and $x^i \in V_1$

$$\begin{gathered} \mathit{bound}(\textsc{UniQVars}(F, V_1, V_2)) = \\ = \mathit{bound}(\textsc{UniQVars}(\mathit{subs}(F_1, x^i, y^i), V_1 \cup \{y^i\}, V_2)) \cup \{y^i\} \\ \text{Thus } \mathit{bound}(\textsc{UniQVars}(F, V_1, V_2)) \cap V_1 = \\ (\mathit{bound}(\textsc{UniQVars}(\mathit{subs}(F_1, x^i, y^i), V_1 \cup \{y^i\}, V_2)) \cap V_1)(\{y^i\} \cap V_1) = \\ \text{By the hypothesis: } = \emptyset \cap \emptyset = \emptyset \end{gathered}$$

□

Lemma 69. *Let F be a formula and let V_1 and V_2 be sets of variables such that F, V_1 and V_2 fulfil the preconditions of* UniQVars*. Then we have:*

$$\mathit{free}(F) = \mathit{free}(\textsc{UniQVars}(F, V_1, V_2))$$

Proof.

- *Case:* $F = P(t_1, \ldots, t_n)$
  Obvious.

- *Case:* $F = F_1 \wedge_{\mathcal{L}} F_2$

$$\begin{gathered} \mathit{free}(\textsc{UniQVars}(F, V_1, V_2)) = \\ = \mathit{free}(\textsc{UniQVars}(F_1, V_1, V_2)) \cup \mathit{free}(\textsc{UniQVars}(F_2, V_1, V_2)) = \\ \text{By the hypothesis: } = \mathit{free}(F_1) \cup \mathit{free}(F_2) = \mathit{free}(F) \end{gathered}$$

- *Case:* $F = \exists_{\mathcal{L}} x^i F_1$ and $x^i \notin V_1$

$$\begin{gathered} \mathit{free}(\textsc{UniQVars}(F, V_1, V_2)) = \\ = \mathit{free}(\textsc{UniQVars}(F_1, V_1 \cup \{x^i\}, V_2) \setminus \{x^i\} = \\ \text{By the hypothesis: } = \mathit{free}(F_1) \setminus \{x^i\} = \mathit{free}(F) \end{gathered}$$

- *Case:* $F = \exists_{\mathcal{L}} x^i F_1$ and $x^i \in V_1$

$$\begin{gathered} \mathit{free}(\textsc{UniQVars}(F, V_1, V_2)) = \\ = \mathit{free}(\textsc{UniQVars}(\mathit{subs}(F_1, x^i, y^i), V_1 \cup \{y^i\}, V_2)) \setminus \{y^i\} \\ \text{By the hypothesis: } = \mathit{free}(\mathit{subs}(F_1, x^i, y^i)) \setminus \{y^i\} = \end{gathered}$$

  Because of the way we have choosen y^i we get that $y^i \notin \mathit{bound}(F_1)$ so we can use Lemma 43. This lemma implies that:

$$\begin{gathered} x^i \in \mathit{free}(F_1) \Rightarrow \mathit{free}(\textsc{UniQVars}(F, V_1, V_2)) = \\ = (\mathit{free}(F_1) \cup \{y^i\}) \setminus \{x^i, y^i\} = \mathit{free}(F_1) \setminus \{x^i\} = \mathit{free}(F) \\ x^i \notin \mathit{free}(F_1) \Rightarrow \mathit{free}(\textsc{UniQVars}(F, V_1, V_2)) = \\ = \mathit{free}(F_1) \setminus \{y^i\} = \mathit{free}(F_1) = \mathit{free}(F) \end{gathered}$$

The other cases can be proved similarly. □

Corollary 70. *Let F be a formula and let V_1 and V_2 be sets of variables such that F, V_1 and V_2 fulfil the preconditions of* UniQVars*. Then we have:*

$$\mathit{free}(\textsc{UniQVars}(F, V_1, V_2)) \subseteq V_1$$

Proof. By Lemma 69 $\mathit{free}(\textsc{UniQVars}(F, V_1, V_2)) = \mathit{free}(F)$. Since the precondition implies $\mathit{free}(F) \subseteq V_1$ we get that $\mathit{free}(\textsc{UniQVars}(F, V_1, V_2)) \subseteq V_1$. □

Corollary 71. *Let F be a formula and let V_2 be a set of variables such that F, $\emptyset$ and V_2 fulfil the preconditions of* UniQVars*. Then we have:*

$$\mathit{free}(\textsc{UniQVars}(F, \emptyset, V_2)) = \emptyset$$

Proof. Since the preconditions are satisfied Corollary 70 implies

$$\mathit{free}(\textsc{UniQVars}(F, \emptyset, V_2)) \subseteq \emptyset.$$

This means

$$\mathit{free}(\textsc{UniQVars}(F, \emptyset, V_2)) = \emptyset$$ □

Theorem 72. *Let F be a formula and let V_1 and V_2 be sets of variables such that F, V_1 and V_2 fulfil the preconditions of* UniQVars*. Then we have:*

$$\mathit{free}(\textsc{UniQVars}(F, V_1, V_2)) \cap \mathit{bound}(\textsc{UniQVars}(F, V_1, V_2)) = \emptyset$$

Proof. By Corollary 71 we get $\mathit{free}(\textsc{UniQVars}(F, V_1, V_2)) \subseteq V_1$. By Lemma 68 we get $\mathit{bound}(\textsc{UniQVars}(F, V_1, V_2)) \cap V_1 = \emptyset$. This implies that

$$\mathit{free}(\textsc{UniQVars}(F, V_1, V_2)) \cap \mathit{bound}(\textsc{UniQVars}(F, V_1, V_2)) = \emptyset$$ □

Corollary 73. *Let F be a formula and let V_1 and V_2 be sets of variables such that F, V_1 and V_2 fulfil the preconditions of* UniQVars*. Then we have:*

$$\forall p \in \mathit{Pos}(\textsc{UniQVars}(F, V_1, V_2)) :$$
$$\mathit{free}(\textsc{UniQVars}(F, V_1, V_2)\langle p\rangle) \cap \mathit{bound}(\textsc{UniQVars}(F, V_1, V_2)\langle p\rangle) = \emptyset$$

Proof. By Theorem 72 the assertion holds for the position ε. So in the following let $p \in \mathit{Pos}(\textsc{UniQVars}(F, V_1, V_2)) \setminus \{\varepsilon\}$

- *Case:* $F = P(t_1, \ldots, t_n)$
  $\mathit{Pos}(\textsc{UniQVars}(F, V_1, V_2)) \setminus \{\varepsilon\} = \emptyset$ so the assertion holds.
- *Case:* $F = F_1 \wedge_{\mathcal{L}} F_2$
  In this case we either have $p = 1p_1$ or $p = 2p_2$ where

  $$p_1 \in \mathit{Pos}(\textsc{UniQVars}(F_1, V_1, V_2)) \wedge p_2 \in \mathit{Pos}(\textsc{UniQVars}(F_2, V_1, V_2)).$$

W.l.o.g. we assume that $p = 1p_1$.

$$\begin{gathered} \mathit{free}(\textsc{UniQVars}(F, V_1, V_2)\langle p\rangle) \cap \mathit{bound}(\textsc{UniQVars}(F, V_1, V_2)\langle p\rangle) = \\ \mathit{free}(\textsc{UniQVars}(F_1, V_1, V_2)\langle p1\rangle) \cap \mathit{bound}(\textsc{UniQVars}(F_1, V_1, V_2)\langle p_1\rangle) = \\ \text{By the hypothesis: } = \emptyset \end{gathered}$$

- *Case:* $F = \exists_{\mathcal{L}} x^i F_1$ and $x^i \notin V_1$
  We have that $p = 1p_1$, where $p_1 \in \mathit{Pos}(\textsc{UniQVars}(F_1, V_1, V_2))$.

$$\begin{gathered} \mathit{free}(\textsc{UniQVars}(F, V_1, V_2)\langle p\rangle) \cap \mathit{bound}(\textsc{UniQVars}(F, V_1, V_2)\langle p\rangle) = \\ = \mathit{free}(\textsc{UniQVars}(F_1, V_1 \cup \{x^i\}, V_2)\langle p_1\rangle) \cap \\ \cap \mathit{bound}(\textsc{UniQVars}(F_1, V_1 \cup \{x^i\}, V_2)\langle p_1\rangle) = \\ \text{By the hypothesis: } = \emptyset \end{gathered}$$

- *Case:* $F = \exists_{\mathcal{L}} x^i F_1$ and $x^i \in V_1$
  We have that $p = 1p_1$, where $p_1 \in \mathit{Pos}(\textsc{UniQVars}(\mathit{subs}(F_1, x^i, y^i), V_1, V_2))$.

$$\begin{gathered} \mathit{free}(\textsc{UniQVars}(F, V_1, V_2)\langle p\rangle) \cap \mathit{bound}(\textsc{UniQVars}(F, V_1 \cup \{y^i\}, V_2)\langle p\rangle) = \\ \mathit{free}(\textsc{UniQVars}(\mathit{subs}(F_1, x^i, y^i), V_1, V_2)\langle p1\rangle) \cap \\ \cap \mathit{bound}(\textsc{UniQVars}(\mathit{subs}(F_1, x^i, y^i), V_1 \cup \{y^i\}, V_2)\langle p_1\rangle) = \\ \text{By the hypothesis: } = \emptyset \end{gathered}$$

The other cases can be proved similarly. □

Lemma 74. *Let F be a formula and let V_1 and V_2 be sets of variables such that F, V_1 and V_2 fulfil the preconditions of* UniQVars*. Then we have:*

$$\begin{gathered} \forall \hat{F}, p \in \mathit{Pos}(\textsc{UniQVars}(F, V_1, V_2)) : \textsc{SinkNeg}(F)\langle p\rangle = \neg_{\mathcal{L}} \hat{F} \Rightarrow \\ \mathit{Thus}\ \exists P : \hat{F} = P(t_1, \ldots, t_n) \end{gathered}$$

Proof. We will not give a proof for this lemma since the proof for this lemma would be a simpler form of the proof of Lemma 63. □

Lemma 75. *Let F be a formula and let V_1 and V_2 be sets of variables such that F, V_1 and V_2 fulfil the preconditions of* UniQVars*. Then we have:*

$$\nexists p \in \mathit{Pos}(\textsc{UniQVars}(F, V_1, V_2)), F_1, F_2 : \textsc{UniQVars}(F, V_1; V_2)\langle p\rangle = F_1 \Rightarrow_{\mathcal{L}} F_2$$

Proof. We will not give a proof for this lemma since it can be proved similarly as Lemma 56. □

Lemma 76. *Let F be a formula and let V_1 and V_2 be sets of variables such that F, V_1 and V_2 fulfil the preconditions of* UniQVars*. Then we have:*

$$\nexists p \in \mathit{Pos}(\textsc{UniQVars}(F, V_1, V_2)), F_1, F_2 : \textsc{UniQVars}(F, V_1, V_2)\langle p\rangle = F_1 \Leftrightarrow_{\mathcal{L}} F_2$$

Proof. For a similar reason as in Lemma 75 we will not give a proof here. □

Theorem 77. *Let F be a formula and let V_1 and V_2 be sets of variables such that F, V_1 and V_2 fulfil the preconditions of* UniQVars. *Then we have:*

$$F \equiv \textsc{UniQVars}(F, V_1, V_2)$$

Proof.

- *Case:* $F = P(t_1, \ldots, t_n)$
  $\textsc{UniQVars}(F, V_1, V_2) = F \equiv F$

- *Case:* $F = F_1 \wedge_{\mathcal{L}} F_2$

$$\textsc{UniQVars}(F, V_1, V_2) =$$
$$= \textsc{UniQVars}(F_1, V_1, V_2) \wedge_{\mathcal{L}} \textsc{UniQVars}(F_2, V_1, V_2) \equiv$$
$$\text{By the hypothesis and Lemma 32: } \equiv F_1 \wedge_{\mathcal{L}} F_2 = F$$

- *Case:* $F = \exists_{\mathcal{L}} x^i F_1$ and $x^i \notin V_1$

$$\textsc{UniQVars}(F, V_1, V_2) = \exists_{\mathcal{L}} x^i \textsc{UniQVars}(F_1, V_1 \cup \{x^i\}, V_2) \equiv$$
$$\text{By the hypothesis and Lemma 33: } \equiv \exists_{\mathcal{L}} x^i F_1 = F$$

- *Case:* $F = \exists_{\mathcal{L}} x^i F_1$ and $x^i \in V_1$
  From the hypothesis we get that $\textsc{UniQVars}(\mathit{subs}(F_1, x^i, y^i) \equiv \mathit{subs}(F_1, x^i, y^i)$. Therefore we must show that $\exists_{\mathcal{L}} x^i F_1 \equiv \exists_{\mathcal{L}} y^i \mathit{subs}(F_1, x^i, y^i)$. So we have to show for all interpretations $\langle \mathcal{A}, M \rangle$ that

$$\{y \in \mathcal{A}_i \mid \langle \mathcal{A}, M[x^i \mapsto y] \rangle \vDash F_1\} \neq \emptyset \Leftrightarrow$$
$$\Leftrightarrow \{y \in \mathcal{A}_i \mid \langle \mathcal{A}, M[y^i \mapsto y] \rangle \vDash \mathit{subs}(F_1, x^i, y^i)\} \neq \emptyset$$

  We fix $y \in \mathcal{A}_i$. Since $x^i \in V_1$ and $y^i \notin V_1$ we get that $x^i \neq y^i$. Since $x^i \neq y^i$ and $y^i \notin \mathit{free}(F)$ we get that $y^i \notin \mathit{free}(F_1)$. Since $y^i \notin V_2$ we get that $y^i \notin \mathit{bound}(F_1)$. By Theorem 47

$$\langle \mathcal{A}, M[y^i \mapsto y] \rangle(\mathit{subs}(F_1, x^i, y^i)) =$$
$$\langle \mathcal{A}, M[y^i \mapsto y, x^i \mapsto \langle \mathcal{A}, M[y^i \mapsto y] \rangle(y^i) \rangle(F_1)$$

  We know that $M[y^i \mapsto y](y^i) = y$ therefore

$$\langle \mathcal{A}, M[y^i \mapsto y, x^i \mapsto \langle \mathcal{A}, M[y^i \mapsto y] \rangle(y^i) \rangle(F_1) = \langle \mathcal{A}, M[x^i \mapsto y, y^i \mapsto y] \rangle(F_1)$$

  By Lemma 36 and since $y^i \notin \mathit{free}(F_1)$

$$\langle \mathcal{A}, M[x^i \mapsto y, y^i \mapsto y] \rangle(F_1) = \langle \mathcal{A}, M[x^i \mapsto y] \rangle(F_1)$$

  But this implies that

$$\{y \in \mathcal{A}_i \mid \langle \mathcal{A}, M[x^i \mapsto y \rangle \vDash F_1\} = \{y \in \mathcal{A}_i \mid \langle \mathcal{A}, M[y^i \mapsto y \rangle \vDash \mathit{subs}(F_1, x^i, y^i)\}$$

  This proves the assertion.

The other cases can be proved similarly. □

4.3.4 Skol

First we show that the preconditions of the function hold in every recursive call. Then we show that the algorithm terminates. After this, we show that the postconditions hold.

Theorem 78. *Let F be a formula and FS a set of operation symbols such that F and FS fulfil the preconditions of* Skol. *Then also the preconditions for all recursive calls are fulfilled.*

Proof. We show this for one precondition after the other. For each precondition we prove this by considering all possible shapes of F. We start with the fourth one. We do not give a proof for the first three preconditions here, because we already had similar proofs. In the following we assume that the preconditions holds for F and FS.

- *Case:* $F = F_1 \wedge_{\mathcal{L}} F_2$
  We assume that the precondition does not hold for one of the recursive calls. We assume it does not hold for the recursive call with respect to F_1, the other case works analogously. Then there is a $p \in Pos(F_1)$ such that

  $$bound(F_1\langle p\rangle) \cap free(F_1\langle p\rangle) \neq \emptyset.$$

  But this yields a contradiction to our assumption since $1p \in Pos(F)$.

- *Case:* $F = \exists_{\mathcal{L}} x^i F_1$
  vars is a sequence consisting of the elements of $free(F)$ (Can be easily proved). Therefore, we get

  $$free(f(vars)) = free(F).$$

  By the assumption and since

  $$bound(F) = bound(F_1) \cup \{x^i\}$$

  we get that

  $$bound(F_1) \cap free(F) = \emptyset.$$

  So also

  $$bound(F_1) \cap free(f(vars)) = \emptyset.$$

  By Lemma 44 we get

  $$free(subs(F_1, x^i, f(vars)) \subseteq (free(F_1) \setminus \{x^i\}) \cup free(f(vars))$$

  By Lemma 41 we get that $bound(subs(F_1, x^i, f(vars))) = bound(F_1)$.

  $$\begin{gathered} free(subs(F_1, x^i, f(vars))) \cap bound(subs(F_1, x^i, f(vars))) \subseteq \\ \subseteq ((free(F_1) \setminus \{x^i\}) \cup free(f(vars))) \cap bound(F_1) = \\ = free(F) \cap bound(F_1) = \emptyset \end{gathered}$$

Next we show that also the fifth precondition is preserved. Here we assume that the algorithm is correct. This assumption would actually require to prove this property together with the correctness. As this would only lead to more tedious consideration without any insights, we still prove the above property separately.

- *Case:* $F = F_1 \wedge_{\mathcal{L}} F_2$
  We have that $getOpSyms(F) = getOpSyms(F_1) \cup getOpSyms(F_2)$. This implies that $getOpSyms(F_1) \subseteq getOpSyms(F)$ and $getOpSyms(F_2) \subseteq getOpSyms(F)$. By the postcondition $FS \subseteq \psi_2$. Therefore $getOpSyms(F_2) \subseteq \psi_2$

- *Case:* $F = \exists_{\mathcal{L}} x^i F_1$
  We have that $getOpSyms(F) = getOpSyms(F_1)$ □

Theorem 79. SKOL *terminates.*

Proof. On the one hand we have to show that recursion does not cause non-termination. On the other hand we have to show that the loop in the case $F = \exists_{\mathcal{L}} x^i F_1$ does not cause non-termination.
First, we show that the loop terminates. Obviously S is a finite set. To show the termination we now give a termination measure for the loop: $T \coloneqq |S|$. It is pretty obvious that this measure is decreased in each loop step and that it is bound from below.
Second we show that no non-termination arises from the recursion. This can be proved as we did in Theorem 52. □

Lemma 80. *Let F be a formula and FS a set of operation symbols such that F, FS fulfil the preconditions of* SKOL*. Then we have*

$$FS \subseteq \text{SKOL}(F, FS)_2$$

Proof.

- *Case:* $F = P(t_1, \ldots, t_n)$
  $\text{SKOL}(F, FS) = (F, FS)$ this implies that $FS \subseteq \text{SKOL}(F, FS)_2$.

- *Case:* $F = F_1 \wedge_{\mathcal{L}} F_2$
  By the hypothesis we get that $FS \subseteq \psi_2$ and $\psi_2 \subseteq \chi_2$. This implies that $FS \subseteq \chi_2$

- *Case:* $\exists_{\mathcal{L}} x^i F_1$
  By the hypothesis: $FS \cup \{f\} \subseteq \text{SKOL}(subs(F_1, x^i, f(vars)), FS \cup \{f\})_2$. This implies that $FS \subseteq \text{SKOL}(subs(F_1, x^i, f(vars)), FS \cup \{f\})_2$.

The other cases can be proved similarly. □

Lemma 81. *Let F be a formula and FS a set of operation symbols such that F and FS fulfil the preconditions. Let* $\text{SKOL}(F, FS) = (F', FS')$*. Then we have:*

$$getOpSyms(F') \subseteq FS'$$

Proof.

- *Case:* $F = P(t_1, \ldots, t_n)$
  $\textsc{Skol}(F, FS) = (F, FS)$ this implies that $getOpSyms(F') \subseteq FS'$.

- *Case:* $F = F_1 \wedge_{\mathcal{L}} F_2$
  Let $\psi = \textsc{Skol}(F_1, FS)$ and $\chi = \textsc{Skol}(F_2, \psi_2)$. By the hypothesis we get that $getOpSyms(\psi_1) \subseteq \psi_2$ and $getOpSyms(\chi_1) \subseteq \chi_2$. Furthermore we know that $FS \subseteq \psi_2 \subseteq \chi_2$. This implies $getOpSyms(\psi_1) \subseteq \chi_2$. From this we can conclude that $getOpSyms(\psi_1 \wedge_{\mathcal{L}} \chi_1) \subseteq \chi_2$.

- *Case:* $F = \exists_{\mathcal{L}} x^i F_1$
  Let $\psi = \textsc{Skol}(subs(F, x^i, f(vars)), FS \cup \{f\})$. By the hypothesis we get that $getOpSyms(\psi_1) \subseteq \psi_2$

The other cases can be proved similarly. □

Lemma 82. *Let F be a formula and FS a set of operation symbols such that F and FS fulfil the preconditions. Furthermore, let $f \in FS$, $f \notin getOpSyms(F)$ and let $\textsc{Skol}(F, FS) = (F', FS')$. Then we have:*

$$f \notin getOpSyms(F').$$

Proof. We assume that F and FS fulfil the premise of the lemma.

- *Case:* $F = P(t_1, \ldots, t_n)$
  $\textsc{Skol}(F, FS) = (F, FS)$ therefore

  $$f \notin getOpSyms(F) \Rightarrow f \notin getOpSyms(\textsc{Skol}(F, FS)_1)$$

- *Case:* $F = F_1 \wedge_{\mathcal{L}} F_2$
  Let $\psi = \textsc{Skol}(F_1, FS)$ and $\chi = \textsc{Skol}(F_2, \psi_2)$. Since F and FS satisfy the premise of the lemma we get that $f \notin getOpSyms(F_1)$ and $f \notin getOpSyms(F_2)$. By Lemma 81 we get that $FS \subseteq \psi_2 \subseteq \chi_2$. This implies that $f \in \psi_2$ and $f \in \chi_2$ so the premise for this lemma is satisfied for the sub-cases. Therefore, by the hypothesis we get that $f \notin getOpSyms(\psi_1)$ and $f \notin getOpSyms(\chi_1)$. This implies that $f \notin getOpSyms(\psi_1 \wedge_{\mathcal{L}} \chi_1) = getOpSyms(\textsc{Skol}(F, FS)_1)$

- *Case:* $F = \exists_{\mathcal{L}} x^i F_1$
  By Lemma 40 we get that:

  $$getOpSyms(subs(F_1, x^i, f(vars))) \subseteq getOpSyms(F_1) \cup getOpSyms(f(vars))$$

  Since *vars* is a sequence of variables we get that $getOpSyms(f(vars)) = \{f\}$. This implies that:

  $$getOpSyms(subs(F_1, x^i, f(vars))) \subseteq getOpSyms(F_1) \cup \{f\}$$

Now let $g \in FS$ and $g \notin getOpSyms(F)$. Then $g \notin getOpSyms(F_1)$. Since $g \in FS$ we get that $f \neq g$. But this implies that $g \notin getOpSyms(F_1) \cup \{f\}$. So we can conclude that $g \notin getOpSyms(subs(F_1, x^i, f(vars)))$ Therefore

$$g \notin getOpSyms(\text{SKOL}(subs(F_1, x^i, f(vars)), FS \cup \{f\})_1)$$

The other cases can be proved similarly. □

Corollary 83. *Let F be a formula and FS a set of operation symbols such that F and FS fulfil the preconditions of* SKOL. *Furthermore let $Q \subseteq FS$ such that $Q \cap getOpSyms(F)) = \emptyset$ and $\text{SKOL}(F, FS) = (F', FS')$. Then we have:*

$$getOpSyms(F') \cap Q = \emptyset.$$

Proof. Assume that $Q \subseteq FS$, $getOpSyms(F) \cap Q = \emptyset$ and $getOpSyms(F') \cap Q \neq \emptyset$. Then we can find a $q \in Q \cap getOpSyms(F')$. This means $q \in FS$, $q \notin getOpSyms(F)$ and $q \in getOpSyms(F')$. But this is a contradiction to Lemma 82, therefore $getOpSyms(F') \cap Q = \emptyset$. □

Lemma 84. *Let F be a formula from a signature Σ and FS a set of operation symbols such that F and FS fulfil the preconditions of* SKOL. *Furthermore, let $\psi = \text{SKOL}(F, FS)$. Then for each T-structure $\mathcal{A}$ there is a T-structure $\mathcal{B}$ over the same signature such that*

$$\forall f \in Op_\Sigma \setminus (\psi_2 \setminus FS) : f^{\mathcal{A}} = f^{\mathcal{B}}$$
$$\forall i \in \Sigma.Sort : \mathcal{A}_i = \mathcal{B}_i$$

and for each variable assignment M, $\langle \mathcal{A}, M \rangle(F) = \langle \mathcal{B}, M \rangle(\psi_1)$.

Proof. Let $\mathcal{A}$ be a structure, M be a variable assignment and $I = \langle \mathcal{A}, M \rangle$.

- *Case:* $F = P(t_1, \ldots, t_n)$
  In this case we have $\mathcal{B} = \mathcal{A}$. Obviously this structure fulfils all the requirements.

- *Case:* $F = F_1 \wedge_{\mathcal{L}} F_2$
  Let $\psi = \text{SKOL}(F_1, FS)$ and let $\chi = \text{SKOL}(F_2, \psi_2)$. By the hypothesis there are structures $\mathcal{B}_1$ and $\mathcal{B}_2$ which satisfy the properties described above. Let $J_1 = \langle \mathcal{B}_1, M \rangle$ and $J_2 = \langle \mathcal{B}_2, M \rangle$. We know that

  $$I(F_1) = J_1(\text{SKOL}(F_1, FS)_1) \text{ and } I(F_2) = J_2(\text{SKOL}(F_2, \psi_2)_1)$$

  By Lemma 81 we get that $getOpSyms(\psi_1) \subseteq \psi_2$ and $getOpSyms(\chi_1) \subseteq \chi_2$. We define the sets Q and Λ as:

  $$Q \coloneqq getOpSyms(\psi_1) \setminus FS \qquad \Lambda \coloneqq getOpSyms(\chi_1) \setminus FS$$

  We get that $Q \subseteq \psi_2$. Since $getOpSyms(F_2) \subseteq FS$ we get $Q \cap getOpSyms(F_2) = \emptyset$. By Corollary 83 we therefore get $Q \cap getOpSyms(\chi_1) = \emptyset$. This implies that

$Q \cap \Lambda = \emptyset$. This means that the operation symbols which where introduced in ψ are different from the operation symbols introduced in χ. Now we can define a structure $\mathcal{B}$ which satisfies all the required properties.

$$\forall i \in Sort : \mathcal{B}_i \coloneqq \mathcal{A}_i$$
$$\forall f \in FS : f^{\mathcal{B}} \coloneqq f^{\mathcal{A}}$$
$$\forall f \in Q : f^{\mathcal{B}} \coloneqq f^{\mathcal{B}_1}$$
$$\forall f \in \Lambda : f^{\mathcal{B}} \coloneqq f^{\mathcal{B}_2}$$
$$\text{all other operation symbols: } f^{\mathcal{B}} \coloneqq f^{\mathcal{A}}$$

Let $J \coloneqq \langle \mathcal{B}, M \rangle$. By Lemma 38 we get that

$$J(\textsc{Skol}(F_1, FS)_1) = J_1(\textsc{Skol}(F_1, FS)_1)$$
$$J(\textsc{Skol}(F_2, \psi_2)_1) = J_2(\textsc{Skol}(F_2, \psi_2)_1)$$

From this we can conclude that $I(F) = J(\textsc{Skol}(F, FS)_1$. This means that $\mathcal{B}$ satisfies all the required conditions.

- *Case:* $F = \exists_{\mathcal{L}} x^i F_1$
  Let $\psi = \textsc{Skol}(subs(F_1, x^i, f(vars)), FS \cup \{f\})$. By the precondition $bound(F) \cap free(F) = \emptyset$, so by Theorem 48 we can conclude that:

  $$\langle \mathcal{A}, M \rangle (\exists_{\mathcal{L}} x^i F_1) = \langle \mathcal{B}, M \rangle (subs(F_1, x^i, f(vars))$$

  Where $\mathcal{B}$ is as in the referenced theorem. From the hypothesis we get that there is a structure $\mathcal{C}$ that satisfies the properties described above such that

  $$\langle \mathcal{B}, M \rangle (subs(F_1, x^i, f(vars)) = \langle \mathcal{C}, M \rangle (\psi_1)$$

  We can see that:
  $$\forall g \in Op_\Sigma \setminus (\psi_2 \setminus FS) : g^{\mathcal{A}} = g^{\mathcal{C}}$$

  So the assertion holds for this case.

The other cases can be proved similarly. □

Lemma 85. *Let F be a formula and FS a set of operation symbols such that F and FS fulfil the preconditions of* Skol. *Then for each T-interpretation $I = \langle \mathcal{A}, M \rangle$ we have:*

$$I \vDash \textsc{Skol}(F, FS)_1 \Rightarrow I \vDash F$$

Proof.

