nanoCoP-Ω: A Non-Clausal Connection Prover with Arithmetic

Leo Repp[1], Mario Frank[2]

[1]TUD Dresden University of Technology, 01062 Dresden, Saxony, Germany
[2]University of Potsdam, Institute of Computer Science, An der Bahn 2, 14476 Potsdam, Brandenburg, Germany

Abstract

In this work, we present nanoCoP-Ω, an extension of the non-clausal connection prover nanoCoP capable of handling arithmetic and arithmetic equations, following the novel approach taken by leanCoP-Ω. We describe these methods and their implementation into nanoCoP. The performance of nanoCoP-Ω is then compared to that of leanCoP-Ω, using suitable TFA problems from the TPTP library.

Keywords

non-clausal connection calculus, arithmetic, equality, presburger, omega test, nanoCoP, leanCoP, automated theorem proving, equations, unification

1. Introduction

State of the art automated theorem provers (ATPs) like Vampire [1], CVC 4[2], iProver[3] and Princess [4] are capable of proving first order logic (FOL) problems including equalities and arithmetic. While the former is usually done by applying methods like paramodulation [5] or decision procedures like the congruence closure algorithm [6], the latter is usually handled by delegating arithmetic expressions to SMT solvers, like Z3 [7].

However, connection based ATP systems are usually not capable of solving FOL problems that include arithmetic equations, i.e., (in-)equalities containing typed arithmetic expressions. One of the few ATP systems that does support this type of problem is leanCoP-Ω [8], which was presented at the CASC J5 competition [9] and performed very well. But leanCoP-Ω has one significant drawback: Every formula is transformed into disjunctive or definitional normal form, which destroys the original formula's structure and reduces the possibility of reusing the generated proofs in interactive theorem provers like Coq [10] or Isabelle [11].

The non-clausal connection prover nanoCoP [12, 13, 14] is capable of proving the problems in the original formula's structure, but is not able to handle equations involving (integer) arithmetic. In this document, we present an extension of nanoCoP that fills this gap. This extension was created in course of a Bachelor's thesis [15]. The new ATP nanoCoP-Ω uses the basic concepts of leanCoP-Ω, with some necessary modifications to the original concept. Note that the ideas are all taken from leanCoP-Ω, we merely present and implement these methods and demonstrate our

AReCCa 2023: Automated Reasoning with Connection Calculi, 18 September 2023, Prague, Czech Republic
leo.repp.0410@gmail.com (L. Repp); mario.frank@uni-potsdam.de (M. Frank)
0000-0003-1021-6959 (L. Repp); 0000-0001-8888-7475 (M. Frank)

experiences in working with them. We expected the resulting system to provide more human-readable and reusable proofs than leanCoP-Ω due to its different matrix representation. Also, the proofs given by leanCoP-Ω do not include information about results from the arithmetic tools, which, in contrast, nanoCoP-Ω's proofs do. Additionally, we expected nanoCoP-Ω to have a better runtime performance and to hopefully solve more problems than leanCoP-Ω. But, since already the original nanoCoP proves slightly fewer problems than leanCoP [16], the probability of proving more problems with nanoCoP-Ω was quite low. While the runtime performance indeed has shown to be better than that of leanCoP-Ω, leanCoP-Ω solved slightly more problems. Importantly, the presented methods still need to be formalized and verified. We discuss the underlying arithmetic tool: the Omega Library, its hook in Prolog and the implementation and performance of nanoCoP-Ω.

In the following, we will discuss relevant preliminaries, like leanCoP, arithmetic, leanCoP-Ω, the Omega Library and nanoCoP. Then, we present the way we implemented the adaptation of nanoCoP to handle arithmetic equations, followed by an in-depth evaluation comparing the performance of both leanCoP-Ω and nanoCoP-Ω on TFA problems from the TPTP [17]. This is followed by ideas for future work, including possible optimisations. Finally, we discuss the potential overall impact of nanoCoP-Ω on connection based theorem proving.

2. Preliminaries

2.1. leanCoP and nanoCoP

leanCoP is an automated lean clausal connection prover for first order logic with regular equalities. It was first presented in [18], further extended to leanCoP 2.0 [19, 20] and, written in Prolog, is very small and elegant and thus easily maintainable. It is based on the clausal connection calculus, meaning it proves a formula by transforming it into a matrix, i.e. a set of clauses in disjunctive normal form. The proof is done by ensuring that each path in the matrix contains a so-called connection. It works with matrices of formulas in a mix of disjunctive and definitional normal form, meaning that the structure of the original formula is usually lost.

