Extending the Automated Theorem Prover nanoCoP with Arithmetic Procedures

Bachelor’s Thesis

Universität Potsdam

Leo Repp

Primary Supervisor: Prof. Dr. Christoph Kreitz

Secondary Supervisor: Mario Frank

Mathematisch-Naturwissenschaftliche Fakultät

Institut für Informatik und Computational Science

Submission on the 9th of September 2022

Inhaltsangabe

In dieser Bachelorarbeit implementiere ich den automatischen Theorembeweiser nanoCoP-Ω. Es handelt sich bei diesem neuen System um das Ergebnis einer Portierung von Arithmetik-behandelnden Prozeduren aus dem automatischen Theorembeweiser mit Arithmetik leanCoP-Ω in das System nanoCoP. Dazu wird zuerst der mathematische Hintergrund zu automatischen Theorembeweisern und Arithmetik gegeben. Ich stelle die Vorgängerprojekte leanCoP, nanoCoP und leanCoP-Ω vor, auf dessen Vorlage nanoCoP-Ω entwickelt wurde. Es folgt eine ausführliche Erklärung der Konzepte, um welche der nicht-klausale Konnektionskalkül erweitert werden muss, um eine Behandlung von arithmetischen Ausdrücken und Gleichheiten in den Kalkül zu integrieren, sowie eine Beschreibung der Implementierung dieser Konzepte in nanoCoP-Ω. Als letztes folgt eine experimentelle Evaluation von nanoCoP-Ω. Es wurde ein ausführlicher Vergleich von Laufzeit und Anzahl gelöster Probleme im Vergleich zum ähnlich aufgebauten Theorembeweiser leanCoP-Ω auf Basis der TPTP-Benchmark durchgeführt. Ich komme zu dem Ergebnis, dass nanoCoP-Ω deutlich schneller ist als leanCoP-Ω ist, jedoch weniger gut geeignet für größere Probleme. Zudem konnte ich feststellen, dass nanoCoP-Ω falsche Beweise liefern kann. Ich bespreche, wie dieses Problem gelöst werden kann, sowie einige mögliche Optimierungen und Erweiterungen des Beweissystems.

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne Hilfe Dritter und ohne Zuhilfenahme anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe. Die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen sind als solche kenntlich gemacht.

Die Plagiatsrichtlinie des Senats, im Internet unter https://www.uni-potsdam.de/am-up/2011/ambek-2011-01-037-039.pdf (Zugriff am 08.09.2022) verfügbar, habe ich zur Kenntnis genommen.

Leo Repp
Bachelor Informatik / Computational Science an der Universität Potsdam

______________________________ ______________________________

Ort, Datum Unterschrift

1 Introduction

Automated theorem proving is an active research field in computer science with many applications [29]. Automated theorem provers are programs capable of solving logic problems without human interaction. These programs combine the logical correctness and completeness of mathematical calculi with the speed and reliability of computers and have many use cases. They can be used to prove mathematical formulas for example from geometry or number theory. The most famous proof by an automated theorem prover is the automated proof of the 4-color-theorem [4]. Other domains where automated theorem provers can be used include software verification, natural language processing, biology, or geography. Model checkers, which are programs that are closely related to automated theorem provers, have been used by Intel to verify hardware for over 15 years [10]. Answer set solvers, which are also related to automated theorem provers, can be used to plan transportation in logistics, for example to plan the movement of robots in automated warehouses [2]. An extensive list of possible use-cases for automated theorem provers has been collected by the TPTP library [30].

Interactive theorem provers can assist in mathematical research and can utilise automated theorem provers. Interactive theorem provers store proof steps and data structures to assist the researcher and can check the proof the researcher writes for correctness of the proof upon command. Interactive theorem provers can be used in domains of mathematics for which first-order logic is not expressive enough and that are thus not available to certain automated theorem provers such as nanoCoP-Ω. It is possible to integrate automated theorem provers into this process though: they can be used to fill gaps in the proof that the researcher does not wish to prove themselves or they can be run in parallel to the researcher's thought process. Of course, the interactive theorem prover system needs to adjust the automated theorem prover's input to be in first-order logic for first-order provers, and this is not possible for all formulas.

Many problems can be described with first-order logic due to its large expressiveness. However, very few use cases can be expressed in first-order logic without arithmetic expressions and equalities. Of the 2231 typed first-order logic problems in the TPTP benchmark 74% use equalities and 49% use arithmetic[1]. This illustrates the need for arithmetic handling capabilities in automated theorem provers.

This thesis extends the automated theorem prover for first-order logic nanoCoP [20], which does not possess arithmetic handling capabilities and only rudimentary non-arithmetic equality handling capabilities, by procedures capable of handling arithmetic expressions and arithmetic equalities. nanoCoP is an automated natural non-clausal connection-based prover developed by Jens Otten [20]. The new handling procedures are those that were added to the automated clausal connection-based

[1] https://tptp.org/cgi-bin/SeeTPTP?Category=Documents&File=TFFSynopsis

prover leanCoP in the project named leanCoP-Ω [15]. The goal of adding these procedures to nanoCoP was to construct the automated typed first-order logic theorem prover nanoCoP-Ω, which was expected to have two major benefits over leanCoP-Ω due to the usage of non-clausal form vs. clausal normal form. First, the prover was hoped to be faster. Second, the prover is expected to provide human-readable proofs more easily for use within interactive theorem provers. Testing shows that nanoCoP-Ω is indeed faster for arithmetic and numeric type problems. As for the latter benefit, the current work hopefully provides groundwork for a prover that is suitable for the use in real-world applications after some additional iterations in the future. Before this can be done, however, a major issue with the prover which was discovered during testing needs to be resolved: the prover is not sound for certain typed arithmetic equalities, i.e., it can return false proofs for invalid formulas.

The thesis is structured as follows. In chapter 2, I introduce the theoretical foundations for logic, arithmetic, equalities and the underlying mathematical calculus. In chapter 3 I describe the two provers nanoCoP-Ω is based on, as well as the benchmarks used for its evaluation. I then describe how the proof procedure must be extended to handle the new challenges presented by arithmetic expressions and equalities (chapter 4). Chapter 5 describes the implementation of the new prover, and chapter 6 evaluates its performance. Chapter 7 outlines methods to re-establish soundness for the prover and describes additional optimizations and features nanoCoP-Ω could be extended by. Chapter 8 concludes this thesis and summarizes the outcome. In the Appendix I provide some additional lists. There is also an electronic appendix.

2 Mathematical Background

nanoCoP-Ω proves formulas in first-order logic with arithmetic and equalities. For solving arithmetic equalities, it uses an algorithm that employs Presburger arithmetic as a theoretical basis for interpreting numbers. This chapter contains the specification of first-order logic and describes Presburger arithmetic. Furthermore, it describes the connection calculus, which directly specifies how nanoCoP searches for proofs.

2.1 First-order logic

First-order logic contains the logical operators $\wedge$ (AND), $\vee$ (OR), $\neg$ (negation), $\Rightarrow$ (implication) and $\Leftrightarrow$ (if-and-only-if). In this thesis we only discuss the classical interpretation of these symbols. First-order logic furthermore contains the constant Boolean values 'true' and 'false', and there are variables, constants, and functions. Functions that return true or false are called *predicates*. Variables can be bound by the two quantifiers $\forall$ (for-all; universal quantification) and $\exists$ (there-exists; existential quantification). First-order logic only allows for quantification over variables, not over functions or expressions. This is what separates it from higher-order logic and allows it to be described by complete and sound algorithms. This is a necessary feature when building a computer program, as it allows us to build proof procedures that are guaranteed to terminate, and always find a proof for a given valid formula. Higher-order logic does not have this property and thus building an automated theorem prover for higher-order logic that is guaranteed to terminate during proof search is impossible. In the following the if-and-only-if operator is replaced by two implications, as this yields an equivalent expression. I also refer to the right side of an implication and to a logical formula that we want to prove as a *conjecture*. First-order logic originally does not include equality relations and arithmetic but can be extended by these to increase useability manifold. We use Presburger arithmetic in this thesis.

2.2 Arithmetic

Arithmetic refers to mathematical expressions with numbers. There have been several attempts at formalising arithmetic with varying degrees of success concerning decidability and consistency. Here we present Presburger arithmetic [23], which the Omega Test presented in chapter 4.2.3 uses as a basis for interpreting numbers. The axioms of Presburger arithmetic, quantified over all natural numbers, are as follows:

[1] $0 \neq x + 1$

[2] $x + 1 = y + 1 \ \rightarrow \ x = y$

[3] $x + 0 = x$

[4] $x + (y + 1) = (x + y) + 1$

[5] For all first order formulas $P(x)$ of at least one free variable x:
$P(0) \land \forall x. (P(x) \Rightarrow P(x + 1)) \Rightarrow \forall x. P(x)$

In Presburger arithmetic it is possible to define addition. Multiplication with constants can also be realised, and the relation '<' can be fully defined. Negative numbers can be defined via subtraction. It is not possible to define multiplication with variables, division or prime numbers. Due to this reduced expressiveness, Presburger arithmetic remains decidable, that is for every formula that can be formulated with Presburger arithmetic it is possible to determine with an algorithm whether the formula follows from the axioms.

2.3 Equalities

Equality relations have the following properties which need to be respected within the prover [1]. The equality relation '=' is reflexive, symmetrical, transitive, and possesses the substitution property: if a=b then also F(a)=F(b) for any expression F. Inequalities other than '≠' are converse and transitive. The substitution property for inequalities only applies to monotonic functions. We cannot reason over monotonic functions within first-order logic, so this property cannot be integrated into the prover.

2.4 Connection Calculus

The origins of automated theorem provers lie in mathematical calculi that specify sound and complete proof search procedures. These were originally developed by mathematicians and allowed them to reason over core aspects of logics, such as completeness. For instance, Vampire, a rather successful automated theorem prover, uses the superposition calculus [12]. Another calculus is the connection calculus. The prover nanoCoP, which we extend to create nanoCoP-Ω, is based on this calculus. To better understand the inner workings of nanoCoP, I will now explain the connection calculus. The calculus has not been extended to handle arithmetic and equality, which is something we will discuss in chapter 4.

Proofs for first-order logic are always structured in the same way. If we wish to prove an implication, we can assume the truth of the left side and prove the right side in further reasoning. Alternatively, we can disprove the left side. If, on the other hand, we assume the truth of the implication we can use that truth to assume the truth of the right side in further reasoning, provided we can prove the implication's left side. For conjunctions, the proof structure also has two formats, depending on whether we wish to prove the conjunction or if the conjunction can be assumed to be true. To prove the conjunction, we need to prove the left side and the right side independently from each other. In contrast, if the conjunction is assumed, we can use either side in reasoning.

Attempts to exploit these systematic similarities between proofs, inherent to the semantics of classical logic, have been undertaken for the purpose of creating algorithms capable of finding proofs reliably. One of these algorithms is the connection calculus. It has been around for more than 50 years [22], but nanoCoP is based on a more recent description by Otten [19].

2.4.1 Syntactic Trees

To apply the connection calculus, it is first necessary to convert the formula we wish to prove, a.k.a. the conjecture, into a matrix that represents it [18]. This is possible by exploiting the clearly defined way of proving the different logical expressions within first-order logic that we illustrated above. We start the conversion by traversing the syntax tree of a formula and assigning each node a polarity (i.e., F or T) and a rule type (i.e., α, β, γ, or ∂), as defined in

Table 1.

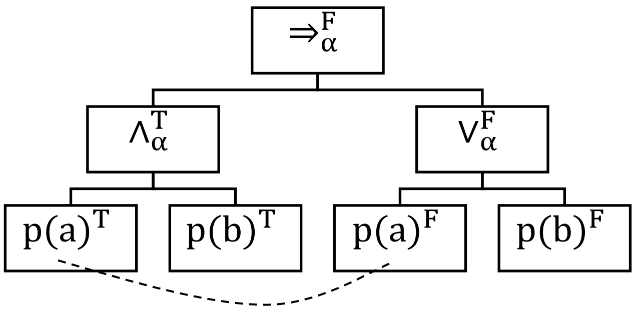
A node is either a logical operator or an atom. Atoms are the leaves of the syntax tree (terminal nodes). Atoms do not have a type. The polarity of a node depends on the parent node. The root operator is assigned the polarity F. This is equivalent to assigning F to the entire formula. An atom that has been assigned a polarity is called a *literal*. The type is determined by the node's polarity and the logical operator. Types tells us about the format the proof needs to have to prove the part of the formula defined by the operator: α corresponds to regular continuation, β corresponds to case distinction, γ corresponds to being allowed to instantiate the variable bound by the quantifier as we wish, and ∂ requires us to declare a new variable.

Table 1. Connection calculus rules for polarity and type of nodes

Operator + polarity	∧F	∧T	∨F	∨T	⇒F	⇒T	¬F	¬T	∀F	∀T	∃F	∃T
Type	β	α	α	β	α	β	α	α	δ	γ	γ	δ
Polarity left side child	F	T	F	T	T	F	T	F	F	T	F	T
Polarity right side child	F	T	F	T	F	T	-	-	-	-	-	-

One possible interpretation of the polarity is that F stands for 'to-be-proven' and T stands for 'can be assumed'. Thus, by assigning the polarity F to the root we mark the entire formula as 'to-be-proven', see Example 1. Finding the 'can be assumed' version of a 'to-be-proven' literal means we have proven the atom described by this literal. In other words, a pair of identical literals of inverse polarities is a proof for the corresponding atom. These pairs are called *connections*. In Example 1 a dashed line highlights a connection for the atom $\mathrm{p(a)}$.

Example 1



$(\mathrm{p(a)} \wedge \mathrm{p(b)}) \Rightarrow (\mathrm{p(a)} \vee \mathrm{p(b)})$ *Theorem*

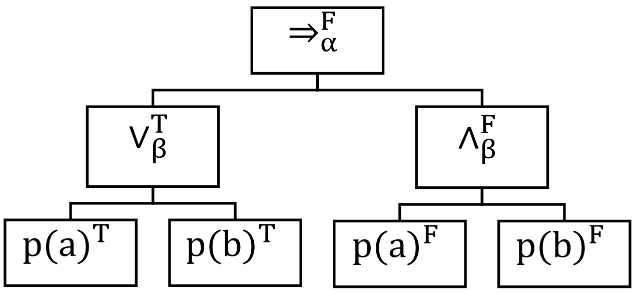
The formula is translated into the syntax tree on the left. A connection between $\mathrm{p(a)}^{T}$ and $\mathrm{p(a)}^{F}$ corresponds to using the left side of the conjunctive assumption to prove $\mathrm{p(a)}$.

The interpretation of pairs of identical literals with inverse polarities is different in nanoCoP. The conjecture is again assigned the polarity F. However, this stands for the truth value 'false'. Thus, literals with polarity T are 'true' and literals with polarity F are 'false'. To carry out a proof, we show that the assignment of the value 'false' to the conjecture is contradictory, thus yielding that the formula is true. Such a contradiction manifests itself in connections: if an atom is both true and false within a formula, we have a contradiction for this part of the formula.