- *Case:* $F = P(t_1, \ldots, t_n)$
  Since $\textsc{Skol}(F, FS)_1 = F$ the assertion obviously holds.

- *Case:* $F = F_1 \wedge_{\mathcal{L}} F_2$

$$\psi \coloneqq \textsc{Skol}(F_1, FS)$$
$$\chi \coloneqq \textsc{Skol}(F_2, \psi_2)$$

Assume that $I \vDash \textsc{Skol}(F_1 \wedge_{\mathcal{L}} F_2, FS)_1$. This implies $I \vDash \psi_1$ and $I \vDash \chi_1$. By the hypothesis we get that $I \vDash F_1$ and $I \vDash F_2$. Therefore $I \vDash F$.

- *Case:* $F = \exists_{\mathcal{L}} x^i F_1$
Let $\psi \coloneqq \textsc{Skol}(subs(F_1, x^i, f(vars)), FS \cup \{f\})$. We assume that $I \vDash \psi_1$. By the hypothesis we get that for each interpretation J

$$J \vDash \psi_1 \Rightarrow J \vDash subs(F_1, x^i, f(vars))$$

So we have $I \vDash subs(F_1, x^i, f(vars))$. We have that $free(f(vars)) = free(F)$ and $free(F) \cap bound(F) = \emptyset$. Thus $free(F(vars)) \cap bound(F_1) = \emptyset$. Now we can apply Theorem 47 and get that:

$$I(subs(F_1, x^i, f(vars))) = \langle \mathcal{A}, M[x^i \mapsto I(f(vars))] \rangle (F_1)$$

This implies that $\langle \mathcal{A}, M[x^i \mapsto I(f(vars))] \rangle \vDash F_1$. From this we can conclude that (since $I(f(vars))$ is in the set)

$$\{y \in \mathcal{A}_i \mid \langle \mathcal{A}, M[x^i \mapsto y] \rangle \vDash F_1\} \neq \emptyset$$

From this we immediately get $I \vDash \exists_{\mathcal{L}} x^i F_1$.

The other cases can be proved similarly. □

Theorem 86. *Let F be a formula and FS a set of operation symbols such that F and FS fulfil the preconditions of* Skol. *Then for every variable assignment M we have that there is a T-structure $\mathcal{A}$ such that $\langle \mathcal{A}, M \rangle \vDash F$ iff there is a T-structure $\mathcal{B}$ such that $\langle \mathcal{B}, M \rangle \vDash \textsc{Skol}(F, FS)_1$*

Proof. The "⇒" part of the theorem follows from Lemma 84. The "⇐" part of the theorem follows from Lemma 85. □

Corollary 87. *Let F be a closed formula and FS a set of operation symbols such that F and FS fulfil the preconditions of* Skol. *Then F and $\textsc{Skol}(F, FS)_1$ are T-equisatisfiable.*

Proof. Follows directly from Theorem 86. □

Lemma 88. *Let F be a formula and FS a set of operation symbols such that F and FS fulfil the preconditions of* Skol. *Then we have:*

$$free(F) = free(\textsc{Skol}(F, FS)_1).$$

Proof.

- *Case:* $F = P(t_1, \ldots, t_n)$
  Obvious.

- *Case:* $F = F_1 \wedge_{\mathcal{L}} F_2$
  Let $\psi = \textsc{Skol}(F_1, FS)$ and let $\chi = \textsc{Skol}(F_2, \psi_2)$. By the hypothesis we get that $\mathit{free}(\psi_1) = \mathit{free}(F_1)$ and $\mathit{free}(\chi_1) = \mathit{free}(F_2)$ So we can conclude:

$$\mathit{free}(\textsc{Skol}(F, FS)_1) = \mathit{free}(\psi_1) \cup \mathit{free}(\chi_1) = \mathit{free}(F_1) \cup \mathit{free}(F_2) = \mathit{free}(F)$$

- *Case:* $F = \exists_{\mathcal{L}} x^i F_1$
  Let $\psi \coloneqq \textsc{Skol}(\mathit{subs}(F_1, x^i, f(\mathit{vars})), FS \cup \{f\})$. By the hypothesis we get:

$$\mathit{free}(\psi_1) = \mathit{free}(\mathit{subs}(F_1, x^i, f(\mathit{vars})))$$

  From the precondition of Skol we can conclude that the variables in *vars* are not bound in F. Therefore by Lemma 43 we get:

$$\begin{aligned} x^i \in \mathit{free}(F_1) &\Rightarrow \mathit{free}(\psi_1) \\ &= (\mathit{free}(F_1) \setminus \{x^i\}) \cup \mathit{free}(f(\mathit{vars})) \\ x^i \notin \mathit{free}(F_1) &\Rightarrow \mathit{free}(\psi_1) = \mathit{free}(F_1) \end{aligned}$$

  We know that $\mathit{free}(f(\mathit{vars})) = \mathit{free}(F)$. From this we can conclude that in both cases we get $\mathit{free}(\psi_1) = \mathit{free}(F)$

The other cases can be proved similarly. □

Corollary 89. *Let F be a formula and FS a set of operation symbols such that F and FS fulfil the preconditions of* Skol. *Then we have:*

$$\mathit{free}(F) = \emptyset \Rightarrow \mathit{free}(\textsc{Skol}(F, FS)_1) = \emptyset$$

Proof. Follows directly from Lemma 88. □

Lemma 90. *Let F be a formula and FS a set of operation symbols such that F and FS fulfil the preconditions of* Skol. *Then we have:*

$$\forall \hat{F}, p \in \mathit{Pos}(\textsc{Skol}(F, FS)_1) : \textsc{SinkNeg}(F)\langle p \rangle = \neg_{\mathcal{L}} \hat{F} \Rightarrow \mathit{Thus}\ \exists P : \hat{F} = P(t_1, \ldots, t_n)$$

Proof. We will not give a proof for this lemma since the proof for this lemma would be a simpler form of the proof of Lemma 63. □

Lemma 91. *Let F be a formula and FS a set of operation symbols such that F, FS fulfil the preconditions of* Skol. *Then we have:*

$$\nexists p \in \mathit{Pos}(\textsc{Skol}(F, FS)_1) : \exists F_1, F_2 : \textsc{Skol}(F, FS)_1\langle p \rangle = F_1 \Rightarrow_{\mathcal{L}} F_2$$

Proof. We will not give a proof for this lemma since it can be proved similarly as Lemma 56 □

Lemma 92. *Let F be a formula and FS a set of operation symbols such that F and FS fulfil the preconditions of* Skol. *Then we have:*

$$\nexists p \in Pos(\textsc{Skol}(F, FS)_1) : \exists F_1, F_2 : \textsc{Skol}(F, FS)_1\langle p\rangle = F_1 \Leftrightarrow_{\mathcal{L}} F_2$$

Proof. For a similar reason as in Lemma 75 we will not give a proof here. □

Lemma 93. *Let F be a formula and FS a set of operation symbols such that F and FS fulfil the preconditions of* Skol. *Then we have:*

$$\nexists p \in Pos(\textsc{Skol}(F, FS)_1)\ \exists x^i \in \mathcal{V}\ \exists F' \in \mathcal{L} : \textsc{Skol}(F, FS)_1\langle p\rangle = \exists_{\mathcal{L}} x^i F'$$

Proof.

- *Case:* $F = P(t_1, \ldots, t_n)$
  Obvious.
- *Case:* $F = F_1 \wedge_{\mathcal{L}} F_2$
  Let $\psi = \textsc{Skol}(F_1, FS)$ and let $\chi = \textsc{Skol}(F_2, \psi_2)$. We have $\textsc{Skol}(F, FS)\langle\varepsilon\rangle = \psi_1 \wedge \chi_1$. Thus, we can conclude that there is no variable x^i and formula F' such that $\textsc{Skol}(F, FS)_1\langle\varepsilon\rangle = \exists_{\mathcal{L}} x^i F'$. For all other positions the assertion immediately follows from the hypothesis.
- *Case:* $F = \exists_{\mathcal{L}} x^i F_1$
  Let $\psi = \textsc{Skol}(subs(F, x^i, f(vars)), FS \cup \{f\})$. We immediately get from the hypothesis that the assertion holds for ψ_1

The other cases can be proved similarly. □

4.3.5 ExpAll

We first show that the preconditions of the function hold in every recursive call. Then we show that the algorithm terminates. After this, we show that the postconditions hold.

Theorem 94. *Let F be a formula such that F fulfils the preconditions of* ExpAll. *Then also the preconditions for all recursive calls are fulfilled.*

Proof. This proof works pretty similar as the previous proofs where we showed that the preconditions are preserved. Therefore, we will not give a proof here. □

Theorem 95. ExpAll *terminates.*

Proof. We can show that no non-termination arises from the recursion, similarly as we did in Theorem 52. This means that ExpAll terminates. □

Lemma 96. *Let F be a formula a such that F fulfils the preconditions of* ExpAll. *Then we have:*

$$\mathit{free}(F) = \emptyset \Rightarrow \mathit{free}(\textsc{ExpAll}(F))$$

Proof. Can be proven similar as Corollary 89. □

Lemma 97. *Let F be a formula a such that F fulfils the preconditions of* ExpAll. *Then we have:*

- $\nexists p \in \mathit{Pos}(\textsc{ExpAll}(F)), x^i \in \mathcal{V}\ \exists F' \in \mathcal{L} : \textsc{ExpAll}(F)\langle p \rangle = \exists_{\mathcal{L}} x^i F'$
- $\nexists p \in \mathit{Pos}(\textsc{ExpAll}(F)), x^i \in \mathcal{V}\ \exists F' \in \mathcal{L} : \textsc{ExpAll}(F)\langle p \rangle = \forall_{\mathcal{L}} x^i F'$

Proof. The proof follows a similar idea as Lemma 93. □

Theorem 98. *Let F be a formula such that F fulfils the preconditions of* ExpAll. *Then for any T-interpretation I we have that:*

$$I(F) = I(\textsc{ExpAll}(F))$$

Proof. In the following let I be a T-interpretation such that $I = \langle \mathcal{A}, M \rangle$.

- *Case:* $F = P(t_1, \ldots, t_n)$
  Obvious,
- *Case:* $F = F_1 \wedge_{\mathcal{L}} F_2$
  By the hypothesis we know that:

  $$I(F_1) = I(\textsc{ExpAll}(F_1))$$
  $$I(F_2) = I(\textsc{ExpAll}(F_2))$$

  Thus we can conclude that:

  $$I(F) = I(\textsc{ExpAll}(F))$$

- *Case:* $F = \forall_{\mathcal{L}} x^i F_1$
  We have that:

  $$\textsc{ExpAll}(F) = \bigwedge_{c \in \mathit{Constants}(i)} \mathit{subs}(\textsc{ExpAll}(F_1), x^i, c)$$

  We now fix a $c \in \mathit{Constants}(i)$. As $\mathit{free}(c) = \emptyset$ we can conclude by Theorem 47 that:

  $$I(\mathit{subs}(F_1, x^i, c)) = \langle \mathcal{A}, M[x^i \mapsto I(c)] \rangle (F_1)$$
  $$I(\mathit{subs}(\textsc{ExpAll}(F_1), x^i, c)) = \langle \mathcal{A}, M[x^i \mapsto I(c)] \rangle (\textsc{ExpAll}(F_1))$$

Moreover from the hypothesis we can conclude that for every T-structure $\mathcal{B}$ and variable assignment N we have that:

$$\langle \mathcal{B}, N \rangle(F_1) = \langle \mathcal{B}, N \rangle(\textsc{ExpAll}(F_1))$$

This in particular means that we have:

$$\langle \mathcal{A}, M[x^i \mapsto I(c)] \rangle(F_1) = \langle \mathcal{A}, M[x^i \mapsto I(c)] \rangle(\textsc{ExpAll}(F_1))$$

Thus we have:

$$\langle \mathcal{A}, M[x^i \mapsto I(c)] \rangle(F_1) = I(\mathit{subs}(\textsc{ExpAll}(F_1), x^i, c))$$

Now we assume that we have $I \vDash F$. This implies, by the above considerations that:

$$I \vDash \mathit{subs}(\textsc{ExpAll}(F_1), x^i, c)$$

As c was arbitrary we have that

$$I \vDash \bigwedge_{c \in \mathit{Constants}(i)} \mathit{subs}(\textsc{ExpAll}(F_1), x^i, c)$$

For the other direction we assume that:

$$I \vDash \bigwedge_{c \in \mathit{Constants}(i)} \mathit{subs}(\textsc{ExpAll}(F_1), x^i, c)$$

It suffices to show that for an arbitrary $a \in \mathcal{A}_i$ (where $\mathcal{A}_i$ denotes the universe of sort i) we have that:

$$\langle \mathcal{A}, M[x^i \mapsto a] \rangle \vDash F_1$$

But this follows directly from the assumptions we made (in particular from the assumptions on *Constants*).

Similarly, we can prove the other cases. □

5 The Translation of Theories

In this chapter we will first introduce a theory R that models a substantial part of the semantics of the RISCAL language and a theory BV that models most of the semantics of the SMT-LIB logic QF_UFBV. We will then describe a procedure TRANSLATE. This procedure takes a quantifier free formula over a signature that covers the signature of R and returns a quantifier free formula over a signature that covers the signature of BV. We will illustrate that the procedure TRANSLATE has the property that for any applicable closed formula F, F is R-satisfiable, iff TRANSLATE(F) is BV-satisfiable.

5.1 The Theory of RISCAL

In this section we will describe a theory $R = \langle \Sigma_R, \mathcal{C}_R \rangle$ which covers the semantic meaning of a substantial part of the RISCAL language. We will denote this theory as the *RISCAL theory*. First, we will discuss the sorts in the theories' signature and the universes which shall be assigned to these sorts. Then we will discuss the operation symbols in the theories' signature and their semantic meanings.

As we want to use R to model a part of the RISCAL language we want that the sorts, respectively the universes from R, correspond to types in the RISCAL language. Thus, we first give a brief overview of the RISCAL types we want to model here. For this purpose we will use the subsequent definition.

Definition 99. Let $a, b \in \mathbb{Z}$ then we define the *integer interval* $I[a, b]$ as follows:

$$I[a, b] \coloneqq \{i \in \mathbb{Z} \mid a \leq i \leq b\} \qquad \bullet$$

The following list contains the types we want to model.

Integers Let $a, b \in \mathbb{Z}$ such that $a \leq b$, then $I[a, b]$ is a type.

Booleans The set $\{\mathbb{T}, \mathbb{F}\}$ is a type.

Sets If T is a type, then $\mathcal{P}(T)$ is a type.

Tuples For every $n \in \mathbb{N}^*$, if $T_1, \cdots, T_n$ are types, then $T_1 \times \cdots \times T_n$ is a type.

Maps If T_1 is a type and T_2 is a type, then $T_2^{T_1}$ is a type.

Subtypes If T is a type, then every non-empty subset of T is a type.

We can show that these types have two properties, which will be of importance later.

Lemma 100. *Every RISCAL Type is finite and not empty.*

Lemma 101. *The set of types is countable.*

Proof. Here we mainly use the fact that the set of finite subsets of the integers is a countable set and that the countable union of countable sets is countable. We will omit the details here. □

As the set of RISCAL types is countable, we can find a set $Sort \subseteq \mathbb{N}$, which we can bijectively map to the single RISCAL types. We use the set $Sort$ as the set of sorts of Σ_R, i.e. $\Sigma_R.Sort = Sort$. (Note that we are not really interested in the actual representation of the sorts, therefore we will refer to the single sorts by using the associated types.) We require that each interpretation in the theory R assigns the sets of values of the associated RISCAL-types to the individual sorts.

Remark We denote the sorts that correspond to integer types respectively to subtypes, whose base-type is an integer type, as *integer sorts*. We denote the universes that are assigned to these sorts as *integer universes*. Similarly, we proceed with tuples, sets and maps.

Next we have to cover the operations of the theory. We demand that the theory R contains operation symbols for the subsequently listed RISCAL constants, functions and predicates. These operation symbols shall get their expected arities and the interpretations from the theory shall assign the expected meaning to these symbols (note that the way we introduced the logical framework, implies that the booleans and logical connectives are in the theory anyway. Therefore, we will not list the logical connectives).

Equalities and Inequalities equals, not equals and the four inequalities

Integers Integer constants, unary minus, addition, subtraction, multiplication, truncated division, remainder, factorial and exponential.

Tuples tuple builder, tuple access and tuple update

Maps map builder, map access and map update

Sets empty set, interval builder, enumerated set builder, union, intersection, set difference, cartesian products, big intersection, big union, cardinality, the three power set operations, is element and is subset.

For details on these operations we refer to [36]. Note that we cannot represent the individual operations by single operation symbols. The reason for this is that there is no overloading in the logic we use. Therefore, for example the operation symbols for the addition of integers in $I[0, 1]$ and for the addition of integers in $I[-2, 2]$, have to be different (we could avoid this problem by generalising the used logical framework to a logic, similar to the one presented in [32]; because such a generalised logic would enlarge several parts of this thesis and the gain would be little, we omit such a

generalisation). A further example for the necessity of different symbols for a single RISCAL operation is given by the enumerated set builders. Here we obviously need different symbols for different numbers of arguments. Nevertheless, for simplicity we will identify all these operations with single names — knowing that operations with the same name may denote different objects.

Additionally, we demand that for each universe of each sort in the theory there is a constant symbol for every value. For example, for the integer types this is also the case in the RISCAL language, but for others it is not, e.g. maps of integers. To retrieve the constant symbols of a given sort, we assume that we have a function

$$Constants : \Sigma_R.Sort \to \mathcal{P}(OpSyms)$$

which maps each sort to the set of constant symbols of this sort.

Remark If not stated otherwise, when we deal with satisfiability modulo R, we will always assume that we are only considering formulae over a signature with the same sorts as in Σ_R.

Remark We want to point out that the theory R fulfils the preconditions to be applicable for the quantifier elimination procedure presented in Chapter 4.

Remark We want to point out that actually the way, we defined the logical framework in Chapter 3, would require some restrictions on this theory. Thus, for example, tuples with boolean components or maps with boolean domains, respectively boolean images, would actually not be possible in the used framework. The reason for this is that such tuples, respectively maps would require *update*, respectively *access* functions that take boolean arguments. But as we saw in Chapter 3, this is not allowed. Nevertheless, we do not apply such restrictions, as we i.a. want to illustrate the special handling, such tuples and maps require in the implementation.

Although not necessary for the definition of a theory, we will later see that we need an enumeration for each universe of this theory. In order to define the enumeration, we first define linear orderings for each universe U. We denote these orderings by $<_U$. Moreover, we denote the reflexive closures of these orderings by $\leq_U$. The orderings we use here, are based on the ordering used for the generation of values in the RISCAL source code [34]. We differentiate between the different kinds of universes.

- Let I be a universe of integer values, then we set $<_I$ to the less than predicate.
- Let B be a universe of boolean values, then we define $<_B$ such that for $a, b \in B$ we have $a <_B b$ iff $a = \mathbb{F}$ and $b = \mathbb{T}$.
- Let T be a universe of tuples with n components and whose components have the universes $T_1 \cdots T_n$. Let $a, b \in T$, then we define $<_T$ as follows:

  $$a <_T b \Leftrightarrow \exists i \in \mathbb{N}^* : i \leq n \wedge a_i <_{T_i} b_i \wedge (\forall j \in \mathbb{N}^* : (j > i \wedge j \leq n) \Rightarrow a_j \leq_{T_i} b_j)$$

- Let M be a universe of map values, whose elements map values from the universe M_1 to elements of the universe M_2. Let $a, b \in M$, then we define $<_M$ as follows:

$$a <_M b \Leftrightarrow \exists x \in M_1 : a(x) <_{M_2} b(x) \wedge (\forall y \in M_1 : x <_{M_1} y \Rightarrow a(y) \leq_{M_2} b(y))$$

- Let S be a universe of set values, whose elements are sets with elements in U. Let $a, b \in S$, then we define $<_S$ as follows:

$$a <_S b \Leftrightarrow \exists x \in b : x \notin a \wedge \forall y \in \{s \in a \mid x <_U s\} : y \in b$$

We omit the necessary proofs. Now we can define for each universe U enumerating functions

$$enum_U : \{i \in \mathbb{N}^* \mid i \leq |U|\} \to U$$

where we enumerate the elements of U in ascending order with respect to $<_U$. This enumeration is a bijective function.

5.2 The Theory QF_UFBV

In this section we will describe a theory $BV = \langle \Sigma_{BV}, \mathcal{C}_{BV} \rangle$ that models a major part of the SMT-LIB logic of *quantifier-free bit vectors with uninterpreted functions* (QF_UFBV)[2]. We will denote the theory BV as the *bit vector theory*. Similarly, to the last section we will first discuss the sorts in Σ_{BV} and their interpretations. Then we will discuss the operation symbols in Σ_{BV} and their semantic meaning.

Before we can start with the sorts of Σ_{BV}, we will give a definition of a bit vector.

Definition 102. Let $n \in \mathbb{N}$, then we define the set of *bit vectors of length n* (denoted by $BitVec(n)$) as

$$BitVec(n) \coloneqq \{0,1\}^{\{x \in \mathbb{N} \mid x < n\}}. \qquad \bullet$$

Remark We usually write bit vectors as sequences or strings of zeros and ones. Note that we use a similar notation to the notation for bit vector literals in SMT-LIB. This means that we start with the last element of the sequence and finish with the first. E.g. 100, respectively $(1, 0, 0)$ denotes the bit vector of length three where the first two bits are zero and the last one is one.

We demand that additionally to the boolean sort, $\Sigma_{BV}.Sort$ contains for each $n \in \mathbb{N}^*$ a sort for the bit vectors of length n. We can now associate to each sort that is different from zero a set of bit vectors. As in the previous section, we are not particularly interested how this association actually looks like. So for a natural number n, we denote the sort that is associated to the bit vectors of length n, as the sort of $BitVec(n)$, respectively as the sort of the bit vectors of length n. We require that each interpretation in $\mathcal{C}_{BV}$ maps these sorts to the associated sets of bit vectors.

Next we will discuss the operations, which shall be contained by the theory BV. We require that most (but not all) operations, which are available in SMT-LIB

with the QF_UFBV logic are contained by the theory BV. In Table 5.1 we list all operations that shall be part of the theory BV (the table is based on the logic description from [2]). We require that for all of those operations, there are operation symbols with an appropriate arity that are interpreted according to the expected meaning of the operation. In Table 5.1 let $n, n_1, n_2, i \in \mathbb{N}^*; j, u, l \in \mathbb{N}$ such that $n > u \geq l$.

Moreover, there shall be a constant symbol for each value of each set of bit vectors. We denote these constants, as we would denote the semantic bit vector value. This means the constant symbol for the bit vector 101 shall be 101.

Remark Similarly as in the previous section, we need different symbols in order to represent operations that are defined for multiple arities. As in the previous section, for simplicity, we will denote these symbols by the same name. We will use the names of the modelled operations.

Remark If not stated otherwise, when we deal with satisfiability modulo BV, we will always assume that we are only considering formulae over a signature with the same sorts as Σ_{BV}.

5.3 The Framework of the Theory Translation

In this section we will describe the framework of the procedure TRANSLATE that takes a closed formula over a signature, containing the signature of the theory R and returns a closed formula over a signature, containing the signature of BV. We will semi-formally argue that this procedure preserves satisfiability – this means the original formula is satisfiable modulo R, iff the resulting formula is satisfied by a suitable BV interpretation. In the subsequent sections, we will then discuss the individual components of this procedure in more detail.

From now on we denote for each $i \in \Sigma_R.Sort$ the universe of sort i, in the theory R, by $\mathcal{R}_i$. For $i \in \Sigma_{BV}.Sort$ we denote the universe of sort i, in the theory BV, by $\mathcal{BV}_i$.

In the following, we will associate the universes of the theory R, with universes of the theory BV. We do this by introducing a function α, which maps the universes from R to the universes of BV.

$$\alpha : \{\mathcal{R}_i \mid i \in \Sigma_R.Sort\} \to \{\mathcal{BV}_i \mid i \in \Sigma_{BV}.Sort\}$$

We will define the function α for the different kinds of universes in the following sections. We also assume to have a function

$$\hat{\alpha} : \Sigma_R.Sort \to \Sigma_{BV}.Sort.$$

This function shall satisfy the property that: $\alpha(\mathcal{R}_i) = \mathcal{BV}_{\hat{\alpha}(i)}$. As we will only need this function for some technicalities and since we are not really interested in the syntactic sorts, we will not give an explicit definition for this function.

Table 5.1: BV operations

Operation	Description
$bvneg : BitVec(n) \to BitVec(n)$	2's complement unary minus
$bvadd : BitVec(n) \times BitVec(n) \to BitVec(n)$	modular addition
$bvmul : BitVec(n) \times BitVec(n) \to BitVec(n)$	modular multiplication
$bvudiv : BitVec(n) \times BitVec(n) \to BitVec(n)$	unsigned division with truncation towards 0
$bvsdiv : BitVec(n) \times BitVec(n) \to BitVec(n)$	signed division with truncation towards 0
$bvurem : BitVec(n) \times BitVec(n) \to BitVec(n)$	unsigned remainder
$bvsrem : BitVec(n) \times BitVec(n) \to BitVec(n)$	signed remainder (sign follows dividend)
$= : BitVec(n) \times BitVec(n) \to Bool$	equality of bit vectors
$distinct : BitVec(n) \times BitVec(n) \to Bool$	distinctness of bit vectors
$bvule : BitVec(n) \times BitVec(n) \to Bool$	unsigned less than or equal
$bvsle : BitVec(n) \times BitVec(n) \to Bool$	signed less than or equal
$bvult : BitVec(n) \times BitVec(n) \to Bool$	unsigned less-than
$bvslt : BitVec(n) \times BitVec(n) \to Bool$	signed less than
$bvuge : BitVec(n) \times BitVec(n) \to Bool$	unsigned greater than or equal
$bvsge : BitVec(n) \times BitVec(n) \to Bool$	signed greater than or equal
$bvugt : BitVec(n) \times BitVec(n) \to Bool$	unsigned greater than
$bvsgt : BitVec(n) \times BitVec(n) \to Bool$	signed greater than
$bvand : BitVec(n) \times BitVec(n) \to BitVec(n)$	bitwise and
$bvor : BitVec(n) \times BitVec(n) \to BitVec(n)$	bitwise or
$bvnot : BitVec(n) \to BitVec(n)$	bitwise negation
$bvshl : BitVec(n) \times BitVec(n) \to BitVec(n)$	shift left
$concat : BitVec(n_1) \times BitVec(n_2) \to BitVec(n_1 + n_2)$	concatenation of bit vectors
$zero\_extend_j : BitVec(n) \to BitVec(n + j)$	extend to a unsigned equivalent bit vector of length $n + j$
$sign\_extend_j : BitVec(n) \to BitVec(n + j)$	extend to an signed equivalent bit vector of length $n + j$
$extract_{u,l} : BitVec(n) \to BitVec(u - l + 1)$	extract the bits from the u^{th} down to the l^{th} bit
$repeat_i : BitVec(n) \to BitVec(i \cdot m)$	repeat the bit vector i times

Moreover, we will need to relate each value, of each universe, of the R theory, with a value of the associated BV universe. For this purpose, we will define, for each sort $s \in \Sigma_R.Sort$, a function that maps the universe $\mathcal{R}_s$ to the associated BV universe.

$$\beta_s : \mathcal{R}_s \to \alpha(\mathcal{R}_s).$$

For now, we only want to point out that we require that $\beta_0(\mathbb{T}) = \mathbb{T}$ and $\beta_0(\mathbb{F}) = \mathbb{F}$. We will define these functions in the following sections. There, we will see that these functions injectively map the respective R universe to the associated BV universe.

We are not interested in treating formulae over arbitrary signatures containing Σ_R, respectively Σ_{BV}. Thus, we will only deal with formulae from signatures fulfilling certain conditions.

In the following, let Σ be a signature such that $\Sigma_R \subseteq \Sigma$ and $\Sigma.Sort = \Sigma_R.Sort$. Additionally let Ξ be a signature such that $\Sigma_{BV} \subseteq \Xi$ and $\Xi.Sort = \Sigma_{BV}.Sort$. Moreover let Λ be a set of operation symbols such that:

$$\begin{aligned}\Lambda \coloneqq &\{Bool2BV, BV2Bool\} \cup \bigcup_{m:R\text{-map sort}} (\{MapAccess_m, MapUpdate_m\}) \\ &\cup \bigcup_{s \in \Sigma_R.Sort} (\{getSingleton_s\}) \cup \bigcup_{i:R\text{-integer sort}} (\{interval_i\}) \\ &\cup \bigcup_{s,t:R\text{-set sort}} (\{card_s, powSet_s, Cart_{(s,t)}\}) \\ &\cup \bigcup_{ps:R\text{-sort of sets of sets}} (\{sizeConstraint_{ps}, bigUnion_{ps}, bigIntersection_{ps}\})\end{aligned}$$

We will need these symbols in order to translate certain operations, which are part of the theory R. Thus, we will discuss the individual symbols, when we are dealing with the respective translation.

With the set Λ, we can now impose further restrictions on Σ and Ξ.

