Automated Strategy Invention for Confluence of Term Rewrite Systems

Liao Zhang[1,2] , **Fabian Mitterwallner**[1] , **Jan Jakubův**[3] and **Cezary Kaliszyk**[4,1]
[1]University of Innsbruck
[2]Shanghai Jiao Tong University
[3]Czech Technical University in Prague
[4]University of Melbourne
zhangliao714@gmail.com, fabian.mitterwallner@uibk.ac.at, {jakubuv, cezarykaliszyk}@gmail.com

Abstract

Term rewriting plays a crucial role in software verification and compiler optimization. With dozens of highly parameterizable techniques developed to prove various system properties, automatic term rewriting tools work in an extensive parameter space. This complexity exceeds human capacity for parameter selection, motivating an investigation into automated strategy invention. In this paper, we focus on confluence of term rewrite systems, and apply AI techniques to invent strategies for automatic confluence proving. Moreover, we randomly generate a large dataset to analyze confluence for term rewrite systems. We improve the state-of-the-art automatic confluence prover CSI: When equipped with our invented strategies, it surpasses its human-designed strategies both on the augmented dataset and on the original human-created benchmark dataset ARI-COPS, proving/disproving the confluence of several term rewrite systems for which no automated proofs were known before.

1 Introduction

Term rewriting studies substituting subterms of a formula with other terms [Baader and Nipkow, 1998], playing an important role in automated reasoning [Bachmair and Ganzinger, 1994], software verification [Meseguer, 2003], and compiler optimization [Willsey *et al.*, 2021]. Mathematicians have developed various techniques to analyze the properties of term rewrite systems (TRSs). However, many properties are undecidable [Baader and Nipkow, 1998], implying that no technique can consistently prove a particular property. To navigate this undecidability, modern term rewriting provers typically employ complicated strategies, incorporating wide arrays of rewriting analysis techniques, with the hope that one will be effective. Each technique often accompanies several flags to control its behavior. The diversity of techniques and their controlling flags result in a vast parameter space for modern automated term rewriting provers.

Manually optimizing strategies for undecidable problems is beyond human capacity given the extensive parameter space. This inspires us to apply AI techniques to search for appropriate strategies automatically. In this paper, we focus on confluence, an important property of term rewriting, and discuss automated strategy invention for the state-of-the-art confluence prover CSI [Nagele *et al.*, 2017]. We modify Grackle [Hůla and Jakubův, 2022], an automatic tool to generate a strategy portfolio, encoding strategies that require transformations and complex schedules such as parallelism.

Directly using a tool like Grackle to randomly generate parameters for CSI may produce unsound results. This is a unique challenge compared to previous applications of Grackle [Hůla and Jakubův, 2022; Aleksandrova *et al.*, 2024]. The solvers to which Grackle was previously applied always produce sound results, while CSI's users need to carefully specify their strategies to ensure soundness.

We also augment the human-built confluence problems database (ARI-COPS)[1], a representative benchmark for the annual confluence competition (CoCo)[2]. Before 2024, CoCo used the COPS database as the benchmark. An unpublished duplicate checker is executed to remove duplicated problems in COPS, resulting in the ARI-COPS database, which is used in CoCo 2024. As ARI-COPS has been created manually, it includes only 566 TRSs. They are of high quality, but the relatively small number is still inadequate for data-driven AI techniques that require large amounts of training data. To handle this problem, we generate a large number of TRSs randomly, but ensure that they are interesting enough to analyze. For this, we develop a procedure to confirm a relative balance in the number of TRSs most quickly solved by different confluence analysis techniques within the dataset.

We evaluate our strategy invention approach in ARI-COPS and the augmented dataset. On both of the datasets, the invented strategies surpass CSI's competition strategy. In particular, we prove (non-)confluence for several TRSs that have not been proved by any automatic confluence provers in the history of the CoCo competition.

As an example, our invented strategy is able to disprove confluence for the ARI-COPS problem `846.ari` (`991.trs` in COPS), never proved by any participant in CoCo. The key is the application of the redundant rule technique [Nagele *et al.*, 2015] with non-standard arguments. CSI's competition strategy performs `redundant`

[1]https://ari-cops.uibk.ac.at/

[2]https://project-coco.uibk.ac.at/

`-narrowfwd -narrowbwd -size 7` prior to performing non-confluence analysis. The flags `narrowfwd` and `narrowbwd` determine the categories of redundant rules to generate. Our tool automatically discovered that by changing the original redundant rule transformation to `redundant -development 6 -size 7`, we can prove this problem. A larger value for the flag `development` causes a larger number of development redundant rules to be added. We notice that the value six is crucial as small values below three are ineffective for `846.ari`. This is only one of the several TRSs which our new strategies can solve as discussed in the later sections.

The main reason why it is difficult to discover new proofs in CoCo, is because CSI's competition strategy developed rewriting experts is very complicated, for which a comprehensive explanation is presented in the technical appendix. For example the competition strategy includes the development redundant rule technique [Nagele *et al.*, 2015]. The original evaluation of it shows no improvement over other redundant rule techniques in COPS at that time. Thus, CSI's developers decided not to use it in the competition strategy. As COPS grows, it becomes helpful in some new TRSs such as `846.ari`. However, the default strategy has only slightly changed over the past years, and the development redundant rule technique has never been tried. One reason for this could be that choosing sound parameters is challenging even for rewriting experts. Meanwhile, competition strategy is highly complicated and has a prohibitively large configuration space both in the number of parameters and structures of the strategy itself. We leverage Grackle to do the tedious strategy search. It can automatically optimize the strategies better than experts as the dataset grows. Other rewriting tools do not discover the proof perhaps because they do not implement the essential techniques for solving the problems.

Contributions. First, to our best knowledge, our work is the first application of AI techniques to automatic confluence provers. We automatically generate a lot of strategies for the state-of-the-art confluence prover CSI and combine them as a unified strategy. Second, we carefully design the parameter search space for CSI to confirm the soundness of strategy invention. Third, we build a large dataset for confluence analysis, comprising randomly generated TRSs and problems in the ARI-COPS dataset. Finally, empirical results show that our strategy invention approach surpasses CSI's competition strategy both in ARI-COPS and the augmented datasets. Notably, we discover several proofs for (non-)confluence that have never been discovered by any automatic confluence provers in the annual confluence competition.

2 Background

2.1 Term Rewriting

We informally define some theoretical properties of term rewriting in this section, hoping to ease the understanding of the behavior underlining automatic confluence provers. A formal description can be found in the technical appendix.

We assume a disjoint set of *variable* symbols and a finite signature of *function* symbols. *Constants* are function symbols with zero arity. The set of *terms* is built up from variables and function symbols. The set of variables occurring in a term t is denoted by $Var(t)$. A term rewrite system (TRS) consists of a set of rewrite rules $l \to r$ where $l, r \in terms$, $l \notin variables$, and $Var(r) \subseteq Var(l)$. We write $t_1 \to^* t_n$ to denote $t_1 \to t_2 \to \dots \to t_n$ where n can be one. A TRS is *confluent* if and only if $\forall s, t, u \in terms(s \to^* t \wedge s \to^* u \Rightarrow \exists v \in terms(t \to^* v \wedge u \to^* v))$. Consider the TRS of $\{f(g(x), h(x)) \to a, g(b) \to d, h(c) \to d\}$ [Gramlich, 1996]. It is not confluent since $f(d, h(b)) \leftarrow f(g(b), h(b)) \to a$, and no rules are applicable to $f(d, h(b))$ and a. A rewrite rule $l \to r$ is called *left-linear* if no variable occurs multiple times in l. A TRS is called left-linear if all its rules are left-linear. Left-linearity is crucial for confluence analysis since most existing confluence techniques only apply to such systems. In this paper, a term is called *complex* if it is neither a variable nor a constant.

2.2 CSI

CSI is one of the state-of-the-art automatic confluence provers that participates in CoCo. It ranked first in five categories of competitions in CoCo 2024. To show (non-)confluence of TRSs, CSI automatically executes a range of techniques, scheduled by a complicated configuration document written by experts in confluence analysis. Subsequently, CSI either outputs `YES`, `NO`, or `MAYBE` indicating confluence, non-confluence, or indetermination, respectively.

CSI implements many techniques applicable to the analysis of TRSs (many of them parametrized or transforming the system into one that can be analyzed by other techniques) and utilizes a complicated strategy language to control them. In CSI, these techniques are called *processors*. They are designed to prove the properties of TRSs, perform various transformations, and check the satisfiability of certain conditions. The strategy language can flexibly combine the execution of processors such as specifying parallel or sequential applications, disregarding unexpected results, assigning time limits, and designating repeated applications. The details of the strategy language are presented in the technical appendix.

Since the generated proofs are almost always large and difficult to check manually, CSI relies on an external certifier CeTA [Thiemann and Sternagel, 2009] to verify its proofs. To utilize CeTA, CSI outputs a certificate of its proof in the certification problem format [Sternagel and Thiemann, 2014]. Given a certificate, CeTA will either answer `CERTIFIED` or present a reason to reject it. Not all processors implemented in CSI are verifiable because CSI cannot produce certificates for all processors, and CeTA does not implement the verification procedures for all processors.

2.3 Grackle

Grackle [Hůla and Jakubův, 2022] is a strategy optimization system designed to automate the generation of various effective strategies for a given solver based on benchmark problems. Such solvers receive a problem and decide the satisfiability of a particular property of the problem. It was originally designed for automated reasoning tools and has been applied to various provers such as Prover9 [McCune, 2005] and Lash [Brown and Kaliszyk, 2022]. We choose Grackle for our research, as it is highly adaptable and we are not aware of

Algorithm 1 GrackleLoop: an outline of the strategy portfolio invention loop.

Input: initial strategies $\mathcal{S}$, benchmark problems $\mathcal{P}$, hyperparameters β
Output: a strategy portfolio Φ

$\Phi_{strat} \leftarrow \mathcal{S}$
while termination criteria is not satisfied **do**
  Evaluate$(\mathcal{P}, \Phi, \beta)$
  $\Phi_{cur} \leftarrow$ Reduce$(\mathcal{P}, \Phi, \beta)$
  $s \leftarrow$ Select$(\mathcal{P}, \Phi, \beta)$
  if s is None **then return** Φ
  $s_0 \leftarrow$ Specialize$(s, \mathcal{P}, \Phi, \beta)$
  $\Phi_{strat} \leftarrow \Phi_{strat} \cup s_0$
end while

any strategy invention program that would allow the kinds of strategies needed for automatic rewriting tools. Additionally, Grackle has achieved good results with the solvers it was previously applied to. The strategy invention problem of Grackle is formally defined below.

Definition 1 (Strategy Invention Problem)**.** *Assume a set of initial strategies $\mathcal{S}$. In the benchmark of examples $\mathcal{P}$, the problem is to invent a bounded set of complementary strategies $\mathcal{S}'$ that can prove the largest number of problems in $\mathcal{P}$. Complementary strategies means that $\forall s'_i \in \mathcal{S}'$, s'_i should master a subset of problems $\mathcal{P}'_i \subseteq \mathcal{P}$, such that $\forall i \neq j$, $s'_j \in \mathcal{S}'$ cannot solve any problem in $\mathcal{P}'_i$ quicker than s'_i.*

Algorithm 1 outlines the strategy portfolio invention loop of Grackle, which invents strategies via a genetic algorithm and parameter tuning with randomness. The variable Φ denotes the current state, including information like all invented strategies Φ_{strat}, and the current generation of strategies Φ_{cur}. The first phase is *generation evaluation (evaluate)*. In this phase, Grackle evaluates all strategies Φ_{strat} in its portfolio on the benchmark $\mathcal{P}$. The evaluation results are stored in Φ to avoid duplicated execution.

Next, Grackle performs *generation reduction (reduce)*. It assigns scores to every strategy in Φ_{strat} based on the evaluation results in the previous phase. A configurable number of strategies with the highest scores becomes the current generation of strategies Φ_{cur}.

The third phase is *strategy selection (select)*. It selects a strategy s from the current generation of strategies Φ_{cur} based on certain criteria, which is then used to invent new strategies. If no strategy can be selected, the algorithm terminates.

Finally, *strategy specialization (specialize)* invents a new strategy s_0 via specializing s over its best-performing problems $\mathcal{P}_s$ in $\mathcal{P}$. Grackle then executes external parameter tuning programs such as ParamILS [Hutter *et al.*, 2009] or SMAC3 [Lindauer *et al.*, 2022], tuning parameters for the selected strategy s with randomness. The goal is to invent a new strategy s_0 such that it performs better than s ion $\mathcal{P}_s$. The new strategy s_0 will be added to the portfolio Φ.

Grackle employs the same approach to describe its parameter search space as ParamILS. The space is described by a set of available parameters, each of which is associated with a default value and several disjoint potential values. Grackle users need to input the potential values based on their domain-specific experiences on the particular solvers. We refer to [Hůla and Jakubův, 2022] for a comprehensive explanation of Grackle.

3 Strategy Invention and Combination

To generate a better strategy for CSI, we first invent a large set of complementary strategies, and then appropriately combine a subset of the invented strategies into a single strategy.

3.1 Strategy Invention

To find new strategies for CSI, we first need to represent the parameter space in a meaningful way. The parameter space needs to be designed with precision to guarantee soundness.

There are three reasons why CSI may produce unsound results given an entirely random strategy. First, some processors are not intended for confluence analysis. They may intend to prove other properties of TRSs, such as termination [Baader and Nipkow, 1998]. Second, even for the same processor, it may be designed to prove different properties of TRSs with different flags. Third, some transformation processors may change the goal of CSI to prove another property of TRSs, which is different from confluence such as relative termination [Zantema, 2004].

We separate CSI's competition strategy into 23 sub-strategies, which, along with CSI's competition strategy, also serve as the initial strategies for Grackle. Among the 23 sub-strategies, nine are mainly used to show confluence, and 14 are used to show non-confluence. A comprehensive explanation of the division is shown in the technical appendix.

We maintain the structure used in CSI's competition strategy during the strategy invention because CSI relies on certain combinations of processors to (dis)prove confluence. There are papers proving theorems for confluence analysis, stating that if some properties of a TRS can be proved, then it is (non-)confluent. Such a theorem can be implemented as a single processor, which checks whether the given TRS satisfies the properties required by the theorem. However, not all such theorems are implemented as a processor. To utilize such theorems, we need to combine CSI's strategy language and processors to perform transformations on the original TRS and prove the necessary properties of the transformed problem. If we generate strategies randomly, it will be difficult to generate such useful structures and may produce unsound strategies due to inappropriate transformations.

We search for three categories of parameters. First, we search for *processor flags* which do not violate the soundness guarantee. For instance, `-development 6` in Section 1 is a processor flag for the `redundant` processor. To ensure soundness, we only search for flags of processors existing in CSI's competition strategy. Second, we include *iteration parameters*, such as time limits or repeated numbers of execution, to regulate the running of a certain sub-strategy. These parameters are defined in CSI's strategy language. Moreover, we add a *boolean execution-controlling parameter* for some parallel or sequential executed sub-strategies, indicating whether to run the particular sub-strategies in confluence analysis. Assume a strategy `A||B`, where `||` denotes a parallel execution. The boolean parameters for `A` and `B` can represent whether to run one, both, or neither of them.

We need to construct a strategy for CSI using the parameters searched by Grackle. To achieve this, we start with CSI's competition strategy, replacing the processor flags and iteration parameters with relevant invented parameters. Then, we disable sub-strategies according to the boolean execution-controlling parameters.

The most challenging part of our work is the proper definition of the parameter space to confirm CSI's soundness. As the exact definition is quite technical and verbose, we present the explanation of the parameter space and show an invented strategy in the technical appendix.

3.2 Strategy Combination

After inventing several complementary strategies, we want to combine them into a single strategy and compare it with the competition strategy of CSI. The combination is performed by choosing a few strategies from Grackle's final portfolio and appropriately assigning a time limit to each of them.

To effectively divide the time, we split the whole one minute into several time splits. Next, we greedily allocate a strategy to each time split in the sequence by order. Each newly chosen strategy aims at proving the largest number of remaining benchmark problems that have not been proved by the previously chosen strategies. We shuffle the sequence 100 times and greedily select strategies for each shuffled sequence, resulting in strategy schedules comprising sequences of pairs of strategies and time splits. To use a strategy schedule, CSI executes each strategy in it by order for a duration of the relevant time split. We split the one-minute duration into many sequences and perform the greedy strategy selection for each. We finally choose the strategy schedule that maximizes the number of provable problems. The details of the strategy combination are explained in the technical appendix.

4 Dataset Augmentation

Although ARI-COPS is meticulously built by term rewriting experts, it is unsuitable for AI techniques. First, it is relatively small which is insufficient for contemporary AI techniques. Second, there may be an imbalance in ARI-COPS because the problems come from rewriting literature. The examples are often of theoretical interest and are constructed to illustrate specific confluence analysis techniques. However, TRSs encountered in practical applications can contain redundant rules that are irrelevant to illustrating a certain property.

4.1 TRS Generation Procedure

We develop a program to randomly generate a large dataset of TRSs, receiving multiple parameters to control the overall generation procedure. First, the maximum number of available function symbols F, constants C, variables V, and rules R establish the upper bound of the respective quantities of symbols and rules. For each of F, C, and V, a value is randomly selected between zero and the specified maximum, determining the actual number of available symbols. The actual number of rules is randomly chosen between one and R. Second, we define a parameter M, used during the initialization of function symbols. For each function symbol, an arity is randomly chosen between one and M

Another important parameter is the probability of generating a left-linear TRS L, which is associated with the likelihood of producing provably confluent TRSs. The majority of contemporary techniques for proving confluence are merely effective for left-linear TRSs. Without regulating the ratios of left-linearity, randomly generated TRSs rarely exhibit left-linearity, making it theoretically difficult to show confluence for them. We also notice that, in practice, CSI can merely prove confluence of very few generated TRSs if the ratios of left-linearity are not controlled. By default, we force 60% of generated TRSs to be left-linear.

Moreover, for a rule $l \to r$, there is a parameter called CT related to the probability of generating l and r that are complex terms. We need it because we prefer complex terms, whereas constants and variables are quite simple.

Algorithm 2 presents the generation procedure of a single term. While choosing the root symbol, we first randomly sample a value between zero and one and compare it with $comp$ to determine whether to only use $funs$ as candidates for the root symbol. Here, $comp$ is a value randomly chosen between zero and CT during the initialization stage of the generation of a TRS. If the $comp$ is larger than one, we can only generate complex terms. Meanwhile, according to the definition of rewrite rules in Section 2.1, the left term l in $l \to r$ cannot be a variable. After choosing a root symbol for the term t, we continuously choose new symbols for undefined function arguments until all of them are defined. After selecting a new variable, we need to remove it from the set of available variables if we are generating a left-linear TRS. The size of the terms generated by us is at most 15, where the *size* of a term is defined as the number of symbols in it. We choose 15 as the maximum value because the sizes of most terms in ARI-COPS are smaller than 15.

To generate a rule $l \to r$, we first execute Algorithm 2 to generate l and then execute it again to generate r. We extract all used variables in l and mark them as available variables for the generation of r, thereby $Var(r) \subseteq Var(l)$, as required by the definition of rewrite rules in Section 2.1.

We repeatedly generate rewrite rules until they reach the expected number and then return the newly generated TRS.