We now discuss the different node types introduced above in greater detail. The interpretation of type α is that both the left and the right side of the operator can be used in the same proof. For β, on the other hand, a case distinction within the proof is necessary: in one case we use the left side and in the other case we use the right side. An example for the importance of distinguishing between type α and type β can be found when comparing Example 1 and Example 2. In Example 1 we only have α-type operators while Example 2 has the operators $\vee^{T}$ and $\wedge^{F}$, which are of type β (see

Table 1). We do have the same connections we had in Example 1, but these are not sufficient in proving all cases.

Example 2



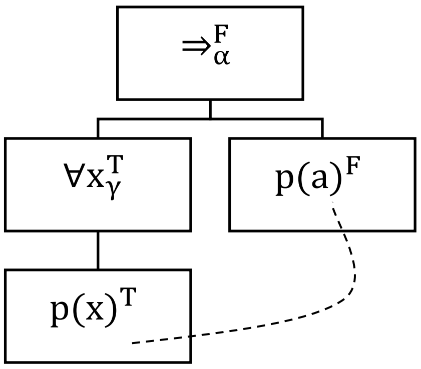
$(p(a) \vee p(b)) \Rightarrow (p(a) \wedge p(b))$

Non-Theorem

The formula is translated into the syntax tree on the left. A connection between $p(a)^{T}$ and $p(a)^{F}$ is possible, but the two β-type operators force two case distinctions for a total of four cases where we can only prove one case per distinction.

The types for the quantifiers work differently as they are used to define the nature of the variables they bind. We have γ-variables in the following situations. If we can assume a constraint for a universally quantified variable, we may insert any variable into the constraint and can be certain that the constraint will hold. Similarly, we may show that a constraint for an existentially quantified variable holds by showing that the constraint holds for any instantiation of this variable. Example 3 shows a syntax tree with a γ-operator that proves a theorem. The connection is again represented by a dashed line.

Example 3

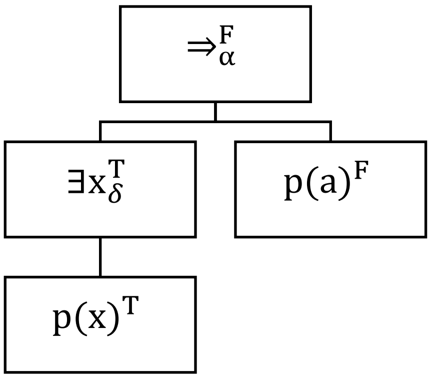


$(\forall x.\, p(x)) \Rightarrow p(a)$ *Theorem*

The formula is translated into the syntax tree on the left. A connection between $p(x)^{T}$ and $p(a)^{F}$ is possible because we can instantiate x with a.

Type δ indicates that the variable bound by the quantifier must be regarded as a new constant. In a natural proof we would declare the δ-variable as 'Let X by a new, unknown but invariant variable'. Type δ applies to the following two situations. First, when proving an attribute of a universally quantified variable we may not instantiate this variable with another variable that already exists but instead must create a new one. And second, if we can assume the existence of a variable which satisfies a certain constraint, we may not declare this variable to be the same variable as another without a reason. Example 4 shows a syntax tree with a ∂-operator. The purpose of γ- vs. δ-typed variables will be discussed in greater detail later on.

Example 4



$(\exists x.\, p(x)) \Rightarrow p(a)$ *Non-Theorem*

The formula is translated into the syntax tree on the left. A connection between $p(x)^{T}$ and $p(a)^{F}$ is not possible because we cannot be sure that it is safe to instantiate x with a. Instead, x must be declared as a new unknown variable.

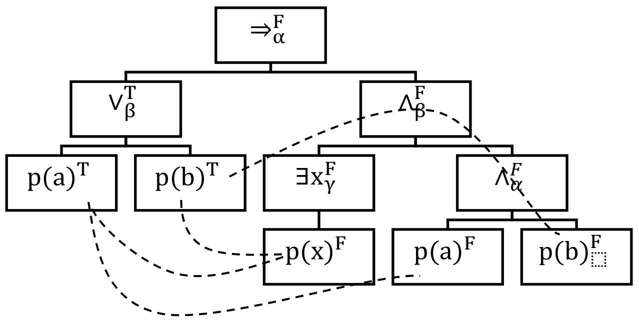
We have seen now how to build syntax trees with annotated types and polarities and how to use them to find proofs: the proofs were found by establishing connections between literals. The rest of the syntax tree was only partially relevant to the proof search. All that was required are two things: the literals with their polarities, and for any two literals the type of the operator that is their earliest common ancestor within the syntax tree. We say that two literals are in α- or β-relation if their earliest common ancestor is of the corresponding type. With this much in hand, we can construct a matrix.

2.4.2 Matrices

In contrast to a syntax tree, a matrix contains literals but not operators. Literals are positioned as follows: two literals are horizontal to each other if they are in α-relation, and vertical to each other if they are in β-relation. Syntactically, literals always occur in a *clause*. Matrices, including the top-level matrix describing the formula, consist of clauses. Clauses contain literals and/or further submatrices. The elements of the same clause are in β-relation, and thus written vertically. The elements of a matrix, in contrast, are in α-relation, and thus written horizontally. Example 5 illustrates this syntax: clauses are delimited by vertical lines and matrices by square brackets.

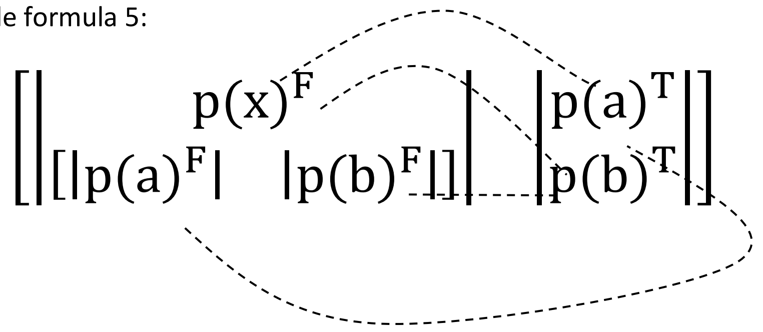
A maximal collection of literals in α-relation to each other is called a *path*. A path represents a particular set of case distinction decisions for all possible case distinctions within the proof. Within the matrix, a path traverses the matrix from left to right horizontally through every clause, while never selecting two literals that are in β-relation to each other. Thus, the number of paths through the matrix grows exponentially with the number of β-type logical operators that are in α-relation to each other. Consider Example 5. Here, the paths are as follows: $\{p(x)^F, p(a)^T\}$, $\{p(x)^F, p(b)^T\}$, $\{p(a)^F, p(b)^F, p(a)^T\}$ and $\{p(a)^F, p(b)^F, p(a)^T\}$. All connections are marked with dashed lines.

Example 5



$(p(a) \vee p(b)) \Rightarrow$ *Theorem*
$(\exists x. p(x) \wedge (p(a) \vee p(b)))$
The formula is translated into the syntax tree on the left. The proof requires four connections. The matrix including connections is displayed below.

Matrix for example formula 5:



Note that a connection need not be between two complementary syntactically identical literals if one or both contain variables. The two literals need only be unifiable. This means that it must be possible for them to become the same under the substitution σ, which replaces variables with other variables or constants. This substitution is the same during the entire proof. In Example 5 and also in Example 3 further above, the substitution replaces x with a in order to connect $p(x)^F$ and $p(a)^T$. Syntactically this is denoted as $\sigma = [a/x]$. For Example 5 we also replace x with b, this is explained further below.

There are several constraints on the substitution that need to be checked by the prover, whether this be a person or the computer. For instance, the necessity of distinguishing between γ- and δ-typed variables mentioned earlier now becomes apparent: γ-typed variables cannot be instantiated with δ-typed variables declared 'after' them within the formula. For example, the calculus uses this to disprove the non-theorem $\exists y. \forall x. (p(x) \Rightarrow p(y))$, where the ∂-variable x is below the γ-variable y within the syntax tree. In the tree this illegal situation would be represented by the γ-type quantifier node being an ancestor of the δ-type quantifier node. This constraint can automatically be enforced by transforming δ-typed variables into Skolem form: the ∂-variables are transformed into newly declared functions that have the γ-variables they depend on, i.e., that are declared above them in the syntax tree, as parameters. Now we perform an occurs-check: we test whether the variable we unify with an expression occurs within this expression. This suffices to enforce the constraint.

There are many algorithms that construct the necessary substitution, such as for example [9], [13], and [26], and this topic is an important part of implementing an automated proof procedure efficiently. We skip this part by letting Prolog, the programming language nanoCoP-Ω is written in, handle it and merely tell Prolog to do an occurs-checks when unifying.

Some clauses may be used multiple times by the proof. This commonly occurs when a universally quantified expression holds. Another example are axioms such as the substitutive property of equality. Example 5 uses the clause on the left multiple times for connecting both $p(a)^F$ and $p(b)^F$ with $p(x)^T$. The substitution thus becomes $\sigma = [a/x_1, b/x_2]$. These multi-use clauses have a multiplicity higher than one. Each clause has a multiplicity that may be arbitrarily high, as long as it is not infinity.

Next, recall that under the first interpretation given earlier, a connection means that we have found the necessary atom for a proof and thus the formula is valid for the particular case distinction at issue, i.e., for this particular path. Also recall, that by the interpretation used in nanoCoP [18], a connection means that the matrix is contradictory for this particular path. Thus, the formula is true, or the matrix is contradictory, if every possible path through the matrix contains a connection. Each path corresponds to one collection of case distinctions. We can now fully characterise the constraints needed to prove a formula with a matrix. Directly quoted from [18], page 228:

[Definition 1:] Matrix Characterization: A matrix is classically valid iff there exists a multiplicity μ, a term substitution σ and a set of complementary connections S, such that every path through M^μ contains a σ-complementary connection $\{L1, L2\} \in S$.

2.4.3 The Calculus

In the last chapters we saw how to prove a matrix by analysing all the paths it contains. However, instead of considering every single path, the non-clausal connection calculus that nanoCoP is based on groups paths together whenever possible. Since a given connection is likely to be part of multiple paths it makes more sense to search for connections. Whenever we find a connection, we can be certain that all paths that are extensions of this connection are complementary and thus proven. Hence, we need not consider paths containing only literals in α-relation to the literals of the connection. These paths contain the connection. Instead, paths not containing the connection, i.e., paths going through the literals in β-relation to the connection's literals, need to be considered. This observation leads to the core idea of the non-clausal connection calculus: we search for connections, starting with a particular clause, and attempt to find connections for all the clause's different literals. Once a connection is found all extending (a.k.a. α-) paths can be considered proven, while all alternative (a.k.a. β-) paths still need to be considered. With this much in hand, the calculus can be defined. The definition is again taken from [18], page 223. ε is the empty clause or the empty path.

> [Definition 2:] Non-Clausal Connection Calculus: The axiom and the rules of the non-clausal connection calculus are given [below]. The words of the calculus are tuples 'C, M, Path', where M is a matrix, C is a clause or ε and Path is a set of literals or ε. C is called the subgoal clause. C_1, C_2 and C_3 are clauses, σ is a term substitution, and $\{L_1, L_2\}$ is a σ-complementary connection. The substitution σ is rigid, i.e. it is applied to the whole derivation.

A derivation for C, M, Path in this calculus is a proof if all leaves are axioms. The path denotes the to-be-proven path, thus containing only literals that we have not been able to find connections for yet.

In the following the rules of the calculus are explained, as they will be extended upon and modified during the work done in this thesis. Each rule has a name, a requirement above the line, a conclusion below the line, and additional restrictions on the right side.

The axiom rule defines what a solved proof branch looks like. As the currently to-be-proven objective is denoted by the clause C we are done if C is empty. All paths through the empty clause (there are none) have a complementary connection.

1. $$Axiom\ (A) \qquad \frac{}{\{\}, \mathrm{M}, \mathrm{Path}}$$

The start rule initialises a proof by selecting a clause from the matrix. This is the first clause that the algorithm using the calculus attempts to solve. The choice of this clause can greatly influence the success of the procedure and it is discussed in chapters 4.2.5, 5.3 and 5.2.6. The proof copies C_2 as a way of introducing and allowing for multiplicity and for the reusability of clauses.

2. $$Start(S) \qquad \frac{\mathrm{C}_2, \mathrm{M}, \{\}}{\varepsilon, \mathrm{M}, \varepsilon} \qquad \text{and } \mathrm{C}_2 \text{ is a copy of } \mathrm{C}_1 \epsilon \mathrm{M}$$

The extension rule describes the main strategy to find connections.

3. $$Extension(E) \qquad \frac{\mathrm{C}_3, \mathrm{M}[\mathrm{C}_1 \backslash \mathrm{C}_2], \mathrm{Path} \cup \mathrm{L}_1 \qquad \mathrm{C}, \mathrm{M}, \mathrm{Path}}{\mathrm{C} \cup \mathrm{L}_1, \mathrm{M}, \mathrm{Path}}$$

$$\text{and } \mathrm{C}_3 \coloneqq \beta - \text{clause}_{\mathrm{L}_2}(\mathrm{C}_2), \mathrm{C}_2 \text{ is a copy of } \mathrm{C}_1, \mathrm{C}_1 \text{ is an extension clause of M with regard to } \mathrm{Path} \cup \{\mathrm{L}_1\}, \mathrm{C}_2 \text{ contains } \mathrm{L}_2 \text{ with } \sigma(\mathrm{L}_1) = \sigma(\overline{\mathrm{L}_2})$$

The rule selects a complementary literal L_2 to the to-be-proven literal L_1, where L_1 is from the subgoal clause $C \cup L_1$, and L_2 is from another clause C_1 within the matrix M. C_1 being an extension clause of M with regard to Path $\cup$ L_1 means that the literals within C_1 are all in α-relation to the literals of the Path and the parent clause is on the path if it exists, or that C_1 is on the Path. We replace C_1 in M with C_2 to allow for correct multiplicity again. C_3 being the β-clause$_{L2}$ to C_2 means that it contains only literals in β-relation to the complementary connection literal L_2, so it contains only literals whose outgoing paths still need to be proven. The last line of the rule enforces the use of the global substitution σ. Thus, the left side of the rule describes how to solve a particular literal from the subgoal clause. The right side merely states that the rest of the subgoal clause also still needs to be proven. The rule is discussed in chapters 4.2.1, 4.2.4, 5.2.1 and 5.3.

The reduction rule ensures that when searching for a complementary literal to the current one we consider connections to literals from the current path, i.e., backwards. These might otherwise not be found, as the extension clause described above only searches for connections to literals within another clause. If a backwards connection can be found we need not prove anything else, as the β-literals to the connected literal have already been handled by the extension rule.

4. $$Reduction(R) \qquad \frac{\mathrm{C, M, Path} \cup \{\mathrm{L_2}\}}{\mathrm{C} \cup \{\mathrm{L_1}\}, \mathrm{M, Path} \cup \{\mathrm{L_2}\}} \qquad \text{with } \sigma(\mathrm{L_1}) = \sigma(\overline{\mathrm{L_2}})$$

The decomposition rule states that we can prove a matrix by proving one of its α-related clauses. It is a rule that deals purely with syntax and will not be discussed further.