This information loss is resolved by nanoCoP, a natural non-clausal connection prover also written in Prolog. The main difference to leanCoP lies in the underlying calculus, i.e. in the matrix representation, a new decomposition rule and a generalised extension rule. The matrices in nanoCoP directly correspond to the structure of the original formula and include sub-matrices. This increases the complexity of representation but has significant advantages. Obviously the understandability for humans can be increased. And even more importantly, the proof search is able to exploit this structure to eliminate large parts of the search tree efficiently. Finally, the proof term is more easily reusable for interactive theorem provers, e.g. via Sledgehammer [21]. Testing these hypotheses are the reasons we chose to transfer the arithmetic and arithmetic equation methods to nanoCoP.

2.2. From leanCoP to leanCoP-Ω

In the following, we will always assume arithmetic bound by Presburger arithmetic [22], but extended by subtraction and multiplication with constants. This means function symbols

within arithmetic terms cannot be evaluated. Whenever talking about equations, inequations or constraints, we mean pairs of arithmetic terms connected by one of the operators "$=$", "$\neq$", "$\leq$", "$\geq$", "$>$" or "$<$".

The Omega Library, developed in course of the Omega Project [23] is one tool that can handle Presburger arithmetic. It uses the Omega Test [24], an extension of the Fourier-Motzkin variable elimination, which can find solutions for an arbitrary set of linear equations and inequations. But the Omega Library is additionally able to reason over formulas containing disjunctions, conjunctions, integer constraints, first-order quantification, subtraction, and multiplication with constants. Notably, it cannot reason over predicates, functions or division. According to the authors this maintains completeness and decidability.

This system was then used by Otten, Troelenberg and Raths to establish leanCoP-Ω. This new prover can handle arithmetic and (in-)equations that lie within the abilities of the Omega Library [23]. As the original connection calculus cannot handle arithmetic and (in-)equations, we must discuss how this extension was possible.

Originally, leanCoP-Ω accessed the Omega Library via a console call to the Omega Calculator command line tool and `grep` to evaluate the output. But the Omega Calculator was buggy, crashing with segmentation faults for some problems at unpredictable times, seemingly in the parser component. Those errors led to a failure return value which was interpreted by leanCoP as negative result. Also, communication via shell is quite slow compared to foreign function interfaces. For those reasons, two bachelor's students at the University of Potsdam were tasked with updating the hook. First, the library itself was updated [25]. This included reducing the codebase of the Omega Project to the part which is really necessary, i.e., the Omega Library and the Omega Calculator. Then, the hook between leanCoP and Omega was rewritten [26] by implementing foreign function interface functionality both in C++ on the Omega side and Prolog on the leanCoP side. Münch implemented the conversion of the constraint-formula to Omega-syntax in Prolog. New C++ methods then access the required parts of the Omega Library, and the hook is transformed into a shared object accessible to Prolog via its Foreign Language Interface. The new hook is reliable and has a speed-up linear in the number of Omega Library calls compared to the Omega Calculator.

2.3. Extending the Connection Calculus

The handling of arithmetic and arithmetic equations is done by leanCoP-Ω in a particular way: When transforming the input formula into a matrix, arithmetic expressions like $f(3) = f(x)^1$, $27+x < 6*y^0$ or $y \neq 4^0$ are treated as atomic terms that cannot be simplified further. The result is that each of these expressions (here also called "e-literals" or "constraints") is a literal within the generated matrix. Note that we refer to both e-literals within a matrix and (in-)equations within the original formula as constraints whenever we want to highlight that they constrain the possible variable evaluations. We can then prove each of these e-literals with rules by using the regular extension rule with a modified unification, or new e-literal specific rules. We will discuss both approaches in the following.

We discuss unification first. If we ever wish to unify two e-literals, e.g. when using the extension or reduction rule to prove an e-literal, or when unifying regular literals that contain arithmetic, the unification must be extended to evaluate those expressions. This is done by

comparing the syntax trees of the literals recursively while evaluating as many constant arithmetic sub-terms, such as $2 * 3 + 5$, as possible. Function, predicate and equality symbols must match exactly. The final, non-reducible to-be-unified pairs are stored as equations in normal form ($side_0 - side_1 = 0$). This equation is simplified again if possible and, finally, tested for validity. A single unification procedure may produce multiple of these equations if they include functions, predicates or equations with multiple parameters. For example, take the following matrix:

$$[|s(2 * a + b, (r + q) * q)^0| \quad |s(x + y, (2 + 1) * z)^1|]$$

(Sub-)matrices are denoted by square brackets and clauses by vertical bars. This matrix requires three equations for the unification to succeed, namely

$$\begin{aligned} 2 * a + b - x - y &= 0 \\ r + q - 3 &= 0 \\ q - z &= 0 \end{aligned} \tag{1}$$

This is because we assume Presburger arithmetic, which disallows non-linear arithmetic. With these equations, we can allow for unification in the rare case that the two sides of the would-be non-linear multiplication are equal for the two literals. The symbol s can be a function, a predicate, or an (in-)equality symbol. The variables are assumed to be existentially quantified.

