DISTRIBUTED AUTOMATED DEDUCTION

A DISSERTATION PRESENTED

BY

MARIA PAOLA BONACINA

TO

THE GRADUATE SCHOOL

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

IN

COMPUTER SCIENCE

STATE UNIVERSITY OF NEW YORK

AT STONY BROOK

December 1992

Abstract of the Dissertation

Distributed Automated Deduction

by

Maria Paola Bonacina

Doctor of Philosophy
in
Computer Science
State University of New York at Stony Brook
1992

This thesis comprises four main contributions: an abstract framework for theorem proving, a methodology for parallel theorem proving in a distributed environment, termed *deduction by Clause-Diffusion*, a study of special topics in distributed deduction with contraction and an implementation, the theorem prover *Aquarius*, of the Clause-Diffusion methodology.

In our abstract framework, Knuth-Bendix type completion procedures are treated as semidecision procedures for theorem proving, rather than procedures to generate confluent systems. A theorem proving derivation is conceived as a process of reducing a proof of the *target* or goal. Accordingly, all the basic notions, such as *contraction* inference rules, *redundancy* of clauses, *refutational completeness* of the inference mechanism and *fairness* of the search plan, are defined in a *target-oriented* fashion. The most important feature of our framework is that it provides the first notion of *fairness for theorem proving*, which is weaker than the fairness property required to generate confluent systems. A weaker fairness requirement means that fewer inference steps are necessary. Therefore, a weaker fairness requirement is a theoretical pre-condition to the design of more efficient theorem proving strategies.

The Clause-Diffusion methodology exploits *parallelism at the search level*, by

having *concurrent, asynchronous* deductive processes searching in parallel the search space of the problem. The search space is partitioned among the processes by distributing the clauses and by subdividing certain classes of inferences. The processes communicate by exchanging data and they halt successfully as soon as one of them finds a proof. Policies for distributing the clauses and for scheduling inference and communication steps complete the picture. While the Clause-Diffusion methodology applies to theorem proving in general, it has been designed to provide solutions to the problems in the parallelization of *contraction-based* strategies, since they have been among the most successful. We describe *backward contraction*, i.e. the task of maintaining clauses reduced in a dynamically changing data base, as the main obstacle in parallel theorem proving with contraction. If a finer granularity of parallelism is adopted, e.g. parallelism at the clause level in shared memory, this difficulty appears as a write-bottleneck, which we have called the *backward contraction bottleneck*. The Clause-Diffusion approach avoids this problem by adopting a mostly distributed memory and *distributed global contraction schemes*.

We have given a definition of *distributed derivations* and characterized their fairness requirements, including fairness of inferences and fairness of communication. We have shown that the Clause-Diffusion methods are fair. Also, we have observed that the uncontrolled application of *subsumption* in a distributed data base may violate the *fairness* and the *monotonicity* - a dual property of soundness - of distributed derivations. This discovery has propelled a re-examination of the subsumption inference rule, which we view as a *replacement* rule, rather than a *deletion* rule. Our new *distributed subsumption* inference rule preserves both fairness and monotonicity, without reducing the contraction power of subsumption.

We conclude with the description of the distributed theorem prover *Aquarius*, including some experiments, and with directions for future research, such as the design of *parallel search plans*.

Contents

Acknowledgements

I sincerely thank Jieh Hsiang, my thesis advisor, for his guidance, for being a constant source of encouragement and optimism and for many exciting scientific discussions.

I also thank David Warren, Leo Bachmair, Gene Stark and Bill McCune for their comments and suggestions. The exchanges of ideas with Leo Bachmair and David Warren have been always extremely valuable. Bill McCune has been very helpful in answering promptly all my questions on Otter.

Claude Kirchner provided suggestions which led to the notion of localized image sets.

Steve Tuecke has been kind enough to reply to most of my questions on PCN by electronic mail within few minutes.

The correspondence with John Hawley and Kathy Yelick was also useful.

Kathy Germana and Betty Knittweis had an effective reply and a friendly smile whenever I asked for help. A thankful thought goes to the memory of Pegi Thomas. Several friends contributed to make these three years at Stony Brook a great experience.

I would like to thank my parents and my brother for their love and kinship.

Chapter 1

Introduction

The topic of this dissertation is distributed automated deduction. First, we give a new theoretical foundation for completion-based theorem proving, which pertains primarily to equational and Horn logic. Then, we study distributed automated deduction and we design a methodology, called the *Clause-Diffusion methodology*, for executing theorem proving strategies in a distributed environment. We prove that this methodology has all the desirable properties, e.g. it yields complete distributed strategies, and we describe its implementation in the distributed theorem prover *Aquarius* for first order logic with equality.

In this chapter, we introduce theorem proving and we give motivation and overview of both the theoretical foundation and the distributed methodology. We conclude the chapter with an outline of the contents of the thesis.

1.1 Theorem proving

A *theorem proving problem* consists in deciding, given a set of clauses S and a clause φ, whether φ is a theorem of S, written $\varphi \in Th(S)$. We call the set S a *presentation*

of the theory $Th(S)$. The theorem φ to be proved is called the *target* or *goal*. A theorem proving strategy $\mathcal{C}$ is specified by a set of *inference rules* I and a *search plan* Σ. The inference rules derive new clauses from existing ones and the search plan Σ chooses the inference rule and the premises for the next step. By iterating the application of I and Σ, a *derivation*

$$S_0 \vdash_{\mathcal{C}} S_1 \vdash_{\mathcal{C}} \ldots S_i \vdash_{\mathcal{C}} \ldots,$$

is constructed. A derivation is *successful* if it reaches a solution. A theorem proving strategy $\mathcal{C}$ is *complete*, if, whenever the input target is indeed a theorem, the derivation constructed by $\mathcal{C}$ halts successfully. Completeness involves both the inference rules I and the search plan Σ. First, it requires that if the input target is a theorem, there exist successful derivations by I (*refutational completeness of the inference mechanism*). Second, it requires that whenever successful derivations exist, the search plan Σ selects a successful derivation among the possible derivations by I from the given input (*fairness of the search plan*).

Refutational completeness and fairness are the fundamental requirements on the inference mechanism and the search plan of a theorem proving strategy. If a method has a refutationally complete inference mechanism and a fair search plan, we know that for every true input target, we can find a proof. The challenging question is to obtain a strategy which is **both complete and efficient**: not only should the strategy succeed, but it should also do it by consuming "reasonable" amounts of resources, i.e. time and space. The effects of improving the efficiency of theorem proving procedures are to extend the class of provable theorems and to speed-up already available proofs. The goal is then to increase the efficiency of the strategy while preserving the completeness.

Clearly, both the inference rules and the search plan affect the efficiency of a strategy. At the inference level, we classify inference rules into *expansion inference rules* and *contraction inference rules*. Expansion rules derive new clauses from

existing ones, by using *unification*, and add them to the data base. Resolution, hyperresolution and paramodulation [28, 91] are examples of expansion inference rules. Contraction rules delete existing clauses or replace them by logically equivalent but smaller ones. The deleted/replaced clauses are *redundant*. Examples of contraction rules are (proper) subsumption [91], simplification [109], tautology elimination [28], conditional simplification and normalization, i.e. repeated simplification until possible. Contraction rules usually employ *matching*, not unification.

The search aspect of theorem proving may be visualized as a game, where a player is searching for a room of the treasure in a labyrinth of rooms. At each stage of the game the player finds herself in a room with a number of doors. She has to choose which door to open and proceed to another room. She wins if she reaches a room containing the treasure. The player has no idea of the whole labyrinth, so that her choice is made locally, step by step. The labyrinth represents the search space and the room where the player is represents the current data base. The doors symbolize the inference steps: opening a door and proceeding to another room is like performing an inference step. Contraction and expansion inferences are like doors with locks and without locks respectively. For a contraction step, the player opens the door, gets in the next room and the door is locked behind her, so that it is not possible to go back to the previous room. A contraction step deletes a clause and disallows the steps which use that clause. There are inferences which are possible before a contraction step commits, but they are no longer possible afterwards. On the other hand, an expansion step does not prevent any other step and thus the door is not locked.

Most search plans work by applying *heuristic criteria* to the available data, in order to determine the next step. For instance, a very simple heuristic consists in selecting as premises the smallest clauses in the existing set, on the ground that they may be closer to the empty clause, whose generation denotes a refutation. More generally, a search criterion provides a measure of the distance between the current set of clauses and a solution. Then the choice is made in order to minimize this

distance. In the above example the measure is the size of the smallest clauses in the data base. If the selected inference rule generates clauses which are smaller than the premises, picking the smallest clauses ensures that even smaller clauses will be generated, thereby getting closer to a solution. Any such measure is just a tentative estimate of the actual distance. If the actual distance were known, the problem would be solved. Indeed, search criteria are heuristic in nature.

Conceptually, a meaningful measure of the efficiency of a method is the portion of the search space which the prover needs to generate before hitting a solution. The larger such a space is, the more memory is consumed and the longer is the time spent in the process. The size of the generated search space depends on both the inference rules and the search plan. At the inference level, expansion rules and contraction rules have opposite effects on the generated search space. Expansion rules expand it, whereas contraction rules reduce it. An expansion-oriented strategy generates in most cases a much larger portion of the search space than a strategy which applies the contraction rules eagerly. Therefore, the former requires in general more memory than the latter. The reason is that the cost of adding a new clause does not consist solely in the amount of memory needed to store the clause itself. Adding a new clause increases the number of clauses which can be generated in the following steps. Each new clause makes the entire portion of search space which depends on that clause available. The effect of contraction is precisely the opposite: deleting a clause cuts the portion of search space which depends on that clause, i.e. all its descending inferences. Thus, in order to keep the size of the generated search space manageable, it is very important to have powerful contraction rules, strong restrictions to the application of expansion rules and a search plan which gives priority to contraction steps. These are the distinguishing features of **contraction-based strategies**. These strategies have been implemented in a number of theorem provers and have obtained impressive results in proving new, challenging theorems (e.g. [4, 6, 78, 98]).

1.2 Completion-based theorem proving

The first part of the thesis focuses on completion-based strategies, i.e. the theorem proving strategies originated from the completion procedure of Knuth and Bendix [84].

1.2.1 Why we need a theoretical framework for completion-based theorem proving

The Knuth-Bendix completion procedure computes a possibly infinite confluent rewrite system equivalent to a given set of equations [66, 84]. If a set of equations E and an equation $s \simeq t$ are given, it semidecides whether $s \simeq t$ is a theorem of E, as first remarked in [67, 88]. These results hold if the procedure does not fail on an unoriented equation. Unoriented equations can be handled by adopting the Unfailing Knuth-Bendix method [13, 62], which gives a ground confluent set of equations. Many completion procedures, related to the Knuth-Bendix procedure to different extents, have been designed. They include procedures for equational theories with special sets of axioms [12, 70, 105], Horn logic with equality [44, 45, 86], first order logic [8, 58, 59, 75], first order logic with equality [15, 16, 60, 61, 63, 64, 109, 130], inductive theorem proving in equational and Horn theories [48, 68, 71, 86] and logic programming [21, 36, 37, 40]. Surveys have been given in [39, 41].

Our motivation in seeking a new theoretical approach to the study of these procedures is to bridge the gap between the empirical evidence of the successes of completion procedures as theorem proving strategies, and the state of the theory of completion procedures. Completion procedures have been conceived primarily as procedures for generating confluent systems of rewrite rules, which are by themselves decision procedures for the theories presented by the input sets of equations. This view imposes serious and unnecessary limitations to the applicability of completion procedures, since few theories are decidable, which means that few theories

have finite confluent systems. We propose a different perspective, where completion procedures are treated as theorem proving methods, that is, as *semidecision procedures*. From an operational point of view, they are used for proving individual target theorems rather than generating decision procedures.

Our approach is inspired by practical experiences in using completion procedures as theorem proving methods, such as those reported in [3, 4, 5, 6]. The experiments show that the forward chaining nature of completion (generating critical pairs from given equations) is neither natural nor efficient enough for many theorem proving purposes. In order to prove the most difficult theorems, one may need to incorporate goal-oriented critical pair criteria and search plans. The function of these features is to use the target theorem to direct the generation of critical pairs, so that the search space can be significantly reduced by not generating those critical pairs which may not contribute to prove the intended theorem. These target-oriented features do not fit naturally in the conventional framework of completion, because completion is traditionally interpreted as a procedure for generating confluent systems. Such a procedure must take into account all critical pairs which may lead to establish the equivalence of any two arbitrary terms in the theory. In a theorem proving application, one only wants to concentrate on the critical pairs which may contribute to a proof of the given target theorem. Thus, deductions (critical pairs) which are not related to the target theorem need not and should not be generated at all, even if they may contribute to prove the confluence of some other equations. Applying a procedure designed to generate confluent systems to demonstrate a specific theorem is like cutting a single tree in a forest by cutting the entire forest: the result is certainly achieved, but the waste of resources is enormous.

In order to capture the application of completion procedures as semidecision theorem proving procedures, we propose a new framework where target-oriented notions can also be formulated. Most of the concepts mentioned in our informal introduction to theorem proving, e.g. the duality of inference systems and search plans, refutational completeness and fairness, expansion versus contraction, are part

of this framework. By incorporating the target theorem explicitly in the derivation process, one can more effectively capture goal-oriented inference rules and search plans. Our intention is to provide a theoretical foundation over which other completion procedures, goal-oriented critical pair criteria, and efficient search strategies can be developed.

1.2.2 Completion procedures as semidecision procedures

The interpretation of completion as semidecision procedure is not new. It first appeared in [67], where it was proved that if the procedure is fair, the limit of an unfailed Knuth-Bendix derivation is a confluent rewrite system. As a side-effect, if a theorem $s \simeq t$ is given to the procedure, it semidecides the validity of $s \simeq t$. The same result was obtained in a more general framework in [7]. This view of completion is not acceptable from the theorem proving perspective, because its goal is still the eventual generation of a confluent system. Since our motivation is to make possible the design of efficient completion-based theorem proving strategies, we reverse the traditional approach to completion procedures: we regard them as *semidecision procedures* with the generation of confluent systems as a potential side-effect.

The key idea in our approach is to consider a theorem proving derivation as a process of **target-oriented proof reduction**. Given a *target theorem* φ and a *presentation* of a theory, i.e. a set of axioms S, the process of proving φ from S can be characterized as a reduction, with respect to a *well founded ordering*, of a proof of φ in S. Success is reached when the proof is *empty*. The intuition that proving a given theorem requires in most cases less work than generating a confluent system can now be formulated in terms of proof reduction: reducing one proof is conceivably a smaller task than reducing all the proofs. Therefore, a target-oriented completion procedure, i.e. a strategy which focuses on reducing the proof of the given target, should be more efficient as a theorem prover than a procedure which

works blindly to reduce all the proofs.

We investigate how a procedure can be made target-oriented both at the level of the *inference rules* and at the level of the *search plan* of a completion procedure. For the inference mechanism, we concentrate on contraction. We characterize the contraction steps in terms of target-oriented proof reduction and target-oriented *redundancy*. Coherently, *refutational completeness* is also re-defined in terms of target-oriented proof reduction. For the search plan, we give a new notion of *fairness*, which is radically different from the previous ones. In [67] and in all the following work on completion [7, 14, 16, 17, 109], fairness of a derivation consists in eventually considering all critical pairs. We call this property *uniform fairness* in order to distinguish it from fairness for theorem proving. Uniform fairness is necessary for the limit of a derivation to be confluent, but it is not necessary for theorem proving, because not all the critical pairs are necessary to prove a given theorem. In fact, the requirement of uniform fairness clashes with the goal of having an efficient search plan for theorem proving. For instance, we may have a problem where the target $s \simeq t$ is an equation on a signature F_1 and the input presentation E is the union of a set E_1 of equations on the signature F_1 and a set E_2 of equations on another signature F_2, disjoint from F_1. Such a problem can occur in definitions of abstract data types, if the signature F_1 contains the constructors and a set of defined symbols, whereas the signature F_2 is another set of defined symbols. Intuitively, a derivation where no inference from E_2 is performed is fair. On the other hand, uniform fairness requires to compute critical pairs from the equations in E_2 as well.

Clearly, theorem proving requires a definition of fairness which is weaker than uniform fairness. We provide such a new definition of fairness by using target-oriented proof reduction as for refutational completeness. Fairness means that all the inference steps which are necessary to prove the goal are eventually done. In particular, all the critical pairs which are necessary to prove the goal are eventually considered. We prove that if the inference rules are complete and the search plan is fair according to our definitions, the procedure is a *semidecision procedure*. By

showing that fairness is sufficient for theorem proving, we prove the classical result in [67] from weaker, strictly theorem proving oriented hypotheses. No confluence property of the limit of the derivation is implied, showing that such properties are not necessary for theorem proving.

Our framework also covers the interpretation of completion procedures as generators of decision procedures. If the search plan is uniformly fair, the limit of the derivation is a *saturated* presentation of the given theory. For instance, in equational logic a confluent system is a saturated set. If a presentation is saturated, all expansion inference steps are redundant: a confluent system has no non-trivial critical pairs. Thus, derivations in a saturated set are *linear*, i.e. made only of inferences on the target. If such linear derivations are guaranteed to halt, the saturated set represents a *decision procedure* for the validity of theorems. Since few theories have finite saturated presentation, the application of completion procedures to generate saturated sets has minor importance from a practical point of view.

1.3 Distributed automated deduction

Once we had obtained this theoretical foundation, we posed the question of how to improve the power of contraction-based strategies by **parallel** computation. This thesis reports the design and implementation of a new methodology for parallel theorem proving, called *distributed deduction by* **Clause-Diffusion**.

1.3.1 Why parallel contraction-based deduction is challenging

Contraction-based strategies are widely recognized to be one of the most promising approaches in automated deduction. There are both experimental evidence (e.g. [3, 4, 5, 6, 78, 98]) and theoretical understanding (e.g. our framework) that they are generally more effective than expansion-oriented methods. On the other hand,

exactly those features, that make sequential contraction-based procedures successful, also make their parallelization difficult.

Parallelism may be introduced in a deduction strategy in several ways. One may parallelize the inner components of an inference, e.g. by using parallel matching. Or, one may choose to have certain groups of inference steps executed in parallel. Most of the existing works in parallel deduction adopt approaches of this type. In other words, they try to parallelize the inference mechanism. These mechanisms are cost-effective under specific assumptions. One such assumption is that the data base of clauses is **static** during the derivation. For instance, if a data base of equations is static, it is possible to *pre-process* all the equations at "compile-time", i.e. before the derivation, for the purpose of fast parallel matching. The equations are compiled in a form convenient for the parallel matching algorithm and the overhead of parallelism during the derivation is greatly reduced. Intuitively, most of the overhead, i.e. extra work, which may be introduced by parallelism, is dealt with once and for all at compile-time. Another condition which is favorable to parallelization is that there are **no conflicts** between inference steps. If there are no conflicts, the inference steps can be executed correctly in parallel.

The majority of the known techniques for parallel deduction (e.g. [23, 31, 43, 51, 82, 110, 111, 121]) applies to classes of strategies, which satisfy, at least to some extent, requirements such as *static data base* and *no conflicts*. Two such classes are the *subgoal-reduction strategies* and the *expansion-oriented strategies*. The evaluation strategies in functional/logic programming and in Prolog technology based theorem proving are subgoal-reduction strategies. Each inference step consists in reducing the current goal to one or more, hopefully simpler, subgoals. Thus, the data base is basically static during a derivation, because only the goal is being modified. In expansion-oriented strategies, the given clauses are used as premises of expansion steps to generate new elements. Assuming that concurrent-read can be safely admitted, any two expansion inference steps are not in conflict, because they simply need to read their premises, but not to re-write them.

Contraction-based strategies are much more difficult to parallelize, because they do not satisfy hypotheses such as *static data base* and *no conflicts*. The reason is the extensive application of contraction and especially *backward contraction*. We term **forward contraction** the contraction of a newly generated clause with respect to those existing when it is generated. **Backward contraction** is the contraction of a clause with respect to those generated afterwards. Forward and backward contraction are both necessary to maintain the data base of clauses to the minimal, which is a cornerstone of the contraction-first philosophy.

For the purpose of parallelization, forward and backward contraction have a very different impact. Forward contraction of a clause can be done once and for all when the clause is derived. In actuality, forward contraction can still be regarded as part of the generation process of a clause. If a clause is deleted during the forward contraction process, it is not even inserted in the data base. Backward contraction of a clause involves *the normalization of the clause with respect to all the clauses which may be generated afterwards*. Thus, all the clauses in the data base are tested for reducibility and subject to contraction repeatedly throughout the computation. This entails several consequences. First, the data base is *highly dynamic* during the derivation, in contrast with the requirements of the parallelization techniques we mentioned above. Second, it is not possible to separate a set of simplifiers and a set of expressions to be reduced: every clause may play both the role of simplifier and the role of expression to be reduced. It follows that *very little pre-processing* can be done. Also, it means that there is *no read-only data*, i.e. all the items in the data base need be accessible not only for reading but also for writing.

Furthermore, contraction inferences require both *read-access* and *write-access* to premises, while expansion inferences simply need read-access. Thus, concurrent expansion and contraction steps may be in **conflict** for the access to common premises. In contraction-based strategies, new clauses are not used for expansion steps before undergoing forward contraction. Therefore, forward contraction does not enter in

conflict with expansion. But backward contraction does, because backward contraction steps and expansion steps are intertwined, since the clauses used as parents of new clauses are also subject to backward contraction.

Finally, a clause which is reduced by a backward contraction step, should be tested for further contraction with respect to all the other clauses. Thus, a single backward-contraction step may induce many. In shared memory implementations (e.g. [94, 128]), this avalanche growth of contraction steps causes a write-bottleneck, which we term the **backward contraction bottleneck**, since all the backward contraction processes ask write-access to the shared memory. Not all of them may be served and an otherwise unnecessary sequentialization is imposed. The clauses which are supposed to be subject to backward contraction may not be made available for other tasks, e.g. expansion steps, so that these are delayed as well.

Because of these issues, most existing approaches do not apply to contraction-based strategies. On the other hand, since contraction-based strategies satisfy less constraints than other strategies, a methodology designed having contraction-based strategies in mind can also be applied to strategies without contraction.

1.3.2 The Clause-Diffusion methodology

The Clause-Diffusion methodology parallelizes a deduction strategy by realizing **parallelism at the search level**. This is a fairly new approach, since most of the existing methods introduce parallelism in the inference mechanism. Among the works we survey, only three, [32], [33] and [55], adopt some parallelism at the search level.

The motivation for choosing parallelism at the search level is twofold. The first motivation comes from the analysis of the difficulties with backward contraction, which we enumerated in the previous section, and from our survey of related works.

The latter mostly implement finer grain parallelism at the inference level. The potentially very high number of clauses generated during a theorem proving derivation, the dynamic character of the data base and the high degree of inter-dependence of inference steps, e.g. the conflicts, lead us to resolve that such fine grain parallelism is probably not cost-effective for contraction-based theorem proving. Therefore, we consider parallelism at the search level as a form of *coarse grain parallelism in deduction*. This technical motivation is strengthened by the following intuition. The purpose of designing a parallel theorem proving method is to improve over existing sequential strategies. Thus we should start by considering where the problem with existing sequential strategies lies. A complete strategy fails to find a proof of the given theorem if it exhausts the available memory, by generating too large a portion of the search space in the attempt to reach a solution. Then, we reason that if we can have several deductive processes searching concurrently the same space, we might have better chances to find a solution before the generated search space becomes unmanageable. In other words, we consider parallelism at the search level, because the problem is at the search level.

The Clause-Diffusion methodology partitions the search space among *concurrent asynchronous processes*, which search in parallel for the solution. As soon as one of them succeeds, the whole process succeeds and terminates. In order to subdivide the search task, we consider that the search space is determined by the given clauses and the inference rules. One may partition the set of clauses or the set of rules or a combination of the two. The first approach amounts to *data-driven parallelism*: each concurrent process is given a subset of the data and it is in charge of all the operations on its data. The second one is *operations-driven parallelism*: each concurrent process is given a subset of the operations, e.g. the inference rules, and it is responsible for applying them to all the data. Since in theorem proving applications, there are many more data items than inference rules, data-driven parallelism is more natural. The input clauses and all those generated afterwards are distributed among the processes. We call the clauses allotted to a process its *residents*. Based on the ownership of clauses, the expansion inferences are also subdivided among the

processes. Contraction inferences are not subdivided, so that each process can perform as much contraction as possible.

Since each process is assigned only a segment of the data, the processes need to communicate, so that a proof involving data at different processes can be found. The processes send their clauses to the other processes in form of *inference messages*. In a distributed environment, e.g. a high-speed network of computers or a loosely coupled asynchronous multiprocessor, each of the theorem proving processes is executed at a different processor node, and inference messages are actually implemented as network messages. In a shared memory environment, the processes communicate through the shared memory, i.e. by writing and reading the inference messages in shared memory. In a mixed shared-distributed environment, e.g. with separate memories at the nodes and a shared memory component, a subset of the clauses may be shared, while the remaining communication is achieved through messages. We shall weigh the advantages and disadvantages of different memory configurations for the Clause-Diffusion methodology. Several factors, among them the coarse granularity of parallelism at the search level, will lead us to choose preferably a mixed shared-distributed environment or else a purely distributed one.

By specifying the different components involved, including the inference mechanism I, the search plan Σ to schedule inference steps and communication steps, the allocation algorithm, the algorithms for routing and broadcasting of messages, one obtains a specific Clause-Diffusion strategy.

Distributed global contraction

The distribution of the work among the processes does not cover yet all the issues related to contraction.

First, there are contraction inferences which require access to the *global data base*, i.e. the union of the sets of residents of the deductive processes. In order to

implement a contraction-first strategy, normalization of a clause means normalization with respect to *all* the equations, i.e. with respect to the global data base. We call such inferences *global contraction* inferences. Second, a contraction-first plan recommends that a process does not perform an expansion inference if the premises have not been reduced as much as possible with respect to the global data base. The problem here is to find a good trade-off between concurrency (e.g. concurrency of expansion) and containment of redundancy (e.g. by backward contraction). In order to enhance concurrency, we would like to let the processes proceed eagerly with their own inferences, disregarding whether their clauses are reduced with respect to those of the other processes. In order to minimize redundancy, we would like to enforce strictly the contraction-first priority rule.

Finally, the amount of global contraction is related to backward contraction. In fact, it is backward contraction which causes most of the need for global contraction. In a contraction-based strategy, a clause which is reduced by a backward contraction step, needs to be tested in principle for further contraction, with respect to *all* the other clauses. Thus, this is a global contraction task. We propose several **global contraction schemes** to perform global contraction with respect to a distributed data base. We give schemes for a mixed shared-distributed environment and for a purely distributed one. We show that our schemes do not suffer from the backward contraction bottleneck, mentioned in the previous section.

Fairness in distributed derivations

We give a formal definition of the *distributed derivations* generated by Clause-Diffusion strategies and we study the fairness requirements for such derivations. We extend the definition of uniform fairness of a sequential derivation to a distributed derivation. Since two persistent clauses may belong to two different deductive processes, **distributed uniform fairness** poses an additional requirement of fairness of the communication part of a distributed strategy. We present a set of conditions,

which comprises a specification of fairness of communication, and prove that they are *sufficient* for the uniform fairness of a distributed derivation. We then describe several techniques, related to different choices of global contraction scheme, for implementing these sufficient conditions for uniform fairness. Since these techniques may induce redundant messages, we design specific contraction rules to delete them.

Distributed subsumption

The unrestricted application of subsumption, and especially *backward subsumption of variants*, is well-known to destroy the completeness of a strategy [91]. This problem becomes much more serious in a distributed derivation and the solutions known for the sequential case cannot be extended straightforwardly to the distributed one. First, it may happen that a combination of concurrent steps of subsumption of variants deletes all the variants of a clause. This makes the distributed derivation *non-monotonic*, as theorems of the original theory may be lost. Thus, in addition to the possibility of losing completeness, subsumption of variants causes the even more fundamental problem of losing monotonicity. Second, not only subsumption of variants, but also a combination of concurrent steps of proper subsumption of residents by inference messages may violate monotonicity.

We show that the source of the problem lies in a misconception of the subsumption inference rule. We reason that proper subsumption is not deletion of a clause, but rather *replacement* of a clause by a more general one. This difference cannot be easily appreciated in the sequential case, where the clauses are stored in the same data base. But it is significant in a distributed environment, where inferences may involve clauses belonging to remote data bases. Based on this understanding of subsumption, we refine the subsumption inference rule into two rules: *replacement subsumption* and *variant subsumption*, with proper subsumption as a derived inference rule. In the distributed case variations of the two inference rules are combined into a *distributed subsumption* inference rule, which has all the desirable properties

of subsumption. It allows proper subsumption between clauses stored in remote data bases without violating monotonicity, and subsumption of variants while preserving monotonicity and completeness.

1.4 Outline of the thesis

The thesis is organized as follows. In Chapter 2, we present our theoretical framework: inference rules and search plans, expansion versus contraction, target-oriented proof reduction and redundancy, refutational completeness and fairness. We show that completion procedures which satisfy our definitions of refutational completeness and fairness are **semidecision procedures**. Next, we consider uniformly fair derivations and the generation of decision procedures. In the second part of the chapter we apply our framework to existing completion procedures for equational logic. We show that the basic *Unfailing Knuth-Bendix procedure* [13, 62] and some of its extensions, such as the *AC-UKB procedure* [2, 12, 70, 105] with *Cancellation laws* [63], the *S-strategy* [62] and the *Inequality Ordered Saturation strategy* [4] fit nicely in our framework. To our knowledge, this is the first presentation of these extensions of the UKB procedure as sets of inference rules. Also the so called *inductionless induction* method is covered by the semidecision concept: completion for inductionless induction [68] is a *semidecision procedure for disproving inductive theorems.*

In Chapter 3 we study the problems in parallel theorem proving, with special attention to the specific problems posed by the parallelization of contraction-based strategies. For this purpose, we classify theorem proving strategies as *subgoal-reduction strategies*, *expansion-oriented strategies* and *contraction-based strategies.* We define *parallelism at the term level*, *parallelism at the clause level* and *parallelism at the search level*, which represent respectively fine grain parallelism, medium grain parallelism and coarse grain parallelism for theorem proving applications. We relate

the granularity of parallelism to the choice of memory configuration, i.e. shared versus distributed memory, and to the problem of *conflicts* between parallel inference steps. After having set this classification of strategies, types of parallelism and conflicts, we apply it to analyze current approaches in parallel rewriting, e.g. [43, 82, 51], parallel Prolog technology theorem proving, e.g. [23, 31, 110], parallel implementations of the Buchberger algorithm [55, 111, 121], parallel expansion-oriented theorem proving, e.g. [32, 69], parallel Knuth-Bendix completion, e.g. [33, 128], and the parallel theorem prover ROO [94]. In particular, we observe the *backward contraction bottleneck* in contraction-based strategies with parallelism at the clause level. The main result of this analysis is that parallelism at the **search level** is the most natural for contraction-based strategies. A hybrid environment, with predominantly distributed memory and a shared memory component, turns out to be the most appropriate to realize parallelism at the search level.

In Chapter 4 we present in full our **Clause-Diffusion methodology** and its variations. We propose and discuss several schemes for distributed global contraction, techniques for the generation and maintainance of a distributed or shared data base of simplifiers, policies for the distributed allocation of residents and algorithms for routing and broadcasting of different types of messages. We describe the inferences at a node and some guidelines for the scheduling of the operations of a node. In Chapter 5 we define distributed derivations and we study distributed fairness, distributed subsumption and inference rules for eliminating redundant messages and updating distributed or shared data bases of simplifiers. We show that if a sequential search plan Σ is fair, the Clause-Diffusion strategies embedding Σ are fair. It follows that if the inference mechanism is refutationally complete, the Clause-Diffusion strategies are complete.

Chapter 6 describes our prototype **Aquarius** and the experiments conducted so far. Aquarius has been developed on a network of Sun workstations, i.e. a purely distributed environment, with no shared memory and message-passing. Aquarius implements the Clause-Diffusion strategies based on the sequential strategies offered

in the theorem prover *Otter* [98, 99]. Therefore, Aquarius embeds Otter and it inherits most of its valuable features. First, it exploits the high efficiency of basic operations and data structures, for which Otter is well-known. Second, it is highly portable, since it is written in C and PCN [27, 47]. The latter is an extension of C, that allows concurrency and communication between concurrent processes through streams. A PCN application can be developed on a single workstation and then be ported to several different parallel architectures. Third, Aquarius maintains the philosophy of Otter of providing the user with a wealth of options to experiment with. In addition to all those of Otter, new parameters related to distributed execution are added. This gives the user the possibility of tailoring the prover to different classes of problems.

In Chapter 7 we summarize our results and we propose directions to continue this work. They include the fine-tuning of Aquarius, the realization of the Clause-Diffusion methodology in a mixed environment, i.e. with both distributed and shared memory, and the design of *parallel search plans*.

Chapter 2

Completion-based deduction

In this chapter we present our abstract framework for contraction-based theorem proving. In Section 2.2, we define proof orderings for theorem proving. In Section 2.3, we introduce inference rules and search plans, emphasizing the distinction between *expansion* and *contraction* inference rules. Section 2.4 is devoted to target-oriented *proof reduction* and *redundancy*. The definition of *completion procedure* summarizes the concepts introduced so far. Section 2.5 contains the notions of *refutational completeness* and *fairness*, and the theorem showing that completion procedures with these properties are *semidecision procedures*. In Section 2.6, we discuss different definitions of redundancy and contraction, such as those in [109, 16, 17]. In Sections 2.7 and 2.8, we consider *uniformly fair* derivations, the generation of *decision procedures* for equational theories and some extensions of these results to Horn theories. In Section 2.9, we present, according to our framework, some completion procedures for equational logic: the basic *Unfailing Knuth-Bendix procedure* [62, 13], *AC-UKB procedure* [105, 70, 12, 2] with *Cancellation laws* [63], the *S-strategy* [62] and the *Inequality Ordered Saturation strategy* [4]. We close the chapter by showing how the so called *inductionless induction* method [68] is covered by the semidecision concept.

2.1 Basic definitions

In this section we recall some basic concepts and notations for theorem proving. Our notation is consistent with [41, 42].

Given a finite set F of constant symbols and function symbols with their arities and a denumerable set X of variable symbols, $T(F, X)$ is the set of *terms* on F and X. A term is *ground* if it does not contain variables. The set of ground terms is denoted by $T(F)$. A term s is a *subterm* of a term t if s occurs in t. We write t as $c[s]$ to indicate that s is a subterm of t in the *context* c. We assume the standard representation of terms as finite ordered trees: a variable x or a constant c is represented as a tree with only a single node labeled by x or c respectively. A term $f(t_1 \dots t_n)$ is represented by a tree whose root has label f and has n ordered outgoing edges labeled $1 \dots n$ pointing to the roots of the trees for the terms $t_1 \dots t_n$.

We write $s = t|u$ to specify that s is the subterm of t at *position* u, where u is the string of natural numbers labeling the edges in the path from the root of the tree for t to the root of the subtree for s. More precisely, the set $\mathcal{O}(t)$ of positions in a term $t = f(t_1 \dots t_n)$ is the set of strings of natural numbers such that $\lambda \in \mathcal{O}(t)$, where λ is the empty string, and if $u \in \mathcal{O}(t_i)$ for some i, $1 \leq i \leq n$, then $i \cdot u \in \mathcal{O}(t)$. The empty string λ denotes the root position, i.e. $t|\lambda = t$, and the string $i \cdot u$ denotes the position u in the i-th subterm of t, i.e. $f(t_1 \dots t_n)|i \cdot u = t_i|u$. For two positions u and v in $\mathcal{O}(t)$, u is a *prefix* of v if $v = uw$ for some other string w and u is a *proper prefix* of v if $v = uw$ and w is not empty. Two distinct positions u and v in $\mathcal{O}(t)$ are *disjoint*, written $u|v$, if neither u is a prefix of v nor v is a prefix of u. The notation $s[t]_u$ represents the term obtained by replacing $s|u$ by t.

A *substitution* σ is a set $\{x_1 \mapsto s_1 \dots x_n \mapsto s_n\}$ such that

- $\forall i, j$, $i \neq j$ implies $x_i \neq x_j$,
- $\forall i, j$, $x_i \notin V(s_j)$, where $V(s_j)$ is the set of variables occurring in the term s_j.

The set $\{x_1 \dots x_n\}$ is called the *domain* of the substitution σ: $Dom(\sigma) = \{x_1 \dots x_n\}$. The set of all the variables occurring in the terms $s_1 \dots s_n$ is called the *range* of σ: $Ran(\sigma) = \bigcup_{j=1}^{n} V(s_j)$. A substitution σ is *ground* if $Ran(\sigma) = \emptyset$. A substitution σ applies to a term as follows:

- $x\sigma = s$ if $x \mapsto s \in \sigma$,
- $x\sigma = x$ if $x \notin Dom(\sigma)$,
- $c\sigma = c$ if c is a constant,
- $f(t_1 \dots t_n)\sigma = f(t_1\sigma \dots t_n\sigma)$, otherwise.

Given two substitutions $\sigma = \{x_1 \mapsto s_1 \dots x_n \mapsto s_n\}$ and $\rho = \{y_1 \mapsto r_1 \dots y_m \mapsto r_m\}$ such that $Dom(\sigma) \cap Ran(\rho) = \emptyset$, their *composition* is the substitution $\sigma\rho = \{x_1 \mapsto s_1\rho \dots x_n \mapsto s_n\rho\} \cup \{y_j \mapsto r_j | y_j \mapsto r_j \in \rho, y_j \notin Dom(\sigma)\}$. Composition of substitutions is associative, i.e. $(\sigma\rho)\theta = \sigma(\rho\theta)$. The second condition in our definition of substitution implies that $Dom(\sigma) \cap Ran(\sigma) = \emptyset$. This property is equivalent to $\sigma\sigma = \sigma$, i.e. *idempotence* of composition.

An *ordering* $\succ$ is a binary relation which is transitive and irreflexive. Transitivity and irreflexivity imply asymmetry. An ordering is *partial* in general. It is *total* if for every two distinct elements s and t in the ordered set, either $s \succ t$ or $t \succ s$. An ordering is *well-founded* if there is no infinite chain $s_1 \succ s_2 \succ \dots s_n \succ \dots$.

The subterm relation is an ordering, called the *subterm ordering*: $t \trianglerighteq s$ if $t = c[s]$ and $t \triangleright s$ if $t \trianglerighteq s$ and $t \neq s$. A term t is an *instance* of a term s if there is a substitution σ such that $t = s\sigma$: we write $t \underline{\succeq} s$ and the relation $\underline{\succeq}$ is called *subsumption ordering*. The substitution σ is called a *matching substitution*. If t is an instance of s and s is an instance of t, the two terms are equal up to a permutation of variables. A substitution which is just a permutation of variables is said to be a *renaming* and the two terms s and t are said to be *variant* of each other, denoted by $s \stackrel{\bullet}{=} t$. Otherwise,

t is a *proper instance* of s: if $t \trianglerighteq s$ and $t \not\doteq s$, $t \triangleright s$. The relation $\triangleright$ is the *proper subsumption ordering*.

The *encompassment ordering* $\trianglerighteq$ is the composition of the subterm ordering and the subsumption ordering: $t \trianglerighteq s$ if $t = c[s\sigma]$ for some context c and substitution σ; $t \triangleright s$ if $t \trianglerighteq s$ and $s \not\doteq t$.

A substitution σ is an instance of a substitution θ if there is a third substitution ρ such that $\forall x \in Dom(\sigma), x\sigma = x\theta\rho$. The substitution σ is a proper instance of θ if ρ is not a renaming. Similar to terms, we write $\sigma \trianglerighteq \theta$ and $\sigma \triangleright \theta$.

Given two terms s and t, a substitution σ is a *unifier* of s and t if $s\sigma = t\sigma$. A substitution σ is a *most general unifier* (mgu) of s and t if σ is a unifier of s and t and for all unifiers ρ of s and t, $\rho \trianglerighteq \sigma$.

We define the following properties for an ordering $\succ$ on terms:

- *monotonicity:* $s \succ t$ implies $c[s] \succ c[t]$ for all contexts c,
- *stability*: $s \succ t$ implies $s\sigma \succ t\sigma$ for all substitutions σ and
- *subterm property*: $c[s] \succ s$ for all terms s and contexts c.

The first property says that if $s \succ t$, then any superterm $c[s]$ of s is greater than the corresponding superterm $c[t]$ of t. The second property says that substitutions preserve the order relation on terms. The third one says that a term is greater than any of its subterms, i.e. an ordering with such property includes the subterm ordering. A monotonic, stable and well-founded ordering is a *reduction ordering*. A monotonic and stable ordering with the subterm property is a *simplification ordering*. Monotonicity, stability and subterm property imply well-foundedness [35]. Therefore a simplification ordering is also a reduction ordering.

An *equation* is an unordered pair of terms $l \simeq r$. A *rewrite rule* is an ordered pair of terms $l \to r$. A set of rewrite rules is called a *term rewriting system* or

rewrite system. An equation $l \simeq r$ is oriented into a rewrite rule $l \to r$, if $l \succ r$ for a reduction ordering $\succ$. A rewrite rule $l \to r$ is *left linear* if no variable occurs in l more than once. If all rules in R are left linear, R itself is left linear.