4.2 Dataset Generation

We utilize the program explained in this section to construct a large dataset, facilitating the application of AI techniques to confluence analysis. First, we randomly generate 100,000 TRSs with the parameters of the maximum number of available function symbols $F = 12$, constants $C = 5$, variables $V = 8$, and rules $R = 15$. Other parameters include the maximum arity of function symbols $M = 8$, the probability of generating left-linear TRSs $L = 0.6$, and the value related to the possibility of generating complex terms $CT = 1.6$.

However, the randomly generated dataset can be imbalanced. First, there may be significant differences in the number of confluent, non-confluent, and indeterminate TRSs. Second, the number of TRSs mastered by different confluence analysis techniques may vary considerably.

Algorithm 2 Term Generation

Input: $consts, vars, funs$
$comp$, the likelihood of making a complex term
$left$, whether the term is on the rewrite rule's left side
$linear$, whether to construct a linear term
Output: a term t

if $random(0, 1) < comp$ **then**
  $root\_symbols \leftarrow funs$
else if left **then**
  $root\_symbols \leftarrow funs + consts$
else
  $root\_symbols \leftarrow funs + consts + vars$
end if
$t \leftarrow random\_choose\_one(root\_symbols)$
$undefs \leftarrow$ undefined function arguments in t
while $undefs$ is not empty **do**
  for all $undef \in undefs$ **do**
    $sym \leftarrow random\_choose\_one(funs + consts + vars)$
    replace the undefined function argument corresponding to $undef$ in t with sym
    if $linear$ and $is\_var(sym)$ and $left$ **then**
      remove sym from $vars$
    end if
  end for
  $undefs \leftarrow$ undefined function arguments of t
end while
return t

We develop a multi-step procedure to build a relatively balanced dataset. First, we execute CSI's competition strategy on all generated TRSs for one minute using a single CPU. CSI outputs `NO`, `YES`, and `MAYBE` for 69317, 25012, and 5671 TRSs, respectively.

Second, we randomly choose 5,000 problems from each set of problems classified as `NO`, `YES`, and `MAYBE` by CSI.

Third, we execute the duplicate checker used in CoCo 2024 to remove the duplications in the 15,000 chosen TRSs and 566 ARI-COPS TRSs. It checks the equivalence of syntactical structures between TRSs modulo renaming of variables and a special renaming on function symbols of their signatures. If TRSs of an equivalence class occur both in the randomly generated dataset and ARI-COPS, we only remove those randomly generated TRSs.

Fourth, we want to mitigate the imbalance in the number of problems mastered by different confluence techniques. We execute 26 strategies for all TRSs, aiming at labeling each of them with the most effective strategy. The labeling strategies contain all initial strategies for Grackle, which are explained in Section 3.1. The other two that are used to prove confluence are extracted from two complicated initial strategies, both consisting of many sub-strategies and integrated with transformation techniques that potentially simplify the search for proofs. Specifically, the two complicated initial strategies parallelly execute two important confluence analysis techniques, development closedness [Van Oostrom, 1997] and decreasing diagrams [Van Oostrom, 1994], not used by the other initial sub-strategies. If we do not use them for labeling, we will not be able to understand whether a TRS is mastered by one of the two important confluence analysis techniques. The details of the two new labeling strategies are explained in the technical appendix. The time limit for using CSI's competition strategy as a labeling strategy is one minute. The time limit for other labeling strategies is 30 seconds, smaller than one minute because the execution of decomposed sub-strategies is more efficient. We calculate the number of problems most quickly solved by each labeling strategy. The details of labeling strategies are presented in the technical appendix. The randomly generated dataset is quite imbalanced, four strategies master more than 1,000 problems; however, 16 strategies master less than 250 problems. To address the imbalance, we randomly choose at most 300 problems for a strategy from its set of mastered problems. We also randomly add 1,200 problems that cannot be solved by any labeling strategy to the dataset.

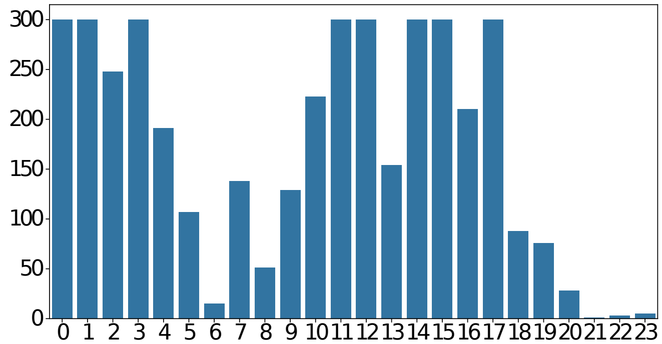

Figure 1: The number of TRSs solved most quickly (y-axis) for each labeling strategy (x-axis). Two labeling strategies that do not master any problems are ignored in the x-axis.

Finally, we obtain a dataset of 5,267 TRSs. Within this dataset, 1,647 TRSs are classified as confluent, 1,910 as non-confluent, and 1,710 as indeterminate when evaluated by CSI using a single CPU within a one-minute time limit.

Figure 1 shows the final distribution of the number of problems mastered by each labeling strategy. It is not perfectly balanced; however, we consider it relatively balanced, given that certain strategies can only master problems that satisfy particular properties. Such properties can be uncommon in randomly generated TRSs and practical applications.

There are infinitely many strategies that can be chosen as labeling strategies, such as strategies obtained by changing processor flags. We do not choose other labeling strategies as we have already decomposed CSI's competition strategy, enabling us to label problems with all categories of confluence analysis techniques implemented in CSI. Further decomposition or modification of processor flags may allocate problems to different labeling strategies that only slightly differ.

5 Experiments

We evaluate our strategy invention method on ARI-COPS and a combination of the randomly generated TRSs and ARI-COPS datasets. In both datasets, CSI with invented strategies outperforms CSI with the competition strategy, the state-of-the-art approach in confluence analysis for TRSs.

	ARI-COPS		augment	
CPU	1	4	1	4
init	475	477	846	852
total	479	484	873	871
confs	73	93	92	104
both in final	6	2	22	12

Table 1: Statistics of Grackle's training procedure. The rows *init* and *total* denote the number of problems solved by Grackle's initial strategy and the number of problems solved by strategies in Grackle's final portfolio, respectively. The row *confs* denotes the number of strategies that remains in Grackle's final portfolio. The row *both in final* represents the number of strategies in the final portfolio that master both confluence and non-confluence of TRSs.

5.1 Experimental Settings

The ARI-COPS 2024 dataset comprises a total of 1,613 problems of which 566 are TRS problems. We focus on evaluating our approach on TRS problems since they are standard term rewriting problems for confluence analysis and represent the major category in ARI-COPS. Another evaluation dataset consists of data from both ARI-COPS and our randomly generated datasets in Section 4.2. For training purposes, we arbitrarily select 283 examples from ARI-COPS and 800 examples from the randomly generated dataset. To build the test dataset, we exclude the examples in the training dataset, subsequently randomly selecting 800 examples from the randomly generated dataset and the remaining 283 examples from ARI-COPS.

The Grackle time limit for proving a TRS is 30 seconds, employed both in the evaluation and the strategy specialization phases. During the specialization phase, Grackle launches ParamILS for parameter tuning. The overall time limit for one strategy specialization phase is 45 minutes. The total execution time of Grackle is two days. Grackle performs parallel execution in both the evaluation and specialization phases; thus, we also limit the number of CPUs it can use. For each dataset, we perform two Grackle runs, configuring the numbers of available CPUs for a single strategy run to be either one or four. When it is set to one and four, the total number of available CPUs for Grackle is set to 52 and 66, respectively. Here, a CPU denotes a core of the AMD EPYC 7513 32-core processor. Grackle's portfolio stores at most 200 of the best strategies.

The use of four CPUs has been selected to match the results of CSI's competition strategy in CoCo 2024 on the competition setup. Given exactly the same problems solved by CSI in our own setup described above with four CPUs and in the CoCo competition in their Starexec [Stump *et al.*, 2014] setup we consider the further comparisons in the paper fair.

5.2 Experimental Results

Performance on ARI-COPS. Table 1 depicts the statistics of Grackle's training procedure. The value *total* shows the number of solved TRSs after the training, while *init* is the number solved by the initial strategies. When using four CPUs, Grackle's final portfolio contains more strategies than those in the final portfolio generated using one CPU. A probable reason is that executing with four CPUs can discover some strategies that are only effective with enough computation resources. The final augmented portfolios contain more strategies that master both confluence and non-confluence of TRSs. The likely reason is that a larger dataset makes training slower, and it is more difficult for Grackle to find optimal strategies for particular theoretical properties of TRSs.

	comp		total		combine		CoCo
CPU	1	4	1	4	1	4	
yes	266	272	271	277	271	276	272
no	203	205	208	207	207	207	205
solved	469	477	479	484	478	483	477

Table 2: Numbers of solved TRSs on ARI-COPS. The column *comp* represents CSI's competition strategy, *total* shows the total number of problems proved by all invented strategies, and *combine* denotes combining invented strategies as a single strategy. CoCo denotes the results obtained by CSI in CoCo 2024.

	never by CSI			never in CoCo		
CPU	yes	no	solved	yes	no	solved
1	2	3	5	1	3	4
4	4	2	6	1	2	3
1&4	6	3	9	2	3	5
1-CeTA	0	3	3	0	3	3
4-CeTA	1	0	1	0	0	0
1&4-CeTA	1	3	4	0	3	3

Table 3: Numbers of TRSs solved by all strategies in Grackle's final portfolio that have never been solved by all versions of CSI or any tool in CoCo. The suffix CeTA denotes the proofs can be certified by CeTA. The notion 1&4 means the union of all strategies invented by employing one CPU and four CPUs per strategy execution.

Table 2 compares the invented strategies with CSI's competition strategy. With a single CPU per each strategy evaluation, Grackle's final portfolio proves ten more problems than CSI's competition strategy. With four CPUs, *total* proves seven more problems than *comp*.

The invented strategies additionally (dis)prove several TRSs that have never been proved by different versions of CSI or all CoCo's participants, as depicted in Table 3. In total, we show (non-)confluence for nine TRSs that could not be solved by any versions of CSI. Five of the nine new proofs have never been proven by all CoCo's participants.

We combine the invented strategies as a single strategy to compare it with CSI's competition strategy. The number of time splits and the exact time assigned for each invented strategy are presented in the technical appendix. With single and four CPUs, *combine* proves nine and six more problems than the competition strategy, respectively.

When using one CPU, we gain more improvements over CSI's competition strategy compared to using four CPUs. A likely reason is that our strategy invention approach is particularly good at generating efficient strategies. With four CPUs, CSI can run several processors in parallelly, effectively reducing the runtime.

Certification. First, we check whether the answers found by the invented strategies are consistent with the answers discovered in CoCo. Second, we execute CeTA to verify the

	comp		combine	
CPU	1	4	1	4
yes	403	412	412	418
no	399	442	450	449
solved	802	854	862	867

Table 4: Numbers of solved TRSs on the testing examples of the augmented dataset.

proofs for the newly solved problems. Table 3 depicts the number of newly solved problems certifiable by CeTA. If we cannot certify the proofs due to the limitation of CeTA and CSI as explained in Section 2.2, we analyze the related strategies. We aim to understand what changes they perform to the original strategy lead to the proofs. From the analysis, we either slightly modify the sub-strategy defined in the competition strategy or directly use some existing sub-strategies to produce the same answers as the invented strategies. These modifications that lead to the answers are employed in the corresponding invented strategies, which are small and sound according to our knowledge of term rewriting. We also check the certification errors output by CeTA to figure out whether they are indeed errors or just caused by limitations of CSI and CeTA. Third, for each strategy in Grackle's final portfolio, we run CSI on its mastered problems and apply CeTA to verify the proofs. Only 234 and 226 proofs can be verified when one and four CPUs are employed for strategy invention, respectively. We manually check the proofs that cannot be verified by CeTA. The details of our certification procedures are shown in the technical appendix.

Performance on the augmented dataset. Table 1 also summarizes Grackle's training procedure in the augmented dataset. Compared to the training in ARI-COPS, Grackle's final portfolios consist of more strategies. The likely reason is that the augmentation dataset comprises more examples, necessitating more diverse strategies to cover them. We notice that with one CPU, the invented strategies prove more problems than those invented with four CPUs. This is probably caused by the randomness in the strategy invention.

The results of the evaluation in the test dataset are presented in Table 4. With one and four CPUs, *combine* respectively proves 60 and 13 more problems than *comp*. Notice that here the training examples are disjoint from the testing examples, whereas in the evaluation for ARI-COPS, they are the same. From this, we can conclude that our invented strategies generalize well to unseen data. With four CPUs, the unified strategy proves more problems than using one CPU. The likely reason is that the invented strategies with four CPUs can discover proofs more quickly, leading to a stronger unified strategy within the one-minute time limit.

6 Examples

Besides the example in Section 1, we present two more examples of the invented strategies that (dis)prove problems unprovable by any participant in CoCo.

The core structure of the first example is `AT`. It proves confluence for `794.ari` in ARI-COPS (`939.trs` in COPS). The sub-strategy `AT`, denoting Aoto-Toyama criteria [Aoto and Toyama, 2012], is defined in CSI's competition configuration document. CSI's competition strategy executes `AT` in parallel with many other sub-strategies, reducing the computational resources allocated to it and failing to find a proof.

Another example is similar to that in Section 1, we discover that if CSI employs `redundant -development 6` to generate redundant rules in the competition strategy, it can disprove confluence for `852.ari` (`997.trs` in COPS), and the proof can be certified by CeTA.

7 Related Work

There have been several attempts to apply machine learning to rewriting; however, none have been applied to automatic confluence provers. While [Winkler and Moser, 2019] investigate feature characterization of term rewrite systems, they do not build any learning models based on the features. There are works analyzing the termination of programs using neural networks to learn from the execution traces of the program [Giacobbe *et al.*, 2022; Abate *et al.*, 2021]. Nevertheless, they do not transform programs to term rewrite systems and apply machine learning to guide automatic term rewriting tools in termination analysis. MCTS-GEB [He *et al.*, 2023] applies reinforcement learning to build equivalence graphs for E-graph rewriting, but it focuses on optimization problems, not on confluence.

There has been extensive research on parameter tuning and strategy portfolio optimization in automated reasoning. Hydra [Xu *et al.*, 2010] employs a boosting algorithm [Freund and Schapire, 1997] to select complementary strategies for SAT solvers. [Ramírez *et al.*, 2016] propose an evolutionary algorithm for strategy generation in the SMT solver Z3 [De Moura and Bjørner, 2008]. A comprehensive review of these approaches is provided by [Kerschke *et al.*, 2019].

8 Conclusion and Future Work

We have proposed an approach to automatically invent strategies for the state-of-the-art confluence analysis prover CSI. We have performed data augmentation by randomly generating a large number of term rewrite systems and mixing these with the human-built dataset ARI-COPS. We have evaluated the invented combined strategy both on the original ARI-COPS dataset and the augmented dataset. The invented strategies discover significantly more proofs than CSI's competition strategy on both datasets. Notably, five of the human-written problems have never been proved by any automatic confluence provers in the annual confluence competitions.

Future work includes applying machine learning to individual term-rewriting techniques, for example those that perform search in a large space. Prioritizing the more promising parts of the search space could improve the individual techniques. Our strategy invention approach could also be extended to other automatic term rewriting provers. It would also be possible to apply neural networks to directly predict appropriate strategies for automatic term rewriting tools, however, soundness of proofs generated using such an approach remains a major challenge.

Acknowledgements

This research was supported by the ERC PoC project *Formal-Web3* no. 101156734, the University of Innsbruck doctoral scholarship *promotion of young talent*, the National Natural Science Foundation of China 92370201, and the Czech Science Foundation project no. 24-12759S.

References

[Abate *et al.*, 2021] Alessandro Abate, Mirco Giacobbe, and Diptarko Roy. Learning probabilistic termination proofs. In *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part II 33*, pages 3–26. Springer, 2021.

[Aleksandrova *et al.*, 2024] Kristina Aleksandrova, Jan Jakubuv, and Cezary Kaliszyk. Prover9 unleashed: Automated configuration for enhanced proof discovery. In *Proceedings of 25th Conference on Logic for Pro*, volume 100, pages 360–369, 2024.

[Aoto and Toyama, 2012] Takahito Aoto and Yoshihito Toyama. A reduction-preserving completion for proving confluence of non-terminating term rewriting systems. *Logical Methods in Computer Science*, 8, 2012.

[Baader and Nipkow, 1998] Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge university press, 1998.

[Bachmair and Ganzinger, 1994] Leo Bachmair and Harald Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *Journal of Logic and Computation*, 4(3):217–247, 1994.

[Brown and Kaliszyk, 2022] Chad E Brown and Cezary Kaliszyk. Lash 1.0 (system description). In *International Joint Conference on Automated Reasoning*, pages 350–358. Springer, 2022.

[De Moura and Bjørner, 2008] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[Freund and Schapire, 1997] Yoav Freund and Robert E Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of computer and system sciences*, 55(1):119–139, 1997.

[Giacobbe *et al.*, 2022] Mirco Giacobbe, Daniel Kroening, and Julian Parsert. Neural termination analysis. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 633–645, 2022.

[Gramlich, 1996] Bernhard Gramlich. Confluence without termination via parallel critical pairs. In *Colloquium on Trees in Algebra and Programming*, pages 211–225. Springer, 1996.

[He *et al.*, 2023] Guoliang He, Zak Singh, and Eiko Yoneki. Mcts-geb: Monte carlo tree search is a good e-graph builder. In *Proceedings of the 3rd Workshop on Machine Learning and Systems*, pages 26–33, 2023.

[Hůla and Jakubův, 2022] Jan Hůla and Jakubův. Targeted configuration of an smt solver. In *International Conference on Intelligent Computer Mathematics*, pages 256–271. Springer, 2022.

[Hutter *et al.*, 2009] Frank Hutter, Holger H Hoos, Kevin Leyton-Brown, and Thomas Stützle. Paramils: an automatic algorithm configuration framework. *Journal of artificial intelligence research*, 36:267–306, 2009.

[Kerschke *et al.*, 2019] Pascal Kerschke, Holger H Hoos, Frank Neumann, and Heike Trautmann. Automated algorithm selection: Survey and perspectives. *Evolutionary computation*, 27(1):3–45, 2019.

[Lindauer *et al.*, 2022] Marius Lindauer, Katharina Eggensperger, Matthias Feurer, André Biedenkapp, Difan Deng, Carolin Benjamins, Tim Ruhkopf, René Sass, and Frank Hutter. Smac3: A versatile bayesian optimization package for hyperparameter optimization. *Journal of Machine Learning Research*, 23(54):1–9, 2022.

[McCune, 2005] W. McCune. Prover9 and Mace4. http://www.cs.unm.edu/~mccune/prover9/, 2005. Accessed: 2025-05-21.

[Meseguer, 2003] José Meseguer. Software specification and verification in rewriting logic. *Nato Science Series Sub Series III Computer and Systems Sciences*, 191:133–194, 2003.

[Nagele *et al.*, 2015] Julian Nagele, Bertram Felgenhauer, and Aart Middeldorp. Improving automatic confluence analysis of rewrite systems by redundant rules. In *26th International Conference on Rewriting Techniques and Applications (RTA 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[Nagele *et al.*, 2017] Julian Nagele, Bertram Felgenhauer, and Aart Middeldorp. Csi: New evidence–a progress report. In *International Conference on Automated Deduction*, pages 385–397. Springer, 2017.

[Ramírez *et al.*, 2016] Nicolás Gálvez Ramírez, Youssef Hamadi, Eric Monfroy, and Frédéric Saubion. Evolving smt strategies. In *2016 IEEE 28th International Conference on Tools with Artificial Intelligence*, 2016.