- $Op_\Xi \setminus Op_{BV} = (Op_\Sigma \setminus Op_R) \cup \Lambda$
- $Op_\Sigma \cap \Lambda = \emptyset$
- $\forall f \in Op_\Sigma \setminus Op_R : args(\Sigma, f) = args(\Xi, f)$
- $\forall f \in Op_\Sigma \setminus Op_R : \forall\, 0 \leq i \leq args(\Sigma, f) : \hat{\alpha}(arg(\Sigma, f, i)) = arg(\Xi, f, i)$

In particular the above restrictions mean that every non theory operation symbol of Σ shall be part of Ξ. Subsequently, we will denote the set of non theory operation symbols from Σ, by Ω, i.e. $\Omega \coloneqq Op_\Sigma \setminus Op_R$. Moreover these restrictions require a correspondence between the arities of these symbols in the two signatures. Besides this syntactic correspondence, we also want that there is a certain semantic correspondence.

Definition 103. We define $\hat{\mathcal{C}}_{BV}$ as the set of all BV interpretations over the signature Ξ, with a structure $\mathcal{B}$, such that:

- If the arguments of a function are in the image of the corresponding β function then the value of the function itself shall be in the image of the corresponding β function, i.e.:

$$\forall f \in \Omega\, \forall v_1 \in \mathcal{R}_{arg(\Sigma,f,1)}, \cdots, v_{args(\Sigma,f)} \in \mathcal{R}_{arg(\Sigma,f,args(\Sigma,f))} :$$
$$f^{\mathcal{B}}(\beta_{arg(\Sigma,f,1)}(v_1), \cdots, \beta_{arg(\Sigma,f,args(\Sigma,f))}(v_{args(\Sigma,f)})) \in \beta_{arg(\Sigma,f,0)}(\mathcal{R}_{arg(\Sigma,f,0)})$$

- The operation symbols in Λ have a fixed semantic meaning. We will define their semantic meaning in the subsequent sections. •

The first part of the above definition can be interpreted in the following way: If we apply, a value that corresponds to a value from an appropriate R universe, to a BV interpretation of an operation symbol from Ω, then the function application itself, corresponds to a value from a suitable universe of the theory R.

Definition 104. Similarly we define $\hat{\mathcal{C}}_R$ as the set of all R interpretations of signature Σ. •

Remark Subsequently, we will only deal with expressions without quantifiers. Hence we will identify interpretations $\langle \mathcal{A}, M \rangle$, with their structure $\mathcal{A}$.

Now we are ready to give the structure of the translation process and to give pre- and postconditions for this translation.

Description of Algorithm 8 The algorithm takes a closed expression over the signature Σ and returns a closed expression over the signature Ξ. The resulting expression has a sort, which is related to the sort of the original expression, in the way we described above. If the original expression is a formula, then it must not contain any quantifiers. Moreover, in this case the resulting expression is again a formula without any quantifiers. With the last two postconditions we imply that satisfiability is preserved. If E is a formula, we can conclude from the second to last postcondition that, if there is an interpretation from $\hat{\mathcal{C}}_R$ that satisfies the formula, then there is an interpretation from $\hat{\mathcal{C}}_{BV}$ that satisfies the resulting formula. The last postcondition implies the opposite direction.

If the given expression E, is the application of a non-theory operation, then we translate the arguments and apply these translated arguments to the function again. We can do this, because we required that the non theory operation symbols from Σ are also part of Ξ with a suitable arity. For other shapes of E, we make use of sub-procedures, where the respective sub-procedure is chosen according to the sort of the expression. We will define these sub-procedures in the subsequent sections.

Lemma 105. *Let F be a formula fulfilling the preconditions of Algorithm 8 and let $E_{out} = Translate(F)$. Then we have:*

- $\nexists p \in Pos(E_{out}), x^i \in \mathcal{V}, F' \in \mathcal{L} : E_{out}\langle p \rangle = \forall_{\mathcal{L}} x^i F'$
- $\nexists p \in Pos(E_{out}), x^i \in \mathcal{V}, F' \in \mathcal{L} : E_{out}\langle p \rangle = \exists_{\mathcal{L}} x^i F'$

Algorithm 8 Theory Translation

Input: An expression E that is defined over the signature Σ of sort s
Require: If $s = 0$ then $\nexists p \in Pos(E), x^i \in \mathcal{V}, F' \in \mathcal{L} : E\langle p \rangle = \forall_{\mathcal{L}} x^i F'$
Require: If $s = 0$ then $\nexists p \in Pos(E), x^i \in \mathcal{V}, F' \in \mathcal{L} : E\langle p \rangle = \exists_{\mathcal{L}} x^i F'$
Require: $\mathit{free}(E) = \emptyset$
Output: An expression E_{out} over the signature Ξ of sort $\hat{\alpha}(s)$
Ensure: If $s = 0$ then $\nexists p \in Pos(E_{out}), x^i \in \mathcal{V}, F' \in \mathcal{L} : E_{out}\langle p \rangle = \forall_{\mathcal{L}} x^i F'$
Ensure: If $s = 0$ then $\nexists p \in Pos(E_{out}), x^i \in \mathcal{V}, F' \in \mathcal{L} : E_{out}\langle p \rangle = \exists_{\mathcal{L}} x^i F'$
Ensure: $\mathit{free}(E_{out}) = \emptyset$
Ensure: $\forall \mathcal{A} \in \hat{\mathcal{C}}_R \; \exists \mathcal{B} \in \hat{\mathcal{C}}_{BV} : \beta_s(\mathcal{A}(E)) = \mathcal{B}(E_{out})$
Ensure: $\forall \mathcal{B} \in \hat{\mathcal{C}}_{BV} \; \exists \mathcal{A} \in \hat{\mathcal{C}}_R : \beta_s(\mathcal{A}(E)) = \mathcal{B}(E_{out})$

```
1:  function TRANSLATE(E)
2:      if E = f(e_1, ··· , e_n) where f ∈ Ω then
3:          return f(TRANSLATE(e_1), ··· ,TRANSLATE(e_n))
4:      end if
5:      if E is a tuple access expression then
6:          return TRANSLATE-TUPLE(E)
7:      end if
8:      if E is a map access expression then
9:          return TRANSLATE-MAP(E)
10:     end if
11:     select case in s
12:         Case s is a boolean sort
13:             return TRANSLATE-BOOL(E)
14:         Case s is an integer sort
15:             return TRANSLATE-INT(E)
16:         Case s is a tuple sort
17:             return TRANSLATE-TUPLE(E)
18:         Case s is a map sort
19:             return TRANSLATE-MAP(E)
20:         Case s is a set sort
21:             return TRANSLATE-SET(E)
22:     end select
23: end function
```

- *Let $s \in \Sigma.Sort$. Then* $\textsc{Translate}(Exp_s^\Sigma) \subseteq Exp^\Xi_{\hat{\alpha}(s)}$.

Proof. The assertions of this lemma are a direct consequence of the definition of the translation Translate, respectively of the definitions of the sub-procedures. □

For the next lemma we need a generalisation of the *positions of formulae*, which we used until now. We can extend the notion of positions of formulae in a straight forward way to positions of expression. For an expression E, we denote this set of positions by $ExpPos(E)$. As this set can be defined almost analogously, as the set of positions of a formula, we do not give a definition. Similar as we defined sub-formulae, by means of the positions of formulae, we can also define sub-expressions, by means of the extended positions. But again, we will not give a definition.

Lemma 106. *The translation fulfils the following properties:*

- *Let $e \in \mathcal{L}^\Sigma$, let $f \in \Omega$, let $e_1, \cdots, e_{args(\Xi,f)} \in \mathcal{L}^\Xi$ of an appropriate sort and let $\hat{e} = \textsc{Translate}(e)$. Then we have*

$$\begin{aligned}&\forall p \in ExpPos(\hat{e}) : \hat{e}\langle p\rangle = f(e_1, \cdots, e_{args(\Xi,f)}) \Rightarrow \\ &\Rightarrow \exists q \in ExpPos(e) : \textsc{Translate}(e\langle q\rangle) = \hat{e}\langle p\rangle.\end{aligned}$$

- *Let $e \in \mathcal{L}^\Sigma$, let $f \in \Omega$, let $e_1, \cdots, e_{args(\Sigma,f)} \in \mathcal{L}^\Sigma$ of an appropriate sort and let $\hat{e} = \textsc{Translate}(e)$. Then we have*

$$\begin{aligned}&\forall p \in ExpPos(e) : e\langle p\rangle = f(e_1, \cdots, e_{args(\Sigma,f)}) \Rightarrow \\ &\Rightarrow \exists q \in ExpPos(\hat{e}) : \textsc{Translate}(e\langle q\rangle) = \hat{e}\langle p\rangle.\end{aligned}$$

Proof. The assertion of this lemma is a direct consequence of the definition of the translation Translate, respectively of the definitions of the sub-procedures. □

The above lemma means that, whenever a translated expression contains the application of an operation symbol from Ω, then there is a sub expression of the original expression, whose translation gives this application. The second part of the lemma states the opposite direction. Thus, whenever an expression contains an operation symbol from Ω, then its translation also contains this symbol and vice versa. From this property, we can conclude the following corollary.

Corollary 107. *Let $e \in \mathcal{L}^\Sigma$ Then we have:*

$$getOpSyms(e) \cap \Omega = getOpSyms(\textsc{Translate}(e)) \cap \Omega.$$

This finishes the part of syntactic properties of the translation. Next we will give some semantic properties.

Lemma 108. *Let $f \in Op_R$, let $e_1, \cdots, e_{args(\Sigma,f)} \in \mathcal{L}^\Sigma$ of an appropriate sort and let $\mathcal{B} \in \hat{\mathcal{C}}_{BV}$. Then we have:*

$$\begin{aligned}&(\forall\, 1 \leq i \leq args(\Sigma, f) : \mathcal{B}(\textsc{Translate}(e_i)) \in \beta_{arg(\Sigma,f,i)}(\mathcal{R}_{arg(\Sigma,f,i)})) \Rightarrow \\ &\Rightarrow \mathcal{B}(\textsc{Translate}(f(e_1, \cdots, e_{args(\Sigma,f)}))) \in \beta_{arg(\Sigma,f,0)}(\mathcal{R}_{arg(\Sigma,f,0)})\end{aligned}$$

The assertion of the above lemma is related to the first part of definition 103. As, here we have that, if the arguments of a translated operation somehow correspond to values from a universe of the theory R, then the operation application itself also does so.

We can now combine the result of the above lemma and the defining property of $\hat{\mathcal{C}}_{BV}$ and get the following result:

Corollary 109. *Let $s \in \Sigma.Sort$ and $e \in Exp_s^\Sigma$. Moreover let $\mathcal{B} \in \hat{\mathcal{C}}_B$ Then we have:*

$$\mathcal{B}(\textsc{Translate}(e)) \in \beta(\mathcal{R}_{\hat{\alpha}(s)})$$

If we would actually perform the necessary proofs we would also need the following lemma.

Lemma 110. *Let $e_1, e_2 \in \mathcal{L}^\Sigma$ with sorts s_1 and s_2. If for each interpretation $\mathcal{A} \in \hat{\mathcal{C}}_R$ there are interpretations $\mathcal{B}_1 \in \hat{\mathcal{C}}_{BV}$ and $\mathcal{B}_2 \in \hat{\mathcal{C}}_{BV}$, such that for each $1 \leq i \leq 2$ we have*

$$\begin{gathered}\beta_{s_i}(\mathcal{A}(e_i)) = \mathcal{B}_i(\textsc{Translate}(e_i)) \\ \forall f \in \mathit{getOpSyms}(e_1) \setminus Op_R \forall v_1 \in \mathcal{R}_{arg(\Sigma,f,1)}, \cdots, v_{args(\Sigma,f)} \in \mathcal{R}_{arg(\Sigma,f,args(\Sigma,f))} : \\ \beta(f^{\mathcal{A}}(v_1, \cdots, v_{args(\Sigma,f)}) = f^{\mathcal{B}_i}(\beta(v_1), \cdots, \beta(v_{args(\Sigma,f)}))\end{gathered}$$

Then there is an interpretation $\mathcal{B} \in \hat{\mathcal{C}}_{BV}$ such that for each $1 \leq i \leq 2$ we have

$$\begin{gathered}\beta_{s_i}(\mathcal{A}(e_i)) = \mathcal{B}(\textsc{Translate}(e_i)) \\ \forall f \in \mathit{getOpSyms}(e_1) \setminus Op_R \forall v_1 \in \mathcal{R}_{arg(\Sigma,f,1)}, \cdots, v_{args(\Sigma,f)} \in \mathcal{R}_{arg(\Sigma,f,args(\Sigma,f))} : \\ \beta(f^{\mathcal{A}}(v_1, \cdots, v_{args(\Sigma,f)}) = f^{\mathcal{B}}(\beta(v_1), \cdots, \beta(v_{args(\Sigma,f)}))\end{gathered}$$

Note that above we omitted the subscripts of the beta functions, as they follow from the context.

Proof. We will construct an interpretation $\mathcal{B}$ and show that this interpretation fulfils the necessary properties. $\mathcal{B}$ shall be an interpretation from $\hat{\mathcal{C}}_B$. Thus, the interpretations of symbols in $Op_{BV} \cup \Lambda$ are fixed. Operation symbols $f \in \Omega$, such that either $f \in \mathit{getOpSyms}(e_1)$ or $f \in \mathit{getOpSyms}(e_2)$ are interpreted equally, as by $\mathcal{B}_1$, respectively $\mathcal{B}_2$. From (Lemma 38) we can conclude that it does not matter for such symbols, whether we apply $\mathcal{B}_1$, respectively $\mathcal{B}_2$ or $\mathcal{B}$. So, only operation symbols g in $\mathit{getOpSyms}(e_1) \cap \mathit{getOpSyms}(e_2) \cap \Omega$ remain to be considered. Because of the hypothesis of the lemma, the interpretations of g by $\mathcal{B}_1$ and $\mathcal{B}_2$ partially coincide. For the arguments, for which the interpretation of g by $\mathcal{B}_1$ and $\mathcal{B}_2$ coincide, the interpretation of g by $\mathcal{B}$ shall be equal to $\mathcal{B}_1$. For the other values the interpretation can be arbitrary. Thus, the interpretation $\mathcal{B}$ fulfils the second assertion of the lemma. From Lemma 106 and Corollary 109 we can conclude that when we interpret e_1 with $\mathcal{B}_1$, the interpretations of function symbols from Ω are only applied with arguments, which are in the images of the respective β functions. A similar property holds for

e_2 and $\mathcal{B}_2$. This means, that the interpretations of these operation symbols , with respect to $\mathcal{B}_1$ and $\mathcal{B}_2$, are effectively equal, as the possible differences do not matter. So, if we sum everything up, we also get, in combination with Lemma 38, the first assertion of the lemma. □

Lemma 111. *We can generalise Lemma 110 to the case of n expressions, where* $n > 2$.

Similarly, we can show the other direction, namely if there is an interpretation $\mathcal{B} \in \hat{\mathcal{C}}_{BV}$, then there is suitable $\mathcal{A} \in \hat{\mathcal{C}}_R$

Now we can state the main theorem of this section. We will use this theorem to show that the translation preserves satisfiability.

Theorem 112. *For each sort* $s \in \Sigma.Sort$ *and for each expression* $e \in Exp_s^\Sigma$ *we have:*

- *For each interpretation* $\mathcal{A} \in \hat{\mathcal{C}}_R$ *there is an interpretation* $\mathcal{B} \in \hat{\mathcal{C}}_{BV}$ *such that:*

$$\forall f \in getOpSyms(e) \setminus Op_R \, \forall v_1 \in \mathcal{R}_{arg(\Sigma,f,1)}, \cdots, v_{args(\Sigma,f)} \in \mathcal{R}_{arg(\Sigma,f,args(\Sigma,f))} :$$
$$\beta(f^{\mathcal{A}}(v_1, \cdots, v_{args(\Sigma,f)}) = f^{\mathcal{B}}(\beta(v_1), \cdots, \beta(v_{args(\Sigma,f)}))$$
$$and \; \beta_s(\mathcal{A}(e)) = \mathcal{B}(\textsc{Translate}(e))$$

- *For each interpretation* $\mathcal{B} \in \hat{\mathcal{C}}_{BV}$ *there is an interpretation* $\mathcal{A} \in \hat{\mathcal{C}}_R$ *such that:*

$$\forall f \in getOpSyms(e) \setminus Op_R \, \forall v_1 \in \mathcal{R}_{arg(\Sigma,f,1)}, \cdots, v_{args(\Sigma,f)} \in \mathcal{R}_{arg(\Sigma,f,args(\Sigma,f))} :$$
$$\beta(f^{\mathcal{A}}(v_1, \cdots, v_{args(\Sigma,f)}) = f^{\mathcal{B}}(\beta(v_1), \cdots, \beta(v_{args(\Sigma,f)}))$$
$$and \; \beta_s(\mathcal{A}(e)) = \mathcal{B}(\textsc{Translate}(e))$$

Note that above we omitted the subscripts of the beta functions, as they follow from the context.

Proof. If $exp = f(e_1, \cdots, e_n)$, where $f \in \Omega$ then the assertion of the theorem follows immediately with the assumption that the assertion holds for the sub-expressions $e_1, \cdots, e_n$ and with Lemma 111. For the other cases, we have to show the assertion of the theorem for the sub-procedures. Note that we will not actually make the necessary proofs. Instead, we will only point out some aspects of these proofs in the subsequent sections. □

Since $\beta_0(\mathbb{T}) = \mathbb{T}$ and $\beta_0(\mathbb{F}) = \mathbb{F}$, a direct result of Theorem 112 is that a closed formula $F \in Exp_0^\Sigma$ is satisfiable by an interpretation from $\hat{\mathcal{C}}_R$, iff the translation of the formula is satisfiable by an interpretation from $\hat{\mathcal{C}}_{BV}$.

Remark We saw that the subscripts, which determine the sorts, can make things a bit tedious. Thus, from now on, we will usually omit them.

Remark In following sections we will only be interested in expressions from $\mathcal{L}^{\Sigma}$, respectively from $\mathcal{L}^{\Xi}$. As we are also only interested in interpretations from $\hat{\mathcal{C}}_R$, respectively $\hat{\mathcal{C}}_{BV}$, any sort that occurs in considered expressions, corresponds to a uniquely defined universe. Therefore, besides retrieving the sort of an expression, we can also retrieve the universe of the expression (namely the uniquely defined universe of the expression's sort). Moreover, we will use the sets of the respective universes, to refer to the respective sorts.

Remark We will denote expression over the signature Σ as R *expressions* and expressions over the signature Ξ as BV *expressions*. For a non boolean BV expression the vector length of this expression shall be the length of the bit vectors in its universe.

Remark The sub-procedures presented in the subsequent sections have pre- and postcondition, like the procedure Translate. Nevertheless, we will not repeat these conditions for the individual procedures.

5.4 The Translation of Booleans

In this section we will discuss the translation of boolean R expressions. As boolean values are both part of the theory R and the theory BV, we can deal with them in a straight forward way.

First we will discuss the functions α and β. Then we will deal with the procedure Translate-Bool.

Definition 113. We define the function α for the universe of the boolean sort, respectively for the universes that correspond to the subtypes of the booleans as:

$\alpha(\{\mathbb{T},\mathbb{F}\}) \coloneqq \{\mathbb{T},\mathbb{F}\}$
$\alpha(\{\mathbb{T}\}) \coloneqq \{\mathbb{T},\mathbb{F}\}$
$\alpha(\{\mathbb{F}\}) \coloneqq \{\mathbb{T},\mathbb{F}\}$ •

Definition 114. We define the β functions for the boolean sort and for the sorts that correspond to the sort of the universe of the boolean subtypes as:

$\beta_{\{\mathbb{T},\mathbb{F}\}}(\mathbb{T}) \coloneqq \mathbb{T}$ and $\beta_{\{\mathbb{T},\mathbb{F}\}}(\mathbb{F}) \coloneqq \mathbb{F}$
$\beta_{\{\mathbb{T}\}}(\mathbb{T}) \coloneqq \mathbb{T}$
$\beta_{\{\mathbb{F}\}}(\mathbb{F}) \coloneqq \mathbb{F}$ •

Note that in the above definition for simplicity we identified the respective sorts with the corresponding universes.

Next we can describe Algorithm 9. This procedure we will only use to translate the boolean constants and the logical connectives. Predicates of the theory R, like the less-then predicate, we will treat when we discuss the translation of expressions with the sorts of the predicate's arguments. This means that we will deal with the less-then predicate when we deal with integer expressions.

Algorithm 9 Translation of Booleans

Input: A formula F that is defined over the signature Σ
Output: A formula F_{out} over the signature Ξ

1: **function** TRANSLATE-BOOL(F)
2: **select case in** F
3: **Case** $F = \mathbb{T}$
4: **return** $\mathbb{T}$
5: **Case** $F = \mathbb{F}$
6: **return** $\mathbb{F}$
7: **Case** $F = \neg_{\mathcal{L}} F_1$
8: **return** $\neg_{\mathcal{L}}$TRANSLATE-BOOL(F_1)
9: **Case** $F = F_1 \wedge_{\mathcal{L}} F_2$
10: **return** TRANSLATE-BOOL(F_1)$\wedge_{\mathcal{L}}$TRANSLATE-BOOL(F_2)
⋮
11: **Case** $F = F_1 \Leftrightarrow_{\mathcal{L}} F_2$
12: **return** TRANSLATE-BOOL($F_1 \Rightarrow_{\mathcal{L}} F_2$)$\wedge_{\mathcal{L}}$ TRANSLATE-BOOL($F_2 \Rightarrow_{\mathcal{L}} F_1$)
13: **Default**
14: **select case in** F
15: **Case** F is a tuple- or map access or a non-theory operation
16: **return** TRANSLATE(F)
17: **Case** F is an integer predicate
18: **return** TRANSLATE-INT(F)
19: **Case** F is a tuple predicate
20: **return** TRANSLATE-TUPLE(F)
21: **Case** F is a map predicate
22: **return** TRANSLATE-MAP(F)
23: **Case** F is a set predicate
24: **return** TRANSLATE-SET(F)
25: **end select**
26: **end select**
27: **end function**

Remark This algorithm has the same pre- and postconditions as Algorithm 8. Hence, we will not repeat them here.

We will not discuss any aspects of the necessary proofs for this sub-procedure here.

5.5 The Translation of Integers

Again we will start with the definitions of the functions α and β. Then we will give the overall idea of the translation. Afterwards we will present the procedure Translate-Int. Finally, we will discuss some aspects of the proofs, that would be necessary.

We start with the representation of integers by bit vectors. We will consider two such representations, on the one hand an unsigned representation (which corresponds to the representation of unsigned integers in C++) and on the other hand the two's complements signed integer representation. Subsequently, we define a function *BV2Nat* that assigns an integer with respect to the unsigned integer representation to a bit vector. Then we define a function *BV2Int* that assigns an integer with respect to the signed integer representation to a bit vector.

Definition 115. Let n be a positive integer and $b \in \mathit{BitVec}(n)$. Then we define the function *BV2Nat* as

$$\mathit{BV2Nat}(b) \coloneqq \sum_{i=0}^{n-1} b(i)2^i$$

and we define the function *BV2Int* as

$$\mathit{BV2Int}(b) \coloneqq -b(n-1)2^{n-1} + \sum_{i=0}^{n-2} b(i)2^i. \qquad \bullet$$

Actually we would need for each vector length separate functions, but for simplicity we identify the different functions with *BV2Nat* respectively with *BV2Int*. If we want to point out the vector length, for which the functions are defined, we put the length in a subscript, e.g. $\mathit{BV2Nat}_3$.

As a consequence of the above definition, we see that a bit vector b of length n can represent the numbers in

$$\{x \in \mathbb{N} \mid x \leq 2^n - 1\}$$

in the unsigned representation and the numbers in

$$\{x \in \mathbb{Z} \mid -2^{n-2} \leq x \leq 2^{n-2} - 1\}$$

in the signed representation.

As here we want to describe a translation form integers to bit vectors we are mainly interested in the opposite direction, i.e. assigning a bit vector to an integer (according to the signed respective the unsigned representation).

As we saw above we need a bit vector of sufficient length to represent a given integer. Thus, in order to represent integers by bit vectors, we first have to determine a vector length that is sufficiently large to represent this integer. Therefore, we define a function *VectorLength* that takes two numbers a, b and gives the minimal vector length, that is sufficient to represent the integers in $I[a, b]$.

Definition 116. Let $a, b \in \mathbb{Z}$ such that $a \leq b$. Then we define *VectorLength(a,b)* as:

$$\mathit{VectorLength}(a,b) \coloneqq \begin{cases} 1, & \text{if } a = b = 0 \\ \lfloor \log_2 b \rfloor + 1, & \text{if } a \geq 0 \wedge b > 0 \\ \lfloor \log_2 b \rfloor + 2, & \text{if } a < 0 \wedge |a| - 1 \leq b \\ \lceil \log_2 |a| \rceil + 1, & \text{if } a < 0 \wedge |a| - 1 > b \end{cases}$$

•

Theorem 117. *Let $a, b \in \mathbb{Z}$ such that $a \leq b$ and let $n = \mathit{VectorLength}(a, b)$. Then n is the minimal vector length to represent the numbers in $I[a, b]$. Thus for $0 \leq a$ we have:*

- $I[a, b] \subseteq \mathit{BV2Nat}(\mathit{BitVec}(n))$
- $\forall \hat{n} \in \mathbb{N}^* : \hat{n} < n \Rightarrow I[a, b] \nsubseteq \mathit{BV2Nat}(\mathit{BitVec}(\hat{n}))$

and for $a < 0$ we have

- $I[a, b] \subseteq \mathit{BV2Int}(\mathit{BitVec}(n))$
- $\forall \hat{n} \in \mathbb{N}^* : \hat{n} < n \Rightarrow I[a, b] \nsubseteq \mathit{BV2Int}(\mathit{BitVec}(\hat{n}))$

Proof. Let $a, b \in \mathbb{Z}$

- *Case:* $a = b = 0$
  For this case the assertion of the theorem obviously holds.
- *Case:* $a \geq 0 \wedge b > 0$
  Let $m \in \mathbb{N}^*$ then we know that

  $$\mathit{Range}(\mathit{BV2Nat}_m) = I[0, 2^m - 1]$$

  Thus in order to prove the first part of the theorem, we have to prove

  $$I[a, b] \subseteq I[0, 2^{\mathit{VectorLength}(a,b)} - 1].$$

  This we prove by showing that

  $$b \leq 2^{\lfloor \log_2 b \rfloor + 1} - 1.$$

  By using the properties of the floor function we get

  $$2^{\lfloor \log_2 b \rfloor} \leq b < 2^{\lfloor \log_2 b \rfloor + 1}$$

And from this it immediately follows that

$$b \leq 2^{\lfloor \log_2 b \rfloor + 1} - 1$$

The second part of the theorem we can conclude from the above proof, as we have

$$2^{\lfloor \log_2 b \rfloor} - 1 < b.$$

- *Case:* Remaining cases
  Can be proven similar. □

Now we can define the function α for integer universes.

Definition 118. Let $I \subset \mathbb{Z}$ be a finite set. Additionally, let $a = \min I$ and let $b = \max I$. Then we have:

$$\alpha(I) \coloneqq \mathit{BitVec}(\mathit{VectorLength}(a, b)). \quad \bullet$$

Remark Note that in the above definition we had to use arbitrary sets of integers, instead of integer intervals. This we had to do, as we not only have to deal with the universes that correspond to integer types but also with the universes that correspond to subtypes of integer types.

We will continue with the β functions. In order to define these functions we will give a procedure INT2BV that takes an integer i and a sufficiently large vector length n and returns a bit vector b, which has length n. If i is negative then b is the two's complement representation of i. If i is non-negative then b is the unsigned representation of i. To define the procedure INT2BV, we will first define a procedure NAT2BV that takes a non-negative integer i and a sufficiently large positive integer n. This procedure shall then return a bit vector b of length n, where b shall be the unsigned representation of i.

Description of Algorithm 10 The procedure NAT2BV works in a straight forward way. We compute the bit vector representation iteratively. We start with the computation of the first bit of the sequence and finish with the last bit. The respective bits are given by the rest of division by two.

If the argument i of INT2BV is non-negative, we can use the auxiliary procedure NAT2BV to compute the required bit vector. For the other case we have to use properties of the two's complement integer representation. On the one hand a negative number i has always a bit representation, whose leading bit is 1. On the other hand we also know that the remaining bits represent $2^{n-1} + i$, where n is again the length of the vector. Therefore, in the case $i < 0$ we compute the bit vector representation for $2^{n-1} + i$ and prepend a one to this bit vector.

With this we are ready to define the β functions for integer universes.