5. $$Decomposition(D) \qquad \frac{\mathrm{C} \cup \mathrm{C_1}, \mathrm{M, Path}}{\mathrm{C} \cup \{\mathrm{M_1}\}, \mathrm{M, Path}} \qquad \text{with } \mathrm{C_1} \in \mathrm{M_1}$$

The lemma 'rule' is not a real rule, but an optimization commonly used in connection-based provers. It states that a literal that has already been proven need not be proven again. To implement it we need to collect the literals that we have already proven during the proof search in an additional list, and check that the current literal is not contained in the list.

3 Previous Work on Connection Calculus Provers

In this chapter I will present previous work that has made the idea of nanoCoP-Ω possible. This includes work on the provers nanoCoP and leanCoP-Ω, of which nanoCoP-Ω is a fusion, the Omega Test, which is used for arithmetic reasoning, and the TPTP library, which is used to evaluate the final result.

3.1 nanoCoP and leanCoP

nanoCoP is a natural non-clausal form automated connection prover. It implements the non-clausal connection calculus presented in chapter 2.4 in the programming language Prolog. Prolog stands for *programming in logic* and was chosen by the author of nanoCoP, Jens Otten, for the possibility of programming very close to the mathematical specification of the calculus. This choice allowed Otten to implement the first version of nanoCoP with only 42 lines of Prolog code [19]. [21] presents version 2.0 along with versions for intuitionistic and modal logics.

The main difference between leanCoP and nanoCoP can already be read out of their names. nanoCoP is a shorthand for *naturalistic non-clausal connection prover*. This means that it uses the non-clausal connection calculus described in chapter 2.4. leanCoP, on the other hand, transforms the matrix into definitional clausal form, disjunctive normal form, or a mix of the two where the axioms are in disjunctive normal form and the conjecture is in definitional normal form. This means that the matrices used in leanCoP are simple: each clause contains only literals instead of possible further submatrices. leanCoP transforms the matrix by using logic rules such as De Morgan's laws or by pushing the quantifiers to the very top of the formula. This causes the matrices to be very flat, but also very wide. Definitional clausal form differs from disjunctive normal form by the way definitional clausal form unfolds conjunctions of further logical expressions.

The normal forms used in leanCoP ensure that each matrix has a simple form the computer can handle easily. However, these forms have the downside of increasing the size of the matrix greatly, worst-case exponentially, because copying formulas can be necessary for instance when applying the distributivity attribute of conjunction. The translation into normal form thus introduces a computational overhead. Additionally, extending the connection calculus to intuitional logic becomes impossible when using definitional normal form [18].

A further concern with the translation into normal form is that the resulting proof becomes unreadable for humans. Yet, if an automated proof system is intended to be used in research it is vital that the returned proof is readable by a human. Research requires understanding, and the knowledge about the validity of a formula is rarely enough to yield understanding, especially since follow-up questions regarding the deeper nature of the research topic become harder to formulate. Due to the transformations done by leanCoP, the structure of the original formula is not discernible from the matrix used in the proof. nanoCoP and, by extension, nanoCoP-Ω, in contrast, support the option of returning a human-readable proof, provided the user knows connection calculus: the matrix structure is reminiscent of the formula's structure. The proof within the connection calculus can even be translated back into a completely human-readable version.

A further, syntactic difference between nanoCoP and leanCoP is that the latter handles ∂-variables differently than the former. This creates an issue described in chapter 5.2.4, which can easily be fixed.

leanCoP 2.0 and nanoCoP 2.0 both come with an optional feature called *restricted backtracking*. nanoCoP-Ω also supports restricted backtracking. The feature was presented by Otten in [17] and its core idea is to remove non-essential backtracking, which is not necessary for finding proofs in many cases and can be detrimental to the runtime of the program. Non-essential backtracking is done whenever we have multiple ways of solving a singular literal during the proof procedure, and the first method of solving the literal ultimately does not lead to a successful proof because of the other literals contained within its clause. When eliminating non-essential backtracking we do not retry proving a literal's β-neighbours after the proof search for this literal has failed. Instead, the entire clause is discarded, as further attempts at proving it would only change the global substitution σ. Additionally, if we were to retry proving one of the literal's β-neighbours we may succeed by, for example, establishing a connection to a different clause than before. This hardly affects the proof for the current literal: only the substitution (and arithmetic constraints, as explained in chapter 4.2.2) could be different. Thus, instead of proving the problematic literal's clause, the prover takes a large step back and draws a connection to a different literal than the problematic one if possible. Using restricted backtracking removes completeness from the prover but often results in a large time save and potentially allows solving problems that would otherwise have been locked behind a time-out [17].

3.2 leanCoP-Ω

leanCoP-Ω is a version of the compact automated theorem prover leanCoP 2.0 extended with the methods of dealing with arithmetic expressions and equalities as discussed in chapter 4. It utilises the Omega Library [24]. The integration of the Omega Test into leanCoP was originally developed by Otten, Trölenberg and Raths [15], updated by Behrens [5] and rewritten again by Münch [14].

The Omega Test can determine whether there exists a solution for an arbitrary set of linear equalities and inequalities (henceforth *LSE*). The test was first presented in [25]. It is based on an extension of Fourier-Motzkin variable elimination. In theory it has a worst-case exponential complexity, but Pugh shows that on average we can expect polynomial time complexity. While the original Omega Test is merely an algorithm to test the solvability of a set of linear equations, the Omega Project developed by the Omega Project Team [24] is a program collection written mainly[2] in C. This project includes the Omega Library, which uses Presburger arithmetic for extended reasoning over LSE. The Omega Library is capable of more than the Omega Test. The library is a useful tool for leanCoP-Ω and nanoCoP-Ω. Crucially, the library is restricted the same way Presburger arithmetic is restricted. For instance, it is not possible to define division or prime numbers, and it is not possible to reason over arithmetic expressions that include predicates. The Omega Library extends Presburger arithmetic by subtraction and realises multiplication with repeated addition while still retaining completeness and decidability, two properties which are very important for automated proving.

3.3 TFA and TPTP Benchmarks

The TPTP (*Thousands of Problems for Theorem Provers*) benchmark [30] is a benchmark specifically designed for automated theorem proving. It contains thousands of problems that are used to evaluate an automated theorem prover such as nanoCoP-Ω. As a benchmark it intends to provide an unambiguous reference and good indicator of the quality of new automated theorem provers. It

[2] https://github.com/davewathaverford/the-omega-project/

provides additional utilities in the form of background materials, guidelines, and format converters for different existing automated theorem provers. Note, however, that nanoCoP-Ω does not need a converter because it can read the TPTP2 format, which is the format the problems are stored in.

The problems in the TPTP benchmark stem from multiple domains from logic, mathematics, computer science, science and engineering, social sciences, arts and humanities, and others. The problems are formulated in first-order logic, conjunctive normal form, typed first-order logic or typed higher-order logic. The typed first- and higher-order logic are also subdivided into monomorphic and polymorphic logics. nanoCoP-Ω only supports typed first-order logic and first-order logic, although its reasoning over types other than integers is limited to testing if the types match during unification.

In addition to the TPTP benchmark v8.0.0 I used the TFA benchmark in this thesis, which is an extract of the ARI domain of the TPTP benchmark v5.0.0 used by Kai Münch in his bachelor thesis on leanCoP-Ω [14]. I used this benchmark for initial testing of nanoCoP-Ω and for discerning the quality of the different optimization settings discussed in chapter 5.3. Whenever I use the name of a problem from the TFA benchmark in this thesis, its unique TPTP identifier is written behind the name in brackets.

4 Extending the Connection Calculus by Arithmetic and Equalities

As already mentioned, arithmetic equalities are not part of the original connection calculus. Thus, it might not be completely clear how these can be integrated into a connection-based theorem prover. The procedure is explained in this chapter.

When reading the target formula, nanoCoP-Ω regards equalities as regular atoms and they thus end up being literals within the matrix. In the following, I refer to literals containing any type of equality relation and their negated forms as *e-literals* or as *constraints*. This also includes inequality and the less-than relation. Some examples are $f(3)=f(x)^{T}$, $27+x<6*y^{F}$ or $y\neq 4^{F}$. E-literals are treated like regular literals by all parts of the proof procedure until we reach the step of trying to prove an e-literal. I will present the different methods nanoCoP-Ω uses to prove an e-literal in chapter 4.2.1 to 4.2.5, which each describe one new method, or the modification of an existing method. Before we can understand these methods, however, it is necessary to describe the unification procedure we use whenever literals contain arithmetic expressions. This is what we discuss first, in chapter 4.1.

4.1 Unification in a Context with Arithmetic Expressions

The existence of arithmetic poses a challenge for unifiability tests. Prolog's in-built predicate `unify_with_occurs_check/2` implements regular unification while avoiding the creation of cyclic terms[3], i.e., expressions that have infinite length. The fact that this degree of unification can already be handled by Prolog and does not need to be addressed in the automated theorem prover at

[3] https://www.swi-prolog.org/pldoc/doc_for?object=unify_with_occurs_check/2

all is one of the main benefits of writing this kind of program in Prolog. However, since Prolog's unification does not handle arithmetic expressions, we need to add support for this ourselves.

It is possible to unify two arithmetic expressions by the following rules. If we have addition or multiplication of constants, we calculate their result. If we have a variable as one of the expressions that we wish to unify and an arithmetic expression as the other, we can unify the two expressions by assigning the arithmetic expression to the variable. Whenever there are two complex arithmetic expressions we need to unify, we instead store an equation that we remember for the entire proof procedure. We calculate the equation to be stored by subtracting one of the arithmetic expressions from the other so that one side of the equation is 0. Since equations within the Omega Test are stored in this format, we save some computation time by doing this. Whenever there is a non-arithmetic operator within either of the two to-be-unified expressions, such as an equality relation or a regular function, we must continue our unification procedure for the parameters of the operator and store all necessary equations. The operator must be present at the same location on both sides. This may seem counterintuitive as it reduces the power of the unification algorithm when we are multiplying two things other than constants. However, we are forced to do this because it is not possible to calculate the product of two variables (or of two expressions containing variables): Presburger arithmetic only supports multiplication with constants. The method of continuing the unification procedure for both parameters allows us to at least unify `A*B` and `C*3` for the rare situation that `B=3`. Overall, we see that nanoCoP-Ω is very limited in the arithmetic expressions it can unify as soon as the expressions contain multiplication of variables.

4.2 New and Modified Methods

4.2.1 Connections

Using connections is the first of the four methods that nanoCoP-Ω uses to prove an e-literal. It is valid to interpret equalities as literals that can be connected to complementary literals. This is because any type of connection is always a valid proof. In theory we could prove a formula by establishing enough connections between composite parts of the formula. This is not done in the connection calculus though, as pattern matching across large composite formulas becomes computationally inefficient very quickly. Since a re-interpretation of connections especially for e-literals is not necessary, the regular extension rule can be used to reason for e-literals. The same goes for the lemma and reduction rules.

4.2.2 Interpretation as a Constraint

Finding a connection is not the only method of proving an e-literal, because e-literals are not statements about the truth value of atoms, unlike normal literals. E-literals are restrictions on the values of integer variables. Therefore, it is sufficient to ensure that the constraint represented by an e-literal holds everywhere where it needs to. Specifically, the e-literal/constraint affects the global substitution developed in the proof. For example, if a constraint specifies that `A<B` then the variables `A` and `B` are not unifiable anymore anywhere in the proof, unless we can prove the constraint differently, because a unification of `A` and `B` requires the equality `A=B`. To ensure the satisfaction of all constraints, the automated prover stores all equations necessary for unification, as described in chapter 4.1, as well as all e-literals on the path, and then passes the entire set of equations to the Omega Test. If the Omega Test confirms that the set of equations is solvable, we know that this constraint is satisfied at this stage of the proof search. The exact details of what the set of equations passed to the Omega Test needs to look like is described in chapter 5.2.3.

4.2.3 Self-satisfying Equalities

Certain e-literals are inherently valid due to arithmetic. The literal `1<2` does not need a connection to be proven. We can use rudimentary arithmetic reasoning over constants to prove these simple e-literals. As soon as variables are involved, an evaluation with the Omega Test is necessary, as we need to respect the global substitution σ and the constraints specified by other e-literals. Additionally, as equations with variables are rarely self-satisfying it makes more sense to pass equalities with variables to the Omega Test as usual to give additional information about the variables.

4.2.4 Deep Omega Extension

Within leanCoP-Ω there is a special extension rule that is not part of the original extension calculus and for which I was not able to find any documentation. The special extension rule has been integrated into nanoCoP-Ω successfully, although its usage is only required for one singular problem of the TFA benchmark. Within leanCoP, the rule is usually locked behind the setting `eq(2)`. It is activated if no other method of proof was successful for the to-be-proven e-literal. The rule tells the prover to extend the proof to another e-literal anywhere within the matrix. Recall that the regular extension rule would only allow the prover to draw a connection to a complementary literal. When the rule is used, the originally to-be-proven e-literal is added to the path, and the e-literal we have extended to is marked as still to be proven. In the regular extension rule, the literal we have extended to would not need to be proven anymore. I will refer to this new variant of the extension rule as the *deep omega extension rule*.

While the deep omega extension rule is not part of the original calculus, it is easy to ascertain its correctness. Recall that proving an e-literal can be done by one of three methods other than the deep omega extension rule. Proof by connection is completely disjoint from deep omega extension: either one or the other happens. The test for self-satisfaction of the e-literal is run before attempting to use the deep omega extension rule. Self-satisfying literals do not impact other e-literals of the matrix and thus the self-satisfaction test is also disjoint from the deep omega extension rule. This leaves the interpretation of the e-literal as a constraint as the last method that needs to be considered to evaluate the correctness of the deep omega extension rule. In fact, the deep omega rule relies on the interpretation method for proving the current e-literal: that literal is written to the path by the deep omega extension rule. When called, the Omega Test considers all e-literals from the current path and thus also the to-be-proven literal (see chapter 5.2.3). So, the to-be-proven e-literal is evaluated and hopefully proven later on in the proof procedure. In other words, the deep omega rule postpones the proof of the current e-literal.

Let us illustrate the deep omega extension rule with an example. As already mentioned, the rule was only necessary for solving a single problem from the TFA benchmark: Problem TFA198=1 (ARI198=1). The problem is displayed below:

```
tff(less_successor,conjecture,(
    ! [X: $int,Z: $int] :
      ( $less(Z,$sum(X,1))
     => $lesseq(Z,X) ) )).
```

The goal is to show that if a number `Z` is smaller than another number's (`X`) successor, `Z` is smaller or equal to `X`. This is translated into two clauses with one literal each by nanoCoP-Ω:

```
(4^_12222)^[]:[
      -((2^[])^t($int)<((1^[])^t($int)+1^t($int))^t($int))],
(6^_12222)^[]:[-((1^[])^t($int)<(2^[])^t($int))]
```

At this point it is only important to consider that the resulting matrix contains two negative literals in α-relation to each other. Thus, the matrix's evaluation does not lead to a valid proof with the regular connection calculus rules. The two equalities themselves are not self-satisfying, or at least the prover is not capable of resolving the addition in the predicate that tests for self-satisfying literals. Thus, we see that a deep omega extension from one literal to the other is necessary for the proof of this theorem. Once both literals are on the path, the prover passes them connected by an OR-operator to the Omega Test. This constitutes an LSE that the Omega Test marks as satisfiable, thus proving the theorem. Problems ARI082=1 and ARI195=1 can also be solved using this rule, but its use is not mandatory, and a proof can also be found using other methods.