[Sternagel and Thiemann, 2014] Christian Sternagel and René Thiemann. The certification problem format. *arXiv preprint arXiv:1410.8220*, 2014.

[Stump *et al.*, 2014] Aaron Stump, Geoff Sutcliffe, and Cesare Tinelli. Starexec: A cross-community infrastructure for logic solving. In *International joint conference on automated reasoning*, pages 367–373. Springer, 2014.

[Thiemann and Sternagel, 2009] René Thiemann and Christian Sternagel. Certification of termination proofs using ceta. In *International Conference on Theorem Proving in Higher Order Logics*, pages 452–468. Springer, 2009.

[Van Oostrom, 1994] Vincent Van Oostrom. Confluence by decreasing diagrams. *Theoretical computer science*, 126(2):259–280, 1994.

[Van Oostrom, 1997] Vincent Van Oostrom. Developing developments. *Theoretical Computer Science*, 175(1):159–181, 1997.

[Willsey *et al.*, 2021] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. Egg: Fast and extensible equality saturation. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–29, 2021.

[Winkler and Moser, 2019] Sarah Winkler and Georg Moser. Smarter features, simpler learning? In *Proceedings of the Second International Workshop on Automated Reasoning: Challenges, Applications, Directions, Exemplary Achievements*, 2019.

[Xu *et al.*, 2010] Lin Xu, Holger Hoos, and Kevin Leyton-Brown. Hydra: Automatically configuring algorithms for portfolio-based selection. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 24, pages 210–216, 2010.

[Zantema, 2004] Hans Zantema. Relative termination in term rewriting. In *WST'04 7th International Workshop on Termination*, page 51, 2004.

Automated Strategy Invention for Confluence of Term Rewrite Systems
Technical Appendix

1 Term Rewriting

We explain the essential concepts of term rewriting in this section.

Various types of rewrite systems exist based on the formalization of objects, with the simplest being the abstract rewrite system (ARS).

Definition 1. An ARS is a pair $\mathcal{A} = (A, \rightarrow)$ of a set A and a binary relation $\rightarrow$ on A.

A (possibly infinite) *rewrite sequence* is a sequence $a_0 \rightarrow a_1 \rightarrow \cdots$ such that $a_i \in A$. We write $a \rightarrow^* b$ if there is a rewrite sequence $a \rightarrow \cdots \rightarrow b$.

Definition 2. An ARS $(A, \rightarrow)$ is *terminating* if $\forall a \in A$, there are no infinite rewrite sequences starting from a.

The notation $a \downarrow b$ denotes that a and b are *joinable*, meaning that there exists an element $c \in A$ such that $a \rightarrow^* c$ and $b \rightarrow^* c$.

Definition 3. An ARS $(A, \rightarrow)$ is *confluent* if $\forall a, b, c \in A$ with $b \leftarrow^* a \rightarrow^* c$, we have $b \downarrow c$.

Consider an abstract reduction system (ARS) $\mathcal{E} = (E, \rightarrow)$, where $E = \{a, b, c, d\}$ and $\rightarrow = \{(a, b), (b, d), (c, b), (d, c)\}$. The ARS $\mathcal{E}$ is non-terminating as it admits an infinite rewrite sequence: $c \rightarrow b \rightarrow d \rightarrow c \rightarrow \cdots$.

Term rewrite systems (TRSs) extend ARS by incorporating first-order variables and employing first-order terms.

We then define the notions of rewriting terms using contexts and holes.

Definition 4. A *hole* is defined as a special symbol $\Box \notin \mathcal{F}$, and a *context* C is a term that contains exactly one hole. The notion $C[t]$ denotes the *application of the term t to the context* C, which is defined as follows:

$$C[t] = \begin{cases} t & \text{if } C = \Box \\ f(t_1, \ldots, C'[t], \ldots, t_n) & \text{if } C = f(t_1, \ldots, C', \ldots, t_n) \end{cases}$$

Definition 5. The *set of variables in a term t* is defined as

$$Var(t) = \begin{cases} \{t\} & \text{if } t \text{ is a variable} \\ \varnothing & \text{if } t \text{ is a constant} \\ \bigcup_{i=1}^{n} Var(t_i) & \text{if } t = f(t_1, \ldots, t_n) \end{cases}$$

Definition 6. A *rewrite rule* for terms l and r is written as $l \rightarrow r$ where $l \notin \mathcal{V}$ and $Var(r) \subseteq Var(l)$. A *term rewrite system* $\mathcal{R}$ consists of a set of rewrite rules. Consider the TRS $\mathcal{R}$, we write the *rewrite relation* $t \rightarrow_{\mathcal{R}} u$ for terms t, u if there exists a rewrite rule $l \rightarrow r \in \mathcal{R}$, a context C, and a substitution σ such that $t = C[l\sigma]$ and $u = C[r\sigma]$.

We write $\rightarrow_{\mathcal{R}}^*$ to denote the transitive-reflexive closure of $\rightarrow_{\mathcal{R}}$. Similar to ARSs, we obtain the definitions of rewrite sequences and $\downarrow_{\mathcal{R}}$ for TRSs. We drop the subscript $\mathcal{R}$ for the relations on terms in the subsequent sections if it is contextually inferrable.

Definition 7. A TRS $\mathcal{R}$ is *terminating* if $\forall t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$, there is not any infinite rewrite sequence $t \rightarrow t_1 \rightarrow \cdots$ starting from t.

The TRS $\mathcal{A} = \{f(x) \rightarrow g(f(x)), g(y) \rightarrow f(g(y))\}$ is not terminating, as it allows the infinite rewrite sequence $f(x) \rightarrow g(f(x)) \rightarrow f(g(f(x))) \rightarrow \cdots$. This sequence is infinite because the term $f(x)$ within $g(f(x))$ can be rewritten back to $g(f(x))$, resulting in an infinite loop.

Definition 8 (confluence)**.** A TRS $\mathcal{R}$ is *confluent* if and only if $\forall s, t, u \in \mathcal{T}(\mathcal{F}, \mathcal{V}), s \rightarrow_{\mathcal{R}}^* t \wedge s \rightarrow_{\mathcal{R}}^* u \Rightarrow t \downarrow_{\mathcal{R}} u$.

The TRS $\mathcal{B} = \{f(x, x) \rightarrow a, f(x, g(x)) \rightarrow b, c \rightarrow g(c)\}$ is not confluent, as it permits the following rewrite sequences: $a \leftarrow f(c, c) \rightarrow f(c, g(c)) \rightarrow b$. Since no rules can be applied to a and b, convergence between them is not achievable.

A term is called *linear* if no variable multiply occurs in it. A rewrite rule $l \rightarrow r$ is called *left-linear* if l is linear. A TRS is called left-linear if all its rules are left-linear. Left-linearity is crucial for confluence analysis since most existing confluence analysis techniques depend on it to determine the confluence of TRSs. In this paper, a term is called *compositional* if it is neither a variable nor a constant.

We will also explain some rewriting concepts that are important for the understanding of the parameter space in Section 4. However, our parameter involves an extensive number of rewriting techniques. Explaining all basic rewriting concepts requires extremely large amount of work, which is beyond the scope of our paper. Moreover, a lot of definitions or theorems rely on previous definitions and theorems. Presenting all the dependent also necessiate too much work for us.

We recommend readers to read some textbooks [Baader and Nipkow, 1998; Bezem *et al.*, 2003] for the concepts that they cannot understand.

You may skip the following definitons and theorems if you can get a feeling of our parameter space.

1.1 Basic Termination Techniques

To prove termination, many techniques try to discover a well-founded monotone algebra that is compatible with the given TRS. The crucial part is the discovery of interpretations. Depending on their formats, there are integer interpretations, polynomial interpretations, matrix interpretations, etc.

Definition 9 (interpretation)**.** Let $\mathcal{F}$ be a signature. An $\mathcal{F}$-*algebra* $\mathcal{A}$ is a set A equipped with operations $f_{\mathcal{A}} : A^n \to A$ for every n-ary function symbol $f \in \mathcal{F}$. The underlying set A is called the *carrier* of $\mathcal{A}$ and $f_{\mathcal{A}}$ is called the interpretation of f.

Definition 10. Let $\mathcal{A}$ be an arbitrary algebra. We inductively define a mapping $[\cdot]_{\mathcal{A}}$ from the set of ground terms to $\mathcal{A}$ as follows: $[f(t_1, ..., t_n)]_{\mathcal{A}} = f_{\mathcal{A}}([t_1]_{\mathcal{A}}, \ldots, [t_n]\mathcal{A})$. In particular, if t is a constant then $[t]_{\mathcal{A}} = t_{\mathcal{A}}$.

Definition 11 (well-founed relation)**.** Let R be a relation on a set A. A relation R is called *well-founded* if there are no infinite descending sequences $a_1 R a_2 R a_3 R \cdots$ of elements of A.

Definition 12 (monotone algebra)**.** A monotone $\mathcal{F}$-algebra $(\mathcal{A}, >)$ consists of a non-empty $\mathcal{F}$-algebra $\mathcal{A}$ and a proper order $>$ on the carrier $\mathcal{A}$ of $\mathcal{A}$ such that every algebra operation is strictly monotone in all its coordinates, i.e., if $f \in F$ has arity $n \geq 1$ then $f_{\mathcal{A}}(a_1, \ldots, a_i, \ldots, an) > f_{\mathcal{A}}(a_1, \ldots, b, \ldots, a_n)$ for all $a_1, \ldots, a_n, b \in A$ and $i \in 1, \ldots, n$ with $a_i > b$. We call a monotone $\mathcal{F}$-algebra $(\mathcal{A}, >)$ well-founded if $>$ is well-founded.

Theorem 1. A TRS is terminating if and only if it is compatible with a well-founded monotone algebra.

Example 1. Consider the TRS $\mathcal{R}_1$ consisting of the single rewrite rule $f(f(x, y), z) \to f(x, f(y, z))$ Let $(\mathcal{A}, >) = (\mathbb{N}, >_{\mathbb{N}})$, the set of natural numbers equipped with the usual order, and define $f_{\mathcal{A}}(x, y) = 2x + y + 1$ for all $x, y \in \mathbb{N}$. The operation $f_{\mathcal{A}}$ is strictly monotone in both coordinates: if $x >_{\mathbb{N}} x'$ and $y >_{\mathbb{N}} y'$ then $2x+y+1 >_{\mathbb{N}} 2x'+y+1$ and $2x+y+1 >_{\mathbb{N}} 2x+y'+1$. We have $f_{\mathcal{A}}(f_{\mathcal{A}}(x, y), z) = 4x+2y+z+3 > \mathbb{N}2x+2y+z+2 = f_{\mathcal{A}}(x, f_{\mathcal{A}}(y, z))$ for all $x, y, z \in \mathbb{N}$. Hence $[\alpha]_{\mathcal{A}}(f(f(x, y), z)) >_{\mathbb{N}} [\alpha]_{\mathcal{A}}(f(x, f(y, z)))$ for every assignment α, yielding the termination of $\mathcal{R}_1$

1.2 Basic Confluence Techniques

One typical way of proving confluence is first proving termination and then proving local confluence.

Definition 13 (local confluence)**.** Let $\mathcal{R}$ be a TRS. An element $a \in \mathcal{T}$ is *locally confluent* if for all elements $b, c \in \mathcal{T}$ with $b \to a \to c$ we have $b \downarrow c$. The TRS $\mathcal{T}$ is locally confluent if all its elements are locally confluent.

Theorem 2 (Newman's Lemma)**.** Every terminating and locally confluent TRS is confluent.

1.3 Basic Non-confluence Techniques

We want to introduce critical pairs since they are crucial for non-confluence analysis.

Definition 14 (substitution)**.** A *substitution* is a mapping σ from $\mathcal{V}$ to $\mathcal{T}(\mathcal{F}, \mathcal{V})$. The *application of the substitution σ to the term t* is defined as:

$$t\sigma = \begin{cases} \sigma(t) & \text{if } t \in \mathcal{V} \\ f(t_1\sigma, \ldots, t_n\sigma) & \text{if } t = f(t_1, \ldots, t_n) \end{cases}$$

Definition 15 (unifiability)**.** Two terms s and t are unifiable if there exists a substitution σ such that $\sigma s = \sigma t$.

Definition 16 (variant)**.** A *variable substitution* is a substitution from $\mathbf{V}$ to $\mathcal{V}$. A *renaming* is a bijective variable substitution. A term s is a variant of a term t if $s = t\sigma$ for some renaming σ.

Definition 17 (Position)**.**

$$\mathcal{P}os(t) = \begin{cases} \{\epsilon\} \text{ if } t \text{ is a variable} \\ \{\epsilon\} \cup \{ip | 1 \leq i \leq n \text{ and } p \in Pos(t_i)\} \\ \quad \text{if } t = f(t_1, \ldots, t_n) \end{cases}$$

Let $p \in \mathcal{P}os(t)$. The subterm of t at position p is denoted by $t|_p$, i.e.,

$$t|_p = \begin{cases} t \text{ if } p = \epsilon \\ t_i|q \text{ if } t = f(t_1, \ldots, t_n) \text{ and } p = iq \end{cases}$$

The symbol $t(p)$ at position p in t is defined as $t(p) = root(t|_p)$. We partition the set $\mathcal{P}os(t)$ into $\mathcal{P}os_{\mathcal{V}}(t) = \{p \in \mathcal{P}os(t) | t|_p \in \mathcal{V}\}$ and $\mathcal{P}os_{\mathcal{F}}(t) = \mathcal{P}os(t) \setminus Pos_{\mathcal{V}}(t)$.

$\mathcal{P}os_{\mathcal{V}}(t)$ denotes the positions of variables in the term t. $\mathcal{P}os_{\mathcal{F}}(t)$ denotes the positions of function symbols in the term t.

Definition 18 (Overlap)**.** An *overlap* of a TRS $(\mathcal{F}, \mathcal{R})$ is a triple $\langle l_1 \to r_1, p, l_2 \to r_2 \rangle$ satisfying the following properties:

1. $l_1 \to r_1$ and $l_2 \to r_2$ are variants of rewrite rules of $\mathcal{R}$ without common variables,
2. $p \in \mathcal{P}os_{\mathcal{F}(l_2)}$
3. l_1 and $l_2|_p$ are unifiable,
4. if $p = \epsilon$ then $l_1 \to r_1$ and $l_2 \to r_2$ are not variants.

Definition 19 (Critical Pair)**.** Suppose $\langle l_1 \to r_1, p, l_2 \to r_2 \rangle$ is an overlap of a TRS $\mathcal{R}$. Let σ be a most general unifier of l_1 and $l_2|_p$. The term $l_2\sigma[l_1\sigma]p = l_2\sigma$ can be rewritten in two different ways: $l_2\sigma[r_1\sigma]p \xleftarrow[p]{l_1 \to r_1} l_2\sigma[l_1\sigma]p = l_2\sigma \xrightarrow[\epsilon]{l_2 \to r_2} r_2\sigma$ We call the quadruple $(l_2\sigma[r_1\sigma]p, p, l_2\sigma, r_2\sigma)$ a *critical peak* and the equation $l_2\sigma[r_1\sigma]p \approx r_2\sigma$ a *critical pair* of $\mathcal{R}$, obtained from the overlap $\langle l_1 \to r_1, p, l_2 \to r_2 \rangle$.

Example 2. Let the TRS $\mathcal{R}$ have two rules $f(a, g(x)) \to f(x, x)$ and $g(b) \to c$. We have an overlap $\langle g(b) \to c, 2, f(a, g(x)) \to f(x, x) \rangle$. It gives rise to the critical peak $f(a, c) \overset{2}{\leftarrow} f(a, g(b)) \overset{\varepsilon}{\to} f(b, b)$ and the critical pair $f(a, c) \approx f(b, b)$.

To disprove confluence of a TRS $\mathcal{R}$, we consider peaks of the form

$$t \leftarrow^{\leq m} t_1 \leftarrow s \rightarrow u_1 \rightarrow^{\leq n} u$$

such that $t_1 = s[r_1\sigma]p \leftarrow s[t_1\sigma]p = s = s[t_2\sigma]q \rightarrow s[r_2\sigma]q = u_1$ with $t_1 \rightarrow r_1$, $t_2 \rightarrow r_2 \in R, q \leq p$, and $p \in \mathcal{P}os(s[t_2]q)$. The basic idea is to show non-joinability of t and u. In order to test non-joinability of t and u we consider ground instances of t and u. Here, ground instances mean substituting all variables with constants. Let c_x be a fresh constant for every variable x and let $\hat{t}$ denote the result of replacing every variable in a term t with the corresponding constant. Since for terms s and w we have $s \rightarrow_{\mathcal{R}} w$ if and only if $\hat{s} \rightarrow_{R} \hat{w}$, it follows that terms t and u are joinable if and only if $\hat{t}$ and $\hat{u}$ are joinable. To test non-joinability of $\hat{t}$ and $\hat{u}$ we overapproximate the sets of reducts for $\hat{t}$ and $\hat{u}$ and check if the intersection of these sets is empty.

2 CSI Strategy Language

Besides the technical appendix, [Nagele *et al.*, 2017] also explains CSI's strategy language.

Overall Grammar A strategy is defined by the grammar

```
s ::= m | (s) | c | i | e
e ::= s% | s! | s[f] | {s}o
i ::= s? | s* | s+ | sn* | s[f]*
c ::= s;s | s|s | s||s
      | if p then s else s
```

where `s` expresses the possible strategies of CSI, `m` denotes the name of any available processor, `p` denotes the name of any available predicate, and `c`, `i`, and `e` define the available combinators, iterators, and specifiers. Here combinators are used to combine two strategies whereas iterators are used to repeat a given strategy a designated number of times. In contrast, specifiers are used to control the behavior of strategies. A strategy works on a confluence problem. Whenever CSI executes a strategy, internally, a so-called proof object is constructed which represents the actual proof. Depending on the shape of the resulting proof object after applying a strategy `s`, we say that `s` succeeded or `s` failed. This should not be confused with the possible answers of the prover: `YES`, `NO`, and `MAYBE`. Here `YES` means that confluence could be proved, `NO` indicates a successful non-confluence proof, and `MAYBE` refers to the case when confluence could neither be proved nor disproved. On the success of a strategy `s`, it depends on the internal proof object whether the final answer is `YES` or `NO`. On failure, the answer is always `MAYBE`. Based on the two possibilities success or failure, the semantics of the strategy operators is as follows.

Combinators

- `s;s'`: First applies `s` to the given problem. If this fails, then `s;s'` fails. Otherwise `s'` is applied to the resulting problems.
- `s|s'`: Applies `s` to the given problem. If this succeeds, its result is returned. Otherwise `s'` is applied to the given problem.
- `s||s'`: Runs `s` and `s'` in parallel on the given problem. As soon as at least one of `s` and `s'` succeeds, the resulting problem is returned.
- `if p then s else s'`: Applies `s` to the given problem if `p` is satisfied by the underlying problem. Otherwise `s'` is applied.

Iterators

- `s?`: Applies `s` to the given problem. On success, its result is returned. Otherwise, the original problem is returned and unmodified.
- `s*`: Applies `s` recursively to the given problem until it cannot be modified anymore. Note that `s*` is always successful.
- `s+`: Applies `s` recursively to the given problem until it cannot be modified anymore. I.e., `s+` is successful if it can prove or disprove termination or confluence of the given problem. Otherwise, it fails. Note that `s+ = s*;s` but `s+` is not equivalent to `s;s*`.
- `sn*`: Applies `s` recursively to the given problem until it cannot be modified anymore or `s` has been applied n times. Note that `sn*` is always successful.
- `s[f]*`: Applies `s` recursively to the given problem until it cannot be modified anymore or `f` seconds are elapsed. Note that `s[f]*` is always successful.