Algorithm 10 Bit Vector Generation

Input: A non-negative integer i and a positive integer n
Require: $\mathit{VectorLength}(i, i) \leq n$
Output: A Bit Vector b of length n
Ensure: $\mathit{BV2Nat}(b) = i$
1: **function** NAT2BV(i, n)
2: $k \leftarrow i \bmod 2$
3: $i \leftarrow (i - k)/2$
4: $bv \leftarrow k$
5: **for** $j \leftarrow 2$ to n **do**
6: $k \leftarrow i \bmod 2$
7: $i \leftarrow (i - k)/2$
8: ▷ In the following <> denotes concatenation of a bit vector and a single bit
9: $bv \leftarrow bv <> k$
10: **end for**
11: **return** bv
12: **end function**

Input: An integer i and a positive integer n
Require: $\mathit{VectorLength}(i, i) \leq n$
Output: A Bit Vector b of length n
Ensure: $i \geq 0 \Rightarrow \mathit{BV2Nat}(b) = i$
Ensure: $i \geq 0 \wedge \mathit{VectorLength}(i, i) < n \Rightarrow \mathit{BV2Int}(b) = i$
Ensure: $i < 0 \Rightarrow \mathit{BV2Int}(b) = i$
13: **function** INT2BV(i, n)
14: **if** $i \geq 0$ **then**
15: **return** NAT2BV(i, n)
16: **else**
17: **return** 1 <>NAT2BV($2^{n-1} + i, n - 1$)
18: **end if**
19: **end function**

Definition 119. Let $I \subset \mathbb{Z}$ be a finite set and let s be the sort of I. Then we define β_s as:

$$\beta_s(x) \coloneqq \textsc{Int2BV}(x, \mathit{VectorLength}(\min I, \max I)) \qquad \bullet$$

Before we will introduce the procedure Translate-Int, we want to explain the idea of the translation of integer expressions.

In both theories R and BV, there are operations for the arithmetic functions and for the comparison predicates. Thus, we want to somehow translate the operations from the one theory, to the corresponding operations from the other theory. This in particular requires the following two problems to be solved:

- The arithmetic operations in the theory BV correspond to a modular arithmetic, while the arithmetic operations in R correspond to the "usual" arithmetic functions.
- The respective operations in the BV theory require bit vectors of equal length as arguments, while in R there are integer operations that can take arguments with different sorts.

We illustrate these problems with two examples:

Example 120. Let us consider the expression $2_{I[2,2]} + 2_{I[2,2]}$ containing R operations. (The subscripts give the sorts of the constants. Remember that we required that there is a constant symbol for each value of the universes of each sort in R. Thus there are actually several 1 and 2. Because of this, we want to point out the sort.) By using the results from before, we can model $2_{I[2,2]}$ by the bit vector 10. As we have two times the same bit vector, we can use the addition function *bvadd* from the BV theory. Now the problem is that $\mathit{bvadd}(10, 10) = 00$ and 00 does not correspond to four.

Example 121. Let us consider the expression $1_{I[1,1]} + 2_{I[2,2]}$. Using the same results as before we can map $1_{I[1,1]}$ to the bit vector 1 and $2_{I[2,2]}$ to the bit vector 10. We can see that the bit vectors have different lengths, thus we cannot use the addition operation from BV.

To resolve these problems we now want to find a suitable bit vector length and associate all integer sub-expressions, in a given integer expression, to bit vector expressions of this length. This obviously solves the second problem. If the length is sufficiently large, obviously also the other problem is solved. In order to determine such a length, we compute the minimal smallest value s of the universes of an expression and its integer sub-expressions. Similar we compute the maximal greatest value g. We compute these values with the procedure Bounds. The vector length, we are looking for is then given by $\mathit{VectorLength}(s, g)$. As this vector length is sufficiently large to represent all integers in $I[s, g]$, we can be sure that no overflows occur.

Algorithm 11 Bounds

Input: An expression e over the signature Σ with an integer universe I
Output: A pair of integers $\langle a, b \rangle$.
Ensure: a is the smallest minimal value of the domains of sub expressions excluding the arguments of non theory operations.
Ensure: b is the greatest maximal value of the domains of sub expressions excluding the arguments of non theory operations.

```
1: function BOUNDS(e)
2:     if e is a constant, tuple- or map access or a non-theory operation then
3:         return ⟨min I, max I⟩
4:     end if
5:     a ← min I
6:     b ← max I
7:     for each sub expression s of e do
8:         ⟨x, y⟩ ←BOUNDS(s)
9:         if x < a then
10:            a ← x
11:        end if
12:        if y > b then
13:            b ← y
14:        end if
15:    end for
16:    return ⟨a, b⟩
17: end function
```

Description of Algorithm 11 In this procedure we recursively traverse the given expression until either a constant, a tuple- or a map access, or a non-theory operation is encountered. In each recursive step we compute the smallest respectively the largest value from the integer universe of the given expression. If the expression has integer sub-expression we recursively compute the smallest respectively the largest value of the integer domains involved in this sub-expression. Finally, we compare the result of the recursive computations with the values computed before and return the smallest respectively the largest one.

Before we can now introduce the actual translation procedure we will introduce a procedure to translate expressions with an integer sort to bit vector expressions of a specific length.

Description of Algorithm 12 ProcIntExp computes a bit vector representation of *exp* that has length *len*. If *sign* is negative, we use the signed bit vector representation and otherwise the unsigned one. If we encounter an operation from the theory R, we first compute bit vector expressions for the operation's arguments. Then we apply the corresponding operation from the theory BV to the these bit vector expressions. If we encounter a constant, we can use the procedure Int2BV to translate it to a bit vector. The only smaller difficulty arises when we encounter tuple- or map access or a non-theory operation. In this case Translate(*exp*) can be a bit vector expression, whose vector length is smaller than *len*. Thus, we have to resize it such that the vector length is equal to *len*. To achieve this we apply the procedure Resize.

Resize computes a BV expression of length m from a R expression e. As before *sign* determines if a signed or an unsigned representation shall be used. We use the operations *zero_extend*, respectively *sign_extend* from the BV theory to extend the given BV expression to a BV expression of length m.

Now we are ready to introduce the actual translation procedure.

Description of Algorithm 13 In each case we use the procedure Bounds and the function *VectorLength* in order to determine a vector length, which is sufficient to represent all the involved integer-expressions. We have to differentiate between two cases, either *exp* is a boolean expression (e.g. $e_1 \leq e_2$) or it is an integer expression. If *exp* is a boolean expression, we can use the procedure ProcIntExp to compute the bit vector representations of the integer sub-expressions. These bit vector representations we can then use as arguments of the corresponding predicates of the BV theory.

In the other case we also have to consider a further step. Similar to the previous case we compute a bit vector representation of e. Now the problem is that this representation could have a vector length, which is longer than necessary. The reason for this is that there could be a sub-expression, that requires a longer length. We illustrate this with the following example:

Algorithm 12 Translation of Integers – using fixed Vector Lengths.

Input: An expression exp over the signature Σ, where e has either an integer universe I and a positive integer len and an integer $sign$.
Require: $sign \leq \min I$
Require: $len \geq VectorLength(\textsc{Bounds}(e))$
Require: $sign < 0 \wedge \min I \geq 0 \Rightarrow len > VectorLength(\textsc{Bounds}(e))$
Output: A expression over the signature Ξ with universe $BitVec(len)$

1: **function** ProcIntExp($exp, len, sign$)
2: **select case in** exp
3: **Case** $exp = c$ where c is a constant
4: ▷ Note that here we identify the syntactic constant symbols with their actual values
5: **return** Int2BV(c, len)
6: **Case** exp is a tuple- or map access or a non-theory operation
7: $t \leftarrow$Translate(exp)
8: **return** Resize($t, len, sign$)
9: **Case** $exp = e_1 + e_2$
10: $bv_1 \leftarrow$ProcIntExp($e_1, len, sign$) ; $bv_2 \leftarrow$ProcIntExp($e_2, len, sign$)
11: **return** $bvadd(bv_1, bv_2)$
⋮
12: **Case** $exp = e_1/e_2$
13: $bv_1 \leftarrow$ProcIntExp($e_1, len, sign$) ; $bv_2 \leftarrow$ProcIntExp($e_2, len, sign$)
14: **if** $sign < 0$ **then**
15: **return** $bvsdiv(bv_1, bv_2)$
16: **else**
17: **return** $bvudiv(bv_1, bv_2)$
18: **end if**
19: **end select**
20: **end function**

Input: A expression e of signature Ξ with sort $BitVec(n)$, a positive integer m and an integer $sign$.
Require: $m \geq n$
Output: A BV expression g of sort $BitVec(m)$

21: **function** Resize($e, m, sign$)
22: $diff \leftarrow m - n$
23: **if** $sign < 0$ **then**
24: **return** $sign\_extend_{diff}(e)$
25: **else**
26: **return** $zero\_extend_{diff}(e)$
27: **end if**
28: **end function**

Algorithm 13 Integer Expression Translation

Input: An expression e over the signature Σ where e has either an integer universe I or e is a comparison predicate applied to expressions with integer universes.

Output: An expression over the signature Ξ

1: **function** TRANSLATE-INT(e)
2: **select case in** e
3: **Case** $exp = e_1 \leq e_2$
4: $\langle a_1, b_1 \rangle \leftarrow$BOUNDS($e_1$); $\langle a_2, b_2 \rangle \leftarrow$BOUNDS($e_2$)
5: $a \leftarrow \min\{a_1, a_2\}$; $b \leftarrow \max\{b_1, b_2\}$
6: $len \leftarrow$ *VectorLength*(a, b)
7: $bv_1 \leftarrow$PROCINTEXP(e_1, len, a)
8: $bv_2 \leftarrow$PROCINTEXP(e_2, len, a)
9: **if** $a < 0$ **then**
10: **return** $bvsle(bv_1, bv_2)$
11: **else**
12: **return** $bvule(bv_1, bv_2)$
13: **end if**
$\vdots$
14: **Case** $exp = e_1 \neq e_2$
15: $\langle a_1, b_1 \rangle \leftarrow$BOUNDS($e_1$); $\langle a_2, b_2 \rangle \leftarrow$BOUNDS($e_2$)
16: $a \leftarrow \min\{a_1, a_2\}$; $b \leftarrow \max\{b_1, b_2\}$
17: $len \leftarrow$ *VectorLength*(a, b)
18: $bv_1 \leftarrow$PROCINTEXP(e_1, len, a)
19: $bv_2 \leftarrow$PROCINTEXP(e_2, len, a)
20: **return** $distinct(bv_1, bv_2)$
21: **Default**
22: $\langle a, b \rangle \leftarrow$BOUNDS($e$)
23: $len \leftarrow$ *VectorLength*(a, b)
24: $newLen \leftarrow$ *VectorLength*$(\min I, \max I)$
25: ▷ The indices of the extract function determine the extracted bits.
26: **return** $extract_{\langle newlen-1,0 \rangle}$ PROCINTEXP(e, len, a)
27: **end select**
28: **end function**

Example 122. Obviously we can represent $2 - 2$ by a bit vector of length one, as we can represent zero by a bit vector of length one. But to represent 2 we need bit vector of length two. Thus, we represent the expression by a BV subtraction expression with length two.

Therefore we cut off the unnecessary leading part of the bit vector representation, by using the *extract* function from the BV theory.

In the following we will state some theorems that would be necessary to prove Theorem 112. But we will not prove these theorems. Instead, we will only illustrate how some of the proofs could be done. Note that actually we would also need further theorems, like an analogue to Lemma 108. But we will not discuss these theorems.

Lemma 123. *In the following let $I \subset \mathbb{Z}$ be a finite set, $a, b \in I$ and let $n \geq$ VectorLength$(\min I, \max I)$. If one of the following assertions hold*

- $0 \leq \min I$ *and* $a + b \leq 2^{n-1} - 1$
- $\min I < 0$ *and* $-2^{n-2} \leq a + b \leq 2^{n-2} - 1$

then we have

$$\textsc{Int2BV}(a + b, n) = bvadd(\textsc{Int2BV}(a, n), \textsc{Int2BV}(b, n)).$$

The result of this lemma corresponds to our usual understanding of arithmetic on bit vectors. Nevertheless, we will not prove this lemma. We can change the conditions of the above lemma and get the following corollary:

Corollary 124. *In the following let $I \subset \mathbb{Z}$ be a finite set, $a, b \in I$ and let $n \geq$ VectorLength$(\min I, \max I)$. If $a + b \in I$ then we have*

$$\textsc{Int2BV}(a + b, n) = bvadd(\textsc{Int2BV}(a, n), \textsc{Int2BV}(a, n)).$$

Proof. We will only prove the case $0 \leq \min I$, the other case can be proven similar. By using the properties of the floor function we get

$$a + b \leq 2^{\lfloor \log_2(a+b) \rfloor + 1} - 1$$

Moreover we know that $a + b \leq \max I$, therefore we get

$$2^{\lfloor \log_2(a+b) \rfloor + 1} - 1 \leq 2^{VectorLength(\min I, \max I)} - 1 \leq 2^n - 1$$

This means that one of the conditions of Lemma 123 hold. Thus, we can conclude the assertion of the theorem. □

Lemma 125. *Let $e \in Exp_s^{\Sigma}$ be an integer expression, n and s be integers such that e, n, s fulfil the precondition of* ProcIntExp. *Then we have:*

- *For each interpretation $\mathcal{A} \in \hat{\mathcal{C}}_R$ there is an interpretation $\mathcal{B} \in \hat{\mathcal{C}}_{BV}$ such that:*

$$\forall f \in \mathit{getOpSyms}(e) \setminus \mathit{Op}_R \, \forall v_1 \in \mathcal{R}_{\mathit{arg}(\Sigma,f,1)}, \cdots, v_{\mathit{args}(\Sigma,f)} \in \mathcal{R}_{\mathit{arg}(\Sigma,f,\mathit{args}(\Sigma,f))} :$$
$$\beta(f^{\mathcal{A}}(v_1, \cdots, v_{\mathit{args}(\Sigma,f)}) = f^{\mathcal{B}}((v_1), \cdots, \beta(v_{\mathit{args}(\Sigma,f)}))$$
$$\textsc{Int2BV}(\mathcal{A}(e), n) = \mathcal{B}(\textsc{ProcIntExp}(e, n, s))$$

- *For each interpretation $\mathcal{B} \in \hat{\mathcal{C}}_{BV}$ there is an interpretation $\mathcal{A} \in \hat{\mathcal{C}}_R$ such that:*

$$\forall f \in \mathit{getOpSyms}(e) \setminus \mathit{Op}_R \, \forall v_1 \in \mathcal{R}_{\mathit{arg}(\Sigma,f,1)}, \cdots, v_{\mathit{args}(\Sigma,f)} \in \mathcal{R}_{\mathit{arg}(\Sigma,f,\mathit{args}(\Sigma,f))} :$$
$$\beta(f^{\mathcal{A}}(v_1, \cdots, v_{\mathit{args}(\Sigma,f)}) = f^{\mathcal{B}}(\beta(v_1), \cdots, \beta(v_{\mathit{args}(\Sigma,f)}))$$
$$\textsc{Int2BV}(\mathcal{A}(e), n) = \mathcal{B}(\textsc{ProcIntExp}(e, n, s))$$

Note that above we omitted the subscripts of the beta functions, as they follow from the context.

Proof. We only illustrate the first part of the theorem for addition-expressions. We assume that $e = e_1 + e_2$. We denote with I the universe of e and with I_1 and I_2 the universes of e_1 respectively e_2. We would prove this theorem inductively, thus we assume that the theorem holds for e_1 respectively e_2. Let $\mathcal{A} \in \hat{\mathcal{C}}_R$. Then by a similar reasoning as in Lemma 110 we get that there is an interpretation $\mathcal{B} \in \hat{\mathcal{C}}_{BV}$ that fulfils the condition, both for e_1 and e_2.

We also know that for any $x \in I_1$ and $y \in I_2$ we have $x + y \in I$. We know that for $(l, u) = \textsc{Bounds}(e)$, I, I_1 and I_2 are contained in $I[l, u]$. Additionally we have that:

$$\textsc{Int2BV}(\mathcal{A}(e), n) = \textsc{Int2BV}(\mathcal{A}(e_1) + \mathcal{A}(e_2), n)$$
$$\mathcal{B}(\textsc{ProcIntExp}(e, n, s)) =$$
$$\mathit{bvadd}(\mathcal{B}(\textsc{ProcIntExp}(e_1, n, s)), \mathcal{B}(\textsc{ProcIntExp}(e_2, n, s)))$$

By the hypothesis we get:

$$\mathcal{B}(\textsc{ProcIntExp}(e, n, l)) = \mathit{bvadd}(\textsc{Int2BV}(\mathcal{A}(e_1), n), \textsc{Int2BV}(\mathcal{A}(e_2), n))$$

Thus by Corollary 124 we get that

$$\mathcal{B}(\textsc{ProcIntExp}(e, n, s)) = \textsc{Int2BV}(\mathcal{A}(e_1) + \mathcal{A}(e_2), n)$$

From this we can conclude:

$$\mathcal{B}(\textsc{ProcIntExp}(e, n, s)) = \textsc{Int2BV}(\mathcal{A}(e), n)$$

The other assertion of the theorem obviously holds as e contains exactly those non-theory functions that occur either in e_1 or in e_2.

□

Lemma 126. *Let e be an integer expression with universe I, let* $\textsc{Bounds}(e) = (l, u)$*, let* $n = \mathit{VectorLength}(\min I, \max I)$ *and let* $\mathcal{A} \in \hat{\mathcal{C}}_R$ *and* $\mathcal{B} \in \hat{\mathcal{C}}_{BV}$*. If we have for* $m > n$ *that*

$$\textsc{Int2BV}(\mathcal{A}(e), m) = \mathcal{B}(\textsc{ProcIntExp}(e, m, l))$$

then we have:

$$\textsc{Int2BV}(\mathcal{A}(e), n) = \mathcal{B}(\mathit{extract}_{\langle n-1,0\rangle}(\textsc{ProcIntExp}(e, m, l)))$$

The result of this lemma is somehow intuitive. The hypothesis means that an interpretation of the translation of an expression is equal to the bit vector representation of an interpretation of the expression itself, where the length of the bit vectors is larger than necessary. For such interpretations we expect that such an equality also holds if we use bit vectors whose length is the minimal sufficient length to represent the involved numbers. Even though the result of this lemma is somehow intuitive, we will not provide a proof.

Theorem 127. *Let s be an integer sort or the boolean sort and let* $e \in \mathit{Exp}_s^\Sigma$ *such that e is either an integer expression or the comparison of integer expressions. Then we have:*

- *For each interpretation* $\mathcal{A} \in \hat{\mathcal{C}}_R$ *there is an interpretation* $\mathcal{B} \in \hat{\mathcal{C}}_{BV}$ *such that:*

$$\forall f \in \mathit{getOpSyms}(e) \setminus \mathit{Op}_R\ \forall v_1 \in \mathcal{R}_{\mathit{arg}(\Sigma,f,1)}, \cdots, v_{\mathit{args}(\Sigma,f)} \in \mathcal{R}_{\mathit{arg}(\Sigma,f,\mathit{args}(\Sigma,f))} :$$
$$\beta(f^{\mathcal{A}}(v_1, \cdots, v_{\mathit{args}(\Sigma,f)}) = f^{\mathcal{B}}(\beta(v_1), \cdots, \beta(v_{\mathit{args}(\Sigma,f)}))$$
$$\beta_s(\mathcal{A}(e)) = \mathcal{B}(\textsc{Translate-Int}(e))$$

- *For each interpretation* $\mathcal{B} \in \hat{\mathcal{C}}_{BV}$ *there is an interpretation* $\mathcal{A} \in \hat{\mathcal{C}}_R$ *such that:*

$$\forall f \in \mathit{getOpSyms}(e) \setminus \mathit{Op}_R\ \forall v_1 \in \mathcal{R}_{\mathit{arg}(\Sigma,f,1)}, \cdots, v_{\mathit{args}(\Sigma,f)} \in \mathcal{R}_{\mathit{arg}(\Sigma,f,\mathit{args}(\Sigma,f))} :$$
$$\beta(f^{\mathcal{A}}(v_1, \cdots, v_{\mathit{args}(\Sigma,f)}) = f^{\mathcal{B}}(\beta(v_1), \cdots, \beta(v_{\mathit{args}(\Sigma,f)}))$$
$$\beta_s(\mathcal{A}(e)) = \mathcal{B}(\textsc{Translate-Int}(e))$$

Note that above we omitted the subscripts of the beta functions, as they follow from the context.

Proof. Similar as for the previous lemma we will not give an actual proof. Instead we will only illustrate the proof for the first case of the theorem for integer sorts and integer expressions.

Let I be the universe of s, $(l, u) = \textsc{Bounds}(e)$ and $n = \mathit{VectorLength}(l, u)$. We see that we can conclude the first assertion of the theorem from Lemma 125. Moreover, we can conclude from Lemma 125 that

$$\textsc{Int2BV}(\mathcal{A}(e), n) = \mathcal{B}(\textsc{ProcIntExp}(e, n, \mathit{sign}))$$

Now we can conclude with Lemma 126 the assertion of the theorem. □

5.6 The Translation of Tuples

Before we will discuss the functions α and β for tuples we want to describe the overall idea for the encoding of tuples by bit vectors.

Let T denote a universe of tuple values, whose components have universes $T_1, \cdots T_n$. First we assume that the universes of the components are different from sets of boolean values. So we can assume that we can represent the values of the universes $T_1, \cdots T_n$ as bit vectors. We can now represent a value v from T by the concatenation of the bit vector representations of the components of v. As we have seen, we do not represent boolean values by bit vectors. Thus the representation, described above, does not work for tuples with boolean components. In order to solver this problem, we represent boolean tuple components by bit vectors of length one. This means we represent the value $\mathbb{T}$ by the bit vector 1 and $\mathbb{F}$ by the bit vector 0.

Example 128. We can represent the tuple $(2, \mathbb{T}, 5)$ by the bit vector 101110. 101 represents the integer 5, 1 represent the boolean value $\mathbb{T}$ and 10 the integer 2.

The above idea now motivates the following definition of the functions α and β.

Remark In this section we use the following convention. As before T shall denote a universe of tuples, whose components have universes $T_1, \cdots, T_n$. Moreover s shall denote the sort of T and $s_1, \cdots, s_n$ shall denote the sorts of the universes $T_1, \cdots, T_n$. For $1 \leq i \leq n$ $length_i$ shall denote an integer such that $length_i = 1$ if T_i is a set of boolean values, otherwise $length_i$ shall satisfy the condition $\alpha(T_i) = BitVec(length_i)$

Definition 129. We define the function α for universes of tuples as:

$$\alpha(T) \coloneqq BitVec(\sum_{i=1}^{n} length_i) \qquad \bullet$$

For the definition of the function β we will use the following functions:

Definition 130.

- We define the function $Bool2BV$ as:

  $Bool2BV : \{\mathbb{F}, \mathbb{T}\} \to BitVec(1)$
  $Bool2BV(\mathbb{F}) \coloneqq 0$
  $Bool2BV(\mathbb{T}) \coloneqq 1$

- Then we define for $1 \leq i \leq n$ a function $comp_i^T$ as:

  $comp_i^T : T \to BitVec(length_i)$
  $$comp_i^T(t) \coloneqq \begin{cases} Bool2BV(t_i), & \text{if } T_i \text{ is a boolean universe} \\ \beta_{s_i}(t_i), & \text{otherwise} \end{cases} \qquad \bullet$$

Definition 131. We define the function β_T as

$$\beta_s(t) \coloneqq (comp_n^T(t), comp_{n-1}^T(t), \cdots, comp_1^T(t)) \qquad \bullet$$

Remark When we discussed the framework of the translation we required that $Bool2BV \in \Lambda$. Moreover we required that $Bool2BV$ has a fixed meaning in $\hat{\mathcal{C}}_{BV}$. The interpretations in $\hat{\mathcal{C}}_{BV}$ shall assign the meaning of the function $Bool2BV$ to the symbol $Bool2BV$. Similarly, we can define the inverse function $BV2Bool$ of $Bool2BV$. The symbol $BV2Bool$ in Λ shall be assigned to the function $BV2Bool$.

Remark We want to point out that the logical framework we, actually does not allow a function $Bool2BV$. Nevertheless, we use such functions in order to illustrate the special handling the boolean values require.

We will now discuss the idea behind the translation of tuple expressions. Afterwards we will present the procedure TRANSLATE-TUPLE.

In contrast to the integers there are only a few operations from the theory R that take tuples as arguments. Thus, we can discuss the translation of the individual operations in more detail than in the last section. A further difference to the translation of integers is that not all of the operations discussed here, have a direct counterpart in the BV theory. This means that we have to find suitable representations for these operations in the BV theory.

Tuple Builder To translate tuple builder expressions we apply a similar idea as we did for the β functions. This means we translate the sub-expression of the tuple builder expression and concatenate the resulting bit vector expression with the *concat* function from the BV theory. Similar as for the β functions, boolean tuple components need a special handling.

Tuple Access In example 128 we had a bit vector which was the result of translating a tuple. We have seen that this bit vector consists of sub vectors that correspond to the bit vector representations of the individual components of the tuples. This observation motivates the encoding of tuple access expressions.

To translate a given tuple access expression we first translate the sub-expression that corresponds to the tuple itself. Next we identify the part of this bit vector expression that corresponds to the component of the tuple that shall be accessed. Then we apply the *extract* operation from the BV theory in order to retrieve the identified sub vector. Finally, we have to check whether the specified component is a component of boolean values. If this is the case, we have to apply the operation $BV2Bool$ to this extraction.

We illustrate this procedure with the subsequent example.

Example 132. Let e be a tuple expression, with three components, where all components have the universe $I[0, 5]$. Assume we have an expression $a = Access_2(e)$. In this expression we retrieve the second component of e. We now assume that $\text{TRANSLATE}(e) = b$, where b is a bit vector of length nine. We can now translate the expression a to the following bit vector expression:

$$extract_{\langle 5,3 \rangle}(b)$$

Tuple Update To translate tuple update expressions, we will make use of an idea that is closely related to the idea for the translations of tuple access expressions.

Similar as before we start with the translation of the sub-expression that corresponds to the tuple itself. Then we translate the sub-expression that represent the value that shall be used for the update. If the respective component contains boolean values, we have to apply the operation *Bool2BV* to this expression. Next we determine the part of the bit vector that corresponds to the respective component of the tuple. If there are parts of the bit vector, before or after the determined part, we extract them from the tuple, by using the *extract* function. Finally, we compose the sub-vectors that occurred before respectively after the identified part with the translation of the new value by using the *concat* function from the BV theory. We illustrate this procedure with the subsequent example.

Example 133. Let e be a tuple expression, with three components where all components have the universe $I[0, 5]$. Assume we have an expression $u = Update_2(e, 4)$. In this expression we update the second component of e by the value 4. We now assume that $\textsc{Translate}(e) = b$, where b is a bit vector of length nine. We can now translate the expression u to the following bit vector expression:

$$concat(extract_{\langle 8,6\rangle}(b), concat(100, extract_{\langle 2,0\rangle}(b)))$$

Tuple Equality and Distinctness We treat equality and distinctness analogously, thus we will only discuss equality.

When we dealt with the integers, we had the problem that certain operations from the theory BV required arguments of the same length, whereas the corresponding operations from the theory R could take arguments with different universes. Now we have a similar problem. We illustrate this with an example:

Example 134. Assume we have the R expression $(1, 1) = (2, 2)$, where $(1, 1)$ is a tuple expression whose two components have the universe $I[1, 1]$ and $(2, 2)$ is a tuple expression, whose two components have the universe $I[2, 2]$. We can translate the first tuple to 11 and the second one to 1010 (Note that for simplicity we use this notation instead of using the *concat* function). Now these two bit vector expressions have a different length, thus we cannot use the equals predicate from the BV theory.

In order to deal with these problems, we enlarge the sub-vectors, which correspond to the individual components, in a suitable way. We will not describe this process here. We will just assume that there is a procedure FindCommonRepresentation that performs this resizing. For more information on this procedure we refer to the source code of the implementation of the translation.

In Algorithm 14 we make use of the ideas described above. Thus, we do not give any further descriptions.

We will neither state nor prove any properties of the described procedure.

Algorithm 14 Tuple Expression Translation

Input: An expression e over the signature Σ, that is either a tuple expression, a tuple-access or a comparison of tuple expressions.