Equations generated by the unification need to hold for the entire proof search and are tested repeatedly whenever variables are assigned new values. The exact configuration of these tests is discussed in the following paragraphs.

Certain ground e-literals such as $(1 < 2)^0$ are inherently valid (called self-fulfilling in the following) and can easily be verified without further work while respecting their polarity. This is the first rule that can only be applied to e-literals. In the non-ground case the e-literal is a constraint on the variable evaluation. Such a constraint is valid iff it is valid under all valid interpretations of the integer variables (respecting the relevant quantifiers) under the current substitution. This yields the second method for proving an e-literal: By constructing a formula including constraints based on the current path, and testing it for validity. This type of formula is called a "constraint-formula" in the following. It must additionally contain the constraints generated by the modified unification procedure, as well as the quantifiers of the occurring variables.

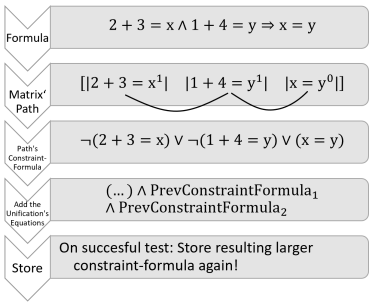

Figure 1: Proving a FOL-formula with arithmetic equations via the constraint-based method.

We will elaborate with an example (see also Figure 1): Say we wish to prove the formula $2 + 3 = x \wedge 1 + 4 = y \Rightarrow x = y$. Then the corresponding matrix will include each equation from the formula in the form of an e-literal, all being in α-relation to each other, i.e., they can be connected by the extension rules. The e-literal for $x = y$ is marked by polarity 0 and the others by polarity 1. A proof for one of the e-literals will then require all the e-literals to be on the same path. This can happen by applying the deep-omega extension rule explained in the next paragraph. We can then construct the constraint-formula by connecting all the e-literals on the path into one formula: $\neg(2 + 3 = x) \vee \neg(1 + 4 = y) \vee (x = y)$. It may appear counterintuitive to use disjunctions here, since we are collecting e-literals from the path. But the reader can easily ascertain that this is the correct representation of the original formula, which is due to the polarity 0 given to the original formula combined with De Morgan's laws. Additionally, we need to add previous constraint-formulas to this constraint-formula with a conjunction. Once we have added the correct quantifiers for all occurring variables, we can pass the formula to an arithmetic validity test. If the formula is valid, it must be stored, to be added to further constraint-formula validity tests in the future. E-literals on the path may have been added there by regular extension rules, but still need to be evaluated by the constraint test. Note that we assumed that the constraint-formula is passed to an external system. This may seem like we are delegating the problem, but the prover still provides the search strategy.

The final rule for proving an e-literal is the deep-omega extension rule. This is an alternative extension rule that is only applicable to e-literals. Its usage is necessary whenever a (sub-)proof is only possible by constructing a constraint-formula that contains multiple e-literals which would not usually be connected by the extension rule. The example matrix $\left[\left|z < x + 1^0\right| \quad \left|x < z^0\right|\right]$ illustrates this: A successful proof is only possible by having both constraints be contained in the same constraint-formula. But a regular connection will never be drawn since arithmetic reasoning is required for the proof. While attempting to prove the first e-literal, the deep-omega extension rule allows the proof procedure to draw a deep-omega extension to any other e-literal. In contrast to the regular extension rule, the connected-to literal must now still be proven, which in this example is possible by the constraint method.

3. Implementation

In the following, we will discuss the implementation of nanoCoP-Ω. Its creation was the focus of this work, took the most effort, and yielded the knowledge discussed in Subsection 2.3.

nanoCoP-Ω can read first-order logic formulas in typed TPTP syntax (v 8.0.0), i.e., TFA theorems from the Specialist Problem Class (SPC) TF0. It supports the types integer, boolean, individual, function, predicate and newly defined types. For an exact specification of these types check the TPTP syntax [27]. For integer-typed expressions, the arithmetic methods and rules (described above) are used. The other types are only matched during unification. Reading original leanCoP-syntax is not fully supported by nanoCoP-Ω, thus we encourage using only the TPTP syntax.