Specifiers

- `s%`: Applies `s` to the given problem. If `s` fails, the computation is aborted and `s%` fails. Otherwise, it succeeds.
- `s!`: Applies `s` to the given problem. If `s` proves or disproves termination or confluence of the given problem, `s!` is successful. Otherwise, it fails.
- `s[f]`: Tries to modify a given problem via `s` for at most `f` seconds. If `s` does not succeed or fail within `f` seconds, `s[f]` fails. Otherwise `s[f]` returns the resulting problem. Hence it succeeds (fails) if `s` succeeds (fails).
- `{s}o`: Applies `s` to the given problem. If `s` fails, `{s}o` fails. Otherwise, the modifier `o` is applied to the resulting problems.

Configuration File Since strategies can get quite complex, CSI provides the possibility to specify a config file. A config file consists of a sequence of abbreviations of the form `N = s` where `N` defines its name and `s` the strategy (in principle arbitrary text) it should expand to. The convention is to use all capital names for abbreviations. Each abbreviation has to be put on a separate line. You can spread a strategy over several lines by terminating each line with a \. Last but not least, you can add comments to config files by putting a # in front of each line.

3 Default Strategy

This section explains the workflows of CSI's default competition strategy. In the CoCo competition, CSI's execution command is roughly `csi -C CR -s AUTO -c x.trs`. Here, `-C CR` denotes to prove (non-)confluence for a TRS. CSI is based on the termination tool `TTT2` [Korp *et al.*, 2009] since some confluence techniques need to first prove termination. There are also other flags for `-C` to prove different properties of TRSs. The flag `-s AUTO` means that CSI executes the `AUTO` strategy below in the default configuration document. Finally, `x.trs` means to prove (non-)confluence for the TRS x. In ARI-COPS, the TRSs are represented by the `.ari` format. However, CSI cannot receive `.ari` documents currently. We need to first execute a tool [James and Fabian, 2023] to convert `.ari` documents to the corresponding `.trs` documents.

```
AUTO = (if trs then (\
  sorted -order*;(AUTO_INNER \
    || (NOTCR | REDUNDANT_FC)3*!) \
 ) else fail)
```

`AUTO` works as below:

1. Check whether the given problem is a TRS problem. There are many different rewrite systems that can be input to CSI like higher-order rewrite systems, etc. CSI fails if it is not a TRS problem.
2. If it is, the default strategy uses ordered sorted decomposition [Felgenhauer *et al.*, 2015] to try to decompose the given TRS problem to a set of sub-problems. The original TRS is confluent if and only if all decomposed problems are confluent [Felgenhauer *et al.*, 2015].
3. `AUTO_INNER` and `(NOTCR | REDUNDANT_FC)3*!` are parallelly executed. `AUTO_INNER` mainly focuses on proving confluence, while `(NOTCR | REDUNDANT_FC)3*!` focuses on disproving confluence. `(NOTCR | REDUNDANT_FC)3*!` first uses `NOTCR` to determine whether the given problem is nonconfluence, if it can be determined, the answer will be returned. Otherwise, it applies `REDUNDANT_FC` which executes the redundant rule technique [Nagele *et al.*, 2015] to transform the given TRS. `(NOTCR | REDUNDANT_FC)` is repeatedly applied for at most three times to discover an answer. The specifier `!` is crucial since the transformation of `REDUNDANT_FC` may be successful, but it does not discover a nonconfluence proof. According to the explanation in Section 2, CSI will return `YES` in this case if we do not use `!`. The details of `NOTCR` and `REDUNDANT_FC` are explained in Section 4.

```
AUTO_INNER = \
  (AUTO_INNER0[30] | CPCS[5]2*)! | \
  ({AUTO_INNER0[30]}nono \
   | CPCS[5]2*)2*!
```

The definition of `AUTO_INNER` is shown above. It first tries to discover a proof using `AUTO_INNER0`. If it cannot discover a proof, the critical pair closing system (CPCS) [Oyamaguchi and Hirokawa, 2014] transformation is applied. Then `AUTO_INNER0` is executed again. Notice that the second `AUTO_INNER0[30]nono` uses `nono` to ignore the proof for nonconfluence produced by `AUTO_INNER0` because after the CPCS transformation, we cannot produce sound proofs for nonconfluence [Oyamaguchi and Hirokawa, 2014].

```
AUTO_INNER0 = (GROUND || KB || AC ||\
  (REDUNDANT_DEL?;(CLOSED || DD \
     || SIMPLE || KB || AC \
     || {GROUND}nono))3*! \
  || (((CLOSED || DD) \
    | REDUNDANT_RHS)3*! \
  || (CLOSED || DD) \
    | REDUNDANT_JS)3*! || KH \
  || AT || SIMPLE || CPCS2 || \
 fail)
```

The above code explains the details of `AUTO_INNER0`. It parallelly executes various techniques. The details of all techniques are explained in Section 4.

4 Parameter Space

4.1 Overall Patterns

The name of a parameter often follows one of the patterns:

- *abbreviation* such as `PRETRS`. It is a *boolean-execution controlling parameter*. When it is set to `no`, we set `PRETRS = fail` in the generated configuration document. The `fail` processor fails immediately. CSI will not do any modification and discover any proof.
- *abbreviation_processor* such as `PRETRS_matrix`. It is a *boolean-execution controlling parameter*. We use such names when a strategy contains several parallelly or sequentially executed processors. When it is set to `no`, we ignore the corresponding processor in the strategy.
- *abbreviation_processor_flag*. It belongs to *processor flags*, e.g., `PRETRS_matrix_ib`. It chooses the flags for the processor. For instance, `PRETRS_matrix_ib` chooses the value for the `ib` flag. If `PRETRS_matrix_ib` chooses the value 6, we may obtain `matrix -ib 6`.
- *name_time* or *name_loop*. It belongs to *iteration parameters*. They are used to control the execution time or the number of repeated application times of a strategy. They control the values of the specifiers. For instance, if `PRETRS_time` is set to 2, we generate the strategy `PRETRS = content of PRETRS[2]`.

4.2 Details of Patameters

In this Section, we explain the strategies in CSI's competition strategy document, following its structure sequentially from the beginning to the end. The explanations of a flag and its soundness will be ignored if we have explained them in the previous strategy definitions. After searching for the parameters, we convert the parameters into a configuration document which is used by CSI. The structures of strategies generally remain the same to confirm soundness, meaning that besides the newly discovered parameters, we do not change the processors and the order of their applications. We give examples to explain how to keep the structures in this section later via examples.

We ignore the processor flags not searched by us since the number of them is indeed intensive, and presenting all explanations is too verbose. We recommend executing CSI's help function to understand the details.

```
PRETRS = ((\
 matrix -dim 1 -ib 3 -ob 5 | \
 matrix -dim 2 -ib 2 -ob 3 | \
 matrix -dim 3 -ib 1 -ob 1 | \
 matrix -dim 3 -ib 1 -ob 3)[2]*)
```

It preprocesses the TRS before trying to discover a termination proof. The preprocess tries to reduce the size of the original TRS by removing some rules from it. The parameter space for `PRETRS` is presented below.

- `PRETRS` {`yes, no`}. `PRETRS` can be chosen as `yes` or `no`. When it is set as `no`, we replace its definition with `fail`, i.e. `PRETRS = fail`. The processor `fail` means the processor simply fails. Replacing it with `fail` is sound since it is only used in `SN`. It is `(PRETRS;(...)`. According to the definition of `;`, replacing it with `fail` only makes the strategy for termination fail immediately.
- `PRETRS_time` {1, 2, 3, 4, 5}. It is used to control the execution time of `PRETRS`. The default execution time is 2 seconds, controlled by `[2]` as in the definition. We set the execution time to be chosen from {`1, 2, 3, 4, 5`} seconds.
- `PRETRS_matrix_dim` {`1, 2, 3, 4, 5, 6, 7, 8, 10, 12, 16`}. The `matrix` processor means applying matrix interpretations [Endrullis *et al.*, 2008]. The parameter `PRETRS_matrix_dim` specifies the dimension of the matrices. Each matrix processor has several processor flags and associative values to choose from as below. Since there are several matrix processors in `PRETRS`, we actually associate each matrix processor with a set of parameters to search from. For simplicity, we only show the parameter space of a single matrix processor. The values {yes, no} indicate whether to use such flags. For instance, when `matrix_real` is set to `yes`, we add the `real` flag to `matrix` and obtain `matrix -real`. In comparison, when it is set to `no`, we do not append the `real` flag and simply use `matrix`. The other flag values indicate the values of the flags. For instance, when `matrix_ib` is set to `3`, we obtain a processor of `matrix -ib 3`.
- `PRETRS_matrix_ib` {`1, 2, 3, 4, 5, 6`}. Defines the number of bits that can be used to represent the smallest number that appears in the intermediate results
- `PRETRS_matrix_ob` {`1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, max`}. Defines the number of bits that can be used to represent the largest number that appears in the intermediate results. Actually `max` is not a valid value to the `ob` flag. However, Grackle has the mechanism to ignore a parameter if it equals the default value. We set `max` as `PRETRS_matrix_ob`'s default value. Moreover, by default, CSI uses the largest 64-bit integers for `ob`. Hence, `max` reasonably represents the maximal value of `ob`.
- `PRETRS_matrix_rat` {`1, 2, 3, 4`}. Sets the denominator of integers. It makes integers become rational numbers. The soundness of non-negative rational numbers for matrix interpretation is explained in in [Gebhardt *et al.*, 2007; Zankl and Middeldorp, 2010]. The `matrix` processor also has a flag `neg`, which is set to `false` by default. Since we do not use `neg` in our parameter space, the interpretations are always non-negative and therefore sound.
- `PRETRS_matrix_db` {`1, 2, 3, 4, max`}. Specifies the bits after the decimal point. The reason for soundness is the same as `PRETRS_matrix_rat`
- `PRETRS_matrix_real` {`yes, no`}. The soundness of non-negative real numbers for matrix interpretation is explained in Lemma 23, Definition 24, and Lemma 25 in [Zankl and Middeldorp, 2010]. Matrix interpretation over real numbers is unsound because negative numbers break the monotone constraints [Endrullis *et al.*, 2008]. However, `matrix` also has a flag `neg`, which is set to `false` by default. Since we do not use `neg` in our parameter space, the real number interpretations are always non-negative and therefore sound.
- `PRETRS_matrix_triangle` {`yes, no`}. Use triangular matrices. It is sound since it only constrains the shape of the matrix. It is less expressive than the traditional matrix interpretation. Originally, it is invented for complexity analysis [Moser *et al.*, 2008].

The structure of `PRETRS` is kept although we search for other parameters. This means the `matrix` processors are still connected by `|` instead of `||`. We may generate something like the strategy below.

```
PRETRS = ((\
 matrix -dim 1 -ib 4 -ob 5 | \
 matrix -dim 3 -ib 3 -ob 3 | \
 matrix -dim 3 -ib 3 -ob 7 | \
 matrix -dim 3 -ib 1 -ob 3)[4]*)
```

Next, we explain the parameter space for `DIRECTTRS`.

```
DIRECTTRS = ((\
 kbo || (lpo | (ref;lpo) \
 || (bounds -rfc -qc))*[7])!
```

- `DIRECTTRS` {`yes, no`}. `DIRECTTRS` and `DIRECTTRS_time` works similar as `PRETRS` and `PRETRS_time`. Hence, we ignore such explanations here and for the following sub-strategy definitions. It is sound as `DIRECTTRS` is only parallelly with other strategies in `SN`. It makes one technique fail and does not affect other parallel executions.

- `DIRECTTRS_time {1, 3, 5, 7, 9, 11}`
- `DIRECTTRS_kbo {yes, no}`. The `kbo` processor means the application of the Knuth-Bendix order (KBO) [Baader and Nipkow, 1998].
- `DIRECTTRS_kbo_ep {yes, no}`. Demands an empty precedence (only for '-pbc'). According to the function `pbc_aux` in csi/src/processors/src/termination/orderings/kbo.ml, it is only used when the PBC prover is invoked. As explained in Definition 1 in [Zankl *et al.*, 2009], KBO has a set of precedences and a set of weight functions. An empty set of precedences makes KBO weaker in discovering termination proofs. Empty precedences have also been used in Example 3 in the paper. Section 9 in [Zankl *et al.*, 2009] shows empty precedences are sometimes useful for the PBC backend.
- `DIRECTTRS_kbo_ib {1, 2, 3, 4, 5, 6}`. Defines the number of bits that can be used to represent the smallest number that appears in the intermediate results
- `DIRECTTRS_kbo_minp {yes, no}`. Minimizes the precedence comparisons (only for '-pbc'). According to the function `solve2` in csi/src/logic/src/solver.ml and the function `context` in csi/src/processors/src/termination/orderings/kbo.ml, it is only used when the PBC prover is invoked. According to the function `solve` in csi/src/logic/src/miniSatP.ml, the usage of `-pbc` is sound without `-minp` or `-minw`. According to Section 9 in [Zankl *et al.*, 2009], it is sound and is helpful for generating human-readable proofs. KBO has a set of precedence and a set of weight functions. This flag tries to discover a termination proof with a small set of precedence.
- `DIRECTTRS_kbo_minw {yes, no}`. Minimize the sum of weights (only for '-pbc'). According to the function `solve2` in csi/src/logic/src/solver.ml and the function `context` in csi/src/processors/src/termination/orderings/kbo.ml, it is only used when the PBC prover is invoked. According to the function `solve` in csi/src/logic/src/miniSatP.ml, the usage of `-pbc` is sound without `-minp` or `-minw`. According to Section 9 in [Zankl *et al.*, 2009], it is sound and is helpful for generating human-readable proofs. Small weights are more readable. KBO has a set of precedence and a set of weight functions. This flag tries to discover a termination proof with weight functions that map symbols to small weights.
- `DIRECTTRS_kbo_ob {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, max}`. Defines the number of bits that can be used to represent the largest number that appears in the intermediate results.
- `DIRECTTRS_kbo_pbc {yes, no}`. Uses PBC backend. PBC, SAT, or SMT are just solvers to search for interpretations. The choice does not affect the soundness.
- `DIRECTTRS_kbo_quasi {yes, no}`. Allows quasi-precedences. It is sound according to Definition 3.1 in [Sternagel and Thiemann, 2013].
- `DIRECTTRS_kbo_rat {1, 2, 3, 4}`. Sets the denumerator (only in combination with '-sat' or '-smt').
- `DIRECTTRS_kbo_sat {yes, no}`. Uses SAT backend (default).
- `DIRECTTRS_kbo_smt {yes, no}`. Uses SMT backend.
- `DIRECTTRS_lpo {yes, no}`. Applies Lexicographic Path Order [Baader and Nipkow, 1998]. There are two `lpo` processors in `DIRECTTRS`. We employ the same parameters for them to reduce the size of the parameter space. Since `lpo` are weak techniques for proving termination, assigning different parameters to two `lpo` processors should only have little influence on the performance of proving termination.
- `DIRECTTRS_lpo_direct {yes, no}`. Try to finish the termination proof. The `lpo` processor can each time prove the termination of a single rule of a TRS and remove the rule from the TRS. Afterwards, it can be repeatedly applied to prove the termination of another rule in the smaller TRS. It shows the termination of the entire TRS by proving the termination of all rules. Notice that `*[7]` in the definition means repeatedly applying `lpo` and `lpo` as much as possible in seven seconds. Using `direct`, it tries to show termination for all rules; thus, it only makes the searching difficult.
- `DIRECTTRS_lpo_quasi {yes, no}`. Allows quasi-precedences (currently not supported together with -dp flag). The proofs of the soundness are presented in Theorem 2.37 and Theorem 2.26 in Chapter 2 Preliminaries in [Hirokawa, 2006].
- `DIRECTTRS_lpo_sat {yes, no}`.
- `DIRECTTRS_lpo_smt {yes, no}`
- `DIRECTTRS_bounds {yes, no}`. This processor proves termination of a given problem by using the match-bound technique [Korp and Middeldorp, 2009].
- `DIRECTTRS_bounds_qc {yes, no}`. Computes quasi-compatible tree automata instead of compatible tree automata. Different values of the parameter are sound according to [Korp and Middeldorp, 2009]. Moreover, it is used in `DIRECTTRS`.
- `DIRECTTRS_bounds_rc {explicit, implicit}`. Defines the algorithm that is used to construct raise-consistent tree automata. Possible values are `explicit` and `implicit` where. Per default `implicit` is used. Different values of the parameter are sound according to [Korp and Middeldorp, 2009].
- `DIRECTTRS_bounds_rfc {yes, no}`. Uses right-hand sides of forward closures. Different values of the parameter are sound according to [Korp and Middeldorp, 2009]. Moreover, it is used in `DIRECTTRS`.

- DIRECTTRS_bounds_steps {-1, 1, 2, 3, 4, 5, 6, 7, 8, 10, 12, 16, 32}. Specifies the maximum number of compatibility violations that should be solved. This guarantees that the procedure always terminates. Otherwise it might happen that the graph approximation does not terminate. The match-bound technique tries to build a tree automata using the tree automata completion technique. The tree automata completion tries to solve all compatibility violations. However, sometimes there are infinite violations. Meanwhile solving all violations may take too much time. According to csi/src/processors/src/termination/bounds/bounds.ml, CSI uses -1 by default, meaning to solve all violations. Other values make the match-bound technique fail earlier. This flag only controls the size of the search space and therefore sound.

```
ARCTICTRS = arctic -dp -ur \
  -dim 2 -ib 2 -ob 2[2]

ARCTICBZTRS = arctic -bz -dp -ur \
  -dim 2 -ib 2 -ob 2[2]
```

Use arctic interpretation [Koprowski and Waldmann, 2008]. The parameter space for ARCTICTRS and ARCTICBZTRS is presented below.