Output: A BV expression b

function TRANSLATE-TUPLE(e)
  select case in e
    Case e is a tuple- or map access or a non-theory operation
      return TRANSLATE(e)
    Case $e = \mathit{TupleBuilder}(e_1, \cdots, e_n)$
      for $i \leftarrow 1$ to n **do**
        $t_i \leftarrow$ TRANSLATE(e_i)
        if e_i is a boolean expression **then**
          $t_i \leftarrow \mathit{Bool2BV}(t_i)$
        end if
      end for
      return $\mathit{concat}(t_n, \mathit{concat}(\cdots, \mathit{concat}(t_2, t_1)\cdots)$
    Case $e = \mathit{Access}_i(e_1)$, where e_1 has universe T
      $t \leftarrow \mathit{extract}_{\langle \sum_{j=1}^{i} \mathit{length}_j - 1, \sum_{j=1}^{i-1} \mathit{length}_j \rangle}$ TRANSLATE(e_1)
      if e is a boolean expression **then**
        return $\mathit{BV2Bool}(t)$
      end if
      return t
    Case $e = \mathit{Update}_i(e_1, v)$, where e_1 has universe T
      $t \leftarrow$ TRANSLATE(e_1)
      $\mathit{val} \leftarrow$ TRANSLATE(v)
      if v is a boolean expression **then**
        $\mathit{val} \leftarrow \mathit{Bool2BV}(\mathit{val})$
      end if
      if $i = 1$ **then**
        return $\mathit{concat}(\mathit{extract}_{\langle \sum_{j=1}^{n} \mathit{length}_j - 1, \mathit{length}_1 \rangle}(t), \mathit{val})$
      else if $i = n$ **then**
        return $\mathit{concat}(\mathit{val}, \mathit{extract}_{\langle \sum_{j=1}^{n-1} \mathit{length}_j - 1, 0 \rangle}(t))$
      else
        $l \leftarrow \sum_{j=1}^{i-1} \mathit{length}_j - 1$
        $u_1 \leftarrow \sum_{j=1}^{n} \mathit{length}_j - 1$
        $u_2 \leftarrow \sum_{j=1}^{i} \mathit{length}_j$
        return $\mathit{concat}(\mathit{extract}_{\langle u_1, u_2 \rangle}(t), \mathit{concat}(\mathit{val}, \mathit{extract}_{\langle l, 0 \rangle}(t))))$
      end if

Algorithm 14 Continuation

```
35:         Case e = e_1 = e_2
36:             t_1 ← Translate(e_1); t_2 ← Translate(e_2)
37:             (t̂_1, t̂_2) ← FindCommonRepresentation(t_1, t_2)
38:             return t̂_1 = t̂_2
39:         Case e = e_1 ≠ e_2
40:             t_1 ← Translate(e_1); t_2 ← Translate(e_2)
41:             (t̂_1, t̂_2) ← FindCommonRepresentation(t_1, t_2)
42:             return distinct(t̂_1, t̂_2)
43:     end select
44: end function
```

5.7 The Translation of Maps

This section is structured similarly, as the previous ones.

Let M denote a universe of map values, whose domain has the universe M_1 and whose image has the universe M_2. To represent map values we can now proceed in the following way: For each value in the domain of a map we compute the bit vector representation of the map applied to the value. We then concatenate these bit vector representations in the order given by the enumeration on the domain of the map. For maps whose image is a set of boolean values we have to relate the boolean values to bit vectors (similar as we did in the previous section). We illustrate this with an example:

Example 135. Let M be the map universe, whose domain is $I[0,3]$ and whose image is $I[0,6]$. Now let m be a map value such that $m(x) = 2 \cdot x$. We can represent this map value by the bit vector 110100010000. This bit vector can read in the following way: The first value from the domain, i.e. zero, is mapped to 000 the second to 010 and so on.

The above idea we will now use, in order to define the functions α and β.

Remark In this section we use the following convention. As before M shall denote a universe of maps, whose domain has universe M_1 and whose image has universe M_2. Moreover *length* shall denote the bit vector length that is necessary to represent the values from M_2. This means $\alpha(M_2) = \mathit{BitVec}(\mathit{length})$, respectively $\mathit{length} = 1$ if M_2 is a universe of boolean values. Additionally n shall denote the cardinality of M_1.

Definition 136. Let M be a universe of map values then we define the function α as:

$$\alpha(M) \coloneqq \mathit{BitVec}(n \cdot \mathit{length}) \qquad \bullet$$

Definition 137. For the definition of β_M we have to differentiate between two cases. If M_2 is a set of boolean values then we define β_M as:

$$\beta_M(m) \coloneqq (\mathit{Bool2BV}(\beta_{M_2}(m(\mathit{enum}_{M_1}(n)))), \cdots, \mathit{Bool2BV}(\beta_{M_2}(m(\mathit{enum}_{M_1}(1)))))$$

Otherwise we have:

$$\beta_M(m) \coloneqq (\beta_{M_2}(m(enum_{M_1}(n))), \cdots, \beta_{M_2}(m(enum_{M_1}(1)))) \quad \bullet$$

Subsequently, we will discuss the idea behind the translation of map expressions. We will see that we can reuse ideas from the translation of tuple expressions. Thus, we will not cover all the details of the individual translations.

Map Builder To translate map builder expressions, we first translate the defining expression, then we concatenate this expression n times. By doing this we represent the map, where each value of the domain is mapped to the given value.

Map Access Here we use a similar idea as for the translation of tuple access expressions. The major difference is that now the position of the value that shall be retrieved is given by a proper argument instead of an index. Hence, the translation is a bit more complicated. For this translation we now use an auxiliary function, which is closely related to the way we translated tuple access expressions. Let s be the sort of map values, whose domain is different from the boolean values. Then we define a function $MapAccess_s$ as:

$$MapAccess_s : \alpha(M) \times BitVec(length) \to \alpha(M_2)$$

$$MapAccess_s(m,p) \coloneqq \begin{cases} extract_{\langle length-1,0\rangle}(m) & \text{if } p = \beta_{M_1}(enum(1)) \\ \cdots & \\ extract_{\langle n\cdot length-1,(n-1)\cdot length\rangle}(m) & \text{if } p = \beta_{M_1}(enum(|M_1|)) \\ 0\cdots 0 & \text{else} \end{cases}$$

If the domain of the map is a set of boolean values we have to additionally apply the function *BV2Bool*. As the idea for this should be clear now, we will not describe it.

The operation symbols $MapAccess_s$ in Λ, shall get the above meaning.

In order to translate a map access expression we can now translate the arguments of the map access, then we pass these translated expressions to the respective auxiliary function.

Example 138. Let m be a map expression of sort s such that the domain of its universe is $I[2,4]$ and its image is $I[0,5]$. Moreover let p be an expression that has the universe $I[2,4]$ and let $a = Access(m,p)$. We now first translate m to b_1 and p to b_2. With this we can translate a to

$$MapAccess_s(b_1, b_2)$$

Map Update Similarly as for the map access we use the ideas from the translation of tuples to translate map update expressions. As before the position of the value that shall be updated is given by a proper expression, thus also the translation of map updates is a bit more complicated than the translation of tuple updates. In

order to deal with this problem we again make use of an auxiliary function. Let s be the sort of map values whose domain is different from the boolean values. Then we use a function with the following arity:

$$MapUpdate_s : \alpha(M) \times BitVec(length) \times \alpha(M_1) \to \alpha(M)$$

In the following let $m \in \alpha(M)$, $p \in Bitvec(length)$ and $v \in \alpha(M_1)$. For the definition of $MapUpdate_s$ we have to differentiate between four cases:

- In the case $p = \beta_{M_1}(enum(1))$ we have

$$MapUpdate_s(m, p, v) \coloneqq concat(extract_{\langle n \cdot length-1, length \rangle}(m), v)$$

- Let $i \in \mathbb{N}$. In the case $1 < i < |M_1|$ and $p = \beta_{M_1}(enum(i))$ we have

$$MapUpdate_s(m, p, v) \coloneqq concat(extract_{\langle n \cdot length-1, i \cdot length \rangle}(m), \\ concat(v, extract_{\langle (i-1) \cdot length-1, 0 \rangle}(m)))$$

- In the case $p = \beta_{M_1}(enum(|M_1|))$ we have

$$MapUpdate_s(m, p, v) \coloneqq concat(v, extract_{\langle (n-1) \cdot length-1, 0 \rangle}(m))$$

- Otherwise we have

$$MapUpdate_s(m, p, v) \coloneqq 0 \cdots 0$$

If the domain of the map is a set of boolean values we have to additionally make use of $Bool2BV$. As the idea for this should be clear, we will not describe it.

The operation symbols $MapUpdate_s$ in Λ shall get the above meaning. Similarly, as for the map access we can now directly use this function to translate map updates.

Example 139. Let m be a map expression of sort s such that the domain of its universe is $I[2, 4]$ and its image is $I[0, 5]$. Moreover let p be an expression that has the universe $I[2, 4]$, v be an expression with universe $I[0, 5]$ and let $u = Update(m, p, v)$. We now first translate m to b_1, p to b_2 and v to b_3. With this we can translate u to

$$MapUpdate_s(b_1, b_2, b_3)$$

Map Equals and Distinctness We can treat map equality and distinctness analogously like tuple equality and distinctness. This means that we again need a procedure FindCommonRepresentation that resizes the individual components of a bit vector in a suitable way. As in the last section, we will not describe this procedure.

In Algorithm 15 we make use of the ideas described above. Thus we do not give any further descriptions.

We will neither state nor prove any properties of the described procedure.

Algorithm 15 Map Expression Translation

Input: An expression e over the signature Σ, that is either a map expression, a map-access or a comparison of map expressions.
Output: A BV expression g

1: **function** TRANSLATE-MAP(e)
2: **select case in** e
3: **Case** e is a tuple- or map access or a non-theory operation
4: **return** TRANSLATE(e)
5: **Case** $e = \mathit{MapBuilder}(e_1)$
6: $t \leftarrow$ TRANSLATE(e_1)
7: **if** e_1 is a boolean expression **then**
8: $t \leftarrow \mathit{Bool2BV}(t)$
9: **end if**
10: ▷ In the following n denotes the cardinality of the domain of the map.
11: **return** $\mathit{repeat}_n(t)$
12: **Case** $e = \mathit{Access}_i(e_1, e_2)$ where s shall be the sort of e
13: **return** $\mathit{MapAccess}_s$(TRANSLATE(e_1), TRANSLATE(e_2))
14: **Case** $e = \mathit{Update}_i(e_1, e_2, e_3)$, where s shall be the sort of e
15: **return** $\mathit{MapUpdate}_s$(TRANSLATE(e_1), TRANSLATE(e_2), TRANSLATE(e_3))
16: **Case** $e = e_1 = e_2$
17: $t_1 \leftarrow$ TRANSLATE(e_1); $t_2 \leftarrow$ TRANSLATE(e_2)
18: $(\hat{t}_1, \hat{t}_2) \leftarrow$ FINDCOMMONREPRESENTATION(t_1, t_2)
19: **return** $\hat{t}_1 = \hat{t}_2$
20: **Case** $e = e_1 \neq e_2$
21: $t_1 \leftarrow$ TRANSLATE(e_1); $t_2 \leftarrow$ TRANSLATE(e_2)
22: $(\hat{t}_1, \hat{t}_2) \leftarrow$ FINDCOMMONREPRESENTATION(t_1, t_2)
23: **return** $\mathit{distinct}(\hat{t}_1, \hat{t}_2)$
24: **end select**
25: **end function**

5.8 The Translation of Sets

Again we will first give a description of the underlying idea of the translation.

To represent sets by bit vectors we use a commonly used method. Subsequently, we give a brief description of this method. Let U be a set with n elements. We assume to have an enumeration of the elements of U such that $U = \{u_1, u_2, \cdots, u_n\}$. We can now represent sets $A \subseteq U$ by bit vectors of length n. This bit vector representation has to fulfil the condition that the i^{th} bit is set iff $u_i \in A$.

Example 140. Let $U = \{1, 2, 3, 4, 5\}$ and let $A = \{2, 4\}$ then we can represent A by the bit vector 01010.

Remark In this section we use the following convention. S shall denote a universe of set values with sort t. Moreover U shall denote the universe of the elements of the values of S. The sort of U we denote with q. Additionally, we denote the cardinality of U by n. We also know that we can enumerate all universes from the theory R. Thus we have $U = \{u_1, \cdots, u_n\}$.

We can now use the above idea to define the functions α and β.

Definition 141. Let S be as before, then we define the function α as:

$$\alpha(S) \coloneqq BitVec(2^n) \qquad \bullet$$

Definition 142. Let S be as before and let $s \in S$. We make use of the subsequent auxiliary function:

$$isElem : S \times \{i \in \mathbb{N}^* \mid i \leq n\} \to \{0, 1\}$$

$$isElem(s, i) \coloneqq \begin{cases} 1 & \text{if } u_i \in s \\ 0 & \text{otherwise} \end{cases}$$

Now we can define β_t as:

$$\beta_t(s) \coloneqq (isElem(s, n), \cdots, isElem(s, 1)) \qquad \bullet$$

This finishes the part on the overall idea of the translation of sets. In the following we will now discuss how we can encode the different set-operations. As there are several operations on sets, we will not discuss each individual operation. For most of the operations we use the commonly used encoding for bit vectors. As these encodings are very intuitive, we will only briefly describe them. To do this we will first assume that the arguments of set operations have the same sort. Later we will discuss how we can generalise this.

We will make use of the following assumptions: A and B shall be sets from S. V shall denote some universe with m elements such that $V = \{v_1, \cdots, v_m\}$. Moreover S' shall be a universe of sets, whose values have elements in V. We denote the sort of S' by t'. C shall be a set in S'. Additionally a, b, c shall denote the bit vector representations of A, B and C. Next $e_1, \cdots, e_n$ shall denote expressions with

universe U and $\hat{e}_1, \cdots \hat{e}_m$ shall denote the bit vector representations of $e_1, \cdots, e_n$. Furthermore x, y shall denote two integers such that $x \leq y$, i_1, i_2 shall denote integer expressions with universe $I[x, y]$ and $\hat{i}_1, \hat{i}_2$ shall denote the bit vector representations of i_1 and i_2. Finally, W shall denote the universe of set values with elements in $I[x, y]$.

Basic operations

$A \cup B$ We use the standard encoding and represent $A \cap B$ by $bvor(a, b)$. We will use this encoding for subsequent encodings. In order to point out that we use the encoding of the union, we write $a \cup_{BV} b$ instead of $bvor(a, b)$.

$A \cap B$ Similar as for the union we use the standard representation of the intersection. This means we represent $A \cap B$ by $bvand(a, b)$. Similarly as in the previous case, we denote $bvand(a, b)$ by $a \cap_{BV} b$.

$A \setminus B$ We use the fact that $A \setminus B = A \cap Co_U(B)$, where $Co_U(B)$ denotes the complement of B with respect to U. We can now use the standard encoding for the complement i.e. we can represent $Co_U(B)$ by $bvnot(b)$. We can now combine this with the above result and get $a \cap_{BV} bvnot(b)$ as a representation for $A \setminus B$.

$\emptyset_t$ We can represent the empty set of universe U, by the bit vector of length n that only contains zeros. For a similar reason as before we denote this bit vector by $\emptyset^t_{BV}$ in order to indicate that we use the encoding of the empty set.

$\{e_1, \cdots, e_m\}$ We know that $\{e_1, \cdots, e_m\} = \bigcup_{i=1}^{n} \{e_i\}$. Thus it suffices to find an encoding for the singleton case. In order to encode singleton sets, we make use of an operation $getSingleton_q$:

$$getSingleton_q : \alpha(U) \to \alpha(S)$$

$$getSingleton_q(x) \coloneqq \begin{cases} 0 \cdots 01 & \text{if } x = \beta_U(enum_U(1)) \\ 0 \cdots 010 & \text{if } x = \beta_U(enum_U(2)) \\ \cdots & \\ 10 \cdots 0 & \text{if } x = \beta_U(enum_U(n)) \\ \emptyset^t_{BV} & \text{else} \end{cases}$$

The symbol $getSingleton_q$ from Λ shall get the meaning of the above function.

We can now encode $\{e_1\}$ by $getSingleton_q(\hat{e}_1)$. To encode $\{e_1, \cdots, e_m\}$ we just have to combine the idea of the encoding for singleton sets, with the idea for the encoding of the union.

$i_1 \cdot\cdot i_2$ Let $j \in I[x, y]$, then we know that j is in the set, represented by $i_1 \cdot\cdot i_2$ iff $i_1 \leq j$ and $j \leq i_2$. The idea for the encoding is to represent exactly this

property: First we introduce the auxiliary function *inInt*

$$inInt : \alpha(I[x,y]) \times \alpha(I[x,y]) \times \alpha(I[x,y]) \to \{0,1\}$$
$$inInt(a,b,c) \coloneqq \begin{cases} 1 & \text{if } x \geq 0 \wedge bvule(b,a) \wedge bvule(a,c) \\ 1 & \text{if } x < 0 \wedge bvsle(b,a) \wedge bvule(a,c) \\ 0 & \text{else} \end{cases}$$

Now let *isort* be the sort of $I[x,y]$. We can now define a function $interval_{isort}$.

$$interval_{isort} : \alpha(I[x,y]) \times \alpha(I[x,y]) \to \alpha(S)$$
$$interval_{isort}(a,b) \coloneqq (inInt(\hat{y},a,b), \cdots inInt(bvadd(\hat{x},\beta_q(1)),a,b), inInt(\hat{x},a,b))$$

The symbol *interval* from Λ shall get the meaning of the above function. We can now encode $i_1 \cdots i_2$ by $interval_q(\hat{i}_1,\hat{i}_2)$.

$A \subseteq B$ We know that $A \subseteq B$ iff $A \cap B = A$. Hence we represent $A \subseteq B$ by $a \cap_{BV} b = a$.

$A = B$ We represent $A = B$ by $a = b$.

$A \neq B$ We represent $A \neq B$ by $distinct(a,b)$.

$e_1 \in A$ We know $e_1 \in A$ iff $\{e_1\} \cap A \neq \emptyset$. We can now use the results from above. Thus, we can now represent $e_1 \in A$ by: $distinct(getSingleton_q(\hat{e}_1) \cap_{BV} a, \emptyset^t_{BV})$. We denote this encoding by $a \in_{BV} b$.

$|A|$ The chosen bit vector representation of sets, implies that the cardinality of the set A is given by the number of ones in its bit vector representation a. This means that we first introduce an auxiliary function $card_t$ such that

$$card_t : \alpha(S) \to \alpha(I[0,n])$$
$$card_t(s) \coloneqq numberOfSetBits(s)$$

We require that *numberOfSetBits* is a function that retrieves the number of ones in a bit vector and returns a bit vector that represents this number. Moreover, we require that the symbol $card_t$ in Λ shall correspond to the above definition. This allows us to represent $|A|$ by $card_t(a)$.

Cartesian Product The set $A \times C$ is the set of all tuples with components in A and C. For the encoding of the Cartesian product we will now make use of the structure of the enumeration of tuples. We know that if $v_1 \notin C$ then the first n elements of $U \times V$ are not part of $A \times C$ (since these elements are $(u_1,v_1),(u_2,v_1),\cdots,(u_n,v_1)$). This means that in this case the first n bits in the representation of $A \times C$ have to be zero. Otherwise if $v_1 \in C$ then from the first n elements of $U \times V$ exactly those values whose first component is in A are part of $A \times C$. Thus the first n bits in the

representation of $A \times C$ must be equal to the bit vector representation of A. We can generalise this idea to arbitrary elements of V. To realise this idea we now first introduce for $1 \leq i \leq |V|$ auxiliary functions $CartComp^i_{(t,t')}$.

$$CartComp^i_{(t,t')} : \alpha(U) \times \alpha(V) \to \alpha(U)$$
$$CartComp^i_{(t,t')}(a,b) \coloneqq \begin{cases} a & \text{if } \beta_V(enum_V(i) \in_{BV} b \\ 0 \cdots 0 & \text{else} \end{cases}$$

We can now introduce a function $Cart_{(t,t')}$:

$$Cart_{(t,t')} : \alpha(S) \times \alpha(S') \to \alpha(S \times S')$$
$$Cart_{(t,t')}(a,b) \coloneqq concat(CartComp^{|V|}_{(t,t')}(a,b), concat(\cdots, CartComp^1_{(t,t')}(a,b)) \cdots)$$

We require that the symbols $Cart$ in Λ get the above meaning. We can now represent $A \times C$ by:

$$Cart_{(t,t')}(a,b)$$

Power Sets Before we can start with the discussion of power sets we need some preparations:

Definition 143. Let M be a set and $g \in M$. Then the set of all subsets of M containing g, $setsWith(g)$ is given by

$$setsWith_M(g) \coloneqq \{S \subseteq M \mid g \in S\} \qquad \bullet$$

With this definition we can now prove a property that we will need for the representation of power sets.

Lemma 144. *Let M be a set and $X \subseteq M$. Then we have:*

$$\mathcal{P}(X) = Co_{\mathcal{P}(M)}(\bigcup_{x \in (M \setminus X)} setsWith_M(x))$$

Proof. We can see that the lemma holds for $X = M$ and $X = \emptyset$. Thus in the following we assume that $X \neq M$ and $X \neq \emptyset$. Let $m \in \mathcal{P}(X)$, we have to show that

$$m \in Co_{\mathcal{P}(M)}(\bigcup_{x \in (M \setminus X)} setsWith_M(x)).$$

We can see that the property holds for $m = \emptyset$. Thus we can assume that $m \neq \emptyset$. We now make a proof by contradiction, thus we can assume that

$$m \in (\bigcup_{x \in (M \setminus X)} setsWith_M(x)).$$

This means that there is a $z \in (M \setminus X)$ such that $z \in m$. But this means that $m \not\subseteq X$, thus $m \notin \mathcal{P}(X)$. This is a contradiction to our assumption. To show the other direction we assume that

$$m \in Co_{\mathcal{P}(M)}(\bigcup_{x \in (M \setminus X)} setsWith_M(x))$$

and show that $m \in \mathcal{P}(X)$. Like in the previous case we can see that the case $m = \emptyset$ is trivial. Thus, we can assume that $m \neq \emptyset$. We again make a proof by contradiction. Thus, we can assume that $m \not\subseteq X$. This means that there is a $a \in m$ such that $a \notin X$. We choose such a value a_0. Thus we now have that $m \in setsWith(a_0)$. This implies that

$$m \notin Co_{\mathcal{P}(M)}(\bigcup_{x \in (M \setminus X)} setsWith_M(x)).$$

So we got a contradiction to our assumption □

Because of the above lemma, we only have to be able to represent $setsWith_M$ in order to represent power sets. Here we can now make use of the structure of the enumeration of sets. We could prove the following lemma

Lemma 145. *Let $1 \leq i \leq n$. Then we have that*

$$setsWith_U(enum_q(i)) = \{enum_t(2 \cdot (j - j\ mod 2^{i-1}) + 2^{i-1} + j\ mod 2^{i-1} + 1) \mid 0 \leq j < 2^{n-1}\}$$

Remark As a consequence of the above lemma we have for example:

- $setsWith_U(enum_q(1)) = \{enum_t(2), enum_t(4), \cdots, enum_t(n)\}$
- $setsWith_U(enum_q(2)) = \{enum_t(3), enum_t(4), enum_t(7), enum_t(8), \cdots, enum_t(n-1), enum_t(n)\}$

From this we can now directly conclude how we can represent $setsWith_U(enum_q(1))$, $\cdots$, $setsWith_U(enum_q(n))$ by bit vectors. We illustrate this with the following example.

Example 146. Let W be a universe with three elements. We denote the sort of W by w. In order to represent sets with elements in W we need bit vectors of length nine. We can use the subsequent bit vector representations.

- Represent $setsWith_W(enum_q(1))$ by 10101010
- Represent $setsWith_W(enum_q(2))$ by 11001100
- Represent $setsWith_W(enum_q(3))$ by 11110000

Now we are ready for the actual representation of power sets. In the following we denote the universe of set values with elements in S by W and we denote the sort of W by w. Moreover, let $1 \leq i \leq n$. Additionally we denote the bit vector representation of $setsWith_q(enum_q(i))$ by σ_i and we denote the bit vector representations of $u_1, \cdots, u_n$ by $\hat{u}_1 \cdots, \hat{u}_n$. We now introduce an auxiliary function $cont_t^i$.

$$cont_t^i : \alpha(S) \to \alpha(W)$$
$$cont_t^i(s) \coloneqq \begin{cases} \sigma_i & \text{if } \hat{u}_i \in_{BV} s \\ \emptyset_{BV}^w & \text{otherwise} \end{cases}$$

We can now introduce a function $powSet_t$:

$$powSet_t : \alpha(S) \to \alpha(W)$$
$$powSet_t(a) \coloneqq bvnot((cont_t^1(a) \cup_{BV}, \cdots, \cup_{BV} cont_t^n(a)))$$

The symbols $powSet$ in Λ shall get the above meaning. We can now encode power sets $set(A)$ by $powSet_t(a)$.

Set of Subsets with Given Cardinality In the following let W and w be as before. Here we start from the representation of power sets. We then have to transform this bit vector expression such that it only represents sets with an appropriate cardinality.

We will rely on the subsequent idea: Assume we have some universe of set values and a predicate on this universe. We can now compute a bit vector, where for any position i, the i^{th} bit is one iff the predicate is true for the value represented by the respective bit. Thus, we can represent such predicates by bit vectors. Now assume we have a set A represented by a and a predicate P represented by p. We can now represent the set of elements of A that make P true by $bvand(a, p)$. Let us illustrate this with an example:

Example 147. Assume we are considering the universe of sets whose values have elements in $I[0, 4]$. Moreover assume we have a predicate *Even* (that has the expected meaning). Then we can represent this predicate by the bit vector 10101. Moreover, we can represent the set $\{3, 4\}$ by 11000. We can see that $bvand(11000, 10101)$ represents the set of even elements in $\{3, 4\}$, i.e. $\{4\}$.

We now want to use this idea to represent the predicate *has cardinality of*. For this purpose we make use of the structure of the enumeration of sets. This means that we use the property given in the following lemma.

Lemma 148. *Let* $1 \leq i \leq |W|$. *Then we have:*

$$|enum_W(i)| = numberOfSetBits(i - 1)$$

This lemma now enables us to represent the predicate *has cardinality of* n, where n is a suitable natural number. We can introduce for $0 \leq i < |W|$ bit vectors

$cardFilter_i$, that represent the above predicate, where a bit at position p is set iff $numberOfSetBits(p) = i$.

We can now introduce an auxiliary function $sizeConstraint_w$. In the following we denote the sort of $I[0,n]$ by $isort$.

$$sizeConstraint_w : \alpha(S) \times \alpha(I[0,n]) \to \alpha(W)$$

$$sizeConstraint_w(a,i) \coloneqq \begin{cases} powSet_s(a) \cap_{BV} cardFilter_0 & \text{if } i = \beta_{isort}(0) \\ \dots & \\ powSet_s(a) \cap_{BV} cardFilter_n & \text{if } i = \beta_{isort}(n) \\ \emptyset^w_{BV} & \text{else} \end{cases}$$

The symbols $sizeConstraint_w$ in Λ shall get the above meaning.

Now let i be an integer expression with domain $I[0,n]$ with a bit vector representation $\hat{i}$. We can now encode the set of all subsets with a cardinality of i ($set(A,i)$) by

$$sizeConstraint_w(a,\hat{i}).$$

Remark We can use the above idea to represent the set of all subsets with a cardinality in a certain range. Nevertheless, we will not cover such expressions here. Instead, we refer to the source code of the implementation.

Big Unions Let D be an expression with universe W. Then $e \coloneqq \bigcup D$ denotes the set with universe S that is the result of uniting the elements of the set represented by D. This means that for $u \in U$ we have that $u \in \bigcup D$ iff the set represented by D contains a set that contains u. By using considerations from before, this implies that $u \in \bigcup D$ iff $D \cap setsWith_U(u) \neq \emptyset$.

Now let, as before, denote for $1 \leq i \leq n$ the representation of $setsWith_U(enum_q(i))$ by σ_i We can now introduce a function $bigUnion_w$:

$$bigUnion_w : \alpha(W) \to \alpha(S)$$

$$bigUnion_w(s) \coloneqq \begin{array}{c}(\text{if } distinct(\sigma_n \cap_{BV} s, \emptyset^w_{BV}) \text{ then 1 else } 0, \cdots, \\ \text{if } distinct(\sigma_1 \cap_{BV} s, \emptyset^w_{BV}) \text{ then 1 else } 0)\end{array}$$

The symbols $bigUnion_w$ in Λ shall get the above meaning.

Now let D be a set of type W and d be its bit vector representation. We can now replace a bigunion ($\bigcup D$) by:

$$bigUnion_w(d)$$

Remark The big intersection can be represented analogously thus we will not cover it here.

This finishes the description of the translations of the different operations. Above we required that the arguments, for example of the union and the intersection, have the same universe. In the following we will explain how we can weaken this restriction. This is necessary as for instance also unions whose arguments have different sorts

have to be treated. Of course these sorts still have to correspond to set universes whose elements are from the same kind – e.g. sets of integers or sets of tuples whose components are of the same kind.

We explain the idea for the union, for the other cases it works completely analogously. Assume we have two universes of set values S_1 and S_2 whose values have elements in U_1 and U_2. We require that there is a universe V where $U_1 \subseteq V$ and $U_2 \subseteq V$. Moreover, let A and B be expressions with universe U_1 respectively with universe U_2 and let a and b be the bit vector representations of A and B. In this case, a and b do not necessarily have the same vector length. If a and b have different vector lengths, it is clear that we cannot use the bitwise-or to represent the union of A and B. But even if they have the same vector length this is not necessarily possible. We illustrate this with the following example.

Example 149. Assume we have the set $\{1\}$ from the set universe, whose values have elements in $I[1,3]$, and the set $\{2\}$ from the universe, whose values have elements in $I[2,4]$. We can now represent both sets by the bit vector 001. But obviously $bvor(001, 001)$ does not represent $\{1,2\}$.