A rewrite system R defines a relation $\to_R$ on terms as follows: $s \to_R t$ if there are a rewrite rule $l \to r \in R$, a substitution σ and a position u such that $s|u = l\sigma$ and t is $s[r\sigma]_u$. The relation $\leftrightarrow_R$ is defined as the union $\to_R \cup \leftarrow_R$. For a set of equations E, $s \leftrightarrow_E t$ if there are an equation $l \simeq r \in E$, a substitution σ and a position u such that $s|u = l\sigma$ and t is $s[r\sigma]_u$; $s \to_E t$ if $s \leftrightarrow_E t$ and $s \succ t$ for a reduction ordering $\succ$. The term $s|u$ which is being replaced is called a *redex*. We denote by $\leftrightarrow_E^*$ the transitive and reflexive closure of $\leftrightarrow_E$. The relation $\leftrightarrow_E^*$ is a congruence, the congruence defined by E on the set of terms. The equality $\leftrightarrow_E = \to_E \cup \leftarrow_E$ does not hold in general: it holds if and only if the ordering $\succ$ is total in every congruence class defined by E.

All the following definitions apply to both a rewrite system R and a set of equations E. The only difference is the way the relations $\to_R$ and $\to_E$ are defined, as shown above. We denote by $\leftrightarrow_E^*$, $\to_E^*$ and $\leftarrow_E^*$ the transitive and reflexive closure of $\leftrightarrow_E$, $\to_E$ and $\leftarrow_E$ respectively. A term s is in *normal form* with respect to E, or equivalently *E-irreducible*, if there is no term t such that $s \to_E t$. The process of reducing a term to normal form is called *normalization*. A set of equations E is *Church-Rosser* if for all terms s and t, $s \leftrightarrow_E^* t$ implies $s \to_E^* \circ \leftarrow_E^* t$. It is *confluent* if for all terms s and t, $s \leftarrow_E^* \circ \to_E^* t$ implies $s \to_E^* \circ \leftarrow_E^* t$. The Church-Rosser property and the confluence property are equivalent if $\leftrightarrow_E = \to_E \cup \leftarrow_E$ holds. A set of equations E is *locally confluent* if for all terms s and t, $s \leftarrow_E \circ \to_E t$ implies $s \to_E^* \circ \leftarrow_E^* t$. It is *canonical* if it is both confluent and *reduced*, that is for all $l \simeq r \in E$, l and r are in normal form with respect to $E - \{l \simeq r\}$. A set of equations E is a *presentation* of the equational theory $Th(E) = \{\forall \bar{x} s \simeq t | E \models \forall \bar{x} s \simeq t\}$. Two sets of equations are said to be *equivalent*, if they are presentations of the same theory. A classical result in equational logic, known as Birkhoff theorem, states that $E \models \forall \bar{x} s \simeq t$ if and only if $\hat{s} \leftrightarrow_E^* \hat{t}$. Thus two sets of equations E and E' are equivalent if and only

if $\leftrightarrow_E^* = \leftrightarrow_{E'}^*$.

Let P be a finite set of predicate symbols with their arities. We denote by $A(P, F, X)$ and $A(P, F)$ the sets of *atoms* and *ground atoms* on $< P, F, X >$. If P includes the equality predicate $\simeq$, an equation is an atom in $A(P, F, X)$. A *literal* is an atom or a negated atom. A *clause* is a disjunction of literals. A *unit clause* is a clause made of one single literal. A *Horn clause* is a clause which contains at most one positive literal, called the *head*, while the negative literals form the *body* of the clause. All variables occurring in a clause are implicitly universally quantified.

The definitions given for terms and substitutions extend to atoms, literals and clauses. In particular, the subsumption ordering extends to clauses and the encompassment ordering extends to equations as follows: $\varphi \trianglerighteq \psi$ means that $\psi\sigma \subseteq \varphi$ for some substitution σ and ψ has no more literals than φ. If $\varphi \trianglerighteq \psi$ and $\psi \trianglerighteq \varphi$, then $\varphi \stackrel{\bullet}{=} \psi$. If $\varphi \trianglerighteq \psi$ and $\varphi \stackrel{\bullet}{\neq} \psi$, then $\varphi \triangleright \psi$. Similarly, $(p \simeq q) \trianglerighteq (l \simeq r)$ means that $p = c[l\sigma]$ and $q = c[r\sigma]$ for some context c and substitution σ. If c is empty and σ is a renaming, $(p \simeq q) \stackrel{\bullet}{=} (l \simeq r)$; $(p \simeq q) \triangleright (l \simeq r)$ holds if $(p \simeq q) \trianglerighteq (l \simeq r)$ and $(p \simeq q) \stackrel{\bullet}{\neq} (l \simeq r)$.

The two most common simplification orderings are the *recursive path ordering* [35] and the *lexicographic path ordering* [74].

Given n partially ordered sets $(A_1, \succ_1) \ldots (A_n, \succ_n)$ the *lexicographic extension* $\succ_{lex}$ of the orderings $\succ_1 \ldots \succ_n$ is the ordering on $A_1 \times \ldots \times A_n$ defined as follows: $(a_1 \ldots a_n) \succ_{lex} (b_1 \ldots b_n)$ if and only if there exists an i, $1 \leq i \leq n$, such that $a_j = b_j$, $\forall j < i$ and $a_i \succ_i b_i$. If the orderings $\succ_1 \ldots \succ_n$ are well-founded, their lexicographic extension is well-founded as well.

Given a partially ordered set $(A, \succ)$ the *multiset extension* $\succ_{mul}$ of the ordering $\succ$ is the ordering on the set of multisets on A, $M(A)$, defined as follows:

- $\{a\} \cup M \succ_{mul} \emptyset$, where $\emptyset$ is the empty multiset.

- $\{a\} \cup M \succ_{mul} \{a\} \cup N$ if $M \succ_{mul} N$.
- $\{a\} \cup M \succ_{mul} \{b\} \cup N$ if $a \succ b$ and $\{a\} \cup M \succ_{mul} N$.

The multiset extension of a well-founded ordering is well-founded [34].

We assume that $>$ is a partial ordering, called *precedence*, on F. A term $s = f(s_1 \ldots s_n)$ is greater than a term $t = g(t_1 \ldots t_m)$ in the *recursive path ordering*, i.e. $s = f(s_1 \ldots s_n) \succ^{rpo} g(t_1 \ldots t_m) = t$, if and only if one of the following conditions holds:

- $s_i \succeq^{rpo} t$ for some i, $1 \leq i \leq n$.
- $f > g$ and $s \succ^{rpo} t_j$, $\forall j$, $1 \leq j \leq m$.
- $f = g$ and $\{s_1 \ldots s_n\} \succ^{rpo}{}_{mul} \{t_1 \ldots t_m\}$ where $\succ^{rpo}{}_{mul}$ is the multiset extension of $\succ^{rpo}$.

A term $s = f(s_1 \ldots s_n)$ is greater than a term $t = g(t_1 \ldots t_m)$ in the *lexicographic path ordering*, i.e. $s = f(s_1 \ldots s_n) \succ^{lpo} g(t_1 \ldots t_m) = t$, if and only if one of the following conditions holds:

- $s_i \succeq^{lpo} t$ for some i, $1 \leq i \leq n$.
- $f > g$ and $s \succ^{lpo} t_j$, $\forall j$, $1 \leq j \leq m$.
- $f = g$, $(s_1 \ldots s_n) \succ^{lpo}{}_{lex} (t_1 \ldots t_m)$ where $\succ^{lpo}{}_{lex}$ is the lexicographic extension of $\succ^{lpo}$ and $\forall j \geq 2$, $s \succ^{lpo} t_j$.

These two orderings are easy to implement, because their definitions are purely *syntactic*. A third popular ordering is the Knuth-Bendix ordering [84]. Another class of orderings is that of *semantic orderings*, which are based on interpreting the signature of the terms on some partially ordered domain. A survey on orderings and their properties is given in [38].

Simplification orderings on terms can be extended in a coherent way to literals and clauses. For instance, given a precedence $>$ on the set $F \cup P$ of function and predicate symbols, a simplification ordering $\succ$ on terms and literals can be defined as follows: $L = A(s_1 \dots s_n) \succ B(t_1 \dots t_m) = M$, if and only if one of the following conditions holds:

- $s_i \succeq M$ for some i, $1 \leq i \leq n$.
- $A > B$ and $L \succ t_j$, $\forall j$, $1 \leq j \leq m$.
- $A = B = \simeq$ and $\{s_1, s_2\} \succ_{mul} \{t_1, t_2\}$ where $\succ_{mul}$ is the multiset extension of $\succ$.
- $A = B \neq \simeq$ and $(s_1 \dots s_n) \succ_{lex} (t_1 \dots t_m)$ where $\succ_{lex}$ is the lexicographic extension of $\succ$.

A *complete simplification ordering* is a simplification ordering which is total on the set $T(F) \cup A(P, F)$ of ground terms and literals. Once we have a (complete) simplification ordering on terms and literals, we can extend it to (complete) simplification orderings on equations, clauses and sets of clauses by applying the multiset extension to the basic ordering.

2.2 Proof orderings for theorem proving

A *theorem proving problem* in equational logic consists in finding a proof of an equation $\forall \bar{x}\ s \simeq t$, with universally quantified variables, in a given set of equations S. An equation $\forall \bar{x} s \simeq t$ which contains only universally quantified variables can be regarded as a ground equation and we write it as $\hat{s} \simeq \hat{t}$. The set S is a *presentation* of the equational theory $Th(S)$ of all the theorems of S, $Th(S) = \{\psi \mid S \models \psi\}$. Here and in the rest of this chapter, ψ and φ denote equations. The equation to be

proved is called the *target* or *goal*. We write $(S; \varphi)$ to denote the problem of proving φ from S. A *theorem proving derivation* is a sequence of deductions

$$(S_0; \varphi_0) \vdash (S_1; \varphi_1) \vdash \ldots \vdash (S_i; \varphi_i) \vdash \ldots,$$

where at each step the problem of deciding $\varphi_i \in Th(S_i)$ reduces to the problem of deciding $\varphi_{i+1} \in Th(S_{i+1})$. A step $(S_i; \varphi_i) \vdash (S_{i+1}; \varphi_i)$, where the presentation is modified, is a *forward reasoning* step. A step $(S_i; \varphi_i) \vdash (S_i; \varphi_{i+1})$, where the target is modified, is a *backward reasoning* step, which derives a new goal from the current one and the presentation. Informally, a derivation halts successfully at stage k if it reaches a solution, denoted by the dummy target $true$. A pair $(S_k; true)$ indicates that the target is proved. As a theorem proving derivation transforms a given problem, it is intuitively desirable that it reduces it to one which is in some sense "smaller". Thus, we need to identify what is being reduced during a theorem proving derivation. We observe that if a target φ_0 is indeed a theorem of the input set S_0, then there exist some proofs of φ_0 in S_0. On the other hand, the proof of the dummy target $true$ is *empty*. At any stage $(S_i; \varphi_i)$ in between there is a (non-unique) *minimal* proof of φ_i in S_i, which represents a least amount of work which still needs to be done in order to prove φ_i from S_i. If the derivation gets closer to a solution, a minimal proof of the target gets reduced, i.e. the amount of work which is left becomes smaller. When the problem is solved, no more work needs to be done. Therefore, we regard theorem proving as *reduction of a minimal proof of the target* to the empty proof.

In order to compare proofs and to have a notion of minimal proofs, we need an *ordering* of proofs. This ordering needs to be *well-founded*, having as bottom element the empty proof, denoted by ε. A notion of well-founded orderings on proofs, called *proof orderings*, was introduced in [7, 9]: a *proof ordering* is a monotonic, stable and well-founded ordering on proofs [7]. Proofs orderings are defined in general starting from some ordering on the data involved in the proofs. Thus, we assume to have a complete simplification ordering $\succ$ on terms and literals. We prefer to have

a simplification ordering, although a well-founded, monotonic and stable ordering total on ground terms would be sufficient. The following example [44] shows how to define a proof ordering starting from an ordering on terms:

Example 2.2.1 *Equational proofs can be represented as chains [7]*

$$s_1 \leftrightarrow_{l_1 \simeq r_1} s_2 \leftrightarrow_{l_2 \simeq r_2} \ldots \leftrightarrow_{l_{n-1} \simeq r_{n-1}} s_n,$$

where $s_1 \leftrightarrow_{l_1 \simeq r_1} s_2$ *means that the equality of* s_1 *and* s_2 *is established by the equation* $l_1 \simeq r_1$*, because* s_1 *and* s_2 *are* $c[l_1\sigma]$ *and* $c[r_1\sigma]$ *for some context* c *and substitution* σ*. We write* $s \rightarrow_{l \simeq r} t$ *if* $s \succ t$ *is known a priori. A proof ordering to compare ground equational proofs can be defined as follows. We associate to a ground equational step* $s \leftrightarrow_{l \simeq r} t$ *the triple* (s, l, t)*, if* $s \succ t$*. We compare these triples by the lexicographic combination* $>^e$ *of the complete simplification ordering* $\succ$*, the strict encompassment ordering* $\triangleright$ *and again the ordering* $\succ$*. Then we compare two proofs* $s \leftrightarrow^*_E t$ *and* $s \leftrightarrow^*_{E'} t$ *by the multiset extension* $>^e{}_{mul}$ *of* $>^e$*.*

Proof orderings were introduced in [7] to prove that the Knuth-Bendix completion procedure generates confluent term rewriting systems. A derivation by Knuth-Bendix completion in that context is a process of transforming a presentation

$$S_0 \vdash S_1 \vdash \ldots \vdash S_i \vdash \ldots.$$

In other words, it is a purely forward derivation, with no target. A confluent rewrite system is a presentation such that for all theorems $s \simeq t$ there is a *rewrite proof*, i.e. a proof in the form $s \rightarrow^* \circ \leftarrow^* t$. Therefore, correctness of Knuth-Bendix completion is proved by showing that *all* the proofs in the theory are eventually reduced to rewrite proofs during the derivation. Since such a derivation transforms *only* the presentation, with the purpose of reducing *all* the proofs, one needs to compare the proof of φ in S_i with the proof of φ in S_{i+1} for all the theorems φ in the theory. For this reason, proof orderings are applied in [7] to compare only proofs of the same

theorem. In a theorem proving derivation

$$(S_0; \varphi_0) \vdash (S_1; \varphi_1) \vdash \ldots \vdash (S_i; \varphi_i) \vdash \ldots$$

both the presentation *and* the target are transformed. In order to compare the proof of φ_i in S_i and the proof of φ_{i+1} in S_{i+1}, we need a proof ordering such that proofs of different theorems may be comparable. Proof orderings with this property do exist and can actually be obtained quite easily. For instance the proof ordering of the previous example can be extended as follows:

Example 2.2.2 *We can compare any two ground equational proofs $s \leftrightarrow^*_E t$ and $s' \leftrightarrow^*_{E'} t'$ by comparing the pairs $(\{s,t\}, s \leftrightarrow^*_E t)$ and $(\{s',t'\}, s' \leftrightarrow^*_{E'} t')$ by the lexicographic combination $>_u$ of the multiset extension $\succ_{mul}$ of the ordering $\succ$ on terms and the multiset extension ${>^e}_{mul}$ of $>^e$.*

Henceforth a *proof ordering* is a monotonic, stable, well-founded ordering on proofs, such that proofs of different theorems may be comparable and the minimum proof is the *empty proof* ε. Correspondingly, the minimum among terms and literals is the dummy element *true*, which represents the theorem whose proof is ε. Given a proof ordering $>_p$, we denote by $\Pi(S, \varphi)$ the set of all the *minimal proofs* of φ from S with respect to $>_p$. By assuming a proof ordering $>_p$, we can regard a successful theorem proving derivation

$$(S_0; \varphi_0) \vdash (S_1; \varphi_1) \vdash \ldots \vdash (S_i; \varphi_i) \vdash \ldots,$$

as a process of reducing a minimal proof of φ_0 in S_0 to the empty proof and φ_0 to *true*. At each step $\Pi(S_i, \varphi_i)$ is replaced by $\Pi(S_{i+1}, \varphi_{i+1})$ and the derivation halts successfully at stage k if $\Pi(S_k, \varphi_k) = \{\varepsilon\}$ and φ_k is *true*. The introduction of the symbol *true* is not a mere formality. For instance, in equational logic a theorem $s \simeq s$ is regarded as trivially true. However, it is not so in equational theorem proving, since a procedure needs to check that the two sides of the equation are

identical before stating that the theorem is true. This requires an inference step and therefore a derivation that has reached the state $(S; s \simeq s)$ is not successful yet. Indeed, the proof of $s \simeq s$ is not empty. Thus we need the symbol *true* to indicate the success of a derivation.

We have extended the classical application of proof orderings to include theorem proving, where proofs of different targets need to be compared. This extension is technically simple, but it is important for the interpretation of completion procedures as semidecision procedures. The classical proof orderings approach was not conceived for theorem proving because it does not provide for the target and the transformation of the target. On the other hand, our proof orderings approach applies to both theorem proving and traditional completion.

2.3 Inference rules and search plans

We start by introducing some basic concepts about theorem proving strategies. A *theorem proving strategy* is a pair $\mathcal{C} =< I; \Sigma >$, where I is a set of *inference rules* and Σ is a *search plan*. Inference rules in I decide what consequences can be deduced from the available data and Σ decides which inference rule and which data to choose next. We discuss first the inference rules and next the search plan. The general form of an inference rule f is:

$$f\colon \frac{S}{S'}$$

where S and S' are sets of sentences. The rule says that given S, the set S' can be inferred. We distinguish between *contraction* inference rules and *expansion* inference rules, as they are called in [44]. A **contraction** inference rule contracts a given set S into a new set S' by either deleting some sentences in S or replacing them by others:

f: $\frac{S}{S'}$ where $S \not\subseteq S'$.

Example 2.3.1 *The most important contraction rule for equality is* Simplification*:*

$$\frac{(S \cup \{p \simeq q, l \simeq r\}; \hat{s} \simeq \hat{t})}{(S \cup \{p[r\sigma]_u \simeq q, l \simeq r\}; \hat{s} \simeq \hat{t})} \quad p|u = l\sigma, \quad p \succ p[r\sigma]_u, \quad p \trianglerighteq l \vee q \succ p[r\sigma]_u.$$

A simplification step replaces an equation by a smaller and yet equivalent equation. The condition $p|u = l\sigma$, which may be read as l *matches* the subterm at position u in p, says that the terms p and $p[r\sigma]_u$ are equal in any equational theory where $l \simeq r$ holds. Thus the equations $p \simeq q$ and $p[r\sigma]_u \simeq q$ are equivalent in any such theory. The condition $p \succ p[r\sigma]_u$ says that p is rewritten to a smaller term and thus $p[r\sigma]_u \simeq q$ is smaller than $p \simeq q$. This makes Simplification a contraction rule. For instance, if $x \cdot y \simeq y \cdot x$ is given and $a \succ b$ in the ordering on terms, $(a \cdot b) \cdot i(a) \simeq b$ can be rewritten to $(b \cdot a) \cdot i(a) \simeq b$. The additional restriction $p \trianglerighteq l \vee q \succ p[r\sigma]_u$ is satisfied in this example, because $(a \cdot b) \cdot i(a) \trianglerighteq x \cdot y$, as the instance $a \cdot b$ of $x \cdot y$ occurs in $(a \cdot b) \cdot i(a)$. An example of a step, where the second disjunct $q \succ p[r\sigma]_u$ is satisfied, is applying $e \cdot x \simeq x$ to reduce $e \cdot x \simeq x \cdot e$ to $x \simeq x \cdot e$: $p \trianglerighteq l$ does not hold, since $p = l = e \cdot x$, but $x \cdot e \succ x$. The condition $p \trianglerighteq l \vee q \succ p[r\sigma]_u$ is explained as follows. Since $p|u = l\sigma$, $p \trianglerighteq l$ holds. If $p \trianglerighteq l$, either p is a proper instance of l or l matches a proper subterm of p. The side l of the applied equation is strictly smaller than the simplified term according to the encompassment ordering and this is a sufficient condition, together with $p \succ p[r\sigma]_u$, for the simplification step to reduce the equational proofs where $p \simeq q$ occurs. If $p \doteq l$, p and l are variants, and therefore uncomparable in the encompassment ordering. In this case, $q \succ p[r\sigma]_u$ is required to hold, that is, the newly generated term $p[r\sigma]_u$ is smaller than both sides of the simplified equation $p \simeq q$. This condition represents a restriction only if $q \not\succ p$, which means simplification applies to the greater side of $p \simeq q$ or $p \simeq q$ is not ordered. If $q \succ p$, simplification applies to the smaller side of $p \simeq q$ and the condition $q \succ p[r\sigma]_u$ is trivially satisfied. These conditions guarantee that Simplification steps reduce equational proofs (see Lemma 2.9.1 in Section 2.9.1).

A simplification step where $l \simeq r$ simplifies $p \simeq q$ is a **forward contraction** step, if $l \simeq r$ has been generated earlier than $p \simeq q$, a **backward contraction** step, otherwise. In general, a contraction step, where the removal of a premise ψ_1 is justified by the remaining premises $\psi_2 \dots \psi_n$, is a forward contraction step, if the premises $\psi_2 \dots \psi_n$ have been generated earlier than ψ_1, a backward contraction step, otherwise.

An **expansion** inference rule expands a given set S into a new set S' by deriving new sentences from sentences in S:

$$f\colon \frac{S}{S'} \text{ where } S \subset S'.$$

Example 2.3.2 *Deduction of a critical pair by* Superposition *[62] is an* expansion *inference rule on the presentation, since it adds to the given set a new equation:*

$$\frac{(S \cup \{p \simeq q, l \simeq r\}; \hat{s} \simeq \hat{t})}{(S \cup \{p \simeq q, l \simeq r, p[r]_u\sigma \simeq q\sigma\}; \hat{s} \simeq \hat{t})} \qquad p|u \notin X, \quad (p|u)\sigma = l\sigma, \quad p\sigma \not\preceq q\sigma, p[r]_u\sigma$$

where X is the set of variables and σ is the most general unifier of $(p|u)$ and l.

The equation $l \simeq r$ superposes onto the equation $p \simeq q$, if there exists a non-variable subterm $p|u$, which unifies with mgu σ with l. This means that the term $p\sigma$ is equal to both $q\sigma$ and $p[r]_u\sigma$. The new equation $p[r]_u\sigma \simeq q\sigma$ is called *critical pair* since the first appearance of superposition in [84]. We shall also use *raw critical pair* to indicate a newly generated equation, which has not been reduced by any other equation. The key point in the Superposition rule is that the step is performed only if $p\sigma \not\preceq q\sigma$ and $p\sigma \not\preceq p[r]_u\sigma$, i.e. the deduction of new equations is restricted by the same ordering used in Simplification. For instance, given the equations $(x \cdot y) \cdot y \simeq x \cdot (y \cdot y)$ and $z \cdot (-w) \simeq -(z \cdot w)$, the second one superimposes onto the first one, by unifying the side $z \cdot (-w)$ with the non-variable subterm $x \cdot y$. The unifier assigns z to x and $-w$ to y, yielding the instance $(z \cdot (-w)) \cdot (-w)$ of $(x \cdot y) \cdot y$. This term can be rewritten to $z \cdot ((-w) \cdot (-w))$ by using the first equation

and to $-(z \cdot w) \cdot (-w)$ by using the second one. These two terms are thus equal in the theory and then we deduce the equation $z \cdot ((-w) \cdot (-w)) \simeq -(z \cdot w) \cdot (-w)$. The ordering-based restriction is satisfied if the ordering $\succ$ on terms is such that $(z \cdot (-w)) \cdot (-w) \succ z \cdot ((-w) \cdot (-w))$ and $(z \cdot (-w)) \cdot (-w) \succ -(z \cdot w) \cdot (-w)$ hold. This is the case if $\succ$ is a lexicographic path ordering and $\cdot > -$ in the precedence on function symbols.

We further distinguish between inference rules which transform the presentation (forward reasoning) and inference rules which transform the target[1] (backward reasoning):

- *Presentation inference rules*:
  - *Expansion inference rules*: f: $\dfrac{(S;\varphi)}{(S';\varphi)}$ where $S \subset S'$.
  - *Contraction inference rules*: f: $\dfrac{(S;\varphi)}{(S';\varphi)}$ where $S \not\subseteq S'$.
- *Target inference rules*:
  - *Expansion inference rules*: f: $\dfrac{(S;\varphi)}{(S;\varphi')}$ where φ logically implies φ'.
  - *Contraction inference rules*: f: $\dfrac{(S;\varphi)}{(S;\varphi')}$ where φ does not logically imply φ'.

Example 2.3.3 Simplification *also applies to the target:*

$$\frac{(S \cup \{l \simeq r\}; \hat{s} \simeq \hat{t})}{(S \cup \{l \simeq r\}; \hat{s}[r\sigma]_u \simeq \hat{t})} \quad \hat{s}|u = l\sigma, \quad \hat{s} \succ \hat{s}[r\sigma]_u.$$

For an interesting example of expansion inference rule for the target, we refer to Section 2.9.3.

The inference rules are required to be *sound*:

[1] The target can be formulated as a set of equations. For convenience of representation, we write the target as a single equation.

Definition 2.3.1 *An inference step* $(S;\varphi) \vdash (S';\varphi')$ *is* sound *if* $Th(S') \subseteq Th(S)$, monotonic *if* $Th(S) \subseteq Th(S')$. *It is* relevant *if* $\varphi' \in Th(S')$ *if and only if* $\varphi \in Th(S)$.

Soundness ensures that a presentation inference step does not create new elements which are not true in the theory. Monotonicity guarantees that all theorems are preserved. Relevance ensures that a target inference step replaces the target by a new target in such a way that proving the latter is equivalent to proving the former. For instance, a simplification step which reduces a target φ to φ' satisfies the relevance requirement because if φ' is true, φ is true as well.

A search plan Σ decides which inference rule should be applied to what data at any given step during a derivation. It may set a *precedence* on the inference rules and a *well-founded ordering* on data and proceed accordingly:

Example 2.3.4 *A* Simplification-first *search plan [62] is a search plan where Simplification has priority over all the expansion rules in the strategy. Therefore, Superposition is considered only if Simplification does not apply to any equation. Consequently, the current set of equations is always kept as reduced as possible. Equations can be sorted by the multiset extension* $\succ_{mul}$ *of the ordering on terms, or by size, or by age such as in a first-in first-out plan.*

Different schemes for inference rules, called *deduction* and *deletion*, are given in [16]. The deduction scheme in [16] is the same as our expansion scheme. The deletion scheme instead is different from our contraction scheme, since it only allows to infer S' from S by deleting a sentence in S. If this deletion scheme is adopted, an inference rule which replaces a sentence by other sentences has to be schematized as the composition of an expansion rule and a deletion rule. For instance, Simplification of $p \simeq q$ into $p \simeq q[r\sigma]_u$ is described as the generation of the equation $p \simeq q[r\sigma]_u$ followed by the deletion of $p \simeq q$. This approach has the drawback that it requires to consider more general inference rules than those actually used in the set of inference rules of a given strategy. For the simplification of $p \simeq q$ into $p \simeq q[r\sigma]_u$, the equation

$p \simeq q[r\sigma]_u$ may not be a critical pair derivable by a superposition step. This is the case if $p \succ q$, that is the right hand side of a rewrite rule is simplified. A general paramodulation inference rule, which is not featured by most actual strategies, e.g. the Unfailing Knuth-Bendix procedure, is then required in order to simulate simplification. Our expansion/contraction schemes have the advantage that they allow us to classify directly the concrete inference rule of existing strategies as either an expansion rule or a contraction rule.

2.4 Theorem proving as proof reduction

Our view of theorem proving as proof reduction leads to the requirement that the inference rules of a strategy are *proof-reducing*. As we derive $(S_{i+1}; \varphi_{i+1})$ from $(S_i; \varphi_i)$, the set $\Pi(S_i, \varphi_i)$ is replaced by $\Pi(S_{i+1}, \varphi_{i+1})$. Clearly, we need to forbid all inference steps which would replace a proof P in $\Pi(S_i, \varphi_i)$ by a proof Q in $\Pi(S_{i+1}, \varphi_{i+1})$ such that $Q >_p P$. Such steps certainly do not help. On the other hand, we cannot impose that at every step a minimal proof of the target be reduced. This is impossible, since theorem proving is a process of search and therefore many steps generally do not contribute to the final result. We require that for every step $(S_i; \varphi_i) \vdash_{\mathcal{C}} (S_{i+1}; \varphi_{i+1})$, every proof P in $\Pi(S_i, \varphi_i)$ is either preserved, i.e. P is also in $\Pi(S_{i+1}, \varphi_{i+1})$, or reduced, i.e. P is replaced by a proof Q in $\Pi(S_{i+1}, \varphi_{i+1})$ such that $Q <_p P$:

Definition 2.4.1 *An inference step* $(S; \varphi) \vdash (S'; \varphi')$ *is* proof-reducing on φ *if for all* $P \in \Pi(S, \varphi)$*, either* $P \in \Pi(S', \varphi')$ *or there exists a* $Q \in \Pi(S', \varphi')$ *such that* $P >_p Q$*. If the latter holds for some* $P \in \Pi(S, \varphi)$*, then the step is* strictly proof-reducing*.*

A target inference step modifies the target and therefore we require that it is proof-reducing on the target itself:

Definition 2.4.2 *A target inference step* $(S; \varphi) \vdash (S; \varphi')$ *is* (strictly) proof-reducing

if it is (strictly) proof-reducing on φ.

Example 2.4.1 *Simplification of the target as given in Example 2.3.3 is strictly proof-reducing. If the target* $\hat{s} \simeq \hat{t}$ *is replaced by the target* $\hat{s}' \simeq \hat{t}'$ *because* $\hat{s}$ *is simplified to* $\hat{s}'$*, we have* $\hat{s} \succ \hat{s}'$*,* $\hat{t} = \hat{t}'$ *and therefore* $\{\hat{s}, \hat{t}\} \succ_{mul} \{\hat{s}', \hat{t}'\}$*. If we assume the proof ordering* $>_u$ *introduced in Example 2.2.2, it follows that* $\hat{s} \leftrightarrow_E^* \hat{t} >_u \hat{s}' \leftrightarrow_E^* \hat{t}'$.

For a presentation inference step we allow more flexibility, because an inference step on the presentation may not immediately reduce any proof of the target and still be necessary to decrease it eventually. The proof reduction effects of a presentation inference step need to be checked on a larger set of theorems in the theory, not just on the given target. We call *domain*, and we denote by $\mathcal{T}$, the set of sentences where the presentation inference rules are proof-reducing:

Definition 2.4.3 *A presentation inference step* $(S; \varphi) \vdash (S'; \varphi)$ *is* proof-reducing *on* $\mathcal{T}$ *if*

1. *either it is strictly proof-reducing on* φ
2. *or*
   - *(a)* $\Pi(S, \varphi) = \Pi(S', \varphi)$,
   - *(b)* $\forall \psi \in \mathcal{T}$, $(S; \psi) \vdash (S', \psi)$ *is proof-reducing on* ψ *and*
   - *(c)* $\exists \psi \in \mathcal{T}$ *such that* $(S; \psi) \vdash (S', \psi)$ *is strictly proof-reducing on* ψ.

This condition says that either the step reduces a proof of the target, or it reduces a proof of at least one theorem in $\mathcal{T}$, while it does not increase any proof of any theorem in $\mathcal{T}$. A step which reduces a proof of the target is proof-reducing, regardless of its effects on other theorems. On the other hand, a step which does not affect any proof of the target is also proof-reducing, provided it does not increase any proof and strictly decreases at least one. For most completion procedures the domain $\mathcal{T}$

is actually the set of all the sentences in the given signature. For instance, for the *Knuth-Bendix completion procedure* $\mathcal{T}$ is the set of all equations. For the *Unfailing Knuth-Bendix procedure*, $\mathcal{T}$ is the set of all ground equations. However, we prefer to give a notion of proof reduction which is parametric with respect to $\mathcal{T}$. In fact, it would be desirable to have completion procedures whose domain is smaller than the set of all ground equations or clauses and is dependent on the given target. Intuitively, a procedure with a target-dependent domain should be more efficient, because it would not spend time in reducing proofs of theorems that are not related to the given target.

Example 2.4.2 *Deduction of a critical pair as given in Example 2.3.2 is proof-reducing on the domain $\mathcal{T}$ of all ground equations. We assume the proof ordering $>_u$ introduced in Example 2.2.2. Given two equations $l \simeq r$ and $p \simeq q$ in S, a* critical overlap *of $l \simeq r$ and $p \simeq q$ is any proof $s \leftarrow_{l\simeq r} v \rightarrow_{p\simeq q} t$, where v is $c[p\tau]$ for some context c and substitution τ, t is $c[q\tau]$, $(p|u)\tau = l\tau$ for some non variable subterm $p|u$ of p and s is $c[p[r]_u\tau]$. The Superposition rule applied to $l \simeq r$ and $p \simeq q$ generates the critical pair $p[r]_u\sigma \simeq q\sigma$, where σ is the mgu of $p|u$ and l and therefore $\tau = \sigma\rho$ for some substitution ρ. The proof $s \leftrightarrow_{p[r]_u\sigma\simeq q\sigma} t$, justified by the critical pair, is smaller than the proof $s \leftarrow_{l\simeq r} v \rightarrow_{p\simeq q} t$: since $v \succ s$ and $v \succ t$, we have $\{(v,l,s),(v,p,t)\} >^e_{mul} \{(s,p[r]_u\sigma,t)\}$ or $\{(v,l,s),(v,p,t)\} >^e_{mul} \{(t,q\sigma,s)\}$, depending on whether $s \succ t$ or $t \succ s$. Therefore, every minimal proof which contains $s \leftarrow_{l\simeq r} v \rightarrow_{p\simeq q} t$ as a subproof is no longer minimal after the generation of the critical pair. Such a proof is replaced by the smaller proof where all occurrences of $s \leftarrow_{l\simeq r} v \rightarrow_{p\simeq q} t$ are replaced by $s \leftrightarrow_{p[r]_u\sigma\simeq q\sigma} t$: for all $\psi \in \mathcal{T}$, $\Pi(S \cup \{p[r]_u\sigma \simeq q\sigma\}, \psi) = \Pi(S,\psi) - \{P[s \leftarrow_{l\simeq r} v \rightarrow_{p\simeq q} t]\} \cup \{P[s \leftrightarrow_{p[r]_u\sigma\simeq q\sigma} t]\}$. If a minimal proof of the target itself contains a critical overlap between $l \simeq r$ and $p \simeq q$, the Deduction step is strictly proof-reducing.*

This notion of proof reduction applies to presentation inference steps which are either expansion steps or contraction steps which replace some sentences by others. A contraction step which deletes sentences without adding any cannot reduce any

minimal proof. In order to characterize these steps, we need a notion of *redundancy*:

Definition 2.4.4 *A sentence φ is* redundant in S on ψ *if* $\Pi(S, \psi) = \Pi(S \cup \{\varphi\}, \psi)$*; it is* redundant in S on domain $\mathcal{T}$ *if it is redundant on all* $\psi \in \mathcal{T}$*.*

A sentence is redundant in a presentation on a specific target, if adding it to the presentation does not affect any minimal proof of the target. If this holds on the entire domain, the sentence is said to be redundant on the domain.

Example 2.4.3 *An inference rule, which deletes an equation without adding any, is* Functional subsumption*:*

$$\frac{(S \cup \{p \simeq q, l \simeq r\}; \hat{s} \simeq \hat{t})}{(S \cup \{l \simeq r\}; \hat{s} \simeq \hat{t})} \; (p \simeq q) \trianglerighteq (l \simeq r).$$

For instance, the equations $x \cdot y \simeq y \cdot x$ and $(w \cdot v) \cdot z + e \simeq z \cdot (w \cdot v) + e$ satisfy the condition $(w{\cdot}v){\cdot}z{+}e \simeq z{\cdot}(w{\cdot}v){+}e \trianglerighteq x{\cdot}y \simeq y{\cdot}x$, as the instance $(w{\cdot}v){\cdot}z \simeq z{\cdot}(w{\cdot}v)$ of $x \cdot y \simeq y \cdot x$ occurs in the context $\ldots{+}e$. The equality $x \cdot y \simeq y \cdot x$ suffices to establish also the equality of $(w \cdot v) \cdot z + e$ and $z \cdot (w \cdot v) + e$ and therefore the larger equation is not necessary. Similar to simplification, a subsumption step is a *forward contraction* step, if $l \simeq r$ has been generated earlier than $p \simeq q$, a *backward contraction* step, otherwise. In fact, the distinction between forward and backward contraction steps was introduced originally for subsumption. An equation $p \simeq q$ subsumed by $l \simeq r$ is redundant according to the proof ordering ${>^e}_{mul}$ and therefore to the proof ordering $>_u$ as defined in Example 2.2.2. If $(p \simeq q) \trianglerighteq (l \simeq r)$, no minimal proof contains a step $s \leftrightarrow_{p \simeq q} t$ since the step $s \leftrightarrow_{l \simeq r} t$ is smaller: either $\{(s, p, t)\} {>^e}_{mul} \{(s, l, t)\}$ or $\{(t, q, s)\} {>^e}_{mul} \{(t, r, s)\}$, depending on whether $s \succ t$ or $t \succ s$, since $p \trianglerighteq l$ and $q \trianglerighteq r$. Thus, for all ground equations $s \simeq t \in \mathcal{T}$, $\Pi(S \cup \{p \simeq q, l \simeq r\}, s \simeq t) = \Pi(S \cup \{l \simeq r\}, s \simeq t)$. If $(p \simeq q) \stackrel{\bullet}{=} (l \simeq r)$, the two equations are equal up to variable renaming and therefore have the same ground instances. For all ground equations $s \simeq t$, $\Pi(S \cup \{p \simeq q, l \simeq r\}, s \simeq t) = \Pi(S \cup \{l \simeq r\}, s \simeq t)$, i.e. $p \simeq q$ is redundant in $S \cup \{l \simeq r\}$. Symmetrically, $l \simeq r$ is redundant is $S \cup \{p \simeq q\}$. Either one of the two

equations may be deleted, but noth both, since neither one is redundant in S.

Definition 2.4.5 *An inference step* $(S;\varphi) \vdash (S';\varphi')$ *is* reducing *on* $\mathcal{T}$ *(on* φ*) if either it is proof-reducing on* $\mathcal{T}$ *(on* φ*) or it deletes a sentence which is redundant in* S *on domain* $\mathcal{T}$ *(on* φ*).*

The notion of reducing steps uses redundancy to characterize those steps which eliminate redundant sentences in the presentation. However, we remark that Definition 2.4.4 does not require that a redundant sentence in S is actually in S. The notion of redundancy also applies to elements which are not in the presentation, but may be derived from the presentation. For instance, it is plain to observe that an expansion step which adds a redundant sentence is not proof-reducing.

Definition 2.4.6 *An inference rule* f *is* reducing *if all the inference steps* $(S;\varphi)\vdash_f(S';\varphi')$ *where* f *is applied are reducing.*

We have finally all the elements to define a completion procedure:

Definition 2.4.7 *A theorem proving strategy* $\mathcal{C} =< I;\Sigma >$ *is a* completion procedure *on domain* $\mathcal{T}$ *if for all pairs* $(S_0;\varphi_0)$*, where* S_0 *is a presentation of a theory and* $\varphi_0 \in \mathcal{T}$*, the derivation*

$$(S_0;\varphi_0)\vdash_{\mathcal{C}}(S_1;\varphi_1)\vdash_{\mathcal{C}}\ldots\vdash_{\mathcal{C}}(S_i;\varphi_i)\vdash_{\mathcal{C}}\ldots$$

has the following properties:

- *soundness:* $\forall i \geq 0$, $Th(S_{i+1}) \subseteq Th(S_i)$,
- *relevance:* $\forall i \geq 0$, $\varphi_i \in \mathcal{T}$ *and* $\varphi_{i+1} \in Th(S_{i+1})$ *if and only if* $\varphi_i \in Th(S_i)$ *and*
- *reduction:* $\forall i \geq 0$, *the step* $(S_i;\varphi_i)\vdash_{\mathcal{C}}(S_{i+1};\varphi_{i+1})$ *is reducing on* $\mathcal{T}$ *(on* φ_i*).*

The definition requires soundness but not monotonicity, because soundness and relevance together are sufficient for theorem proving. Monotonicity will be added only for the application of completion to the generation of confluent sets. *Reduction* is the fundamental property of completion procedures. Clearly, if all the inference rules of a procedure are reducing, the procedure has the reduction property. We shall see in Section 2.9 that the inference rules of the known equational completion procedures are reducing. Most inference rules are reducing because they are suitably restricted by the complete simplification ordering $\succ$ on terms. A complete simplification ordering on data turns out to be a key element in characterizing a theorem proving strategy as a completion procedure.

2.5 Fairness and completeness

A theorem proving method is complete if, whenever φ_0 is a theorem of S_0, the derivation from $(S_0; \varphi_0)$ succeeds. Completeness involves both the inference rules and the search plan. First, it requires that if $\varphi_0 \in Th(S_0)$, there exist successful derivations by the inference rules of the procedure. Second, it requires that whenever successful derivations exist, the search plan guarantees that the computed derivation is successful. We call these two properties *refutational completeness* of the inference rules and *fairness* of the search plan respectively. In order to describe them, we introduce a structure called *I-tree*. Given a theorem proving problem $(S_0; \varphi_0)$ and a set of inference rules I, the application of I to $(S_0; \varphi_0)$ defines a tree, the *I-tree rooted at* $(S_0; \varphi_0)$. The nodes of the tree are labeled by pairs $(S; \varphi)$. The root is labeled by the input pair $(S_0; \varphi_0)$. A node $(S; \varphi)$ has a child $(S'; \varphi')$ if $(S'; \varphi')$ can be derived from $(S; \varphi)$ in one step by an inference rule in I. In general, the I-tree is a directed graph, rather than a tree, since a node may be reachable starting from the root by more than one path. However, it is always possible to transform it into a tree by allowing different nodes to have the same label. The I-tree rooted at $(S_0; \varphi_0)$ represents all the possible derivations by the inference rules in I starting

from $(S_0; \varphi_0)$.