Types were not available in nanoCoP-2.0. Additionally, nanoCoP-Ω can discern certain formula types: namely the conjecture from the axioms. Once an internal representation has been parsed, the equality axioms reflexivity, symmetry, transitivity, and the substitution rules

for all occurring predicates and functions are added to the internal representation. Then a non-clausal matrix including the e-literals is generated. The implementation of the new proof methods is discussed in the following subsections. We added a feature that allows the tracking of their usage. The final already human readable proof output of nanoCoP was adjusted to show new explanations whenever arithmetic and constraint specific methods were used, as shown in the following listing.

```
[...]
1.1.1 Assume (29^[])^t($int)=(30^[])^t($int) is false.
      This constraint-formula (in combination with all other
         constraints from the path) is valid as proven with the
         Omega Test.
[...]
```

In the compact proof, the arithmetic related output contains a constraint-formula proven by the Omega Library as shown in the following listing.

```
[arithOmega,(- ((1^t($int)+4^t($int))^t($i)=(30^[])^t($int));-
    ((2^t($int)+3^t($int))^t($i)=(29^[])^t($int));(29^[])^t($int
    )=(30^[])^t($int))]
```

3.1. Implementing the New Methods

The new methods introduced in leanCoP-Ω were successfully imported to nanoCoP-Ω. The modified unification procedure is realised by the predicate `unify_with_arith(LitA,LitB,Eqs,Settings)`. It is used when applying the reduction and extension rules. It will do regular Prolog unification, not requiring any equations, unless encountering typed expressions. If the latter occurs, if one of the literals is simply a variable, it will attempt to evaluate the other side as far as possible and then do a regular unification. If both sides are complex arithmetic expressions, it will return all necessary equations, again evaluating simple arithmetic. It cannot return non-linear equations and will, when encountering non-linear arithmetic, instead recursively call the unification procedure to attempt to match the two sides of the non-linear term separately. The same occurs when the literals have predicate symbols as their topmost symbol. The unification predicate can return function symbols within its equations. We added a test to prevent omega calls for these cases, which is not done in leanCoP-Ω, causing the Omega Library to crash for these cases. Each call to the unification predicate is followed by a call of the `omega` predicate (see below).

The classic proof methods of the connection-based method as present in nanoCoP 2.0 [16] were adapted to use the modified unification procedure, followed by an `omega` call. The proof search was extended to store generated constraint-formulas. The predicate `leanari(Lit)` implements the test for self-fulfilling literals. If this fails, we attempt a constraint-based proof. This includes collecting the e-literals from the path with the `path_eq(Path,Lit,EqsPath)` predicate and calling the `omega(cFormula)` predicate on the returned constraint-formula conjoined with all previously constructed constraint-formulas. The `omega` predicate will convert the input into the Omega Library's syntax. It is the Omega Hook rewritten by Münch in [26]. The deep-omega

extension rule is the final rule attempted and is very close in its implementation to the regular extension rule, except that the connected to e-literal needs to still be proven. It is locked behind the setting `deep_omega`, formerly called `eq(2)` in leanCoP-Ω.

An important difference between nanoCoP 2.0 and nanoCoP-Ω lies in their respective start rules, the method by which the start clause is selected. In the classic connection-calculus it is a valid optimization to restrict the choice of the start clause to one that contains only positive literals. This is because, if no such clause exists, the matrix has a path containing only negative literals, which thus cannot contain any connections. This optimization is not valid for matrices containing e-literals, as e-literals can be proven with methods other than connections irrespective of their polarity. To combat this issue, we have implemented a second start rule that is called if finding a proof with the first fails: It interprets all e-literals as positive for the purpose of selecting a start clause, thus allowing the prover to select another start clause. This means that the prover will still prioritize selecting clauses that are purely positive, but can attempt a different angle if this fails. nanoCoP-Ω supports the optimizations present in nanoCoP 2.0 [16], such as restricted backtracking [20] and random reordered restarts. Like nanoCoP 2.0 it uses iterative deepening to ensure completeness.

3.2. Restrictions of nanoCoP-Ω

For clarity, let us now discuss the things that nanoCoP-Ω cannot do. Simply put, this includes everything that is not formalizable in Presburger arithmetic. Note that this restriction only applies to constraints after they have been simplified by Prolog: nanoCoP-Ω will succeed in proving $\forall A, B \in Int: \ p(A * B * B) \Rightarrow p(B * B * A)$, because the `unify_with_arith` predicate evaluates the parameter of p: $A * B * B - B * B * A = 0$ becomes $0 = 0$, a linear equation. On the other hand, nanoCoP-Ω cannot prove the formula $\forall X, Y, Z \in Int: \ Y = Z \ \wedge p(X, 3 + s(Y)) \Rightarrow p(X, 3 + s(Z))$, with p being an uninterpreted predicate and s being an uninterpreted function symbol. This is because it has no way of collecting the information $Y = Z$ before attempting to validate the equation $Y - Z = 0$ in order to unify the two literals containing p. The deep-omega extension rule is not applicable here, because the literals containing p are not e-literals. Perhaps merely a modification of the presented methods would suffice to find a proof here, or possibly a new method based on an "α-constraint-closure" needs to be developed.

nanoCoP-Ω may prove non-theorems when invoked on typed formulas containing real or rational numbers. This behaviour was deemed non-detrimental, as nanoCoP-Ω was solely created for formulas with integer arithmetic. For soundness testing, nanoCoP-Ω was tested on all applicable non-theorems from the integer TFA segment of the TPTP v8.0.0 and none of them was falsely proven.