In order to solve this problem, we first have to determine a universe of set values S, whose values have elements in a universe U, such that U contains both U_1 and U_2. Then we transform a and b to a' and b' where a' and b' represent the sets A and B with respect to the universe S. In order to find such common universes we use the procedure Common-Universe. The idea for this procedure is straight forward, thus we will not further discuss Algorithm 16.

The next step is to define a procedure that takes the bit vector representation of a set and its universe and another universe that covers the first one. The result of this procedure shall be the bit vector representation of the given set with respect to the second universe. The idea for this procedure is to determine those elements of the larger universe that do not occur in the smaller universe. Those elements can obviously be not part of the represented set – thus in the representation of the set with respect to the larger universe, the bits that correspond to these elements are zero.

Description of Algorithm 17 If U is equal to V, then a obviously already has the required property. Otherwise, we first determine the number of elements that lie between two consecutive (with respect to the used ordering) elements of U in V. We can consider this number as the size of the gaps between the elements of U. Thus, for $1 \leq i \leq |U| - 1$ the i^{th} entry of the array *gaps* contains the number of elements of V that are between the i^{th} and the $i + 1^{\text{th}}$ element of U. Moreover the zeroth element of *gaps* determines the number of elements of V that come before the first element of U and the last element of *gaps* determines the number of elements of V that come after the last element of U.

As we pointed out before, interpretations of the resulting bit vector representation shall contain zeroes at the positions corresponding to the gaps. Therefore, in the

Algorithm 16 Common universes

Input: Two R universes U_1, U_2 of the same kind.
Output: A R universe U, s.t. $U_1 \subseteq U$ and $U_2 \subseteq U$.

1: **function** COMMON-UNIVERSE(U_1, U_2)
2: **select case in** U_1
3: **Case** U_1 is a set of booleans
4: **return** $\{\mathbb{T}, \mathbb{F}\}$
5: **Case** U_1 is a set of integers
6: $a \leftarrow \min\{\min U_1, \min U_2\}$
7: $b \leftarrow \max\{\max U_1, \max U_2\}$
8: **return** $I[a, b]$
9: **Case** U_1 is a set of tuples
10: ▷ In this case there is some natural number n such that U_1 and U_2 have n elements.
11: ▷ We denote the components of U_1 by $U_1^1, \cdots, U_1^n$, respectively the components of U_2 by $U_2^1, \cdots, U_2^n$.
12: **for** $i \leftarrow 1$ to n **do**
13: $T_i \leftarrow$ COMMON-UNIVERSE(U_1^i, U_2^i)
14: **end for**
15: **return** $T_1 \times \cdots \times T_n$
16: **Case** U_1 is a set of maps
17: ▷ In this case there is a universe D and two universes R_1 and R_2 such that $U_1 = R_1^D$ and $U_2 = R_2^D$.
18: $R \leftarrow$ COMMON-UNIVERSE(R_1, R_2)
19: **return** R^D
20: **Case** U_1 is a set of sets
21: ▷ Let us denote the universe of the elements of values of U_1 by V_1 and the ones of U_2 by V_2.
22: $V \leftarrow$ COMMON-UNIVERSE(V_1, V_2)
23: **return** $\mathcal{P}(V)$
24: **end select**
25: **end function**

Algorithm 17 Resize Set

Input: Two R universes U and V with sorts u and v and a bit vector expression a representing a set in U
Require: $U \subseteq V$
Output: A bit vector expression representing the same set as a, but with respect to V.

```
1:  function RESIZE-SET(a, U, V)
2:      if U = V then
3:          return a
4:      end if
5:      gaps ← (0, ··· , 0); k ← 0; Counter ← 0
6:      for i ← 1 to |U| do
7:          x ← enum_u(i)
8:          for j ← k + 1 to |V| do
9:              if x = enum_v(j) then
10:                 k ← j
11:                 break
12:             end if
13:         end for
14:         gaps[i − 1] ← (k − (i + Counter))
15:         Counter ← Counter + gaps[i]
16:     end for
17:     gaps[|U|] ← |V| − (|U| + Counter)
18:     FirstGap ← −1
19:     for i ← 0 to |U| − 1 do
20:         if gaps[i] ≠ 0 then
21:             FirstGap ← i
22:             break
23:         end if
24:     end for
25:     if FirstGap = −1 then
26:         if gaps[|U| − 1] = 0 then
27:             return a
28:         else
29:             return zero_extend_{gaps[|U|−1]}(a)
30:         end if
31:     end if
32:     ▷ In the following we denote with zeroBV_n, the bit vector of length
          n containing only zeros.
33:     if FirstGap = 0 then
34:         b ← zeroBV_{gaps[0]}
```

Algorithm 17 Continuation

35: **else**
36: $b \leftarrow extract_{\langle FirstGap-1,0\rangle}(a)$
37: $b \leftarrow zero\_extend_{gaps[FirstGap]}(b)$
38: **end if**
39: $lastGap \leftarrow FirstGap$
40: **for** $i \leftarrow FirstGap + 1$ to $|U|$ **do**
41: **if** $gaps[i] \neq 0$ **then**
42: $c \leftarrow extract_{\langle i-1,lastGap\rangle}(a)$
43: $c \leftarrow zero\_extend_{gaps[i]}(c)$
44: $b \leftarrow concat(c, b)$
45: $lastGap \leftarrow i$
46: **end if**
47: **end for**
48: **if** $lastGap \neq |U|$ **then**
49: $c \leftarrow extract_{\langle |U|-1,lastGap\rangle}(a)$
50: $b \leftarrow concat(c, b)$
51: **end if**
52: **return** b
53: **end function**

remaining part of the algorithm we insert zeroes, to the given expression, with respect to the positions and lengths of the respective gaps.

We can now combine all the presented ideas and get the Algorithm 18.

Remark In the implementation, the translation of set expressions is done slightly different compared to Algorithm 18. In the implementation we determine a universe that covers the universes of the expression of interest and the universes of all its subexpressions. We then use this universe for the translation of the expressions and for the translations of the subexpressions. This means that in the implementation, we use larger universes than in Algorithm 18. But the advantage is that fewer resizes have to be done.

Algorithm 18 Set Expression Translation

Input: An expression e over the signature Σ
Output: A BV expression g

function TRANSLATE-SET(e)
  select case in e
    Case e is a tuple- or map access or a non-theory operation
      return $t \leftarrow$TRANSLATE(e)
    ▷ In the following U_1 and U_2 denote the universes of the subexpressions
    Case $e = e_1 \cup e_2$
      $U \leftarrow$ COMMON-UNIVERSE(U_1, U_2)
      $b_1 \leftarrow$ RESIZE-SET(TRANSLATE(e_1), U_1, U)
      $b_2 \leftarrow$ RESIZE-SET(TRANSLATE(e_2), U_2, U)
      return $bvor(b_1, b_2)$
    Case $e = e_1 \cap e_2$
      $U \leftarrow$ COMMON-UNIVERSE(U_1, U_2)
      $b_1 \leftarrow$ RESIZE-SET(TRANSLATE(e_1), U_1, U)
      $b_2 \leftarrow$ RESIZE-SET(TRANSLATE(e_2), U_2, U)
      return $bvand(b_1, b_2)$
    Case $e = e_1 \setminus e_2$
      $U \leftarrow$ COMMON-UNIVERSE(U_1, U_2)
      $b_1 \leftarrow$ RESIZE-SET(TRANSLATE(e_1), U_1, U)
      $b_2 \leftarrow$ RESIZE-SET(TRANSLATE(e_2), U_2, U)
      return $bvand(b_1, bvnot(b_2))$
    Case $e = \emptyset$
      return $0 \cdots 0$
    Case $e = \{e_1, \cdots, e_n\}$, where $e_1, \cdots, e_n$ have sort q
      $b_1 \leftarrow getSingleton_q$(TRANSLATE($e_1$))
      $\cdots$
      $b_n \leftarrow getSingleton_q$(TRANSLATE($e_n$))
      ▷ Since $bvor$ is associative, we write $bvor(a, b, c)$ instead of $bvor(a, bvor(b, c))$.
      return $bvor(b_1, \cdots, b_n)$
    Case $e = i_1 \cdot\cdot i_2$, where i_1, i_2 are expressions with sort i
      $int_1 \leftarrow$ TRANSLATE(i_1)
      $int_2 \leftarrow$ TRANSLATE(i_2)
      return $interval_i(int_1, int_2)$

Algorithm 18 Continuation

33: **Case** $e = e_1 = e_2$
34: $U \leftarrow$ COMMON-UNIVERSE(U_1, U_2)
35: $b_1 \leftarrow$ RESIZE-SET(TRANSLATE$(e_1), U_1, U)$
36: $b_2 \leftarrow$ RESIZE-SET(TRANSLATE$(e_2), U_2, U)$
37: **return** $b_1 = b_2$
38: **Case** $e = e_1 \neq e_2$
39: $U \leftarrow$ COMMON-UNIVERSE(U_1, U_2)
40: $b_1 \leftarrow$ RESIZE-SET(TRANSLATE$(e_1), U_1, U)$
41: $b_2 \leftarrow$ RESIZE-SET(TRANSLATE$(e_2), U_2, U)$
42: **return** $distinct(b_1, b_2)$
43: **Case** $e = e_1 \in e_2$, where e_1 has sort q
44: $b_1 \leftarrow$ TRANSLATE(e_1)
45: $b_2 \leftarrow$ TRANSLATE(e_2)
46: **return** $distinct(bvand(getSingleton_q(b_1), b_2), 0 \cdots 0)$
47: **Case** $e = |e_1|$, where e_1 has sort t
48: $b_1 \leftarrow$ TRANSLATE(e_1)
49: **return** $card_t(b_1)$
50: **Case** $e = e_1 \times e_2$, where e_1 has sort t_1 and e_2 has sort t_2
51: ▷ In the following n denotes the cardinality of the universe of e_2
52: $b_1 \leftarrow$ TRANSLATE(e_1)
53: $b_2 \leftarrow$ TRANSLATE(e_2)
54: **return** $Cart_{(t_1,t_2)}(b_1, b_2)$
55: **Case** $e = set(e_1)$, where e_1 has sort t
56: $b \leftarrow$ TRANSLATE(e_1)
57: **return** $powSet_t(b)$
58: **Case** $e = set(e_1, e_2)$, where e_1 has sort t
59: $b_1 \leftarrow$ TRANSLATE(e_1)
60: $b_2 \leftarrow$ TRANSLATE(e_2)
61: **return** $sizeConstraint_t(b_1, b_2)$
62: **Case** $e = set(e_1, e_2, e_3)$
63: $\cdots$
64: **Case** $e = \bigcup e_1$, where e_1 has sort t
65: $b_1 \leftarrow$ TRANSLATE(e_1)
66: **return** $bigunion_t(b_1)$
67: **Case** $e = \bigcap e_2$
68: $\cdots$
69: **end select**
70: **end function**

6 The Complete Translation

In the previous chapters we introduced the algorithms UNQUANTIFY and TRANSLATE, where the first one eliminates the quantifiers from a given formula and the second one transforms a given formula over a certain signature containing Σ_R to a formula over a certain signature containing Σ_{BV}. In this chapter we will describe the algorithm TRANSFORM that combines the previous two algorithms.

Subsequently, let Σ and Ξ be signatures as in the last chapter. Moreover let $\hat{\mathcal{C}}_R$ and $\hat{\mathcal{C}}_{BV}$ be sets of interpretations as in the last chapter.

Algorithm 19 Transform formula

Input: A formula F that is defined over the signature Σ
Require: $\mathit{free}(F) = \emptyset$
Output: A formula F_{out} over the signature Ξ
Ensure: $\nexists p \in Pos(F_{out}), x^i \in \mathcal{V}, F' \in \mathcal{L} : F_{out}\langle p \rangle = \forall_{\mathcal{L}} x^i F'$
Ensure: $\nexists p \in Pos(F_{out}), x^i \in \mathcal{V}, F' \in \mathcal{L} : F_{out}\langle p \rangle = \exists_{\mathcal{L}} x^i F'$
Ensure: $\mathit{free}(F_{out}) = \emptyset$
Ensure: F is R-satisfiable iff F_{out} satisfiable with respect to $\hat{\mathcal{C}}_{BV}$.

```
1: function TRANSFORM(F)
2:     F_1 ← UNQUANTIFY(F)
3:     F_2 ← TRANSLATE(F_1)
4:     return F_2
5: end function
```

Description of Algorithm 19 The algorithm takes a closed formula defined over the signature Σ and returns a closed formula without quantifiers over the signature Ξ. Moreover the algorithm preserves the satisfiability of the initial formula in the sense that the initial formula is R-satisfiable iff the resulting formula is satisfiable with respect to $\hat{\mathcal{C}}_{BV}$.

We remember that in the theory R we required that there is for each value of each universe a constant symbol. Moreover, the precondition of TRANSFORM asserts that the given formula is closed. Thus, we can apply the algorithm UNQUANTIFY. The result of this formula is a closed formula without quantifiers that is R satisfiable iff the initial formula is R-satisfiable.

From the postcondition of UNQUANTIFY we can now conclude that the preconditions of TRANSLATE hold. Thus, we can apply TRANSLATE to F_1. The result of TRANSLATE is again a closed formula without quantifiers. Moreover, the postcondition of TRANSLATE implies that F_2 is satisfiable with respect to $\hat{\mathcal{C}}_{BV}$ iff F_1 is R

satisfiable. From this we get that F is R satisfiable iff F_2 is satisfiable with respect to $\hat{\mathcal{C}}_{BV}$. This means TRANSFORM is the algorithm we looked for.

7 The Implementation of the Translation

In this chapter we will discuss selected aspects of the implementation of the translation that are not covered by the theoretical model of the translation.

7.1 Differences Between the Implementation and the Theoretical Description

If we compare the theoretical framework, which was introduced in the previous chapters, with the RISCAL system, we can see that the chosen framework does not cover the whole RISCAL language. In this chapter we will discuss some of the major differences between this framework and RISCAL and how the translation can be generalised to the actual RISCAL language. Note that here we will only give rather informal descriptions and that we will only discuss the most important adaptations of the translation that are necessary for the generalisation. For additional information on this topic we refer to the source code of the implementation.

7.1.1 Quantifiers

RISCAL provides various kinds of quantifiers. Only the universal and the existential quantifiers over a type can be directly be associated to quantifiers from the theoretical framework. In this section we will describe how to deal with the remaining quantifiers.

Universal and Existential Quantifier In RISCAL there are universal quantifiers that do not directly correspond to quantifiers from the theoretical framework. These are quantifiers over sets, quantifiers with conditions and quantifiers with several variables. We process such quantifiers in the following way:

1. If a quantifier contains multiple variables, rewrite it to a chain of quantifiers with one variable. If there are conditions on the individual variables, add the conditions to the corresponding variables of the single variable quantifiers. If there is a condition for the entire quantifier, add the condition to the inner most single variable quantifier.

2. If a variable is quantified over a set instead of a type, determine the type of the elements of the set. Transform the quantifier such that the variable is quantified over this type. Add a condition to the quantifier that assures that the variable is in the set.

3. If a universal quantifier contains a condition, incorporate it into the quantified expression by using an implication. If an existential quantifier contains a condition, incorporate it into the quantified expression by using a conjunction.

Remark For quantifiers with multiple variables we also reorder the variables in certain cases. We refer to the source code for more information on this topic.

The above procedure is illustrated with the following example:

Example 150. Assume we have:

$\forall$`x:`$\mathbb{N}$`[4] with x`$\neq$`2,y`$\in${`1,3}.x+y<8`

If we apply the above procedure, we get:

$\forall$`x:`$\mathbb{N}$`[4]. x`$\neq$`2` $\Rightarrow$ $\forall$`y:`$\mathbb{Z}$`[1,3]. y`$\in${`1,3}` $\Rightarrow$ `x+y<8`

We can see that the quantifiers in the resulting formula, correspond to the way we introduced quantifiers in the theoretical framework.

After this processing, we can essentially apply the unquantification algorithm discussed in the theoretical part. We will later see that there are exceptions to this rule. Later we will discuss how we can adapt the unquantification such that it still works. Moreover, we have to point out that in RISCAL there are types whose values do not have constant symbols. This means that we cannot use the quantifier expansion with respect to the RISCAL language itself. As in SMT-LIB with the QF_UFBV logic we have constants for every type we can perform the elimination of quantifiers during the translation of the theories. This means that in the actual implementation, we do not have the separation of unquantification and theory translation, we have in the theoretical description.

Moreover, we remember that we used substitutions in order to replace the quantified variables by the skolem functions, respectively by the constants of the quantified type. As SMT-LIB provides a *let* construct, we use such *lets* instead of substitutions.

Sum and Product To process sums and products we make use of a similar idea as for the universal and the existential quantifier.

1. If a sum or a product contains multiple variables, transform it to the single variable case similar as for the universal and existential quantifiers.

2. If a variable is quantified over a set, proceed as with the logical quantifiers.

3. If a sum contains a condition, introduce an If-Expression, whose condition is the condition of the quantifier. Put the summand into the Then-Part of the If-Expression and 0 into the Else-Part. Then remove the condition from the sum, and change the summand to the If-Expression. If a product contains a condition, proceed as with sums, with the only difference that in the Else-part of the If-Expression there shall be 1 instead of 0.

4. Expand the sum respectively the product similar as it is done with universal quantifiers. This means, determine all values of the quantified type and add the summands, respectively multiply the factors with respect to these values.

For a similar reason, as for the universal and the existential quantifiers, this expansion is performed during the theory translation.

We illustrate the above procedure with the following example:

Example 151. Assume we have:

```
∑ x:ℕ[1] with x≠1,y∈{1,3}. x+y
```

By applying the first three steps we get:

```
∑ x:ℕ[1]. if x≠1 then ( ∑ x:ℤ[1,3]. if y∈{1,3} then x+y else
    0) else 0
```

For a better understanding we give an adaption of the last step of the procedure. This means we give the expansion within RISCAL and not within the SMT-LIB set up. Thus, we can use integers instead of bit vectors. Moreover, we directly plug the expanded values in, instead of using let-expressions. With this in mind we get:

```
if 0≠1 then ((if 1∈{1,3} then 0+1 else 0)+(if 2∈{1,3} then
   0+2 else 0)+(if 3∈{1,3} then 0+3 else 0)+
if 1≠1 then ((if 1∈{1,3} then 1+1 else 0)+(if 2∈{1,3} then
   1+2 else 0)+(if 3∈{1,3} then 1+3 else 0)
```

Number of Values of a Quantification The *number of values of a quantification* is a special case of the sum. This kind of expression is equivalent to a sum with the same variables over the same types, respectively the same sets, with the same conditions, where the summand is 1.

Example 152. Thus we can treat

```
#x:ℕ[1] with  x≠1,y∈{1,3}
```

analogously as

```
∑ x:ℕ[1] with x≠1,y∈{1,3}. 1
```

Choose RISCAL provides three different kinds of choose expressions, here we will only discuss the choose expression that takes a quantified variable and an expression (`choose` *QV* `in` *exp*) (see [36]). The other ones can be treated in a similar way.

We have to point out that we consider a choose expression as an abbreviation for an application of a fresh function (with an appropriate arity), where we only know that the resulting values of this function have the properties given by the choose. As a consequence of this, we do not necessarily have:

`choose` *QV* `in` *exp* `=` `choose` *QV* `in` *exp*

(where QV denotes a quantified variable and *exp* an expression). As in this case both, the left and the right-hand side, abbreviate the application of a different function. This also means that we make use of a different understanding of the choice-function as for example in [29]. There, two choices are equal, if the involved conditions are equivalent.

For the representation of the choose expression we first assume that the choice is actually possible. For many real world RISCAL choose expressions the assumption is satisfied, as we assume that the RISCAL proof obligation that asserts the choice is possible has to hold. But there are also choices where this proof obligation holds, but still the choice is not always possible. The reason for this is that the described proof obligation, only reasons about values that can actually be attained in the given context. For example, if a RISCAL function contains a choose expression, it might be the case that the choice is not possible for value assignments for the arguments of the function that do not satisfy the precondition of the function.

We can now process `choose` QV `in` *exp* in the following way:

1. Determine the free variables in the expression. There can be free variables in *exp*, but there can also be free variables in QV. For example a condition in QV can contain a free variable.

2. Determine all the existentially quantified variables and add them to the variables from the previous step (this means that existentially quantified variables are considered as free variables even if they occur neither in QV nor in *exp*).

3. Introduce a function f that takes for each free variable an argument of the type of the free variable. This function shall return a value that has the type of *exp*.

4. Next introduce an axiom. This axiom shall state that for each argument value of the new function we have that

   $$\forall \mathit{freeVars}\,.\,\exists QV\,.\;\; f(\mathit{freeVars}) = \mathit{exp}$$

   Here it is necessary that the choice is actually possible. If there is a value assignment for the free variables, where the choice is impossible, then the above axiom is invalid.

5. Replace the choose expression by an application of the new function applied to the free variables.

We now illustrate the presented procedure for the representation of choose expressions, where the choice is actually possible. We illustrate this procedure with the following example:

Example 153. Let us consider the RISCAL expression

```
∀x:ℕ[3]. (choose y:ℕ[3] with y ≥ x in y) ≥ x
```

We can see that in the choose expression x is a free variable. Thus, we introduce a new function *chooseFun* by:

`fun chooseFun(x:ℕ[3]):ℕ[3]`

To ensure the condition we also introduce an axiom:

`axiom chooseAxiom ⇔ ∀x:ℕ[3]. ∃y:ℕ[3]. y ≥ x ∧ chooseFun(x)=y`

Finally we replace the choose by the function application and get:

`∀x:ℕ[3]. chooseFun(x) ≥ x`

In the above procedure we added all the existentially quantified variables to the free variables. In the following we give an example that illustrates the reason for this.

Example 154. In the following P shall denote a binary predicate that takes arguments from `ℕ[1]`. We require that $P(0,0)$ and $P(1,1)$ hold. For the remaining combinations of arguments the predicate shall be false. Let us now consider the RISCAL formula

`∃x:ℕ[1]. P(x,choose y:ℕ[1] in y)`

This formula is not valid as for every possible value for x we can find a suitable value for y such that P does not hold. If we would not rewrite the choose expression without considering the existentially quantified variable x, we would represent the choice by a nullary function that can be either 0 or 1. If we would now replace the choose expression by this function, we would get a formula that is obviously valid.

Last but not least, we also briefly discuss how we can treat choose expressions, where the choice is not always possible. Here we will not go into detail. For more information on this topic we refer to the implementation of the translation.

We start these considerations with an example for such a choose expression.

Example 155. Assume we have a RISCAL specification containing the following formula:

`∀x:ℕ[5] with x>0. (choose y:ℕ[5] with y<x in y)>0`

We can see that the proof obligation that asserts that the choice is possible holds, as we have $x > 0$. Nevertheless, this is not sufficient for the translation of this choose expressions. Because, if we would apply the procedure from before, we would get:

`∀x:ℕ[5]. ∃y:ℕ[5]. y<x ∧ chooseFun(x)=y`

Where *chooseFun* denotes the function that we use for the representation of the choose expression. But this formula is not valid.

In order to solve this problem the translation provides the option to treat such choose expressions differently. This can be done by the option *-smt-cguards 1*, respectively *-smt-cguards 2*. If the option *-smt-cguards 1* is set, all conditions

containing only one variable are stored. If there is such a restriction for one of the free variables in a choose expression, we incorporate this condition into the axiom for the choose. For the above example this would result in:

$\forall$`x:`$\mathbb{N}$`[5]. x>0` $\Rightarrow$ $\exists$`y:`$\mathbb{N}$`[5]. y<x` $\wedge$ `chooseFun(x)=y`

We can see that the formula now holds. Unfortunately, this option is not always sufficient, as we only store conditions with just one variable. Let us now consider the following example:

Example 156. Assume we have:

$\forall$`x:`$\mathbb{N}$`[5],y:`$\mathbb{N}$`[5] with x+y<5. (choose z:`$\mathbb{N}$`[5] with z>x in z)<5`

Again, we can see that the proof obligation that asserts that the choice is possible holds. We can also see that the restriction of importance contains two variables – thus it is not stored. Even, if we would remember the condition this would not be of help for us. Since the incorporation of this condition into the axiom for the choose, would require that the choose function also depends on y, what we want to avoid.

For cases, such as the above example, the option *-smt-cguards 2* has to be used. We first check whether the choice is possible, with this option set. Only in this case, restrictions on the function are required. For the given example, this would result in:

$\forall$`x:`$\mathbb{N}$`[5],y:`$\mathbb{N}$`[5].(`$\exists$`z:`$\mathbb{N}$`[5] z>x)` $\Rightarrow$ `(`$\exists$`z:`$\mathbb{N}$`[5]. z>x` $\wedge$ `chooseFun(x)=y)`

Where *chooseFun* again denotes the function that we use for the representation of the choose expression. Unfortunately, this means that we get an additional existential quantifier that we cannot eliminate by means of skolemisation. In section 8.2 we will discuss this problem with actual RISCAL specifications.

Minimum and Maximum We can treat minimum and maximum expressions similarly as the above choose expressions. The difference is that we have to use an additional axiom here. For Minimum expressions we introduce a axiom that shall state that for each argument value of the new function, we have:

$\forall freeVars\,.\ \forall QV\,.\ f(freeVars) \leq exp$

For Maximum expressions we introduce an axiom that shall state that for each argument value of the new function, we have:

$\forall freeVars\,.\ \forall QV\,.\ f(freeVars) \geq exp$

Above the function f denotes the function that is used to represent the respective expression.

Set Builder The idea for processing Set Builders is related to the idea for processing sums and products. We describe the idea for a single variable, for multiple variables the idea works similar.

1. If a variable is quantified over a set, rewrite it to a variable that is quantified over a type, by using conditions.

2. Determine all values of the quantified type.

3. Construct singleton sets, containing the expression of the set builder, with respect to the values determined in the previous step.

4. If there are no conditions on the quantified variable, rewrite the set builder by a union of the singleton sets from the previous step. If there is a condition, we make use of If-Then-Else expressions. This means we get:

   `(If` *Condition*(v_1) `then {`v_1`} else ∅)` $\cup \cdots \cup$ `(If` *Condition*(v_N) `then {`v_N`} else ∅)`

   The above *Condition* denotes the condition of the quantified variable and $v_1, \cdots, v_N$ denote the values of the type of the variable.

As before, we have to perform the generation of values during the theory translation.

In the following example we illustrate the above procedure. Similar as for sum and product expression, here we also give the expansions within RISCAL and not SMT-LIB.

Example 157. We can transform

```
{x+1 | x:ℤ[2,4] with x>2}
```

by the above procedure to

```
(If 2>2 then {2+1} else ∅) ∪ (If 3>2 then {3+1} else ∅) ∪
   (If 4>2 then {4+1} else ∅)
```

7.1.2 Functions

The RISCAL language is much less restrictive with functions than we were in our theoretical framework. Subsequently, we give some of the important differences and how we can deal with them.

Functions with same Name In RISCAL, functions can have the same name, as long as the types of the arguments are different. This is not only forbidden in the theoretical framework we introduced here, but also in SMT-LIB. Thus, we have to rename functions with the same names.

Definitions of Functions In RISCAL it is possible to define boolean functions, where in the defining expressions quantifiers may occur. We must not apply skolemisation to such quantifiers. This means that in this case, we have to remove both, existential and universal quantifiers by quantifier expansions.

Example 158. Assume we have a boolean constant b, which is defined as follows:

```
pred b ⇔ exists x:ℕ[4]. x>3;
```

and a theorem that states that b is true. This theorem obviously holds. If we would now apply skolemisation in b, we would get:

```
pred b ⇔ f>3
```

where f denotes a new constant, which has to be declared. In this case the negation of the theorem is satisfiable (e.g. set f to 2).

Functions with Boolean Arguments In the theoretical framework we prohibited functions with boolean arguments. But in RISCAL, there can be functions with boolean arguments. This means that in RISCAL we can have function applications, where arguments are quantified expressions. Similar as in the previous case, we must not apply skolemisation to such quantifiers. Therefore, we have to expand all quantifiers. We illustrate the problem that would arise, if we would apply skolemisation, with the following example.

Example 159. Assume we have a function P that takes a boolean value and returns a boolean value. Moreover, we assume that $P(\mathbb{T}) = \mathbb{F}$ and $P(\mathbb{F}) = \mathbb{T}$. Now assume we have a theorem that states that $\neg P(\exists x : Int[0, 2].x > 1)$. We can see that this theorem is valid. If we would now apply skolemisation to the theorem, we would get $\neg P(f > 1)$ for a suitable function f. We can see that the theorem is no longer valid, because if f is set to 0, then the assertion of the theorem is false.

Function Arguments of Smaller Types In RISCAL functions can be called with arguments, whose type is contained in the respective type, from the arity of the function. This is neither allowed in the used theoretical framework, nor in SMT-LIB. This means that when we translate a RISCAL function application to an SMT-LIB function application, we have to ensure that we translate the function arguments to SMT-LIB expression of an appropriate sort.

We could generalise the used logic to an ordered-sorted logic similar as presented in [32]. This would allow function applications even in certain cases, where the sorts of the arguments are different from the sorts, given by the arity of a function symbol. We omitted such a generalisation, as the idea of the translation can also be illustrated by the simpler logic we use here.