Intuitively, a set I of inference rules is *refutationally complete* if whenever $\varphi_0 \in Th(S_0)$, the I-tree rooted at $(S_0; \varphi_0)$ contains successful nodes, nodes of the form $(S; true)$. We use the term "refutational completeness" for the inference rules to differentiate it from the completeness of the theorem proving strategy. Furthermore, "refutational" emphasizes that the goal is to prove a specific theorem. The following definition is an equivalent characterization of this concept in terms of proof reduction:

Definition 2.5.1 *A set I of inference rules is* refutationally complete *if whenever $\varphi \in Th(S)$ and $\Pi(S, \varphi) \neq \{\varepsilon\}$, $\forall P \in \Pi(S, \varphi)$ there exists a path*

$$(S; \varphi) \vdash_I (S_1; \varphi_1) \vdash_I \ldots \vdash_I (S'; \varphi')$$

such that $P >_p Q$ for some $Q \in \Pi(S', \varphi')$.

A set of inference rules is refutationally complete if it can reduce any non empty proof of the target. Since a proof ordering is well-founded, it follows that if $\varphi \in Th(S)$, the I-tree rooted at $(S; \varphi)$ contains successful nodes. The advantage of giving the definition of completeness in terms of proof reduction is that the problem of proving completeness of I is reduced to the problem of exhibiting a suitable proof ordering [14].

Given a completion procedure $\mathcal{C} =< I; \Sigma >$, the I-tree rooted at $(S_0; \varphi_0)$ represents all the derivations that the procedure can generate from the input $(S_0; \varphi_0)$. The search plan Σ selects a path in the I-tree: the derivation from input $(S_0; \varphi_0)$ controlled by Σ is the path selected by Σ in the I-tree rooted at $(S_0; \varphi_0)$. Once both a set of inference rules and a search plan are given, the derivation from $(S_0; \varphi_0)$ is unique. A pair $(S_i; \varphi_i)$ reached at stage i of the derivation is a *visited node* in the I-tree. Each visited node $(S_i; \varphi_i)$ may have many children, but the search plan selects only one of them to be $(S_{i+1}; \varphi_{i+1})$. A search plan Σ is *fair* if whenever the I-tree

rooted at $(S_0; \varphi_0)$ contains successful nodes, the derivation controlled by Σ starting at $(S_0; \varphi_0)$ is guaranteed to reach a successful node. Similar to completeness, we rephrase this concept in terms of proof reduction:

Definition 2.5.2 *A derivation*

$$(S_0; \varphi_0) \vdash_{\mathcal{C}} (S_1; \varphi_1) \vdash_{\mathcal{C}} \ldots \vdash_{\mathcal{C}} (S_i; \varphi_i) \vdash_{\mathcal{C}} \ldots$$

controlled by a search plan Σ is fair *if and only if for all $i \geq 0$, for all $P \in \Pi(S_i, \varphi_i)$, if there exists a path*

$$(S_i; \varphi_i) \vdash_I \ldots \vdash_I (S'; \varphi')$$

in the I-tree rooted at $(S_0; \varphi_0)$ such that $P >_p Q$, for some $Q \in \Pi(S', \varphi')$, then there exists an $(S_j; \varphi_j)$, for some $j > i$, and an $R \in \Pi(S_j, \varphi_j)$ such that $Q \geq_p R$. A search plan Σ is fair *if all the derivations controlled by Σ are fair.*

In other words, if the inference rules can reduce a proof of the target at $(S_i; \varphi_i)$, a fair search plan guarantees that such proof will have been reduced at a later stage $(S_j; \varphi_j)$. This definition is target-oriented because it only requires that the proofs of the intended target are reduced. Actually it only requires that *one* proof of the target is reduced. If a proof of φ is reduced to ε at stage j, the set of minimal proofs of φ collapses to $\{\varepsilon\}$ and $P >_p \varepsilon$ for every proof P considered at all stages earlier than j. In theorem proving we are only interested in finding one proof of the target and therefore a search plan may trim the search space considerably and still be fair as long as it does not remove the possibility of finding any proof.

If the inference rules are refutationally complete and the search plan is fair, a completion procedure on domain $\mathcal{T}$ is *complete*, i.e. it is a *semidecision procedure* for $Th(S) \cap \mathcal{T}$ for all presentations S:

Theorem 2.5.1 *If a completion procedure $\mathcal{C}$ on domain $\mathcal{T}$ has refutationally complete inference rules and fair search plan, then for all derivations*

$$(S_0;\varphi_0)\vdash_{\mathcal{C}}(S_1;\varphi_1)\vdash_{\mathcal{C}}\ldots\vdash_{\mathcal{C}}(S_i;\varphi_i)\vdash_{\mathcal{C}}\ldots,$$

where $\varphi_0 \in Th(S_0)$, $\forall i \geq 0$, *if* $\Pi(S_i,\varphi_i) \neq \{\varepsilon\}$, *then for all* $P \in \Pi(S_i,\varphi_i)$, *there exists an* (S_j,φ_j), *for some* $j > i$, *such that* $P >_p R$ *for some* $R \in \Pi(S_j,\varphi_j)$.

Proof: if $\Pi(S_i,\varphi_i) \neq \{\varepsilon\}$, then by completeness of the inference rules, for all $P \in \Pi(S_i,\varphi_i)$ there exists a path $(S_i;\varphi_i)\vdash_I\ldots\vdash_I(S';\varphi')$ such that $P >_p Q$ for some $Q \in \Pi(S',\varphi')$. By fairness of the search plan, there exists an $(S_j;\varphi_j)$, for some $j > i$, and an $R \in \Pi(S_j,\varphi_j)$ such that $P >_p Q \geq_p R$. □

Corollary 2.5.1 *If a completion procedure* $\mathcal{C}$ *on domain* $\mathcal{T}$ *has refutationally complete inference rules and fair search plan, then for all inputs* $(S_0;\varphi_0)$, *if* $\varphi_0 \in Th(S_0)$, *the derivation*

$$(S_0;\varphi_0)\vdash_{\mathcal{C}}(S_1;\varphi_1)\vdash_{\mathcal{C}}\ldots\vdash_{\mathcal{C}}(S_i;\varphi_i)\vdash_{\mathcal{C}}\ldots$$

reaches a stage k, $k \geq 0$, *such that* φ_k *is the clause true.*

Proof: let P be any proof in $\Pi(S_0,\varphi_0)$. By Theorem 2.5.1 and the well-foundedness of $>_p$ the derivation reaches a stage k such that P has been reduced to ε. Then $\Pi(S_k,\varphi_k) = \{\varepsilon\}$ and φ_k is the clause *true*. □

2.6 Redundancy

In Section 2.4 we introduced a notion of *redundancy*. The interest in redundancy of data elements in a theorem proving derivation resides in the importance of contraction inference rules. Although, contraction inference rules are necessary to make theorem proving feasible, few of them are known. The purpose of studying redundancy is to get some insight about how to design new and powerful contraction rules. A notion of redundant clauses appeared in [109] and in [16]. We show that redundant clauses according to these works are redundant in our sense. On the other

hand, there are clauses which are intuitively redundant and redundant according to our definition, but not according to the definitions in [109] and [16].

Definition 2.6.1 (Rusinowitch 1988) [109] *A clause φ is* R-redundant *in a set S if there exists a clause $\psi \in S$ such that ψ properly subsumes φ, i.e. $\varphi \succ\!\!\!\cdot\, \psi$. We also write $\varphi \in R_{\succ\!\!\!\cdot}(S)$.*

R-redundancy has been investigated in [112] in the context of proofs by resolution in first order logic. Very high numbers of R-redundant clauses may be generated in such derivations, resulting in waste of space to hold them and in waste of time to perform the subsumption test to detect them. Two techniques to limit the generation of R-redundant clauses are proposed in [112].

Definition 2.6.2 (Bachmair and Ganzinger 1990) [16] *A clause φ is* B-redundant *in a set S if there exists an ordering $>^d$ on clauses, which is monotonic, stable, well-founded and total on ground, such that the following holds: for all ground instances $\varphi\sigma$ of φ, there are ground instances $\psi_1 \ldots \psi_n$ of clauses in S such that $\{\psi_1 \ldots \psi_n\} \models \varphi\sigma$ and $\forall j, 1 \leq j \leq n, \varphi\sigma >^d \psi_j$. We also write $\varphi \in R_{>^d}(S)$.*

Lemma 2.6.1 (Bachmair and Ganzinger 1990) [16] *R-redundant clauses are B-redundant.*

Proof: we assume an ordering on literals $\succ$, which is well-founded and total on ground. We use its multiset extension $\succ_{mul}$ for clauses. Given such an ordering, it is possible to define the ordering $>^d$, in such a way that it is also well-founded, total on ground and incorporates the proper subsumption ordering $\succ\!\!\!\cdot$: for two ground instances of clauses $\varphi\sigma_1$ and $\psi\sigma_2$, $\varphi\sigma_1 >^d \psi\sigma_2$ holds if and only if $\varphi\sigma_1 \succ_{mul} \psi\sigma_2$ or else $\varphi\sigma_1 = \psi\sigma_2$ and $\varphi \succ\!\!\!\cdot\, \psi$. If φ is R-redundant in S, there exists a clause $\psi \in S$ such that $\varphi \succ\!\!\!\cdot\, \psi$, i.e. $\psi\sigma \subseteq \varphi$ for some substitution σ. For all ground instances $\varphi\tau$ of φ, we have $\psi\sigma\tau \subseteq \varphi\tau$. Then $\psi\sigma\tau \models \varphi\tau$ is trivially true. The condition $\psi\sigma\tau \subseteq \varphi\tau$

means either $\psi\sigma\tau \subset \varphi\tau$ or $\psi\sigma\tau = \varphi\tau$. In the first case $\varphi\tau >^d \psi\sigma\tau$ holds because $\varphi\tau \succ_{mul} \psi\sigma\tau$. In the second case $\varphi\tau >^d \psi\sigma\tau$ holds because $\varphi\tau = \psi\sigma\tau$ and $\varphi \succ\!\!\!\cdot\, \psi$. □

In our view, the intuition behind the notion of redundancy is that a clause φ is redundant in S if adding φ to S does not decrease any minimal proof in S (Definition 2.4.4). In fact our definition captures the meaning of Definition 2.6.2. We write $\varphi \in R_p(S, \psi)$, if φ is redundant in S on ψ, $\varphi \in R_p(S, \mathcal{T})$, if φ is redundant in S on $\mathcal{T}$. The subscript p recalls that redundancy is relative to the assumed proof ordering $>_p$:

Theorem 2.6.1 *If $\varphi \in R_{>^d}(S)$, then there exists a proof ordering $>_p$ such that $\varphi \in R_p(S, \mathcal{T})$, where $\mathcal{T}$ is the domain of all ground clauses.*

Proof: for all ground clauses ψ, we regard any set $\{\psi_1 \dots \psi_n\}$ of ground instances of clauses in S, such that $\{\psi_1 \dots \psi_n\} \models \psi$, as a proof in S of ψ. Since the ordering $>^d$ assumed in the definition of B-redundancy is well-founded and total on ground clauses, its multiset extension $>^d{}_{mul}$ is also well-defined and total on multisets of ground clauses. Let the proof ordering $>_p$ be $>^d{}_{mul}$. Since $>^d{}_{mul}$ is total, the minimal proof in S of a ground clause ψ is unique. By slightly abusing our notation, we use $\Pi(S, \psi)$ to denote the unique minimal proof of ψ in S. Let φ be B-redundant in S. We show that $\Pi(S \cup \{\varphi\}, \psi) = \Pi(S, \psi)$ for all ground theorems ψ. Since $S \subset S \cup \{\varphi\}$, $\Pi(S \cup \{\varphi\}, \psi) \leq_p \Pi(S, \psi)$ trivially holds and therefore we simply have to show that $\Pi(S \cup \{\varphi\}, \psi) \not<_p \Pi(S, \psi)$. The proof is done by way of contradiction: if $\Pi(S \cup \{\varphi\}, \psi) <_p \Pi(S, \psi)$, then the smallest set of ground instances of clauses in $S \cup \{\varphi\}$ which logically entails ψ has the form $S' \cup \{\varphi\sigma_1 \dots \varphi\sigma_k\}$ for some set S' of ground instances of clauses in S and some ground substitutions $\sigma_1 \dots \sigma_k$. Since φ is B-redundant in S, for all $\varphi\sigma_i$, $1 \leq i \leq k$, there are ground instances $\{\psi^i_1 \dots \psi^i_{n_i}\}$ of clauses in S such that $\{\psi^i_1 \dots \psi^i_{n_i}\} \models \varphi\sigma_i$ and $\varphi\sigma_i >^d \psi^i_j$, $\forall j, 1 \leq j \leq n_i$. Therefore, $S' \cup \{\psi^i_1 \dots \psi^i_{n_i}\}^k_{i=1} <^d{}_{mul} S' \cup \{\varphi\sigma_1 \dots \varphi\sigma_k\}$ and $S' \cup \{\psi^i_1 \dots \psi^i_n\}^k_{i=1} \models \psi$, that is $S' \cup \{\varphi\sigma_1 \dots \varphi\sigma_k\}$ cannot be the smallest set entailing ψ. It follows that

$\Pi(S \cup \{\varphi\}, \psi) = \Pi(S, \psi)$. $\square$

On the other hand, there are cases where trivially redundant clauses are not B-redundant, whereas they are redundant according to our definition:

Example 2.6.1 *If $S = \{P, \neg R, R\}$, where P and R are ground atoms, P is intuitively redundant and it is redundant according to our Definition 2.4.4: the minimal proof of every ground theorem is given by $\{\neg R, R\}$, since $\{\neg R, R\}$ yields the empty clause and therefore any clause. However, if $R \succ P$ and thus $R >^d P$, then P is not B-redundant.*

This example shows a limitation of a notion of redundancy based on an ordering on clauses: different precedences on predicate symbols may be needed in order to characterize as redundant different clauses during a computation.

In our definition of completion, we have required that a contraction step which simply deletes a sentence, delete a redundant sentence. It remains to clarify the relationship between redundancy and contraction steps that replace sentences by others. We show that for such steps the replaced sentence is redundant on the specific target, if the step is strictly proof-reducing, on the entire domain, otherwise:

Lemma 2.6.2 *If a contraction inference step $(S \cup \{\psi\}; \varphi) \vdash (S \cup \{\psi'\}; \varphi)$ is proof-reducing on $\mathcal{T}$ by Condition 1 in Definition 2.4.3, then $\psi \in R_p(S \cup \{\psi'\}, \varphi)$; if it is proof-reducing by Condition 2 in Definition 2.4.3, then $\psi \in R_p(S \cup \{\psi'\}, \mathcal{T})$.*

Proof: if ψ does not occur as an axiom in any proof $P \in \Pi(S \cup \{\psi\}, \varphi)$, then $\Pi(S \cup \{\psi\}, \varphi) = \Pi(S, \varphi)$, i.e. ψ is redundant on φ in S and therefore also in $S \cup \{\psi'\}$. If ψ is an axiom in some proof $P \in \Pi(S \cup \{\psi\}, \varphi)$, then $P \notin \Pi(S \cup \{\psi'\}, \varphi)$, since $\psi \notin S \cup \{\psi'\}$. By Condition 1 in Definition 2.4.3, there exists a $Q \in \Pi(S \cup \{\psi'\}, \varphi)$ such that $P >_p Q$. In other words, all the proofs of φ where ψ occurs as an axiom are replaced by smaller proofs in $\Pi(S \cup \{\psi'\}, \varphi)$. Adding ψ to $S \cup \{\psi'\}$ would not

reduce any proof in $\Pi(S \cup \{\psi'\}, \varphi)$ and therefore ψ is redundant on φ in $S \cup \{\psi'\}$. By applying the same argument to all theorems in the domain we obtain the second part of the lemma. □

This lemma demonstrates that the notions of redundancy on the target and redundancy on the domain correspond to the two ways a contraction step on the presentation can be proof-reducing: either it is strictly proof-reducing on the target and deletes a sentence which is redundant on the target, or it is proof-reducing on the domain and deletes a sentence which is redundant on the entire domain. It follows that all sentences deleted by contraction steps are redundant: if φ is deleted at stage i, either it is redundant at stage i, (it has been just deleted), or it is redundant at stage $i+1$ (it has been replaced). A similar relationship between deletion and B-redundancy holds according to the approach proposed in [16]. In fact, in [16] B-redundancy is used to define the deletion scheme itself: a deletion inference rule is a rule which deletes redundant sentences.

In [17], the approach to redundancy of [16] has been extended by providing a general definition of *redundancy criterion*, of which B-redundancy is an instance:

Definition 2.6.3 (Bachmair and Ganzinger 1992) [17] *A mapping R from sets of sentences to sets of sentences is called a* redundancy criterion *if*

1. $S - R(S) \models R(S)$ *(soundness of a redundancy criterion),*
2. *if* $S \subseteq S'$, *then* $R(S) \subseteq R(S')$ *(monotonicity with respect to the subset relation) and*
3. *if* $(S' - S) \subseteq R(S')$, *then* $R(S') \subseteq R(S)$ *(monotonicity with respect to the deletion of redundant sentences).*

The first two conditions express a *soundness* and *monotonicity* requirement respectively. The third condition says that the redundancy of a clause does not depend on

other redundant clauses, only on non-redundant ones. Intuitively, this means that if a clause is redundant at a certain stage of a derivation, it will also be redundant at the subsequent stages. Namely, if we derive S from S' by deleting elements in $R(S')$, all the elements in $R(S')$ are still redundant in S. For this reason we call it *monotonicity with respect to the deletion of redundant sentences.* In the following, we show that our definition of redundancy on the domain is a redundancy criterion in the context of derivations without target, where the inferences are monotonic in the sense of Definition 2.3.1. We give one lemma for each condition and then we summarize the results in a theorem. The first lemma simply says that the requirement of monotonicity of the derivation implies trivially the requirement of soundness of the redundancy criterion applied during the derivation:

Lemma 2.6.3 *For all derivations*

$$S_0 \vdash_{\mathcal{C}} S_1 \vdash_{\mathcal{C}} \ldots S_i \vdash_{\mathcal{C}} \ldots$$

by a completion procedure $\mathcal{C}$ on domain $\mathcal{T}$, which satisfies the monotonicity property of Definition 2.3.1, $\forall i, j,\ j > i \geq 0$, $\forall A \subseteq R_p(S_i, \mathcal{T}) \cap Th(S_i)$, if $S_j = S_i - A$, then $S_j \models A$.

Proof: by the monotonicity property of Definition 2.3.1, $\forall i \geq 0$, $Th(S_i) \subseteq Th(S_{i+1})$. It follows that $Th(S_i) \subseteq Th(S_j)$. Thus, $A \subseteq Th(S_i) \subseteq Th(S_j)$, i.e. $S_j \models A$. □

In particular, if $A = R_p(S_i, \mathcal{T})$, we have $S_i - R_p(S_i, \mathcal{T}) \models R_p(S_i, \mathcal{T})$, which is the soundness of a redundancy criterion. Definition 2.6.3 does not require that $R(S) \subseteq Th(S)$. However, when one is considering a derivation by a sound inference mechanism, where only elements of the theory may be generated, we are interested only in redundancy of elements of the theory, i.e. in $R(S) \cap Th(S)$. In summary, the requirement of monotonicity of the derivation plays in our framework the role which is played by soundness of the redundancy criterion in the framework of [17].

The second lemma shows that if a clause is redundant on the domain at a certain

stage of a derivation, it will be redundant at all subsequent stages:

Lemma 2.6.4 *For all derivations*

$$S_0 \vdash_{\mathcal{C}} S_1 \vdash_{\mathcal{C}} \ldots S_i \vdash_{\mathcal{C}} \ldots$$

by a completion procedure $\mathcal{C}$ *on domain* $\mathcal{T}$, $\forall i, j,\ j > i \geq 0$, $\forall \psi \in \mathcal{T} \cap Th(S_i) \cap Th(S_j)$, *if* $\psi \in R_p(S_i, \mathcal{T})$, *then* $\psi \in R_p(S_j, \mathcal{T})$.

Proof: if ψ is redundant in S_i on $\mathcal{T}$, ψ does not occur as an axiom in any minimal proof in S_i of theorems in $\mathcal{T}$. This also holds for ψ itself. In other words, there exists at least a proof $P \in \Pi(S_i, \psi)$, which is smaller than the proof represented by ψ itself: $\psi >_p P$. Since there is no target, the derivation is proof-reducing by Condition 2 in Definition 2.4.3. It follows that $\forall j,\ j > i$, either $P \in \Pi(S_j, \psi)$ or P is replaced by a proof $Q \in \Pi(S_j, \psi)$ such that $P >_p Q$. In both cases, there is a proof $R \in \Pi(S_j, \psi)$ such that $\psi >_p R$. By monotonicity and stability of $>_p$, $C[\psi\sigma] >_p C[R\sigma]$ for all proof contexts C and substitutions σ. In other words, ψ is not involved in any minimal proof in S_j of a theorem in $\mathcal{T}$, since any occurrence of ψ in a proof can be replaced by a proof of ψ smaller than ψ itself. It follows that ψ is redundant in S_j on $\mathcal{T}$. □

If S_j is derived from S_i by a sequence of inference steps, $Th(S_j) \subseteq Th(S_j)$ by soundness of the inferences. If, in addition, $S_i \subseteq S_j$, we have $Th(S_i) \subseteq Th(S_j)$, by monotonicity of logical entailment used to define Th. Thus, it is $Th(S_i) = Th(S_j)$. By the above lemma, we have $(R_p(S_i, \mathcal{T}) \cap Th(S_i)) \subseteq (R_p(S_j, \mathcal{T}) \cap Th(S_j))$. In other words, Lemma 2.6.4 shows that our definition of redundancy on the domain satisfies the second property of a redundancy criterion (if $S \subseteq S'$, then $R(S) \subseteq R(S')$) if S' is generated from S in a derivation. Also, similar to Lemma 2.6.3, the attention is restricted to the theory of interest during the derivation.

The third lemma proves that R_p satisfies the third property of a redundancy criterion, for all those clauses that are theorems of both sets involved:

Lemma 2.6.5 *If* $(S' - S) \subseteq R_p(S', \mathcal{T})$, $\forall \psi \in Th(S') \cap Th(S)$, *if* $\psi \in R_p(S', \mathcal{T})$, *then* $\psi \in R_p(S, \mathcal{T})$.

Proof: we show that if $\forall \varphi \in S' - S$, $\varphi \in R_p(S', \mathcal{T})$, then $\forall \psi \in Th(S') \cap Th(S) \cap R_p(S', \mathcal{T})$, it is also $\psi \in R_p(S, \mathcal{T})$. If $\psi \in R_p(S', \mathcal{T})$, it follows, by the same reasoning as in the previous proof, that there exists a $P \in \Pi(S', \psi)$, such that $\psi >_p P$. Since $\forall \varphi \in S' - S$, $\varphi \in R_p(S', \mathcal{T})$, no $\varphi \in S' - S$ appear in any proof in $\Pi(S', \psi)$. Thus, no $\varphi \in S' - S$ appear in P, i.e. P is also a proof in S. Furthermore, $P \in \Pi(S, \psi)$, i.e. P is minimal: if there is a $Q \in \Pi(S, \psi)$ such that $P >_p Q$, since Q is also a proof in S', $P \notin \Pi(S', \psi)$, which is a contradiction. Thus, $P \in \Pi(S, \psi)$ and since $\psi >_p P$, $\psi \in R_p(S, \mathcal{T})$. □

These three lemmas imply the following:

Theorem 2.6.2 *For all derivations*

$$S_0 \vdash_{\mathcal{C}} S_1 \vdash_{\mathcal{C}} \ldots S_i \vdash_{\mathcal{C}} \ldots$$

by a completion procedure $\mathcal{C}$ *on domain* $\mathcal{T}$, *which satisfies the monotonicity property of Definition 2.3.1,* $\forall i, j$, $j > i \geq 0$, *the following properties hold:*

- $\forall A \subseteq R_p(S_i, \mathcal{T}) \cap Th(S_0)$, *if* $S_j = S_i - A$, *then* $S_j \models A$,
- $(R_p(S_i, \mathcal{T}) \cap Th(S_0)) \subseteq (R_p(S_j, \mathcal{T}) \cap Th(S_0))$ *and*
- *if* $(S_i - S_j) \subseteq R_p(S_i, \mathcal{T})$, *then* $(R_p(S_i, \mathcal{T}) \cap Th(S_0)) \subseteq (R_p(S_j, \mathcal{T}) \cap Th(S_0))$.

Proof: it follows from the three lemmas, recalling that by soundness and monotonicity of the derivation, $\forall i > 0$, it is $Th(S_i) = Th(S_0)$. □

This theorem shows that our notion of redundancy on the domain is basically compatible with the approach to redundancy of [16, 17]. The main difference is that we

have both a notion of redundancy on the domain and a notion of redundancy on the target. We have two notions because we distinguish between two different types of derivations, with two different purposes. For derivations without target, e.g. derivations which generate a confluent system, we have soundness, monotonicity, i.e. the entire theory is preserved, proof-reduction on the domain and redundancy on the domain. For such derivations our approach is similar to that in [16, 17], as showed by Theorem 2.6.2.

For derivations with target, e.g. theorem proving derivations, our approach is different from that in [16, 17]. The study of redundancy in [16, 17], through the notion of B-redundancy and the general definition of redundancy criterion, captures redundancy on the domain, not redundancy on the target. Redundancy on the domain allows to delete only those sentences that are not necessary to reduce any minimal proof of any theorem in the domain. On the other hand, our approach for derivations with target is consistently target-oriented. We require soundness and relevance, but not monotonicity; we allow proof-reduction and redundancy on the target, in addition to proof-reduction and redundancy on the domain. By relevance, a derivation does not have to preserve all the theorems, as long as the target is preserved. By proof reduction and redundancy on the target, contraction steps may replace sentences which are not redundant on the whole domain, provided they are redundant on the specific target. In other words, contraction steps may increase the complexity of some proofs, as long as they reduce a proof of the target. In this way our definitions allow in principle very strong contraction inference rules.

So far, we have differentiated derivations for theorem proving and derivations for generating confluent systems at the level of the inference mechanism: relevance, monotonicity, proof reduction and redundancy are properties of the inference mechanism. In the next section, we shall see that the dichotomy between the two types of derivations applies also to the search plan level.

2.7 Uniform fairness and saturated sets

Our definition of fairness (Definition 2.5.2) is sufficient for theorem proving as shown by Theorem 2.5.1. If it is applied to Knuth-Bendix completion, though, it is not sufficient to guarantee that a confluent rewrite system is eventually generated, since it does not guarantee that all critical pairs are eventually considered. This requires a much stronger fairness property, which we call *uniform fairness*.

The first definition of uniform fairness appeared in [67], where it is required that the search plan sorts the rewrite rules in the data set by a well-founded ordering, in order to ensure that no rule is indefinitely postponed. We recall this very first notion of (uniform) fairness, because it states explicitly that fairness is a property of the search plan. The later definitions of (uniform) fairness [7, 14, 16, 109] are given at a much higher abstraction level, which may prevent the reader from seeing that fairness is a property of the search mechanism.

In this section and in the next one we show how uniform fairness and the classical results on traditional completion are incorporated in our framework. Therefore, we are going to consider derivations without target. For such derivations we assume that the *monotonicity* property (see Definition 2.3.1) is added to our definition of completion. Given a derivation by completion starting from a presentation S_0, the *limit* S_∞ of the derivation is the possibly infinite set $\bigcup_{j\geq 0}\bigcap_{i\geq j} S_i$ of all the *persistent* sentences, that is the sentences which are generated at some stage and never deleted afterwards [67, 9]. The advantage of dealing with the limit S_∞ is that if the derivation halts at some stage k, $S_\infty = S_k$. Therefore, properties stated in terms of the limit S_∞ apply uniformly to both halting and non-halting derivations. We denote by $I_e(S)$ the set of clauses which can be generated in one step by the expansion rules of the completion procedure applied to S. Also, we denote by R the redundancy criterion associated to $\mathcal{C}$, in the sense that whenever a contraction rule of I deletes or replaces a clause ψ in S, ψ is in $R(S)$.

Definition 2.7.1 (Bachmair and Ganzinger 1992) [17]

A derivation $S_0 \vdash_{\mathcal{C}} S_1 \vdash_{\mathcal{C}} \ldots S_i \vdash_{\mathcal{C}} \ldots$ *is* uniformly fair *if* $I_e(S_\infty - R(S_\infty)) \subseteq \bigcup_{j \geq 0} S_j$.

In our case, the redundancy criterion is R_p on the domain $\mathcal{T}$ of the completion procedure $\mathcal{C}$. This definition of fairness generalizes previous definitions given in [7, 9, 14, 16, 67, 109]. It says that every clause φ that can be derived by an expansion rule from persistent non-redundant parents is generated eventually. For instance, a Knuth-Bendix derivation such that all critical pairs from persistent non-redundant equations are eventually generated is uniformly fair.

Uniform fairness has been studied and progressively refined in [7, 14, 16, 17, 109] with the purpose of solving the problem of the interaction between expansion inference rules and contraction inference rules. The intuitive meaning of uniform fairness is to be fair to the inference rules, that is to apply all the inference rules to all the data. However, this is impossible in the presence of contraction rules: if a clause φ is deleted by a contraction step before an expansion rule f is applied to φ, the derivation is not fair to f. The problem has been then to define fairness in such a way that the application of contraction rules is fair. This problem is solved in the definition of uniform fairness by establishing that it is fair not to perform an expansion inference step if its premises are not persistent or redundant. Remark that persistent elements may be redundant. In this way it is fair to apply contraction inference rules such as simplification and subsumption. In actuality, a uniformly fair procedure will perform exhaustively all expansion steps which are not inhibited by contraction steps.

Fairness and uniform fairness are different in several basic aspects. Fairness is *target-oriented*, whereas uniform fairness is defined for a derivation without a target. Indeed Definition 2.7.1 is not a definition of fairness for theorem proving. In [109, 16], Definition 2.7.1 is applied to refutational theorem proving, where S_0 contains the negation of the target. In this case the only persisting clause is the

empty clause $\Box$ and $I_e(\bigcup_{i\geq 0}\bigcap_{j\geq i} S_j) = \{\Box\}$. Then Definition 2.7.1 says that the limit of the derivation is the empty clause. A notion of fairness given in terms of the limit does not represent useful information for the design of search plans, because it does not say anything about how a search plan should choose the successor at any given stage of the derivation.

Uniform fairness emphasizes expansion rules and the role of contraction inference rules is buried in the restriction to persistent non-redundant clauses. Fairness does not differentiate between expansion rules and contraction rules nor between persisting and non-persisting clauses. The reason is that in theorem proving the idea of fairness is not to be fair to the inference rules, but to the target. Therefore the interaction of expansion and contraction rules is no longer an issue. All inference rules are treated uniformly by considering their effect with respect to the goal of reducing the proof of the target.

The following example illustrates a set of conditions for an Unfailing Knuth-Bendix derivation which have been proved in [9] to be sufficient for uniform fairness. These conditions represent the most well-known definition of (uniform) fairness for a completion procedure:

Example 2.7.1 *A derivation* $E_0 \vdash_{UKB} E_1 \vdash_{UKB} \ldots \vdash_{UKB} E_i \vdash_{UKB} \ldots$ *is uniformly fair if*

- *for all critical pairs* $g \simeq d \in I_e(E_\infty)$, $g \simeq d \in \bigcup_{i\geq 0} E_i$ *and*
- E_∞ *is reduced.*

The first condition says that all critical pairs derivable from persisting equations are eventually generated. The second condition says that all persisting equations are eventually simplified as much as possible. As was remarked above, the application of contraction rules is allowed but not required: the first condition alone is sufficient for uniform fairness. Since at any stage of the computation it is not known which

equations are going to persist and which equations are going to be simplified, the above conditions for uniform fairness prescribe in practice to apply exhaustively all the inference rules of Unfailing Knuth-Bendix completion until none applies.

The concept of uniform fairness leads to the following notion of *saturated* presentation:

Definition 2.7.2 (Bachmair and Ganzinger 1992) [17] *A presentation S is* saturated *on domain $\mathcal{T}$ if and only if $I_e(S - R(S)) \subseteq S \cup R(S)$.*

In other words, a presentation is saturated if no non-trivial consequences can be added. This definition covers previous definitions in [16] and in [86], where the word "saturated" was first used. In the equational case, as remarked in [86], a set of equations is saturated if no divergent critical pairs can be deduced, or equivalently, the set is *locally confluent* [41]. As in the definition of uniform fairness, the application of contraction inference rules is allowed but not required: contraction inference rules may still be applicable to a saturated set. For instance, a locally confluent equational presentation is not necessarily reduced.

If a derivation is uniformly fair, S_∞ is saturated. Since the notions of uniform fairness and saturated set are defined in terms of a redundancy criterion and our redundancy criterion is different than those in [109, 16, 17], we give a new proof of this result for our redundancy criterion $R_p(S, \mathcal{T})$. First we recall the following:

Corollary 2.7.1 *For all derivations $S_0 \vdash_{\mathcal{C}} S_1 \vdash_{\mathcal{C}} \ldots S_i \vdash_{\mathcal{C}} \ldots$, by a completion procedure $\mathcal{C}$ on domain $\mathcal{T}$, $\forall \psi \in Th(S_0)$, $\forall i \geq 0$, if $\psi \in R_p(S_i, \mathcal{T})$, then $\psi \in R_p(S_\infty, \mathcal{T})$.*

Proof: it follows from Lemma 2.6.4, keeping in mind that $\forall i \geq 0$, $Th(S_i) = Th(S_0)$ by monotonicity. □

Theorem 2.7.1 (Kounalis and Rusinowitch 1988) [86], (Bachmair and Ganzinger

1990, 1992) [16, 17] *If a derivation* $S_0 \vdash_{\mathcal{C}} S_1 \vdash_{\mathcal{C}} \ldots S_i \vdash_{\mathcal{C}} \ldots$ *is* uniformly fair*, then* S_∞ *is* saturated*.*

Proof: we show that for all $\varphi \in I_e(S_\infty - R_p(S_\infty, \mathcal{T}))$, either $\varphi \in S_\infty$ or $\varphi \in R_p(S_\infty, \mathcal{T})$. By uniform fairness of the derivation, there exists an S_j, for some $j \geq 0$, such that $\varphi \in S_j$. Then, either φ is not deleted afterwards, that is $\varphi \in S_\infty$, or φ is deleted at some stage $i > j$. If φ is simply deleted, $\varphi \in R_p(S_i, \mathcal{T})$ by Definition 2.4.7 of completion. If φ is replaced by another sentence, $\varphi \in R_p(S_{i+1}, \mathcal{T})$by Lemma 2.6.2. In both cases, $\varphi \in R_p(S_\infty, \mathcal{T})$ by Corollary 2.7.1. □

This theorem generalizes the following classical results:

Theorem 2.7.2 (Knuth and Bendix 1970) [84], (Huet 1981) [67], (Bachmair, Dershowitz and Hsiang 1986) [7] *If a derivation* $E_0 \vdash_{KB} E_1 \vdash_{KB} \ldots E_i \vdash_{KB} \ldots$ *by the Knuth-Bendix completion procedure does not fail and is* uniformly fair *on the domain* $\mathcal{T}$ *of all equations, then* E_∞ *is a* confluent *term rewriting system.*

Knuth-Bendix completion fails if an unoriented equation persists. If a derivation by Knuth-Bendix completion does not fail, all the persistent equations are oriented into rewrite rules according to a reduction ordering and therefore E_∞ is a terminating rewrite system. By Theorem 2.7.1, E_∞ is saturated, i.e. locally confluent. By Newman's lemma [41], a terminating rewrite system is confluent if and only if it is locally confluent. Therefore E_∞ is confluent.

Theorem 2.7.3 (Hsiang and Rusinowitch 1987) [62], (Bachmair, Dershowitz and Plaisted 1989) [13] *If a derivation* $E_0 \vdash_{UKB} E_1 \vdash_{UKB} \ldots E_i \vdash_{UKB} \ldots$ *by the Unfailing Knuth-Bendix completion procedure is* uniformly fair *on the domain* $\mathcal{T}$ *of all ground equations, then* E_∞ *is a* ground confluent *set of equations.*

For this second result we recall that given a set of equations E, $s \rightarrow_E t$ if $s \leftrightarrow_E t$ and $s \succ t$ for a reduction ordering $\succ$. The Unfailing Knuth-Bendix procedure

assumes that $\succ$ is a complete simplification ordering. Since a complete simplification ordering is total on ground terms, $\leftrightarrow_{E_\infty} = \rightarrow_{E_\infty} \cup \leftarrow_{E_\infty}$ holds for ground terms and E_∞ is Church-Rosser on ground terms if and only if it is ground confluent. Since a complete simplification ordering is well-founded, E_∞ is terminating on ground terms. The domain $\mathcal{T}$ of Unfailing Knuth-Bendix is the set of ground equations. By Theorem 2.7.1, E_∞ is saturated on $\mathcal{T}$, i.e. it is locally confluent on ground terms. By Newman's lemma E_∞ is ground confluent and therefore Church-Rosser on ground terms. The Church-Rosser property on ground terms is important because $E \models \forall \bar{x} s \simeq t$ if and only if $\hat{s} \leftrightarrow_E^* \hat{t}$. If E is Church-Rosser on ground terms, $\hat{s} \leftrightarrow_E^* \hat{t}$ if and only if $\hat{s} \rightarrow_E^* \circ \leftarrow_E^* \hat{t}$ and therefore $E \models \forall \bar{x} s \simeq t$ can be decided by well-founded reduction by E. This introduces us to the topic of the next section.

2.8 Decision procedures

In this section we study the properties of derivations in a finite, saturated presentation. We shall show that, under appropriate hypotheses, a saturated presentation is a *decision procedure* for its theory. First, we define under which conditions a presentation is a decision procedure:

Definition 2.8.1 *Let $\mathcal{C}$ be a complete completion procedure on domain $\mathcal{T}$. A presentation S of a theory is a* decision procedure *for $Th(S) \cap \mathcal{T}$, if S is finite and for all $\varphi_0 \in \mathcal{T}$, the derivation*

$$(S; \varphi_0) \vdash_{\mathcal{C}} (S; \varphi_1) \vdash_{\mathcal{C}} \ldots \vdash_{\mathcal{C}} (S; \varphi_i) \vdash_{\mathcal{C}} \ldots,$$

is guaranteed to halt at a stage k, for some $k > 0$, such that $\varphi_k = true$ if $\varphi_0 \in Th(S)$ and $\varphi_k \neq true$ if $\varphi_0 \notin Th(S)$.

We regard the presentation S as an algorithm, which, if interpreted by the procedure $\mathcal{C}$, decides the validity of sentences in $\mathcal{T}$ in the theory of S. Clearly, once termination

of the derivation is ensured, the correctness of the result is a consequence of the completeness of the completion procedure. Therefore, the key property of a decision procedure is that all derivations are guaranteed to halt regardless of the truth of the given target. Sufficient conditions for the termination of derivations can be given, if it is possible to exclude the application of expansion inference rules. This is exactly where the assumption of having a saturated presentation plays a role:

Lemma 2.8.1 *If a presentation S is* saturated *on $\mathcal{T}$, then no expansion inference rule which is proof-reducing on $\mathcal{T}$ applies to S.*

Proof: if a proof-reducing expansion inference rule derives $S \cup \{\varphi\}$ from S, then, there is a $\psi \in \mathcal{T}$ such that $P >_p Q$ for some $P \in \Pi(S, \psi)$ and $Q \in \Pi(S \cup \{\varphi\}, \psi)$, i.e. $\Pi(S, \psi) \neq \Pi(S \cup \{\varphi\}, \psi)$. Since S is saturated on $\mathcal{T}$, either $\varphi \in S$ or $\varphi \in R_p(S, \mathcal{T})$. Thus, $\forall \psi \in \mathcal{T}$, $\Pi(S, \psi) = \Pi(S \cup \{\varphi\}, \psi)$. □

In other words, if the presentation is saturated, all derivations are made only of target inference steps and contraction steps on the presentation. Termination conditions for these kinds of inferences can be given by using the well-founded orderings $\succ$ that we have assumed throughout our work:

Definition 2.8.2 *A target inference step $(S; \varphi) \vdash (S; \varphi')$ is* target-reducing *if $\varphi \succ \varphi'$. A contraction inference step $(S; \varphi) \vdash (S'; \varphi)$ is* data-reducing *if either it deletes a redundant sentence or it replaces a sentence ψ by a sentence ψ' such that $\psi \succ \psi'$.*

Similar to proof-reduction, an inference rule is target-reducing (data-reducing) if all the steps where it is applied are target-reducing (data-reducing). For instance, the Simplification inference rule is target-reducing (data-reducing if applied to the presentation: see Examples 2.3.1, 2.3.3 and 2.4.1).

Lemma 2.8.2 *A derivation*

$$(S_0;\varphi_0)\vdash_{\mathcal{C}}(S_1;\varphi_1)\vdash_{\mathcal{C}}\ldots\vdash_{\mathcal{C}}(S_i;\varphi_i)\vdash_{\mathcal{C}}\ldots$$

where every step is either target-reducing or data-reducing is guaranteed to halt.