- ARCTICTRS {yes, no}. Notice that ARCTICTRS and ARCTICBZTRS use the flag ur because they are employed in MAINTRS, which first applies the transformation using the ur processor. The ur processor removes all rules of the given dependency pair (DP) problem which are not usable [Suzuki *et al.*, 2011]. The -ur flag for arctic uses usable rules with respect to interpretation. If we remove -ur here, we will produce unsoundness. Therefore, we always choose to use the -ur flag for arcitc and only search for other parameters. It is sound as ARCTICTRS is only parallelly executed with other strategies in MAINTRS. It makes one technique fail and does not affect other parallel executions.
- ARCTICTRS_time {1, 2, 3, 4}
- ARCTICTRS_arctic_dim {1, 2, 3, 4, 5, 6, 7, 8, 10, 12, 16}. The reasons for the soundness of dim, ib, ob, direct, rat, real have been explained for the previous strategy definitions.
- ARCTICTRS_arctic_ib {1, 2, 4, 8, 16}.
- ARCTICTRS_arctic_ob {1, 2, 4, 8, max}. Defines the number of bits that can be used to represent the largest number that appears in the intermediate results.
- ARCTICTRS_arctic_bz {yes, no}. Since both yes and no are empolyed in ARCTICTRS and ARCTICBZTRS, and they are used in the same strategy definition MAINTRS, we conclude either the choice of yes or no is sound. Moreover, its soundness is explained in Section 7 [Koprowski and Waldmann, 2008].
- ARCTICTRS_arctic_direct {yes, no}
- ARCTICTRS_arctic_dp {yes, no}. Allows non-monotone interpretations, i.e., '0' as a coefficient. In the original definition of ARCTICTRS, -dp is used; hence, yes is sound. Choosing no merely makes the interpretation monotone. Basically, termination techniques require monotone interpretations. The -dp flag is an exception since it works in the dependency pair (DP) frameworks [Giesl *et al.*, 2005a]. The soundness of non-monotone interpretations is explained in Section 6 [Koprowski and Waldmann, 2008].
- ARCTICTRS_arctic_rat {1, 2, 3, 4}. Use rational numbers for arctic interpretations. The soundness is explained in Section 5 [Sternagel and Thiemann, 2014].
- ARCTICTRS_arctic_real {yes, no}. Use real numbers for arctic interpretations. The soundness is explained in Section 6 [Sternagel and Thiemann, 2014].
- ARCTICBZTRS {yes, no}. The parameter space for ARCTICBZTRS is the same as that for ARCTICTRS; hence, we ignore it here. It is sound as ARCTICBZTRS is only parallelly executed with other strategies in MAINTRS. It makes one technique fail and does not affect other parallel executions.

```
BOUNDS = (bounds -dp -rfc -qc \
  || bounds -dp -all -rfc -qc \
  || bounds -rfc -qc)
```

- BOUNDS {yes, no}. Prove termination of a given problem by using the match-bound technique [Korp and Middeldorp, 2009]. It is sound as ARCTICBZTRS is only parallelly executed with other strategies in MAINTRS. It makes one technique fail and does not affect other parallel executions.
- BOUNDS_bounds {yes, no}. There are three bounds, we only explain the parameter space for one for simplicity.
- BOUNDS_bounds_all {yes, no}. This flag is only effective if a DP, critical pair, or relative termination problem is given. In that case, all rewrite rules are proved to be finite (relative terminating) instead of a single rewrite rule. It is sound since it works like the -direct flag that proves a certain property for all rewrite rules at the same time. Moreover, both yes and no are used for bounds in BOUNDS.
- BOUNDS_bounds_dp {yes, no}. Uses the enrichments match-DP and top-DP instead of match and top if a DP problem is given. Make sure that as enrichment either top or match has been chosen because the soundness of roof-DP is unknown. The enrichments are determined by the flag -e. Since we do not search for -e here, it uses the default value match and hence is sound.
- BOUNDS_bounds_qc {yes, no}, BOUNDS_bounds_rc {explicit,

implicit}, BOUNDS_bounds_rfc {yes, no}, BOUNDS_bounds_steps {-1, 1, 2, 4, 8, 16, 32}. Their soundness has been explained for DIRECTTRS

```
MAINTRS = (dp;edg[0.5]?;(sccs | \
 (sc || sct || \
 {ur?;( \
  (matrix -dp -ur -dim 1 \
    -ib 3 -ob 5 | \
   matrix -dp -ur -dim 2 \
    -ib 2 -ob 3 | \
   matrix -dp -ur -dim 3 \
    -ib 1 -ob 1 | \
   matrix -dp -ur -dim 3 \
    -ib 1 -ob 3) || \
  (kbo -ur -af | lpo -ur -af) || \
  ARCTICTRS || \
  ARCTICBZTRS ) \
 }restore || \
 BOUNDS[1]
 ))*[6])!
```

The MAINTRS is the main strategy for proving termination. It first transforms TRS problems into dependency pair (DP) problems using the dp processor. Next, the edg processor reduces the sizes of the DP problems. Afterwards, the sccs processors tries to decompose each DP problems into several smaller DP problems. Finally, many processors are executed to solve the DP problems. In particular, the ur processor removes all rules of the given DP Problem which are not usable. Note that this processor is not sound if the given DP problem is duplicating. To confirm the soundness, the termination processors after the application of the transformation processor ur must use the flag -ur. The processor restore remains unchanged to confirm soundness. It restores the original TRS within the given DP problem. In CSI, TRS rules and DP rules are stored in different data structures. Each execution of processors like matrix in MAINTRS may remove TRS rules or DP rules, but the TRS rule will be restored afterward. The details of the parameter space are presented below.

- MAINTRS {yes, no}. The main technique to prove termination. It is sound as MAINTRS is only parallelly executed with other strategies in SN. It makes one technique fail and does not affect other parallel executions.
- MAINTRS_time {2, 4, 6, 8}
- MAINTRS_edg_time {0.2, 0.5, 1}. The edg processor [Giesl *et al.*, 2005b] removes all edges from the current dependency graph (DG) that are not contained in the EDG (approximation of DG based on recursive unification and symmetry). Here, the parameter controls its execution time.
- MAINTRS_edg_gtcap {yes, no}. Use a general tcap-like non-reachability analysis. Sound as explained in Theorem 13 of [Giesl *et al.*, 2005b].
- MAINTRS_edg_nl {yes, no}. Try to exploit non-linearity for -gtcap. Non-linear order is more expressive than linear order. The soundness is explained in Section 3 in [Giesl *et al.*, 2005b].
- MAINTRS_BOUNDS_time {0.5, 1, 2, 3}
- MAINTRS_sc {yes, no}. Applies the subterm criterion processor [Sternagel, 2016].
- MAINTRS_sc_sat {yes, no}. Uses SAT backend (default).
- MAINTRS_sc_smt {yes, no}. Uses Yices backend (default).
- MAINTRS_sc_rec {yes, no}. Allow recursive simple projections. It is sound because in [Sternagel, 2016], it is explained below Definition 5 and is used in the proving termination as shown in Table 1.
- MAINTRS_sc_mulex {yes, no}. Allow projections to multisets of terms. It is sound because in [Sternagel, 2016], it is explained in Definition 5 and is used in the proving termination as shown in Table 1.
- MAINTRS_sc_defs {yes, no}. Allow projection of defined symbols (only relevant for -rec and -mulex; default false). It is sound according to the explanation below Definition 5 in [Sternagel, 2016].
- MAINTRS_sc_mbits {1, 2, 3, 4, 5}. Bits used for multiplicity of terms in multisets corresponding to left- and right-hand sides (default 2). The soundness is explained in Section 3 [Sternagel, 2016].
- MAINTRS_sc_wbits {1, 2, 3, 4, 5}. Bits used for multiplicity (weight) of arguments in projections (default 2). The soundness is explained in Section 3 [Sternagel, 2016].
- MAINTRS_sc_nsteps {0, 1, 2, 3, 4}. Number of rewrite steps before checking for subterms (default 0). It only uses the rules to rewrite the TRS. The properties of the original TRS remain the same.
- MAINTRS_sct {yes, no}. Applies the size-change termination processor to a DP problem.
- MAINTRS_matrix. The parameter space for the four matrix processors is similar to that for PRETRS. Hence, we ignore its details here. The only difference is that we always use flags -ur and -dp for every matrix processor here. The usage -ur is for soundness, while the usage of -dp aims at making the matrix interpretation stronger in discovering termination.
- MAINTRS_kbo {yes, no}. Applies Knuth-Bendix order [Baader and Nipkow, 1998]. Always use the -ur -af flags.
- MAINTRS_kbo_ib {1, 2, 3, 4, 5, 6}. Defines the number of bits that can be used to represent the smallest number that appears in the intermediate results
- MAINTRS_kbo_ob {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, max}. Defines the number of bits that can be used to represent the largest number that appears in the intermediate results. Actually max is not

a valid value to the `ob` flag. However, Grackle has the mechanism to ignore a parameter if it equals the default value. We set `max` as `MAINTRS_kbo_ob`'s default value. Moreover, by default, CSI uses the largest 64-bit integers for `ob`. Hence, `max` reasonably represents the maximal value of `ob`.

- `MAINTRS_kbo_quasi {yes, no}`. Allow quasi-precedences. It is sound as explained in [Sternagel and Thiemann, 2013]
- `MAINTRS_lpo {yes, no}`. The soundness has been explained previously. Always use the `-ur -af` flags.
- `MAINTRS_lpo_sat {yes, no}`.
- `MAINTRS_lpo_smt {yes, no}`.
- `MAINTRS_lpo_direct {yes, no}`.

The structures are also kept unchanged, meaning that we first execute `dp` and then `edg`, etc. We may probably generate a strategy like below. The flags to `matrix` processors and the time limit of `edg` are changed. However, the overall structure remains the same, and all termination processors always employ the `-ur` and `-dp` flags.

```
MAINTRS = (dp;edg[1]?;(sccs | \
 (sct || \
 {ur?;( \
  (matrix -dp -ur -dim 2 \
    -ib 4 -ob 5 | \
   matrix -dp -ur -dim 2 \
     -ib 2 -ob 3 | \
   matrix -dp -ur -dim 4 \
     -ib 2 -ob 2 | \
   matrix -dp -ur -dim 3 \
     -ib 1 -ob 3) || \
  (kbo -ur -af | lpo -ur -af) || \
  ARCTICTRS || \
  ARCTICBZTRS ) \
 }restore || \
 BOUNDS[2]
 ))*[8])!
```

We will subsequently explain the parameter space for `SN`. It is a sub-strategy for proving termination, which is also called *strong normalization*.

```
SN = (PRETRS;(MAINTRS \
  || DIRECTTRS || \
  (rev;(MAINTRS || DIRECTTRS))))
```

The only parameter is `SN_rev {yes, no}`. If it is `no`, `(rev;(MAINTRS || DIRECTTRS))` is not used in `SN`. Otherwise, it remains in `SN`. It is sound as `SN_rev` is only parallelly executed with other strategies in `SN`. It makes one technique fail and does not affect other parallel executions.

```
SNRELATIVE_STEP = ( \
 lpo -quasi || \
 (matrix -dim 1 -ib 3 -ob 4 | \
  matrix -dim 2 -ib 2 -ob 2 | \
  matrix -dim 3 -ib 1 -ob 2 | \
  arctic -dim 2 -ib 2 -ob 2) || \
 (if duplicating then fail else
   (bounds -rt || \
     bounds -rt -qc))[1] || \
 poly -ib 2 -ob 4 -nl2 -heuristic 1)

SNRELATIVE = (SNRELATIVE_STEP[5]*)
```

`SNRELATIVE_STEP` tries to prove the relative termination of a rule. `SNRELATIVE` repeatedly apply `SNRELATIVE_STEP` in five seconds until all rules are proven to be relatively terminated.

- `SNRELATIVE_STEP_time {1, 2, 3, 4, 5, 6, 7, 8}`. Assign an execution time for `SNRELATIVE_STEP`. Notice that CSI's default strategy assigns five seconds.
- `SNRELATIVE_lpo {yes, no}`. The parameters of `lpo` are the same as those of `lpo` in `DIRECTTRS`.
- `SNRELATIVE_matrix {yes, no}`. The parameters of `matrix` are the same as those of `matrix` in `PRETRS`.
- `SNRELATIVE_arctic {yes, no}`. The parameters of `arctic` are `ib`, `ob`, `direct`, `rat`, `real`. Their soundness have been explained for `ARCTICTRS`.
- `SNRELATIVE_poly {yes, no}`. Applies polynomial interpretations.
- `SNRELATIVE_poly_dim {1, 2, 3, 4, 5, 6, 7, 8, 10, 12, 16}`. Specifies the dimension of the matrices.
- `SNRELATIVE_poly_direct {yes, no}`. Try to finish the termination proof.
- `SNRELATIVE_poly_ib {1, 2, 3, 4, 5, 6}`. Defines the number of bits that can be used to represent the minimal weight that appears in the intermediate results.
- `SNRELATIVE_poly_neg {yes, no}`. Allow negative numbers (only for non-linear interpretations) for some coefficients. It is sound for non-linear interpretation according to Corollary 3.9 in [Neurauter, 2012]. The combination of `neg` flag and the default linear interpretation may cause unsoundenss. We avoid it by using Grackle's forbidden mechanism that disallows the combination of the `neg` flag and the default linear interpretation as shown in Section 4.3. The combination of `neg`, the nonlinear interpretation (`nl` or `nl2`), and the real number interpretation (`real` or `rat > 1`) is also unsound. But CSI can detect it and forbade such combinations according to the function `context` in csi/src/processors/src/termination/interpretations/polynomialInterpretation.ml.

- SNRELATIVE_poly_ob {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, max} Defines the number of bits that can be used to represent the largest number that appears in the intermediate results.
- SNRELATIVE_poly_rat {1, 2, 3, 4}. Sets the denumerators for rational numbers. The soundness is explained in Section 2.1.2 [Neurauter, 2012].
- SNRELATIVE_poly_real {yes, no}. Uses reals. The soundness is explained in Section 2.1.2 [Neurauter, 2012].
- SNRELATIVE_poly_nl {yes, no}. Allow $x^2 + y^2$. By default, linear interpretation is used, which denotes the format of $f(x_1, ..., x_n) = a_0 + a_1x_1 + \cdots + a_nx_n$ where $x_i \geq 0$. The -nl flag enables the interpretation in the format of $f(x_1, ..., x_n) = a_0 + a_1x_1 + \cdots + a_nx_n + a_1'x_1^2 + \cdots + a_n'x_n^2$ where $x_i \geq 0$ according to Section 3.2.2 in [Neurauter, 2012] and the function quadratic in csi/src/processors/src/termination/interpretations/polynomialInterpretation.ml. It is sound according to [Neurauter, 2012].
- SNRELATIVE_poly_nl2 {yes, no}. Allow $x^2 + x * y + y^2$. The -nl2 flag enables the interpretation in the format of

  $$f(x_1, ..., x_n) = a_0 + \sum_{j=1}^{n} a_jx_j + \sum_{1 \leq j \leq k \leq n}^{n} a_{jk}x_jx_k$$

  where $x_i \geq 0$ according to Section 3.2.2 in [Neurauter, 2012] and the function quadratic in csi/src/processors/src/termination/interpretations/polynomialInterpretation.ml. It is sound according to [Neurauter, 2012].
- SNRELATIVE_poly_heuristic {-1, 0, 1, 2, 3, 4}. -1 → all symbols (default); 0 → no symbols ; 1 → symbols appearing at most once in each left-hand side (lhs)/ right-hand side (rhs); 2 → symbols appearing at most twice in each lhs/rhs; 3 → symbols appearing at parallel positions in each lhs/rhs; 4 → defined symbols. It decides which symbols should be interpreted by non-linear polynomials and is sound according to Section 5 in [Neurauter *et al.*, 2010].

```
MATRIXSTAR=(( \
 matrix -dim 1 -ib 2 \
   -ob 2 -strict_empty -lstar | \
 matrix -dim 2 -ib 2 -ob 2 \
   -strict_empty -lstar)[2])
```

- MATRIXSTAR_time {1, 2, 3, 4}
- The parameter space for each matrix processor is the same as that for a matrix processor in PRETRS. Notice that both processors matrix here employ the flags -strict_empty -lstar. We fix the usage of matrix -strict_empty -lstar and search for the other parameters.

```
MATRIXREDEX=(( \
 matrix -dim 1 -ib 2 -ob 2 \
 -strict_empty -lredex)[2])
```

- MATRIXREDEX_time {1, 2, 3, 4, 5}
- The parameter space for each matrix processor is the same as that for a matrix processor in PRETRS. Notice that both processors matrix here employ the flags -strict_empty -lredex. We fix the usage of -strict_empty -lredex and search for the other parameters.

```
LDH = (shift -dd;SNRELATIVE; \
  shift -ldh -force)
LDHF = (shift -dd -force; \
  SNRELATIVE;shift -ldh -force)
SSTAR = (cr M -star;MATRIXSTAR*; \
  shift -sstar)
DUP = (cr M -dup;SNRELATIVE; \
  shift -lstar)
REDEX = (cr M -redex;MATRIXREDEX*; \
  shift -lstar)
```

The strategies in the above code block are very complicated, and we cannot understand them. Therefore, we keep them unchanged and do not search for the parameters. We only have a parameter REDEX {yes, no}. It is sound as it is only used in COR3 = (REDEX; ...)!, and COR3 is parallelly executed with other techniques in DD. It only makes COR3 immediately fail.

```
GROUND = (if ground \
  then uncurry -curry?; \
  groundcr else fail)
```

The decision procedure for ground systems [Felgenhauer, 2012]. We only have GROUND {yes, no}. It is sound as it is only parallel executed with other confluence techniques.

```
NOTCR = ( \
  (nonconfluence -steps 0 \
    -tcap -fun | \
   nonconfluence -steps 2 \
    -tcap -fun | \
   nonconfluence -steps 25 \
    -width 1 -tcap -fun | \
   nonconfluence -steps 2 \
    -idem -fun) || \
  (nonconfluence -steps 2 \
    -tcap -var | \
   nonconfluence -steps 25 \
     -width 1 -tcap -var) || \
  (nonconfluence -steps 0 \
    -tree -fun | \
   nonconfluence -steps 0 \
     -tree -var | \
   nonconfluence -steps 1 \
     -tree -fun | \
   nonconfluence -steps 1 \
     -tree -var | \
   nonconfluence -steps 2 \
     -tree -fun | \
   nonconfluence -steps 2 \
     -tree -var | \
   nonconfluence -steps 25 \
     -width 1 -tree -fun | \
   nonconfluence -steps 25 \
     -width 1 -tree -var) \
)[10]
```

The `NOTCR` strategy is used to disprove confluence. It uses the `||` combinator to parallelly execute three groups of nonconfluence techinuqes. Each group contains several `nonconfluence` processors employed with different parameters. These parameters determine the search space of the `nonconfluence` processors. To improve the execution speed, the processors using smaller search spaces are invoked before those using larger search spaces in each group. We only define the parameter space for one `nonconfluence` processor due to the following reasons. First, although the sorted decomposition technique may decompose a TRS to several sub-TRSs, we only need to disprove confluence for a sub-TRS to disprove confluence for the original TRS [Felgenhauer *et al.*, 2015]. CSI's solution in CoCo for a nonconfluence problem also shows that we only need to prove nonconfluence for a sub-TRS to disprove confluence for the original TRS. Second, we want to invent a set of complementary strategies and then combine them in the approach explained in Section Strategy Combination in our paper. CSI's default strategy combines sequential and parallel execution to try various nonconfluence techniques and increase the execution speed. In contrast, our goal of defining a parameter space is simply to invent a technique suitable for a set of problems. The combination of invented strategies will be considered later. Therefore, our parameter space will produce a strategy like `NOTCR = nonconfluence -steps 0 -tcap -fun[10]` where the execution time of `nonconfluence` and its flags are decided by Grackle.