7.1.3 Further Language Constructs

Besides of the differences stated above there are several other language constructs from RISCAL that have no direct counterpart in the theoretical framework. In the following we will give the most important ones.

Let and If Both RISCAL and SMT-LIB provide If-expressions. Thus, we can translate the if-expressions from RISCAL to If-expressions from SMT-LIB. Moreover, both RISCAL and SMT-LIB also provide let-constructs. Here, the only thing to consider is that the *let* from SMT-LIB binds variables in parallel, while RISCAL also provides a *let* that binds variables sequentially. Therefore, we rewrite the sequential *lets* to nested parallel ones.

Records, Arrays, Natural Numbers and the Unit Type We can treat records similar as we treat tuples, arrays similar as maps and natural numbers as we treat integers. We represent the unit type by a bit vector of length one. The single value of the unit type we represent by the bit vector 0.

Recursive Data Types Although RISCAL provides recursive data types, the implementation does not support these types. This is the major restriction of the implementation of the translation.

7.2 Selected Techniques for Improving the Translation

In this section we will discuss selected techniques that are applied in the implementation of the translation. The aim of these techniques is to reduce the size of the result of the translation. On the one hand, this shall simplify the work for the SMT-Solvers — such that a better performance can be achieved (we will discuss in the next section, if this is actually the case). On the other hand, these techniques shall reduce the amount of memory needed for the translation (if we do not make use of subsection 7.2.1 and subsection 7.2.2 the available memory could become a limiting factor).

7.2.1 Cut Unnecessary Parts of RISCAL Specifications

If a theorem in a RISCAL specification shall be checked, it is often the case that the theorem does not depend on the whole specification. This motivates the idea of skipping unnecessary parts of specifications.

Thus, the implementation provides the possibility to ignore all the unneeded parts of a specification. This means that all functions and values, the theorem of interest, neither directly nor indirectly depends on, shall be ignored. Moreover, axioms and theorems, which are different from the one that shall be checked, are also ignored. As it is possible that a specification contains functions that allow more than one interpretation (e.g. functions that shall be introduced via their specification), the implementation also provides the option to skip unused parts of the specification, but to retain axioms and theorems that contain these functions. The reason for this is that such axioms/theorems could forbid certain interpretations, thus skipping them could give incorrect results.

7.2.2 Use Auxiliary Functions for the Expansion of Quantifiers

As we discussed before, we eliminate universal quantifiers by expanding the quantifiers. Similarly, we use quantifier expansion for existential quantifiers as an alternative to skolemisation. The result of this procedure can be quite lengthy formulae.

Example 160. Expanding the universal quantifiers in

```
∀x:ℕ[9]. ∀y:ℕ[9]. P(x,y)
```

results in a conjunction with 100 arguments.

The above example shows that especially nested quantifiers are very likely to result in lengthy formulae.

Thus, we want to use auxiliary functions that shall represent the quantified expressions. We illustrate this procedure with the above example.

Example 161. We define a function:

```
fun f(x:ℕ[9]):Bool = ∀y:ℕ[9]. P(x,y)
```

We can then represent the initial formula by :

```
∀x:ℕ[9]. f(x)
```

We can now expand both quantifiers to conjunctions with ten arguments, respectively.

For quantifiers, with small types and/or small quantified expressions, it seems not so likely that this procedure pays off. Thus, the implementation also provides the option of using auxiliary functions only for cases, where we think that it pays off. This means, the technique is only used, if the size of the quantified type, times the size of the quantified expression, surpasses some threshold.

7.2.3 Limit Use of Skolemisation

If we apply skolemisation, we get a function that we have to represent by a bit vector valued function. The problem is that for many RISCAL types, the set of bit vectors that is used to represent the type is larger than the type itself. This means, that such bit vector sets contain values that do not represent a value of the associated type. Thus, we have to ensure that the bit vector representation of the skolem function actually attains a value that represents a value of the respective type. We illustrate this with the following example:

Example 162. Assume we have:

```
∀x:ℕ[10]. ∃y:ℤ[1,2]. P(x,y)
```

We can now skolemise this formula by a function

$$f : \mathbb{Z}[0, 10] \to \mathbb{Z}[1, 2]$$

and get the formula

`∀x:ℕ[10]. P(x,f(x))`

We now have to represent the function f by a function

$$\hat{f} : BitVec(4) \to BitVec(2)$$

Now the problem is that for example the bit vector 11 does not represent a number in $\mathbb{Z}[1, 2]$. Thus, we have to assert that:

$$01 \leq_{BV} \hat{f}(0000) \leq_{BV} 10 \wedge 01 \leq_{BV} \hat{f}(0001) \leq_{BV} 1010 \wedge \cdots \wedge 01 \leq_{BV} \hat{f}(1010) \leq_{BV} 1010$$

In the above formula $\leq_{BV}$ denotes the unsigned less equal predicate on bit vectors.

This motivates the idea of using quantifier expansion, instead of skolemisation, for cases where asserting the required properties of skolem functions, seems to be costly.

To sum up, the implementation is capable of either always using skolemisation, when it is applicable, never using skolemisation, or to use skolemisation whenever it seems to pay off (in the above sense).

8 Experimental Evaluation of the Implementation

In this chapter we present and analyse the results of the application of the implementation of the translation to various RISCAL theorems.

8.1 The Tests

In this section we will describe the setup of the tests, and we will present the results of the tests.

For testing the program a computer with an Intel Xeon Gold 6128 processor, 1 546 665 MB of RAM and running the 64-Bit version of Debian 10 was used[1]. For compiling and running the software, OpenJDK 11 was used.

For the tests we used a slightly adapted version of RISCAL[2]. The differences to the original version are, on the one hand that the output is given in a different way that simplifies the analysis of the results. On the other hand, all checks of theorems were performed twice, where we used a short timeout for the first run. The reason for this was to reduce the impact of the Java Just-in-time compilation.

Furthermore, the following versions of the SMT solvers were used.

- Boolector version 3.2.0 (built with support for the SAT solver CaDiCaL)
- CVC4 version 1.7
- Yices version 2.6.1
- Z3 version 4.8.7

All solvers were run with their default settings (not considering options regarding the input, like setting the input language to SMT-LIB).

For the timing of the program, we only considered the time that was actually needed for checking the respective theorem. This i.a. means that the time, needed for the parsing and type checking of a specification was not considered. This applies both to checks done by RISCAL and to checks done by means of the translation and an SMT-Solver. Furthermore, we used a timeout of 20 minutes – this means that we terminated checks done by RISCAL after 20 minutes, and we terminated the respective SMT-solver after 20 minutes. We did not consider the time, needed

[1] The computing environment was provided by RISC (Research Institute for Symbolic Computation).
[2] `https://www.risc.jku.at/research/formal/software/RISCAL/papers/thesis-Reichl.tgz`

for the translation of a specification, for the timeout, as the translation usually only took a fraction of the time needed by the SMT-solvers. In fact, it would not have made a difference, if the time needed for the translation, would have been considered for the timeout. To measure the time we used wall clock time, more specifically all timings were conducted by the help of the Java method System.currentTimeMillis(). All times given in the following, are given in milliseconds. Thus, subsequently we will not mention the units of the times.

The test cases were selected with the idea in mind of analysing the performance of the translation on the basis of real world RISCAL specifications. Thus, the majority of the test cases were taken from existing RISCAL specifications. The RISCAL specifications *graphs.txt*, *relation.txt* and *sets.txt* are taken from [9], where the specification *sets.txt* was modified by additional theorems. The RISCAL specifications *bubble.txt*, *catlogic.txt*, *gcd2.txt*, *gsort.txt*, *poly.txt*, *sat3.txt*, *search.txt* and *sort2.txt* are part of the RISCAL software, as sample specification and can be found on [34]. The specification *gcd2.txt* was renamed, its original name was *gcd.txt*. In addition to the renaming, this specification was also slightly modified by additional theorems. The RISCAL specifications *divide.txt* and *gcd1.txt* are taken from [38]. The specification *gcd1.txt* was renamed, its original name was *gcd.txt*.

Further goals of the selection of the test cases were:

- Covering a relatively large fragment of the RISCAL language. In particular, this means that for all the supported RISCAL types (except the unit type), there are dedicated test cases.

- Covering both, valid and invalid theorems. Nevertheless, the majority of the test cases contains valid theorems. This is partially due to the fact that the available RISCAL specifications provide more valid than invalid theorems that are suitable for the application of the translation.

- In order to highlight the differences between the different checking methods, we mainly used model parameters, ensuring that at least one of the methods needs at least one second.

Table 8.1 gives a description of the used test cases. The first column gives an index for the test cases. The second one determines the RISCAL specification, where the theorem that shall be checked can be found. The third one gives the theorem that was checked. Note that this column also contains the names of the theorems that were generated by the RISCAL system. The names of these theorems may be different in future versions of RISCAL. The fourth and the fifth column give the model parameters for the respective test case. All the theorems listed in Table 8.1 were analysed by RISCAL and by means of the translation, in combination with the SMT solvers Boolector, CVC4, Yices and Z3. For each run of an SMT-solver a separate translation was done.

For the translation we used the options *-smt-mod 0* (all function definitions shall be preserved) *-smt-cut 1* (cut the unused parts of specifications off) *-smt-skol 1*

(skolemise whenever it seems to pay off) *-smt-af 1* (use auxiliary functions for quantifier bodies whenever it seems to pay off). For all other settings, the translation provides, the default options were used. The exceptions to this are, on the one hand the tests 34 and 40, where the option *-smt-cguards 2* was used and the test 35, where the option *-smt-cguards 1* was used. On the other hand, tests 71 – 85 make use of the option described in Table 8.3 and Table 8.5.

The names for the SMT-LIB scripts that were generated for the above described tests are given by appending the name of the theorem to the name of the file without the file extension.

Table 8.2 contains the results of the tests. The first column again gives the index of the test case (these values correspond to the values given in the first column of Table 8.1). The second column indicates whether the analysed theorem is valid or invalid. The third one gives the time needed by RISCAL to check the theorem. If the check was aborted because of a timeout, the column contains "Timeout". It contains "Memory", if the program threw an Out Of Memory Exception. The fourth column contains the average time needed for the translation. Note that for this purpose, only times from terminated checks were considered. Thus, no translation time is given for test 29. The remaining columns give the total time needed by the translation and the analysis with the respective SMT-solver. If an SMT-solver did not terminate within 20 minutes, the respective entry is "Timeout".

In Table 8.3 and Table 8.4 we discuss additional tests, where we analyse the impact of certain options of the translation on the performance. The first five columns of Table 8.3 have the same meaning as the columns of Table 8.1. The sixth column gives the options for the translation that differ from the standard options we described before. The last column gives the name of the generated SMT-LIB script. Table 8.4 contains the same kind of information as Table 8.2.

In Table 8.5 and Table 8.6 we show that the wrong settings for the translation can imply that the translation plus the SMT solver can give incorrect results. Table 8.5 contains similar information as Table 8.3. In Table 8.6 we give the results of the application of RISCAL and of the translation plus Yices. We do not give any times, as the timings are not of interest for considerations of the correctness. Moreover, we only used the SMT solver Yices, as the other solvers would have given the same results as Yices (if they would not have been terminated because of a timeout).

For the tested RISCAL specifications and the generated SMT-LIB scripts see[3].

8.2 The Analysis of the Tests

In this section we will discuss the results from Table 8.2. We want to point out that we deliberately give no detailed statistical analysis. Thus, we will for example not give assertions, like on average method A is x% faster than method B. We justify this with the observation from Table 8.2 that the performance of the tested methods is highly dependent on the chosen theorem and specification. We can see that a

[3] `https://www.risc.jku.at/research/formal/software/RISCAL/papers/thesis-Reichl.tgz`

Table 8.1: Description

Nr.	File	Theorem	Value 1	Value 2
1	catlogic.txt	Imp1	N=2	M=1
2	catlogic.txt	Forall1	N=2	M=1
3	catlogic.txt	Exists1	N=2	M=1
4	catlogic.txt	And1	N=2	M=1
5	sets.txt	cardinalityT	N=5	
6	sets.txt	associativeUnionT	N=8	
7	sets.txt	_unionR_14_OutputCorrect0	N=7	
8	sets.txt	unth1	N=7	
9	sets.txt	unth2	N=7	
10	sets.txt	unth3	N=7	
11	sets.txt	coth1	N=11	
12	sets.txt	coth2	N=11	
13	sets.txt	coth3	N=11	
14	relation.txt	compositionFromAtoC	N=4	
15	relation.txt	inverseBijectiveT	N=4	
16	relation.txt	composeInverseT	N=4	
17	relation.txt	_isRelationP_1_PostNotTrivialSome	N=3	
18	graphs.txt	handshakingTheorem	N=4	
19	graphs.txt	_handshakingTheorem_8_PreSat	N=3	
20	graphs.txt	_inducedSubGraph_13_PostUnique	N=4	
21	graphs.txt	_getCompleteUndirectedGraph_4_ OutputCorrect0	N=18	
22	graphs.txt	numberOfVerticesOfOddDegree	N=5	
23	gcd1.txt	_gcdf_6_OutputCorrect0	N=100	
24	gcd2.txt	_gcdf_8_OutputCorrect0	N=100	
25	gcd1.txt	_gcdf_6_PostUnique	N=150	
26	gcd2.txt	_gcdf_8_PostUnique	N=100	
27	gcd2.txt	gcdp1	N=100	
28	gcd2.txt	gcdinv	N=150	
29	gcd2.txt	gcdTh1	N=100	
30	bubble.txt	_swap_0_CorrOp0	N=6	M=6
31	bubble.txt	_bubbleSort3_2_CorrOp0	N=4	M=4
32	bubble.txt	_bubbleSort_3_PostNotTrivialAll	N=6	M=4
33	bubble.txt	_bubbleSort_3_PostNotTrivialSome	N=5	M=3
34	poly.txt	_polyRem_38_OutputCorrect0	C=1	D=2
35	poly.txt	_polyRem_38_PreNotTrivial	C=2	D=2
36	poly.txt	polyEqProdSumCorrectness	C=1	D=2
37	poly.txt	hasNegation	C=2	D=2
38	search.txt	_bsearch_3_OutputCorrect0	N=4	M=4
39	search.txt	_bsearchp_1_CorrOp0	N=4	M=4
40	search.txt	_bsearchp_1_CorrOp3	N=4	M=4
41	search.txt	_bsearch_2_PostUnique	N=6	M=4
42	partition.txt	_partition_1_CorrOp0	N=6	M=4
43	partition.txt	_partition_1_PostUnique	N=6	M=4
44	partition.txt	_partition_1_PostNotTrivialAll	N=6	M=4
45	partition.txt	_partition_1_LoopOp4	N=6	M=4
46	sat3.txt	notValidDual	n=6	cn=4
47	sat3.txt	notValid	n=3	cn=2
48	sat3.txt	_DPLL2_14_CorrOp0	n=2	cn=2

Table 8.2: Results

Nr.	Result	RISCAL	Translation	Boolector	Z3	Yices	CVC4
1	valid	71748	30	57	61	92	311
2	valid	2262	7	74	252	51	3649
3	valid	2770	11	63	210	89	9289
4	valid	27894	7	59	46	51	2154
5	valid	102576	8	12654	438	261	27710
6	valid	235413	1	3	6	2	5
7	valid	43427	9	Timeout	66591	61553	Timeout
8	valid	182	2	4	13	4	554
9	valid	137552	5	Timeout	7692	454139	Timeout
10	valid	260395	8	Timeout	Timeout	488063	Timeout
11	valid	39	2	3	9	3	423
12	valid	Timeout	8	134940	207	7653	Timeout
13	valid	Timeout	11	Timeout	747960	33881	Timeout
14	valid	Memory	3	8	18	8	Timeout
15	valid	Memory	8	30	31	11	939
16	valid	Memory	4	11	43	7	Timeout
17	valid	1	75	244	156808	127	926033
18	valid	Memory	4	372	59	44	962
19	valid	0	969	77474	63434	9682	375195
20	invalid	Memory	3	10	9	4	15
21	valid	48624	1026	Timeout	681586	8765	Timeout
22	valid	Timeout	4	426	38	22	8180
23	valid	68964	6	Timeout	524347	3526	Timeout
24	valid	Timeout	11	Timeout	285244	10713	Timeout
25	valid	300740	4	5896	8202	5914	Timeout
26	valid	Timeout	4	8	222400	55	Timeout
27	valid	617606	8	Timeout	235074	5148	Timeout
28	invalid	39	3	Timeout	Timeout	120	Timeout
29	valid	165776	8	Timeout	234459	3775	Timeout
30	valid	253559	1	7	23	6	224
31	valid	10399	32	Timeout	Timeout	25695	Timeout
32	valid	4938	418	125294	428360	1480	Timeout
33	valid	0	33	Timeout	Timeout	82	Timeout
34	valid	Timeout	14	Timeout	46928	7340	Timeout
35	valid	52362	-	Timeout	Timeout	Timeout	Timeout
36	valid	643418	9	333376	469	182	9148
37	valid	178	8	Timeout	7605	178508	851115
38	valid	24	9	Timeout	Timeout	938	Timeout
39	valid	184	2	4	10	5	15
40	valid	205	4	Timeout	Timeout	4635	Timeout
41	invalid	4766	2	6	11	4	22
42	valid	Timeout	197	41468	165199	866	867435
43	invalid	Timeout	204	49735	213906	984	1115391
44	valid	3037	283	Timeout	Timeout	23415	Timeout
45	valid	Timeout	204	41480	164114	906	864440
46	invalid	580919	650	Timeout	Timeout	21488	466905
47	valid	28766	14	52	64	16	170
48	valid	Timeout	7	7	26	10	33

Table 8.1: Description (continuation)

Nr.	File	Theorem	Value 1	Value 2
49	sat3.txt	_DPLL2_14_PostNotTrivialSome	n=3	cn=2
50	sat3.txt	_DPLL_11_OutputCorrect0	n=3	cn=2
51	gsort.txt	_gsort_2_CorrOp0	N=5	M=3
52	gsort.txt	_gsort_2_PostUnique	N=5	M=3
53	gsort.txt	_gsort_2_LoopOp4	N=5	M=3
54	divide.txt	_quotRemProc_11_CorrOp0	N=100	
55	divide.txt	uniqueOutput	N=100	
56	divide.txt	someOutput	N=100	
57	sort2.txt	_sort_3_CorrOp0	N=4	M=4
58	sort2.txt	_sort_3_PostUnique	N=4	M=4
59	sort2.txt	_sort_3_LoopOp7	N=4	M=4
60	matrices.txt	symAdd	N=4	
61	matrices.txt	inv	N=4	
62	matrices.txt	matmultT	N=5	
63	matrices.txt	transpT	N=5	
64	matrices.txt	detT	N=5	
65	matrices.txt	posDef1	N=5	
66	matrices.txt	posDef2	N=5	
67	setPartitions.txt	th1	N=5	
68	setPartitions.txt	th2	N=5	
69	groupExp.txt	assoc	N=6	
70	groupExp.txt	identityInverse	N=10	

Table 8.3: Description

Nr.	File	Theorem	N	M	Option	Scipt
71	demo1.txt	setth1	12			invalidDemo1
72	demo1.txt	setth2	12			invalidDemo2
73	demo2.txt	skolth	10		-smt-skol 0	alwaysSkolemise
74	demo2.txt	skolth	10			skolemiseHeuristic
75	demo2.txt	orderth	8	6	-smt-af 2	noAuxFuns
76	demo2.txt	orderth	8	6		useAuxFuns
77	groupExp.txt	comm	8		-smt-cut 0	noCuts
78	groupExp.txt	comm	8			useCuts

Table 8.5: Description

Nr.	Theorem	N	Option	Scipt
79	chooseGuards1	5	-smt-cguards 0	CGs1_NoGuard
80	chooseGuards1	5	-smt-cguards 1	CGs1_SimpleGuards
81	chooseGuards2	5	-smt-cguards 0	CGs2_NoGuard
82	chooseGuards2	5	-smt-cguards 1	CGs2_SimpleGuards
83	chooseGuards2	5	-smt-cguards 2	CGs2_Guards
84	unAxTh	5	-smt-mod 1 -smt-cut 1	demoNoAxiom
85	unAxTh	5	-smt-mod 1 -smt-cut 3	demoUseAxiom

Table 8.2: Results (continuation)

Nr.	Result	RISCAL	Translation	Boolector	Z3	Yices	CVC4
49	valid	0	426	1190	456711	920	Timeout
50	valid	Timeout	1257	Timeout	Timeout	14885	Timeout
51	valid	Timeout	15	568	2380	91	8874
52	valid	Timeout	16	1147	3333	991	11720
53	valid	Timeout	14	556	2758	92	8881
54	valid	24328	2	4	5	3	8
55	valid	158027	3	146	98	71	68
56	valid	1889	4	26596	9226	11477	342949
57	valid	346001	6	19	39	5	126
58	valid	38694	5	82	97	86	270
59	valid	447211	5	74	96	49	749
60	valid	Timeout	2	6	12	4	13
61	invalid	5318	1529	Timeout	Timeout	43850	Timeout
62	invalid	54117	3	14	22	9	36
63	invalid	83479	4	25	25	9	72
64	invalid	46074	3	20	33	13	57
65	invalid	1089	6	12	149	7	49
66	valid	Timeout	5	81	155	70	148
67	invalid	Timeout	5	54	74	12	155
68	valid	Timeout	4	71	74	12	229
69	valid	17022	2	8	14	6	57
70	valid	37491	11	Timeout	Timeout	29559	Timeout

Table 8.4: Results

Nr.	Result	RISCAL	Translation	Boolector	Z3	Yices	CVC4
71	invalid	42577	1	2	8	3	9
72	invalid	0	1	4	9	2	339
73	invalid	4345	8	142773	Timeout	2863	695026
74	invalid	4578	7	16609	62366	1062	286293
75	valid	0	313	3162	7354	2235	34006
76	valid	0	3	808	2304	6	15950
77	valid	4101	30	Timeout	Timeout	30096	Timeout
78	valid	4074	2	4	40	6	8

Table 8.6: Results

Nr.	Result RISCAL	Result Yices
79	invalid	valid
80	invalid	invalid
81	invalid	valid
82	invalid	valid
83	invalid	invalid
84	valid	invalid
85	valid	valid

Table 8.7: Substantially Fastest Method

	RISCAL	Boolector	Z3	Yices	CVC4
Total	23%	43%	40%	76%	16%
Valid	24%	39%	39%	75%	10%
Invalid	18%	64%	45%	82%	46%

Table 8.8: Substantially Faster as RISCAL

	Boolector	Z3	Yices	CVC4
Total	60%	67%	76%	50%
Valid	58%	66%	75%	44%
Invalid	73%	73%	81%	81%

certain method could deliver a very good performance for one example, whereas it delivered a relatively bad performance for other examples (e.g. using the translation with the solver Z3 delivered a good performance for test 1 while it delivered a bad performance for test 21). For the above reason, we think that a statistical analysis does not give a lot of insight, as it cannot really be generalised.

Instead, we will focus on patterns in RISCAL specifications, respectively in their translation that seem to have an impact on the performance of the SMT-solvers. We will therefore analyse the results given in Table 8.2.

We want to point out that we lack a deep knowledge of the behaviour of the used SMT solvers. Thus, we have to rely on the observations gained through the tests.

We start our discussion of the results with a summary of Table 8.2. We therefore show in Table 8.7, the percentage of tests, where the respective methods were the fastest. In Table 8.8 we show the percentage of tests, where the translation plus one of the SMT solvers was faster than RISCAL. We point out that we did not use a strict ordering for the generation of these tables. This means that in Table 8.2 we count all methods as *fastest* that were either less than 50 milliseconds slower than the absolutely fastest method or that were not more than 10% slower than the absolutely fastest method. Similarly, in Table 8.8 we only consider a method to be faster as RISCAL, if it was more than 50 milliseconds faster than RISCAL and if it was at least 10% faster than RISCAL. This is, on the one hand motivated by restrictions on the precision of the measurement of the time (we use wall clock time, thus a certain deviation cannot be excluded) On the other hand, small differences are not of importance for us. E.g. in test 48 the important observation for us is that the translation with any SMT-solver was much faster than RISCAL, but for us it is not of importance that the translation plus Boolector was 19 milliseconds faster than the translation plus Z3. As a consequence, of this understanding of a fastest method, the results in Table 8.7 do not sum up to 100.

Having in mind the understanding, of what we consider as faster, we can now

give a description of Table 8.7 and Table 8.8. The first row of Table 8.7 gives the percentage of tests where the respective method was the fastest in the above sense. The second row gives the fastest methods for the tests, where the theorem that should be checked was valid. The third row gives the fastest methods for tests with invalid theorems. Similarly, the first row of Table 8.7 gives the percentage of tests, where the respective method was faster than RISCAL, while the second and the third one only cover tests where the theorem that should be checked was valid, respectively invalid.

As we already pointed out at the beginning of this section, we cannot necessarily generalise the results of the tests. This is due to the observation that the relative performance seems to be strongly dependent on the theorem and specification that shall be checked. We have to keep this in mind when we analyse Table 8.7 and Table 8.8.

Nonetheless, Table 8.7 and Table 8.8 illustrate that the translation, presented in this thesis, actually has the potential to substantially speed up the checking of theorems in RISCAL. Moreover, the tables indicate that, for the kind of SMT-LIB scripts that were generated by the translation, the SMT solver Yices seems to be the most suitable one[4]. We can see that the differences between the tested SMT solvers, in terms of performance (also at least partially) correspond to the results of [39] for the Single Query Track of the QF_UFBV logic.

We also want to point out that the two tables show that for the selected test cases the, SMT solvers performed comparably good for invalid as for valid theorems, with respect to RISCAL. We want to highlight this, as initially we thought that RISCAL will perform significantly better for invalid theorems, with universal quantifiers, as for valid ones, relatively to the SMT solvers. Because of this, in the following we discuss the checking of such theorems. Note that similar considerations also apply to valid theorems with existential quantifiers.

In order to check the validity of a theorem, RISCAL evaluates the underlying formula with respect to value assignments for the involved variables. This means for a universally quantified formula:

$$\forall x : T.P(x)$$

Where T is a type and P a predicate, RISCAL evaluates P for the values of T. If RISCAL finds a value that makes P false, RISCAL concludes that the theorem is invalid; if it cannot find such a value, it concludes that the formula is valid. RISCAL proceeds with existentially quantified formulae similarly. For the evaluation, RISCAL makes use of certain enumerations, for all types, that determine the order in which the value assignments are evaluated. If in the above example T would be $\mathbb{Z}[0, 10]$, this would for example mean that RISCAL first evaluates $P(0)$, then $P(1)$

[4] Additionally to the presented tests, various further tests that were conducted during the implementation of the test showed that Yices was regularly substantially faster than all or at least some of the other tested SMT solvers. Moreover, Yices was only in a few cases substantially slower than at least one of the other solvers.

Listing 8.1: additionalSpecs2.txt

```
val N:Nat;

type nat=Nat[N];
type natset=Set[nat];

theorem setth1(a:natset,b:natset)<=>|a∩b|<=N;
theorem setth2(a:natset,b:natset)<=>|a∩b|>0;
```

and so on until $P(10)$ is reached. We can see that the position (with respect to RISCAL's ordering) of the first falsifying value assignment has a significant influence on RISCAL's performance. For a similar reason, we can also see that RISCAL can often analyse invalid theorems with universal quantifiers significantly faster, as valid theorems with a similar structure. The reason for this is that for the valid theorems all value assignments have to be evaluated, while for an invalid theorem potentially much fewer value assignments have to be evaluated.

This behaviour of RISCAL for invalid theorems is described by the tests 71 and 72 from Table 8.2. We give the specification for both tests in Listing 8.1. We can see that the assertion of `setth1` is only false for $a = b = \{0, 1 \cdots, N\}$. This is disadvantageous for RISCAL as $a = b = \{0, 1 \cdots, N\}$ is the last value assignment that is used for the evaluation. This means that RISCAL has to evaluate the assertion of the theorem for all other possible values for a and b, which takes a long time as it can be seen in Table 8.2. In contrast to this, the assertion of `setth2` is false for $a = b = \emptyset$. This is advantageous for RISCAL, as this is the first value assignment RISCAL evaluates. Thus, RISCAL can determine the invalidity of the theorem in a very fast way. We can see that there is a significant difference in the time RISCAL needed to analyse the two theorems, although they can be considered as comparable complex. On the other hand the SMT solvers deliver comparable results for both theorems (not considering the results of CVC4).

In summary, the above considerations justify the assumption that a generalisation of the results of the invalid tests is even more difficult, as for the valid ones. Moreover, the above considerations might raise the idea that the usage of RISCAL is preferable for such invalid theorems, compared to other theorems (as we initially thought). But we want to point out that this is not necessarily the case. Firstly, we have seen that the tests give a differentiated picture. But more importantly, it also seems to be reasonable to assume that the SMT solvers will also give a result faster for such invalid theorems, as for comparable valid theorems. This is motivated by the following intuition: To check a theorem containing universal quantifiers, we get existential quantifiers because of negation. These existential quantifiers are then eliminated by means of skolemisation. If the SMT solver finds an interpretation for the skolem functions satisfying the formula that shall be checked, then the SMT solver does not need to consider further interpretations. Whereas, if the theorem is valid, all interpretations of the skolem functions need to be considered.