Proof: let $>_r$ be the lexicographic combination of the multiset extension $\succ_{mul}$ and of $\succ$ itself. By the definition of target-reducing and data-reducing steps, we have that $\forall i \geq 0$, $(S_i;\varphi_i) >_r (S_{i+1};\varphi_{i+1})$. Since the ordering $>_r$ is well-founded, the derivation is guaranteed to halt. □

We can now prove that a saturated set is a decision procedure:

Theorem 2.8.1 *Let $\mathcal{C}$ be a complete completion procedure on domain $\mathcal{T}$, such that all its target inference rules are target-reducing and all its contraction inference rules on the presentation are data-reducing. Then a presentation S which is* saturated *on $\mathcal{T}$ is a* decision procedure *for $\mathcal{T}\cap Th(S)$.*

Proof: by Lemma 2.8.1, for all $\varphi_0 \in \mathcal{T}$ the derivation from $(S;\varphi_0)$ may contain only target inference steps and contraction steps on the presentation. By the hypotheses on the inference rules and Lemma 2.8.2, such a derivation is guaranteed to halt at some stage k. Either $\varphi_k = true$ or $\varphi_k \neq true$. By completeness of $\mathcal{C}$, φ_k is $true$ if and only if $\varphi_0 \in Th(S)$. Therefore, S is a decision procedure for $Th(S)\cap\mathcal{T}$. □

If we also assume that no contraction rule applies to the presentation, then all derivations from S are made only of target-reducing inference steps. In equational logic this corresponds to assume that S is not just confluent, but also *reduced*, i.e. *canonical* [41]. Derivations made only of target inference steps are traditionally called *linear* [28]. Therefore, a saturated set such that no contraction rule applies to the presentation yields only linear and terminating derivations.

Finally, we can characterize a completion procedure as a *generator of decision procedures*:

Theorem 2.8.2 *Let $\mathcal{C} =< I; \Sigma >$ be a completion procedure on domain $\mathcal{T}$ such that*

- *the procedure satisfies the monotonicity property,*
- *I is refutationally complete,*
- *Σ is uniformly fair and*
- *all the target inference rules are target-reducing and all the contraction inference rules on the presentation are data-reducing.*

For all presentations S_0, if the limit S_∞ of the derivation

$$S_0 \vdash_{\mathcal{C}} S_1 \vdash_{\mathcal{C}} \ldots \vdash_{\mathcal{C}} S_i \vdash_{\mathcal{C}} \ldots$$

is finite, then S_∞ is a decision procedure for $\mathcal{T} \cap Th(S_0)$.

Proof: by Theorem 2.7.1, S_∞ is saturated on $\mathcal{T}$. S_∞ is a decision procedure for $\mathcal{T} \cap Th(S_\infty)$ by Theorem 2.8.1. By monotonicity, $Th(S_\infty) = Th(S_0)$ and therefore S_∞ is a decision procedure for $\mathcal{T} \cap Th(S_0)$. □

It follows that uniform fairness "implies" fairness, in the following sense: since the limit S_∞ of a uniformly fair derivation is a decision procedure, for all $\varphi_0 \in Th(S_0)$, if φ_0 is given as target and if the search plan is also fair in applying the elements of S_∞ to the target, the target will be eventually proved. This theorem generalizes the classical results for equational logic and their extensions to Horn logic with equality. In equational logic, the (Unfailing) Knuth-Bendix completion procedure generates a (ground) confluent presentation that can be used to decide the validity of theorems in the form $\forall \bar{x} s \simeq t$ by well-founded simplification. Extensions to Horn logic with equality have been studied in [86, 16, 45]. Given a complete completion procedure for Horn logic with equality, such as those in [86, 16, 45], the issue is how to guarantee that derivations in a saturated and reduced presentation are target-reducing and therefore terminating. Targets have the form $B_1 \wedge \ldots \wedge B_m$, where each

B_i is a ground positive literal. In [86] and [16] the problem is solved by imposing special restrictions on the clauses in the saturated set:

Definition 2.8.3 (Kounalis and Rusinowitch 1988) [86] *A Horn clause* $A :- B_1 \ldots B_n$ *is* ground-preserving *if the following two conditions hold:*

- *all variables occurring in a negated literal also occur in* A *and*
- *if* A *is an equation* $s \simeq t$*, either it can be oriented into a rewrite rule or* s *and* t *have the same set of variables.*

The conditions given in [16] are slightly different, but the purpose is basically the same: the ground-preserving condition is designed to ensure that whenever an inference step is applied between a clause in the saturated set and a ground target, the newly generated target is ground as well. The resolution and paramodulation inference rules in [86] and [16] are *ordered*, i.e. they are restricted by a given complete simplification ordering on terms and literals in such a way that at each step a ground literal in the target is replaced by a set of smaller ground literals. Therefore ordered resolution and ordered paramodulation steps between a ground-preserving clause and a ground target are target-reducing. A saturated presentation containing only ground-preserving clauses is then a decision procedure by Lemma 2.8.1, Lemma 2.8.2 and Theorem 2.8.1:

Theorem 2.8.3 (Kounalis and Rusinowitch 1988) [86], (Bachmair and Ganzinger 1990) [16] *Let* S *be a presentation in Horn logic with equality such that* S *is saturated on the domain of ground clauses and all clauses in* S *are ground-preserving. Then* S *is a decision procedure for targets in the form* $B_1 \wedge \ldots \wedge B_m$*, where each* B_i *is a ground positive literal.*

The requirement that all clauses are ground-preserving is quite strong. For instance the Horn clause $T(x, y) \vee \neg R(x, z) \vee \neg T(z, y)$ in the definition of the transitive

closure T of a relation R is not ground-preserving, because of the variable z. The restriction to ground-preserving clauses is resemblant of the restriction to oriented equations for Knuth-Bendix completion. This restriction is lifted in the Unfailing Knuth-Bendix procedure by assuming a complete simplification ordering on terms and by designing a simplification rule that applies oriented instances of equations as simplifiers. In this way, a "static" requirement on the presentation, that it contains only oriented equations, is replaced by a "dynamic" property of the inferences, that use only oriented instances of equations. A similar result has been obtained for Horn logic with equality in [45] by assuming a complete simplification ordering $\succ$ on terms and literals. There is no restriction on the clauses in the presentation, but the target inference rules are designed in such a way that only *decreasing* instances of clauses are applied:

Definition 2.8.4 (Dershowitz 1991) [45] *A Horn clause* $l \simeq r :- p_1 \simeq q_1 \ldots p_n \simeq q_n$ *is* decreasing *if for all ground substitutions* σ, $l\sigma \succ r\sigma$, $l\sigma \succ p_i\sigma$, $l\sigma \succ q_i\sigma$, $1 \geq i \geq n$.

Whenever a decreasing instance of a clause in the saturated presentation is applied to a ground target, the newly generated ground target is smaller, i.e. the derivation is target-reducing:

Theorem 2.8.4 (Dershowitz 1991) [45] *A presentation saturated on the domain of ground clauses in Horn logic with equality is a decision procedure for targets in the form* $B_1 \wedge \ldots \wedge B_m$, *where each* B_i *is a ground positive literal.*

The practical importance of the interpretation of completion procedures as generators of decision procedures is strongly limited by the observation that very few theories have a finite saturated presentation. This consideration applies in equational logic and the problem is even more serious in Horn logic with equality. For instance, the *Maximal Unit Strategy* of [44] is a complete method for Horn logic with equality, with unit resolution, unit paramodulation and several contraction inference rules.

The name derives from the restriction that a unit clause resolves/paramodulates into a negative literal which is maximal among all the negative literals in the clause. The method is strongly oriented toward forward reasoning, since it basically works by inferring facts from the given facts and implications. Therefore, the saturated set is infinite in most cases, since it contains all the true facts in the theory:

Theorem 2.8.5 *If S_∞ is the saturated limit of a derivation by the Maximal Unit Strategy, then every non-unit clause is redundant in S_∞ on the domain of the ground clauses.*

Proof: the proof is done by way of contradiction. We assume that there are a ground target $B_1 \wedge \ldots \wedge B_m$ and a minimal proof $P \in \Pi(S_\infty, B_1 \wedge \ldots \wedge B_m)$ where non-unit clauses occur as axioms. Without loss of generality, we can assume that $m = 1$ and that a non-unit clause $A :- C_1 \ldots C_n$ is the first clause applied in P, i.e. $B_1 = A\sigma$ for some ground substitution σ. Let $A :- C_1 \ldots C_n$ be the shortest clause that can be applied to B_1. This step generates the subgoal $C_1\sigma \ldots C_n\sigma$. Let $C_1\sigma$ be maximal in $C_1\sigma \ldots C_n\sigma$. By completeness of the unit strategy, there exists a unit clause $G \in S_\infty$ such that $G\rho = C_1\sigma\rho$ for some substitution ρ. The literals G and C_1 have a common instance and therefore there is an mgu ρ' such that $G\rho' = C_1\rho'$. Thus, the clause $A\rho' :- C_2\rho' \ldots C_n\rho'$ can be generated in one unit resolution step from G and $A :- C_1 \ldots C_n$. Since S_∞ has been saturated by the Maximal Unit Strategy, $A\rho' :- C_2\rho' \ldots C_n\rho'$ is in S_∞. Furthermore, there exists a substitution τ such that $A\rho'\tau = B_1$, since $\rho' \preceq \sigma\rho$. It follows that $A\rho' :- C_2\rho' \ldots C_n\rho'$ can be applied at the place of $A :- C_1 \ldots C_n$ in P, contradicting the hypothesis that $A :- C_1 \ldots C_n$ is the shortest applicable clause. □

The Maximal Unit Strategy represents a rather extreme case. It remains that most theories have infinite saturated presentations under most completion procedures. Therefore, the interpretation of completion procedures as semidecision procedures is the most useful one in practice.

2.9 Simplification-based strategies for equational reasoning

In this section we give a new presentation of some Knuth-Bendix type completion procedures for equational logic, in the framework developed so far.

2.9.1 Unfailing Knuth-Bendix completion

The *Unfailing Knuth-Bendix procedure* [62, 13] is a semidecision procedure for equational theories. A derivation by UKB has the form

$$(E_0; \hat{s}_0 \simeq \hat{t}_0) \vdash_{UKB} (E_1; \hat{s}_1 \simeq \hat{t}_1) \vdash_{UKB} \ldots (E_i; \hat{s}_i \simeq \hat{t}_i) \vdash_{UKB} \ldots$$

and it succeeds at stage k if $\hat{s}_k$ and $\hat{t}_k$ are identical. At each step of the completion process the pair $(E_{i+1}; \hat{s}_{i+1} \simeq \hat{t}_{i+1})$ is derived from the pair $(E_i; \hat{s}_i \simeq \hat{t}_i)$ by applying one of the following inference rules:

- *Presentation inference rules*:
  - *Simplification*:
    $$\frac{(E \cup \{p \simeq q, l \simeq r\}; \hat{s} \simeq \hat{t})}{(E \cup \{p[r\sigma]_u \simeq q, l \simeq r\}; \hat{s} \simeq \hat{t})} \quad \begin{array}{l} p|u = l\sigma \qquad\qquad p \succ p[r\sigma]_u \\ p \triangleright l \vee q \succ p[r\sigma]_u \end{array}$$
  - *Superposition*:
    $$\frac{(E \cup \{p \simeq q, l \simeq r\}; \hat{s} \simeq \hat{t})}{(E \cup \{p \simeq q, l \simeq r, p[r]_u\sigma \simeq q\sigma\}; \hat{s} \simeq \hat{t})} \quad \begin{array}{l} p|u \notin X \qquad\qquad (p|u)\sigma = l\sigma \\ p\sigma \not\preceq q\sigma, p[r]_u\sigma \end{array}$$
  - *Deletion*:
    $$\frac{(E \cup \{l \simeq l\}; \hat{s} \simeq \hat{t})}{(E; \hat{s} \simeq \hat{t})}$$
  - *Functional subsumption*:
    $$\frac{(E \cup \{p \simeq q, l \simeq r\}; \hat{s} \simeq \hat{t})}{(E \cup \{l \simeq r\}; \hat{s} \simeq \hat{t})} \; (p \simeq q) \triangleright (l \simeq r)$$
- *Target inference rules*:

– *Simplification*:

$$\frac{(E \cup \{l \simeq r\}; \hat{s} \simeq \hat{t})}{(E \cup \{l \simeq r\}; \hat{s}[r\sigma]_u \simeq \hat{t})} \quad \begin{array}{l} \hat{s}|u = l\sigma \\ \hat{s} \succ \hat{s}[r\sigma]_u \end{array}$$

– *Deletion*:

$$\frac{(E; \hat{s} \simeq \hat{s}\})}{(E; true)}$$

We have already presented some of these inference rules in Examples 2.3.1, 2.3.2, 2.3.3, 2.4.1, 2.4.2 and 2.4.3. The original definitions of Simplification and Superposition given in [62] have slightly different conditions. Simplification requires that $l\sigma \succ r\sigma$ and superposition requires that $p\sigma \not\preceq q\sigma$ and $l\sigma \not\preceq r\sigma$. We adopt here the conditions given in [44], because they put weaker requirements on simplification and stronger requirements on superposition than the original ones. However, these conditions may be more expensive to check, since they require to perform both substitution application and term replacement.

Simplification is the most important among the above inference rules, because it reduces dramatically the number and the size of the generated equations. If Simplification is not applied, the Superposition inference rule rapidly saturates the memory space with equations, making impossible to reach a proof in reasonable time. Thus, a search plan for UKB should be a *Simplification-first* plan (Example 2.3.4). We characterize the UKB procedure as a completion procedure by using the ordering $>_u$ introduced in Example 2.2.2:

Lemma 2.9.1 *The presentation inference rules of the UKB procedure are reducing.*

Proof: we show that Superposition and Simplification are proof-reducing, Deletion and Functional subsumption delete redundant equations:

- the proof for Superposition was given in Example 2.4.2.

- A Simplification step where an equation $p \simeq q$ is simplified to $p[r\sigma]_u \simeq q$ by an equation $l \simeq r$, affects a minimal proof by replacing a step $s \leftrightarrow_{p \simeq q} t$ by two steps $s \rightarrow_{l \simeq r} v \leftrightarrow_{p[r\sigma]_u \simeq q} t$.
  - If $t \succ s$, we have $\{(t, q, s)\} >^e_{mul} \{(s, l, v), (t, q, v)\}$ since $t \succ s$ and $s \succ v$.
  - If $s \succ t$,
    - if $p \triangleright l$, we have
      - if $t \succ v$, $\{(s, p, t)\} >^e_{mul} \{(s, l, v), (t, q, v)\}$ since $p \triangleright l$ and $s \succ t$,
      - if $v \succ t$, $\{(s, p, t)\} >^e_{mul} \{(s, l, v), (v, q, t)\}$ since $p \triangleright l$ and $s \succ v$;
    - if $p \stackrel{\bullet}{=} l$ and $q \succ p[r\sigma]_u$, $t \succ v$ follows from $q \succ p[r\sigma]_u$ by stability and monotonicity of $\succ$ and we have $\{(s, p, t)\} >^e_{mul} \{(s, l, v), (t, q, v)\}$ since $t \succ v$ and $s \succ t$.
- A trivial equation $l \simeq l$ is redundant: no minimal proof contains a step $s \leftrightarrow_{l \simeq l} s$ since the subproof given by the single term s is smaller: $\{(s, l, s)\} >^e_{mul} \{\epsilon\}$, where the empty triple ϵ is the proof complexity of s.
- The proof for Functional subsumption was given right after Example 2.4.3. □

Lemma 2.9.2 *The target inference rules of the UKB procedure are strictly proof-reducing.*

Proof: the proof for Simplification was given in Example 2.4.1. For a Deletion step it is $\{\hat{s}, \hat{s}\} \succ_{mul} \{true\}$, since *true* is smaller than any term. □

We can then show that UKB is a completion procedure:

Theorem 2.9.1 *The Unfailing Knuth-Bendix procedure is a completion procedure on the domain $\mathcal{T}$ of all ground equalities.*

Proof: for all equational presentations E_0 and for all ground targets $\hat{s}_0 \simeq \hat{t}_0$ the derivation

$$(E_0; \hat{s}_0 \simeq \hat{t}_0) \vdash_{UKB} (E_1; \hat{s}_1 \simeq \hat{t}_1) \vdash_{UKB} \ldots (E_i; \hat{s}_i \simeq \hat{t}_i) \vdash_{UKB} \ldots$$

has the soundness, relevance and reduction properties. Soundness and relevance were proved among others in [67, 7, 9]. Reduction follows from Lemma 2.9.1 and Lemma 2.9.2. □

If a *fair* search plan is provided, the UKB procedure is a semidecision procedure for equational theories:

Theorem 2.9.2 (Hsiang and Rusinowitch 1987) [62], (Bachmair, Dershowitz and Plaisted 1989) [13] *An equation* $\forall \bar{x} s \simeq t$ *is a theorem of an equational theory* E *if and only if the Unfailing Knuth-Bendix procedure derives* true *from* $(E; \hat{s} \simeq \hat{t})$.

2.9.2 Extensions: AC-UKB and cancellation laws

Many equational problems involve associative and commutative (AC) operators. An AC function f satisfies the equations

$f(f(x, y), z) \simeq f(x, f(y, z))$ *(associativity)* and

$f(x, y) \simeq f(y, x)$ *(commutativity).*

Handling associativity and commutativity as any other equation turns out to be very inefficient, since commutativity may generate a very high number of equations through the Superposition inference rule. Also, many instances of commutativity may not be ordered by the chosen simplification ordering, so that simplification does not apply as often as it is desirable to reduce the size and the number of the equations.

The efficiency of the UKB strategy can be greatly improved if associativity and commutativity are not given in the input, but built in the inference rules. The UKB

procedure with associativity and commutativity built-in is called *AC-UKB* [2]. The basic idea is to replace syntactic identity by equality *modulo AC*. If AC denotes a set of associativity and commutativity axioms, two terms s and t are *equal modulo* AC, if $s \simeq t$ is a theorem of AC, which we write $s =_{AC} t$. The inference rules of the UKB procedure are modified in such a way that any two terms which are equal modulo AC are regarded as identical.

The first modification is to require that the complete simplification ordering on terms $\succ$ is in some sense "compatible" with replacing identity by equality modulo AC. More precisely, this "compatibility" requirement is a *commutation* property. Given two relations R and S, we say that R *commutes* over S if $S \circ R \subseteq R \circ S$, where $\circ$ is composition of relations. The complete simplification ordering $\succ$ is required to commute over $=_{AC}$: this means that for any two terms s and t, if there is a third term r such that $s =_{AC} r$ and $r \succ t$, there is also a term r' such that $s \succ r'$ and $r' =_{AC} t$. Secondly, matching and unification are replaced by AC-matching and AC-unification. A term s matches a term t *modulo* AC if there is a substitution σ such that $s\sigma =_{AC} t$. Similarly, two terms s and t unify *modulo* AC if there is a substitution σ such that $s\sigma =_{AC} t\sigma$. Finally, the strict encompassment ordering $\triangleright$ is replaced by the ordering $\triangleright_{AC}$, that is $s \triangleright_{AC} t$ if and only if $s \triangleright r$ and $r =_{AC} t$ for some term r.

The set of inference rules of the UKB procedure is therefore modified as follows:

- *Presentation inference rules*:
  - *Simplification*:

$$\frac{(E \cup \{p \simeq q, l \simeq r\}; \hat{s} \simeq \hat{t})}{(E \cup \{p[r\sigma]_u \simeq q, l \simeq r\}; \hat{s} \simeq \hat{t})} \quad \begin{array}{l} p|u =_{AC} l\sigma \\ p \triangleright_{AC} l \vee q \succ p[r\sigma]_u \end{array} \qquad p \succ p[r\sigma]_u$$

  - *Superposition*:

$$\frac{(E \cup \{p \simeq q, l \simeq r\}; \hat{s} \simeq \hat{t})}{(E \cup \{p \simeq q, l \simeq r, p[r]_u\sigma \simeq q\sigma\}; \hat{s} \simeq \hat{t})} \quad \begin{array}{l} p|u \notin X \\ p\sigma \not\preceq q\sigma, p[r]_u\sigma \end{array} \qquad (p|u)\sigma =_{AC} l\sigma$$

  - *Extension*:

$$\frac{(E \cup \{f(p,q) \simeq r\}; \hat{s} \simeq \hat{t})}{(E \cup \{f(p,q) \simeq r, f(p,q,z) \simeq f(r,z)\}; \hat{s} \simeq \hat{t})} \quad \begin{array}{l} f \text{ is } AC \\ f(p,q) \not\preceq r \end{array}$$

- *Deletion*:
  $$\frac{(E \cup \{l \simeq l\}; \hat{s} \simeq \hat{t})}{(E; \hat{s} \simeq \hat{t})}$$
- *Functional subsumption*:
  $$\frac{(E \cup \{p \simeq q, l \simeq r\}; \hat{s} \simeq \hat{t})}{(E \cup \{l \simeq r\}; \hat{s} \simeq \hat{t})} \; (p \simeq q) \triangleright_{AC} (l \simeq r)$$

- *Target inference rules*:
  - *Simplification*:
    $$\frac{(E \cup \{l \simeq r\}; \hat{s} \simeq \hat{t})}{(E \cup \{l \simeq r\}; \hat{s}[r\sigma]_u \simeq \hat{t})} \quad \begin{array}{l} \hat{s}|u =_{AC} l\sigma \\ \hat{s} \succ \hat{s}[r\sigma]_u \end{array}$$
  - *Deletion*:
    $$\frac{(E; \hat{s} \simeq \hat{s})}{(E; true)}$$

This set of inference rules is obtained from the set of inference rules of the UKB procedure by replacing identity by equality modulo AC as explained above and by adding a new inference rule, called *Extension*. The *Extension* inference rule is a specialized version of the *Superposition* inference rule, designed to compute superpositions of equations in E onto associativity axioms. Namely, if $f(p,q) \simeq r$ is an equation in E, f is AC and $f(p,q) \not\preceq r$, the equation $f(p,q) \simeq r$ trivially superposes onto the associativity axiom $f(f(x,y),z) \simeq f(x,f(y,z))$, yielding the critical pair $f(p,f(q,z)) \simeq f(r,z)$, which we write in *flattened* form as $f(p,q,z) \simeq f(r,z)$. These critical pairs are called *extended rules*. Computing the extended rules is sufficient to ensure completeness of the AC-UKB procedure: no other critical pairs between E and AC need to be computed [105].

The extension of UKB to AC-UKB is feasible because algorithms for AC-matching and AC-unification are available. An algorithm for AC-unification, its application in a completion procedure and the extended rules first appeared in [114, 105]. The correctness of the AC-unification algorithm was proved in [46]. General theoretical frameworks for working with equations modulo a set of axioms A

are given in [70] and in [12]. These results are surveyed in [41] and more specifically for unification problems in [72].

The UKB or AC-UKB procedure can be further improved by adding inference rules for the *cancellation laws*. A function F is *right cancellable* if it satisfies the *right cancellation law*

$$\forall x, y, z \quad f(x, y) = f(z, y) \supset x = z$$

The *left cancellation law* is defined symmetrically. Cancellation laws may reduce considerably the size of the equations. They are implemented as inference rules as follows [63]:

Cancellation 1:

$$\frac{(E \cup \{f(p, u) \simeq f(q, v)\}; \hat{s} \simeq \hat{t})}{(E \cup \{f(p, u) \simeq f(q, v), p\sigma \simeq q\sigma\}; \hat{s} \simeq \hat{t})} \; u\sigma = v\sigma$$

Cancellation 2:

$$\frac{(E \cup \{f(d_1, d_2) \simeq y\}; \hat{s} \simeq \hat{t})}{(E \cup \{f(d_1, d_2) \simeq y, d_1\sigma \simeq x\}; \hat{s} \simeq \hat{t})} \quad \begin{array}{ll} y \in V(d_1) & \sigma = \{y \mapsto f(x, d_2)\} \\ y \notin V(d_2) & x \textit{ is a new variable} \end{array}$$

Cancellation 3:

$$\frac{(E \cup \{f(p_1, q_1) \simeq r_1, f(p_2, q_2) \simeq r_2\}; \hat{s} \simeq \hat{t})}{(E \cup \{f(p_1, q_1) \simeq r_1, f(p_2, q_2) \simeq r_2, p_1\sigma \simeq p_2\sigma\}; \hat{s} \simeq \hat{t})} \quad \begin{array}{l} q_1\sigma = q_2\sigma \\ r_1\sigma = r_2\sigma \end{array}$$

Cancellation 4:

$$\frac{(E \cup \{f(p, u) \simeq f(q, u)\}; \hat{s} \simeq \hat{t})}{(E \cup \{p \simeq q\}; \hat{s} \simeq \hat{t})}$$

where the function f is right cancellable. In *Cancellation 2*, if the substitution $\sigma = \{y \mapsto f(x, d_2)\}$ is applied to the given equation, it becomes $f(d_1\sigma, d_2) \simeq f(x, d_2)$,

since y does not occur in d_2. The cancellation law reduces this equation to $d_1\sigma \simeq x$.

Cancellation 4 is not necessary for the purpose of completeness, since the same effect can be obtained by a step of *Cancellation 1* with empty mgu followed by a step of Functional subsumption. It is added to improve the efficiency.

In order to prove that the UKB procedure with the cancellation inference rules is a completion procedure, we need to prove that the Cancellation inference rules are proof-reducing. We adopt as proof ordering a slight modification of $>_u$, which we call $>_{uc}$: a ground equational step $s \simeq t$ justified by an equation $l \simeq r$ has complexity measure $(s, l\sigma, l, t)$, if s is $c[l\sigma]$, t is $c[r\sigma]$ and $s \succ t$. Complexity measures are compared by the lexicographic combination $>^{ec}$ of the orderings $\succ$, $\trianglerighteq$, $\trianglerighteq$ and $\succ$. Proofs are compared by the lexicographic combination $>_{uc}$ of the multiset extansions $\succ_{mul}$ and $>^{ec}{}_{mul}$. The proof of Lemma 2.9.1 is unaffected if $>_{UKBC}$ replaces $>_{UKB}$.

Lemma 2.9.3 *The* Cancellation *inference rules are proof-reducing.*

Proof: we assume that $(E_i; \hat{s}_i \simeq \hat{t}_i) \vdash_{UKB} (E_{i+1}; \hat{s}_i \simeq \hat{t}_i)$ is a Cancellation step:

- an application of the rule Cancellation 1 to an equation $f(p,u) \simeq f(q,v)$ affects any minimal proof in E_i which contains a step $s \leftrightarrow t$ such that $s = c[f(p,u)\tau]$, $t = c[f(q,v)\tau]$ and $\tau \trianglerighteq \sigma$, where $\trianglerighteq$ is the subsumption ordering and σ is the mgu such that $u\sigma = v\sigma$ of the application of Cancellation 1. The step $s \leftrightarrow_{f(p,u)\simeq f(q,v)} t$ has complexity $(s, f(p,u)\tau, f(p,u), t)$, if $s \succ t$. In the minimal proofs in E_{i+1} the step $s \leftrightarrow_{f(p,u)\simeq f(q,v)} t$ is replaced by a step $s \leftrightarrow_{p\sigma\simeq q\sigma} t$ justified by the new equation $p\sigma \simeq q\sigma$ generated by the application of Cancellation 1. The step $s \leftrightarrow_{p\sigma\simeq q\sigma} t$ has complexity $(s, p\tau, p\sigma, t)$. Since $f(p,u)\tau \trianglerighteq p\tau$, $(s, f(p,u)\tau, f(p,u), t) >^{ec} (s, p\tau, p\sigma, t)$ follows. A symmetric argument applies if $t \succ s$.
- An application of the rule Cancellation 2 to an equation $f(d_1, d_2) \simeq y$ affects any minimal proof in E_i which contains a step $s \leftrightarrow t$ such that $s = c[f(d_1, d_2)\tau]$,

$t = c[y\tau]$ and $\tau \succeq \sigma$, where σ is $\{y \mapsto f(x, d_2)\}$. Since $y \in V(d_1)$, we have $f(d_1, d_2)\tau \succ y\tau$ by the subterm property and therefore $s \succ t$ by monotonicity, so that the step $s \leftrightarrow t$ has complexity $(s, f(d_1, d_2)\tau, f(d_1, d_2), t)$. In the minimal proofs in E_{i+1} the step $s \leftrightarrow t$ is replaced by a step $s \leftrightarrow_{d_1\sigma \simeq x} t$ justified by the new equation $d_1\sigma \simeq x$ generated by the application of Cancellation 2. The step $s \leftrightarrow_{d_1\sigma \simeq x} t$ has complexity $(s, d_1\tau, d_1\sigma, t)$. Since $f(d_1, d_2)\tau \bullet\!\!\rhd d_1\tau$, $(s, f(d_1, d_2)\tau, f(d_1, d_2), t) >^{ec} (s, d_1\tau, d_1\sigma, t)$ follows.

- An application of the rule Cancellation 3 to two equations $f(p_1, q_1) \simeq r_1$ and $f(p_2, q_2) \simeq r_2$ affects any minimal proof in E_i which contains a subproof $s \leftrightarrow u \leftrightarrow t$ such that $s = c[f(p_1, q_1)\tau]$, $u = c[r_1\tau]$, $t = c[f(p_2, q_2)\tau]$ and $\tau \succeq \sigma$, where σ is the mgu such that $q_1\sigma = q_2\sigma$ and $r_1\sigma = r_2\sigma$ of the application of Cancellation 3. It follows that $q_1\tau = q_2\tau$ and $r_1\tau = r_2\tau$. The subproof $s \leftrightarrow u \leftrightarrow t$ is replaced in any minimal proof in E_{i+1} by a single step $s \leftrightarrow_{p_1\sigma \simeq p_2\sigma} t$ justified by the new equation $p_1\sigma \simeq p_2\sigma$ generated by the application of Cancellation 3.

  1. If $s \succ t \succ u$, the subproof $s \leftrightarrow u \leftrightarrow t$ has complexity

     $$\{(s, f(p_1, q_1)\tau, f(p_1, q_1), u), (t, f(p_2, q_2)\tau, f(p_2, q_2), u)\}$$

     and the step $s \leftrightarrow_{p_1\sigma \simeq p_2\sigma} t$ has complexity $(s, p_1\tau, p_1\sigma, t)$. Since $f(p_1, q_1)\tau \bullet\!\!\rhd p_1\tau$, the result follows. A symmetric argument applies if $t \succ s \succ u$.

  2. If $s \succ u \succ t$, the subproof $s \leftrightarrow u \leftrightarrow t$ has complexity

     $$\{(s, f(p_1, q_1)\tau, f(p_1, q_1), u), (u, r_1\tau, r_1, t)\}$$

     and the step $s \leftrightarrow_{p_1\sigma \simeq p_2\sigma} t$ has complexity $(s, p_1\tau, p_1\sigma, t)$. Since $f(p_1, q_1)\tau \bullet\!\!\rhd p_1\tau$, the result follows. A symmetric argument applies if $t \succ u \succ s$.

  3. If $u \succ s \succ t$, the subproof $s \leftrightarrow u \leftrightarrow t$ has complexity

     $$\{(u, r_1\tau, r_1, s), (u, r_1\tau, r_1, t)\}$$

     and the step $s \leftrightarrow_{p_1\sigma \simeq p_2\sigma} t$ has complexity $(s, p_1\tau, p_1\sigma, t)$. Since $u \succ s$, the result trivially follows. A symmetric argument applies if $u \succ t \succ s$. □

Completeness of the inference rules for cancellation is proved in [63]. Most of the experimental results reported in [2, 3, 20, 4, 6] are obtained by AC-UKB with the inference rules for cancellation.

2.9.3 The Inequality Ordered-Saturation strategy

The UKB procedure is complete, but it is not very efficient in general. The main source of inefficiency is the Superposition inference rule, that is the forward reasoning component of UKB. This problem has been considered first for the application of Knuth-Bendix completion to the generation of confluent systems. Generating all the critical pairs is a sufficient but not necessary condition for the limit of a derivation to be confluent. A *critical pair criterion* is a criterion to establish that certain critical pairs are not necessary to obtain a confluent system. Such criteria are studied in [11]. We rather analyze the problem from the theorem proving point of view. In the theorem proving context criteria such as those in [11] are still useful, but certainly not satisfactory, since they are not target-oriented.

All the backward reasoning steps are Simplification steps, which are strictly proof-reducing. On the other hand, a Superposition step is guaranteed to reduce the proof of some theorem, but not necessarily the proof of the target. The UKB procedure is inefficient because it computes many critical pairs which do not help in proving the target. Therefore, our goal is to reduce the number of critical pairs generated or equivalently to perform less forward reasoning and more backward reasoning.

For the forward reasoning part, a possible approach to the problem consists in designing search plans which generate first the critical pairs that are estimated to be likely to reduce the proof of the target. Such search plans are based on *heuristical criteria* that measure how useful a critical pair is expected to be with respect to the task of simplifying the goal. Some examples of these heuristics are given in [4, 5].

For the backward reasoning part, we observe that if the target $\hat{s}_i \simeq \hat{t}_i$ is fully simplified with respect to E_i, $\hat{s}_i \simeq \hat{t}_i$ is minimal in the ordering $\succ_{mul}$ among all the ground equations E-equivalent to the input target $s_0 \simeq t_0$, where $E = \bigcup_{0 \leq j \leq i} E_j$. If a Simplification-first plan is adopted, UKB maintains a minimal target. Therefore, it could seem that no improvement can be obtained on the target side. However, this is not the case. The notion of minimal target is relative to the assumed partially ordered set (poset) of targets. If we assume the poset of ground equalities ordered by $\succ_{mul}$, $\hat{s}_i \simeq \hat{t}_i$ is minimal among the ground equations E-equivalent to the input target $s_0 \simeq t_0$. The situation changes if we assume as poset of targets the poset of disjunctions of ground equalities ordered by an ordering $\succ'_{mul}$ defined as follows: $N_1 \succ'_{mul} N_2$ if $min(N_1) \succ_{mul} min(N_2)$, where N_1 and N_2 are disjunctions of ground equalities and $min(N)$ is the smallest equality in N according to $\succ_{mul}$. Since the equalities are ground and the simplification ordering is assumed to be total on ground terms, there is a smallest element in a disjunction and this ordering is well defined. The poset of equalities is embedded in the poset of disjunctions by regarding an equality as a one-element disjunction.

We show why the backward reasoning part of UKB is not guaranteed to compute a minimal target if the poset of disjunctions is assumed. Let $(E_i; \hat{s}_i \simeq \hat{t}_i)$ be the current stage in an UKB derivation and $l \simeq r$ be an un-orientable equation in E_i, such that $\hat{s}_i|u = l\sigma$ for some position u and substitution σ, but $\hat{s}_i \prec \hat{s}_i[r\sigma]_u$. In other words, l matches a subterm of $\hat{s}_i$ but Simplification does not apply because $\hat{s}_i$ would not be replaced by a smaller term. However, we assume that the target $\hat{s}_i[r\sigma]_u \simeq \hat{t}_i$ is generated nonetheless and that by simplification it reduces to an equation which is smaller than $\hat{s}_i \simeq \hat{t}_i$, that is $\hat{s}_i[r\sigma]_u \rightarrow^*_{E_i} \hat{s}'$, $\hat{t}_i \rightarrow^*_{E_i} \hat{t}'$ and $\{\hat{s}', \hat{t}'\} \prec_{mul} \{\hat{s}_i, \hat{t}_i\}$. If these conditions hold, we have that the disjunction $\hat{s}_i \simeq \hat{t}_i \vee \hat{s}' \simeq \hat{t}'$ is smaller than the disjunction given by $\hat{s}_i \simeq \hat{t}_i$ alone in the poset of disjunctions defined above. Therefore, if we assume the poset of disjunctions as posets of targets, it is not true that UKB maintains a minimal target.

The intuition behind the choice of considering disjunctions of equalities rather

than equalities is that if we consider more than one target equality, we have a greater chance to find a short proof. In order to work on disjunctions of equalities, we need to add to the UKB procedure an expansion inference rule, so that the target is eventually expanded into a disjunction of ground equalities. Such an expansion inference rule must satisfy the relevance requirement, so that proving the validity of any of the equalities in the disjunction is equivalent to prove the input target $s_0 \simeq t_0$. Also, the application of such rule must be restricted, in order to avoid the generation of a high number of target equalities, which may slow down the search for a solution. This new inference rule is superposition of an un-orientable equation onto a target equality $\hat{s} \simeq \hat{t}$ to generate a new target equality. A newly generated target equality is first simplified as much as possible and then it is kept only if it is not greater than any already existing target:

Ordered saturation:

$$\frac{(E \cup \{l \simeq r\}; N \cup \{\hat{s} \simeq \hat{t}\})}{(E \cup \{l \simeq r\}; N \cup \{\hat{s} \simeq \hat{t}, \hat{s}' \simeq \hat{t}'\})} \quad \begin{array}{ll} \hat{s}|u = l\sigma & \hat{s}[r\sigma]_u \to_E^* \hat{s}',\ \hat{t} \to_E^* \hat{t}' \\ \{\hat{s}', \hat{t}'\} \not\succeq_{mul} \{\hat{g}, \hat{d}\}, & \forall \hat{g} \simeq \hat{d} \in N \cup \{\hat{s} \simeq \hat{t}\} \end{array}$$

Ordered saturation applies if $\hat{s} \prec \hat{s}[r\sigma]_u$, since if $\hat{s} \succ \hat{s}[r\sigma]_u$ holds, simplification would apply. If $\succ$ is total on ground terms, the condition $\{\hat{s}', \hat{t}'\} \not\succeq_{mul} \{\hat{g}, \hat{d}\}$, $\forall \hat{g} \simeq \hat{d} \in N \cup \{\hat{s} \simeq \hat{t}\}$ becomes $\{\hat{s}', \hat{t}'\} \prec_{mul} \{\hat{g}, \hat{d}\}$, $\forall \hat{g} \simeq \hat{d} \in N \cup \{\hat{s} \simeq \hat{t}\}$. We have given the inference rule for the more general case: in fact, the ordering is not assumed to be total in [4], where a version of this inference rule first appeared. The target equality $\hat{s}' \simeq \hat{t}'$ might have a shorter proof than the other target equalities. We do not know which one has the shortest proof. We keep all of them to broaden our chance of reaching the proof as soon as possible.

In addition, we need to modify the *Deletion* inference rule, since the computation halts successfully as soon as an equality in the disjunction is reduced to a trivial equality:

Deletion:

$$\frac{(E; N \cup \{\hat{s} \simeq \hat{s}\})}{(E; true)}$$

The procedure obtained by adding Ordered saturation to UKB and by modifying Deletion as above, is called the *Inequality Ordered-Saturation strategy* (IOS) [4]. A derivation by the IOS strategy has the form

$$(E_0; N_0) \vdash_{IOS} (E_1; N_1) \vdash_{IOS} \ldots \vdash_{IOS} (E_i; N_i) \vdash_{IOS} \ldots$$

where the set N_0 contains the initial goal $\hat{s}_0 \simeq \hat{t}_0$ and at stage i, N_i is the current set of target equalities. The derivation succeeds at stage k if N_k contains a target $\hat{s}_i \simeq \hat{t}_i$ such that $\hat{s}_i$ and $\hat{t}_i$ are identical and the clause in N_k reduces to *true*.

In order to show that the IOS strategy is a completion procedure, we assume that the ordering $\succ$ is total on ground terms, coherently with the treatment of the other completion procedures for equational logic. Then we order proofs as follows: the proof of a disjunction N is represented by the proof of the smallest equality in N, i.e. $min(N)$, and proofs of equalities are ordered by $>_u$.

Lemma 2.9.4 *The* Ordered saturation *inference rule under a total ordering on terms is strictly proof-reducing.*

Proof: if $(E_i; N_i) \vdash_{IOS} (E_i; N_{i+1})$ is an Ordered saturation step, then $N_i \subset N_{i+1}$ and therefore $min(N_i) \succeq_{mul} min(N_{i+1})$. Since the ordering $\succ$ is total on ground terms, we have $min(N_i) \succ_{mul} min(N_{i+1})$ and Ordered saturation is strictly proof-reducing. □

Theorem 2.9.3 *The Inequality Ordered-Saturation strategy is a completion procedure.*

Proof: it follows from Theorem 2.9.1 and Lemma 2.9.4. □

The IOS strategy has been implemented and observed to perform better than the

UKB procedure [4]. In practice, few target equalities are kept, so that the overhead of handling them is negligible with respect to the advantage of keeping more than one target.