4.2.5 A New Start Rule

The task of a start rule is to select a clause from which to begin the proof. In the classical connection calculus, a valid optimization is to restrict the choice of a start clause to only positive clauses. This rule is not universally applicable when solving arithmetic problems.

If a matrix lacks positive clauses, i.e., clauses with only positive literals, it is not valid. The absence of a positive clause means that there is a path through the matrix that contains only negative literals. This path cannot contain any complementary connections, thus rendering the matrix invalid. For this reason, it is usually a valid optimization to restrict the choice of the start clause to positive clauses. This was done in nanoCoP 2.0 and led to false negatives in the first iterations of nanoCoP-Ω. The reason was that negative e-literals are interpreted as negative literals due to the notation to be explained in chapter 5.1.3: they are marked with a '−' sign, just like negative literals. Selected formulas from the TFA benchmark thus did not contain any positive clauses. This was incorrect because the negative literals were just e-literals, which can be proven by methods other than complementary connections.

In leanCoP-Ω this problem was handled by interpreting all e-literals as positive when checking for positive clauses. In nanoCoP-Ω this is not done when beginning the proof search. This has the advantage that regular positive clauses are chosen as start clauses over clauses that are only positive if we disregard e-literals: regular positive clauses are expected by me to be more likely to lead to a successful proof, because conjectures more often contain to-be-proven predicates with constraints on variables than being purely arithmetic. This is a heuristic decision. In any case, I believe this decision not to be detrimental, because the test for pure positivity of a potential start clause is fast. To still be able to solve the aforementioned formulas from the TFA benchmark, the prover restarts the proof after the first failed attempt by calling a new start rule: the new rule is the same as the original start rule except that it interprets e-literals as positive.

5 Implementation

This chapter describes the implementation of nanoCoP-Ω. The chapter strongly refers to the new methods of proving e-literals described in the last chapter. It also discusses differences in implementation to leanCoP-Ω and the issues that arose from these differences.

nanoCoP-Ω is based on nanoCoP 2.0 with extensions taken from leanCoP-Ω, which were adjusted to the slightly different syntax. Due to substantial similarities between the two systems the initial transfer of arithmetic reasoning into nanoCoP 2.0 was possible without major changes. The file reading procedure, in contrast, required an extensive rewrite, as nanoCoP 2.0 could not read typed formulas and equalities. I have also attempted to integrate into the file reading process some soft constraints on problems in the TPTP syntax which were only introduced in newer versions of the TPTP library.

nanoCoP-Ω can read problems in TPTP syntax or in its internal syntax. I recommend looking at this website[4] for an extensive BNF documentation. Note that the polarity of literals as discussed in chapter 2.4 is marked by nanoCoP-Ω as follows: F (false/to-be-proven) is not marked, while T (true/can-be-

[4] https://tptp.org/TPTP/SyntaxBNF.html

assumed) is marked with a ‘−’. This choice is arbitrary, as only the search for complementary connections is relevant.

The prover works as follows. It reads the target file, converts it into an internal format as specified by the file `nanocop20Omega_tptp2.pl` and then calculates the matrix representation with the predicate `bmatrix/3`. Then it begins the proof search by calling the predicate `prove/4`, which contains the start rules. The file `nanocop20Omega_proof.pl` specifies how the prover will print the proof to the output stream. The entire process can be started by calling the predicate `nanocop_main(File, Settings, Result)` from the file `nanoCoP20Omega_main.pl`, but I recommend having a look at the shell script instead.

nanoCoP-Ω currently only supports SWI-Prolog[5].

5.1 Basic Implementation Properties

5.1.1 Axioms, Conjectures and Types

Within the TPTP benchmark used here every formula is marked as an axiom, a type specification or a conjecture. nanoCoP-Ω needs to respect these roles. It collects the conjecture separate from the axioms. The final formula it proves after having read the input file is of the shape `axioms => conjecture`, where the axioms are connected conjunctively. If there are no axioms and if the problem contains any equalities nanoCoP-Ω uses the equality axioms as the axioms. Otherwise it proves the conjecture as is. The TPTP syntax specifies additional roles, which, however, are not used within the benchmark used to evaluate nanoCoP-Ω. Therefore, the prover only differentiates between the roles conjecture, type and others.

Type specifications are collected while reading the input file. The types supported by nanoCoP-Ω are integer, Boolean, individual, and new, formula-specified types. Once the first pass is done, nanoCoP-Ω does a second pass across the whole formula to write down each expression’s type. This includes marking predicates as Booleans, and constants and variables as individuals. At this stage, the prover has collected the type for each variable. It also writes down the result types for all pre-defined operators, for instance all addition results are marked as integer types. nanoCoP-Ω does not do anything with types other than test that they match whenever unifying two expressions.

5.1.2 Equality Properties

The equality properties reflexivity, symmetry, transitivity, and substitution are added as axioms to the formula as soon as the formula has been transformed into the internal syntax. For the substitution property it is necessary for Prolog to discern all predicates and functions contained within the formula and then add the substitution rule for each. This also ensures that the property is added for the <

[5] https://www.swi-prolog.org/pldoc/doc_for?object=manual

relation. The restricted substitution properties for inequalities are not added, as nanoCoP-Ω cannot reason about monotonic functions due to first-order logic.

5.1.3 Presentation of Equalities

Equalities are represented very similarly to regular literals in nanoCoP-Ω. Equalities can be presented the exact same way as literals but with an equal or smaller-than sign inside the e-literal. nanoCoP-Ω does not use the not-equal, greater, greater-or-equal, or less-or-equal signs. Instead, we represent these variants via transformation, that is by switching the left and right side of the equation literals and/or negating the literal with the '−' sign used for regular negative literals accordingly. The use of this sign is responsible for the misinterpretation of inequalities as negative literals as described in chapter 4.2.5. The representation using the equal, smaller-than and '−' signs allows the usual unification procedure for literals with their negated counterparts to also work for e-literals without any further adjustments being necessary. Thus, additional lemma, reduction or extension rules are not necessary. Furthermore, Prolog only needs to test for one of the following shapes to recognize an e-literal: `(_=_)`, `-(_=_)`, `(_<_)` or `-(_<_)`. The conversions to one of these four patterns for example from a greater-than relation happens within the predicate `op_tptp2/4` used in the predicate `nanocop_tptp2`.

5.1.4 Unification

Unification of expressions for the purpose of creating complementary connections and satisfying other proof rules is realised by the predicate `unify_with_arith(A,B,Equalities,Set)`. This predicate implements the ideas discussed in chapter 4.1. It unifies with occurs-check where possible, evaluates constant arithmetic expressions, translates necessary equations into linear arithmetic equalities, respects types, and returns all necessary equations to be stored by the proof procedure. It is also capable of handling functions and predicates, as long as they are the same for both unification partners. It interprets equality and less-than as functions to handle them correctly. The stored equalities are later passed to the `omega/1` predicate. An example for a necessary equality is the following: To unify $-((3*A)+(5*B)=(3*A)+(5*B))$ and $-((3*A)\ +(5*B)\ =23)$ the equality $-23+(3*A)\ +(5*B)\ =0$ is required. The second equality is the negated form of the currently to-be-proven e-literal during an actual proof step for a TPTP problem, in which we use the extension rule.

5.2 Implementation of New and Modified Methods

This chapter discusses the implementation of the ideas discussed in chapter 4.2.

5.2.1 The `prove` Predicate

The actual proving procedure of the nanoCoP-Ω prover takes place within the predicate `prove([Lit|Cla],MI,Path,PI,PathLim,Lem,Set,Proof,EqIn,EqOut).` In this chapter I will explain the meaning of the different parameters and in the following chapters I describe how this predicate implements the various rules of the connection calculus.

The first parameter contains the to-be-proven clause, which in turn contains the to-be-proven literal. `MI` is the matrix in which we have replaced the clauses already connected to by new copies, thus ensuring that it is possible to use the same clause multiple times. This clause duplication happens within the predicate `prove_ec`, which constructs the clause that still needs to be proven within the regular and deep omega extension rules.

`Path` is the current path to be proven. It is used to ensure that the prover does not extend to a literal already on the path and it is used by the Omega Test as described in chapter 5.2.3. `PI` is a version of the path that stores only the identification numbers of the path's literals. It is used to store these identifiers, as well as to grasp the depth of the current path. `PathLim` is used for iterative deepening. Iterative deepening ensures completeness of the automated prover despite Prolog's depth-first search approach. It works as follows: The nanoCoP-Ω theorem prover lets the current `prove` call fail whenever an extension to a literal with free variables would cause the path's depth to exceed the current `PathLim`. A clause within the start rule restarts the proof search with a `PathLim` increased by one if and only if the proof search failed due to exceeding the limit set by the previous `PathLim`. This ensures that Prolog does not get lost in deadlock loops because it is forced to try the different connections once it reaches the depth specified by `PathLim`. If the proof search failed for a different reason than reaching `PathLim`, it is safe to assume that the problem is a non-theorem. This version of iterative deepening was introduced in leanCoP 2.0 [16].

`Lem`, short for lemmata, contains all literals that have already been proven. nanoCoP-Ω tests if the to-be-proven literal is contained in this list as a way of ensuring that it does not prove the same literal twice. `Set` contains the current settings used by the prover. `Proof` is the returned proof, which is later converted to the form specified by the user. The user can ask for a compact, readable, leantptp or connection proof to be printed by the prover by asserting `proof(Form)`[6] in Prolog for one of these. `EqIn` and `EqOut` are used to receive and pass on equalities from and to the arithmetic unification procedure and the Omega Test. They are used to continuously update the constraints for the unification and for the Omega Test.

The prove predicate contains the different methods for proving a literal, which are discussed in the next chapters. The priorities between the methods are defined by the order they appear in Prolog code. They are connected by disjunctions, which ensures that each literal is proven by only one of the existing methods. Additionally, this ensures that nanoCoP-Ω only tries a different rule if the previous

[6] In Prolog: `?- assert(proof(FORM)).`

rule fails for all possible variable assignments. The lemma rule has the highest priority. Next is the reduction rule, then the extension rule and finally the rules for handling equation literals. If the to-be-proven literal is an e-literal, the prover first attempts to apply the `leanari` predicate (see chapter 5.2.2). If that fails, it continues with an Omega Test for constraint satisfaction (see chapter 5.2.3) and then a deep omega extension (see chapter 5.2.5). If one of the methods succeeds, we call the `prove/10` predicate on the rest of the current clause.

5.2.2 Leanari

As discussed in chapter 4.2.3 some basic arithmetic reasoning not using the Omega Test might be worth integrating into the prover. As the Omega Test is called with a potentially very large LSE it is worth attempting to solve simple self-fulfilling e-literals with Prolog first. The predicate `leanari/1` does this and can solve only e-literals by directly comparing two integer numbers with no additional operators involved. It is used by eight different problems within the TFA benchmark (see chapter 6.3).

5.2.3 The Omega Test

The Omega Test is implemented by the predicate `omega/1` within the prove predicate, which calls the predicate `omega_check/1` defined within the file `nanocop20Omega_omega_swi.pl`. The predicate is passed the LSE and transforms it into a form readable by the Omega Library. The `omega_check/1` function was implemented by Kai Münch, for details see [14]. The constraints passed to the Omega Test are collected during the proof search. As mentioned in chapter 5.1.4, the most regular calls to the Omega Test occur when a unification occurs within the lemma, reduction or extension rules: the Omega Test checks whether the equations collected by `unify_with_arith` and the equations within `EqIn` are satisfiable. If the Omega Test is successful and deems this LSE satisfiable, the equalities collected by `unify_with_arith` are passed on within the `Eq` line of parameters. The `Eq` parameters store equalities for later use. If the Omega Test is not successful, we need to try a different proof method. In chapter 7.1 we will discuss why the current equality handling is not correct and allows for false positives when testing whether a problem is a theorem.

Let us now explain how the LSE passed to the Omega Test needs to be constructed. This will clarify how equalities interpreted as constraints are tested for satisfiability. Recall that required equalities are stored both within the `Path` when doing an extension (deep omega or regular) and within the `Eq` line of parameters.

Consider at first the equalities stored within the `Path`: these equalities need to all be satisfiable at the same time, because they are all in α-relation to each other. Thus, these equalities construct a LSE made up of only conjunctions. Passing the Path to the Omega Test with conjunctions between the e-literals would be wrong, because nanoCoP-Ω does a search for contradictions (a.k.a. connections) when proving a formula, while the Omega Test does a satisfiability test. A satisfiability test for a formula is true if and only if the *negation* of said formula is contradictory. Thus, we need to turn the conjunction into its inverse operation by De Morgan, which is the disjunction. Thus, the conjunctive LSE of equalities along the path must instead be a disjunctive LSE when passed to the Omega Test. Additionally, the literals must be complementary to their counterparts within the original formula as by De Morgan. This is already taken care of due to the way nanoCoP-Ω stores its polarities: minus for T and nothing for F. Here an example with the problem ARI082=1 from the TPTP benchmark:

```
tff(associative_sum,conjecture,
    ! [Z1: $int,Z2: $int,Z3: $int,Z4: $int] :
      ( ( ( $sum(2,3) = Z1 )
        & ( $sum(Z1,6) = Z2 )
        & ( $sum(3,6) = Z3 )
        & ( $sum(2,Z3) = Z4 ) )
     => ( Z2 = Z4 ) ) ).
```

nanoCoP-Ω can only prove this formula by connecting all the e-literals via deep omega extensions. This is possible: they are all in α-relation to each other. The `$sum` type equalities end up having polarity T and thus have a minus before them within nanoCoP-Ω: The equality for Z3 is denoted as follows: `-((3^t($int)+6^t($int))^t($int)=(31^[])^t($int))`. The final equality is as follows: `(30^[])^t($int)=(32^[])^t($int)`. The LSE is passed to the Omega Test as follows:

```
[(-((2^t($int)+(31^[])^t($int))^t($int)=(32^[])^t($int));
-((3^t($int)+6^t($int))^t($int)=(31^[])^t($int));
-(((29^[])^t($int)+6^t($int))^t($int)=(30^[])^t($int));
-((29^[])^t($int)=(2^t($int)+3^t($int))^t($int));
-((2^t($int)+3^t($int))^t($int)=(29^[])^t($int));
(30^[])^t($int)=(32^[])^t($int))]
```

Note that the semicolon is the disjunction and a number with an `^[]` is actually a ð-variable. The reader may easily ascertain that this is a correct representation of the above formula for a satisfiability test.