- `NOTCR` {`yes, no`}. Disprove confluence. It is sound because if we set it to `fail`, the strategy simply mainly tries to discover confluence proofs. It will not cause unexpected transformations.
- `nonconfluence_time` {`1, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 25, 30`}
- `nonconfluence_steps` {`0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 16, 25, 32`}. Number of rewrite steps that are performed from critical pairs to test terms nonconfluent [default: 2]. Critical pairs are explained in Section 1. It is sound as it only changes the size of the search space.
- `nonconfluence_width` {`-1, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 16`}. Width of search tree for rewrite sequences; -1 means unbounded [default: -1]. It is sound as it only changes the size of the search space.
- `nonconfluence_fun` {`yes, no`}. Use overlaps at function positions only. As explained in Section 1, an overlap roughly means that at a certain position, a subterm can be applied with two rewrite rules [Baader and Nipkow, 1998]. As explained in Section 1, the basic idea for disproving confluence is to discover an overlap, discover a critical pair from the overlap, and check the nonjoinability of the pair. This flag only determines where to find such an overlap and thereby is sound.
- `nonconfluence_var` {`yes, no`}. Use overlaps at variable positions only. As explained in the last flag, this flag only determines where to find such an overlap and thereby is sound.
- `nonconfluence_iter` {`-1, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 16`}. Specifies the maximum number of compatibility violations that should be solved. This guarantees that the procedure always terminates. Otherwise, it might happen that non-confluence check does not terminate. It is only used for tree automata technique. According to Theorem 4 in [Nagele *et al.*, 2017], for a critical pair (s, t), it first tries to respectively construct compatible tree automates $\mathcal{A}_1$ and $\mathcal{A}_2$. Then, it checks the non-joinability of term reachable from $\mathcal{A}_1$ and $\mathcal{A}_2$. The flag uses the tree automata completion technique [Korp and Middeldorp, 2009] to build tree automata, which solves compatibility violations during the constructions. If it cannot solve all compatibility violations, the flag fails and cannot disprove confluence. According to the `filter` function in csi/src/processors/src/confluence/nonconfluence.ml, when `-iter` is -1, it tries to solve all compatibility violations. However, the construction process may not terminate and fail. Other values only make the tree automate completion fails earlier.
- `nonconfluence_tcap` {`yes, no`}. Show nonconfluence by tcap (default on). It is sound as it is one of the nonconfluence techniques. It is explained in Lemma 1 of [Zankl *et al.*, 2011].

- `nonconfluence_tree` {yes, no}. Show nonconfluence by tree automata (default off). It is sound as it is one of the nonconfluence techniques used in NOTCR.. It is explained in Theorem 4 of [Zankl *et al.*, 2011]

- `nonconfluence_idem` {yes, no}. Show nonconfluence by idem (default off). It is sound as it is one of the nonconfluence techniques. Meanwhile, according to the function `idem` in csi/src/processors/src/confluence/nonconfluence.ml, it simply checks the nonjoinability of the reducts of the two terms in a critical pair. The nonjoinability is approximated via defined symbols. Moreover, it is used in the original NOTCR.

- `nonconfluence_nf` {yes, no}. Show no unique normal forms exist by finding distinct normal forms (default off). No unique normal forms imply nonconfluence [Baader and Nipkow, 1998]; hence, the flag is sound.

```
KB = (cr -kb;SN)!
RL = (rule_labeling \
  | rule_labeling -left)
DECPAR = ((shift -par; \
  decreasing -par)| \
  (shift -par -m 2; decreasing -par))
DECWLL = ((rule_labeling -left \
  -persist;decreasing)|DECPAR)
DDLAB = (LDH;(decreasing | \
  RL?;decreasing))!
```

- `KB` {yes, no}. Denote the Knuth-Bendix criterion [Knuth and Bendix, 1983]. It is sound because it is only a technique parallelly executed with other techniques.

- `DECPAR` {yes, no}. One technique in decreasing diagrams [Aoto *et al.*, 2014]. We cannot entirely understand it. Thus, there is only one boolean execution-controlling parameter. It is sound because DECPAR is simply used at the end of DECWLL. DECWLL is only used at the end of DD. DD is only a technique parallelly executed with other techniques. Setting DECPAR to `fail` only makes the strategy weaker in discovering proofs.

- `DECPAR_shift_m` {0, 1, 2, 4, 6}. Search for (minimal+m)-length joins. It changes the value of `-m` of the second `shift` in DECPAR. If $m < 0$, it does not search for joins. Otherwise, it first searches for the $minimal$-length of joins. After that, according to the value of `-m`, it searches for joins of length $minimal+m$. It is sound as it only changes the size of the search space. Meanwhile, different values of `-m` are used in DECPAR.

- `DDLAB` {yes, no}. One technique in decreasing diagrams. We cannot entirely understand it. Thus, there is only one boolean execution-controlling parameter.

```
COR1 = (DUP;DDLAB)!
COR2 = (SSTAR;DDLAB)!
COR3 = (REDEX;LDH;(decreasing \
 | rule_labeling?;decreasing))!
DDWLL = (LDHF;RL?;DECWLL)!
DD = (if left-linear then \
  (COR1 || COR2 || COR3 || \
  (cr -force;DDWLL)!) else fail)!
```

- `COR1` {yes, no}. One technique in decreasing diagrams [Aoto *et al.*, 2014]. We cannot entirely understand it. Thus, there is only one boolean execution-controlling parameter.
- `COR2` {yes, no}. One technique in decreasing diagrams [Aoto *et al.*, 2014]. We cannot entirely understand it. Thus, there is only one boolean execution-controlling parameter.
- `COR3` {yes, no}. One technique in decreasing diagrams [Aoto *et al.*, 2014]. We cannot entirely understand it. Thus, there is only one boolean execution-controlling parameter.
- `DD` {yes, no}. Techniques in decreasing diagrams [Aoto *et al.*, 2014].

```
CLOSED_LINEAR = (if linear \
  then cr -closed -redundant -m -1; \
    (closed -feeble \
      | closed -strongly 7) \
  else fail)
CLOSED_LEFT = (if left-linear \
 then ((cr -closed -redundant -m -1;\
    (closed -feeble \
     | closed -development \
     | closed -upside \
     | closed -outside)) \
    | cr -okui) else fail)
CLOSED = (CLOSED_LINEAR || \
 CLOSED_LEFT)!
```

- `CLOSED` {yes, no}. Test whether the critical pairs of a TRS are strongly or development closed [Huet, 1980; Nagele and Middeldorp, 2016].
- `CLOSED_LINEAR` {yes, no}
- `CLOSED_LINEAR_strongly` {-1, 1, 3, 5, 7, 9, 11}. Check critical pairs strongly closed (in $\leq n$ steps). It is sound since it only changes the number of rewrite steps before checking whether critical pairs are strongly closed. It determines the value of `-strongly 7` in CLOSED_LINEAR.
- `CLOSED_LEFT` {yes, no}. Test whether the critical pairs of a left-linear system are development closed [Van Oostrom, 1997].
- `CLOSED_LEFT_closed_strongly` {-1, 1, 3, 5, 7, 9, 11}. There are four `closed` processors with this flag in CLOSED_LEFT, we only explain one.

```
CR_AUX = (sorted -order | \
    (KB || ((((CLOSED \
      || DD) | add)2*)!)))*
KH = (cr -rt;SNRELATIVE; \
  kh -mace;CR_AUX)!
```

- CR_AUX_loop {1, 2, 3, 4, 5}. It determines the times of the application of CR_AUX.
- KH {yes, no}. Perform the confluence test for associative communicative (AC) problems by using the theorem of Klein and Hirokawa [Klein and Hirokawa, 2012].
- KH_mace {yes, no}. Use mace4 theorem prover if available. The yes value is sound since kh -mace is used in the default configuration. According to the csi/src/processors/src/confluence/kleinHirokawa.ml in CSI's source code, when it is set to no, no theorem prover will be invoked. According to the paper [Klein and Hirokawa, 2012], when no theorem prover is invoked, the method merely becomes weaker in constructing confluence proofs.

```
AT1 = (at -theorem 1; SN)!
AT2 = (at -theorem 2; SN)!
AT3 = (at -theorem 3; SN)!
AT = (AT2 || AT3)
```

- AT {yes, no}. The confluence test for associative-communicative (AC) problems by using the theorems of Aoto and Toyama [2012].
- AT2 {yes, no}
- AT2_theorem {1, 2, 3} Indicates which of the three theorems is used. By default, theorem 3 will be used. The three theorems are explained in [Aoto and Toyama, 2012] and are all sound. The value 1, 2, and 3 respectively correspond to Theorem 3.8, Theorem 3.18, and Theorem 3.28 in [Aoto and Toyama, 2012],
- AT2_bound {1, 2, 4, 8, 12, 14, 16, 24}. Indicates an upper bound for the number of rewrite rules. If the number of rewrite rules is $>= b$, then the processor ends and fails. By default, $b = 12$ will be used. It is sound as it only changes the size of the search space.
- AT3 {yes, no}
- AT3_theorem {1, 2, 3}
- AT3_bound {1, 2, 4, 8, 12, 14, 16, 24}

```
CPCS = (cr -cpcs; SNRELATIVE; \
 shift -lstar)
CPCS2 = (cr -cpcs2; SN)!
```

- CPCS {yes, no}. The processor cr -cpcs computes the critical pair closing system by Theorem 2.4 in [Oyamaguchi and Hirokawa, 2014]. It is sound since if it is set to fail, it simply does not transform the problem in AUTO_INNER.
- CPCS2 {yes, no}. The processor cr -cpcs2 computes the critical pair closing system by Theorem 2.11 in [Oyamaguchi and Hirokawa, 2014]. It is sound since it is parallelly executed with other techniques in AUTO_INNER0.

```
REDUNDANT_JS = (( \
 cr -force -redundant); \
 (redundant))
REDUNDANT_RHS = ( \
 (cr -m -1 -force -redundant); \
 (redundant -rhs))
REDUNDANT_DEL = ((cr -m -1 -force); \
 (redundant -remove 4))
```

A group of redundant rule techniques. REDUNDANT_FC is used in nonconfluence analysis, while the other three are mainly used for confluence analysis.

- REDUNDANT_JS_cr_m {-1, 0, 1, 2, 3, 4, 5}. Search for (minimal+m)-length joins. For a critical pair (s, t), According to csi/src/processors/src/transformation/redundant.ml and csi/src/rewriting/src/rewrite.ml, When $m = -1$, only critical pairs are returned, and no joins will be found. When $m = 0$, it tries to discover the minimal number M of rewrite steps, such that $\exists u, s \rightarrow^M u \wedge t \rightarrow^M u$. Then joins reachable from M steps are returned. When $m > 0$, joins reachable from $M + m$ steps are returned. Some redundant rule techniques use joins of critical pairs to generate redundant rules [Nagele *et al.*, 2015]. This parameter is sound since it only controls the length of joins to be generated.
- REDUNDANT_JS_redundant_size {-1, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 16, 32}. Only add rules whose size is less than n (default: -1, i.e., unrestricted). It is sound since values other than the default merely limit the number of redundant rules to generate.
- REDUNDANT_RHS_cr_m {-1, 0, 1, 2, 3, 4, 5}
- REDUNDANT_RHS_redundant_size {-1, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 16, 32}
- REDUNDANT_DEL_cr_m {-1, 0, 1, 2, 3, 4, 5}
- REDUNDANT_DEL_redundant_js {yes, no}. Add joining sequences of critical peaks as rules. It is sound as explained in Collary 6 and Section 5 in [Nagele *et al.*, 2015].
- REDUNDANT_DEL_redundant_development {-1, 1, 2, 3, 4, 5, 6}. Add rules to make critical peaks development closed. It is sound as explained in Collary 6 and Section 5 in [Nagele *et al.*, 2015].

- REDUNDANT_DEL_redundant_rhs {yes, no}. Add rules by rewriting right-hand sides 1 step. It is sound as explained in Collary 6 and Section 5 in [Nagele *et al.*, 2015].

- REDUNDANT_DEL_redundant_remove {-1, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}. Remove rules whose left- and right-hand sides are joinable in n steps. By default, it is -1, meaning no limitations on the number of rewrite steps. The parameter is sound as other values are weaker than the default in removing rules.

- REDUNDANT_DEL_redundant_reverse {-1, 1, 2, 3, 4, 5, 6}. Add reversible rules. It is sound as explained in Collary 6 and Section 5 in [Nagele *et al.*, 2015].

- REDUNDANT_DEL_redundant_size {-1, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 16, 32}

```
REDUNDANT_FC = ((cr -m -1 -force); \
  (redundant -narrowfwd \
   -narrowbwd -size 7))
```

- REDUNDANT_FC {yes, no}. It is sound because if it is set to fail, it simply does not do the redundant rule transformation for non-confluence analysis.

- REDUNDANT_FC_cr_m {-1, 0, 1, 2, 3, 4, 5}. The soundness has been explained before.

- REDUNDANT_FC_redundant_js {yes, no}. The soundness has been explained before.

- REDUNDANT_FC_redundant_development {-1, 1, 2, 3, 4, 5, 6}. The soundness has been explained before.

- REDUNDANT_FC_redundant_rhs {yes, no}. The soundness has been explained before.

- REDUNDANT_FC_redundant_narrowfwd {yes, no}. Use narrowing forwards to generate new rules. It is sound since it is in the original REDUNDANT_FC.

- REDUNDANT_FC_redundant_narrowbwd {yes, no}. Use narrowing backwards to generate new rules. It is sound since it is in the original REDUNDANT_FC.

- REDUNDANT_FC_redundant_size {-1, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 16, 32}. The soundness has been explained before.

```
SIMPLE = FULL( \
  (if right-linear \
   then if left-linear -ie \
    then if strongly-non-overlapping
      then succ -reason \
       ToyamaOyamaguchi95Cor22
      else fail else fail else fail)|\
  (if collapsing then fail else
    if shallow -ws then
      if strongly-non-overlapping then
        succ -reason \
          SakaiOyamaguchiOgawa14
      else fail else fail) | \
  fail)
```

Simple criteria for proving confluence [Sakai *et al.*, 2015; Toyama and Oyamaguchi, 1994]. The FULL keyword is a trick in the competition strategy for easily taking part in different categories of competitions in CoCo. For us, it means nothing and can be ignored. The only parameter is SIMPLE {yes, no}.

```
AC_SN = ((acrpo \
 ||  ackbo -ib 3 -ob 5 -q -nt -sc  \
 || ackbo -ib 3 -ob 5 -kv2)*[10])
AC = (cr -ac;AC_SN)!
```

- AC {yes, no}. Main techniques for proving confluence for AC problems.
- AC_time {1, 2, 4, 6, 8, 10, 15}
- AC_acrpo {yes, no}. Applies AC-Recursive Path Order [Yamada *et al.*, 2016].
- AC_acrpo_direct {yes, no}
- AC_acrpo_sat {yes, no}
- AC_acrpo_smt {yes, no}
- AC_ackbo {yes, no}. Apply standard, Korovin/Voronkov's, and Steinbachs AC-KBO [Yamada *et al.*, 2016]. There are two ackbo processors in AC_SN, we only explain the parameter for one for simplification. The flags -kv, kv2, and st respectively correspond to Korovin &Voronkov, KV', and Steinbach methods in Table 1 of [Yamada *et al.*, 2016]. When none of -kv, kv2, and st is used, by default, ackbo uses the AC-KBO method in Table 1.
- AC_ackbo_ac0 {yes, no}. In case of Steinbach's order, give AC symbols weight 0. Its soundness is explained in Theorem 3.3 [Yamada *et al.*, 2016] and the related footnote.
- AC_ackbo_direct {yes, no}. Try to finish the termination proof.
- AC_ackbo_ib {1, 2, 3, 4, 5, 6}. Defines the number of bits that can be used to represent the smallest number that appears in the intermediate results.

- AC_ackbo_kv2 {yes, no}. Use corrected Korovin and Voronkov's ordering. The paper [Yamada *et al.*, 2016] finds a bug in the original Korovin and Voronkov's ordering and has corrected it.
- AC_ackbo_ob {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, max}. Defines the number of bits that can be used to represent the largest number that appears in the intermediate results.
- AC_ackbo_q {yes, no}. Uses quasi-precedences. According to the annotation of the function quasi_adm in csi/src/processors/src/termination/orderings/ackbo.ml in CSI's source code, here, quasi-precedences mean that AC-symbols never equal to non-AC symbols. Using the flag only makes the order more strict and thereby is sound. Moreover, the ackbo processors have both use and do not use -q in AC_SN.
- AC_ackbo_sat {yes, no}. Uses SAT backend (default).
- AC_ackbo_sc {yes, no}. Uses subterm coefficients. It is sound since ackbo both employs it and ignores it in AC_SN. Moreover, its soundness is explained in Theorem 8.2 in [Yamada *et al.*, 2016].
- AC_ackbo_smt {yes, no}. Uses SMT backend.
- AC_ackbo_st {yes, no}. Use Steinbach's ordering. Its soundness is explained in Theorem 3.3 in [Yamada *et al.*, 2016].
- AC_ackbo_nt {yes, no}. Allow non-total precedences (not compatible with -kv). Precedences are defined between function symbols or constants. It is essential as explained in Example 5.11, sometimes we need to make two function symbols incomparable to obtain a working order. We use Grackle's forbidden mechanism to forbidde both -kv and -nt are employed by ackbo. According to AC_SN, the combination of -nt and AC-KBO is sound. According to Definition 4.1 and Example 5.11, the combination of -nt and kv2 is sound. According to Definition 3.1, the combination of -nt and st is sound.

```
AUTO_INNER0_DEL = (REDUNDANT_DEL?; \
 (CLOSED || DD || SIMPLE || KB \
  || AC || {GROUND}nono))3*!

AUTO_INNER0_CLOSED_DD_REDUNDANT =
 (((CLOSED || DD) \
   | REDUNDANT_RHS)3*! || \
  (CLOSED || DD) \
   | REDUNDANT_JS)3*!

AUTO_INNER0 = (GROUND || KB || AC || \
 AUTO_INNER0_DEL || \
 AUTO_INNER0_CLOSED_DD_REDUNDANT || \
 KH || AT || SIMPLE || CPCS2)
```

As explained in Section 4, AUTO_INNER0 parallelly executes a set of techniques. When searching for parameters, we group some techniques in AUTO_INNER0 into AUTO_INNER0_DEL and AUTO_INNER0_CLOSED_-DD_REDUNDANT. We perform this modification since techniques inside the two groups have two common features that not exist in the other techniques in AUTO_INNER0. Grouping them is helpful for using a boolean execution-controlling flag to determine whether to execute the two groups of techniques. The first distinct feature is that both groups utilize redundant rule techniques. Moreover, both groups consist of multiple techniques which follow a specific invocation procedure.

- AUTO_INNER0 {yes, no}.
- AUTO_INNER0_time {2, 4, 6, 8, 12, 16, 30, 60}. Control the running time of AUTO_INNER0.
- AUTO_INNER0_DEL {yes, no}.
- AUTO_INNER0_DEL_loop {1, 2, 3, 4, 5, 6, 7}
- AUTO_INNER0_GROUND {yes, no}. Control whether to execute GROUND in AUTO_INNER0.
- AUTO_INNER0_KB {yes, no}. Control whether to execute KB in AUTO_INNER0.
- AUTO_INNER0_AC {yes, no}. Control whether to execute AC in AUTO_INNER0.
- AUTO_INNER0_CLOSED_DD_REDUNDANT {yes, no}
- AUTO_INNER0_CLOSED_DD_REDUNDANT_loop {1, 2, 3, 4, 5, 6, 7}. Control the repeated application times of ((CLOSED || DD) | REDUNDANT_RHS)3*! || (CLOSED || DD) | REDUNDANT_JS).
- AUTO_INNER0_CLOSED_DD_REDUNDANT_inner_loop {1, 2, 3, 4, 5, 6, 7}. Control the repeated application times of (CLOSED || DD) | REDUNDANT_RHS.
- AUTO_INNER0_KH {yes, no}. Control whether to execute KH in AUTO_INNER0.
- AUTO_INNER0_AT {yes, no}. Control whether to execute AT in AUTO_INNER0.
- AUTO_INNER0_SIMPLE {yes, no}. Control whether to execute SIMPLE in AUTO_INNER0.
- AUTO_INNER0_CPCS2 {yes, no}. Control whether to execute CPCS2 in AUTO_INNER0.

```
AUTO_INNER = (AUTO_INNER0[30] \
  | CPCS[5]2*)! \
    | ({AUTO_INNER0[30]}nono \
      | CPCS[5]2*)2*!
```

- AUTO_INNER {yes, no}
- AUTO_INNER_CPCS_time1 {1, 2, 3, 4, 5, 6, 7, 8, 9}. Control the running time of the first CPCS.