2.9.4 The S-strategy

The *S-strategy* [62] is an extension of the UKB procedure to the logic of equality and inequality. A presentation is a set of equations E_0 and a theorem φ is a sentence $\bar{Q}\bar{x}\ s_0 \simeq t_0 \vee \ldots \vee s_n \simeq t_n$, where $\bar{Q}\bar{x}$ is any sequence of quantifier-variable pairs. A theorem φ in this form is transformed into a target $N_0 = s_0 \simeq t_0 \vee \ldots \vee s_n \simeq t_n$, where all variables are implicitly existentially quantified, by replacing all the universally quantified variables by constants and by dropping the quantifiers. If φ is $\forall\bar{x} s_0 \simeq t_0$, N_0 is $\hat{s}_0 \simeq \hat{t}_0$ and the S-strategy reduces to the UKB procedure. A computation has the form

$$(E_0; N_0) \vdash_S (E_1; N_1) \vdash_S \ldots \vdash_S (E_i; N_i) \vdash_S \ldots$$

where $\forall i \geq 0$, E_i is a set of equalities and N_i is a disjunction of target equalities with existentially quantified variables. A derivation succeeds at stage k if N_k contains a target $s_i \simeq t_i$ whose sides are unifiable. The set of inference rules of UKB is modified as follows:

- *Presentation inference rules*:
  - *Simplification*:
    $$\frac{(E \cup \{p \simeq q, l \simeq r\}; N)}{(E \cup \{p[r\sigma]_u \simeq q, l \simeq r\}; N)} \quad \begin{array}{l} p|u = l\sigma \qquad p \succ p[r\sigma]_u \\ p \blacktriangleright l \vee q \succ p[r\sigma]_u \end{array}$$
  - *Deduction*:
    $$\frac{(E \cup \{p \simeq q, l \simeq r\}; N)}{(E \cup \{p \simeq q, l \simeq r, p[r]_u\sigma \simeq q\sigma\}; N)} \quad \begin{array}{l} p|u \notin X \qquad (p|u)\sigma = l\sigma \\ p\sigma \not\trianglelefteq q\sigma, p[r]_u\sigma \end{array}$$
  - *Deletion*:

$$\frac{(E \cup \{l \simeq l\}; N)}{(E; N)}$$

– *Functional subsumption*:

$$\frac{(E \cup \{p \simeq q, l \simeq r\}; N)}{(E \cup \{l \simeq r\}; N)} \; (p \simeq q) \triangleright (l \simeq r)$$

- *Target inference rules*:

  – *Simplification*:

  $$\frac{(E \cup \{l \simeq r\}; N \cup \{s \simeq t\})}{(E \cup \{l \simeq r\}; N \cup \{s[r\sigma]_u \simeq t\})} \quad \begin{array}{l} s|u = l\sigma \\ s \succ s[r\sigma]_u \end{array}$$

  – *Deduction*:

  $$\frac{(E \cup \{l \simeq r\}; N \cup \{s \simeq t\})}{(E \cup \{l \simeq r\}; N \cup \{s \simeq t, s[r]_u\sigma \simeq t\sigma\})} \quad \begin{array}{l} s|u \notin X \qquad (s|u)\sigma = l\sigma \\ s\sigma \not\preceq s[r]_u\sigma \end{array}$$

  – *Deletion*:

  $$\frac{(E; N \cup \{s \simeq t\})}{(E; true)} \; s\sigma = t\sigma$$

The *Deduction* inference rule applies to both equalities and inequalities. In the second case no ordering based condition applies to the inequality. The *Deletion* rule for the target is modified because the target contains variables: a contradiction is detected when the two sides of a target equality unify.

In order to measure the complexity of proofs of disjunctions, we observe the following: a target N is a theorem of E if and only if $E \cup \neg N$ is unsatisfiable, where N is a disjunction of equations $s_0 \simeq t_0 \vee \ldots \vee s_n \simeq t_n$ with existentially quantified variables and therefore $\neg N$ is a conjunction of inequalities $s_0 \neq t_0 \wedge \ldots \wedge s_n \neq t_n$ with universally quantified variables. By the Herbrand Theorem [28], the set $E \cup \neg N$ is unsatisfiable if and only if there is a finite set of ground instances of clauses in $E \cup \neg N$ which is unsatisfiable. Since $\neg N$ is a set of inequalities with universally quantified variables, an unsatisfiable set of ground instances of clauses in $E \cup \neg N$ needs to contain just one ground inequality: let $\hat{E} \cup \{\hat{s} \neq \hat{t}\}$ be the smallest such set with respect to the ordering $\succ_{mul}$. Since $\succ$ is total on ground terms, there exists a smallest set. Then the minimal proof of N in E is represented by the minimal ground equational proof of $\hat{s} \simeq \hat{t}$ in $\hat{E}$. Ground equational proofs are ordered by

the ordering $>_u$. This approach is correct if to every inference step on $(\hat{E}_i; \hat{s}_i \simeq \hat{t}_i)$ corresponds an inference steps on $(E_i; N_i)$. This is proved by the *Paramodulation Lifting Lemma* for S-strategy. We recall that a ground substitution is E-irreducible if it does contain any pair $\{x \mapsto t\}$ such that t can be simplified by an equation in E:

Lemma 2.9.5 (Peterson 1983) [106], (Hsiang and Rusinowitch 1987) [64] *If σ is a ground, E-irreducible substitution, then for all inference rules f of S-strategy, if $(E\sigma; s\sigma \simeq t\sigma) \vdash_f (E'; s' \simeq t')$, then $(E; s \simeq t) \vdash_f (E''; s'' \simeq t'')$, where E' and $s' \simeq t'$ are ground instances of E'' and $s'' \simeq t''$ respectively.*

Since $\hat{E} \cup \{\hat{s} \simeq \hat{t}\}$ is the smallest unsatisfiable set, $\hat{E} \subseteq E\sigma$ and $\hat{s} \simeq \hat{t} \in N\sigma$ for an E-irreducible substitution σ. Therefore, Lemma 2.9.5 applies and to every inference step on $(\hat{E}; \hat{s} \simeq \hat{t})$ corresponds an inference step on $(E; N)$. We can finally state the following theorem:

Theorem 2.9.4 *The S-strategy is a completion procedure on the domain $\mathcal{T}$ of all ground equalities.*

Proof: soundness and relevance were proved in [62]. By the above discussion on the complexity of proofs of disjunctions, an inference step on $(E; N)$ is proof-reducing if it is proof-reducing on the minimal proof of $\hat{s} \simeq \hat{t}$ in $\hat{E}$. Thus, the inference rules of S-strategy are proof-reducing if they are proof-reducing on ground equational proofs with respect to the ordering $>_u$. This follows from Lemma 2.9.1 and Lemma 2.9.2, since Deduction on the target is just Simplification if the target is ground. □

If a *fair* search plan is provided, the S-strategy is is a semidecision procedure for theories in the logic of equality and inequality:

Theorem 2.9.5 (Hsiang and Rusinowitch 1987) [62] *A sentence $\bar{Q}\bar{x}\ s_0 \simeq t_0 \vee \ldots \vee s_n \simeq t_n$ is a theorem of an equational theory E if and only if the S-strategy derives*

true *from* $(E; s_0 \simeq t_0 \vee \ldots \vee s_n \simeq t_n)$.

2.10 Semidecision procedures for disproving inductive theorems

The Knuth-Bendix completion procedure has also been applied to *disprove inductive theorem* in equational theories. This method has been called *inductionless induction*, *proof by consistency* or *proof by the lack of inconsistency* by several authors [101, 50, 68, 89, 48, 77, 76, 10, 71]. Extensions of this method to Horn logic with equality are explored in [86].

We show that a completion procedure applied to disproving inductive theorems is a *semidecision procedure*. We denote by $G(S)$ the set of all ground terms on the signature of a presentation S and we use $Ran(\sigma)$ to represent the range of a substitution σ, so that a ground substitution is a substitution such that $Ran(\sigma) \subset G(S)$. A clause φ is an *inductive theorem* of S, written $S \models_{Ind} \varphi$, if and only if for all ground substitutions σ, $\varphi\sigma \in Th(S)$. We denote by $Ind(S)$ the set of all the inductive theorems of S, $Ind(S) = \{\varphi |\ S \models_{Ind} \varphi\}$, by $GTh(S)$ the set of all the ground theorems of S, $GTh(S) = \{\varphi |\ \varphi \in Th(S), \varphi\ ground\}$ and by $G(\varphi)$ the set of all the ground instances of φ, $G(\varphi) = \{\varphi\sigma |\ Ran(\sigma) \subset G(S)\}$.

The set $Ind(S)$ is not semidecidable. Even if we have a decision procedure for $G(\varphi) \cap GTh(S)$, we still cannot prove that φ is an inductive theorem, because the set $G(\varphi)$ is infinite. However, the complement problem, that is proving that φ is *not* an inductive theorem of S, is semidecidable in certain theories. If $\varphi \notin Ind(S)$, then there is a ground instance $\varphi\sigma$ such that $\varphi\sigma \notin GTh(S)$. Therefore $GTh(S \cup \{\varphi\}) \neq GTh(S)$, since $\varphi\sigma \in GTh(S \cup \{\varphi\})$ for all ground instances $\varphi\sigma$. Thus, we can prove that φ is not an inductive theorem of S by proving the following target:

$$\Phi_0 = \exists\sigma\ Ran(\sigma) \subset G(S)\ \exists\psi \in S \cup \{\varphi\}\ such\ that\ \psi\sigma \in GTh(S \cup \{\varphi\}) - GTh(S).$$

If there exists an oracle $\mathcal{O}$ to decide such target, a completion procedure $\mathcal{C} =< I; \Sigma; \mathcal{O} >$ equipped with the oracle $\mathcal{O}$ will be a semidecision procedure for disproving inductive theorems. A derivation has the form

$$(S \cup \{\varphi\}; \Phi_0) \vdash_{\mathcal{C}} (S_1; \Phi_1) \vdash_{\mathcal{C}} \ldots (S_i; \Phi_i) \vdash_{\mathcal{C}} \ldots,$$

where at each step the target is

$$\Phi_i = \exists \sigma \; Ran(\sigma) \subset G(S) \; \exists \psi \in S_i \; such \; that \; \psi\sigma \in GTh(S_i) - GTh(S).$$

No inference step applies to the target: the procedure takes as input the presentation $S \cup \{\varphi\}$ given by the original presentation and the inductive conjecture and it proceeds by applying inference rules to the presentation until it obtains a presentation S_k such that the oracle applied to S_k answers positively and replaces Φ_k by *true*. In the equational case, the target is

$$\Phi_i = \exists \sigma \; \exists s_i \simeq t_i \in E_i \; such \; that \; \sigma \; is \; ground \; and \; (s_i \simeq t_i)\sigma \in GTh(E_i) - GTh(E).$$

Oracles to decide Φ_i are known if the input set of equations E is ground confluent. Under this hypothesis, $(s_i \simeq t_i)\sigma \in GTh(E_i) - GTh(E)$ if and only if there are E-irreducible terms s and t such that $s_i\sigma \rightarrow_E^* s$, $t_i\sigma \rightarrow_E^* t$ and $s \simeq t \in GTh(E_i)$. Therefore, we can restrict our attention to ground E-irreducible terms.

A first oracle was given in [68] for equational presentations satisfying the *principle of definition*. The signature of E is given by the disjoint union of a set C of *constructors* and a set D of *defined symbols*. The set $T(C)$ of all ground constructor terms is *free* and all function symbols in D are *completely defined* on C, that is for all ground term $t \in T(F)$, there exists a unique ground constructor term $t' \in T(C)$ such that $t \leftrightarrow_E^* t'$. If a presentation E satisfies the principle of definition, the ground E-irreducible terms are the ground terms made only of constructor symbols. Therefore, Φ_i is true if and only if there are two ground constructor terms t_1 and t_2 such that $t_1 \leftrightarrow_{E_i}^* t_2$. The following inference rules implement this test [68]:

- *Disproof 1*:
$$\frac{(E \cup \{f(t_1 \ldots t_n) \simeq g(s_1 \ldots s_n)\}; \Phi)}{(E \cup \{f(t_1 \ldots t_n) \simeq g(s_1 \ldots s_n)\}; true)} \; f, g \in C, \; f \neq g$$
- *Disproof 2*:
$$\frac{(E \cup \{f(t_1 \ldots t_n) \simeq x\}; \Phi)}{(E \cup \{f(t_1 \ldots t_n) \simeq x\}; true)} \; f \in C$$
- *Decompose*:
$$\frac{(E \cup \{f(t_1 \ldots t_n) \simeq f(s_1 \ldots s_n)\}; \Phi)}{(E \cup \{t_1 \simeq s_1 \ldots t_n \simeq s_n\}; \Phi')} \; f \in C$$

where Φ' is Φ_i for $E_i = E \cup \{t_1 \simeq s_1 \ldots t_n \simeq s_n\}$. The two *Disproof* inference rules detect that equalities between constructor terms have been derived. By the principle of definition, the theory of E_0 does not include such equalities. They are a consequence of adding the inductive conjecture $s \simeq t$, which is therefore disproved. The *Decompose* rule is added for the purpose of efficiency. It replaces an equation $f(t_1 \ldots t_n) \simeq f(s_1 \ldots s_n)$, where f is a constructor, by the equations $t_1 \simeq s_1 \ldots t_n \simeq s_n$: since f is a constructor, by the principle of definition two terms $f(t_1 \ldots t_n)$ and $f(s_1 \ldots s_n)$ may be equal only if their arguments are equal.

Theorem 2.10.1 (Huet and Hullot 1982) [68], (Bachmair 1988) [10] *If E is a ground confluent equational presentation, satisfying the principle of definition, the Unfailing Knuth-Bendix completion procedure enriched with the inference rules* Decompose, Disproof 1 *and* Disproof 2 *is a semidecision procedure for the complement of $Ind(E)$.*

A more general oracle was proposed in [71] for the Knuth-Bendix completion procedure and extended to the UKB procedure in [10]. This test is based on *ground reducibility*: a term t is *ground E-reducible* if for all ground substitutions σ, $t\sigma$ is E-reducible. Ground E-reducibility is decidable [107] only if E is a ground confluent rewrite system. Therefore, the test in [71, 10] applies only if the input presentation E is ground confluent and all the input equations can be oriented into rewrite rules.

We assume that E has these properties and we call it R. An equation $l \simeq r$ is *ground R-reducible* if for all ground substitutions σ, such that $l\sigma$ and $r\sigma$ are distinct, either $l\sigma$ or $r\sigma$ is R-reducible. If an equation $l \simeq r$ which is not ground R-reducible is derived from $R \cup \{s \simeq t\}$ at stage i, there is a ground instance $l\sigma \simeq r\sigma$ of the equation such that $l\sigma$ and $r\sigma$ are distinct and R-irreducible, but $l\sigma \simeq r\sigma \in GTh(E_i)$. This means that Φ_i is true and the inductive conjecture is disproved. The following inference rule implements this test [10]:

Disproof 3:

$$\frac{(E \cup \{l \simeq r\}; \Phi)}{(E \cup \{l \simeq r\}; true)} \; l \simeq r \; is \; not \; ground \; R - reducible$$

Theorem 2.10.2 (Jouannaud and Kounalis 1986) [71], (Bachmair 1988) [10] *If R is a ground confluent rewrite system, the Unfailing Knuth-Bendix completion procedure enriched with the inference rule* Disproof 3 *is a semidecision procedure for the complement of $Ind(R)$.*

Both Theorem 2.10.1 and Theorem 2.10.2 assume a uniformly fair search plan on the domain of all ground equations. The ground reducibility test is not a practical solution to the problem of inductive theorem proving, because its complexity is very high. Furthermore, most results about the UKB procedure for disproving inductive conjectures have been obtained in contexts where completion was considered a generator of confluent systems and the capability of disproving inductive conjectures was regarded as a side effect. This explains why a uniformly fair search plan was assumed. On the other hand, disproving an inductive conjecture is a semidecision process of a specific target. Therefore, only a proof of the target needs to be reduced. We define the set of the minimal proofs of the target Φ_i as follows:

$$\Pi(S_i, \Phi_i) = \Pi(S_i, min\{\psi\sigma |\ \psi \in S_i, \psi\sigma \in GTh(S_i) - GTh(S)\}),$$

that is a minimal proof of Φ_i is given by a minimal proof of the smallest ground

instance of some clause in S_i which is a theorem in S_i but not in S. In the equational case, a completion procedure which eventually generates a ground confluent set of equations, is able to reduce the proofs of all ground theorems and therefore the proof of the target. However, this is not necessary. Since a proof of the target is a proof of the smallest ground theorem which is not a theorem of the original presentation, we can restrict our attention to a smaller set of ground theorems:

Definition 2.10.1 (Fribourg 1986) [48] *Given a ground confluent presentation E, a set of substitutions H is* E-inductively complete *if for all ground substitutions ρ, there exist a substitution $\sigma \in H$ and a ground substitution τ such that $\rho \rightarrow_E^* \sigma\tau$.*

For instance, if E includes the axioms $0 + x \simeq x$ and $succ(x) + y \simeq succ(x + y)$, a set of substitutions H is E-inductively complete if it contains two substitutions σ_1 and σ_2 such that $\{x \mapsto 0\} \in \sigma_1$ and $\{x \mapsto succ(y)\} \in \sigma_2$. Indeed, all ground terms reduce either to 0 or to some term $succ^n(0)$, so that a set of substitutions which covers the instances $x \mapsto 0$ and $x \mapsto succ(y)$ cover all instances.

Clearly, we are interested in minimal E-inductively complete sets of substitutions. All such sets are equivalent for our purposes, since they all have the property of covering all the ground substitutions. We denote by H_E one such set and by $\mathcal{IT}_E$ the domain of all the ground equations which are instances of substitutions in H_E, that is $\mathcal{IT}_E = \{(l \simeq r)\sigma\tau \mid \sigma \in H_E, (l \simeq r)\sigma\tau \text{ is ground}\}$. A minimal proof of the target is a minimal proof of the smallest ground theorem which is not a theorem of the original presentation. This smallest ground theorem is in $\mathcal{IT}_E$ and therefore reducing the proofs of the theorems in $\mathcal{IT}_E$ is sufficient to guarantee that the proof of the target is reduced. This result was first proved in [48] for the application of Knuth-Bendix completion to disprove inductive theorems in equational theories:

Theorem 2.10.3 (Fribourg 1986) [48] *A completion procedure $\mathcal{C} =< I; \Sigma; \mathcal{O} >$ on the domain $\mathcal{IT}_E$, with complete inference rules and fair search plan is a semidecision procedure for the complement of $Ind(E)$ for all equational presentations E, for which*

the oracle $\mathcal{O}$ is computable.

As a consequence, the Deduction inference rule of UKB can be restricted considerably, while still preserving the completeness of UKB as semidecision procedure to disprove inductive conjectures [48]. At stage i of the derivation, superpositions on $p \simeq q$ at position u are performed only if the set of mgus $\{\sigma | l\sigma = (p|u)\sigma, l \simeq r \in E_i\}$ is E-inductively complete. A position u with this property is called *completely superposable* in [48]. Furthermore, for all equations $p \simeq q$, generated during the derivation, it is sufficient to perform superpositions on just one completely superposable position in $p \simeq q$. In other words, a search plan which selects just one completely superposable position in every equation is fair. These modifcations require an algorithm to detect the completely superposable positions. An equivalent characterization is the following: a position u in p is completely superposable if for all ground instances $(p|u)\rho$ there is an equation $l \simeq r$ in E such that $(p|u)\rho = l\sigma$ and $l\sigma \succ r\sigma$. The problem of detecting completely superposable positions reduces to the ground reducibility problem. However, if the presentation satisfies the principle of definition, a position u is completely superposable if $p|u$ is a term which has a defined symbol at the root and only constructor symbols and variables at the positions below the root. Therefore, the restriction to completely superposable positions can be applied in practice to presentations satisfying the principle of definition.

Chapter 3

Problems and current approaches in parallel deduction

In this chapter we analyze the problems in parallel theorem proving through a survey of some state-of-the-art parallel provers. We classify theorem proving strategies as *subgoal-reduction* strategies, *expansion-oriented* strategies and *contraction-based* strategies. We define three types of parallelism: *parallelism at the term level* (fine granularity), *parallelism at the clause level* (medium granularity) and *parallelism at the search level* (coarse granularity). We observe that as we move from subgoal-reduction strategies to expansion-oriented strategies and finally to contraction-based strategies, the data base becomes more and more *dynamic* and the amount of *inter-dependence* between inferences grows. Therefore, while subgoal-reduction strategies are amenable to all three types of parallelism and expansion-oriented strategies may still exploit both clause level and search level parallelism, only the latter is suitable for contraction-based strategies.

We reach these conclusions by considering, in this order, parallel term rewriting for equational languages [43, 51, 82], parallel Prolog technology theorem proving [23, 31, 110], parallel expansion-oriented methods [69, 32], parallel implementations of

the Buchberger algorithm [55, 111, 121], parallel implementations of Knuth-Bendix completion [33, 127] and the parallel theorem prover ROO [94]. In the closing section, we focuse on contraction-based strategies and parallelism at the search level. We analyze the advantages and disadvantages of *distributed memory* versus those of *shared memory* to implement a coarse grain parallelization of contraction-based strategies. The outcome is in favor of a mixed approach, i.e. a distributed environment with a shared memory component.

3.1 The classification of strategies

For the purpose of parallelization, we classify strategies into three categories:

- **Subgoal-reduction strategies**:

  Strategies for functional and logic programming and some theorem proving strategies, such as Prolog technology based methods, belong to this class. The presentation, e.g. a Prolog program, is not modified during the computation. The elements in the presentation are applied to reduce the target to subgoals, until a solution is reached. Thus a derivation has the form

  $(S_0; \varphi_0; A_0) \vdash_{\mathcal{C}} (S_1; \varphi_1; A_1) \vdash_{\mathcal{C}} \ldots (S_i; \varphi_i; A_i) \vdash_{\mathcal{C}} \ldots$

  where $S_0 = S_1 = \ldots = S_i = \ldots$. The target φ_i is the current goal and A_i is the set of its ancestors, which may be saved for the purpose of backtracking. The data base is *static*.

- **Expansion-oriented strategies**:

  this class includes

  - theorem proving strategies which feature only expansion rules and
  - theorem proving strategies which feature also contraction rules, but apply them only for *forward contraction*.

A derivation has the form

$(S_0; N_0) \vdash_{\mathcal{C}} (S_1; N_1) \vdash_{\mathcal{C}} \ldots (S_i; N_i) \vdash_{\mathcal{C}} \ldots$

where $S_0 \subseteq S_1 \subseteq \ldots \subseteq S_i \subseteq \ldots$. The component N_i is the set of *raw clauses*, i.e. the newly generated clauses. We call them "raw clauses", because they have not been contracted yet. Whenever a raw clause ψ is generated from two elements in S_i, it is added to N_i. Then ψ is reduced to its normal form ψ' with respect to S_i. This is forward contraction, because the clauses in S_i have been generated before ψ. If it has not been deleted, ψ' is added to S_{i+1}. No contraction is ever applied to any clause in S_i, i.e. there is no *backward contraction*, so that the data base is *monotonically increasing.*

- **Contraction-based strategies**:

  this class includes theorem proving strategies which feature both expansion and contraction rules, and apply the latter eagerly, for both *forward and backward contraction.* A derivation has the form

  $S_0 \vdash_{\mathcal{C}} S_1 \vdash_{\mathcal{C}} \ldots S_i \vdash_{\mathcal{C}} \ldots$

  where $R(S_0) \subseteq R(S_1) \subseteq \ldots \subseteq R(S_i) \subseteq \ldots$. Whenever a raw clause ψ is generated from two elements in S_i, it is added to S_i to form S_{i+1}. Then ψ is reduced to its normal form ψ' with respect to S_i. If it has not been deleted, ψ' is used to try to contract all elements in S_i. This is backward contraction, because ψ' has been generated after the clauses in S_i. Since contraction is applied to clauses in S_i, the data base is *dynamic* and the only monotonicity property is $R(S_0) \subseteq R(S_1) \subseteq \ldots \subseteq R(S_i) \subseteq \ldots$, where R denotes the redundancy criterion (see Section 2.6) associated to the strategy.

This classification is not meant to be complete, nor to capture all properties of the strategies. Rather, it emphasizes those aspects which will be relevant to parallelization, such as the static or dynamic nature of the data base.

3.2 The granularity of parallelism

After having classified the types of strategies, we classify the types of parallelism, in terms of *granularity*. We regard *parallelism at the term level* as fine grain parallelism, *parallelism at the clause level* as medium grain parallelism and *parallelism at the search level* as coarse grain parallelism. For each one of these types of parallelism we identify the granularity of data and the granularity of operations. The granularity of data tells which type of datum represents a *grain* of memory with its own access rights. The granularity of operations tells which type of operation is executed by each concurrent process:

	granularity of data	granularity of operations
parallelism at the term level	term	subtask of inference step
parallelism at the clause level	clause	inference step
parallelism at the search level	set of clauses	many inference steps

In **parallelism at the term level** every term is a *grain* of memory. Two processes may access concurrently two subterms of a same term, provided certain conditions, such as disjointness, are met. Thus, the data of the problem may be partitioned among the concurrent processes by giving to concurrent processes disjoint subterms and concurrent processes may cooperate to achieve a single inference step. Parallel matching, parallel unification and parallel rewriting (e.g. [43, 51, 82]) are examples of parallelism at the term level. At the **clause level**, every clause is a grain of memory: the data are partitioned by giving to concurrent processes distinct clauses and

typically each process performs one inference step or relatively few inference steps with a common premise. Most of the methods which we shall survey in the following sections work at the clause level (e.g. [94, 69, 111, 121, 128]). In **parallelism at the search level**, concurrent, asynchronous processes search in parallel for a solution. As soon as one of them succeeds, the whole process succeeds. The search space is partitioned among the processes by distributing the clauses. Unlike in fine and medium grain parallelism, each process is given a large portion of the data, e.g. a fairly large set of clauses, and the data sets of the processes are physically separated. Each process develops its own derivation, while communicating with the other processes. Some parallelism at the search level will be observed in [32, 33, 55].

3.2.1 Shared memory versus distributed memory

The granularity of parallelism is related to the choice between *shared memory* and *distributed memory*. The *local memory* of a process p_i is the memory which is accessible only to p_i itself. We use *distributed memory* for a configuration where the entire memory of the system is represented by the union of the local memories. If a process p_j needs to access data belonging to another process p_i, process p_j has to send a *message* to p_i. All non-local references are handled through messages. We term, instead, *shared memory* a memory which can be accessed directly by all the processes.

Common data is shared in shared memory, while it is duplicated in distributed memory. Sharing requires some sort of protection to prevent concurrent-write, e.g. the *critical regions* applied in [121] or the *locks* used in [94, 128]. Protection may slow down write-access to the shared data, imposing synchronization delays. A distributed approach does not require synchronization and therefore it may allow in principle a higher degree of parallelism. The cost is represented by a larger amount of memory and the communication delay of message-passing.

Parallelism at the term level and parallelism at the clause level lead in general

to adopt a shared memory: for instance, it would not be realistic to scatter the terms of a clause over a distributed memory. Parallelism at the search level can be implemented in principle in either shared or distributed memory, provided that the data sets of the processes are physically distinct.

3.2.2 Conflicts

The granularity of parallelism is also related to the problem of conflicts between concurrent processes. We say that two concurrent processes are in **conflict** if they need to access the same grain of data. First we describe conflicts at the clause level and then we discuss the relation between granularity and conflicts. Two inference steps whose sets of premises are disjoint are trivially not in conflict. In Appendix A.2, we examine all possible combinations of two expansion or contraction inferences with a premise in common. Expansion steps do not modify their premises. In other words, expansion steps simply need to be granted *read-access* to their premises. We assume that concurrent read can be safely admitted and therefore expansion steps do not cause conflicts. In the presence of contraction steps, three types of conflicts may emerge:

- *write-write conflict between contraction steps*: two concurrent contraction steps try to re-write the same clause φ,
- *read-write conflict between contraction steps*: a contraction step tries to re-write a clause φ, which is being read by another contraction step (which would use φ to contract another clause ψ) and
- *read-write conflict between contraction steps and expansion steps*: a contraction step tries to re-write a clause φ, which is being read by an expansion step (which would use φ as parent).

Under the assumption that raw clauses are not involved in any inference before

undergoing forward contraction, both types of read-write conflicts are caused by backward contraction. In Appendix A.2, we show that read-write conflicts between contraction steps do not represent real conflicts: if the step deleting the common premise φ prevents the application of φ to contract ψ, the latter can still be reduced. Therefore in the following we shall mention only the two remaining types of conflicts. Write-write conflict between contraction steps are real conflicts, since one step prevents the other. However, even if just one of the two steps commits, the common premise φ is reduced to a smaller form. Read-write conflict between contraction steps and expansion steps are the most critical ones and they are due to backward contraction. If the expansion step proceeds, it may generate a clause which is very likely to be redundant, because one of its parents is not fully reduced. On the other hand, suspending the expansion step until it is guaranteed that its premises are fully reduced may not be possible without introducing a very long delay.

Different granularities mean different approaches to the problem of conflicts. Given the conflicts at the clause level, one approach is to eliminate some of them by adopting a finer granularity, i.e. moving from the clause level to the term level. If access is at the term level, steps accessing disjoint positions within a clause are not in conflict. The cost of this choice is the overhead of handling access at the term level. Another approach moves in the opposite direction, i.e. from the clause level to the search level. In parallelism at the search level *there are no conflicts, because the concurrent processes access separate sets of data.* Two concurrent processes may re-write concurrently the same clause, because they are re-writing two distinct copies of the same clause. In other words, conflicts are eliminated by duplicating premises, at the cost of introducing additional redundancy with respect to a sequential execution.

3.3 Subgoal-reduction strategies

3.3.1 Prolog technology parallel theorem provers

The idea of a *Prolog technology theorem prover (PTTP)* appeared first in [115, 117]. A Prolog technology prover is basically a resolution-based theorem prover realized on top of a WAM (Warren Abstract Machine) [122], the virtual machine designed for the fast implementation of Prolog. The inference system of a PTTP is an extension of Prolog's inference mechanism to first order logic, based on the *model elimination* principle [90]. The search plan is *depth-first iterative deepening* [85, 116], a modification of the depth-first search procedure of Prolog. The latter is "incomplete" (unfair, in our terminology), whereas the former is "complete" (fair). Depth-first iterative deepening is depth-first search with a limit k on the depth of the descent. If, after k steps of depth-first search along a path, no solution has been found, the procedure backtracks and selects another path. If all paths have been pursued down to depth k and yet no solution has been found, the limit k is raised to $k + n$, for some $n \geq 0$, and the search restarts.

Two parallel Prolog technology theorem provers are the PARTHENON [23, 31] and PARTHEO [110] systems. In this Prolog-based setting, parallelism is *or-parallelism*: each concurrent process tries to apply a clause, or "rule" in Prolog, to one of the current goals. All processes have access to the input set of clauses. At first, each process selects an input clause, possibly a different one for each process, and tries to resolve it with the input goal, i.e. the negation of the target theorem, generating new subgoals. A goal is regarded as a *task*, namely the task of trying to derive the empty clause from that goal. The distribution of tasks among the processes is done through *task stealing*: if a process runs out of active tasks, either because no rule applies to its tasks, or because of the limit k in the descent, it gets more tasks from the queues of other processes. Iterative deepening can be parallelized in two ways: either the same limit k is set for all processes, or each process

has its own limit. In the first case, the derivation reaches the limit when the processes fall in a deadlock, all of them asking each other for tasks. The deadlock may be solved by the intervention of the user, who resets the limit to a higher value. In the second case, the scheduling procedure becomes more complicated as the queues of the processes may contain tasks pertaining to different values of k. In order to preserve the fairness of sequential depth-first iterative deepening, tasks originated under lower values of the limit need be given higher priority.

These are just the basic elements of a parallel Prolog technology theorem prover. PARTHENON implements them for shared memory machines, whereas PARTHEO implements them on a network of transputers. We refer to [23, 31, 110] for specific descriptions of how the principles outlined above are realized in these two systems. One aspect of PARTHEO which is relevant to our discussion is the mechanism for task stealing. In PARTHEO, task stealing is realized by message-passing. A message representing a subgoal φ is an encoding of the WAM operations that are necessary to derive φ from the input clauses. Upon receipt of such a message, a processor derives φ by executing the steps encoded in the message. Thus it happens routinely that the processors repeat steps that other processors have already performed. The rationale for this choice lies in the general philosophy of Prolog technology theorem proving. PTTP makes a limited use of contraction inference rules. For instance, in PARTHEO, tests for subsumption, tautology elimination and purity reduction [28] are applied to the input set only in a pre-processing phase. Prolog technology theorem proving is a representative of the choice of not investing significant resources in contraction and concentrating all efforts in speeding the expansion process. One hopes to make expansion steps so fast, that it becomes irrelevant, performance-wise, whether they are redundant or not. Accordingly, the authors of PARTHEO assume that the WAM operations are so fast, that the redundancy represented by repeating them upon receipt of messages does not harm the performances.

3.3.2 Parallel term rewriting for equational languages

A large part of the existing work on parallel simplification has been done with the purpose of designing parallel interpreters for equational languages and parallel architectures to support them [43, 51, 82]. Most interpreters for equational languages fit in the paradigm of *functional programming*: a program (i.e. the presentation) is a set of equations E, the input (i.e. the target or goal) is a term t and an execution consists in reducing t by the equations in the program until it is E-irreducible. The term in normal form is the output.

The definitions of equational languages often impose restrictions on the equations forming a program, in order to ensure the *uniqueness of normal forms* and thus the uniqueness of the output of a program for a given input. A sufficient condition for the uniqueness of normal forms is that the set of equations E is confluent (see Section 2.1 for all the definitions). In turn, if E is locally confluent and terminating, i.e. there are no infinite sequences $t_1 \rightarrow_E t_2 \rightarrow_E \ldots t_n \rightarrow_E \ldots$, E is confluent. The termination requirement is trivially satisfied if $t \rightarrow_E s$ is defined as $t \leftrightarrow_E s$ and $t \succ s$, for a well-founded ordering $\succ$, as we have done in Section 2.1. However, when sets of equations are interpreted as programs, it may not be regarded as desirable that termination is built in the definition of the computation mechanism itself. Most authors prefer to regard equations in programs as rewrite rules $l \rightarrow r$, where it is not necessarily the case that $l \succ r$. In the absence of termination, sufficient conditions for confluence are local confluence and left-linearity of the rewrite rules. Finally, local confluence is trivially implied by the requirement that the equations be *non-overlapping* [57]: it is never the case that a side of an equation unifies with a subterm of a side of another equation. Most of the work done on equational programs applies indeed to programs defined as sets of non-overlapping and left-linear rewrite rules.

3.3.3 Parallelization of subgoal-reduction strategies

The following properties of subgoal-reduction strategies can be observed in both equational programs and Prolog technology theorem proving:

- **Static data base**:

  The input presentation, e.g. the equations of an equational program or the "rules" in the sense of Prolog, remains unchanged throughout the execution. In PTTP, the subgoals are not subject to contraction after having been generated. This has three consequences:

  - **Pre-processing**:

    It is possible to *pre-process* the input clauses for fast execution. For instance, the equations of an equational program can be pre-processed for the purpose of parallel matching, e.g. [82], or compiled in a lower level language with explicit parallelism.

  - **Read-only data**:

    The input clauses are read-only data, which may be stored conveniently in a shared memory.

  - **Specialized data structures**:

    Since there is a very definite distinction between the goal and the "rules", or the term to be rewritten and the simplifiers, these two different types of data may be represented by different, specialized data structures. For instance, in PTTP, it is possible to represent a subgoal by an encoding of the WAM operations which derive it (e.g. [110]). In equational programming, specialized data structures are used to prevent conflicts among concurrent rewriting steps [82].

- **No conflicts**:

  In equational programming, the only conflicts which may arise are write-write conflicts between contraction steps. If two non-overlapping equations $l_1 \simeq$

r_1 and $l_2 \simeq r_2$ can simplify a given term t, the non-overlapping property implies that l_1 and l_2 match two subterms of t at disjoint positions. Intuitively, two simplification steps which reduce disjoint subterms need write-access to different portions of the term and therefore can be applied in parallel. If all the equations in a program are non-overlapping, this is true for *any* two steps in any execution of such program. In other words, there are *no conflicts*.

Furthermore, by exploiting pre-processing of equations and specialized data structures, it is possible to weaken significantly the requirements that the equations be left-linear and non-overlapping, while maintaining the *no conflicts* property [82].

In PTTP the basic inference mechanism features only expansion steps, i.e. the resolution-type expansion inferences which derive from a goal and a "rule" a new set of subgoals. Therefore, there are *no conflicts* between inference steps. For instance, two processes which read the same "rule" and apply it to different subgoals can be executed in parallel.

Because of these properties, i.e. *static data base* and *no conflicts*, subgoal-reduction strategies are amenable to all three types of parallelism: parallelism at the term level, e.g. parallel rewriting, parallelism at the clause level, e.g. or-parallelism, and parallelism at the search level, e.g. the distribution of subgoals to be solved independently by concurrent processes.

3.4 Expansion-oriented strategies

Examples of approaches to parallel expansion-oriented theorem proving are the method in [69], the DARES system [32] and the three parallel implementations of the Buchberger algorithm in [55, 111, 121]. We shall explain in the following why we regard the Buchberger algorithm as expansion-oriented for the purpose of parallelization.

The method in [69] utilizes parallelism at the clause level in shared memory. We do not describe this system in detail, because most of the issues and considerations it may raise will be covered by the study of parallelism at the clause level in shared memory for contraction-based strategies (see the analysis of [94] in Section 3.5.2).

The DARES project is interesting to us as an example of application of coarse grain parallelism to expansion-oriented theorem proving, and we describe it in the following.

3.4.1 The DARES system

DARES (Distributed Automated REasoning System) [32] is an application to theorem proving of technologies for distributed problem solving in artificial intelligence. It assumes a resolution-based strategy with a *level-saturation* search plan. The inference mechanism features resolution and forward subsumption, but no backward contraction. A number of processes, called *agents*, work concurrently. In turn, each agent comprises many sub-processes: several deductive processes and one *mail process*, which is dedicated to inter-agents communication. An agent is a sequential process: the processes inside an agent are scheduled for execution according to a round-robin scheduling. All deductive processes of all agents execute the given strategy. In general, different deductive processes work on different problems. It is not clear in the paper whether this means that more than one problem may be given in input or whether the input problem is reduced to sub-problems during the computation. In the second hypothesis, it is not specified how the given problem is reduced to sub-problems to be given to different processes. The only constraint is that no two processes of the same agent work on the same problem.

The authors observe that theorem proving is a computationally intensive application and therefore the agents should be *loosely coupled*, in the sense of devoting more time to computation than to communication. Indeed, each agent has several

deductive processes and just one mail process. Each agent has "incomplete knowledge". This means that it has access only to a subset of the clauses involved. It follows that the agents need to "import knowledge" from other agents, in order to be able to solve the problems. This communication is organized as follows. Whenever an agent A cannot derive new resolvents from its clauses, it sends to other agents a request for more data. Upon receiving such a request, any other agent B will reply by sending to A a subset of its clauses. Agent A will include the received clauses in its data base and proceed with the inferences.

This scheme of communication uses three heuristic criteria to decide, respectively, when to issue a request, what to put in the request and what to put in the reply. In addition to sending a request when it cannot generate new resolvents, an agent applies an heuristic criterion to establish whether it is making progress toward the solution or not. If the decision is negative, it emits a request for more data. One such criterion is described in [32]. It is based on measuring the length of the clauses and the number of distinct predicates in successive sets of new resolvents. Inference steps which reduce these measures, together with steps which generate or use unit clauses, are regarded as "proof-advancing". If the derivation is not proof-advancing for two consecutive stages, the agent sends a request-message to import more clauses. The decision is based only on the two most recent stages of the derivation. A request-message consists of a set of literals. Another heuristic is employed to select the literals to be put in the message. Subsumption is applied to make sure that a request-message does not contain instances of other literals in the message. The receiver of a request-message tests the literals in its clauses for unifiability with the literals in the message. If at least a literal in a clause unifies with at least a literal in the message, the clause is selected. Then, all the selected clauses will be sent as a reply-message to the agent which issued the request-message. Subsumption is used also to delete instances in a reply-message. This application of subsumption is actually the only backward contraction in DARES. Clearly, this does not change the expansion-oriented nature of the method, because it is applied only within the messages, not directly to the data bases of the processes.

DARES has been implemented on a simulator of distributed systems, which runs on a network of Lisp machines. Because of the presence of the simulator, the absolute run times of such a system are probably not very significant and indeed they are not given in [32]. Experimental measures of how the run time varies with the number of agents and the amount of redundancy in the system are reported. However, this study refers the results for just one theorem proving problem and it is not said which one. More experiments may be found in [95]. In this context, redundancy simply means duplication: the *redundancy factor* for a clause is 0 if there is just one copy of that clause in the whole system, it is 1 if all agents have a copy of that clause. This study is more concerned with redundancy as a property of the given distribution of clauses, than with redundancy generated by the system itself. Indeed, redundancy is treated as an independent variable. Not surprisingly, as redundancy grows, each agent's behaviour becomes closer to that of a sequential theorem prover and the performances of DARES get worse.

The effectiveness and efficiency of a system like DARES rest on the heuristics used in the communication part, which seem rather simple. There is no treatment of the *fairness* of such heuristics, i.e. whether they ensure that whenever there is a proof, it will be found. The selection of clauses for the reply-messages seems very expensive, since it consists in actually trying resolution between the clauses in the data base and the literals in the request-message. These resolution steps will be performed again when the reply-message reaches its destination. Finally, these heuristics may not be as effective as desirable in partitioning the search space. For instance, if the literals in the request-message are sufficiently general, e.g. literals all of whose arguments are variables, almost all the clauses may be selected, so that the data bases at the sender and at the receiver of the request-message will be almost identical. Furthermore, no distinction is made between generated clauses and received clauses: if a clause φ is generated by agent A and then sent to agent B, both A and B will perform all the inference steps they can with φ. As more and more clauses are exchanged, this also contributes to increase the overlap between the data bases and thus the portions of search space allotted to different agents.

3.4.2 Parallel implementations of the Buchberger algorithm

Problems related to those of parallel theorem proving have been addressed by the study of parallel and distributed implementations of the *Buchberger algorithm* [55, 111, 121]. The Buchberger algorithm works on polynomials, equated to 0 and treated as oriented equations. It takes as input a set of polynomials and gives as output a set of polynomials, which is a *basis* for the ideal generated by the input polynomials. A basis has the property that it reduces to 0 all and only the polynomials belonging to the ideal [24].