Another way to derive the construction of the LSE is by doing an exemplary representation with a logical formula. The following equivalence obviously holds:

$$\mathrm{A} \wedge \mathrm{B} \Leftrightarrow \neg\neg(\mathrm{A} \wedge \mathrm{B}) \Leftrightarrow \neg(\neg\mathrm{A} \vee \neg\mathrm{B})$$

We now interpret the formula on the left side ($\mathrm{A} \wedge \mathrm{B}$) as a to-be-proven LSE, which is proven within a system that proves by contradiction, such as nanoCoP-Ω. Then the right side's outer negation represents the equivalence of (i) a satisfiability proof for a formula, and (ii) a proof by contradiction for the negated formula, in this case ($\neg\mathrm{A} \vee \neg\mathrm{B}$). The negation of the contained atoms (A and B) is handled by the fact that nanoCoP-Ω represents the polarities T by adding a '−' and F without adding '−'. The construction of the disjunctive form ($\neg\mathrm{A} \vee \neg\mathrm{B}$) from the path is done by the predicate `path_eq/3`. It takes the path and the current literal, which is also part of the path but has not been written to `Path` yet, and returns a disjunction of only the e-literals without the non-e-literals that may have been on the path. This predicate is discussed again in chapter 7.4.1 when I discuss some possible optimizations for future work.

It is now clear how to handle equalities from the path. There are additional equalities required by the proof that are not part of the original formula and that are stored within the various `Eq` variables, including `EqIn`. These equalities are collected in two different situations. The first situation is the unification of two, possibly arithmetic, expressions, which occurs within the lemma, reduction, and extension rules. The unification is done by the predicate `unify_with_arith/4` as discussed in chapter 5.1.4. The predicate returns necessary arithmetic equations as its third parameter. These are passed to subsequent proof steps within the `EqPreOut` and `EqOut` parameters. The second situation in which additional equations are recognized as necessary is after `omega/1` calls. When we prove an e-literal with the Omega Test we must from now on assert an additional constraint: The entire LSE must hold for the remainder of the proof. Thus, the entire LSE is stored in the `Eq` line of parameters.

We now have collected all constraints that must hold for the proof. The constraints collected in the various `Eq` variables already follow the idea of being satisfiability constraints. They are not constraints formulated for a proof by contradiction. Hence, they are passed to the Omega Test connected by conjunctions, rather than by disjunctions. This has been referenced at the beginning of this chapter.

To construct the entire LSE that we need to pass to the Omega Test, we combine the equalities from the path and the equalities from the various `Eq` variables. Recall that the equalities from the path are connected by disjunctions, and the equalities from the various `Eq` variables are connected by conjunctions. We now pass a conjunction of both collections to the Omega Test. The disjunction consisting of path e-literals must be enclosed within brackets. This constructed LSE is written to the `Eq` variables as mentioned directly above. Thus, the constructed LSE may hold further LSE from former Omega calls, which may in turn also hold older LSE.

5.2.4 Omega Call Crashes

The underlying Omega Library crashes for some problems. A list of problems affected by this crash in leanCoP-Ω can be found in chapter 6.1.4, although I am not certain whether this is a full list. The crash is caused by calling the Omega Test for equations that contain predicates. This type of equation cannot be expressed in Presburger arithmetic and cannot be handled by the Omega Test.

A crash that is unique to nanoCoP-Ω occurs when a ∂-variable that is dependent on multiple γ-variables is passed to the Omega Test. It occurs for example for the problem TFA212=1 (NUM859=1). A variable of this type looks as follows: `(32^[_123,_456])^t($int)`. Calling the Omega Test with this kind of variable results in an error. This error does not occur within leanCoP-Ω as it uses a different representation for ∂-variables.

To take care of both errors I have added a test for both types of equations to the `omega/1` predicate: `removeReplacementBrackets/2`. This predicate's purpose is to remove `^[…]` expressions and replace them with `^[]`, which the Omega Test can handle. Since leanCoP-Ω does not pass information about the ∂-variable's dependencies (see Skolem form) to the Omega Test, the Omega Test cannot use this information. We can thus cut the `^[…]` expressions, which store exactly these dependencies in nanoCoP-Ω. Additionally, the new predicate checks for predicates within the equalities. If it finds one it prevents nanoCoP-Ω from calling the Omega Test, as it would only crash anyway, and instead informs the user of this occurrence. This warning can be hidden by asserting `hide_warnings` within Prolog.

5.2.5 Deep Omega Extension

The deep omega extension rule was implemented in nanoCoP-Ω successfully. It is the last rule the prover invokes for solving an e-literal and it cannot be used for regular literals. Its code is almost the same as in leanCoP-Ω, except for an adjustment to the way nanoCoP finds the literals that still need to be proven, see the extension rule. This was done by invoking the `prove_ec` predicate. The extension part of the deep omega extension rule is implemented by calling `prove/10`. The use of the deep omega extension rule is locked behind the setting `deep_omega`.

5.2.6 *The New Start Rule*

nanoCoP-Ω contains an alternative start rule solving the problem described in chapter 4.2.5. This rule is an exact copy of the original start rule placed beneath the original start rule within the code. This placement means that the new rule is called only if the original rule fails. In the original start rule it is asserted that the chosen start clause is positive by calling the predicate `positiveC(Cla,Cla1)`. This predicate fails if the passed clause is not positive, and it returns the original clause if it is positive. The alternative start clause uses an alternative predicate called `positiveOrEqC(Cla,Cla1)`, which succeeds if the clause is positive when interpreting all e-literals as positive. The alternative start clause can be deactivated by passing the setting `eq_nev_pos`. This leads to incomplete behaviour, namely the false declaration of problems as non-theorems for 10 out of 43 problems from the TFA benchmark that were solvable within the time-limit. Using the setting `eq_nev_pos` does not lead to faster computation of results. Another setting provided by nanoCoP-Ω called `eq_is_pos` activates the use of the alternative start clause for the first pass. This does not seem to improve the runtime of the prover either, as seen in chapter 6.3. In the next chapter I discuss further settings nanoCoP-Ω provides.

5.3 Optimization settings

nanoCoP-Ω supports several settings that are used to hopefully speed up computation. Some settings eliminate completeness. Most of these settings were taken from the original nanoCoP 2.0 and the others were inspired by their usage in leanCoP-Ω, albeit under different names. Settings are passed to the initial proof call `nanocop_main(File, Set, ReturnedProof)` in the list parameter `Set`. `File` is the location of the to-be-proven problem's file.

A setting that is very successful for small arithmetic problems is `conj`. It only works for problems written in TPTP syntax, that is problems containing the role 'conjecture'. It forces the prover to select the clause that is marked as the conjecture as the start clause, and to fail if proving the conjecture is not successful. This setting is obviously not complete, but its usage is often optimal, because a problem that is solved without a connection to a literal of the conjecture is most likely stated wrongly anyway.

As mentioned in chapter 3.1 nanoCoP-Ω supports restricted backtracking. The two modes of restricted backtracking are within the start rule and within the `prove/10` predicate introduced in chapter 5.2.1. The setting `scut` activates restricted backtracking within the start rule. Under this setting the prover will not attempt to solve alternative start clauses once the first eligible one has been selected. This is done with the Prolog functor `!,` which discards alternative choices within the current node proof construction[7]. The setting `cut` restricts backtracking within the `prove/10` predicate. If it is activated,

[7] https://www.swi-prolog.org/pldoc/doc_for?object=!/0

the ! functor is added behind the Prolog code describing the different proof methods for the current literal, but before the call of prove/10 for the β-neighbours of that literal, that is before attempting to prove the rest of the current clause. This leads to the desired behaviour: once a valid method of proving the current literal has been found Prolog advances to the next expression we still need to prove. This is the cut (!). Hence the choice points, which are the other possible methods of proving the current literal, are discarded. If the now following call of prove for the β-neighbours fails, the entire predicate fails, as the remaining options for proving this literal have been discarded by the cut. As a consequence, the prover backtracks instead of attempting to prove the current clause again.

The setting deep_omega activates the use of the deep omega extension rule. Recall that it is the last rule used to attempt to solve a literal. This is because it takes the longest to potentially lead to a successful proof: another entire clause needs to be proven, and in contrast to the regular extension rule, the literal we have connected to still needs to be proven. Use of the lemma rule or the reduction rule is obviously more desirable, and so is the use of basic arithmetic or the satisfiability test, as these all require no further path extension.

nanoCoP-Ω supports the setting no_eq_arith. This setting deactivates the use of the unify_with_arith predicate described in chapter 5.1.4 and forces the prover to use only unify_with_occurs_check, the in-built Prolog predicate to unify two terms while performing an occurs-check. I have not tested the potential performance boost gained by the use of this setting because it should be safe to assume that this setting comes with no major advantages. unify_with_arith increases the functionality of the prover with almost no drawback as it relies entirely on relatively simple Prolog pattern matching, which should be very fast. Using the setting no_eq_arith only means that the prover cannot unify several otherwise unifiable arithmetic expressions. This is clearly detrimental when otherwise the lemma, reduction, or extension rules would have been applicable.

The setting reo(I) increases randomness of the proof search. Often a proof is found within automated theorem provers by starting with the conjecture and building a path with complementary connections through the entire matrix. Often the proof-relevant decision points between two possible connections are few and finding the correct one after having chosen the wrong one may require large amounts of backtracking. Instead of committing to backtracking, it can be a good idea to restart the proof, starting from the conjecture again, but with reordered clauses. This results in an additional level of randomness which can allow the prover to select the correct clause at the critical choice points. This heuristic motivates the structure of the shell script provided with the nanoCoP-Ω release from this thesis. The user can force the reordering of clauses by specifying the setting reo(I), where I is a positive integer. If this setting is specified nanoCoP-Ω will reshuffle the matrix representing the formula I times. This setting was already available in the original nanoCoP 2.0.

The setting comp(I) ensures completeness of the proof search. Multiple non-complete settings for the automated theorem prover have been presented in the last paragraphs. These settings often increase the speed of the proof search, which makes their use desirable. If the user wishes to use these faster settings while still preserving completeness for the proof search, they can use the setting comp(I), where I is a positive integer. This setting activates the following behaviour. In the prover's first attempt at proving the conjecture it uses the other settings passed alongside comp(I), and the prover is not allowed to exceed path depth I. If it fails in finding a proof within this depth limit, it restarts the search with no settings activated, thus ensuring completeness. The setting comp_eq(I) has the same behaviour, but the prover is instead restarted with the settings [deep_omega] to maximise the number of problems relying on arithmetic expressions that the prover can solve.

The setting noeq deactivates the behaviour described in chapter 5.1.2 so that the prover does not add the equality axioms to the formula. The possible benefits of this setting have not yet been evaluated, but one could reason that its completeness-harming properties outweigh the potential benefits. In theory, the equality axioms could slow down the proof search by serving as new targets for the extension rule, even though the literals of the original formula would be much better targets. This does not happen if no reordering is done, because the equality axioms are always at the end of the matrix. Consequently, we can assume that extensions to the equality axioms only happen if no other literals are available as partners for the extension rule. If the cut setting (see above) is active and there are other partners for the extension rule, the equality axioms will not even be connected to in backtracking, as the cut setting deactivates this feature. Thus, the equality axioms hardly ever slow down the proof procedure and their removal seems unnecessary.

For a description of the settings eq_is_pos and eq_nev_pos see chapter 5.2.6 on the alternative start rule.

6 Evaluation

In this chapter I present the results gathered by using the TPTP benchmark discussed in chapter 3.3 to compare leanCoP-Ω and nanoCoP-Ω. Comparing nanoCoP-Ω to other existing automated theorem provers is not within the scope of this thesis. Due to the similarity of nanoCoP-Ω and leanCoP-Ω (both are written in Prolog and implement the same arithmetic handling methods) this comparison is a direct comparison between the non-clausal connection calculus and the clausal connection calculus.

Not all problems from the TPTP library can be used for the evaluation of nanoCoP-Ω. I filtered all problems from the benchmark that are in typed first-order logic and that are theorems by using the script ‘tptp2T’ provided with the distribution of the benchmark. Polymorphic logic problems were discarded using `grep TF0` (vs. otherwise TF1). I selected only theorems because determining the non-theorem status of a formula within the non-clausal connection calculus requires the proof algorithm to make a full search through the search space described by the matrix. Thus, the calculation of the non-theorem status would always be slow, which I considered detrimental for testing because nanoCoP-Ω would almost exclusively time out. Furthermore, the typical use case of a prover is the search for a proof for a problem that is actually suspected to be valid.

The evaluation was done on a laptop at home. The specifications are as follows: x64 MS Windows 10 Home, 4 core (8 logical) 11th generation Intel Core i7-1165G7 with 2.8 GHz (and an activated overclocking feature), an SSD hard drive and 16 GB memory. The benchmark script used by Kai Münch in his bachelor thesis [14] was used almost unchanged, merely a recompilation on the test system to adjust the name of the started shell script was necessary. The script called the prover for each problem within the target directory one hundred times if the first execution took less than one second in user and system time, nine times if it took longer but was successful, and twice if the execution resulted in a time-out (which might not have been the behaviour intended by Münch). The script treated an unknown or invalid result (see chapters 6.1.2 and 6.1.3) like a valid result in terms of execution count. Each problem was in theory given 60 seconds of execution time, although the results reveal that they have slightly longer real time execution times.

In this chapter I discuss the different invalid or missing return values of nanoCoP-Ω and discuss their meaningfulness. I also evaluate the performance in comparison to leanCoP-Ω, and finally give a short evaluation of the different settings of nanoCoP-Ω.

6.1 Errors

The filters applied to the benchmark problems were insufficient. Bugs within the Omega Library and the benchmark script resulted in unexpected behaviour. This chapter discusses the different problems and how they were dealt with.

6.1.1 Non-Domain Problems

The TPTP library contains a multitude of problems. As mentioned above nanoCoP-Ω and leanCoP-Ω can only solve integer arithmetic and equality problems in typed first-order logic, which is why filtering for typed first-order theorems was the first step in collecting the problems for this benchmark. Of these 1404 problems 145 use real numbers, 94 use rational numbers and 157 use division. These problems cannot be solved correctly by leanCoP-Ω or nanoCoP-Ω. I collected these problems by running a script over the contents of the 1404 problem files and searching for ‘$real’ or ‘$rat’ within the problem formula, and ‘/’ within the problem’s name, respectively. The 312 problems were cut from the benchmark results.

6.1.2 Result Unknown

The benchmark executable created by Kai Münch in his bachelor thesis [14] writes the result 'unknown' to the benchmark execution result table if it cannot find an indicator within the output of nanoCoP/leanCoP-Ω to tell it whether the proof was successful. Ideally this case would never occur, but it does for 43 problems with nanoCoP-Ω and for 14 problems with leanCoP-Ω. All problems that fall into this category in either of the provers are ignored when comparing runtimes, because the measured execution times become unreliable. However, the problems are counted when counting the number of time-outs and the number of 'Theorem' return values. nanoCoP-Ω and leanCoP-Ω do not have unknown return values for the same problems. In fact, the only problems both benchmark executions return 'unknown' for are 'SCT171_1.p', 'ARI711=1.p' and 'COM003_1.p'.