- AUTO_INNER_CPCS_time2 {1, 2, 3, 4, 5, 6, 7, 8, 9} Control the running time of the second CPCS.

- AUTO_INNER_CPCS_loop1 {1, 2, 3, 4, 5, 6, 7}. Control the number of repeated application times of the first CPCS.

- AUTO_INNER_CPCS_loop2 {1, 2, 3, 4, 5, 6, 7}. Control the number of repeated application times of the second CPCS.

- AUTO_INNER_loop {1, 2, 3, 4, 5, 6, 7}. Control the number of repeated application times of the compound processor (AUTO_INNER0[30] | CPCS[5]2*)! | (AUTO_INNER0[30]nono | CPCS[5]2*).

```
AUTO = (if trs then (\
 sorted -order*;(AUTO_INNER \
 || (NOTCR | REDUNDANT_FC)3*!) \
 ) else fail)
```

- AUTO_sorted_order {yes, no}. Decompose a problem due to sorted information [Felgenhauer *et al.*, 2015]. The flag -order tries order-sorted decomposition.

- AUTO_sorted_ms {yes, no}. Try many-sorted decomposition. It is weaker than -order in decomposing TRSs [Felgenhauer *et al.*, 2015].

- NOTCR_loop {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}. Control the number of repeated applications of (NOTCR | REDUNDANT_FC)3*!, default 3.

4.3 Forbidden Parameters

Grackle can forbid the occurrence of certain parameters. We have listed the parameters we forbid below. We forbid them either to confirm soundness or to reduce the size of the parameter search space.

```
{AC_ackbo1_nt=yes, AC_ackbo1_kv=yes}
{AC_ackbo2_nt=yes, AC_ackbo2_kv=yes}
{MAINTRS_edg_nl=yes,
  MAINTRS_edg_gtcap=no}
{AUTO_INNER_sorted_order=no,
  AUTO_INNER_sorted_ms=no}
{DIRECTTRS_kbo_pbc=no,
  DIRECTTRS_kbo_eq=yes}
{DIRECTTRS_kbo_pbc=no,
  DIRECTTRS_kbo_minp=yes}
{DIRECTTRS_kbo_pbc=no,
  DIRECTTRS_kbo_minw=yes}
{DIRECTTRS_kbo_sat=no,
  DIRECTTRS_kbo_smt=no,
  DIRECTTRS_kbo_rat=2}
{DIRECTTRS_kbo_sat=no,
  DIRECTTRS_kbo_smt=no,
  DIRECTTRS_kbo_rat=3}
{DIRECTTRS_kbo_sat=no,
  DIRECTTRS_kbo_smt=no,
  DIRECTTRS_kbo_rat=4}
{MAINTRS_kbo_sat=no,
  MAINTRS_kbo_smt=no,
  MAINTRS_kbo_rat=2}
{MAINTRS_kbo_sat=no,
  MAINTRS_kbo_smt=no,
  MAINTRS_kbo_rat=3}
{MAINTRS_kbo_sat=no,
  MAINTRS_kbo_smt=no,
  MAINTRS_kbo_rat=4}
{SNRELATIVE_poly_neg=yes,
  SNRELATIVE_poly_nl=no,
  SNRELATIVE_poly_nl2=no}
```

5 Examples of Invented Strategy

Figure 1 presents the invented strategy csi-0d232dbb588232c4fa2a8db3585ab8b2d0c28c44bdbcfb555-98ae901. It follows the basic structure of the original competition strategy. Boolean-execution controlling flags disable some sub-strategies be replacing their definitions to fail. In SNRELATIVE_STEP, the values of -ib, -ob, and -dim are chosen by Grackle, which differs from those in the original SNRELATIVE_STEP in Seciton 3.

6 Grackle's Initial Strategies

As explained in Section 3, AUTO_INNER0 parallelly executes a set of strategies mainly for confluence. We separate each of them as an initial strategy. This means we use the default competition strategy except that we change NOTCR to fail and REDUNDANT_FC to fail via parameters defined in Section 4. Moreover, we respectively change the definition of AUTO_INNER0 to one of the items below and obtain nine initial strategies.

- AUTO_INNER0 = GROUND
- AUTO_INNER0 = KB
- AUTO_INNER0 = AC

Figure 1: The invented strategy csi-0d232dbb588232c4fa2a8db3585ab8b2d0c28c44bdbcfb55598ae901

```
AUTO = (if trs then (sorted -order*; \
  (AUTO_INNER || (NOTCR | REDUNDANT_FC)3*!)) else fail)

AUTO_INNER = (AUTO_INNER0[30] | CPCS[5]2*)! \
  | ({AUTO_INNER0[30]}nono | CPCS[5]2*)2*!

AUTO_INNER0 = (AUTO_INNER0_GROUND || AUTO_INNER0_KB || AUTO_INNER0_AC \
  || AUTO_INNER0_DEL || AUTO_INNER0_CLOSED_DD_REDUNDANT || AUTO_INNER0_KH \
  || AUTO_INNER0_AT || AUTO_INNER0_SIMPLE || AUTO_INNER0_CPCS2 || fail)

AUTO_INNER0_AC = fail

AUTO_INNER0_AT = fail

AUTO_INNER0_CLOSED_DD_REDUNDANT = fail

AUTO_INNER0_CPCS2 = fail

AUTO_INNER0_DEL = fail

AUTO_INNER0_GROUND = fail

AUTO_INNER0_KB = fail

AUTO_INNER0_KH = fail

AUTO_INNER0_SIMPLE = SIMPLE

CPCS = (cr -cpcs; SNRELATIVE; shift -lstar)

NOTCR = fail

REDUNDANT_FC = fail

SIMPLE = FULL((if right-linear then if left-linear -ie then \
    if strongly-non-overlapping then succ -reason ToyamaOyamaguchi95Cor22 \
  else fail else fail else fail)
  | (if collapsing then fail else if shallow -ws then \
    if strongly-non-overlapping then succ -reason SakaiOyamaguchiOgawa14 \
    else fail else fail) | fail)

SNRELATIVE = (SNRELATIVE_STEP[5]*)

SNRELATIVE_STEP = (lpo -quasi \
  || (matrix -ib 6 -ob 6 | matrix -dim 2 -ib 2 -ob 2 \
    | matrix -dim 3 -ob 2 | arctic -dim 2 -ib 2 -ob 8) \
  || (if duplicating then fail else \
    (bounds -rt || bounds -rt -qc))[1] \
  || poly -heuristic 1 -ib 2 -nl2 -ob 4)
```

- `AUTO_INNER0 = AUTO_INNER0_DEL`
- `AUTO_INNER0 = AUTO_INNER0_CLOSED_-DD_REDUNDANT`
- `AUTO_INNER0 = KH`
- `AUTO_INNER0 = AT`
- `AUTO_INNER0 = SIMPLE`
- `AUTO_INNER0 = CPCS2`

We also separate each strategy in the nonconfluence analysis strategy `NOTCR` as an initial strategy. This means we use the default competition strategy except that we change `AUTO_INNER` to `fail` using parameters defined in Section 4. Meanwhile, we respectively change the definition of `NOTCR` to one of the items below and obtain 14 initial strategies.

- `NOTCR = nonconfluence -steps 0 -tcap -fun[10]`
- `NOTCR = nonconfluence -steps 2 -tcap -fun[10]`
- `NOTCR = nonconfluence -steps 25 -width 1 -tcap -fun[10]`
- `NOTCR = nonconfluence -steps 2 -idem -fun[10]`
- `NOTCR = nonconfluence -steps 2 -tcap -var[10]`
- `NOTCR = nonconfluence -steps 25 -width 1 -tcap -var[10]`
- `NOTCR = nonconfluence -steps 0 -tree -fun[10]`
- `NOTCR = nonconfluence -steps 0 -tree -var[10]`
- `NOTCR = nonconfluence -steps 1 -tree -fun[10]`
- `NOTCR = nonconfluence -steps 1 -tree -var[10]`
- `NOTCR = nonconfluence -steps 2 -tree -fun[10]`
- `NOTCR = nonconfluence -steps 2 -tree -var[10]`
- `NOTCR = nonconfluence -steps 25 -width 1 -tree -fun[10]`
- `NOTCR = nonconfluence -steps 25 -width 1 -tree -var[10]`

To generate the dataset, we further add two other strategies besides the initial strategies. This means we use the default competition strategy except that we change `NOTCR` to `fail` and `REDUNDANT_FC` to `fail`. Moreover, we respectively change the definition of `AUTO_INNER0` to one of the items below and obtain two strategies.

- `AUTO_INNER0 = CLOSED`
- `AUTO_INNER0 = DD`

We particularly extract them since `CLOSED` and `DD` are combined with redundant rule techniques in `AUTO_INNER0_DEL` and `AUTO_INNER0_CLOSED_DD_REDUNDANT` among the initial strategies. Moreover, `AUTO_INNER0_DEL` and `AUTO_INNER0_CLOSED_DD_REDUNDANT` parallelly execute `CLOSED` and `DD`. If we do not use them for labeling, we will not be able to understand whether a TRS is mastered only by `CLOSED` or `DD`.

In the augmented dataset, we notice eight initial strategies can only prove confluence, and 14 initial strategies can only prove non-confluence. One can prove both.

7 Strategy Combination

To combine several strategies, we first assign a time limit for each using the method in Section 3.2. Each strategy is written into a document. We use a Python program to invoke CSI with each strategy document within the assigned time limit. The advantage of writing each strategy into a document is avoiding naming conflict since our strategy invention does not change the name of strategies. We only change the parameters. Every strategy document is still invoked from the definition `AUTO` via Python.

When we choose a strategy, we make sure this strategy can solve the largest number of problems in the training data unsolvable by previously chosen strategies.

When we run experiments on the ARI-COPS database and use four CPUs for each CSI execution, the assigning time limit for each strategy is presented below in the format of *strategy, time by seconds*.

- csi-0ccc287f955294ea83afdc800030b453a8e37b1ee371-57d83dcf04eb, 0.5
- csi-9f19c01a538808477a8980f80d3d2face7303ce6f058-f8b6170f9461, 0.5
- csi-224dc655e2c823145f6a545fa78601d91e647dcfeffb-317d8f57b340 0.5
- csi-2ebf247f07a5706eaf7a7a399dcd47bb52bbb40694af-808ec8624cf1, 0.5
- csi-5deb4704be2267c6724151a6417770ceb5ffc4799560-7bbf3a169c28, 1
- csi-c3e1c6423a6b61ebf5d3b3c3999fca2d098d4461b23a4-b50d77171b3, 2
- csi-af40c043b23e299af5c85347534fe62d2c963a8f7d6b0-dac873b8846, 3
- csi-5deb4704be2267c6724151a6417770ceb5ffc47995607-bbf3a169c28, 3
- csi-a4390ec528e4f6c2616765939087c14c1ca0bd47cf1dd-d3b6d055ba0, 3
- csi-112ecf2d9f970dad13f1bb4e09c25c2b51385f7324062-deb49084b46, 5
- csi-5c115df06bbafa913b50c0fd62a28f53e6739d1358646-b7f3cea19bf, 6
- csi-14c86024498e9a3ce1b4b79795e5ab8f6c49a19f5df8c-924ce82f577, 6

- csi-ec0fb3f8ce9f2d586b23f5cd9c4a399845157f2854ef5-f8e2f9d3f3a, 6
- csi-af40c043b23e299af5c85347534fe62d2c963a8f7d6b0-dac873b8846, 7
- csi-3f5a5b338144451e2d7e22bcc184940e796a74510ec87-4966c88b93d, 8
- csi-fc9509efa9ad65cb01f89548200a9ae5480ef616e48df-fb3e0d790a2, 8

When using one CPU, the time splits are presented below

- csi-1789a05ac5304d685e89b710f75edd8a273fc474526-6a3c7faadba34, 0.5,
- csi-c3e1c6423a6b61ebf5d3b3c3999fca2d098d4461b23-a4b50d77171b3, 0.5,
- csi-075d10c4a77649791eac301cda263f977e1a66186e7-20906f67a2c34, 0.5,
- csi-ddc25446660daf33870ef6991406ec360404a5af909-928d5ea21794f, 0.5,
- csi-6c8b5e7b4af3c2970275389b01434b25395a13d8f72-4e4a8aba48e24, 0.5,
- csi-b3ab9764cd4f0717a3037a5849e47d41c56af511e84-d4a72659e0829, 0.5,
- csi-0ccc287f955294ea83afdc800030b453a8e37b1ee37-157d83dcf04eb, 0.5,
- csi-224dc655e2c823145f6a545fa78601d91e647dcfeff-b317d8f57b340, 0.5,
- csi-a8f7d2a391815f9c401550acbb694bae346f443a43a-098c5e072aae9, 0.5,
- csi-e2877b030e4118e870207e14ef1a44e240c33e16a0e-79c8f57adaacd, 0.5,
- csi-ddc25446660daf33870ef6991406ec360404a5af909-928d5ea21794f, 1,
- csi-af40c043b23e299af5c85347534fe62d2c963a8f7d6-b0dac873b8846, 4,
- csi-d56e502247dfc3854f0a5360649b5be5357e2b60693-fdfdc40dc5527, 8,
- csi-0e7e3ba5c042fabdf8151556dc8b26717d98bcd49c2-3193ec7f29d33, 9,
- csi-36920a8f429f37ae80839d40e3cd805ef096f1aafbb-2dd6d7b845531, 10,
- csi-5deb4704be2267c6724151a6417770ceb5ffc479956-07bbf3a169c28, 11,
- csi-33d868984681a7cc4455c15daaa16bf675ba3056c6f-f1f186057216e, 12

When using one CPU for the augmented dataset, the time splits are presented below

- csi-04117237bacc588c78fade8bc184e29c2f22ce95f0f-3d8fc051128e4, 0.5
- csi-a4390ec528e4f6c2616765939087c14c1ca0bd47cf1-ddd3b6d055ba0, 0.5
- csi-379e7f8304b587a34081a91ae8a1624f0fb9a83ef83-2090b72713d9f, 0.5
- csi-931e7ced256711e95501f485d5b130953464871eebe-6e41dd72957e9, 0.5
- csi-c3e1c6423a6b61ebf5d3b3c3999fca2d098d4461b23-a4b50d77171b3, 0.5
- csi-af40c043b23e299af5c85347534fe62d2c963a8f7d6-b0dac873b8846, 0.5
- csi-224dc655e2c823145f6a545fa78601d91e647dcfeff-b317d8f57b340, 0.5
- csi-5d8720a14f25d32a3a4cecd80b5cc818cfafce6f753-b2acd33280292, 0.5
- csi-aecfb984dbfbfe0cb3ba80326dcc23b15a9c2f0a01c-1e9620916c9e4, 0.5
- csi-0ccc287f955294ea83afdc800030b453a8e37b1ee37-157d83dcf04eb, 0.5
- csi-f660fd4bbdf8ee1f7e501b27bc2a8223d96e0f47cda-55d550d72973c, 0.5
- csi-f660fd4bbdf8ee1f7e501b27bc2a8223d96e0f47cda-55d550d72973c, 0.5
- csi-f660fd4bbdf8ee1f7e501b27bc2a8223d96e0f47cda-55d550d72973c, 0.5
- csi-f660fd4bbdf8ee1f7e501b27bc2a8223d96e0f47cda-55d550d72973c, 0.5
- csi-aa24b5e64bf83e78a707b01ad35bc3b6e22e06f9402-a758631eee1bb, 1
- csi-af40c043b23e299af5c85347534fe62d2c963a8f7d6-b0dac873b8846, 6
- csi-f1d55b73677f250ea72db5a2658ffe0f92fc0197ea7-9717d372870d1, 7
- csi-1bccc2cf890e2f92f021b591ec34371ebdd2a68741d-08de7fff0924e, 8
- csi-5deb4704be2267c6724151a6417770ceb5ffc479956-07bbf3a169c28, 9
- csi-a4390ec528e4f6c2616765939087c14c1ca0bd47cf1-ddd3b6d055ba0, 10
- csi-36920a8f429f37ae80839d40e3cd805ef096f1aafbb-2dd6d7b845531, 12

When using four CPU for the augmented dataset, the time splits are presented below

- (csi-1f0dd05ed5c19b06c8c11b6dda27bd743c3b691c8f-f22150acb2c5e3, 0.5),
- (csi-95e8deb6c1089354822fc19425ea6cab098fff49ea-9ec5afb65da9da, 0.5),
- (csi-f660fd4bbdf8ee1f7e501b27bc2a8223d96e0f47cd-a55d550d72973c, 0.5),
- (csi-36920a8f429f37ae80839d40e3cd805ef096f1aafb-b2dd6d7b845531, 0.5),
- (csi-042b74efcc92c96a15db35030eea542a97329bf422-cced8e7fc07453, 0.5),
- (csi-224dc655e2c823145f6a545fa78601d91e647dcfef-fb317d8f57b340, 0.5),

- (csi-a8f7d2a391815f9c401550acbb694bae346f443a43-a098c5e072aae9, 0.5),
- (csi-4c4f985d3c24d4e988879b93a20ba27aef4d1b5e43-1cff21ed9c304f, 0.5),
- (csi-007a111242d4e9cc17cbb487d338934ef25edfe677-bec379b08b5002, 0.5),
- (csi-007a111242d4e9cc17cbb487d338934ef25edfe677-bec379b08b5002, 0.5),
- (csi-042b74efcc92c96a15db35030eea542a97329bf422-cced8e7fc07453, 1),
- (csi-5704ae7a7f958a112c4ab6707b5b6708c839a7a23a-8927ff774d5a38, 1),
- (csi-af40c043b23e299af5c85347534fe62d2c963a8f7d-6b0dac873b8846, 4),
- (csi-535aa6a8940b19cbf86cda227104df4ff1f3c76e36-eb17aa89585b91, 4),
- (csi-5deb4704be2267c6724151a6417770ceb5ffc47995-607bbf3a169c28, 8),
- (csi-a4390ec528e4f6c2616765939087c14c1ca0bd47cf-1ddd3b6d055ba0, 9),
- (csi-3f5a5b338144451e2d7e22bcc184940e796a74510e-c874966c88b93d, 12),
- (csi-953d891df68f6a2fd85d53da81602b86b0f04a395d-a9aa547b02a4a3, 16),

8 Certification

Besides carefully designing the parameter space of Grackle, we also perform various verification procedures to ensure the soundness of the invented strategies.

8.1 Proof Consistence Checking

One typical way to verify the correctness of proofs in CoCo is to check whether the proofs of a prover are consistent with other provers. Here, the consistency means that we do not prove confluence (non-confluence) for a problem for which other provers prove its non-confluence (confluence). We check whether the proofs found by invented strategies are consistent with all provers in CoCo. The proofs found by Grackle are depicted in the final portfolio *grackle.flee*. The check is done by stats/dif_coco_grackle.py, which compares the difference between results in *grackle.flee* and the results obtained by CSI in CoCo2024. For proofs found in *grackle.flee* but not by CSI in CoCo 2024, we manually check the consistency between them and proofs of all provers in the previous CoCo competitions. We also confirm that the proofs obtained by the unified strategies are consistent with all provers in all CoCo competitions. This is done by stats/consistency.py in our code.