The Buchberger algorithm features an expansion inference rule which is similar to superposition and a contraction rule which is similar to simplification. We present these parallel implementations of the Buchberger algorithm as parallel expansion-oriented strategies, because they do very little backward contraction. Indeed, the importance of backward contraction is much higher for theorem proving than for the Buchberger algorithm. The latter is an algorithm, i.e. it is guaranteed to halt, regardless of the amount of contraction performed. If the output basis is not fully reduced, it is possible to follow the execution of the parallel algorithm by a final phase, where a sequential normalization algorithm is applied to all the elements of the basis. This is done for instance in [111, 121]. On the other hand, theorem proving strategies are semidecision procedures. In theorem proving, the growth of the generated search space may cause the saturation of the available memory, so that the prover fails to find a proof. The extensive application of backward contraction is of dramatic importance to contain such phenomena. Also, the expansion inferences in the Buchberger algorithm do not use unification, since there are no variables, as the "variables" in the polynomials are constants logically. It follows that expansion steps are much less expensive than in theorem proving. Therefore, expansion-oriented implementations of the Buchberger algorithm are less expensive and thus more feasible than expansion-oriented approaches in theorem proving.

The three algorithms presented in [55, 111, 121] are conceived for three different models of parallel computation: a *shared memory multiprocessor* in [121], a *data-flow machine* in [111] and a *distributed memory multiprocessor* in [55]. In [121], the data base of equations is stored in shared memory with **Concurrent-Read-Exclusive-Write (CREW)** access. The algorithm is a parallel version of the Buchberger algorithm with *critical regions* to enforce exclusive-write. The algorithm is very simple and a formal proof of correctness is given in [121]. However, it seems to contain an unnecessarily high degree of sequentiality. The reason is that the granularity of memory protection is very large: entire sets of equations, e.g. the current version of the basis and the set of raw critical pairs, are accessed in exclusive-write mode. Only one expansion process at any given time can access the set of raw critical pairs to add a new pair. It follows that the expansion part of the algorithm is basically sequential. No *backward contraction* is included.

The method in [111] is inspired by an analogy, due to Buchberger himself, between the Buchberger algorithm and the **Erathostenes's sieve** algorithm to generate all the prime numbers smaller than or equal to a given n. A data-flow version of Erathostene's sieve works as a *pipe* with stages $s_1 \dots s_i$ to store the prime numbers $m_1 \dots m_i$ discovered so far. The numbers from 1 to n are input in increasing order at one extreme of the pipe and travel through its stages. At each stage s_j, any candidate for primality k is tested for divisibility by m_j. If m_j divides k, k is eliminated. If no m_j's divides k, k is saved as m_{i+1} at a new stage s_{i+1}. When the input stream is exhausted, all the numbers remained in the pipe are the prime numbers.

According to the analogy between the Buchberger algorithm and Erathostene's sieve, simplification corresponds to the divisibility test and the basis of equations corresponds to the set of prime numbers. Any raw critical pair $l \simeq r$ travels through the pipe and the equations already stored in the stages of the pipe are applied to reduce $l \simeq r$. If $l \simeq r$ is not deleted in the process, i.e. it is reduced to a non-trivial equation $l' \simeq r'$, the latter is added as a new stage at the end of the pipe.

Then, a copy of $l' \simeq r'$ travels backward and at each stage it tries to generate raw critical pairs with the equation stored at that stage. Raw critical pairs are collected at the input extreme of the pipe and fed back in the pipe for simplification. To summarize, forward contraction is performed as new equations traverse the pipe in forward direction. Expansion is performed as the equations traverse the pipe in backward direction. When the algorithm halts, the contents of the pipe is a basis. Since each stage of the pipe hosts an equation, this is an instance of parallelism at the clause level.

This approach is weak with respect to backward contraction. The analogy with Erathostene's sieve is not helpful here: if a number k is not divided by any prime number smaller than k, then k is prime. It follows that no stage of the pipe ever needs to be deleted. On the other hand, for an equation $l \simeq r$ to be part of a fully reduced version of the basis, it is not sufficient that $l \simeq r$ is fully reduced with respect to the equations generated before $l \simeq r$. It is necessary to keep trying to reduce $l \simeq r$ by equations generated afterwards. According to [111], some backward contraction is performed when the equations travel in backward direction. However, it does not guarantee that the output basis is fully reduced. Indeed, the method features a final phase, where a sequential normalization algorithm is applied to all the elements of the basis output by the pipe.

A **distributed version of the Buchberger algorithm** is presented in [55]. The algorithm is designed to be applied to solve polynomial constraints during the interpretation of a program in a logic programming language with constraints. Thus the input to the Buchberger algorithm is not a given set of polynomials, but a stream of polynomials progressively generated by an interpreter. Each input equation is sent through a shared bus to all the processors. A processor p_i maintains two sets of equations: a set CP_i of raw critical pairs, just received from the input or just generated at the node, and a current version of the basis B_i. The distribution of the work among the processors is regulated by a *work-load function* w, which assigns equations to processors. Each processor p_i is responsible for normalizing with respect

to B_i those equations in CP_i that w assigns to p_i. Whenever the smallest equation $l \simeq r$ in CP_i is B_i-irreducible and belongs to p_i, node p_i broadcasts $l \simeq r$ to all the other nodes, generates all the critical pairs between $l \simeq r$ and elements in B_i, stores them in CP_i and moves $l \simeq r$ from CP_i to B_i. Any other processor p_j, upon receipt of $l \simeq r$, performs the same steps, i.e. it generates all the critical pairs from $l \simeq r$ and B_j, stores them in CP_j and moves $l \simeq r$ from CP_j to B_j.

All nodes do all the expansion steps independently. The choice of having each node doing all expansion steps is clearly related to the low cost, e.g. relative to theorem proving, of expansion steps in the Buchberger algorithm. Forward contraction is distributed among the processors according to the work-load function. The latter is originally a partition of the input and is updated dynamically during the computation. By partitioning the equations and by subdividing the forward contraction tasks, this approach realizes some parallelism at the search level. Once again, backward contraction is the weakest part. As a background task, not included in the main algorithm, each processor p_i is supposed to normalize with respect to B_i the equations of B_i which belongs to p_i. However, if an equation in B_i is simplified, the reduced form is not moved to CP_i and the B_j's at the other nodes are not modified accordingly.

3.4.3 Parallelization of expansion-oriented strategies

Expansion-oriented strategies do not have most of the properties of subgoal-reduction strategies. The picture changes as follows:

- **Monotonically increasing data base**:

  The data base grows during the derivation because of expansion steps. Therefore, it is no longer possible to pre-process all the clauses at "compile-time", i.e. before the derivation. Also, while a single clause in S_i can still be considered

as read-only data, because there is no backward contraction, the whole component S_i is no longer read-only, since expansion processes need write-access to add new clauses.

- **Conflicts**:
  - Read-write conflicts do not arise, because there is no backward contraction.
  - Write-write conflicts may appear as conflicts between forward contraction steps on a raw clause.

    The hypotheses which can be made for subgoal-reduction strategies, e.g. non-overlapping equations, cannot be assumed. The expressive power of systems of non-overlapping equations is regarded by several authors (e.g. [57, 81]) as sufficient for programming, but it is not in theorem proving, since many theories of interest have presentations including overlapping equations.

    In the absence of the non-overlapping assumption, interpreters of equational programs may feature specialized data structures and techniques, e.g. those in [82], to allow parallel rewriting. In equational programming, this machinery needs to be applied to just one term. In theorem proving, it should be applied to *all raw clauses ever generated*. Also, these techniques often require some pre-processing of the simplifiers. In equational programming, this is done at compile-time. In theorem proving, because the data base is growing, such pre-processing must be repeated *before every forward contraction task*, which may become very expensive.

    Thus, it may be better to perform forward contraction sequentially. Indeed, in a theorem proving derivation, normalization can be conceived as a single inference step.

While parallelism at the clause level and parallelism at the search level are still feasible (see [69, 111] and [32, 55] respectively), parallelism at the term level becomes

much less appealing, when moving from subgoal-reduction strategies to expansion-oriented strategies. The main reason is scale. In equational programming, the entire computation is a normalization of a single term. In theorem proving, a normalization is a small task with respect to the amount of work represented by the whole derivation. Then, the overhead of parallelism at the term level is likely to be excessive, defeating the advantages of parallelization.

3.5 Contraction-based strategies

In this section we overview a parallel transition-based implementation of Kunth-Bendix completion [127, 128], the parallel theorem prover ROO [94] and the *team-work method* [33].

3.5.1 Parallel transition-based Knuth-Bendix completion

The project described in [128] applies to Knuth-Bendix completion a **transition-based approach to parallel programming**, which was formerly developed in [127]. In the application to completion, the transitions are basically the inference rules of the completion procedure. In fact, the list of transitions defined in [128] follows closely the presentation of Kunth-Bendix completion as a set of inference rules which was given first in [7]. Examples of such transitions are "normalizing an equation", "finding whether an equation is trivial" or "applying a rule to try to simplify all left hand sides of rules" and the like. Each process executes the transitions according to the same scheduler. The basic scheduler described in [128] implements a simplification-first search plan, since it prescribes to select first the transitions performing simplification and to apply them until no longer applicable.

The data base of equations and rewrite rules is kept in a shared memory, organized as first-in first-out queues of pointers to equations. For instance, the transition

"generate all the critical pairs between two selected rules" add raw critical pairs to the $NewEqs$ queue. The transition "normalize an equation" selects an equation from the queue $NewEqs$ and puts its normalized form in the queue $NormEqs$. The usage of pointers allow to save some memory space by having equations occurring in more than one queue without being duplicated. Conflicts in the access to queues are avoided by using locks on the pointers to the front and the rear of each queue. Since the processes execute the same scheduler, the activities of different processes differ in the selection of the data. For instance, let p_1 and p_2 be two processes that are both scheduled to execute a transition of type "find whether an equation is trivial". Process p_1 accesses the queue $NormEqs$ and extracts an equation. As soon as process p_1 has released the lock at the pointer to the front of $NormEqs$, process p_2 may get it and extract another equation. After the test, if the equations are not trivial, they are appended at the rear of the queue $NontrivEqs$.

This method has been implemented on a Firefly with 6 CVAX and on a Sequent. In the theoretical description of the application of the transition-based approach to completion, each transition corresponds to an inference rule. The transitions actually implemented either apply an inference rule at most once to at most all the equations, or apply all inference rules that apply, as many times as possible, to a single selected equation. Thus parallelism is at the clause level.

The strong points of this system appear to be its simplification-first scheduler and a skillful implementation of the access to the shared queues. In particular, there are few critical regions and they are few instructions long. Also, equations are rewritten without locking. This is possible, because the hardware of the shared memory used in the implementation guarantees that if a location is read and written at the same time, the read instruction observes either the value before the write instruction or the value after the write instruction, but not an intermediate, unfinished value. If the read instruction is issued by a transition performing an expansion inference and the write instruction by a transition performing simplification, it may happen that the expansion inference applies to the equation before the simplification. Similarly,

two write instructions may try to access the same location at the same time and in such a case one write instruction may be lost.

According to the authors, these phenomena, e.g. the expansion of non-simplified premises and the loss of some rewrite steps, do not influence significantly the performances. The method seems to benefit from the advantages of shared memory, i.e. the effects of the steps performed by one process are immediately visible to all the others, without suffering too severly from its limitations, i.e. the delay in accessing shared structures. The speed-up reported in [128] is $\frac{4}{5}n$ for n processors on the Firefly. However, the Firefly is a very small machine and the $\frac{4}{5}n$ speed-up is obtained when the machine is dedicated exclusively to the Knuth-Bendix program. The implementation on the Sequent has suffered so far from the lack of garbage collection [129]. Also, the authors have considered the most basic version of Kunth-Bendix completion, rather than the Unfailing Knuth-Bendix procedure of [62, 13], so that whenever an unoriented equation is generated, the strategy fails.

3.5.2 The ROO parallel theorem prover

The **ROO theorem prover** [94] is a parallel, shared memory version of the theorem prover Otter [98, 99] for first order logic with equality. We describe Otter first. Otter is a refutational, resolution-based theorem prover. The data base of clauses is divided into two main components, the *Set of Support (Sos)* and the set of $Usable$ clauses. According to the Set of Support Strategy [28] each expansion inference step uses at least one parent from the Sos. If $Usable$ contains the presentation and Sos contains the negation of the target, Otter generates a derivation which resembles backward reasoning: all the generated clauses are descendants of the negated target. If $Usable$ contains the negation of the target and Sos contains the presentation, the derivation resembles forward reasoning: it generates clauses from the presentation until it obtains one that resolves with the negated target, or one of its descendants, to give the empty clause. Intermediate variations may be obtained

by different partitions of the input between Sos and $Usable$. In addition, Otter features a $Demodulators$ list, which contains equations to be used as simplifiers, and a $Passive$ list, whose clauses are used for forward subsumption and unit-conflict only. A unit-conflict step is a binary resolution step which generates the empty clause. In some problems, forward reasoning is realized by putting part of the presentation in $Usable$, part of the presentation in Sos and the negation of the target in $Passive$. The resulting derivation is even more strongly oriented toward forward reasoning than one where the negated target is in Usable, because no clause except the empty clause may be generated from the input target.

Otter works by executing a basic loop. At each iteration, Otter selects a clause, termed $given\_clause$, from the Sos. Let $i_1, \ldots, i_n$ be the set of expansion inference rules in Otter. For each rule i_k, $1 \leq k \leq n$, Otter executes two phases:

1. First, it generates all the clauses, $\psi_1 \ldots \psi_n$, that can be derived by rule i_k from the $given\_clause$ and any clause in the $Usable$ list. Each newly generated clause is *pre-processed*, i.e. subject to forward contraction, right after having been generated. For instance, Otter pre-processes ψ_1 before generating ψ_2. If ψ_1 is not deleted during pre-processing, ψ_1, or possibly a reduced form of ψ_1, is appended to Sos.

2. Second, the clauses which have been just appended to Sos are applied to contract pre-existing clauses. This stage is called *post-processing* and it corresponds to backward contraction in our terminology.

After these two phases have been performed for all expansion rules in $i_1, \ldots, i_n$, Otter appends the $given\_clause$ to $Usable$ and proceeds to the next iteration of the loop body.

This order of operations implements an almost simplification-first search plan. The reason why Otter is not strictly simplification-first is that only pre-processing, but not post-processing, is performed after the generation of every single new clause.

Post-processing is performed only after the generation of the batch of new clauses, derivable from the *given_clause* and the clauses in $Usable$, by each expansion rule.

The loop described above is the very basic mechanism of Otter. The prover features a wealth of options which allow the user to choose selections of inference rules, criteria to sort clauses in the Sos, criteria to retain or discard clauses, just to mention a few. The advantage of giving to the user so much leeway is that she/he can experiment with a variety of strategies. For instance, since not all combinations of options yield complete strategies, the user has the opportunity to play with incomplete strategies, which may be very interesting, as they may turn out to be especially efficient on specific problems.

The basic idea of the parallelization of Otter realized in ROO is to have several *given_clause*'s active in parallel, thereby realizing another instance of parallelism at the clause level. The authors call *Task A* the task of performing the body of the basic loop of Otter for a *given_clause*. Thus in ROO there are many concurrent processes executing Task A on different *given_clause*'s. This parallelization, though, causes three main problems:

- Concurrent append to Sos: it happens whenever two processes executing Task A try to append to Sos their newly generated clauses. Two such processes compete for write-access to Sos in shared memory.
- Addition of redundant clauses to Sos: redundancy is introduced because as process p_1 appends its batch of new clauses S_1 to Sos and process p_2 appends its batch S_2, the clauses in S_1 are not reduced with respect to the clauses in S_2 and vice versa. In fact S_1 and S_2 may even contain identical clauses and all these redundant clauses would be appended to Sos. This is an instance of conflict between parallel expansion and contraction.
- Concurrent deletion of clauses in Sos or $Usable$: process p_1, post-processing its new clause ψ_1, and process p_2, post-processing its new clause ψ_2, compete for

write-access to Sos or $Usable$ with the purpose of deleting clauses contracted by ψ_1 and ψ_2 respectively. This is clearly a problem with parallel backward contraction.

The authors of ROO have tried to overcome these obstacles by delaying additions of clauses to Sos and deletions from Sos and $Usable$. A process executing Task A is not allowed to append its new clauses to Sos. It appends them to another list, termed *K-list*. Similarly, a process executing Task A is not allowed to delete clauses from Sos or $Usable$ during post-processing. Rather, it appends an identifier of the clause to be deleted to another list, called *To-be-deleted*. A different task, *Task B*, consists in selecting a clause from the K-list, repeating pre-processing and post-processing for that clause and finally appending it to Sos. If processes p_1 p_2 have generated two clauses ψ_1 and ψ_2, such that ψ_1 can contract ψ_2, both ψ_1 and ψ_2 end up in the K-list. If ψ_1 is extracted first, ψ_1 is applied to contract ψ_2 as part of the post-processing of ψ_1. If ψ_2 is selected first, ψ_2 is contracted by ψ_1 as part of the pre-processing of ψ_2. In either case the redundant clause ψ_2 does not make it to the Sos. Also, the process executing Task B has the right to carry out deletions on Sos and $Usable$ according to the contents of To-be-deleted. All processes obey the same scheduler, which prescribes to execute Task B, if the K-list is not empty and no other process is doing it, Task A otherwise. Thus, only one process is allowed to be executing Task B at any given time.

Being implemented on top of Otter, ROO inherits the efficiency in the basic operations and the high portability that are among the outstanding features of Otter. The experimental results are remarkable, as ROO achieves near-linear speed-up on many problems, including difficult ones, such as those presented in [6]. On the other hand, the performances of ROO show some irregularities: first, the results are unstable, i.e. two executions of ROO on the same input may report different timings, due to the non-determinism of the parallel computation. Second, the prover obtains even super-linear speed-up on certain problems, but is very disappointing on others. The latter are certain equational problems, where high amounts of backward

simplification and backward subsumption are required. The problem is that in these examples Task B turns out to be a bottleneck. The K-list grows very long and the single process performing Task B falls behind in moving clauses from the K-list to *Sos*. It follows that other processes scheduled to execute Task A starve for work, in the absence of clauses to be picked as *given_clause*. This happens for instance if a very good simplifier ψ, which reduces most of the clauses in the data base, is generated. During the post-processing of ψ, almost all the clauses will be moved to the K-list, so that the process executing Task B is overwhelmed and the other processes are idle.

Another reason for concern is memory usage. The amount of memory required by ROO grows with the number of active processes. This is sort of expected to some extent. However, poor utilization of the allocated memory has been observed. Otter maintains a list of available records for each data type, e.g. "clause" or "symbol table entry" and the like. Whenever a clause is deleted, its record is put in the list of available records. Whenever a new clause is created, a record is taken from the list of available records, if it is not empty. Otherwise, memory is allocated from the operating system. Since most deletions are performed by Task B, the processes executing Task A may keep asking for allocation of new records, while the list of available records at the process executing Task B grows and remains unused. This problem is being addressed by having available records released by Task B made available to other processes.

3.5.3 The team-work method

The **team-work method** [33] is another application to theorem proving of technologies developed for artificial intelligence systems. It is applied to parallelize strategies based on Knuth-Bendix completion, thus contraction-based. It distinguishes four types of processes: "experts", "referees", "specialized experts" and a "supervisor". The experts are in charge of performing the inferences. Each expert receives a copy

of the input set of equations and it proceeds by using the inference rules of the strategy to develop its own derivation independently from the other experts. Thus the team-work method implements parallelism at the search level. The derivations produced by different experts are different in general, because different experts use different search plans. Periodically, all the experts halt and the second family of processes, the referees, start. The referees evaluate the derivations according to several measures. Some measures apply to the latest state of a derivation, i.e. the data base of equations the expert has generated so far. One very simple such measure is the number of equations in the data base. More refined measures, akin to those in [5], involve heuristics to estimate how useful the equations in the data base are, with respect to the purpose of reducing the target theorem. Other measures refer to the history of the derivation, e.g. the number of simplification steps performed so far. These criteria are used to rank the derivations and to extract "good" equations from the data bases the experts have produced. All these data are passed by the referees to the supervisor. The task of this process is to select the best derivation based on the informations provided by the referees. The latest state of the best derivation is given as new data base to all the experts. Also, "good" equations from other derivations may be added. Then all the experts restart from this common state. In addition, the experts may invoke specialized experts, i.e. processes devoted to a specific task, such as normalization or constraint solving.

The team-work method has some similarity with the DARES system. Both methods apply to distributed theorem proving techniques originally conceived for other artificial intelligence applications, e.g. experts systems. Both methods implement parallelism at the search level in a distributed environment. On the other hand, they address different classes of strategies, i.e. expansion-oriented for DARES and contraction-based for the team-work method. In DARES the input clauses are distributed among the agents, whereas all processes have all the input clauses in the team-work method. The latter allows different processes to execute different search plans, whereas in DARES all agents utilize level-saturation. Perhaps the most important difference lies in the control of communication. In DARES an agent

decides to communicate to others based solely on *local* information, i.e. the state of its own derivation. In the team-work method, the evaluation is done by the referees using *global* information, i.e. the states of all the derivations. The overall computation proceeds by alternating phases where the experts work independently to phases where the referees and the supervisor reconstruct a common state. This idea of periodical, global evaluation seems to be the distinctive feature of the team-work method.

The experimental results reported in [33] seem encouraging. The authors claim that interesting speed-up's may be obtained by allowing some experts to follow efficient, although unfair, search plans. The completeness of the method is saved, provided that the overall derivation is still fair. However, no definitions of global and local fairness and no proof of fairness of the method or of any specific version of it are given in [33]. Clearly, the basic idea in this approach is that the evaluation of the derivations and the consequent construction of a best data base gives a special edge. On the negative side, when the experts replace their own data base by the one indicated by the supervisor, a great amount of potentially useful work may be wasted. The quality of the measures employed by the referees is critical to contain such waste. Another cost is represented by the synchronization delay imposed on the experts to let the referees and the supervisor intervene. Such synchronization limits the capability of the method to exploit parallelism at the search level.

3.5.4 Parallelization of contraction-based strategies

For contraction-based strategies, the dynamic character of the data base and the incidence of conflicts increase even further, with respect to expansion-oriented strategies:

- **Highly dynamic data base**:

  Because of **backward contraction**, the data base S_i is no longer monotonic

during the derivation: additions by expansion are intertwined with deletions by contraction. All clauses are repeatedly subject to contraction, i.e. all the items in the data base need be accessible not only for reading but also for writing. It follows that there is **no read-only data**. Since each contraction step requires both read-access and write-access, the ratio of read-accesses versus write-accesses may be expected to be lower than in the expansion-oriented strategies. This makes shared memory, where write-access is more expensive than read-access, less appealing.

- **Conflicts**:

  In addition to write-write conflicts between contraction steps, read-write conflicts between expansion and contraction steps also appear, because of backward contraction.

All the considerations against parallelism at the term level in expansion-oriented theorem proving apply even more strongly to the case of contraction-based strategies. Furthermore, the presence of backward contraction challenges also the applicability of parallelism at the clause level. A clause which is reduced by a backward contraction step should be tested for further contraction with respect to all the other clauses. Thus, each backward contraction step may induce many. If one works with parallelism at the clause level in shared memory, this avalanche growth of contraction steps may cause a write-bottleneck, since all the backward contraction processes ask write-access to the data base in shared memory. We call this phenomenon the **backward contraction bottleneck**. Not all the backward contraction processes may be served and an otherwise unnecessary sequentialization is imposed. The clauses which are supposed to be subject to backward contraction may not be made available for other tasks, e.g. expansion steps, so that these are delayed in turn.

This phenomenon can be observed for instance in the transition-based parallel Knuth-Bendix procedure of [128] and in ROO [94], which both realize parallelism at the clause level in shared memory. In ROO, the concurrent processes executing

Task A are in conflict, when they try to access *Sos* to perform backward contraction. This is an instance of the backward contraction bottleneck. In order to avoid it, the concurrent processes executing Task A do not do backward contraction and leave it to the single process executing Task B. But the backward contraction bottleneck re-appears, since the K-list grows too long and Task B becomes the bottleneck.

The source of the problem is the choice of a granularity, i.e. the clause level, which is too fine for contraction-based theorem proving. Therefore, the solution is to adopt a coarser granularity and realize parallelism at the search level.

3.5.5 Parallelism at the search level

Parallelism at the search level tries to realize an intuitive idea of *parallel search*. A sequential deductive process may search along one path only, whereas many concurrent processes may search in parallel along different paths. For this intuition to be productive, it seems desirable that the parallel processes *do not overlap*. If they do, it is likely that they are in fact exploring the same paths in the search space. If the concurrent processes search along the same paths, the advantage of having more processes rather than one is wasted. On the other hand, processes which do not overlap and search different portions of the search space, seem more likely to find a solution faster than by sequential search. Thus, the concurrent deductive processes should be *largely independent, asynchronous* and cooperate *loosely*. Each process searches for a proof by developing its own derivation and as soon as one of them halts successfully, the whole process halts. The search space is partitioned among the processes by distributing the clauses and possibly subdividing the inferences. Because the search space is partitioned, the processes need to cooperate by exchanging data, e.g. clauses.

Parallelism at the search level may be realized in principle in either shared or distributed memory. However, **distributed memory** is preferable, because it implements naturally the requirement that the data sets of the deductive processes are

physically separated. The concurrent processes may access data for both reading and writing in a totally independent, asynchronous fashion. The costs are represented by the amount of memory and the communication delay of message-passing. For the latter, we observe that one important source of the need for message-passing in distributed systems is the maintainance of *agreement*. This property is also called *consistency* in the terminology of distributed data bases and *coherence* in the terminology of parallel architectures. It means that if a datum is duplicated, e.g. one copy at p_i and one copy at p_j, the two copies need to *agree*: whenever the copy at p_i is modified, the copy at p_j should be updated accordingly as soon as possible. In many applications of distributed systems, agreement is necessary for the operations of the system to be correct. Automated deduction is *not* such an application. If we have two copies of a clause φ, one at p_i and one at p_j, and the copy at p_i is simplified to φ', the two clauses φ and φ' are *logically equivalent*. It follows that the monotonicity and completeness of a theorem proving strategy are not hindered if agreement is not strictly enforced. It is a matter of performance, and not of correctness, whether and how promptly the copy of φ at p_j is updated. The fact that agreement is not necessary is a point in favor of distributed memory for theorem proving applications, since enforcing agreement may be expensive.

In the following, we shall consider either a purely distributed configuration or a *mixed* configuration with predominantly distributed memory and an additional shared memory. Shared memory is especially convenient for concurrent processes which cooperate closely and synchronously. We feel that most of the activities in contraction-based deduction are better implemented as intrinsically independent, asynchronous activities. However, there is an exception: forward contraction. The simplification steps in a normalization process are tightly cooperating processes, to such an extent that the whole normalization process may be conceived as a single inference step. Thus, it may be useful to have all the simplifiers stored in one place, e.g. in a shared memory component.

Chapter 4

The Clause-Diffusion methodology for distributed automated deduction

In this chapter we present the **Clause-Diffusion** methodology for distributed automated deduction. We assume an abstract distributed environment, which may represent a network of computers or a loosely coupled multiprocessor. Given such an environment and a sequential strategy $\mathcal{C}$, we describe in Section 4.1 how $\mathcal{C}$ is executed according to the Clause-Diffusion methodology. Depending on the choice of global contraction scheme, algorithm for distributed allocation, algorithm for routing and broadcasting of messages and schedule of operations at the nodes, the Clause-Diffusion methodology may yield different specific methods. For each of the above issues, and more which arise from them, we propose and discuss several solutions. First we consider distributed global contraction. Then we give guidelines for the scheduling of the operations at the nodes and we analyze, in this order, distributed allocation, routing/broadcasting, inferences on residents and inference messages, control messages.

4.1 The Clause-Diffusion methodology

Given a complete theorem proving strategy $\mathcal{C} =< I; \Sigma >$, we describe how $\mathcal{C}$ is executed according to the Clause-Diffusion methodology. In this section we give the main ideas of the method, and we shall dwell on the specific components in the following sections.

We consider a loosely coupled, asynchronous multiprocessor, or a network, with n nodes, connected according to some topology and possibly endowed with a shared memory component. Our methodology does not depend on a specific architecture; it can be realized on different ones. Parameters such as the amount of *memory at each node*, the availability of *shared memory* and the *topology* of the interconnection, are variable. Later, we shall discuss the suitability of some alternative variants of our methodology with respect to different memory configurations and topologies. For instance, when we give examples of specific routing algorithms, we shall select the *cube-connected* topology as an interesting case.

The basic idea in our approach is to have a deductive process running at each node and *to partition the search space* among the nodes. Since there is a one-one correspondence between the deductive processes and the nodes, we shall use $p_1 \dots p_n$ to denote ambiguously both the deductive processes and the nodes. The search space is determined by the input clauses and the inference rules. At the clauses level, the input and the generated clauses are distributed among the nodes. For this purpose we need an **allocation algorithm**, which decides where to allocate a clause. Once a clause ψ is assigned to processor p_i, ψ becomes a **resident** of p_i. We denote by S^i the set of residents at node p_i. In this way each node p_i is allotted just a subset of the clauses ever generated during the derivation. The union of all the S^i's, which are not necessarily disjoint, forms the current *global data base*. Each processor is responsible for applying the inference rules in I to its residents, according to the search plan Σ.

Since the global data base is partitioned among the deductive processes, none of them is guaranteed to find a proof by using only its own residents. To assure that a solution will be found when one exists, the nodes need to exchange information, by sending each other their residents in form of messages, called **inference messages**. Each node uses the received inference messages to perform inferences with its own residents. The inference messages issued by a process p_i let the other processes know which clauses belong to p_i, so that they can use them for inferences.

The communication of inference messages may be realized in different ways. In a purely distributed system, inference messages are implemented as messages, which may be routed or broadcast. If it is desirable to have all the nodes receiving the inference messages as soon as possible, then the inference messages should be broadcast. Depending on the broadcasting algorithm, a node may forward copies of the inference message to different nodes, while still retaining a copy for its own inferences. Thus, there may be several inference messages, all carrying the same clause, active at different nodes. In a system with a shared memory component, the exchange of inference messages is implemented through the shared memory. A process p_i "sends" its resident ψ by storing a copy of ψ in the shared memory. All the other processes "receive" ψ by reading it from the shared memory. More details on these mechanisms will be given in the following (see Section 4.2.2).

The separation of residents and inference messages is also used to partition the search space at the inference level. Using the paramodulation inference rule as an example of expansion step, we establish that the inference messages are paramodulated *into* the residents, but not vice versa. This restriction has two purposes. First, it distributes the expansion inference steps among the nodes. Second, it prevents a systematic duplication of steps: if this restriction were not in place, then each paramodulation step between two residents ψ_1 of p_1 and ψ_2 of p_2 would be performed twice, once when ψ_1 visits p_2 and once when ψ_2 visits p_1. Other expansion inference rules fit naturally in this pattern (see Section 4.6.3).

While subdividing the expansion steps serves its purpose, it is not productive to subdivide the contraction steps. Since the motivation behind contraction is to keep the data base always at the minimal, we let each node use both residents and inference messages to perform as much contraction as possible. In a contraction-based strategy, an expansion step should be performed only if all the premises are fully reduced, at least with respect to the local data base. To ensure this, we require that each processor keep its residents as reduced as possible. When an inference message is received at a node, it is also subject to contraction. If it has not been deleted, it is used to contract the residents. We allow an inference message to perform expansion only after the message and the local data base are fully contracted.

We recall that a clause newly generated from an expansion step is termed a *raw clause*. Input clauses are also considered as raw clauses. In the presence of contraction rules, a raw clause should not become a resident until it has been fully contracted. Thus, our method also features a number of *distributed global contraction schemes* to reduce a raw clause with respect to the global data base. This implements *forward contraction*. The distributed global contraction schemes are also used to perform *backward contraction*, i.e. to reduce with respect to the global data base a resident which has been reduced locally. After contraction, a raw clause becomes a *new settler*. New settlers are given to the allocation algorithm to be assigned to some node. We do not assume a central control process devoted to execute the allocation algorithm. Every process executes the allocation algorithm for its new settlers: it may decide either to retain a new settler or to send it to another node. The purpose of the allocation algorithm is to partition the search space and keep the work-load balanced as much as possible.

This is the basic working of the Clause-Diffusion methodology: local contraction and local expansion inferences at the nodes among residents and inference messages, distributed global contraction, allocation of new settlers and mechanisms for passing inference messages. By specifying the inference mechanism I, the search plan

Σ to schedule inference steps and communication steps, the allocation algorithm, the distributed contraction scheme and the mechanisms for the communication of messages, one obtains a specific strategy.

4.2 Distributed global contraction

We term *global contraction* the task of reducing a clause with respect to the global data base, i.e. the union of the sets of residents of the deduction processes. We indicate by N the graph representing the distributed system: its nodes are the processors and any two nodes are connected if the corresponding processors are neighbours in the topology of interconnection of the distributed system. We give four schemes for distributed global contraction. These global contraction schemes implement both global forward contraction and global backward contraction, depending on whether the clause being contracted is a raw clause or a resident.

We distinguish between **global contraction by travelling** and **global contraction at the source**. In the first, we assume that no node has access to the global data base and thus global contraction of a raw clause is achieved by having the raw clause travelling to visit all the nodes. In global contraction at the source, we assume that every node has access to an approximation of the global data base, so that a raw clause can be contracted at the node where it was born. Each of two schemes has in turn two variants:

- global contraction by travelling,
- global contraction by travelling with selected simplifiers,
- global contraction at the source by localized image sets and
- global contraction at the source by global image set in shared memory.

For global contraction by travelling, we start by observing that duplicating a raw

clause and giving distinct copies to distinct processes is not convenient for the purpose of global contraction. Let p_1 and p_2 be two processes with their local data bases and ψ_1 and ψ_2 be two copies of a raw clause, one per process. The two copies will be reduced in general to two different forms ψ_1' and ψ_2'. In order to have them reduced with respect to the union of the two data bases, the two processes should exchange them. By doing so, each process will try to apply its simplifiers twice, e.g., p_1 will apply them once on ψ_1 and once on ψ_2'. Furthermore, neither the two local data bases nor their union are confluent in general, and thus normal forms are not unique. The two processes may obtain two different, irreducible forms. We choose then to maintain one single copy of a raw clause throughout the global contraction process.

Then we describe the two main schemes for global contraction by travelling.

- In **global contraction by travelling**, the processor p_i where the raw clause ψ has been generated, turns it into a *message*. We call these messages *travelling raw clauses*. The travelling raw clause goes *sequentially* through all the nodes and it is normalized at each node p_i with respect to the local data base S^i. The main drawback is that after one traversal of N, there is no guarantee that the raw clause is in normal form with respect to the global data base. Then, there are two more variants: *single traversal*, i.e. the travel of the raw clause halts after it has visited each node once, and *multiple traversal*, i.e. the travelling raw clause keeps repeating the traversal until it traverses N once without being modified. Single traversal produces outputs which are likely to be far from normalized. In order to implement properly a contraction-first strategy, we ought to choose multiple traversal. On the other hand, multiple traversal can be intolerably expensive, especially as the number of nodes grows. Therefore, we need a way to limit the cost of multiple traversal.

- Intuitively, if we can make a traversal "more effective", the number of required traversals will be reduced. One way to do so is to select equations that are

deemed to be "most effective simplifiers", e.g. the distributivity axiom. These *selected simplifiers* are made available to all nodes and are applied at each stage of a traversal. Conceivably, the contraction power of each node and thus of an entire traversal is greatly enhanced. We call such a scheme **global contraction by travelling with selected simplifiers**.

In **global contraction at the source**, each node p_i has access to an approximated version of the global data base, called an *image set*. The name "image set" says that such set contains "images", i.e. copies, of residents in the system. We describe **global contraction at the source** in two scenarios, depending on the availability of a shared memory.

- **Global contraction at the source by localized image sets**: we assume that the local memory of each node p_i is large enough to hold an approximated version SH^i of the global data base $\bigcup_{i=1}^{p} S^i$. The SH^i's are called *localized image sets*. Each node p_i uses its localized image set SH^i as set of simplifiers to perform global contraction of residents, raw clauses and incoming messages. The localized image sets can be built by utilizing the inference messages: *whenever a node p_i receives an inference message, it stores the clause carried by the message in SH^i*. The identities $SH^j = \bigcup_{i=1}^{n} S^i$ for all j, $1 \leq j \leq n$, which should hold in principle, are not enforced strictly, because the sets of residents S^i's keep evolving. Thus a localized image set is just an approximation of the global data base. However, each of the SH^i's is logically equivalent to the global data base $\bigcup_{i=1}^{p} S^i$, if all the persistent residents are broadcast as inference messages.

- **Global contraction at the source by global image set in shared memory**: we assume that a shared memory is available and an approximated version SH of the global data base $\bigcup_{i=1}^{p} S^i$ is stored in the shared memory. The set SH is called *global image set*. Each process accesses SH to get the simplifiers to perform, at each node, global contraction of residents, raw clauses

and incoming messages. The global image set SH can be formed as follows: whenever a new settler ψ settles down at a node p_i as a resident, node p_i also takes care of adding ψ to SH in shared memory. Similar to the previous case, the identity $SH = \bigcup_{i=1}^{n} S^i$ is not enforced strictly, so that the global image set is just an approximation of the global data base. However, SH and $\bigcup_{i=1}^{p} S^i$ are logically equivalent, if an image of every persistent resident is included in SH.

A key issue in global contraction at the source is whether and how to perform contraction of the simplifiers in the SH^i's or in SH. Techniques to update with respect to contraction the SH^i's at the nodes or SH in shared memory will be discussed in the following (see Section 4.2.2). One additional advantage of keeping a global image set, either in shared memory or with a copy per node, is that such large sets of simplifiers can be implemented as *discrimination nets* [30, 118] for the purpose of fast simplification.

Our global contraction schemes do not suffer from the backward contraction bottleneck, because in all our schemes *the clauses being rewritten by contraction are held in the local memories of the nodes.* Therefore, concurrent contractions are done independently in the local memories at the nodes, with no need to wait to get write-access to a shared memory. Even when our schemes employ a shared memory, the latter is only used to store clauses used as simplifiers, while the clauses subject to contraction are kept in the individual nodes. Therefore, concurrent contractions can always be done at the nodes without creating any global bottleneck, as long as the architecture supports concurrent reads on the shared memory.

The choice of the appropriate global contraction scheme is related to the available resources: global contraction by travelling requires very fast communication, while global contraction at the source requires either sufficiently large local memories to hold the localized image sets SH^i's or a shared memory to hold the global image set SH.

4.2.1 Global contraction by travelling

The virtual ring

A travelling raw clause needs to go through all the nodes of N. Such traversal can be realized by implementing over the topology of N a *virtual ring topology*, which we represent as a fixed permutation $P = p_{x_1}, p_{x_2}, \ldots p_{x_k}, \ldots p_{x_n}$ of the nodes of N. Since a permutation has no repetitions, all nodes of N occur in P once. A virtual ring topology can be implemented over N either *directly* or *indirectly*. We say that it is implemented directly if each virtual link in P corresponds to a real link in N; it is implemented indirectly if a virtual link in P is realized by a path in N. Clearly, it is possible to implement a ring directly on N, if the graph N admits a *Hamiltonian path*, i.e. a path in the graph where all nodes appear and they appear at most once. The existence of a Hamiltonian path P depends on the topology of N. For instance, the *cube* topology admits trivially Hamiltonian paths. If N admits such paths, one of them is chosen as P. Under this hypothesis, the virtual ring can be implemented by having each processor storing in its memory the *name of the next node* in P: if p_i is p_{x_k}, it stores $p_{x_{k+1}}$.

However, there are graphs which do not have Hamiltonian paths. A simple example is a *binary tree*: after having visited a leaf we have to go back through its parent to reach any other part of the tree. If N is a generic graph without Hamiltonian paths, the virtual ring may be implemented indirectly, as follows. Each node p_i stores a *destinations-neighbours table* DN^i, i.e. a table of entries $< p_j, p_g >$, where p_j ranges over all the nodes of N, whereas p_g ranges over the set of the neighbours of p_i. The meaning of an entry $< p_j, p_g >$ in DN^i is that p_g is the most appropriate among all neighbours of p_i in order to reach the destination p_j from p_i. Thus, if the node next to p_i in P is p_j, p_i consults DN^i, retrieves $< p_j, p_g >$ and sends the message to p_g. The next step will be done similarly by p_g. In this way, even if the ring is implemented indirectly, each node needs to be concerned only with *local* choices, i.e. it simply needs to choose a neighbour. This simplicity is

possible at the cost of storing at each p_i the table DN^i. Furthermore, this indirect implementation of the ring *requires* that messages carry their destination, so that it is possible to distinguish at every node p_j, whether an incoming message has reached p_j because p_j is its destination or because the message is going through p_j in order to reach another node.

Henceforth, we assume that the ring P is available, namely each node knows its successor in P and can forward a message to it, regardless of whether this is realized directly or indirectly. We write *a node p_i forwards a message on P*, meaning that p_i sends the message to its successor in P. The traversal of the virtual ring P is performed as follows. A raw clause ψ, generated at node p_i, is turned into a travelling raw clause, i.e. a message in the form $< \psi, p_i >$, where the second field is the sender of the message, termed the *birth-place* of the raw clause. Node p_i forwards it on P. Any node p_k on P, upon receipt of $< \psi, p_i >$, tests whether $i = k$. If it is true, the process is completed. Otherwise, p_k reduces ψ as much as possible with respect to its data base. Then p_k forwards the result of the normalization on P. If P is implemented indirectly, a travelling critical pair needs to incorporate its current destination, i.e. we need to use a three-fields format $< \psi, p_i, p_j >$, where p_i is the birth-place and p_j is the current destination.