Returning 'unknown' is usually erroneous, for example the problem ARI685=1.p is solved by nanoCoP-Ω and is returned as 'Theorem', but when solved via the benchmark executable the benchmark table holds 'unknown' for this problem. This error of the benchmark most likely occurs due to the output proof or matrix being too large and C#, the programming language the benchmark script is written in, not being able to read the file properly. I was able to read the actual returned results from a collection of files created during benchmark execution that store the provers' outputs. I thus discovered that nanoCoP-Ω actually returned 'Theorem' correctly for 18 of the 43 'unknown' results and timed out in 19 other cases. leanCoP-Ω never returned 'Theorem' correctly. It timed out in one of the 14 cases. All remaining cases were marked as 'None' by the script I used. This means that the provers had not written anything indicative to the result files. Upon manual re-execution of the problems that had delivered 'None' I discovered that 5 of the 6 problems for nanoCoP-Ω were theorems and the last problem timed out. leanCoP-Ω returned 'theorem' for all 13 'None' problems.

6.1.3 Result Invalid

If nanoCoP-Ω tells us that a problem is a non-theorem this is marked as 'invalid' within the benchmark table. This occurs for 43 problems within the TPTP benchmark constructed for this thesis. The table did not contain 'invalid' for leanCoP-Ω. These 'invalid' problems use division and had not been cut by the procedure for the removal of non-domain problems described in chapter 6.1.1, or they include predicates within arithmetic expressions. Since nanoCoP-Ω was not built to handle these problems, returning 'invalid' is expected behaviour. leanCoP-Ω never returns 'invalid' and instead enters a time-out. However, for some problems leanCoP-Ω returns a false proof. These are the problems ARI203=1.p, ARI209=1.p, ARI230=1.p and ARI236=1.p. They all are problems concerned with fractions. Since leanCoP-Ω is not supposed to handle fractions, this fact could be disregarded, but the fact that the prover manages to find a "proof" is concerning. This is an issue for future research.

A further set of problems that are solved by leanCoP-Ω, but marked as non-theorems by nanoCoP-Ω are ARI582=1.p, ARI583=1.p, ARI584=1.p, ARI586=1.p and ARI587=1.p. This discrepancy in output is worrying at first sight, because the problems are within the domain of both provers: the problems only contain regular addition and e-literals for integers. Upon further research it turned out that the false behaviour of nanoCoP-Ω is due to the prover not using the setting `deep_omega` for most of its execution time. leanCoP-Ω manages to solve the problem using the deep omega extension rule, activated by its setting `eq(2)`. Both provers are restarted multiple times during the proof search, each time with a different set of strategies. Thus, each setting is given a different amount of time. In theory nanoCoP-Ω could solve the above problems using the setting `deep_omega`, but we give nanoCoP-Ω too little time on the setting `deep_omega`. In contrast, leanCoP-Ω has enough time on the setting `eq(2)` to find a proof. To ensure completeness the script has been adjusted to use `[deep_omega]` as the final strategy in the new version.

6.1.4 leanCoP-Ω Crashes

For certain problems leanCoP-Ω encounters obscure segmentation fault crashes within its Omega Library. These crashes most likely occur because predicates within arithmetic expressions are passed to the Omega Test, although this is not certain. nanoCoP-Ω avoids this issue by refusing to pass arithmetic expressions containing predicates to the Omega Test (see chapter 5.2.4). leanCoP-Ω crashes for the problems ARI253=1.p, ARI254=1.p, ARI255=1.p, ARI264=1.p, ARI266=1.p, ARI283=1.p, ARI293=1.p, ARI294=1.p, ARI304=1.p, ARI306=1.p, ARI307=1.p, ARI337=1.p, ITP004_1.p, ITP005_1.p, ITP011_1.p, ITP015_1.p, ITP367_1.p, ITP377_1.p, NUM901=1.p, NUM923_1.p, SWW472_2.p, SWW592=2.p, SWW627=2.p and SYO522=1.p.

6.1.5 Invalid Execution Times

Five problems are marked as having an execution time of 0ms when executed by nanoCoP-Ω by the benchmark. For leanCoP-Ω, 26 problems are marked this way. All of these problems are marked with time-outs, but they may actually be solvable by the provers. I cut these 31 problems from the benchmark.

6.2 Performance

To evaluate the performance of the two provers reliably we needed to take care of the issues described in the previous chapters. Non-Domain problems and problems returning invalid execution times were removed from the benchmark entirely. Furthermore, we could not use the execution time of problems for which the benchmark table stores ‘unknown’ or ‘invalid’. The latter problems could be used when comparing the number of time-outs though.

We are now able to evaluate the success rates of both provers, i.e., the number of problems they were able to solve vs. the number of problems their execution timed out for. Then we can compare the runtimes of the two programs for the problems both succeeded in proving. While doing this we pay attention to the domains these problems stem from.

6.2.1 Success Rate

Table 2 shows the results for the evaluation of the 1061 domain problems. We see that nanoCoP-Ω has a lower success rate than leanCoP-Ω by approx. 6%. This is a considerable downgrade in solvable problems in comparison to leanCoP-Ω. While it may seem that nanoCoP-Ω is better than leanCoP-Ω concerning time-outs, this is not actually the case, as the 'invalid' cases will most likely result in time-outs once the fix discussed in chapter 6.1.3 has been applied. Hence, nanoCoP-Ω is more likely to take an unreasonable amount of time to solve a problem.

Table 2. Success rates of the two provers for the 1061 problems in the final benchmark.

Prover	# time-out	# valid	# invalid	success rate
leanCoP-Ω	714	347	0	32.7%
nanoCoP-Ω	705	282	74	26.6%

We now discuss the time-outs in greater detail. The distribution of time-outs, and of 'invalid' and 'valid' return values across these same return values of the other prover are all displayed in Table 3. We see that the two provers share a large number of time-outs, which suggests that these problems are very hard. We also see some differences in the specific problems the two provers solve. leanCoP-Ω solves 75 problems that nanoCoP-Ω does not solve. nanoCoP-Ω, on the other hand, solves 32 problems that leanCoP-Ω does not solve. This is an interesting finding, as it shows that the two calculi the provers are based on are differently well suited for proving different formulas. We also discover that the issue described in chapter 6.1.3 is perhaps more severe than expected: nanoCoP-Ω returns 'invalid' due to the settings with which it is called for 7 problems that leanCoP-Ω succeeds in proving. Had I fixed this issue earlier nanoCoP-Ω would have perhaps performed better during evaluation.

Table 3. Shared results of both provers, 'unknown' problems excluded.

		nanoCoP-Ω			
		valid	**invalid**	**time-out**	**total**
leanCoP-Ω	**valid**	225	7	75	307
	time-out	32	67	601	700
	total	257	74	676	1007

The time-outs and the ‘invalids’ unique to nanoCoP-Ω are mainly within the domain SWW (Software Verification). A few additional problems come from the domains ITP (Interactive Theorem Proving), ARI (Arithmetic), NUM (Number Theory) and a singular problem comes from the DAT (Data Structures) domain. Time-outs unique to leanCoP-Ω occur mostly for problems from the SWW domain and for some problems from the domain ARI, as well as a singular problem from each of the domains NUM, DAT times out, and two from the domain SEV (Set Theory). Both full lists can be found in the Appendix.

From these observations we can gather only that leanCoP-Ω is slightly better suited for handling problems from the ITP domain. This is perhaps the most important domain, as it is close to a possible use case for nanoCoP-Ω or leanCoP-Ω: to use the automated provers as assistants to fill in the gaps within interactive theorem provers’ proofs. The user can activate the automated provers when they wish for the computer to fill in a gap. The ITP problems within the TPTP library were generated automatically from an intermediate proof situation and goal within an interactive theorem prover (see ITP223_1.p). But even leanCoP-Ω has a very low success rate for these problems: Table 4 reveals that it has a success rate of approx. 5% in this domain.

We now analyse the success rates for the two provers by the domain the problem stems from. The data is listed in Table 4. First, consider another potentially important use case for automated theorem provers: hardware verification. For the corresponding TPTP domain HWV both provers have a success rate of 0%. As this domain is not a time-sensitive application the success rate is perhaps not as detrimental as for ITP problems, but this is only the case if leanCoP-Ω and nanoCoP-Ω are correct and complete. Further comparison with other automated provers is necessary to rate the importance of this outcome. Another domain where neither prover performs well is the domain Data Structures (DAT).

For problems of the domain SWW (Software Verification) leanCoP-Ω performs notably better than nanoCoP-Ω with a success rate of approx. 34% vs. 13%. The domain SWW is usually an important and interesting use case for automated theorem provers. However, this is not the case for nanoCoP-Ω, because this domain only requires a true/false answer, not a human-readable proof. Thus, if a normal form prover has more success in this domain this is entirely acceptable. Both provers perform well for the domains ARI (arithmetic) with success rates of approx. 55% for both provers, and NUM (Numeric) with a success rate of approx. 45% for nanoCoP-Ω and approx. 52% for leanCoP-Ω. This is a success, as it shows that nanoCoP-Ω deals well with arithmetic and equalities, the main features it has been expanded by. The other domains do not contain enough problems to give indicative data.

Table 4. Domain-specific time-outs. Ratios rounded to two decimals.

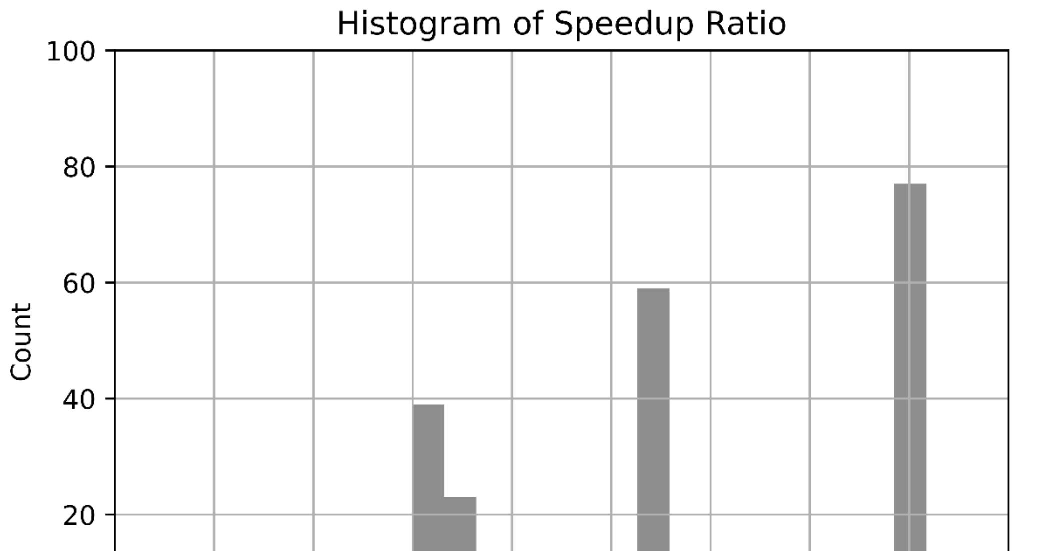
Domain	Result	Ratio Nano	Count Nano	Ratio Lean	Count Lean
ARI	invalid	0.20	71	0.00	0
	time-out	0.25	86	0.45	156
	valid	0.55	191	0.55	192
COM	valid	1.00	3	1.00	3
DAT	invalid	0.01	1	0.00	0
	time-out	0.91	63	0.94	65
	valid	0.07	5	0.06	4
GEG	time-out	1.00	5	1.00	5
GEO	time-out	1.00	1	1.00	1
HWV	time-out	1.00	136	1.00	136
ITP	time-out	0.99	174	0.95	167
	valid	0.01	2	0.05	9
KRS	valid	1.00	1	1.00	1
MSC	valid	1.00	1	1.00	1
NUM	invalid	0.02	1	0.00	0
	time-out	0.52	21	0.48	19
	valid	0.45	18	0.52	21
PUZ	time-out	0.62	5	0.50	4
	valid	0.38	3	0.50	4
SCT	time-out	1.00	6	1.00	6
SEV	time-out	0.25	1	0.75	3
	valid	0.75	3	0.25	1
SWV	valid	1.00	2	1.00	2
SWW	time-out	0.87	175	0.66	134
	valid	0.13	27	0.34	68
SYN	valid	1.00	1	1.00	1
SYO	invalid	0.25	1	0.00	0
	time-out	0.75	3	1.00	4

6.2.2 Speedup

We now analyse the speedup reached by nanoCoP-Ω in comparison to leanCoP-Ω. Here we can only regard problems that both provers succeeded in proving. The speedup ratio is calculated by dividing leanCoP-Ω's execution time by that of nanoCoP-Ω. The speedup value in ms is calculated by subtracting nanoCoP-Ω's execution time from leanCoP-Ω's. Thus, a speedup ratio bigger than one means that nanoCoP-Ω is faster, and a value bigger than zero means that leanCoP is exactly that much slower.

In total we get a real-time average speedup ratio of approx. 2.30., indicating that nanoCoP-Ω is on average slightly more than two times faster at finding its proof than leanCoP-Ω. We get an average speedup ratio of 13.70 for time spent in user mode, and of 6.83 for time spent in system mode. These last two numbers are disregarded in the following. The distribution of the real-time speedup ratio is displayed in Figure 1. The Figure shows the number of problems with a specific real-time speedup ratio for nanoCoP-Ω. nanoCoP-Ω is slower than leanCoP-Ω for only 13 of the 255 problems solved by both provers. These are mainly problems from the domains SWW and SWV, as well as single problems from the domains ARI, PUZ (Puzzles: here 'The Mislabelled Boxes'), SEV (Set Theory) and ITP. A full list can be found in the Appendix. nanoCoP-Ω has an average real-time speedup of approx. 0.24 for these 13 problems, i.e., it takes 4.2 times as long as leanCoP-Ω.

Figure 1. Real-Time Speedup Ratio. For each speedup ratio, the number of problems solved that much faster is drawn on the y-axis. Note that the figure does not include the problem 'ARI081=1.p' with a real-time speedup of approx. 8, which does not fit the scale.



The speedup distribution by domain is provided in Table 5. From it we gather that nanoCoP-Ω is faster for the domains ARI and NUM, but slower for the domain Software Verification (SWW and SWV). The entries for the other domains are not informative, because not more than 3 problems could be solved by both provers in these domains.

Table 5. Speedup by Domain. All entries have been rounded to two decimals.

Domain	Real-Time Speedup Ratio	User Speedup Ratio	System Speedup Ratio	Real-Time Speedup Value in ms	User Speedup Value in ms	System Speedup Value in ms	Problem Count
ARI	**2.41**	**13.09**	**7.96**	**1740.79**	**787.8**	**436.32**	**183**
COM	1.15	0.38	1.20	161.64	-63.09	13.65	3
DAT	1.91	6.52	2.05	996.07	771.52	64.27	3
ITP	0.69	0.24	0.94	-1962.82	-2150.73	-45.43	2
KRS	1.14	0.29	1.29	157.25	-77.78	18.65	1
MSC	3.43	45.81	3.65	2673.48	2465.86	171.52	1
NUM	**3.14**	**36.40**	**3.35**	**2887.21**	**2496.12**	**186.80**	**17**
PUZ	0.89	0.37	0.90	-552.96	-450.99	-17.31	3
SEV	0.13	0.01	0.19	-8341.59	-9165.80	-348.03	1
SWV	**0.16**	**0.01**	**0.25**	**-10116.65**	**-10321.18**	**-616.76**	**2**
SWW	**0.31**	**0.07**	**0.31**	**-16150.75**	**-15551.08**	**-1617.52**	**8**
SYN	1.15	0.56	1.49	165.87	-25.88	25.62	1

To summarize, we have ascertained that leanCoP-Ω can solve slightly more problems within the time limit. This particularly applies to problems of the domains SWW and ITP. The provers share time-outs for 601 problems, but each also proves problems that the other does not prove. nanoCoP-Ω is considerably faster than leanCoP-Ω at proving ARI and NUM problems. These problems partially have a very shallow depth, so the computational overhead of transforming a matrix into normal form, which only leanCoP-Ω needs to do, might have a considerable impact, as the total computation time is still very low. leanCoP-Ω solves the larger problems from the SWW/SWV domain faster than nanoCoP-Ω. This indicates that the clausal connection calculus is better for dealing with large problems.