4. Evaluation

nanoCoP-Ω was evaluated with a comparison to leanCoP-Ω, specifically the version of Behrens and Münch. Due to the similarity of the systems, this is also a direct comparison between leanCoP and nanoCoP. Evaluation was done on 1092 non-polymorphic typed first-order logic theorems (TFA) from the TPTP benchmark v8.0.0 [17]. The experimental setup was transferred

from Münch's work: each prover was given a maximum time of 60 seconds. To provide a more robust execution time, easy problems ($< 1s$) were run 100 times and harder problems 9 times. A first optimized strategy optimization was used, but some further strategies can be added and improvements can still be made. The strategy sequence that was used can be found in the nanocop_omega.sh shell script on the project page.

Table 1
Shared results of both provers.

		nanoCoP-Ω			
leanCoP-Ω		#theorem	#non-theorem	#time-out	#total
	#theorem	259	7	83	349
	#time-out	45	69	629	743
	#total	304	76	712	1092

According to the results shown in Table 1, nanoCoP-Ω marks 76 problems as non-theorems. In 43 of these cases the theorems used division or function symbols within arithmetic expressions and had evaded pruning. For these problems, leanCoP-Ω instead times out other than in four cases, in which it manages to find a false proof using literals with fractions in them. Five problems were solved by leanCoP-Ω but not nanoCoP-Ω because the deep-omega extension was not activated during much of nanoCoP-Ω's proof search. The rest of the problems marked as non-theorems by nanoCoP-Ω were most likely not provable without the deep-omega extension rule, although leanCoP-Ω not proving them within the time limit indicates that nanoCoP-Ω may also not have proved them. leanCoP-Ω crashes for at least 24 theorems, most likely due to calling the Omega Library on a constraint-formula including function symbols.

nanoCoP-Ω has a success rate of approximately 27.8%, while leanCoP-Ω has one of about 32.0%. Thus nanoCoP-Ω's is lower by approximately 4.2%, indeed being a downgrade in the number of solvable problems. But nanoCoP-Ω possibly would have been able to solve more problems if the deep-omega extension rule had been allocated more of its execution time.

Thus, we compare the cross-distribution of the outcomes of both systems in more detail, as shown in Table 1. The most interesting aspect is that leanCoP-Ω solves 90 problems that nanoCoP-Ω does not solve. nanoCoP-Ω, on the other hand, solves 45 problems that leanCoP-Ω does not solve. Obviously, both connection calculus implementations are differently well suited for proving different formulas.

The problems leanCoP-Ω solves over nanoCoP-Ω stem to 90% from the software verification domain. These problems are particularly large and were generated automatically. The other 10% stem from the interactive theorem prover, arithmetic, and number theory domains. The problems nanoCoP-Ω solves over leanCoP-Ω are to 60% from the software verification domain, with further problems from arithmetic, data structures, number theory and set theory.

In order to assess whether there is some specific hardness of domains for leanCoP-Ω or nanoCoP-Ω we compared the success rate domain-wise, as displayed in Table 2. Here, only domains with sufficiently many (40) problems within the benchmark are shown.

Both systems have a weak performance in the interactive theorem proving and hardware verification domains and identical performance in the arithmetic domain. In the most other

Table 2
Domain-wise success rates of the two provers, percentages rounded to one decimal.

Domain	nanoCoP-Ω	#nanoCoP-Ω	leanCoP-Ω	#leanCoP-Ω
ARI	55.3%	197	55.3%	197
DAT	28.1%	29	17.5%	18
HWV	0.0%	0	0.0%	0
ITP	6.8%	14	12.6%	26
NUM	45.2%	19	50.0%	21
SWW	13.6%	28	33.0%	68

domains, leanCoP-Ω performs better than nanoCoP-Ω. Merely in the data structures domain nanoCoP-Ω has a better performance. Nevertheless, both provers were successful for their most important use-case: proving arithmetic and number theory problems.

Since nanoCoP-Ω does not outperform leanCoP-Ω concerning the count of solved problems, we analysed the runtime performance of the two systems.