Although, we do not think that for a general invalid theorem, RISCAL is usually faster, we still think, the application of RISCAL is preferable for certain invalid theorems – namely, for theorems with major flaws. This assumption is motivated by the consideration that for a theorem with a major flaw, it is likely that the first few value assignments, RISCAL tries, already revealed the invalidity of the theorem. If this is the case, RISCAL can almost instantaneous determine the invalidity of the theorem (see test 72). On the other hand, we think that SMT solvers will generally be not able to benefit of this in the same extent. As a consequence of this consideration, we think that it might be a good idea to use RISCAL to analyse the first drafts of theorems, as such theorems will often contain crucial flaws.

Next, we want to discuss two properties of RISCAL specifications that seem to have a negative impact on the performance of the SMT solvers. First, we will discuss specifications where the theorem that shall be checked, contains mainly existential quantifiers. Secondly, we will discuss specifications, whose translations require multiple SMT-LIB assertions. Consequences of both properties are apparently more complex[5] SMT-LIB scripts – as RISCAL specifications, with the above properties, often result in longer SMT-LIB scripts, respectively in SMT-LIB scripts with multiple assertions.

If we want to analyse a RISCAL theorem with existential quantifiers, we know that we have to translate the existential quantifiers to universal quantifiers, as we have to negate the theorem of interest. As we must not use quantifiers in the resulting SMT-LIB scripts, we have to expand these quantifiers. A result of this expansion is that formulae with existential quantifiers result in substantially larger formulae than comparable formulae that contain universal quantifiers instead (as universal quantifiers are translated to existential quantifiers and thus usually skolemisation can be applied). An example for such a difference in the sizes of the translation is given by the tests 48 and 49. As already mentioned earlier, we think that the size of the translation is an indicator for the complexity of the analysis that has to be performed by the SMT-solvers. Thus, it seems to be reasonable to assume that in general, checking RISCAL theorems with existential quantifiers by SMT solvers takes longer than checking RISCAL theorems with universal quantifiers. This assumption also reflects the results from the tests (e.g. compare test 48 with test 49 or test 17 with test 18).

Similarly, as for the invalid theorems with universal quantifiers, we can also recognise that the structure of theorems has a major impact on the performance. Thus, for example RISCAL could deliver results for the test cases 17, 19, 33 and 49 very fast, as RISCAL soon tried a value assignment that makes the respective formula true.

We want to point out that the dominance of theorems with universal quantifiers in the specifications, that were available to us, was the reason why the tests contain

[5] Again we want to point out that we lack a deep understanding for all of the used SMT solvers. Thus, we cannot be sure, if such SMT-LIB scripts are actually more complex. Nevertheless, it seems to be reasonable to assume that longer scripts, respectively scripts with several *asserts*, are generally more complex for the used SMT solvers.

more RISCAL theorems with universal quantifiers than with existential quantifiers.

Next we want to consider RISCAL specifications whose translations contain multiple assertions. We can differentiate between two reasons for this. Firstly, if a specification contains multiple theorems to translate, multiple assertions have to be used. Secondly, the translation of some components of the RISCAL language requires to introduce additional auxiliary assertions.

We start our considerations with the first cause. If a specification contains several theorems, then the additional theorems are in principle treated similar as the theorem that shall be checked. The difference is that – in contrast to the theorem that shall be checked – these additional theorems are not negated. The influence, such theorems can have on the performance of the SMT solvers, is then determined by the complexity of the underlying formula. Thus, a theorem without any quantifiers will usually have only a very small and limited impact on the performance. But especially universally quantified formulae often seem to have a significant negative impact on the performance, as these quantifiers need to be expanded. When we compare the results of the test cases 77 and 78, we see that test 77 was substantially slower checked, by the SMT solvers, than test 78, as the unnecessary theorems are not cut off in this test case. Fortunately, these theorems are rarely needed. Only if a specification contains functions that, on the one hand, shall be treated in a modular way (i.e. it shall be introduced via its specification) and on the other hand are not uniquely defined by the specification, such theorems can be necessary. Because, if a theorem contains such a function, this theorem can potentially forbid certain interpretations of the function. We illustrate this with the tests 84 and 85. In these tests the following RISCAL declarations are of importance:

```
...
fun unionF(a:set1,b:set1):set1
modular;
ensures a⊆result ∧ b⊆result;
=a∪b;

axiom unAx(a:set1,b:set1)
<=>let r=unionF(a,b) in
forall s:set2 with (s⊆r∧s≠r).¬(a⊆r)∨¬(b⊆r);

theorem unAxTh(a:set1,b:set1)<=>
unionF(a,b)=a∪b;
```

The options we used for the tests imply that the function `unionF` was introduced by its specification instead of its definition. We can see that the specification does not fully cover the meaning of the definition. In test 84 we do not use the axiom `unAx`, thus all we know about the `unionF` is given by the specification. Therefore, the SMT solver reports that the theorem `unAxTh` does not hold, as `unionF` can give any common super set of its arguments and not necessarily the union. On the other hand, in test 85 we use the axiom. Thus, in the translation, `unionF` corresponds to the union. As a consequence of this, the SMT solver reports that the theorem holds.

As additional theorems are only needed in the case described above, they can often

be cut away (all additional theorems in all the test cases except 84 and 85 could be cut off). To sum up, this means that the described kind of theorems can have a significant influence on the performance. But as they are often not needed they are usually not a problem.

Secondly, the translation of several kinds of expressions require the usage of undefined functions. As we often have to impose certain restrictions on these functions, they often require the usage of additional assertions. Subsequently, we will discuss the important reasons that cause the usage of such functions and of additional assertions.

Functions that Shall be Introduced via Their Specification This involves, on the one hand procedures and on the other hand, functions that shall be treated in a modular way. For the representation of such functions in SMT-LIB, first a function with an appropriate arity is declared. Then a single formula is computed from the specification. This formula involves a universal quantification over the types of the arguments of the function. The SMT-LIB representation of this formula is then asserted. Next, a differentiation between two cases has to be done – either the domain of the declared SMT-LIB function can be bijectively mapped to the domain of the corresponding RISCAL function or not. In the first case, the previous steps suffice in order to represent the given function. In the second case, we have to assert that the SMT-LIB function attains, for all possible arguments, a value that represents (as discussed in Chapter 5) some value from the image of the original RISCAL function. The second assertion would not be necessary, if the specification would uniquely define the function. Nevertheless, the translation always generates this assertion, such that functions, whose specification do not uniquely define the function, also can be used. In the following, we will give a basic example, which shall in particular point out the necessity of the second assertion.

Assume we have to translate a RISCAL specification, with the following procedure:

```
...
proc P(i:ℤ[-2,2]):ℤ[-2,2]
ensures result≥0;
...
```

then we get an SMT-LIB script containing the following:

```
...
(declare-fun P ((_ BitVec 3)) (_ BitVec 3))
(assert (and
(let ((i #b110)) (and (bvsle #b110 (P i)) (bvsge #b010 (P i)))) ···
(let ((i #b010)) (and (bvsle #b110 (P i)) (bvsge #b010 (P i))))))
(assert (and
(let ((i #b110)) (let (( result (P i))) (bvsge result #b000))) ···
(let ((i #b010)) (let (( result (P i))) (bvsge result #b000)))))
...
```

(Note that in the above translation the second assertion contains the information from the specification of the procedure.) We can see: if we would omit the first assertion, the function could for example attain the value 0111 for some argument values. As 0111 represents the number 7, this means that the function would no longer fit to the RISCAL procedure.

The above considerations illustrate that such functions have the potential to significantly increase the size of translations. Indeed, the tests 8 and 9 respectively 11 and 12, indicate that a procedure can result in a substantially larger translation, than a similar RISCAL function. Moreover, these tests show that the tests with the procedures are checked much slower than the others by the SMT solvers. Nevertheless, these tests also indicate that for RISCAL the test with the procedure is also more difficult to analyse. So, RISCAL is not necessarily more suitable for specifications with such functions.

Recursive Functions RISCAL specifications may contain recursively defined functions. We cannot directly use recursively defined functions in the translations[6]. Thus, we introduce recursive functions with defining axioms. Subsequently, we illustrate this procedure with an example. Assume we have a RISCAL specification with the following recursive function:

```
...
fun f(n:Nat[2]):Nat = if n=0 then 0 else 1+f(n-1)
...
```

For this function the defining axiom in the RISCAL snytax would be given by:

```
...
axiom fDef(n:Nat[2])<=>f(n)= (if n=0 then 0 else 1+f(n-1))
...
```

The translation will give the following representation of this function.

```
...
(declare-fun f ((_ BitVec 2)) (_ BitVec 2))
(assert (and (let ((n #b00)) (= ((_ zero_extend 1) (f n))
(ite (= n #b00) #b000 (bvadd #b001 ((_ zero_extend 1)
(f ((_ extract 1 0) (bvsub ((_ zero_extend 1) n) #b001)))))))) ...
(let ((n #b11)) (= ((_ zero_extend 1) (f n))
(ite (= n #b00) #b000 (bvadd #b001 ((_ zero_extend 1)
(f ((_ extract 1 0) (bvsub ((_ zero_extend 1) n) #b001)))))))))
...
```

The above considerations illustrate that recursive functions have the potential to increase the size of the translation, as the defining axiom has to be expanded. The influence of recursive functions is illustrated with the tests 8 and 10,

[6] Z3 and CVC4 could support the direct definition of recursive functions via the SMT-LIB keyword *define-fun-rec*. On the other hand Boolector and Yices do not support the keyword *define-fun-rec*. Thus, we cannot recursively define functions in Boolector and Yices directly. As we want that all of the four SMT-solvers can be applied to the translation, we do not make use of *define-fun-rec*.

respectively with 11 and 13. Note that in the test 10 and 13, additionally to the recursive functions, choose expressions are also present. In the following, we will discuss that also choose expressions seem to also have an influence on the performance.

Chose Expressions In subsection 7.1.1 we already have discussed that for the translation of the three kinds of choose expressions, respectively for minimum and maximum expressions, we require additional theorems. This means that (most) expressions, of the above types, require additional assertions.

We want to point out that in contrast to the previous reasons for additional assertions, the choose expressions occurred more frequently. Thus, for the conducted tests, they seem to generally have a higher influence on the performance.

We want to highlight three factors that seem to be indicators for the added complexity by a choose expression. Firstly, and most important, the number of free variables in the expression (this includes also all the existentially quantified variables, even if they are actually no free variables in the expression) and the sizes of the types of the variables. This factor is of importance, as it determines the extent of quantifier expansions that are necessary. Secondly, the number of the quantified variables of the expression as well as the size of the types of the quantified variables, are of importance. As these variables are used for an existential quantification, they often have a smaller influence, as the first factor, as existential quantifiers can often be eliminated by skolemisation. Finally, choose expressions that require the option *-smt-cguards 2* can be a problem, as they require an additional existential quantifier that cannot be skolemised. These factors seem to be important, as they have a major influence on the size of the representation of the choose expressions.

Examples for the best identification of the influence of choose expressions, are the tests 30 and 31, respectively 39 and 40. We can see that in the translations, of both, tests 31 and 40, a large part of the translation is dedicated to the representation of the choose expressions. Moreover, we can see that, while RISCAL checked test 31 much faster than 30, the opposite is true for the SMT solvers. For us the only plausible explanation for this are the choose expressions in 31. Similarly, we can see that RISCAL analysed tests 39 and 40 comparably fast, while the SMT solvers analysed test 40 much slower than 39.

Before we finish the considerations about the choose expressions, we will briefly discuss the significance of the option *-smt-cguards 1* and *-smt-cguards 2*. We will therefore regard the tests 79 – 83. For the tests 79 and 80 we used the following RISCAL theorem:

```
...
theorem chooseGuards1(x:nat)
requires x≠0;
⇔
```

```
(choose y:nat with y<x)>0;
...
```

We can see that this theorem does not hold. Nevertheless, if we use the translation with the default settings, the system will report that the theorem is valid. The reason for this is that the assertion restricting the function that represents the choose, is unsatisfiable. Thus, the SMT solver will report *unsat*. Because of this, the system then reports that the theorem is valid. If we use the option *-smt-cguards 1* or *-smt-cguards 2* instead, the system will report that the theorem is invalid. Because with these options, the assertion that was unsatisfiable before, is now satisfiable.

For the tests 81 – 83 we used the following RISCAL theorem:

```
...
pred P(x:Nat[N],y:Nat[N])<=>(x+y)<N;
theorem chooseGuards2(x:nat,y:nat)
requires P(x,y);
⇔
(choose z:nat with z>x)<N;
...
```

We immediately see that the theorem is invalid. Nevertheless, if we do not use the option *-smt-cguards 2*, the system will report that the theorem is valid. The reason for this is analogous to the previous example.

Skolem Functions As we already discussed in subsection 7.2.3, usually skolem functions require certain restrictions. These restrictions are no problem for existential quantifiers that are on top of a formula, as in this case we only need a skolem constant. Thus, we can just require that this constant has the required property. But on the other hand, if we have an existential quantifier that depends on universal quantifiers, then the skolem function has to depend on the universally quantified variables. Thus, asserting the restriction, requires the expansion of universal quantifiers. This can result in long and costly assertions. Test 73 illustrates this possible negative influence of skolemisation.

Before we finish this section, we also want to discuss the influence of some of the optimisation options, the translation provides, briefly. We want to highlight that, in order to determine the most suitable options, additional tests, as those given in Table 8.3, were conducted for the different options. Nevertheless, we think that those mentioned exemplary tests have to suffice.

Limit Use of Skolemisation In subsection 7.2.3 we discussed that the usage of skolemisation can be more costly than the expansion of existential quantifiers. We want to illustrate this with the tests 73 and 74. In test 73 the program uses skolemisation for all eligible existential quantifiers (*-smt-skol 0*), while in test 74 the program heuristically decides whether skolemisation pays off (*-smt-skol 1*). In both tests we analyse the following theorem:

```
...
theorem skolth()⇔
¬(forall a:set,b:set.∃n:nat. |a∪b|=n);
...
```

In order to translate the theorem, we have to negate the theorem. Thus, we get a formula with two universal quantifiers and one existential quantifier. We can see that we need bit vectors of length four to represent `nat` ($\mathbb{N}[10]$). Obviously, bit vectors of length four can represent more values than `nat` provides. Thus, the skolem function has to be restricted. But, as in this test case the skolem function has to depend on a and b. This can be costly. This means that the skolemisation that is used in test 73 seems to be costly. In contrast to this, in test 74 the program decides that skolemisation does not pay off. Thus, the existential quantifier is expanded.

We can on the one hand, see that the translation of test 73 is substantially larger than the other translation. Moreover, all SMT solvers analysed the theorem in test 74 much faster than the theorem in test 73. This shows, that at least in this example, that the skolemisation heuristic pays off.

Auxiliary Functions For Quantifier Bodies. In Section 7.2.2 we discussed that the translation provides the option, of using auxiliary functions, for the translation of quantifiers, in order to reduce the size of the translations of nested quantifiers. The tests 75 and 76 illustrate the influence of the option. While we make use of these auxiliary functions in test 76, we do not use them in test 75. We can see that, as a consequence of this option, the translation of test 75 is drastically larger, than the translation of test 76. Moreover, all SMT-solvers finish much faster for test 76 as for test 75.

Cut Unnecessary Parts of RISCAL Specifications In Section 7.2.1 we discussed that the translation provides the option of cutting away unneeded pars of specifications. We illustrate the influence of this option with the tests 77 and 78. While in test 77 we do not apply any cuts, we remove all unneeded parts in 78. We can see that the cutting in test 78 leads to a smaller translation and to a shorter run time of all the SMT solvers.

9 Conclusion

In this thesis we developed a translation for the subset of the RISCAL language without recursive types to the SMT-LIB lnaguage with the QF_UFBV logic. This includes, on the one hand, the implementation of this translation in the programming language Java and on the other hand a formal description of the core components of this translation.

This translation, in particular, required two problems to be solved. The first problem was that while RISCAL supports quantifiers, SMT-LIB with the QF_UFBV logic does not. Therefore, we had to eliminate RISCAL's quantifiers. This was achieved by the techniques of *quantifier expansions* and *skolemisation*. The second and more difficult problem was that there are substantial differences between the data types and the operations that are provided by RISCAL and by SMT-LIB: on the one hand, RISCAL provides a comprehensive collection of data types like integers, arrays and sets with various operations for the individual types; on the other hand SMT-LIB provides booleans and bit vectors. Thus, we had to find suitable representations for all of RISCAL's types, by bit vectors and boolean values. Moreover, we also had to represent the large collection of operations RISCAL provides by the operations SMT-LIB provides.

Finally, we then tested the implementation of the translation, with several different RISCAL specifications. Here we tried to use several existing RISCAL specifications, instead of artificially constructed ones, in order to demonstrate the translation on the basis of specifications as they can occur in real life. We analysed the translations of these tests with the SMT solvers Boolector, Z3, Yices and CVC4 and compared the results with checks done by RISCAL. This comparison indicated that the translation, in particular used together with the SMT solver Yices, is regularly able to analyse theorems substantially faster than RISCAL. Nevertheless, the translation with the different solvers could not perform better than RISCAL for all of the executed tests. We thus think that this translation represents a useful supplement for the RISCAL system, although it can certainly not replace RISCAL's checking mechanism.

There are several interesting directions to extend this thesis.

- In this thesis we only considered RISCAL specifications with non-recursive types. Thus, the translation could be extended to recursive types such that a full coverage of the RISCAL language can be provided.

- In RISCAL we usually have to check several proof obligations for a single function/procedure/theorem/$\cdots$. These theorems often depend on the same parts of the specification. Thus, it seems to be inefficient to individually translate and check these proof obligations. Instead, we could translate multiple

different proof obligations to a single SMT-LIB script. This script could then by analysed by an incremental application of an SMT solver (the solvers used in this theses, all support an incremental application).

- We saw that it can be inefficient to check theorems that mostly contain existential quantifiers instead of universal quantifiers. If such a theorem does not contain any choose-like expressions and the translation of the associated specification only contains uniquely defined functions (the representations of functions that shall be either introduced via a defining axiom or by their specification only need to be uniquely given for arguments that correspond to some argument of the original RISCAL function) the translation could be adapted. In this case we do not need to negate the theorem. Because in this case, if the unnegated translation of the theorem is satisfiable, the theorem holds. As we do not negate the theorem — this means the existential quantifiers remain — we can apply skolemisation instead of quantifier expansion to eliminate the quantifiers.

- For invalid theorems it is often not sufficient to know that the theorem is invalid – instead one usually also wants to know counterexamples for the theorem. Thus, it would be interesting to use the model generation capabilities of the SMT solvers, to find counterexamples for invalid theorems. For a universally quantified theorem, where all the resulting existential quantifiers can be eliminated by skolemisation, this would require to backward translate the models of the skolem functions. More work would be required for theorems, where the resulting existential quantifiers are expanded – as the solvers would not give falsifying value assignments in this case.

- As soon as the SMT solvers provide a (better) support for quantifiers and recursive functions, it would be interesting to use these capabilities in the translation.

Although the translation is subject to certain restrictions and there is still potential for improvements, this thesis illustrates the potential of the application of SMT solvers.

Bibliography

[1] Jean-Raymond Abrial. *Modeling in Event-B. System and Software Engineering.* Cambridge, UK: Cambridge University Press, 2010.

[2] Clark Barrett, Pascal Fontaine and Cesare Tinelli. *The Satisfiability Modulo Theories Library (SMT-LIB).* www.SMT-LIB.org. 2016. (Visited on 2nd Apr. 2020).

[3] Clark Barrett, Pascal Fontaine and Cesare Tinelli. *The SMT-LIB Standard: Version 2.6.* Tech. rep. Available at www.SMT-LIB.org. Department of Computer Science, The University of Iowa, 2017.

[4] Clark Barrett and Cesare Tinelli. *Satisfiability Modulo Theories.* In: *Handbook of Model Checking.* Ed. by Edmund M. Clarke et al. Springer, 2018, pp. 305–343. DOI: 10.1007/978-3-319-10575-8_11.

[5] Clark Barrett et al. *CVC4.* In: *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11).* Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Vol. 6806. Lecture Notes in Computer Science. Snowbird, Utah. Springer, July 2011, pp. 171–177. URL: http://www.cs.stanford.edu/~barrett/pubs/BCD+11.pdf (visited on 2nd Apr. 2020).

[6] Armin Biere and Daniel Kröning. *SAT-Based Model Checking.* In: *Handbook of Model Checking.* Ed. by Edmund M. Clarke et al. Springer, 2018, pp. 277–303. DOI: 10.1007/978-3-319-10575-8_10.

[7] *Boolector.* URL: https://boolector.github.io/ (visited on 2nd Apr. 2020).

[8] Aaron R. Bradley and Zohar Manna. *The Calculus of Computation. Decision Procedures with Applications to Verification.* Berlin, Germany: Springer, 2007. DOI: 10.1007/978-3-540-74113-8.

[9] Alexander Brunhuemer. *Validating the Formalization of Theories and Algorithms of Discrete Mathematics by the Computer-Supported Checking of Finite Models.* Bachelor Thesis. Johannes Kepler University, Linz, Austria: Research Institute for Symbolic Computation (RISC), Sept. 2017.

[10] Edmund M. Clarke, Thomas A. Henzinger and Helmut Veith. *Introduction to Model Checking.* In: *Handbook of Model Checking.* Ed. by Edmund M. Clarke et al. Springer, 2018, pp. 1–26. DOI: 10.1007/978-3-319-10575-8_1.

[11] Edmund Clarke et al. *Bounded Model Checking Using Satisfiability Solving.* In: *Formal Methods in System Design* 19 (2001), pp. 7–34. DOI: 10.1023/A:1011276507260.

[12] David R. Cok. *OpenJML: Software verification for Java 7 using JML, OpenJDK, and Eclipse.* In: *Proceedings 1st Workshop on Formal Integrated Development Environment, F-IDE 2014, Grenoble, France, April 6, 2014.* Ed. by Catherine Dubois, Dimitra Giannakopoulou and Dominique Méry. Vol. 149. EPTCS. 2014, pp. 79–92. DOI: 10.4204/EPTCS.149.8.

[13] *CVC4.* URL: http://cvc4.cs.stanford.edu/web/ (visited on 2nd Apr. 2020).

[14] Leonardo De Moura and Nikolaj Bjørner. *Satisfiability modulo Theories: Introduction and Applications.* In: *Commun. ACM* 54.9 (Sept. 2011), pp. 69–77. ISSN: 0001-0782. DOI: 10.1145/1995376.1995394.

[15] David Déharbe. *Integration of SMT-solvers in B and Event-B development environments.* In: *Science of Computer Programming* 78.3 (2013). Abstract State Machines, Alloy, B and Z - Selected Papers from ABZ 2010, pp. 310–326. ISSN: 0167-6423. DOI: 10.1016/j.scico.2011.03.007.

[16] David Déharbe et al. *Integrating SMT solvers in Rodin.* In: *Science of Computer Programming* 94 (2014). Abstract State Machines, Alloy, B, VDM, and Z, pp. 130–143. ISSN: 0167-6423. DOI: 10.1016/j.scico.2014.04.012.

[17] Edsger W. Dijkstra. *The Humble Programmer.* In: *Commun. ACM* 15.10 (Oct. 1972), pp. 859–866. ISSN: 0001-0782. DOI: 10.1145/355604.361591.

[18] Bruno Dutertre. *Yices 2.2.* In: *Computer-Aided Verification (CAV'2014).* Ed. by Armin Biere and Roderick Bloem. Vol. 8559. Lecture Notes in Computer Science. Springer, July 2014, pp. 737–744. DOI: 10.1007/978-3-319-08867-9_49.

[19] Aboubakr Achraf El Ghazi and Mana Taghdiri. *Analyzing Alloy Constraints using an SMT Solver: A Case Study.* In: *5th International Workshop on Automated Formal Methods (AFM); 14.07.2010, Edinburgh.* 2010, pp. 1–8. DOI: 10.5445/IR/1000042224.

[20] Aboubakr Achraf El Ghazi et al. *A Dual-Engine for Early Analysis of Critical Systems.* In: *Workshop on Dependable Software for Critical Infrastructures (DSCI).* Berlin, Germany, 2011.

[21] John Harrison. *Handbook of Practical Logic and Automated Reasoning.* Cambridge, UK: Cambridge University Press, 2009.

[22] Daniel Jackson. *Software Abstractions. Logic, Language, and Analysis.* MIT Press, 2006.

[23] Daniel Kroening and Ofer Strichman. *Decision Procedures. An Algorithmic Point of View.* Berlin, Germany: Springer, 2016. DOI: 10.1007/978-3-662-50497-0.

[24] Teimuraz Kutsia. *Rewriting in Computer Science and Logic.* Linz, Austria: Research Institut for Symbolic Computation (RISC), Johannes Kepler University, Summer Semester 2019. URL: https://www3.risc.jku.at/education/courses/ss2019/rewriting/ (visited on 2nd Apr. 2020).

[25] Leslie Lamport. *Specifying Systems. The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[26] Gary T. Leavens et al. *JML Reference Manual*. 2013. URL: http://www.eecs.ucf.edu/~leavens/JML//refman/jmlrefman.pdf (visited on 2nd Apr. 2020).

[27] María Manzano. *Extensions of First Order Logic*. Cambridge Tracts in Theoretical Computer Science 19. Cambridge, UK: Cambridge University Press, 1996.

[28] Yuri V. Matiyasevich. *Hilbert's 10th Problem*. Foundations of Computing. 1993.

[29] Stephan Merz and Hernán Vanzetto. *Encoding TLA+ into Many-Sorted First-Order Logic*. In: *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 5th International Conference, ABZ 2016*. Ed. by Michael J. Butler et al. Vol. 9675. Lecture Notes in Computer Science. Linz, Austria: Springer International Publishing, 2016, pp. 54–69. ISBN: 978-3-319-33600-8. DOI: 10.1007/978-3-319-33600-8_3.

[30] Leonardo de Moura and Nikolaj Bjørner. *Z3: An Efficient SMT Solver*. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340. ISBN: 978-3-540-78800-3.

[31] Aina Niemetz, Mathias Preiner and Armin Biere. *Boolector 2.0 system description*. In: *Journal on Satisfiability, Boolean Modeling and Computation* 9 (2014 (published 2015)), pp. 53–58.

[32] Arnold Oberschelp. *Untersuchungen zur mehrsortigen Quantorenlogik*. German. In: *Mathematische Annalen*. Vol. 145. Springer, 1962, pp. 297–333.

[33] Sandip Ray. *Scalable Techniques for Formal Verification*. Boston, MA: Springer, 2010. DOI: 10.1007/978-1-4419-5998-0.

[34] Wolfgang Schreiner. *RISCAL*. Version 2.12.2. URL: https://www3.risc.jku.at/research/formal/software/RISCAL/ (visited on 2nd Apr. 2020).

[35] Wolfgang Schreiner. *The "parallel" Command*. 2013. URL: https://www3.risc.jku.at/people/schreine/ (visited on 2nd Apr. 2020).

[36] Wolfgang Schreiner. *The RISC Algorithm Language (RISCAL). Tutorial and Reference Manual*. Version 2.12.*. Research Institut for Symbolic Computation (RISC), Johannes Kepler University. Linz, Austria, 2020.

[37] Wolfgang Schreiner, Alexander Brunhuemer and Christoph Fürst. *Teaching the Formalization of Mathematical Theories and Algorithms via the Automatic Checking of Finite Models*. In: *Proceedings 6th International Workshop on Theorem proving components for Educational software (ThEdu'17)* (Gothenburg, Sweden, 6th Aug. 2017). Ed. by Pedro Quaresma and Walther Neuper. Vol. 267. Electronic Proceedings in Theoretical Computer Science (EPTCS). 2018, pp. 120–139. DOI: 10.4204/EPTCS.267.8.

[38] Wolfgang Schreiner, Josef Schicho and Wolfgang Windsteiger. *Formal Modeling*. Linz, Austria: Research Institut for Symbolic Computation (RISC), Johannes Kepler University, Summer Semester 2019. URL: https://moodle.jku.at/jku/course/view.php?id=5130 (visited on 2nd Apr. 2020).

[39] *SMT-COMP 2019*. URL: https://smt-comp.github.io/2019/ (visited on 2nd Apr. 2020).

[40] Hernán Vanzetto. *Proof automation and type synthesis for set theory in the context of TLA+*. PhD thesis. Université de Lorraine, Dec. 2014. URL: https://hal.inria.fr/tel-01751181.

[41] Franz Winkler. *Polynomial Algorithms in Computer Algebra*. Berlin, Heidelberg: Springer-Verlag, 1996. ISBN: 3211827595.

[42] *Yices2*. URL: https://yices.csl.sri.com/ (visited on 2nd Apr. 2020).

[43] *Z3*. URL: https://github.com/Z3Prover/z3 (visited on 2nd Apr. 2020).

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Masterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe. Die vorliegende Masterarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

.....

Franz Reichl