6.3 Comparison of Settings

To evaluate the different settings supported by nanoCoP-Ω a custom benchmark was run on the TFA benchmark (see chapter 3.3) prior to the execution of the TPTP benchmark. Recall that the TFA benchmark consists of 75 select problems from the TPTP library version 5.0.0. The evaluation was carried out by running nanoCoP-Ω for every combination of the settings `conj`, `cut`, `deep_omega`, eq_is_pos, eq_nev_pos and scut. The different setting combinations are called *strategies*. In the evaluation they were ranked by their average runtime across all problems and by their success rate. The success rate of a strategy is the ratio of problems that can be solved in relation to the total number of problems.

The success rate is a factor we need to discuss because the settings can influence completeness. The setting eq_nev_pos caused the prover to falsely mark a problem as a non-theorem for 9 different problems. The problem TFA198=1 was only solved when using the setting deep_omega.

The best settings in both success rate and average execution time were used to create the shell script nanocop20Omega.sh, which is used to start the nanoCoP-Ω prover. The shell script restarts the proof search with nanoCoP-Ω multiple times, each time with a different strategy and a different number of reorderings of the matrix (see setting reo(I) from chapter 5.3). The first set of strategies consists of non-complete strategies and each strategy is given very little time. This maximises the benefit we can gain from randomness. These non-complete strategies were all selected based on success rate and average execution time measured on the TFA benchmark. The final proof strategy (formerly [] and now [deep_omega]) is given the bulk of execution time in case the others fail.

For the execution of the TFA benchmark additional logging was added to the prover. The log informs us when the prover uses one of the new arithmetic handling proof methods, that is when the prover:

- uses the leanari predicate successfully to prove a self-satisfying literal (see chapter 4.2.3).
- proves a literal by treating it as a constraint passed to the Omega Test (see chapter 4.2.2).
- does a deep omega extension (see chapter 4.2.4) and succeeds in proving the proof branch emanating from that extension.
- uses the alternative start rule presented in chapter 4.2.5 successfully.

The logging can be activated by asserting the predicate omega_statistics in Prolog.

Evaluating the log revealed that the deep omega extension is only ever successful in 3 of the 43 solvable problems within the TFA benchmark. In two cases the method greatly decreases the runtime, showing that the deep omega extension can be worth activating even if it is not necessary. leanari is necessary for a few more problems. The new alternative start rule is used sometimes, but not consistently for individual problems, meaning it is not necessary. Since I do not know how such results could come to pass, I will not speculate about the importance of this rule for the prover. Solving a literal by treating it as a constraint is done more regularly (approx. 25% of problems) and a problem either requires the use of this method, or it does not. The major purpose of introducing this logging is to make future research into possible optimizations for the different arithmetic proof methods more accessible.

The TFA benchmark was also used to compare the runtimes of the different settings. I wanted to derive the improvement in runtime every setting grants but was not successful in finding conclusive results. I will briefly describe the attempt. To calculate the improvement in runtime for a setting I subtracted the mean runtime for strategies containing the setting from the runtime for the exact same strategy minus the setting. So, for example the mean runtime of the strategy `[cut,deep_omega,scut]` is compared with the runtime of the strategy `[cut,scut]` to calculate the improvement granted by `deep_omega`. Once all possible strategy pairs had been compared I calculated the mean improvement for every setting. However, it turned out that the sample was too small. When investigating two problems which were solved considerably faster when using the setting `deep_omega` (TFA082=1 (ARI082=1) and TFA195=1 (ARI195=1)), I found that the removal of the two problems from the database caused extreme shifts of the values in the entire table. Hence, I abandoned this experiment.

7 Future Work

This section will discuss a major issue discovered while working on nanoCoP-Ω that needs to be addressed in future work. A potential future research project is described, possible extensions of the domain of nanoCoP-Ω are outlined, and two optimizations of the code that could be made are presented.

7.1 Soundness

In chapter 5.1.4 I presented the predicate `unify_with_arith`. It passes necessary equations it discovers to the Omega Test. However, it does not pass enough equalities to the Omega Test, as discovered with the following formula:

```
tff(b,type,( p: $int > $o )).
tff(a,conjecture,(
    ? [A: $int] : ( $less(A,5) & ( (5=5 &  p(6) )  => p(A) ) )
)).
```

The first statement in the formula describes the type of the predicate `p`. `p` takes an integer and returns a Boolean. The conjecture is a non-theorem: there is no `A` such that `A` is smaller than 5 and `p(A)` follows from `p(6)`. The `5=5` literal in the conjecture may seem redundant, but curiously its presence allows nanoCoP-Ω to find a ‘proof’ and mark the conjecture as a theorem, which it is not. The process nanoCoP-Ω goes through to reach this wrong conclusion is presented in this chapter.

First nanoCoP-Ω transforms the formula into the following matrix:

$$\left[\left|\begin{matrix} \mathrm{A} < 5^{\mathrm{F}} \\ [|5 = 5^{\mathrm{T}}| \quad |\mathrm{p}(6)^{\mathrm{T}}| \quad |\mathrm{p}(\mathrm{A})^{\mathrm{F}}|] \end{matrix}\right|\right]$$

It then attempts to prove the matrix by starting with the $\mathrm{A} < 5^F$ literal. This can be done easily with the Omega Test, as the e-literal is satisfiable. It then writes this e-literal to the correct `Eq` variable. We now need to prove the β-clause to this literal: the lower sub-matrix. The literals contained within this clause are all in α-relation, thus a connection between $\mathrm{p}(6)^{\mathrm{T}}$ and $\mathrm{p}(\mathrm{A})^{\mathrm{F}}$ is made. Now, if nanoCoP-Ω worked correctly, it would disallow unifying these two literals, because of the constraint $\mathrm{A} < 5$. However, nanoCoP-Ω fails at enforcing the constraint. Here is the corresponding call of `unify_with_arith` within a regular Prolog console:

```
?- unify_with_arith(p(A),p(6),Eq,[]).
A = 6,
Eq = [].
```

We see that Prolog recognizes that `A` and `6` need to be unified. The predicate does not return this within `Eq` though. As a consequence, the then following call of `omega/1` does not catch the mistake:

```
…,unify_with_arith(NegL,NegLit,EqArith,Set),
append(EqIn,EqArith,Eq1), omega(Eq1),…
```

The only equality passed to the Omega Test is `[_2900^t($int)<5^t($int)]`, where `A`, which Prolog stores as the variable `_2900`, has not been replaced with `6`, which would perhaps be the behaviour intended by the developers of leanCoP-Ω. Consequently, the Omega Test marks the LSE as satisfiable because `A` could for instance be `4`.

This erroneous behaviour does not occur if the order of `$less(A,5)` and the implication are switched in the original conjecture. Curiously, if we remove the literal `5=5` from the conjecture nanoCoP-Ω times out when searching for a proof. This makes no sense for two reasons. First, the literal is not used in the false proof anyway. Second, the conjecture's matrix is very small, so a time-out is surprising.

We have found a counterexample for nanoCoP-Ω's soundness. We cannot rely on the returned result 'theorem' to be certain that the conjecture is in fact a theorem. I will discuss some possible solutions to the problem in chapter 7.4.2. It is not clear if the problem can occur in leanCoP-Ω. I have not been able to reproduce this behaviour in leanCoP-Ω.

To see if this erroneous behaviour is caught by the typed first-order logic non-theorems from the TPTP benchmark I have run nanoCoP-Ω on these as well. nanoCoP-Ω did not return false proofs for these problems and instead timed out for all but one problem. For the non-domain problem ARI533=1 it returned a false proof, but since ARI533=1 contains real numbers this is not worrying. Interestingly, nanoCoP-Ω seems to round the real numbers, which is behaviour that needs to be eliminated in future work. This shows us that the non-theorems provided in the TPTP library are insufficient for an extensive soundness check.

7.2 Combination of nanoCoP-Ω and leanCoP-Ω

As discovered in chapter 6.2.1, nanoCoP-Ω and leanCoP-Ω have a different coverage of problems they can solve. Thus, a combination of the two systems would achieve a larger coverage of solvable problems. Building such a combined system that first attempts to solve the problem with nanoCoP-Ω and after a certain time limit with leanCoP-Ω could yield a powerful automated prover.

Since nanoCoP-Ω is considerably faster than leanCoP-Ω (at least for purely arithmetic and numeric problems), starting with it could lead to the combined system often finding human-readable proofs quickly. If a proof with nanoCoP-Ω takes too long, switching to leanCoP-Ω may increase the chance of success, because the clausal connection prover seems to be better suited for large problems. Recall that within a 60 second time limit leanCoP-Ω has an approx. 5% success rate for ITP domain problems within the TPTP benchmark, while nanoCoP-Ω only solves approx. 2% of them.

Something that needs to be considered when building a combined system is that nanoCoP-Ω seems to perform best with restricted backtracking enabled via the settings `cut` and `conj`. This was discovered when comparing the settings as described in chapter 6.3. So, it is perhaps not the best strategy to start the nanoCoP-Ω prover with the settings `[]` (or `[deep_omega]` to increase chance of success for arithmetic problems) when encountering a large problem, as was done for the benchmark within this thesis. This thesis has not succeeded in evaluating the success rate of restricted backtracking settings when these settings are given large amounts of time to execute. Most of the execution time (40 of the 60 assigned seconds) was given to the setting `[]`.

A more ambitious project would be to create a new CoP system capable of solving a problem within Prolog with both nano- and lean-style calculi in parallel. If it is possible to create such a system entirely within Prolog, the system gains very much control over its search strategy. It would be able to calculate the matrix in its different representations and then begin running different strategies in parallel. This would remove the overhead of starting Prolog for different possible strategies. Programming such a system would require extensive knowledge of Prolog to introduce parallelism without leaking data between executions.

Either version of the proposed system would be well-suited for integration into an interactive theorem prover. Sledgehammer [32] is a tool for integrating automated theorem provers into Isabelle, an interactive theorem prover. With a button click the user starts the automated theorem prover of their choice, and Isabelle passes the current proof situation to the prover whilst the user may continue working in parallel to the computer. The human-readable proofs returned by nanoCoP-Ω contribute to the user's research. Meanwhile leanCoP-Ω provides a fallback for tougher problems that nanoCoP-Ω may not handle well.

7.3 Additional Number Domains and Arithmetic Procedures

The Omega Test is based on Presburger arithmetic and as such cannot reason over predicates within arithmetic expressions. However, use cases for predicates within arithmetic expressions are plentiful: although I do not have exact numbers it is safe to assume that a large portion of the problems that nanoCoP-Ω timed out for within the TPTP benchmark include first-order predicates within arithmetic expressions. Expanding the Omega Test or nanoCoP-Ω to be able to handle these expressions would greatly increase the domain it can be used in.

The Omega Test is not the only arithmetic decision algorithm. Others include Arith [7] , Sup-Inf [28] or Cooper [8]. These algorithms, while all being LSE satisfiability tests, may solve different problems within different time limits. A system that uses multiple of these algorithms in parallel may perform considerably better. The Arith procedure has already been implemented in Prolog by Troelenberg in [31], and as such would be easy to add to nanoCoP-Ω once support for parallelism has been introduced.

The aforementioned algorithms and the Omega Test are only capable of handling integer LSE. Within the TPTP benchmark excerpt used to evaluate nanoCoP-Ω at least 145 problems use real numbers, 94 use rational numbers and 157 use division. An extension to these domains would be useful. This would require different algorithms. Some algorithms that can solve real number LSE include Hodes [11], Loop [27] and automaton-based algorithms [6]. Division is problematic, as all of the LSE testing problems I know of are based on Presburger arithmetic, which cannot formalise division [23]. An algorithm based on another axiomatization of (ideally) real numbers that can handle division would be. The Omega Test provides methods for calculating intermediate solutions, so perhaps a combination of two algorithms in which the first algorithm (e.g., the Omega Test) handles all but the equations with division and then passes the intermediate solution the second algorithm, which is in turn based on a theory capable of handling division, would maximize expressiveness. However, we cannot achieve completeness for all of arithmetic, as the arithmetic described by the Peano Axioms, which are strong enough to formalise both division and addition, is undecidable [3].

7.4 Optimizations

7.4.1 `path_eq` Optimization

The predicate `omega_check/1` transforms the input equalities into the shape necessary for the Omega Test. This is the shape `a*X+b*Y+…+c*Z+d=0` (potentially '<' instead of '='). The `unify_with_arith` predicate applies the same transformation when constructing the equalities it stores for the proof procedure. Because of the latter, all equalities stored in the `Eq` line of parameters are already of the correct shape. This allows us to make the following observation: the only equalities passed to the `omega_check` predicate not in the correct shape are the ones from the current path.

This allows us to derive a possible optimization. If we were to add the transformation to the `path_eq` predicate, we could remove the transformation from the `omega_check` predicate. To save more execution time we could even store all transformed e-literals from the path in a separate variable, thus enforcing that each e-literal is transformed only once. This would only allow for a $O(n)$ optimization in problem size but should be easy to implement.

7.4.2 Redundant Omega Test Calls

When analysing the calls to the `omega` predicate that occur during the proof search it becomes obvious that the Omega Test is consistently called redundantly. The test is often called with the exact same parameters two to four times in a row. This is not necessarily an error. I first make two observations that allow me to propose a new optimization. I then discuss whether these observations could be superficial and explain their connection to the problem that allowed us to notice the non-soundness of nanoCoP-Ω. This leads to speculation over how to restore soundness.

These redundant calls to the Omega Test most likely occur when the prover considers multiple candidates for the reduction or extension rules: The negated version of the to-be-proven literal is unified with said candidate. If this unification is successful, the prover calls the Omega Test. This is where multiple redundant calls can be observed: the passed LSE are the same multiple times in a row. It appears that `unify_with_arith` has returned an empty set of necessary equalities for each potential extension and reduction partner. This is the first observation: `unify_with_arith` can collect no further equations and this leads to the LSE not changing in this proof step.

The second observation is concerned with the construction of `EqIn`, which stores the equations that are necessary for the proof. The `Eq` line of parameters only ever gains new equalities after the prover has tested the current LSE for satisfiability and this test was successful. This means that all equalities within the `Eq` line of parameters have already been tested by the `omega` predicate.