8.2 Certifying Newly Found Proofs

We run CeTA for each problem solved by invented strategies but not by CSI in CoCo. If it can be certified by CeTA, we trust the results. Otherwise, we manually look at the error information to see whether it is really an error and try to reproduce the proof and the certification error using the strategy defined in CSI's competition strategy. We aim to understand what changes they perform to the original strategy lead to the proofs. From the analysis, we either slightly modify the sub-strategy defined in the competition strategy or directly use some existing sub-strategies to produce the same answers as the invented strategies. These modifications that lead to the answers are employed in the corresponding invented strategies, which are small and sound according to our knowledge of term rewriting. We also check the certification errors output by CeTA to figure out whether they are indeed errors or just caused by limitations of CSI and CeTA. The statistics of the certifications are shown in Table 2 of the paper.

When we use four CPUs per CSI execution on ARI-COPS, we prove the following problems that are unprovable by CSI in CoCo. We analyze each of them. The format is *(strategy, newly proved problems in ARI-COPS, corresponding problems in COPS)*. The results `CERTIFIED` means the proof is certified by CeTA.

- (csi-e5535657f8e54081f79c2291ebad9d81992f6e7248-49f0fa92a83cc9, YES, 1499.ari, 1652.trs). The output is `CERTIFIED`.
- (csi-9f19c01a538808477a8980f80d3d2face7303ce6f0-58f8b6170f9461, YES, 879.ari, 1024.trs. The output is `./csi: XML output is not supported for this method`. The reason why CeTA fails to certify is CeTA does not support the certification of Aoto-Toyama criteria. It can be proved by `(at -bound 16; SN)!`. The strategy only changes the value of the `-bound` flag for `AT3`, which has been used in the original competition strategy. Using `(at -bound 16; SN)!` leads to the same certification error as the invented strategy.
- (csi-9f6172de97a8148d22ba4b910ba8b16dccd0196c276-c568d9ab2c0b5, NO, 852.ari (997.trs), 846.ari (991.trs)) CeTA cannot support the verification of `nonconfluence -idem`. However, the essential for solving such two problems is the usage of `redundant -development 6`, which is discovered by Grackle. Two problems can be solved if we change `redundant -narrowfwd -narrowbwd -size 7` in `REDUNDANT_FC` to `redundant -development 6 -size 7`. Moreover, they are certifiable.
- csi-beb87b539aa3911f6c65d5e2a97ef40cb45898dfafc-7283f152a217a, YES, 794.ari, 939.trs, UNSUPPORTED. CeTA cannot certify AoTo-Toyama criteria. We can use `AT` defined in the competition strategy to prove it. Using `AT` leads to the same certification error as the invented strategy.
- csi-ec0fb3f8ce9f2d586b23f5cd9c4a399845157f2854e-f5f8e2f9d3f3a, YES, 167.ari, 170.trs. UNSUPPORTED `Fatal: parse error on <acRuleRemoval> at [trsTerminationProof, wcrAndSN, crProof, redundantRules, crProof, proof, certificationProblem]`. CeTA does not support AC confluence proving techniques;

however, they are used in CSI's original competition strategy. From the invented strategy, we learn that it can be proven with two modifications to the original competition strategy. First, increase the number of repeated applications from two to five in AUTO_INNER. It leads to AUTO_INNER = (AUTO_INNER0[30] | CPCS[5]2*)! | (AUTO_INNER0[30]nono | CPCS[5]2*)5*!. Second, only run a subset of techniques in AUTO_INNER0. It leads to AUTO_INNER0 = (REDUNDANT_DEL?;(CLOSED || DD || SIMPLE || KB || AC || GROUNDnono))3*! If we only change AUTO_INNER0 and AUTO_INNER as mentioned above, CeTA can produce the same certification error as the invented strategy.

- (csi-b994d65167954d35cdd2b7a70646a4f8d550ec9e0de-bc650285a0d90, YES, 158.ari, 160.trs). Ackbo: no XML output for SCFs. CSI cannot output a certificate for the ackbo processor with the flag -sc. The soundness of it has been explained in Section 4. Moreover, CSI cannot produce a certificate for the CPCS transformation. We can prove it by changing the definition of AUTO_INNER0 to AUTO_INNER0 = (REDUNDANT_DEL?;(AC))3*!, which is used in the original AUTO_INNER0. The new definition does not parallelly execute all techniques in AUTO_INNER0 and makes the execution faster. Using AUTO_INNER0 = (REDUNDANT_DEL?;(AC))3*! leads to the same certification error as the invented strategy. It is not solved by CSI in CoCo 2024 but was solved by CSI in the previous CoCo competitions.
- csi-b3ab9764cd4f0717a3037a5849e47d41c56af511e84-d4a72659e0829, YES, 1500.ari, 1653.trs. CPCS cannot be certified by CeTA. It can be proven by CPCS*. CPCS is defined in the competition strategy and the soundness is guaranteed. Using CPCS* leads to the same certification error as the invented strategy. It is not solved by CSI in CoCo 2024 but was solved by CSI in the previous CoCo competitions.

When we use one CPU per CSI execution on ARI-COPS, we prove the following problems that are unprovable by CSI in CoCo.

- csi-0d382fd14a431f6befe533e561bbaf68c777bf48b92-ccbbea5fd4346, NO, 449.ari, 540.trs, CERTIFIED.
- csi-9e016d5ab9730720dcd426c5368545f09f0f5b9b464-9cd6af41284ad, YES, 463.ari, 554.trs. the critical pair g(h(f(f(b, b), b))) <- . -> g(h(h(f(f(h(k(k(b, b), b)), h(k(k(b, b), b))), h(k(k(b, b), b)))))) is not (almost) parallel closed within None steps. hence the following TRS is not (almost) parallel closed. CeTA cannot certify the cr -okui technique. We can prove it if we change AUTO_INNER0 to AUTO_INNER0 = (REDUNDANT_DEL?;(CLOSED))3*! || ((CLOSED | REDUNDANT_RHS)3*! || (CLOSED) | REDUNDANT_JS)3*!, which is used in the original definition of AUTO_INNER0. The modified definition causes the same certification error as the invented strategy.
- csi-0e7e3ba5c042fabdf8151556dc8b26717d98bcd49c2-3193ec7f29d33, YES, 166.ari, 169.trs. ./csi: XML output is not supported for this method. CSI cannot output certificates for Aoto-Toyama criteria. It can be proven by (at -bound 16 -theorem 2; SN)! where SN is defined in the original competition strategy. We only change -bound 16 to increase the search space as explained in Section 4. The strategy (at -bound 16 -theorem 2; SN)! causes the same certification error as the invented strategy.
- csi-aabfb22bdd5b365b18568e462dd644b3a94146e63d2-c44ff35c98a2c, NO, 846.ari(991.trs), 852.ari(997.trs), CERTIFIED.
- (csi-d42ec2e614b9f4287137c1772a4a13176783da5195-2c630b016bc7c4, YES, 158.ari, 160.trs). Ackbo: no XML output for SCFs. CSI cannot output a certificate for the ackbo processor with the flag -sc. The soundness of it has been explained in Section 4. Moreover, CSI cannot produce a certificate for the CPCS transformation. We can prove it by changing the definition of AUTO_INNER0 to AUTO_INNER0 = (REDUNDANT_DEL?;(AC))3*!, which is used in the original AUTO_INNER0. The new definition does not parallelly execute all techniques in AUTO_INNER0 and makes the execution faster. It is not solved by CSI in CoCo 2024 but was solved by CSI in the previous CoCo competitions. The strategy AUTO_INNER0 = (REDUNDANT_DEL?;(AC))3*! causes the same certification error as the invented strategy.
- csi-b3ab9764cd4f0717a3037a5849e47d41c56af511e84-d4a72659e0829, YES, 1500.ari, 1653.trs. CPCS cannot be certified by CeTA. It can be proven by CPCS*. CPCS is defined in the competition strategy and the soundness is guaranteed. It is not solved by CSI in CoCo 2024 but was solved by CSI in the previous CoCo competition. CPCS* causes the same certification error as the invented strategy.

8.3 Certifying Strategies on Mastered Problems

For every invented strategy in the ARI-COPS dataset, we run it on the problems it matsered and try to certify the proofs. The problems mastered by each strategy is calculated by Grackle. Since the outputs of CeTA on such problems are indeed lengthy, we do not present them in the technical appendix. The outputs exist in the attached code. We refer readers to read such logs for details. As explained in the main paper, CeTA may fail to certify the proofs due to several reasons. We manually check whether the outputs indeed denote errors. We have not found any unsoundness. The typical reasons why CeTA's rejection information does not indicate unsoundness are shown below.

- Fatal: parse error on <ac> at [statusPrecedenceEntry, statusPrecedence, pathOrder, redPair, orderingConstraintProof, ruleRemoval, trsTerminationProof, wcrAndSN, crProof, proof, certificationProblem]. CeTA cannot verify AC processors.

- Fatal: parse error on <acRuleRemoval> at [trsTerminationProof, wcrAndSN, crProof, redundantRules, crProof, proof, certificationProblem]. CeTA does not support AC confluence proving techniques
- Error in checking parallel closedness for the rewrite system ... The critical pair XXX is not (almost) parallel closed within None steps. hence the following TRS is not (almost) parallel closed. CSI outputs an empty certificate for the CPCS transformation and Church Rosser Transformation Processor (okui), confusing CeTA there are no proof steps.
- Fatal: parse error on <unknownAssumption> at [proof, certificationProblem]. CSI uses a theorem that is not supported by CeTA.
- ./csi: order-sorted decomposition: xml proof not supported. CeTA does not support order-sorted decomposition.
- Fatal: parse error on text element "-1" at [stronglyClosed, crProof, redundantRules, crProof, proof, certificationProblem]. CSI implements some development closedness techniques that cannot be verified by CeTA.
- Fatal: parse error on <uncurry> at [proof, certificationProblem]. Uncurry is not supported by CeTA.
- Could not infer that X and Y are not joinable, could not ensure closure under rewriting for first automaton, problem when ensuring (state-)compatibility of TRS with TA. The processor nonconfluence -tree cannot be verified. Need to use nonconfluence -tree -cert.
- Fatal: parse error on <magic> at [nonJoinableFork, crDisproof, proof, certificationProblem]. The technique nonconfluence -idem is not supported by CeTA.
- Error when closing critical pairs of rules, C not a subsystem of R, hence the following TRS is not critical pair closing rewrite system. The technique cr -cpcs2 is not certifiable. It should be changed to cr -cpcs2 -cpcscert for certification. But cr -cpcs2 is used in the original competition strategy.
- Error below strong normalization + wcr; R is not empty in the following termination-problem. The usage of AC processors makes CSI output an empty proof; thus, confusing CeTA.
- ./csi: MatrixInterpretation.fprintfx: XML output not supported expecting "<", but found: '' CPCS is not supported, and sometimes outputs entire empty certificates.
- ./csi: XML output is not supported for this method. CSI does not implement the functions to output the certificates for some processors.
- parser error : Excessive depth in document. The certificate is too large for CeTA to parse.
- Ackbo: no XML output for SCFs CSI cannot output a certificate for ackbo -sc. However it is used in the original competition strategy, and its soundness has been explained in Section 4.
- ./csi: not an integer. CSI cannot generate a certificate if kbo uses rational weights. The soundness of rational weights is explained in Section 4
- could not apply the reduction pair processor with the following polynomial interpretation over polynomial interpretation. CeTA cannot certify the flag -heuristic 1 for the poly processor. But poly -heuristic 1 is used in the original competition strategy, and its soundness has been explained in Section 4. From the proofs, we know we can use DD to prove the problems, which are defined in the competition strategy. We can also reproduce the certification error if we only use DD and CeTA.

References

[Aoto and Toyama, 2012] Takahito Aoto and Yoshihito Toyama. A reduction-preserving completion for proving confluence of non-terminating term rewriting systems. *Logical Methods in Computer Science*, 8, 2012.

[Aoto *et al.*, 2014] Takahito Aoto, Yoshihito Toyama, and Kazumasa Uchida. Proving confluence of term rewriting systems via persistency and decreasing diagrams. In *Rewriting and Typed Lambda Calculi: Joint International Conference, RTA-TLCA 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings 25*, pages 46–60. Springer, 2014.

[Baader and Nipkow, 1998] Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge university press, 1998.

[Bezem *et al.*, 2003] Marc Bezem, Jan Willem Klop, and Roel de Vrijer. *Term rewriting systems*. Cambridge University Press, 2003.

[Endrullis *et al.*, 2008] Jörg Endrullis, Johannes Waldmann, and Hans Zantema. Matrix interpretations for proving termination of term rewriting. *Journal of Automated Reasoning*, 40:195–220, 2008.

[Felgenhauer *et al.*, 2015] Bertram Felgenhauer, Aart Middeldorp, Harald Zankl, and Vincent Van Oostrom. Layer systems for proving confluence. *ACM Transactions on Computational Logic (TOCL)*, 16(2):1–32, 2015.

[Felgenhauer, 2012] Bertram Felgenhauer. Deciding confluence of ground term rewrite systems in cubic time. In *23rd International Conference on Rewriting Techniques and Applications (RTA'12)(2012)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2012.

[Gebhardt *et al.*, 2007] Andreas Gebhardt, Dieter Hofbauer, and Johannes Waldmann. Matrix evolutions. In *Proc. Workshop on Termination, Paris*, 2007.

[Giesl *et al.*, 2005a] Jürgen Giesl, René Thiemann, and Peter Schneider-Kamp. The dependency pair framework: Combining techniques for automated termination proofs. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 301–331. Springer, 2005.

[Giesl *et al.*, 2005b] Jürgen Giesl, René Thiemann, and Peter Schneider-Kamp. Proving and disproving termination of higher-order functions. In *International Workshop on Frontiers of Combining Systems*, pages 216–231. Springer, 2005.

[Hirokawa, 2006] Nao Hirokawa. *Automated termination analysis for term rewriting*. Citeseer, 2006.

[Huet, 1980] Gérard Huet. Confluent reductions: Abstract properties and applications to term rewriting systems: Abstract properties and applications to term rewriting systems. *Journal of the ACM (JACM)*, 27(4):797–821, 1980.

[James and Fabian, 2023] Fox James and Mitterwallner Fabian, 2023.

[Klein and Hirokawa, 2012] Dominik Klein and Nao Hirokawa. Confluence of non-left-linear trss via relative termination. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 258–273. Springer, 2012.

[Knuth and Bendix, 1983] Donald E Knuth and Peter B Bendix. Simple word problems in universal algebras. *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970*, pages 342–376, 1983.

[Koprowski and Waldmann, 2008] Adam Koprowski and Johannes Waldmann. Arctic termination... below zero. In *International Conference on Rewriting Techniques and Applications*, pages 202–216. Springer, 2008.

[Korp and Middeldorp, 2009] Martin Korp and Aart Middeldorp. Match-bounds revisited. *Information and Computation*, 207(11):1259–1283, 2009.

[Korp *et al.*, 2009] Martin Korp, Christian Sternagel, Harald Zankl, and Aart Middeldorp. Tyrolean termination tool 2. In *Rewriting Techniques and Applications: 20th International Conference, RTA 2009 Brasília, Brazil, June 29-July 1, 2009 Proceedings 20*, pages 295–304. Springer, 2009.

[Moser *et al.*, 2008] Georg Moser, Andreas Schnabl, and Johannes Waldmann. Complexity analysis of term rewriting based on matrix and context dependent interpretations. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (2008)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2008.

[Nagele and Middeldorp, 2016] Julian Nagele and Aart Middeldorp. Certification of classical confluence results for left-linear term rewrite systems. In *International Conference on Interactive Theorem Proving*, pages 290–306. Springer, 2016.

[Nagele *et al.*, 2015] Julian Nagele, Bertram Felgenhauer, and Aart Middeldorp. Improving automatic confluence analysis of rewrite systems by redundant rules. In *26th International Conference on Rewriting Techniques and Applications (RTA 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[Nagele *et al.*, 2017] Julian Nagele, Bertram Felgenhauer, and Aart Middeldorp. Csi: New evidence–a progress report. In *International Conference on Automated Deduction*, pages 385–397. Springer, 2017.

[Neurauter *et al.*, 2010] Friedrich Neurauter, Aart Middeldorp, and Harald Zankl. Monotonicity criteria for polynomial interpretations over the naturals. In *Automated Reasoning: 5th International Joint Conference, IJCAR 2010, Edinburgh, UK, July 16-19, 2010. Proceedings 5*, pages 502–517. Springer, 2010.

[Neurauter, 2012] Friedrich Neurauter. *Termination analysis of term rewriting by polynomial interpretations and matrix interpretations*. na, 2012.

[Oyamaguchi and Hirokawa, 2014] Michio Oyamaguchi and Nao Hirokawa. Confluence and critical-pair-closing systems. *Proc. 3rd IWC*, pages 29–33, 2014.

[Sakai *et al.*, 2015] Masahiko Sakai, Michio Oyamaguchi, and Mizuhito Ogawa. Non-e-overlapping, weakly shallow, and non-collapsing trss are confluent. In *Automated Deduction-CADE-25: 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings 25*, pages 111–126. Springer, 2015.

[Sternagel and Thiemann, 2013] Christian Sternagel and René Thiemann. Formalizing knuth-bendix orders and knuth-bendix completion. In *24th International Conference on Rewriting Techniques and Applications (RTA 2013)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2013.

[Sternagel and Thiemann, 2014] Christian Sternagel and René Thiemann. Formalizing monotone algebras for certification of termination and complexity proofs. In *International Conference on Rewriting Techniques and Applications*, pages 441–455. Springer, 2014.

[Sternagel, 2016] Christian Sternagel. The generalized subterm criterion in ttt2. *arXiv preprint arXiv:1609.03432*, 2016.

[Suzuki *et al.*, 2011] Sho Suzuki, Keiichirou Kusakari, and Frédéric Blanqui. Argument filterings and usable rules in higher-order rewrite systems. *IPSJ Online Transactions*, 4:114–125, 2011.

[Toyama and Oyamaguchi, 1994] Yoshihito Toyama and Michio Oyamaguchi. Church-rosser property and unique normal form property of non-duplicating term rewriting systems. In *International Workshop on Conditional Term Rewriting Systems*, pages 316–331. Springer, 1994.

[Van Oostrom, 1997] Vincent Van Oostrom. Developing developments. *Theoretical Computer Science*, 175(1):159–181, 1997.

[Yamada *et al.*, 2016] Akihisa Yamada, Sarah Winkler, Nao Hirokawa, and Aart Middeldorp. Ac-kbo revisited. *Theory and Practice of Logic Programming*, 16(2):163–188, 2016.

[Zankl and Middeldorp, 2010] Harald Zankl and Aart Middeldorp. Satisfiability of non-linear (ir) rational arithmetic. In *Logic for Programming, Artificial Intelligence, and Reasoning: 16th International Conference, LPAR-16, Dakar, Senegal, April 25–May 1, 2010, Revised Selected Papers 16*, pages 481–500. Springer, 2010.

[Zankl *et al.*, 2009] Harald Zankl, Nao Hirokawa, and Aart Middeldorp. Kbo orientability. *Journal of Automated Reasoning*, 43(2):173–201, 2009.

[Zankl *et al.*, 2011] Harald Zankl, Bertram Felgenhauer, and Aart Middeldorp. Csi–a confluence tool. In *Automated Deduction–CADE-23: 23rd International Conference on Automated Deduction, Wrocław, Poland, July 31-August 5, 2011. Proceedings 23*, pages 499–505. Springer, 2011.