The use of the fixed route P for all the travelling raw clauses, may cause congestion and represent a bottleneck for the entire computation. Such a phenomenon may be contained by balancing the work-load among the processors. If the work-load, e.g. the number of residents, does not vary dramatically between any two nodes, the generations of raw clauses should be distributed rather uniformly over P. Raw clauses generated at different nodes traverse different cyclic permutations of P. Then, the route P should not get too badly congested. Clearly, though, global contraction by travelling relies very heavily on communication, and therefore it may be considered only if communication is extremely fast.

The following three subsections discuss specific topics within the travelling raw

clauses approach.

Single traversal versus multiple traversal

There are two basic modes for the traversal of a travelling raw clause $< \psi, p_i >$: *single traversal* and *multiple traversal*. In single traversal, a travelling raw clause follows the circular route P only *once* and therefore visits each node only once. As soon as $< \psi, p_i >$ is back at its birth-place p_i, the process is terminated and the travelling raw clause is turned into a *new settler*. In multiple traversal, a travelling raw clause keeps circulating on P, until *it has traversed P once without being modified*. This effect is obtained by requiring that *whenever a processor p_j simplifies a travelling raw clause, the birth-place of the travelling raw clause is set to p_j*. In this way a raw clause reduced at p_j is regarded as a raw clause born at p_j. The travel of a raw clause terminates when it reaches again its birth-place. By updating the birth-place wherever reduction occurs, we guarantee that a travelling critical pair travels on P until it has traversed P once without being reduced.

Multiple traversal is clearly much more expensive than single traversal. Single traversal requires exactly n communication steps interleaved with n normalization phases, where n is the number of nodes. Multiple traversal requires at least n communication steps interleaved with n contraction phases, since a final traversal is always required just to check that the raw clause is not reducible. As for effectiveness, neither single traversal nor multiple traversal imply that when a raw clause is turned into a new settler, it is actually in normal form with respect to the current state of the global data base. Such normalization may not be achieved if each processor has only local information. When a travelling raw clause is at p_i, node p_i can ensure that it is normalized with respect to its data base, but it cannot determine whether it is normalized with respect to the data bases at the other nodes. However, it is reasonable to expect that multiple traversal will allow in general to perform much more contraction steps than single traversal, at the cost of a longer process.

Single traversal is faster, but it yields a new settler which is likely to be less reduced than the one output by multiple traversal. Since contraction inferences make clauses smaller, the reduced form ψ_s of ψ produced by single traversal is likely to be "larger" than the reduced form ψ_m of ψ produced by multiple traversal. Intuitively, the larger an equation is, the more raw clauses it may generate. Therefore, ψ_s may be expected to generate more critical pairs than ψ_m. Furthermore, the raw clauses produced by ψ_s are likely to be larger than those produced by ψ_m. In turn, larger raw clauses will require more contraction steps or will yield more, even larger raw clauses. In other words, single traversal saves some work in the short run, but it may induce much more work in the long run.

The selection of the selected simplifiers

The purpose of having *selected simplifiers* in global contraction by travelling is to enhance the contraction power of each node, by granting it access to a common data base of *"most effective simplifiers"*. Therefore, we need criteria to decide which equations can be expected to be especially effective as simplifiers. Such criteria are clearly heuristic in nature as the dynamic of the derivation cannot be predicted. Most heuristics which can be automated are based on the syntax of the equations. For instance, one may regard the so called *collapse equations*, i.e. oriented equations $l \to x$, where the right hand side is a variable, as strong simplifiers. A collapse equation replaces any instance $l\sigma$ of l by the instance $x\sigma$, which is in general a much smaller term, where terms may be compared by the ordering $\succ$ of simplification or by size. More generally, oriented equations $l \to r$, where the right hand side r is "small", are good candidates. Oriented equations $l \to r$ with a small left hand side l may also be considered: intuitively, the smaller the left hand side is, the smaller is the cost of the matching test and the greater is the chance that the matching test succeeds. By similar reasons, unoriented equations $l \simeq r$ such that both sides are sufficiently small may be interesting. How small l or r should be, for $l \to r$ to be considered, is a parameter which can be adjusted experimentally. The threshold

should not be too high, otherwise the number of selected simplifiers would not be significant. It should not be too low either, otherwise too many equations would be selected. In turn, how many equations are "too many" depends on the amount of memory available at each node.

Criteria based on the structure of the equations may not be sufficient. For instance, the *distributivity* axiom, i.e. $x * (y + z) \rightarrow x * y + x * z$, is an important simplifier, which is advisable to share, but its sides may not be sufficiently small. More precisely, if the threshold for considering a term small is lowered to include the sides of distributivity, it may happen that too many terms are regarded as small and too many simplifiers are selected. In other words, it may be desirable to share a specific equation, e.g. distributivity, but not all the equations of similar size. Therefore, the user should be allowed to pick some selected simplifiers, both in the input set of equations and during the derivation. In this way, the knowledge of the theory at hand and empirical observations of the derivations can contribute to the formation of the set of selected simplifiers.

Generation of a data base of selected simplifiers

Once we have ways to decide which simplifiers should be selected, we need a mechanism to build a copy SE^i of the data base of selected simplifiers at each node p_i. In order to analyze such mechanisms, we introduce two notions of *agreement*: we say that a mechanism to generate a selected data base SE guarantees *strong agreement* if for every two nodes p_i and p_j, if p_i adds ψ to SE^i, then p_j eventually adds ψ to SE^j. It guarantees *weak agreement* if for every two nodes p_i and p_j, if p_i adds ψ to SE^i, then p_j eventually adds to SE^j a ψ', which is logically equivalent to ψ in the theory. As we have already remarked in Section 4.1, neither strong nor weak agreement are necessary to preserve the completeness of a theorem proving strategy. The issue at a stake here is *efficiency*. Completeness may be harmed only by some mechanism which inappropriately deletes clauses. The generation of a common data

base does not delete anything, rather it adds copies of clauses. In fact, having a copy of SE at each node is a form of redundancy. The goal is to generate and handle such redundancy in such a way that it increases the efficiency of global contraction of raw clauses. We have defined strong agreement mostly for descriptive purposes, as identity is clearly too strong a form of equivalence for a theorem proving application. Weak agreement, instead, is more significant to the following analysis.

The first mechanism we may consider is **centralized selection with no contraction**. For all nodes p_i and residents ψ in S^i, p_i tests whether ψ satisfies the requirement to be a selected simplifier. Let ψ be a resident for which the test succeeds. Node p_i broadcasts ψ as a message. Messages of this type are called *selected new settlers*, because they carry clauses which have been chosen to be selected simplifiers. Contraction of selected new settlers during their travel is forbidden. As soon as a node p_j receives a selected new settler ψ, it adds it to SE^j. This procedure guarantees strong agreement. However, it has two drawbacks. First, if every resident is already scheduled to be broadcast as an inference message, it may objected that it is a waste to broadcast ψ twice, once as a selected new settler and once as an inference message. Second, forbidding contraction of selected new settlers implies that since its generation the selected data base is not as reduced as it could be. Both these two disadvantages are less significant the higher is the threshold to be elected as selected simplifer: the stricter the requirement is, the fewer selected new settlers will be generated and the less likely is that a clause, which is sufficiently small to be selected, is reducible. In conclusion, centralized selection with no contraction may be reasonable only if the data base of selected simplifiers represents a small fraction of the set of all clauses ever generated during the derivation.

We consider then two selection mechanisms which assume that inference messages are broadcast and exploit the broadcasting of residents as inference messages to build a data base of selected simplifiers. Since inference messages are subject to contraction, we call these two mechanisms **centralized selection with contraction** and **distributed selection with contraction** respectively. In the first one,

node p_i tests whether ψ in S^i satisfies the condition to be selected, *before* broadcasting it as an inference message. If the test succeeds, the inference message for ψ includes the indication that the carried clause should be retained as a selected simplifier. Upon receipt of such a message, each node p_j inserts the clause, either ψ or some reduced form ψ', in SE^j. In distributed selection with contraction, each node acts independently. Each node p_i tests for election to selected simplifier both its own residents and the inference messages it receives. If ψ satisfies the test, p_i adds it to SE^i.

It is plain to see that no selection mechanism which relies on messages subject to can enforce strong agreement, because different nodes may receive different reduced forms of the clause carried by the message. Therefore, we discuss centralized versus distributed selection with contraction with respect to weak agreement. Let ψ be a resident of p_i which is broadcast, reduced as an inference message and received at another node p_j in the reduced form ψ'. Such a sequel of events may induce disagreement between p_i and p_j in two ways: either ψ satisfies the test and ψ' does not or vice versa. The first case can be excluded by requiring that the criterion to decide whether a clause should be a selected simplifier has the following property: if ψ satisfies the test, any reduced form of ψ also does. For instance, for the simplification inference rule, this requirement is easily fulfilled by measuring terms according to the ordering $\succ$ used in simplification itself. If ψ is sufficiently small to be chosen as a selected simplifier, any reduced form ψ' of ψ will be also, since $\psi \succeq \psi'$ by definition of simplification.

We are left with the case where a resident ψ at p_i does not satisfy the test, whereas its reduced form ψ' received at p_j does. Such a situation affects centralized selection versus distributed selection very differently. In centralized selection, the decision is taken once and for all at p_i, based on ψ. Thus, if ψ does not satisfy the test, neither ψ is added to SE^i nor ψ' is added to SE^j. Centralized selection guarantees weak agreement, even in the presence of contraction. The negative side is that ψ' actually deserves to be added to SE^j, but it is not. If ψ has been reduced

as an inference message, it may also be reduced as a resident eventually. If ψ at p_i is reduced to a clause ψ'' identical to or smaller than ψ', then p_i will decide to share ψ''. To summarize, centralized selection guarantees weak agreement, at the cost of possibly delaying the insertion of good simplifiers in the SE^i's. In distributed selection, instead, p_j adds ψ' to SE^j immediately, whereas p_i does not add ψ to SE^i. If, eventually, ψ is also reduced to a sufficiently small clause ψ'', p_i will insert it in SE^i. In the mean time, SE^i and SE^j disagree. Distributed selection represents the opposite trade-off than centralized selection: effective simplifiers are selected eagerly and weak agreement is possibly delayed.

One may wonder whether weak agreement is guaranteed eventually in distributed selection. The answer is negative in general. It may be positive under the hypothesis of *uniform fairness* of the derivation. We shall treat formally uniform fairness of distributed derivations in Section 5.2. We recall from Section 2.7 that a uniformly fair has the power of generating eventually a saturated set of clauses or a confluent set of equations. In other words, the union of all the persistent residents at all the nodes of N form a confluent set. Since the normal form of any term with respect to a confluent set is unique, the normal forms of ψ and ψ' will be unique eventually. If, in addition, all contraction steps are eventually applied, ψ and ψ' will be eventually reduced to the same form and weak agreement is eventually achieved. Such a form of weak agreement is of little practical importance. First, "eventually" means after an arbitrary number of steps. Second, theorem proving derivations are not required to be uniformly fair, because the purpose of the derivation is to prove a specific theorem and not to generate a confluent set. Therefore, if the selected data base is generated by a distributed mechanism with contraction of messages, weak agreement is not guaranteed in general.

4.2.2 Global contraction at the source

Global contraction by travelling may be very inefficient, even in the presence of selected simplifiers. For instance, if the rewrite rules $g(f(a)) \rightarrow h(f(a))$ and $h(f(a)) \rightarrow f(a)$ are allocated at two different nodes and are not selected simplifiers, it takes m traversals to normalize the equation $g^m(f(a)) \simeq a$. The alternative is to provide each node with access to an approximated version of the global data base and adopt **global contraction at the source**. It can be realized either by the *localized image sets* SH^i's or by a global image set SH in shared memory.

A fundamental issue in global contraction at the source, is whether contraction of the simplifiers in the SH^i's (or in SH) should be allowed. The question is whether the advantage of maintaining the SH^i's (SH) fully reduced is worth the cost of updating them. In case of localized image sets, one possibility is to have each p_i performing contraction on SH^i just like on S^i. We call this policy "maintenance by direct contraction". Each node contracts its own raw clauses, but all nodes execute independently contraction of all residents: if ψ is a resident at p_i which is also stored in the SH^j components at the other nodes, contraction of ψ is performed at all nodes which have a copy of ψ. If contraction inferences are sufficiently fast, this may be a reasonable choice. On the other hand, this approach is undesirable in case of SH in shared memory, because it would re-introduce the backward contraction bottleneck.

At the other extreme, another possibility is to forbid contraction on the SH^i's (on SH) altogether. This is the "no contraction" policy. Only insertion of new elements is allowed. If $\psi \in SH^j - S^j$ is reducible, it may be reduced at the node p_i, such that $\psi \in S^i$, and a reduced form of ψ will be added to SH^j eventually. If SH^j is used only as a data base of simplifiers, the presence of both ψ and a reduced form ψ' should not represent serious redundancy. In fact, especially if the SH^j's are implemented as discrimination nets, frequent updates of the elements in the net may not be cost-effective. However, if no element is ever deleted from the

SH^j's, their sizes may grow up to compromise their performances. Furthermore, if the elements in SH^j are used for expansion steps, redundant clauses in SH^j would induce the generation of more redundant clauses. These considerations also apply to the case where we have a global image set SH in shared memory. In order to avoid such phenomena, we may design mechanisms to update the SH^i's (SH) without resorting to the direct application of contraction inferences.

We consider first the shared memory case. We associate to every resident of a node a unique *identifier*: for every node p_i and for every resident ψ of p_i, ψ receives an identifier a, so that a is the unique identifier of ψ within the local data base S^i at p_i. It follows that $< p_i, a >$ is the unique *global identifier* of ψ over N. We also establish that a resident ψ at p_i has another attribute, the *birth-time*, i.e. the time at p_i's clock when ψ was recorded as a resident of p_i. Overall the format of a resident is $< \psi, a, x > \in S^i$, where a is the identifier and x is the birth-time. The global identifiers of the residents can be used to index the clauses in SH. For instance, SH may be implemented as a *hash table*, with the global identifier as key. When a resident $< \psi, a, x > \in S^i$ is deleted or replaced by $< \psi', a, y >$, where $y > x$ is the current time at p_i's clock, node p_i also retrieves index $< p_i, a >$ in SH and deletes ψ or replaces it by ψ' in SH as well. This technique is called "maintenance by direct update". The potential risk of this approach is that it may turn out to mimic too closely the direct application of contraction to the shared memory. The replacement of ψ by ψ' still requires a write-access and if a very high number of such accesses is generated, the conditions for a backward contraction bottleneck in shared memory might develop.

To prevent such a phenomenon, we may adopt a mechanism of "delayed update with garbage collection". When $< \psi, a, x >$ is replaced by $< \psi', a, y >$, $y > x$, in S^i, node p_i does not retrieve and delete ψ in SH, but simply add ψ' to the bucket with key $< p_i, a >$ in the hash table SH. Therefore there is only minimal write-conflict. Each bucket may be organized as a last-in-first-out list, so that the most recently added, and thus the most reduced, element is easily retrieved at the top, to

be used as simplifier. Note that different processes may use as simplifiers different elements from the same bucket, without incurring in a read-write conflict, because all the data with the same key ($< i, a >$) are logically equivalent. Periodically, a garbage collection process visits all the buckets of the hash table and delete all the elements in each bucket except the topmost one. It may be determined empirically how often garbage collection should be executed, keeping into account the amount of shared memory available. The disadvantage of this technique is represented by the drawbacks of garbage collection. Namely, if the garbage collection process monopolizes the access to the shared memory for too long, the deductive processes at the nodes may be forced to suspend, waiting to access the shared memory to get the simplifiers.

Identifiers and birth-times of residents also help in case of localized image sets. We require that an inference message carries a clause together with its global identifier and birth-time. An inference message for a resident $< \psi, a, x > \in S^i$ has the form $< \psi, p_i, a, x >$. These additional fields allow a node to recognize that an inference message is carrying a reduced form of a previously received clause. If $< \psi, a, x >$ is reduced to $< \psi', a, y >$, where $y > x$, at p_i, a new inference message $< \psi', p_i, a, y >$ will be broadcast eventually. The localized image sets can also be implemented as hash tables with the global identifier of the clauses as key. Whenever a node p_j receives an inference message, e.g. $< \psi', p_i, a, y >$, it checks whether an element ψ with the same global identifier $< p_i, a >$ is stored in SH^j. If this is the case, node p_j compares ψ and ψ' according to the ordering on clauses and saves the smallest in SH^j. If the two clauses are not comparable, the one with most recent birth-time is saved. We call this solution "update by inference messages" and we shall formalize the comparison between messages as an inference rule in Section 5.3.1. Such an immediate replacement is much more reasonable in the distributed memory configuration than in the one with shared memory, because an access to SH^j is just an access to the local memory of p_j.

In case of update by inference messages, the situation is less favorable if $<$

$\psi, a, x > \in S^i$ is deleted, rather than replaced, by a contraction step. In such case, no more messages with identifier $< p_i, a >$ will be issued and therefore, localized image sets may never be updated. However, inference messages may still help: whenever an inference message $< \psi, p_i, a, x >$ is deleted at a node p_k, it is possible to check whether SH^k contains any clause with identifier $< p_i, a >$ and delete it. This is not sufficient in general to update all the localized image sets, because clause ψ may not be deleted at p_k. Then, if performance is hindered by not updating the localized image sets with respect to deletions, one may consider broadcasting a special message with identifier $< p_i, a >$ to inform all the nodes that the resident at $< p_i, a >$ has been deleted.

4.2.3 Discussion of distributed global contraction schemes

All our schemes for distributed global contraction fall into three basic classes: *global contraction by travelling*, *global contraction at the source with localized image sets* and *global contraction at the source with shared memory.*

Global contraction by travelling was our first approach to the problem. It originated from the consideration that in a contraction-oriented strategy a clause is subject to forward contraction before being used for expansion. We emphasized the distinction between these two stages, forward contraction and expansion. By viewing the problem in terms of the "life" of a clause, we thought of a clause as being first a raw clause, then a new settler and finally a resident. As a raw clause, it undergoes forward contraction and it is not used for expansion. As a resident, it is used for expansion and to reduce successively generated raw clauses. The classification of clauses depending on whether they are allowed to expand seemed a key concept. Accordingly, we decided that raw clauses would travel, while clauses allowed to expand would be settled as residents. The rationale was that it is wasteful to allocate a raw clause, since its life-time may be very short, as it may be deleted by forward contraction. This consideration, however, soon turned against the idea

of global contraction by travelling. We realized that communication is probably bounded to be the most expensive activity in a distributed system. Exactly because raw clauses are very short-lived on average, it is not cost-effective to generate and transmit messages for them. Also, it was remarked to us [83] that the number of raw clauses is likely to be much higher than the number of residents at any stage of a typical contraction-oriented derivation. Indeed the vast majority of raw clauses are deleted during forward contraction. Therefore, if raw clauses are messages, the overhead of communication will be most likely too high.

These reflections led us to seek an alternative to the idea of travelling raw clauses. In order to realize global forward contraction without travelling, i.e. at the source, one needs to give access to some approximation of the global data base to each node. Thus, our second solution was global contraction at the source with localized image sets. We thought that since a node receives the residents of the other nodes as inference messages for the purpose of expansion, it may as well save them and use them as simplifiers. Discarding the inference messages after one usage seemed an unreasonable waste, since so much communication effort had been done to diffuse them in the first place.

The next turning point was the understanding that the key issue is backward contraction, not the dichotomy between a forward contraction stage and an expansion stage. We saw that the most difficult requirement in the parallelization of a contraction-oriented strategy is not to make sure that a clause is forward-contracted before expanding. The most challenging requirement is to maintain the residents reduced, and especially, whenever a resident is backward-contracted, to be able to test it for further contraction with respect to the global data base. In terms of the metaphor of the life of a clause, it turns out that a clause is never definitely settled, because a backward contraction step may contract a resident at any time.

This development has many consequences. First, backward contraction induces inference messages, because a backward-contracted resident needs to be re-issued

eventually. It appeared that the number of inference messages was going to be probably higher than expected. In other words, even after having renounced travelling raw clauses, the overhead in communication due to inference messages alone was still worrisome. Second, backward contraction increases the cost of saving received inference messages, because the localized image sets need to be updated with respect to backward contraction. If the clauses carried by inference messages were guaranteed to be persistent, the advantage of saving them would certainly exceed the cost. However, it is not so. Forming a localized image set at each node gives to each process the capability of utilizing an approximation of the global data base as set of simplifiers. But it also gives the burden of maintaining n images of the global data base reduced. Third, the usage of localized image sets might make more difficult to prevent the derivations constructed by the different processes from overlapping. Intuitively, the deductive processes end up gathering the same data, i.e. the same clauses in their localized image sets. This phenomenon may challenge the possibility of partitioning the search space effectively by partitioning the clauses. It may become more relevant to differentiate the derivations by explicitly differentiating the search plans executed at distinct nodes. Thus, while the travelling raw clauses approach suffers from too much communication, the localized image sets approach seems to suffer from too much duplication of data.

As a third solution, we resorted to global contraction at the source with shared memory. It reduces duplication because there is just one image set rather than n sets. It reduces communication because it is possible to implement send/receive of inference messages as read/write in shared memory. The negative side is that the shared memory may become a bottleneck. This may happen in two ways. First, the shared memory might become a bottleneck for backward contraction, as in the purely shared memory approaches that we have surveyed. Our solution is to associate different functions to different copies of a clause. The copy of a clause φ in shared memory, i.e. the "image" of φ, is used mostly as simplifier, while possible backward contraction steps re-write the copy of φ at a node, i.e. φ as a resident. Images in shared memory are updated with some delay. Second, the

shared memory may become a bottleneck of communication, as all the processes try to access it to read/write inference messages. In order to avoid this, we propose to use both communication via message-passing and communication through shared memory. It is not necessary that all the inference messages are exchanged through the shared memory. For instance, we may establish that the first issue of a resident as inference message is broadcast. Then, if a resident is backward-contracted, rather than broadcasting a new inference message, we update the image of the resident in shared memory and we make sure that the other processes are aware of the update. If it appears that too much traffic goes through the shared memory, it may be better to use message-passing more intensively, e.g. by establishing that the first k updates of a resident are diffused by broadcasting. The appropriate trade-off between communication through shared memory and communication via message-passing, e.g. a good value for the above parameter k, needs to be determined empirically for different architectures, e.g. different sizes and speeds of the shared memory.

In summary, global contraction by travelling is a *communication-oriented* approach, which may be adopted when fast communication is available, the local memories at the nodes are small and there is no shared memory. Global contraction at the source with localized image sets is the *duplication-oriented* approach, which we have implemented in *Aquarius*. Global contraction at the source with shared memory is a mixed approach, developed based on criticism of the first two solutions and on the empirical observations of our implementation.

4.3 Guidelines for the schedule of the operations at a node

In this section we outline some basic *priorities* between classes of operations at a node, for a contraction-based strategy. The schedule of the operations at a node

should first of all be *fair*, i.e. ensure that no needed step is postponed indefinitely. We shall concern ourselves with fairness in our formal treatment of distributed derivations (Section 5.2). Provided fairness is preserved, the following priorities may be recommended as guidelines. The general philosophy is that the operations which *reduce* the amount of data stored at the node have higher priority than those which *increase* it. The only exception to this rule is represented by the operation of *receiving* a message from the outside, since such an operation is determined by an external event which the node services as soon as possible. At a very high abstraction level, the schedule of a node is:

1. **Receive** messages from the outside.
2. **Contraction inferences**.
3. **Communication**:
   (a) **Settle** new settlers,
   (b) **Forward** already existing messages and
   (c) **Send** new messages.
4. **Expansion inferences**.

In a contraction-first strategy, *contraction* has higher priority than *expansion* and *communication*. The goal of this rule is to make sure that each process uses as parents in expansion steps and sends/forwards to other processes clauses which have been reduced as much as possible according to the global contraction scheme. Furthermore, contraction steps may delete clauses, eliminating the need of communication steps for them. Among *contraction* steps, a good rule is to give forward contraction priority over backward contraction. It is known since [91] that performing forward subsumption before backward subsumption is sufficient to prevent certain violations of fairness, which may result from the uncontrolled application of backward subsumption of variants. This issue becomes significantly more complicated in distributed deduction and we shall study it in detail in Section 5.4.

Communication steps have higher priority than *expansion* inferences, in order to enhance parallelism. For instance, if a processor does not forward an inference message until after its expansion phase, the broadcasting process may be delayed too severely. Among the *communication* steps, *Forward* has higher priority than *Send*, since the former lets already existing messages proceed, whereas the latter creates new ones. Also, it may be preferable to *forward/send* first travelling raw clauses, next new settlers and finally inference messages, on the ground that travelling raw clauses may be engaged in a longer travel than the other messages. Furthermore, raw clauses are subject only to contraction steps, whereas new settlers and inference messages are already able to or getting close to be able to expand. When sorting *expansion* steps, it may be convenient to perform paramodulation of inference messages into residents rather eagerly, since after this phase the inference message can be discarded.

In the following sections we consider policies for allocation of new settlers, routing/broadcasting of messages and the inferences within each process.

4.4 Policies for the distribution of new settlers

After a raw clause ψ born at p_i has undergone the global contraction phase, it needs to be allocated at a node as a resident. Node p_i computes the *allocation algorithm* to determine a *final destination* p_q and it creates the *new settler* $< \psi, p_q >$. Several allocation policies can be designed. Since the assignment of clauses to processors determines the work-load at the processors, one would like the allocation policy to distribute the clauses as evenly as possible, so that the work-load is well-balanced. On the other hand, deciding the allocation of clauses to nodes is part of the overhead of working in a distributed environment. An allocation algorithm which diffuses clauses evenly but is very time-consuming may not be reasonable, because the advantage of maintaining the work-load balanced may be outdone by the cost of computing a complex allocation algorithm for each and every new settler.

A *best-fit* allocation algorithm identifies a node p_q where the number of residents is minimum[1] and sends the new settler to p_q. A best-fit policy has the advantage of ensuring a well-balanced work-load among the processors. However, such a result is obtained at the cost of executing a *distributed selection* algorithm to select a node with the minimum number of residents. For this purpose, one may assume that each processor p_i maintains a counter w^i for the number of its residents. Whenever a resident is deleted from S^i or inserted in S^i, w^i is decremented or incremented.

If global contraction is performed by travelling, the distributed selection algorithm can be computed during the contraction trip of the travelling raw clause. This requires to add to a travelling raw clause an additional field, called *final destination.* The value of this field is the identifier of the processor which is the best candidate for allocation found so far. A travelling raw clause has then the form $< \psi, p_i, (p_q, w^q) >$, where ψ is the clause, p_i is its birth-place, p_q is its final destination and w^q is the number of residents at p_q. When the travelling raw clause is first created at p_i, its last field is initialized with the values for p_i itself, i.e. (p_i, w^i). Whenever the raw clause ψ visits a processor p_k on route P, it compares w^q with w^k. If $w^k < w^q$, the final destination field in the travelling raw clause is set to (p_k, w^k). When the global contraction process is completed, the travelling raw clause is back at its birth-place p_i and the value p_q of its last field indicates where it should go. Thus, p_i generates the new settler $< \psi, p_q >$.

If global contraction at the source in shared memory is adopted, we may require that the work-loads w^i of all nodes are stored in the shared memory. Each p_i is responsible for periodically updating the value of w^i in shared memory. If there is no shared memory and the nodes have the localized global data bases SH^i's, each node p_j may compute and keep in memory an estimate of the work-load w^i of any other node p_i, by counting the inference messages it receives from p_i. Then, when a processor p_j needs to determine the final destination of a new settler, it computes

[1] Without loss of generality, we assume that the work-load is measured by the number of residents. More refined measures, such as measures involving the size of clauses, may be adopted.

the minimum w^q among all w^i's and generates the new settler $< \psi, p_q >$.

Under an *alternate-fit* policy, each node p_i saves the most recently used destination, i.e. the identifier q of the node p_q, where p_i sent its most recent new settler. When p_i needs to determine the destination for the next new settler, p_i picks $(q+1)$ *mod* n, where n is the total number of processors. A variant of alternate-fit, which we call *half-alternate-fit*, works as follows: each node p_i saves the two most recently used destinations q_1 and q_2, where q_1 is the most recent one. If q_1 is p_i itself, p_i chooses (q_2+1) *mod* p. Otherwise, i.e. $q_1 \neq p_i$, the new settler is allocated to p_i itself. Then q_2 and q_1 are updated. In other words, half-alternate-fit consists in applying alternate-fit to every other new settler and keep the remaining ones. Switching from alternate-fit to half-alternate-fit may be useful when doing experiments, if one observes that the nodes would profit from keeping for themselves more of their output. If we push this idea to the extreme, we obtain a *first-fit* allocation policy, according to which the raw clauses become residents at their birth-places. If first-fit allocation is applied to the input clauses, all input clauses and all their descendants, i.e. all the clauses ever generated during the derivation, become residents at the node which read the input, so that the distributed derivation collapses onto a sequential derivation. The alternate-fit and first-fit policies may be combined in the *alternate-first-fit* policy, where the input clauses are distributed according to the alternate-fit policy and all successively generated clauses are assigned to the nodes where they are generated. More generally, the nodes may switch back and forth between alternate-fit and first-fit. We may set a threshold m: after using alternate-fit for m times, a node selects the first-fit mode and after using first-fit m times it switches back to alternate-fit.

Another solution, which may be considered in case of global contraction by travelling, is to allocate the raw clause at the node next to its birth-place on route P. We call such policy a *next-fit* algorithm. A next-fit policy does not guarantee a balanced work-load among the processors at all the times. If a node p_i generates many raw clauses, its successor p_l on P will receive many residents. Then, we may

expect that p_l will in turn generate many raw clauses, since it has many residents. Thus, p_l's successor will receive them next and so on. Eventually, all the members of the ring P may be evenly loaded.

Among these policies, the best-fit algorithm has the advantage of being *adaptive*, as it takes into account how the work-loads at the nodes evolve during the computation. However, simple policies such as alternate-fit may fare better in practice than more refined allocation algorithms, which guarantee an even distribution of clauses at the cost of an higher overhead. Complicated allocation algorithms seem to be worthy in applications where there are chances that some processors be idle for a long time, while others are overwhelmed with work. In theorem proving work is anything but scarce. According to observations of experiments such as those reported in [31], it seems unlikely that processors be idle in distributed theorem proving. Thus, simple allocation algorithms may be considered. It is very useful in general to implement many allocation policies, so that it is possible to experiment with them on different problems.

4.5 Routing and broadcasting

After a new settler $< \psi, p_q >$ has been generated, it needs to reach its final destination p_q. The communication mode consisting of transmitting a message from a source node to a specified destination node is called *routing*. For inference messages we consider also *broadcasting*, i.e. routing from a source node to all the other nodes in N. We describe first routing and then broadcasting.

4.5.1 Routing

The virtual ring used for travelling raw clauses can also be used to route new settlers. Routing on the ring requires that the message m to be routed carries its destination

p_j and new settlers satisfy this requirement. The source forwards m on P. Any node p_k on P, which receives m, tests whether $k = j$. If it is true, routing is completed. Otherwise, p_k forwards m on P. Since all nodes occur in P, m is guaranteed to reach its destination in at most n steps, where n is the total number of nodes. Routing by means of the virtual ring should be considered only if no better routing algorithm is known for the topology of N.

We assume now that N is a q-dimensional cube, with $n = 2^q$ nodes and we describe a routing algorithm given in [1]. The cube is regarded as a $2^r \times 2^s$ array, where $r + s = q$. Thus there are 2^r rows numbered $0 \ldots 2^r - 1$ and 2^s columns numbered $0 \ldots 2^s - 1$. The array is indexed in *row-major order*: processor p_i, for $0 \leq i \leq 2^q$, occupies position $[j, k]$, such that $i = j \cdot 2^s + k + 1$, for $0 \leq j \leq 2^r - 1$ and $0 \leq k \leq 2^s - 1$. For instance, if $q = 5$ and there are 32 nodes, we may have $r = 2$ and $s = 3$, so that the cube is regarded as a 4×8 array, with nodes indexed $0 \ldots 7$ in the first row, nodes indexed $8 \ldots 15$ in the second row and so on:

00000	00001	00010	00011	00100	00101	00110	00111
01000	01001	01010	01011	01100	01101	01110	01111
10000	10001	10010	10011	10100	10101	10110	10111
11000	11001	11010	11011	11100	11101	11110	11111

If we consider the binary representations of the indexes of a node, made of $r + s$ bits, we observe that the r most significant bits give the number of the row and the s least significant bits give the number of the column. For instance, in the first row of the above 4×8 array, all indexes have 00 as the two most significant bits, indicating that the row is row 0, while the three lest significant bits range from 000 through 111, indicating the columns. Symmetrically, in the first column the

three least significant bits are set to 0 to indicate this is column 0, while the two most significant bits range from 00 through 11, indicating the rows. In an $r + s$-dimensional cube, each processor p_i has $r + s$ neighbours. The index of each of these neighbours differs from the index of p_i at one of the $r + s$ bits of the binary representation. For instance, node 00000 has neighbours 00001, 00010, 00100, 01000 and 10000, marked by a dot in the above picture. If we regard the cube as an array, each processor has r column-neighbours, i.e. r neighbours on its same column, and s row-neighbours, i.e. neighbours on its same row. For node 0, nodes 1, 2 and 4 are row-neighbours, whereas 8 and 16 are column-neighbours. In general, let $i^{(b)}$ denote the index whose binary representation differs from that of i only at position b, i.e. at the b-th least significant bit. Then the column-neighbours of p_i are the nodes $p_{i^{(b)}}$, where $s \leq b \leq s+r-1$, and the row-neighbours of p_i are the nodes $p_{i^{(b)}}$, where $0 \leq b \leq s-1$.

Having gathered all the elements of the description, we can see how the routing algorithm works. Let p_i be the source and p_j be the destination of a message m. The algorithm compares the binary representations of i and j, reading them right to left, i.e. starting with the least significant bit. Let b be the least significant bit where i and j differ. Then p_i sends m to its neighbour p_k where $k = i^{(b)}$. Upon receipt of the message, p_k checks whether $k = j$: if $k = j$, then the message has reached its destination. Otherwise, p_k proceeds as above: it compares the binary representations of k and j from right to left and sends m to its neighbour $p_{i^{(c)}}$, where $c > b$ is the least significant disagreeing bit between k and j. The same procedure is followed at each node, until the destination is reached. The algorithm takes at most $q = logn$ steps, as there are at most q bits to be switched. Since the binary representations are read right to left, m moves first along p_i's row until it intersects p_j's column and then along p_j's column until it hits p_j. For instance, a message routed by this algorithm from node 0 = 00000 to node 22 = 10110, goes through nodes 2 = 00010 and 6 = 00110, still on row 00, and then down to 22 = 10110 on column 110:

00000	00001	00010	00011	00100	00101	00110	00111
01000	01001	01010	01011	01100	01101	01110	01111
10000	10001	10010	10011	10100	10101	10110	10111
11000	11001	11010	11011	11100	11101	11110	11111

We call this routing algorithm on the cube *row-first routing*. Symmetrically, we have *column-first routing*, if the binary representations are compared left to right, so that the message moves first along p_i's column and then along p_j's row.

4.5.2 Broadcasting

The routing algorithm described in the previous subsection can be easily extended to broadcasting [1]. First, a message m originated at p_i can be broadcast in s steps to all the elements in the same row as p_i. At step 0, p_i starts the process by sending m to its neighbour p_j with $j = i^{(0)}$. At step 1, p_i and p_j send m to $p_{i^{(1)}}$ and $p_{j^{(1)}}$ respectively. At step b, each node p_k which has already m sends it to $p_{k^{(b)}}$. The algorithm proceeds in this way for s steps, one for each bit in $0 \leq b \leq s-1$. At every step the number of elements in the row which have received m doubles. Therefore, it takes s steps to reach 2^s nodes, i.e. all the elements in the row. Symmetrically, m can be broadcast in r steps on the column of p_i, by using the bits $s \leq b \leq s+r-1$. In order to broadcast m to the entire N, we broadcast first m on the row (or on the column) of its source p_i and then in parallel on all the columns (on all the rows), starting from the nodes in p_i's row (in p_i's column). The entire process takes $s+r$ steps, i.e. $logn$ steps.

For simplicity, we have described this broadcasting algorithm as a synchronous parallel algorithm. In an asynchronous system, like those we are considering, the

algorithm will be executed asynchronously. In other words, there is no guarantee that at stage b of broadcasting on a row, all nodes p_k's which have received the message so far send it simultaneously to their respective neighbours of indexes $k^{(b)}$. The nodes simply execute independently the above algorithm and broadcasting is eventually achieved. In the asynchronous version the broadcasting process takes at most $logn$ communication steps, as a message needs at most $logn$ hops, but not necessarily $logn$ time units, since the nodes may freely interleave other activities with the communication steps.

A possible technique for broadcasting of inference messages on a generic topology is *flooding* [119]. The source of the message forwards it to all its neighbours. Then, any node, upon receipt of the message, forwards it to all its neighbours, except the one where the message came from. Flooding is trivially an *optimal* broadcasting method: since all paths are pursued in parallel, for all nodes p_j a copy of the message reaches p_j by going through the shortest path between the source p_i and p_j. We say that a broadcasting algorithm generates *duplicates*, if it may cause a node to receive more than one copy of the same message. Pure flooding is irrealistic, because it generates infinitely many duplicates. A well-known technique to induce termination of flooding is based on the knowledge of the distance a message needs to travel [119]. If p_i is the source and $dist(p_i, p_j)$ is the length of the shortest path between p_i and p_j, then $D = max_{1 \leq j \leq p, j \neq i}\{dist(p_i, p_j)\}$ is the maximum distance that the message m needs to travel. Then, we attach to m a counter c, initialized to D, and we require that whenever a node receives m with counter c, it forwards a copy of m with counter $c - 1$. In other words the counter is decremented at each hop. A message whose counter is 0 is discarded. Since D is finite, this adjustment guarantees the termination of the flooding process, although there is still a finite number of duplicates to deal with.

Duplicates are especially undesirable in a theorem proving application, since it is already in the nature of the theorem proving problem to generate many clauses which are not necessary to obtain a proof. The broadcasting algorithm available on

a cube achieves the same performance as flooding without producing any duplicates. On the other hand, if the topology of N is less favorable to broadcasting than the cube, one may have to resort to a broadcasting algorithm which creates duplicates.

4.6 Inferences on residents and inference messages

When a new settler $< \psi, p_i >$ has reached its final destination p_i, it settles down as a resident at p_i. Each resident is assigned two attributes: an *identifier* and a *birth-time*. A clause ψ is given an identifier a never used before at p_i, so that a is the unique identifier of ψ within the local data base at p_i and $< p_i, a >$ is the unique identifier of ψ within the global data base. The birth-time of ψ is *the current time at p_i's clock, when ψ settles as resident at p_i*. The format of a resident is then $< \psi, a, x >$, where a is the identifier and x is the birth-time. Identifier and birth-time are employed to keep track of the modifications of residents, as we shall see shortly. A node p_i progressively accumulates its set of residents S^i. Node p_i also receives inference messages, in the form $< \varphi, p_j, b, y >$, for a resident $< \varphi, b, y >$ at p_j, where the birth-time of the resident is given as time-stamp to the message. We denote by M^i the set of the inference messages currently held at p_i. Therefore, a node p_i has typically a set of residents S^i, a set of inference messages M^i, possibly a set SE^i of selected simplifiers or a localized image set SH^i.

4.6.1 Contraction

Contraction has been already largely covered by the description of the schemes for distributed global contraction. In addition, there may be local contraction steps, which involve only the local data base at each node. The distinction between global and local contraction depends in fact on the assumed global contraction scheme. In case of global contraction at the source with localized image sets, an approximation of the global data base is stored in the local data base at each node. Therefore, there

is no difference between global and local contraction steps, because any contraction task performed at a node may be done with respect to an approximation of the global data base. In case of global contraction at the source in shared memory, instead, there is a distinction, because global contraction requires to access the shared memory, whereas local contraction does not. Similarly, in case of global contraction by travelling, global contraction utilizes messages, whereas local contraction does not.

An example of local contraction task is the normalization of a received new settler with respect to S^i. This is *local forward contraction*. In turn, the new settler is applied to reduce the residents and this is *local backward contraction*. Similar treatment may be applied to a received inference message. Local forward contraction should be performed *before* local backward contraction.

The amount of local contraction depends on the effectiveness of the global contraction scheme. In case of global contraction at the source, it may not be necessary to normalize a received inference message or an incoming new settler, because it can be assumed that such clauses have been normalized with respect to an approximation of the global data base at the sender. In other words, the more effective is the global contraction scheme, the less stringent is the need of local forward contraction. However, it is important to perform local backward contraction. If a resident is modified by a local backward contraction step, it should be re-scheduled for global contraction. This provision can be implemented by requiring that if a resident $< \varphi, a, x >$ is reduced to φ', *its birth-time is reset to the current time at* p_i*'s clock*, i.e. $< \varphi', a, y >$ replaces $< \varphi, a, x >$ in S^i, where y is the current time. In this way the resident φ' is regarded as a raw clause, which needs to undergo global contraction. Note, however, that *a reduced resident remains a resident*, i.e. it is treated as a raw clause as far as global contraction goes, but it is not re-allocated. The identifier remains unchanged, since φ', a contracted form of φ, is logically equivalent to φ. If a resident is deleted, e.g. by subsumption or because it has been reduced to a trivial equation, both its identifier and birth-time are erased. In the special case of rewrite

rules, i.e. oriented equations in the form $l \to r$, it is possible to stipulate that the birth-time is updated only when the left hand side is modified, since a reduction on the right hand side does not induce new inference steps.