These two observations lead to the possible optimisation: We could completely forgo doing an Omega Test if `unify_with_arith` returns an empty set of equations. This optimisation may be incorrect. It is possible that it only appears like the `unify_with_arith` does not affect the LSE of the proof whenever it does not return addition equalities. Here is an example of a call to `unify_with_arith` that does not return any equations but might change the LSE subtly. The example is taken from chapter 7.1.

```
?- unify_with_arith(p(A),p(6),Eq,[]).
A = 6,
Eq = [].
```

`unify_with_arith` unifies `A` with `6`. It does not store this unification (which is an equation) in `Eq` and instead Prolog stores this internally. In theory, Prolog could now adjust the LSE by replacing all occurrences of `A` with `6`. We have seen that this does not actually happen in chapter 7.1.

From the past observations we can derive two separate adjustments to nanoCoP-Ω that each restore soundness, or that at least prevent nanoCoP-Ω from finding the 'proof' presented in chapter 7.1. The first adjustment would affect the predicate `unify_with_arith`. As seen in the above example `unify_with_arith` does not return Prolog unifications as equalities (`A=6`). If we were to change this, i.e., `unify_with_arith` would also return the equality `A=6` in the above example, we may have re-established soundness. This is because the additional equalities store the equalities that go missing in the current implementation of nanoCoP-Ω. I do not know how this method could be implemented. If we were to implement this adjustment the optimization discussed during this chapter could be applied, as the returned `Eq` would not be empty in the potentially problematic cases.

The second possible adjustment is to adjust nanoCoP-Ω to replace occurrences of unified expressions with their constant versions when passing the LSE to the Omega Test. This would lead to the 'proof' from chapter 7.1 becoming impossible, because the LSE is adjusted to contain `6` where the variable `A` was located before. It is unclear how nanoCoP-Ω would determine if a variable needs to be replaced and how it would determine what to replace the variable by. It is possible that leanCoP-Ω already 'implements' this adjustment because of the different way ∂-variables are stored. Perhaps this adjustment already is the default behaviour for Prolog, but the new version of ∂-variables in nanoCoP-Ω deactivated this feature somehow. If we were to implement this adjustment the optimization discussed during this chapter could not be applied, as `Eq` may be empty when the LSE has been changed.

7.5 The TPTP Benchmark

nanoCoP-Ω has been compared with leanCoP-Ω via an excerpt of the TPTP benchmark containing only typed first-order theorems. A comparison via first-order theorems from the TPTP library is something that was not done due to these problems not containing arithmetic expressions. Thus, a comparison would yield similar results to the comparison between leanCoP and nanoCoP in [19]. A comparison to other automated provers would be more interesting.

8 Conclusion

The work of this thesis has ended in a partial success. The integration of arithmetic and equality handling procedures was successful. The implementation of new arithmetic handling methods has provided us with a capable prover, although completeness of nanoCoP-Ω for arithmetic problems can never be achieved due to undecidability. We have learnt of the importance of the deep omega extension rule and of the alternative start rule during testing, as certain theorems require these rules to be proven at all. The result: nanoCoP-Ω can solve approx. 55% of purely arithmetic problems and approx. 45% of numeric problems from the TPTP benchmark, a very comparable score to leanCoP-Ω. As hoped, nanoCoP-Ω solves these problems considerably faster due to its non-clausal approach. The non-clausal approach does not seem to be strictly better than the clausal approach though: We have learned that leanCoP-Ω seems to be better suited for the larger problems from the Interactive Theorem Prover and Software Verification domains. This is worrying, as the main benefit of nanoCoP-Ω over clausal form provers is the more human-readable returned proof, which would be particularly useful in ITP for tools like Sledgehammer [32]. Because leanCoP-Ω and nanoCoP-Ω both solve problems from the TPTP benchmark the other prover does not solve, I have proposed the combination of these two systems into a singular CoP system, hopefully maximizing success rates. We have also learnt that in its current form nanoCoP-Ω is not suitable for use in any way, as it is not sound. I was able to exploit the way the predicate `unify_with_arith` returns necessary equations to the Omega Test to make nanoCoP-Ω prove a non-theorem. This problem may or may not affect leanCoP-Ω, as the procedure for proving equalities and the corresponding predicates used are the same, but variables are stored slightly differently. I have also outlined further optimizations to nanoCoP-Ω that could be done in future work.

I. References

[1] Equality axioms. Encyclopedia of Mathematics. Access 9/2021 URL: http://encyclopediaofmath.org/index.php?title=Equality_axioms&oldid=46837.

[2] GreyOrange Robotics. Access 9/2022. URL: www.greyorange.com/products/butler.

[3] Peano axioms. Encyclopedia of Mathematics. Access 9/2022 URL: http://encyclopediaofmath.org/index.php?title=Peano_axioms&oldid=43518.

[4] Kenneth Appel and Wolfgang Haken. Every planar map is four colorable. *Bulletin of the American mathematical Society*, 82(5):711–712, 1976.

[5] Alexander Benedict Behrens. Modernisierung von Legacy Beweissystemen. Bachelors Thesis, Universität Potsdam, 2019.

[6] Bernard Boigelot, Sébastien Jodogne, and Pierre Wolper. An effective decision procedure for linear arithmetic over the integers and reals. *ACM Transactions on Computational Logic (TOCL)*, 6(3):614–633, 2005.

[7] Tat Chan. An Algorithm for Checking PL/CV Arithmetic Inferences. Technical report, USA, 1977.

[8] David C Cooper. Theorem proving in arithmetic without multiplication. *Machine intelligence*, 7(91-99):300, 1972.

[9] Gonzalo Escalada-Imaz and Malik Ghallab. A Practically Efficient and Almost Linear Unification Algorithm. *Artif. Intell.*, 36(2):249–263, 1988.

[10] Limor Fix. Fifteen Years of Formal Property Verification in Intel. In Orna Grumberg and Helmut Veith, editors, *25 Years of Model Checking*, Lecture Notes in Computer Science, pages 139–144. Springer, Berlin, Heidelberg, 2008.

[11] Louis Hodes. Solving problems by formula manipulation in logic and linear inequalities. In *Proceedings of the 2nd International Joint Conference on Artificial Intelligence*, pages 553–559, 1971.

[12] Laura Kovács and Andrei Voronkov. First-Order Theorem Proving and Vampire, 2013.

[13] Alberto Martelli and Ugo Montanari. An Efficient Unification Algorithm. 4:258–282, 1982.

[14] Kai Münch. Entwicklung einer performanten und verlässlichen Schnittstelle für leancop-Ω. Master's thesis, Universität Potsdam, 2021. Bachelors Thesis.

[15] J. Otten, H. Trölenberg, and T. Raths. leancop-Ω. Access 9/2022. URL: http://www.tptp.org/CASC/J5/leanCoP-Omega---0.1.pdf

[16] Jens Otten. leanCoP 2.0 and ileanCoP 1.2: High performance lean theorem proving in classical and intuitionistic logic. (System descriptions). In *Automated reasoning. 4th international joint conference, IJCAR 2008, Sydney, Australia, August 12–15, 2008 Proceedings*, pages 283–291. Berlin: Springer, 2008.

[17] Jens Otten. Restricting backtracking in connection calculi. *AI Communications*, 23(2-3):159–182, 2010.

[18] Jens Otten. A Non-clausal Connection Calculus. In Kai Brünnler and George Metcalfe, editors, *Automated Reasoning with Analytic Tableaux and Related Methods - 20th International Conference, TABLEAUX 2011, Bern, Switzerland, July 4-8, 2011. Proceedings*, volume 6793 of *Lecture Notes in Computer Science*, pages 226–241. Springer, 2011.

[19] Jens Otten. Nanocop: A Non-clausal Connection Prover. In Nicola Olivetti and Ashish Tiwari, editors, *Automated Reasoning - 8th International Joint Conference, IJCAR 2016, Coimbra, Portugal,*

June 27 - July 2, 2016, Proceedings, volume 9706 of *Lecture Notes in Computer Science*, pages 302–312. Springer, 2016.

[20] Jens Otten. nanocop: Natural Non-clausal Theorem Proving. In Carles Sierra, editor, *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pages 4924–4928. ijcai.org, 2017.

[21] Jens Otten. The nanocop 2.0 Connection Provers for Classical, Intuitionistic and Modal Logics. In Anupam Das and Sara Negri, editors, *Automated Reasoning with Analytic Tableaux and Related Methods - 30th International Conference, TABLEAUX 2021, Birmingham, UK, September 6-9, 2021, Proceedings*, volume 12842 of *Lecture Notes in Computer Science*, pages 236–249. Springer, 2021.

[22] Dag Prawitz. A proof procedure with matrix reduction. In *Symposium on Automatic Demonstration (Versailles, 1968)*, Lecture Notes in Math., Vol. 125, pages 207–214. Springer, Berlin, 1970.

[23] M. Presburger. Uber die Vollstandigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchen die Addition als einzige Operation hervortritt. *Comptes-Rendus du ler Congres des Mathematiciens des Pays Slavs*, 1929.

[24] William Pugh and the entire Omega Project Team. The Omega Project: Frameworks and Algorithms for the Analysis and Transformation of Scientific Programs. Access 9/2022. URL: https://www.cs.umd.edu/projects/omega/

[25] William W. Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In Joanne L. Martin, editor, *Proceedings Supercomputing '91, Albuquerque, NM, USA, November 18-22, 1991*, pages 4–13. ACM, 1991.

[26] John Alan Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *J. ACM*, 12(1):23–41, 1965.

[27] Robert Shostak. Deciding linear inequalities by computing loop residues. *Journal of the ACM (JACM)*, 28(4):769–779, 1981.

[28] Robert E. Shostak. On the SUP-INF Method for Proving Presburger Formulas. *J. ACM*, 24(4):529–543, 1977.

[29] G. Sutcliffe. The CADE ATP System Competition - CASC. *AI Magazine*, 37(2):99–101, 2016.

[30] G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure. From CNF to TH0, TPTP v6.4.0. *Journal of Automated Reasoning*, 59(4):483–502, 2017.

[31] Holger Trölenberg. Arithmetik im Automatischen Theorembeweisen. Diplomarbeit, Universität Potsdam, 2010

[32] Makarius Wenzel. *The Isabelle/Isar Reference Manual*, 2009. p. 162-164.

II. Appendix

The Domain Structure of the TPTP. Available under https://tptp.org/TPTP/TR/TPTPTR.shtml .

Logic	Combinatory logic		COL
	Logic calculi		LCL
	Henkin models		HEN
Mathematics	Set theory		SET, SEU, and SEV
	Graph theory		GRA
	Algebra	Relation algebra	REL
		MV Algebras	MVA
		Boolean algebra	BOO
		Robbins algebra	ROB
		Left distributive	LDA
		Lattices	LAT
		Quantales	QUA
		Kleene algebra	KLE
		Groups	GRP
		Rings	RNG
		Fields	FLD
		Homological algebra	HAL
		Real Algebra	RAL
		General algebra	ALG
	Number theory		NUM and NUN
	Topology		TOP
	Analysis		ANA
	Geometry		GEO
	Category theory		CAT
Computer science	Computing theory		COM
	Knowledge representation		KRS
	Natural Language Processing		NLP
	Planning		PLA
	Agents		AGT
	Commonsense Reasoning		CSR
	Semantic Web		SWB
	Interactive Theorem Proving		ITP
	Data Structures		DAT
	Software creation		SWC
	Software verification		SWV and SWW
Science and Engineering	Biology		BIO
	Hardware creation		HWC
	Hardware verification		HWV
	Medicine		MED
	Processes		PRO
	Products		PRD

Social sciences	Social Choice Theory	SCT
	Management	MGT
	Geography	GEG
Arts and Humanities	Philosophy	PHI
Other	Arithmetic	ARI
	Syntactic	SYN and SYO
	Puzzles	PUZ
	Miscellaneous	MSC

The following lists are the result of the benchmark testing conducted within this thesis.

List of Problems that nanoCoP Omega times out for (not including 'invalid' problems), while leanCoP Omega solves them:

```
['SWW012=1.p', 'SWW046=1.p', 'SWW058=1.p', 'PUZ018_1.p',
'SWW026=1.p', 'SWW087=1.p', 'SWW051=1.p', 'SWW018=1.p',
'SWW047=1.p', 'ITP227_1.p', 'SWW035=1.p', 'SWW004=1.p',
'SWW473_3.p', 'SWW084=1.p', 'SWW629=2.p', 'NUM925_1.p',
'SWW008=1.p', 'SWW042=1.p', 'SWW032=1.p', 'ITP322_1.p',
'SWW049=1.p', 'SWW021=1.p', 'SWW043=1.p', 'SWW071=1.p',
'DAT058=1.p', 'ARI615=1.p', 'SWW094=1.p', 'SWW088=1.p',
'SWW054=1.p', 'SWW060=1.p', 'SWW023=1.p', 'ITP302_1.p',
'ARI592=1.p', 'SWW473_1.p', 'ITP006_1.p', 'SWW057=1.p',
'SWW061=1.p', 'NUM891=1.p', 'SWW052=1.p', 'SWW011=1.p',
'SWW010=1.p', 'SWW015=1.p', 'SWW039=1.p', 'SWW090=1.p',
'SWW040=1.p', 'SWW031=1.p', 'SWW025=1.p', 'SWW022=1.p',
'SWW029=1.p', 'ITP222_1.p', 'SWW020=1.p', 'NUM892=1.p',
'SWW002=1.p', 'NUM925_3.p', 'SWW055=1.p', 'SWW072=1.p',
'SWW050=1.p', 'SWW013=1.p', 'SWW024=1.p', 'SWW037=1.p',
'SWW082=1.p', 'SWW017=1.p', 'SWW478_3.p', 'SWW030=1.p',
'SWW038=1.p', 'SWW070=1.p', 'ITP327_1.p', 'SWW033=1.p',
'SWW000=1.p', 'SWW064=1.p', 'SWW014=1.p', 'SWW048=1.p',
'SWW473_2.p', 'ITP239_1.p', 'SWW089=1.p']
```

List of Problems that leanCoP Omega times out for, while nanoCoP Omega solves them:

```
['SWW590=2.p', 'SWW601=2.p', 'SWW607=2.p', 'SWW655=2.p',
'ARI588=1.p', 'ARI684=1.p', 'ARI399=1.p', 'SWW632=2.p',
'ARI186=1.p', 'SWW635=2.p', 'SWW620=2.p', 'ARI585=1.p',
'SWW633=2.p', 'SWW581=2.p', 'ARI602=1.p', 'SWW653=2.p',
'NUM919=1.p', 'SEV422=1.p', 'ARI701=1.p', 'SWW652=2.p',
'SEV425=1.p', 'SWW621=2.p', 'SWW634=2.p', 'DAT101=1.p',
'SWW679=1.p', 'SWW656=2.p', 'SWW672=2.p', 'DAT103=1.p',
'SWW654=2.p', 'SWW597=2.p', 'SWW658=2.p', 'ARI691=1.p']
```

List of Problems that leanCoP Omega solved faster than nanoCoP Omega:

```
['ARI734=1.p', 'PUZ012_1.p', 'SEV421=1.p', 'SWV997=1.p',
'SWV998=1.p', 'SWW003=1.p', 'SWW006=1.p', 'SWW478_1.p',
'SWW478_2.p', 'SWW600=2.p', 'SWW613=2.p', 'SWW619=2.p',
'ITP010_2.p']
```