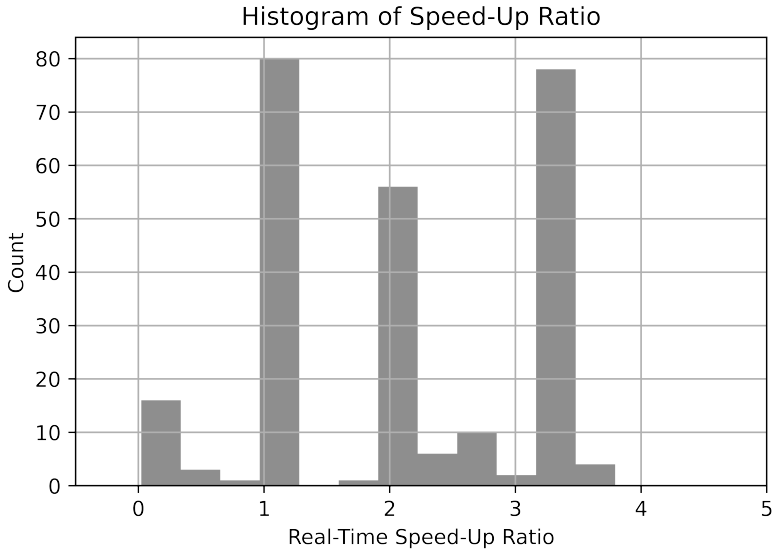

Figure 2: The speed-up ratio between the count of problems.

The distribution of real-time (wall-time) speed-up ratios is displayed in Figure 2. For each ratio r, the number of problems solved r times faster is drawn on the y-axis[1]. Here we see a real-time average relative speed-up of nanoCoP-Ω over leanCoP-Ω of 2.3, i.e. nanoCoP-Ω is approximately 2 times faster than leanCoP-Ω. We chose to analyse the real-time speed-up as the average user-time speed-up ratio lies at 12.63, a value vastly different from the real-time

[1]The figure does not include the problem $ARI081 = 1.p$ with a speed-up of 8, as well as $COM003\_1.p$ with a speed-up of 31, which both exceed the scale.

speed-up ratio and thus not deemed indicative. Also, nanoCoP-Ω is slower than leanCoP-Ω for only 20 of the 259 shared proofs. These are mainly theorems from the domains software verification and interactive theorem proving, as well as single theorems from the domains puzzles and set theory. Here, nanoCoP-Ω takes on average approximately 4.9 times as long to find a proof.

By doing a domain-wise speed-up comparison, as displayed in Table 3, we learn that nanoCoP-Ω is faster for theorems from the arithmetic and number theory domains and slower for those from software verification.

Table 3
Average speed-up by domain, restricted to domains with at least three theorems proven by both systems.

Domain	rel. speed-up	abs. speed-up in s	#problems
ARI	2.43	1.7	188
COM	8.72	8.2	4
DAT	1.54	1.2	17
ITP	0.67	-11.4	14
NUM	3.14	2.9	17
PUZ	0.96	-0.3	4
SWV	0.16	-10.1	2
SWW	0.31	-16.2	8

5. Future Work

In principle, the concept of leanCoP-Ω and nanoCoP-Ω should be transferable to nanoCoP-M and nanoCoP-i [16] by adopting the `prefix_unify` predicate. Additionally, the Omega Test is not the only decision procedure for Presburger arithmetic. For example, ARITH [28], SUP-INF [29, 30] and Cooper's [31] method are all suitable for Presburger arithmetic but differ in capabilities and performance. Thus, implementing them for nanoCoP-Ω and introducing a clever scheduling or heuristic could make more problems solvable. In fact, the ARITH procedure has already been implemented in Prolog by Troelenberg [32], but was never successfully integrated in leanCoP or nanoCoP. An extension of nanoCoP-Ω to real numbers with the algorithms provided in [33], [30] and [34] would be possible and desirable, too.

During evaluation we have learnt that the performance of leanCoP-Ω and nanoCoP-Ω differs depending on the domain and formula structure. For instance, both provers can solve problems which the other one cannot, which is consistent with the results for leanCoP and nanoCoP presented in [16]. Thus, a combination of leanCoP-Ω and nanoCoP-Ω has the potential to yield significantly better results. This could be done either by starting proof attempts in parallel by both systems, or first by nanoCoP-Ω and, on failure, by leanCoP-Ω.

The deep-omega extension rule was not active during much of nanoCoP-Ω's proof search. In future extensions, this should be one of the strategies used during the bulk of nanoCoP-Ω's execution time. Additionally, backtracking could be restricted upon a successful call to `unify_with_arith`, for example as an optional strategy similar to restricted backtracking.

This would reduce the count of calls to omega, which undoubtedly would improve the runtime performance. Another untested strategy lies in ommiting to add the equality axioms to the matrix. The transitivity axiom can easily be applied and thus potentially slow down the proof search. Moreover, the predicate `unify_with_arith` returns equations of the shape $a * X + b * Y + \cdots + c * Z + d = 0$. This normal form is enforced again before passing constraint-formulas to the Omega Library. An elimination of this redundant transformation could improve the runtime slightly. A potentially more powerful optimization again lies in the number of calls to the Omega Library: It appears that there are many redundant calls of the validity test with the exact same constraint-formulas. Sadly, extending the Omega Library to recognise this is hardly possible due to the complex project structure. But an additional module that caches previous constraint checks would give the possibility to detect and eliminate redundant calls. Even more, we neither fully optimised the strategy sequences for nanoCoP-Ω, nor test the optimization strategies proposed above, which are both subject to future work. Then, nanoCoP-Ω could be reasonably compared to other theorem provers for the TFA division.

6. Conclusion

We have successfully integrated arithmetic and arithmetic (in-)equation handling methods into nanoCoP-Ω. It has proven to be slightly worse at proving large theorems compared to leanCoP-Ω, but it is considerably faster for small problems, and overall equally well-suited for proving arithmetic and number theory theorems. Moreover, both systems prove problems that the other does not. The resulting system should provide readable proofs for use-cases in which these are required. We have presented ideas for combining nanoCoP-Ω and leanCoP-Ω, for optimizing and for extending nanoCoP-Ω. The source code is available on the nanoCoP-Ω project page[2].

References

[1] L. Kovács, A. Voronkov, First-Order Theorem Proving and Vampire, in: N. Sharygina, H. Veith (Eds.), Computer Aided Verification, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 1–35. doi:10.1007/978-3-642-39799-8_1.

[2] C. Barrett, C. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, C. Tinelli, CVC4, in: G. Gopalakrishnan, S. Qadeer (Eds.), Proceedings of the 23rd International Conference on Computer Aided Verification, number 6806 in Lecture Notes in Computer Science, Springer-Verlag, 2011, pp. 171–177.

[3] A. Duarte, K. Korovin, Implementing superposition in iprover (system description), in: N. Peltier, V. Sofronie-Stokkermans (Eds.), Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part II, volume 12167 of *Lecture Notes in Computer Science*, Springer, 2020, pp. 388–397. doi:10.1007/978-3-030-51054-1_24.

[4] P. Rümmer, A constraint sequent calculus for first-order logic with linear integer arithmetic,

[2] https://gitup.uni-potsdam.de/atp/nanocop-omega

in: Proceedings, 15th International Conference on Logic for Programming, Artificial Intelligence and Reasoning, volume 5330 of *LNCS*, Springer, 2008, pp. 274–289.

[5] R. Nieuwenhuis, A. Rubio, Paramodulation-Based Theorem Proving, in: A. Robinson, A. Voronkov (Eds.), Handbook of Automated Reasoning, North-Holland, Amsterdam, 2001, pp. 371–443. doi:10.1016/B978-044450813-3/50009-6.

[6] R. Nieuwenhuis, A. Oliveras, Proof-Producing Congruence Closure, in: J. Giesl (Ed.), Term Rewriting and Applications, Springer Berlin Heidelberg, Berlin, Heidelberg, 2005, pp. 453–468. doi:10.1007/978-3-540-32033-3_33.

[7] L. de Moura, N. Bjørner, Z3: An Efficient SMT Solver, in: C. R. Ramakrishnan, J. Rehof (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 337–340. doi:10.1007/978-3-540-78800-3_24.

[8] J. Otten, H. Trölenberg, T. Raths, leanCoP-Ω, 2010. URL: https://www.tptp.org/CASC/J5/leanCoP-Omega---0.1.pdf.

[9] G. Sutcliffe, The 5th IJCAR Automated Theorem Proving System Competition - CASC-J5, AI Commun. 24 (2011) 75–89.

[10] Y. Bertot, P. Castéran, Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions, Texts in Theoretical Computer Science, Springer Verlag, 2004. doi:10.1007/978-3-662-07964-5.

[11] M. Wenzel, The Isabelle/Isar Reference Manual, 2009. P. 162-164.

[12] J. Otten, A Non-clausal Connection Calculus, in: K. Brünnler, G. Metcalfe (Eds.), Automated Reasoning with Analytic Tableaux and Related Methods, Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 226–241. doi:10.1007/978-3-642-22119-4_18.

[13] J. Otten, nanoCoP: A Non-clausal Connection Prover, in: N. Olivetti, A. Tiwari (Eds.), Automated Reasoning - 8th International Joint Conference, IJCAR 2016, Coimbra, Portugal, June 27 - July 2, 2016, Proceedings, volume 9706 of *Lecture Notes in Computer Science*, Springer, 2016, pp. 302–312. doi:10.1007/978-3-319-40229-1_21.

[14] J. Otten, nanoCoP: Natural Non-clausal Theorem Proving, in: C. Sierra (Ed.), Proceedings of the 26th International Joint Conference on Artificial Intelligence, IJCAI'17, AAAI Press, 2017, pp. 4924–4928. doi:10.24963/ijcai.2017/695.

[15] L. Repp, Extending the automatic theorem prover nanoCoP with arithmetic procedures, BSc Thesis, Institute for Computer Science, University of Potsdam, Potsdam, Germany, 2023. doi:10.25932/publishup-57619.

[16] J. Otten, The nanoCoP 2.0 Connection Provers for Classical, Intuitionistic and Modal Logics, in: A. Das, S. Negri (Eds.), 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*, Springer-Verlag, Berlin, Heidelberg, 2021, pp. 236–249. doi:10.1007/978-3-030-86059-2_14.

[17] G. Sutcliffe, The TPTP Problem Library and Associated Infrastructure, J. Autom. Reason. 59 (2017) 483–502. doi:10.1007/s10817-017-9407-7.

[18] J. Otten, W. Bibel, leanCoP: lean connection-based theorem proving, Journal of Symbolic Computation 36 (2003) 139–161. doi:10.1016/S0747-7171(03)00037-3.

[19] J. 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, Berlin: Springer, 2008, pp. 283–291. doi:10.1007/978-3-540-71070-7_23.

[20] J. Otten, Restricting Backtracking in Connection Calculi, AI Communications 23 (2010) 159–182. doi:10.3233/AIC-2010-0464.

[21] L. C. Paulson, J. C. Blanchette, Three years of experience with sledgehammer, a practical link between automatic and interactive theorem provers, in: G. Sutcliffe, S. Schulz, E. Ternovska (Eds.), IWIL 2010. The 8th International Workshop on the Implementation of Logics, volume 2 of *EPiC Series in Computing*, EasyChair, 2012, pp. 1–11. doi:10.29007/36dt.

[22] M. Presburger, Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt., in: Comptes-Rendus du ler Congress des Mathematiciens des Pays Slavs, 1929, p. 92–101.

[23] W. Pugh, the entire Omega Project Team, The Omega Project: Frameworks and Algorithms for the Analysis and Transformation of Scientific Programs, 2013. URL: https://www.cs.umd.edu/projects/omega/.

[24] W. Pugh, The Omega Test: A Fast and Practical Integer Programming Algorithm for Dependence Analysis, in: Proceedings of the 1991 ACM/IEEE Conference on Supercomputing, Supercomputing '91, Association for Computing Machinery, New York, NY, USA, 1991, p. 4–13. doi:10.1145/125826.125848.

[25] A. B. Behrens, Modernisierung von Legacy Beweissystemen, BSc Thesis, Institute for Computer Science, University of Potsdam, Potsdam, Germany, 2019.

[26] K. Münch, Entwicklung einer performanten und verlässlichen Schnittstelle für leanCoP-Ω, BSc Thesis, Institute for Computer Science, University of Potsdam, Potsdam, Germany, 2021.

[27] G. Sutcliffe, TPTP Syntax, 2023. URL: https://tptp.org/TPTP/SyntaxBNF.html#tff_formula.

[28] T. Chan, An Algorithm for Checking PL/CV Arithmetic Inferences, Technical Report, Cornell University, USA, 1977. URL: https://dl.acm.org/doi/book/10.5555/867428.

[29] R. E. Shostak, On the SUP-INF method for proving presburger formulas, J. ACM 24 (1977) 529–543. doi:10.1145/322033.322034.

[30] R. Shostak, Deciding Linear Inequalities by Computing Loop Residues, J. ACM 28 (1981) 769–779. doi:10.1145/322276.322288.

[31] D. C. Cooper, Theorem Proving in Arithmetic without Multiplication, Machine intelligence 7 (1972) 300.

[32] H. Trölenberg, Arithmetik im Automatischen Theorembeweisen, Diploma Thesis, Institute for Computer Science, University of Potsdam, Potsdam, Germany, 2010.

[33] L. Hodes, Solving problems by formula manipulation in logic and linear inequalities, in: Proceedings of the 2nd International Joint Conference on Artificial Intelligence, 1971, pp. 553–559. doi:10.1016/0004-3702(72)90046-X.

[34] B. Boigelot, S. Jodogne, P. Wolper, An effective decision procedure for linear arithmetic over the integers and reals, ACM Transactions on Computational Logic (TOCL) 6 (2005) 614–633. doi:10.1145/1071596.1071601.