4.6.2 Communication

A resident $< \psi, a, x >$ at p_i needs to be emitted eventually as an inference message. If a resident $< \psi, a, x >$ is replaced by $< \psi', a, y >$, $y > x$, by a contraction step, the resident with identifier a should be re-scheduled for broadcasting eventually. Intuitively, ψ' is a different clause and thus it deserves to be sent again. We shall see in Section 5.2 that this is a policy which may be adopted to fulfill the requirement of *fairness* of a distributed derivation. It follows that more than one inference message may be issued for the resident with a certain identifier a at p_i during a derivation. Updating the birth-times of reduced residents, as we have seen in the previous section, allows to know for which residents the inference messages need to be repeated.

Nodes receiving inference messages forward them according to the broadcasting algorithm. However, we may resolve that if the clause carried by an inference message is reduced, the message is deleted and not forwarded further. For instance, if node p_j receives $< \psi, p_i, a, x >$ and reduces ψ to ψ', p_j may simply delete $< \psi, p_i, a, x >$, rather than delete it and forward $< \psi', p_i, a, x >$. This decision is taken on the ground that, since ψ is reducible, it will be reduced at p_i to some ψ'', possibly $\psi'' = \psi'$, and p_i will emit $< \psi'', p_i, a, y >$. The choice of forwarding an inference message even if it has been reduced has the advantage of diffusing a reduced clause as soon as possible. The disadvantage is the higher level of redundancy represented by the circulation of inference messages carrying different reduced forms of the same clause. On the other hand, the choice of not to forward reduced messages induces less redundancy in the short term, at the cost of slowing down the distribution of potentially useful clauses.

Because of the combination of contraction and communication, distinct inference messages, e.g. $m =< \psi, p_i, a, x >$ and $m' =< \psi', p_i, a, y >$, with the same global identifier $< p_i, a >$ but different clauses and/or different time-stamps may be circulating. Two ways of generating them are the following. One possibility is that the broadcasting algorithm creates duplicates, whose clauses are reduced to two different forms by contraction along different paths. Another possibility is that a resident $< \psi, a, x >$ at p_i issues an inference message $< \psi, p_i, a, x >$, then the resident is reduced to $< \psi', a, y >$ and it emits another message $< \psi', p_i, a, y >$. We call such messages *generalized duplicates*. If a node p_j receives two generalized duplicates, one of them is redundant: the two messages carry clauses which are logically equivalent and thus p_j should not use them both for inferences. Generalized duplicates may be deleted by contraction of the clauses they carry. In addition, we shall give in Section 5.3.1 specific inference rules to detect and delete generalized duplicates based on the other fields in the messages.

4.6.3 Expansion

After contraction and communication, we consider expansion inferences. A node p_i executes two types of expansion inferences: expansion steps between two residents and expansion steps where a clause from an inference message paramodulates into a resident. An inference message paramodulates into a resident, but not vice versa. This restriction, which distributes expansion inferences among the nodes and prevents duplications of expansion steps, can be applied to expansion inference rules other than paramodulation as follows. For binary resolution, we say that the clause which provides the positive literal resolved upon "paramodulates into" the clause which provides the negative literal resolved upon. For hyperresolution, negative hyperresolution and unit-resulting resolution, we say that the satellites "paramodulate into" the nucleus. Thus, negative literals of inference messages are not resolved upon and inference messages serve as satellites, not as nuclei. For expansion inference rules with just one premise, such as factoring, one may establish that each

node applies the rule to its residents only.

This distribution of expansion inferences, however, causes a problem with respect to the requirement that all expansion steps between persistent residents are considered (this property is part of the *uniform fairness* of a derivation, as we shall see formally in Section 5.2). A node p_j sends an inference message for every one of its residents, unless the resident is contracted beforehand. If a resident φ is reduced to some φ' at p_j, after having been sent, a new inference message for φ' will be generated eventually. Let φ be a *persistent* resident: φ becomes a resident with identifier a and birth-time y at p_j, and it is never reduced afterwards. As part of its treatment as a resident, φ is broadcast as an inference message $< \varphi, p_j, a, y >$. Let ψ be an equation which becomes a resident at some other node p_i , *after* p_i has received, processed and discarded the inference message with φ. It is guaranteed that paramodulation of ψ into φ will be tried, when ψ is sent as inference message by p_i and reaches p_j. However, φ will not have a chance to paramodulate into ψ, because a message for φ has been already generated and consumed. Since φ is persistent, it will not be simplified and thus it will never be the case that an inference message for a reduced form of φ is issued.

This problem may be solved in different ways depending on whether the processes have access to some approximation of the global data base. In they do, regardless of whether the nodes can access SH in shared memory or the SH^i's at the nodes themselves, the solution is immediate: the persistent resident φ which is missing in the above scenario is stored eventually in all the SH^i's (or in SH). Thus, it is sufficient to allow the clauses in $SH^i - S^i$ (or $SH - S^i$) to paramodulate into the clauses in S^i at each node p_i. If the processes do not have access to an approximation of the global data base, we may address the problem by introducing a type of control message, termed *wake-up calls*. The purpose of a wake-up call directed to a node p_j is to induce node p_j to issue again an inference message for one of its residents. For instance, continuing the above example, node p_i needs a new inference message from node p_j, containing the persistent resident φ. For this purpose, p_i sends a

wake-up call $< p_i, y, p_j, a >$, meaning that p_i asks to see the resident with identifier a in processor p_j. In addition, the wake-up call carries the time-stamp y of the last inference message from $< p_j, a >$ received at p_i. We shall treat wake-up calls in detail in Section 4.7. In particular, we shall see that transmitting with the wake-up call the time-stamp of the last inference message allows us to delete some redundant wake-up calls.

Finally, we observe that it is not desirable to reply to a wake-up call by broadcasting an inference message with the requested resident φ. This would force *all* nodes to repeat the inferences with φ, inducing many redundant inferences. We introduce instead a special type of inference message, a *five-field inference message* $< \varphi, p_j, a, y, p_i >$ destined only to p_i. Similarly, trying to solve the problem, without resorting to wake-up calls, by requiring that all nodes send periodically the inference messages for all their residents, is not acceptable. It would lead the processors to repeat inference steps which they have already performed, as a processor does not have a way to detect whether it has already received an inference message with the same clause.

4.6.4 Inferences on the target theorem

For those strategies which separate the target theorem from the other clauses, it is important that the target is available to all deduction processes. Completion-based strategies for equational logic are an example of such strategies. In these methods a successful derivation involves target inference steps. If a process does not have knowledge of the target, it cannot obtain a refutation. Therefore, it is important to have the target at every processor. The input target is broadcast, so that all the nodes start with a copy of the input target. The nodes derive different targets, which form a disjunction $\varphi^1 \vee \ldots \vee \varphi^n$ logically. It is sufficient that one of the nodes derives *true* to terminate the derivation with success. The treatment of the target theorem at a node is similar to that of any other resident: the target receives an

identifier and a birth-time and the inferences performed at the node include the normalization of the target. Identifier and birth-time are updated for the target just like for a resident.

In the previous section we considered the problem of ensuring that expansion steps between remote persistent residents be considered. An analogous problem is to make sure that all persistent equations be applied to simplify the target. In a sequential derivation, there is just one data base, and therefore it is simple to satisfy this requirement. The problem is more involved when the data base is distributed. If ψ_1 is a persistent resident at p_i and ψ_2 is a persistent resident at p_j, ψ_1 is applied to the target φ^i at p_i and ψ_2 is applied to the target φ^j at p_j. We need that both ψ_1 and ψ_2 are applied to at least one of φ^i and φ^j. More generally, it is necessary that at least one of the φ^i's in $\varphi^1 \vee \ldots \vee \varphi^n$ is subject to simplification by all the persistent residents. If the nodes have access to an approximation of the global data base, the situation is close to the sequential case. It is sufficient to ensure that SH (or the SH^i's) receive eventually all the persistent residents and all the persistent elements in SH (or in the SH^i's) are applied to the target. If no node has access to an approximation of the global data base, we may consider other techniques. One possibility is to issue *wake-up calls* on behalf of targets, just like it is done for residents. The definition of wake-up call is the same as illustrated in Section 4.6.3 and the detailed treatment of such messages, regardless of whether they are issued on behalf of residents or targets, will be given in Section 4.7. Another possibility is to introduce *target inference messages*: if φ^i is reduced and thus its birth-time is updated, p_i will eventually issue an inference message for its target, just like it would for any of its residents. The treatment of target inference messages is similar to that of plain inference messages.

4.7 Wake-up calls

In this section we assume that no node has access to an approximation of the global data base. Global contraction is done by travelling and wake-up calls are used to guarantee that all expansion steps are considered eventually. We describe first the generation and routing of wake-up calls and five-field inference messages and then techniques to detect and delete redundant wake-up calls.

4.7.1 Policies for the generation of wake-up calls

In order to decide whether to send wake-up calls, p_i needs to know when the residents at the other nodes last visited its own residents. For this purpose, we may establish that each node p_i maintains a data structure, called the *last-seen table* and denoted by LS^i. LS^i is a table of entries $< p_j, a, x, y >$, where $< p_j, a >$ is a global identifier, x and y are time-stamps. An entry $< p_j, a, x, y >$ in LS^i means that x is the time at which an inference message m from $< p_j, a >$ was last processed at p_i. The time-stamp y which was carried by m is also recorded. The time-stamp x was taken at p_i's clock, while the time-stamp y was taken at p_j's clock. The table LS^i may be implemented as an hash table with the nodes' identifiers as key: the entry $LS^i[j]$ stores all the entries $< p_j, a, x, y >$ relative to the residents of p_j. The entries in LS^i are created and updated when processing the inference messages. The last-seen table is used to decide whether to send wake-up calls as follows. A processor p_i sends a wake-up call $< p_i, y, p_j, a >$ to the resident at $< p_j, a >$, if $x < t$, where $< p_j, a, x, y > \in L^i[j]$, i.e. x is the most recent time when an inference message from $< p_j, a >$ was processed at p_i, and t is the birth-time of some resident φ of p_i. The condition $x < t$ means that φ was established as a resident or modified, *after* the most recent message from $< p_j, a >$ was processed at p_i. Therefore, p_i needs to call for the resident at $< p_j, a >$ to come and visit again. Node p_i compares two times from the same clock in order to determine whether a wake-up call is needed. Thus, the asynchrony of different clocks does not affect this procedure.

We are left with the problem of how often p_i should consult LS^i to issue wake-up calls. In principle, this check should be performed after every modification of a resident at p_i. In practice, this would be an excessive overhead. Therefore, we need to design a policy for the generation of wake-up calls. Any such policy has to satisfy a *fairness* requirement, to ensure that no necessary wake-up call is indefinitely postponed: if $x < t$ holds for $< p_j, a, x, y > \in LS^i[j]$ and $< \varphi, b, t > \in S^i$, then a wake-up call $< p_i, y, p_j, a >$ must be sent eventually.

A simple solution is to establish a fixed *period* τ and stipulate that p_i consults LS^i every τ time units. In order to implement this policy, a variable T^i is maintained at each node p_i: whenever p_i consults LS^i, it records in T^i the current time at its clock. Then, as soon as the condition $clock^i - T^i \geq \tau$ is true, where $clock^i$ represents the current time at p_i's clock, p_i consults LS^i and generates *all* the needed wake-up calls. Since τ is finite, the policy is fair. Remark that the provision that all the wake-up calls are generated together upon consultation of LS^i, is necessary for the fairness of this policy. Since we keep one single time T^i, we need to generate all the wake-up calls together, otherwise a wake-up call to a certain $< p_j, b >$ may be indefinitely postponed as T^i is updated when wake-up calls to other identifiers $< p_k, c >$ are issued. We term this policy an *all-destinations* policy, because all the wake-up calls to all the destinations are generated together.

This policy can be easily turned into a *one-destination* policy by replacing the variable T^i by an array with an entry for each destination: $T^i[j]$ is the time when p_i last checked $L^i[j]$. Then, whenever $clock^i - T^i[j] \geq \tau$ is true, p_i consults $L^i[j]$, generates *all* the needed wake-up calls to p_j and then updates $T^i[j]$ to $clock^i$. Since we keep one time $T^i[j]$ for each processor, we simply need to generate all the wake-up calls to p_j in one batch. This provision and the finiteness of τ ensure the fairness of the policy.

A weakness of these policies is that they do not keep into account the states of the nodes. A more *adaptive* policy, i.e. a policy which admits variations according

to the state of the node may be obtained as follows. We observe that the purpose of sending wake-up calls is to receive inference messages. Thus, the frequency with which a node p_i consults LS^i may depend on the state of its set of messages M^i. If both M^i and LS^i are organized as hash tables with the processors' indexes as key, so that $M^i[j]$ contains all the inference messages from p_j currently held at p_i, we can relate the consultation of $LS^i[j]$ to the state of $M^i[j]$. Intuitively, if $M^i[j]$ is *empty*, it is desirable to issue as soon as possible wake-up calls directed to p_j, in order to get inference messages from p_j. Then, whenever either $M^i[j] = \emptyset$ or $clock^i - T^i[j] \geq \tau$, p_i consults $LS^i[j]$, generates *all* the needed wake-up calls to p_j and then updates $T^i[j]$ to $clock^i$. We call this policy an *adaptive-one-destination* policy. Similarly, an *adaptive-all-destinations* policy establishes that whenever either $M^i = \emptyset$ or $clock^i - T^i \geq \tau$, p_i consults LS^i, generates all the needed wake-up calls to all destinations and updates T^i to $clock^i$.

A one-destination policy generates smaller batches of wake-up calls than an all-destinations policy. Also, in case of adaptive policies, an adaptive-one-destination policy may be expected to generate wake-up calls more frequently than an adaptive-all-destinations policy, since the former generates wake-up calls when $M^i[j] = \emptyset$ for at least one j, whereas the latter generates wake-up calls only when $M^i[j] = \emptyset$ for all j. In other words one policy leads to generate small batches often, whereas the other one leads to generate large batches rarely. The condition based on the period τ is added to adaptive policies in order to ensure fairness, since a derivation may be infinite and thus there is no guarantee that M^i or any of its component will ever become empty. However, the values of the period τ should be chosen in order not to defeat the philosophy of the above policies. For instance, the period for an adaptive-all-destinations policy should not be chosen "too small", otherwise wake-up calls would be generated even if M^i is far from being empty. Appropriate values for the periods for the different policies may be determined experimentally. Moreover, it is possible to devise further variations of the above policies by allowing the processors to have different periods τ^i and possibly by relating the period τ^i to some measure of the work-load of p_i, e.g. the number of messages in M^i.

The above policies are policies for *generating* wake-up calls. Another policy may be needed to decide when *to send* them. When p_i generates a wake-up call, it does not have to send it immediately. In fact, sending wake-up calls as soon as they are generated might cause some congestion, as wake-up calls are generated by batches. One way of reducing the number of wake-up calls to be sent is to pack into one single message all the generated wake-up calls with the same destination. In this way we send fewer, although larger, messages. If we adopt this choice and a one-destination policy, we basically have to send one message at a time. Under an all-destinations policy, we still have to send m messages at a time, where m is the number of selected destinations, with $1 \leq m \leq m$. If m is large, it may be necessary to split the set of outgoing messages into subsets, send those in the first subset and delay the others. Then those in the second subset are sent and so on until the entire load of messages has been output.

4.7.2 Routing of wake-up calls and five-field inference messages

A wake-up call $< p_i, y, p_k, a >$ needs to be routed from node p_i to node p_k. Symmetrically a five-field inference message $< \psi, p_k, a, y, p_i >$, issued to reply to the wake-up call, needs to be routed from p_k to p_i. A convenient choice is to route wake-up calls and inference messages, regardless of whether they are four-fields or five-field inference messages, along *reverse paths*. In other words, for every two nodes p_k and p_i, if all inference messages from p_k reach p_i by going through a path $p_{x_1}, p_{x_2}, \ldots p_{x_{m-1}}, p_{x_m}$, then all wake-up calls from p_i to p_k go through the reverse path $p_{x_m}, p_{x_{m-1}}, \ldots p_{x_2}, p_{x_1}$. If the routing algorithm has this property, we say that the *reverse paths assumption* is satisfied. This can be easily done if routing and broadcasting are performed on the cube, as described in Sections 4.5.1 and 4.5.2. It is sufficient to broadcast/route inference messages row-first and to route wake-up calls column-first or vice versa.

Under the reverse paths assumption, we may generalize the format of wake-up

calls to $< L, y, p_k, a >$, where the first field indicates not just the sender of the wake-up call, but a list $L = [p_{r_1} \ldots p_{r_h}]$ of processors calling for the resident at $< p_k, a >$. The last element p_{r_h} in the list is the node where the wake-up call was originated. The list L of processors on a wake-up call $< L, y, p_k, a >$ is handled as a *last-in first-out* list. During the routing of the wake-up call, each node on the route, upon receipt of $< L, y, p_k, a >$, consults its last-seen table, and, if it also needs to see the resident at $< p_k, a >$, it pushes its name on top of the list L and forwards the call. Otherwise, it just forwards it as it is. When the wake-up call arrives at its destination p_k, node p_k replies by issuing a five-field inference message $m =< \psi, p_k, a, y, L >$. During its routing, message m meets the nodes in L in reverse order with respect to the wake-up call, i.e. it reaches first the last node which endorsed the wake-up call. A node which is not in L simply forwards the message m. A node in L processes m like a standard four-fields inference message and then pops its name from the list. When the list is empty, the five-field inference message is discarded.

4.7.3 Wake-up calls and redundant inferences

The purpose of wake-up calls is to obtain re-editions of *persistent* residents. However, during a derivation it is not known which residents will persist and which will not. Therefore, it happens in practice that wake-up calls are sent to residents which are not persistent. This causes redundant inferences. For instance, if a node p_k receives a wake-up call for its resident ψ, generates a five-field inference message m for ψ and then, later on, reduces ψ to ψ', the message m and all the inferences performed with m as a premise are in a sense redundant, because a new inference message will be issued for ψ' any way. Clearly, such redundancy cannot be completely eliminated. In any derivation, regardless of whether it is sequential or distributed, it is not known which clauses will persist and non-persistent clauses are necessarily used. Intuitively, if we knew which clauses are going to persist we would have already solved the input problem of the derivation.

In this section we explore ways of limiting the redundancy introduced by wake-up calls. This can be done at three stages:

1. during the routing of a wake-up call,
2. at the destination of a wake-up call and
3. at the destination of a five-field inference message issued in reply to a wake-up call.

We consider these three stages in this order.

During the routing of a wake-up call

The information carried by a wake-up call $< L, y, p_k, a >$ can be used to detect whether the wake-up call is redundant even before it reaches its destination. We consider a node p_g which receives a wake-up call $< [L, p_i], y, p_k, a >$ originated at p_i. Processor p_g is any intermediate node on the route between p_i and p_k. Node p_g consults its last-seen table LS^g and retrieves the entry $< p_k, a, z, y' >$, saying that the most recent inference message from $< p_k, a >$ seen at p_g carried time stamp y'. Then it compares the time-stamp y' with the time-stamp y carried by the wake-up call. We recall that y is the time-stamp of the last inference message from $< p_k, a >$ processed at p_i, according to the state of p_i's last-seen table, when the wake-up call was created at p_i. The two time-stamps y and y' can be safely compared, because they are both taken at p_k's clock.

If $y' > y$, p_g discards the wake-up call. The inference message $< \psi, p_k, a, y' >$, which has been seen at p_g, is being broadcast and thus it is guaranteed to reach all the nodes in $[L, p_i]$, making the wake-up call redundant. The possibility that, at some node, a contraction inference steps is applied to ψ in $< \psi, p_k, a, y' >$ does not affect the decision of discarding the wake-up call. If ψ is replaced by ψ', a "better"

message $< \psi', p_k, a, y' >$ will reach. If ψ is deleted, e.g. subsumed, there is no need to receive it.

If $y = y'$, p_g is in the same situation as p_i as far as knowledge of the resident at $< p_k, a >$ is concerned. Node p_g checks whether it needs to send a wake-up call to $< p_k, a >$ and if that is the case, rather than generating another wake-up call, it piggybacks its wake-up call onto the existing one: if $< p_k, a, z, y' >$ is the entry for $< p_k, a >$ in $LS[k]^g$ and if there exists a resident of p_g, whose birth-time t is more recent than z, i.e. $t > z$, p_g pushes its name on top of the list of nodes carried by the wake-up call and forwards it as $< [p_g|L, p_i], y, p_k, a >$.

These two cases exhaust all possibilities, because, under the reverse paths assumption, it is guaranteed that $y' \geq y$. All the wake-up calls from p_i to p_k go through p_g and all the inference messages from p_k to p_i also go through p_g, while travelling in the opposite direction. Therefore, it cannot be $y' < y$, because the inference message from p_k with time-stamp y, which has been at p_i, must have been at p_g also, as it had to go through p_g to reach p_i.

At the destination of a wake-up call

A principle similar to the one applied during the routing of a wake-up call can be enforced at its destination. When a node p_k receives a wake-up call $< L, y, p_k, a >$, it checks first that there is still a resident with identifier a. If there is not, it means that that resident has been deleted, and therefore the wake-up call is discarded. Otherwise, p_k compares the birth-time t of its resident ψ with identifier a with the time-stamp y carried by the wake-up call. If $y < t$, it means that the resident at $< p_k, a >$ has been reduced at p_k since the inference message with time-stamp y was generated. Thus, another inference message $< \psi, p_k, a, t >$ has been issued in the mean time or it will soon, regardless of the wake-up call. Since inference messages sent on behalf of reduced residents are broadcast, the inference message $< \psi, p_k, a, t >$ will reach all the nodes in L. Therefore the wake-up call is useless

and p_k discards it. If $y = t$, p_k replies to the wake-up call by sending a five-field inference message $< \psi, p_k, a, y, L >$.

At the destination of a five-field inference message

We see first why a five-field inference message may cause redundant inferences. Let p_i be any node in the list L of a wake-up call $< L, y, p_k, a >$. If p_k has replied to a wake-up call $< L, y, p_k, a >$ by a five-field inference message $< \psi, p_k, a, y, L >$, it means that ψ has not been modified at p_k, since its previous inference message, which also carried time-stamp y, was broadcast. Therefore, the residents of p_i which were already residents at the time of the previous visit by ψ, have already interacted with it and it is redundant to try those inference steps again. However, a node p_i receiving a five-field inference message has in its last-seen table sufficient information to avoid redundant steps of this type. Node p_i consults LS^i at entry k and retrieves the entry $< p_k, a, x, y > \in LS^i[k]$. Obviously, LS^i is consulted *before* being updated because of the arrival of the five-field inference message itself. Then, only the residents $< \varphi, b, t > \in S^i$ such that $t > x$, i.e. those established or reduced *after* the previous inference message from $< p_k, a >$ was processed at p_i, are selected to perform inferences with ψ. The condition $t > x$ is the same which is used to determine whether a wake-up call has to be sent. This ensures that the residents of p_i, on whose behalf the wake-up call was issued, are selected to meet ψ. Also, all the residents established or reduced after the wake-up call was generated are selected. Clearly, this treatment relies on the assumption that whenever a resident is simplified, its birth-time is updated.

Unfortunately, even if all the above criteria to limit redundancy are adopted, a wake-up call may still induce redundant steps. Since p_k has issued a five-field inference message, we know that ψ has been persistent so far at p_k. Persistency at p_k does not imply persistency all over N, though. Consider for instance the following scenario: when ψ undertakes its first trip as $< \psi, p_k, a, y >$, it is reduced to ψ_1 on the

path from p_k to p_i. Thus the residents of p_i interact with ψ_1. For simplicity, let φ be the single simplifier which simplified $< \psi, p_k, a, y >$ to $< \psi_1, p_k, a, y >$. A message carrying φ is guaranteed to reach ψ at p_k eventually. However, we assume that φ is late. After $< \psi_1, p_k, a, y >$ has been processed and discarded at p_i, some residents of p_i are reduced and p_i emits a wake-up call to $< p_k, a >$ on their behalf. If p_k receives the wake-up call and reacts to it *before* receiving and processing φ, p_k will send a message $< \psi, p_k, a, y, p_i >$. Since a five-field inference message is forwarded without being simplified by the nodes which are not in its destination list, it may happen that ψ reaches p_i unchanged. Then ψ will perfom inferences only with a subset $A \subseteq S^i$ of selected residents, according to the above policy. However, *all* these steps are *redundant*, because after φ finally reaches p_k, the resident ψ will be simplified to ψ_1 and p_k will issue any way a brand new, four-field message m carrying ψ_1. Notice that introducing simplification of five-field inference messages at all nodes would not fix the problem, because then the steps performed between m and the residents in A would be redundant.

This scenario may look concocted, as it seems unlikely that the wake-up call will be processed before the simplification of ψ by φ takes place at p_k. However it is not impossible, because the processors work asynchronously and therefore we cannot make assumptions on the order of events. The conclusion of this discussion is that since wake-up calls have an intrinsic potential for causing redundant steps, it is important to implement mechanisms to trim their number and to contain their effects, as those introduced in this section. Finally, it is also important to choose a good policy for the generation of wake-up calls, in order not to generate them "too eagerly" (see Section 4.7.1).

4.7.4 Order of the operations on wake-up calls

Since wake-up calls are a source of redundancy, a node should always try first to delete them, then possibly to piggyback its own wake-up call onto an existing one

and, as a last resort, to send a new one. Indeed, our general schedule above lists first *Contraction*, where redundant wake-up calls are deleted, then *Forward*, where the piggybacking mechanism is applied, and then *Send*, where new wake-up calls are emitted. The treatment of the last-seen table can also be inserted appropriately in the above general schedule. A node p_i first detects and eliminates as many generalized duplicates as possible among the inference messages in M^i. Next, for every new inference message $< \psi, p_k, a, y >$, the entry for $< p_k, a >$ in $LS^i[k]$ is set to $< p_k, a, \infty, y >$, where ∞ means that the inference message has yet to be processed. Having the third field set to ∞ ensures that no wake-up call to $< p_k, a >$ is issued, while a message $< \psi, p_k, a, y >$ from $< p_k, a >$ is held in $M^i[k]$. A wake-up call is sent if the birth-time of a resident is greater than the third field and no birth-time is greater than ∞. Then, p_i checks for redundant wake-up calls. It is important to check for redundant wake-up calls *after* the last-seen table has been updated, so that we can use the most recent information to delete as many wake-up calls as possible. We recall that we use the fourth field in the entries in the last-seen table, in order to detect redundant wake-up calls, and therefore this process is not affected by having the third field temporarily set to ∞. When an inference message $< \psi, p_k, a, y >$ is discarded, the entry $< p_k, a, \infty, y >$ in $LS^i[k]$ is set to $< p_k, a, x, y >$, where x is the current time at p_i's clock.

Chapter 5

Contraction and fairness in distributed derivations

We have described the Clause-Diffusion methodology assuming that a sequential strategy $\mathcal{C} =< I; \Sigma >$ is given. By embedding a sequential strategy $\mathcal{C}$ into a Clause-Diffusion method, we obtain a specific Clause-Diffusion strategy. In this chapter we give a formal treatment of distributed derivations generated by Clause-Diffusion strategies. First, we define the distributed derivations. Next, we consider the problem of uniform fairness of distributed derivations. As we have seen in Section 2.7, the uniform fairness of a derivation requires that all the clauses which can be derived from non-redundant, persistent parents be generated eventually. In a distributed derivation, uniform fairness poses a requirement of *fairness of communication*: ensuring that any two non-redundant, persistent clauses meet eventually. To provide a solution, we extend first the definition of uniform fairness to distributed derivations. Then, we devise a set of more concrete conditions to be satisfied by the policies for message-passing and we prove that these conditions imply distributed uniform fairness. Finally, we briefly recall how the techniques featured by the Clause-Diffusion methods, described in the previous chapter, implement these conditions for fairness.

It follows that if the search plan Σ of the given sequential strategy $\mathcal{C} =< I; \Sigma >$ is (uniformly) fair, the Clause-Diffusion strategies based on $\mathcal{C}$ are (uniformly) fair. Thus, if the inference mechanism I is refutationally complete, the Clause-Diffusion strategies are complete.

Then, we treat some special topics on distributed contraction. First, we introduce contraction inference rules to delete redundant messages. These inference rules formalize some of the techniques described in the previous chapter. Second, we study *subsumption in distributed derivations*. It is well-known since [91] that the unrestricted application of subsumption, and especially *backward subsumption of variants*, may destroy the fairness and thus the completeness of a strategy. We observe that in the distributed case, subsumption may violate, in addition to fairness, the *monotonicity* of a derivation, and the solutions known for the sequential case do not apply. We discover that these problems are due to a basic misconception of the subsumption rule, which is commonly regarded as a *deletion* rule. We consider subsumption as a *replacement* rule. Based on this understanding, we define a new *distributed subsumption* inferences rule, which has all the desirable properties: it allows subsumption in a distributed data base, while preserving fairness and monotonicity, and it works for both sequential and distributed derivations.

5.1 Distributed derivations

A distributed derivation is given by a family of derivations, one for each node. Every processor p_k computes a derivation

$$(S; M; CP; NS)_0^k \underset{\mathcal{C}}{\vdash} (S; M; CP; NS)_1^k \underset{\mathcal{C}}{\vdash} \ldots (S; M; CP; NS)_i^k \underset{\mathcal{C}}{\vdash} \ldots$$

where

- S_i^k is the set of *residents* at p_k at stage i,
- M_i^k is the set of *inference messages* at p_k at stage i,

- CP_i^k is the set of *raw clauses* at p_k at stage i and
- NS_i^k is the set of *new settlers* at p_k at stage i.

The derivations at the different processors proceed *asynchronously*. The *state* of the derivations at processor p_k and stage i is represented by the tuple $(S; M; CP; NS)_i^k$. More components may be added if indicated by a specific strategy. For example, if global contraction by travelling is used, an additional field for wake-up calls, WU, needs to be included. A distributed derivation succeeds as soon as the derivation at one node finds a proof. A step in a distributed derivation can be either an *expansion* step or a *contraction* step or a *communication* step, which includes *Send*, *Forward* and *Settle*. For instance, sending an inference message for $\psi \in S^k$ from node k to an adjacent node j can be written as $(S^k \cup \{\psi\}, M^j) \vdash (S^k \cup \{\psi\}, M^j \cup \{\psi\})$. Settling a new settler at node k can be written as $(S^k, NS^k \cup \{\psi\}) \vdash (S^k \cup \{\psi\}, NS^k)$. This representation assumes that communication between any two adjacent nodes is instantaneous. It does *not* assume, however, that communication between *any* two nodes is instantaneous. If an inference message sent by p_i reaches p_j through $p_{x_1} \ldots p_{x_m}$, it appears first in M^{x_1}, then in M^{x_2} and so on. The time elapsed in going from the source to the destination is captured in our description, by showing the message stored, at successive stages, in the appropriate component of all the nodes on the path.

5.2 Uniform fairness of distributed derivations

In order to extend the definition of uniform fairness to the distributed case, we need to define the limit of a distributed derivation. First, we define the *local data base* at node p_k at stage i as the union $G_i^k = S_i^k \cup M_i^k \cup CP_i^k \cup NS_i^k$. Then, the *local limit* at processor p_k is $G_\infty^k = \bigcup_{i \geq 0} \bigcap_{j \geq i} G_j^k$. The *global data base* at stage i is the union of the local data bases, i.e. $G_i = \bigcup_{k=1}^n G_i^k$, and the *global limit* is $G_\infty = \bigcup_{k=1}^n G_\infty^k$. Local and global limits may be defined similarly for each component of the states

in a distributed derivation, e.g. S^k_∞ and S_∞, M^k_∞ and M_∞. Then, Definition 2.7.1 is extended to a distributed derivation as follows:

Definition 5.2.1 *A distributed derivation*

$$(S; M; CP; NS)^k_0 \vdash_{\mathcal{C}} (S; M; CP; NS)^k_1 \vdash_{\mathcal{C}} \ldots (S; M; CP; NS)^k_i \vdash_{\mathcal{C}} \ldots,$$

for all k, $1 \leq k \leq n$, *is* uniformly fair *if* $I_e(G_\infty - R(G_\infty)) \subseteq \bigcup_{k=1}^{n} \bigcup_{j \geq 0} G^k_j$.

The following three conditions form a more concrete specification of uniform fairness of distributed derivations.

1. All messages should be processed eventually and thus there are *no persistent messages*:

   $\forall k$, $1 \leq k \leq n$, $M^k_\infty = CP^k_\infty = NS^k_\infty = \emptyset$.

2. Given k and $\psi \in S^k_\infty$, we define the *abstract birth-time* of ψ in k to be the smallest index $i \geq 0$ such that $\psi \in \bigcap_{j \geq i} S^k_j$.[1] Then for every node p_k and for every persistent, non-redundant resident φ at p_k, *all persistent, non-redundant residents at the other nodes will eventually appear as inference messages at* p_k, *after the birth of* φ:

   $\forall k$, $1 \leq k \leq n$, $\forall \varphi \in (S^k_\infty - R(S^k_\infty))$, if i is the abstract birth-time of φ, then $\forall h$, $1 \leq h \neq k \leq n$, $\forall \psi \in (S^h_\infty - R(S^h_\infty))$, there exists an $l > i$ such that $\psi \in M^k_l$.

   Notice that i and l are stages of the same derivation, i.e. the derivation at p_k.

3. The derivation is uniformly fair with respect to the local inferences at each node. That is, every clause that can be generated from persistent, non-redundant clauses at p_k will be generated: $\forall k$, $1 \leq k \leq n$, $I_e(G^k_\infty - R(G^k_\infty)) \subseteq \bigcup_{i \geq 0} CP^k_i$.

[1] The adjective *abstract* indicates that i is an index in the abstract view of the derivation, and not a time of any processor's clock.

While Condition 3 paraphrases the requirement that the sequential strategy to begin with is fair, Conditions 1 and 2 take care of the distributed part of the derivation. Intuitively, Conditions 1 and 2 guarantee that a clause which can be generated from two persistent non-redundant clauses φ_1 and φ_2 residing at two different nodes, will be considered. Condition 2 ensures that φ_1 and φ_2 will eventually meet each other through inference messages. Condition 1 makes sure that all inference messages be processed ($M_\infty = \emptyset$), all raw clauses (those that, in the presence of contraction rules, remain non-trivial after having been fully contracted) become new settlers ($CP_\infty = \emptyset$) and all new settlers become residents at some place ($NS_\infty = \emptyset$). Condition 3 takes care of fairness of the sequential derivation at each node.

Because the definition of uniform fairness, and consequently our three conditions, focus only on persistent, non-redundant clauses, it is fairly simple to show that these three conditions imply Definition 5.2.1:

Theorem 5.2.1 *If a distributed derivation is such that*

1. $\forall k,\ 1 \leq k \leq n,\ M_\infty^k = CP_\infty^k = NS_\infty^k = \emptyset$,
2. $\forall k,\ 1 \leq k \leq n,\ \forall \varphi \in (S_\infty^k - R(S_\infty^k))$, *if* i *is the abstract birth-time of* φ, *then,* $\forall h,\ 1 \leq h \neq k \leq n,\ \forall \psi \in (S_\infty^h - R(S_\infty^h))$, *there exists an* $l > i$ *such that* $\psi \in M_l^k$ *and*
3. $\forall k,\ 1 \leq k \leq n,\ I_e(S_\infty^k - R(S_\infty^k)) \subseteq \bigcup_{i \geq 0} CP_i^k$.

then $I_e(G_\infty - R(G_\infty)) \subseteq \bigcup_{k=1}^n \bigcup_{i \geq 0} G_i^k$, *i.e. the distributed derivation is* uniformly fair.

Proof: let φ be any clause in $I_e(G_\infty - R(G_\infty))$ with parents ψ_1 and ψ_2. Since $M_\infty = CP_\infty = NS_\infty = \emptyset$, $G_\infty = S_\infty$, i.e. $\psi_1, \psi_2 \in S_\infty$. It follows that $\psi_1 \in (S_\infty^k - R(S_\infty))$ and $\psi_2 \in (S_\infty^h - R(S_\infty))$ for some $1 \leq k, h \leq p$.

If $k = h$, then $\varphi \in I_e(S^k_\infty - R(S_\infty))$. Since $S^k_\infty \subseteq S_\infty$, by the monotonicity of the redundancy criterion (see Definition 2.6.3), $R(S^k_\infty) \subseteq R(S_\infty)$ and thus $I_e(S^k_\infty - R(S_\infty)) \subseteq I_e(S^k_\infty - R(S^k_\infty))$. By Condition 3, there exists an i such that $\varphi \in CP^k_i \subseteq G^k_i \subseteq \bigcup_{k=1}^n \bigcup_{i\geq 0} G^k_i$.

If $k \neq h$, let i_1 and i_2 be the abstract birth-times of ψ_1 and ψ_2 respectively. By Condition 2, we have $\psi_1 \in M^h_{l_1}$ for some $l_1 > i_2$ and $\psi_2 \in M^k_{l_2}$ for some $l_2 > i_1$. Since $M_\infty = \emptyset$ by Condition 1, we know that the inference message ψ_1 does not persist at p_h and the inference message ψ_2 does not persist at p_k. An inference message may be deleted before performing expansion steps, by a contraction step. Since ψ_1 and ψ_2 are in $G_\infty - R(G_\infty)$, i.e. they are globally persistent and non-redundant, this is impossible. It follows that the inference messages $\psi_1 \in M^h_{l_1}$ and $\psi_2 \in M^k_{l_2}$ are deleted only after having been processed. Thus, paramodulation of ψ_1 into ψ_2 is tried at p_h and paramodulation of ψ_2 into ψ_1 is tried at p_k. Either one of these two steps generates φ, i.e. either $\varphi \in CP^h_i$ or $\varphi \in CP^k_i$ at some stage i, i.e. $\varphi \in \bigcup_{k=1}^n \bigcup_{i\geq 0} G^k_i$. □

By this theorem, the abstract definition of uniform fairness is reduced to three more concrete requirements:

Corollary 5.2.1 *Let $\mathcal{C} =< I; \Sigma >$ be a complete (sequential) theorem proving strategy and $\mathcal{D}$ be its distributed version. If the algorithms and policies handling messages satisfy Conditions 1 and 2, then $\mathcal{D}$ is a complete distributed theorem proving strategy.*

The Clause-Diffusion methods described in Chapter 4 satisfy Condition 2 as follows. Condition 2 requires that for every non-redundant, persistent resident ψ_1 at any node p_k, any other non-redundant, persistent resident ψ_2 appear at p_k as inference messages, after the abstract birth-time of ψ_1. We assume that ψ_1 is born "before" ψ_2, i.e. $i_1 \leq i_2$ for i_1 and i_2 the birth-times of ψ_1 and ψ_2 respectively. The case where $i_2 \leq i_1$ is clearly symmetrical.

In order to make sure that the "younger" clause ψ_2 paramodulates into the "older" clause ψ_1, it is sufficient that ψ_2 is broadcast and reaches all nodes. For this purpose, we have required that any new settler ψ, which settles down at a node, be emitted as inference message eventually, unless it is deleted before hand. However, it is not sufficient to consider new settlers, because residents are subject to backward contraction. Thus we have introduced the policy that whenever a resident is reduced, it is regarded as a new clause and thus will be re-scheduled for the emission as inference message (see Section 4.6.1). If a non-persistent resident ψ is reduced to ψ', an inference message for ψ' is broadcast eventually, regardless of whether one for ψ was already sent, because ψ' may be persistent. In order to ensure that the "younger" equation ψ_2 is paramodulated into by the "older" clause ψ_1, we may use either the wake-up calls or the policy of saving the clauses carried by inference messages in the SH^k's or in SH in shared memory, as explained in Section 4.6.3. Thus, at node p_h, the home node of ψ_2, ψ_1 is saved in SH^h (or can be accessed from SH) and has a chance to paramodulate into ψ_2.

We have considered the symmetry between $i_1 \leq i_2$ and $i_2 \leq i_1$. Another way to look at the problem of guaranteeing the interaction of ψ_1 and ψ_2 is to consider the symmetry between "ψ_1 paramodulates into ψ_2" and "ψ_2 paramodulates into ψ_1". Under this perspective, broadcasting ψ_1 allows ψ_1 to paramodulate into the clauses born *"before"* ψ_1, while saving ψ_1 in the SH^h's (in SH) or using wake-up calls allows ψ_1 to paramodulate into the clauses born *"after"* ψ_1.

5.3 Inference rules to delete redundant messages

5.3.1 Deletion of redundant inference messages

We have seen in Section 4.6.2 that the interaction of broadcasting and contraction may generate *generalized duplicates*, that is distinct inference messages, e.g. $m_1 =< \psi_1, p_k, a, x >$ and $m_2 =< \psi_2, p_k, a, y >$, with the same global identifier $< p_k, a >$.

Because they have the same global identifier, we know that they carry contracted forms of the same resident and thus they are logically equivalent. If $\psi_1 = \psi_2$ and $x = y$, the two messages are just two plain duplicates. If $\psi_1 \neq \psi_2$, but $x = y$, the two messages were originally plain duplicates, whose clauses have been rewritten to two different forms during their travel. If $x \neq y$, the two messages carried different clauses since their origin, as testified by the different time-stamps. If $x < y$, then the original clause of the message m_2 is a contracted form of the original clause of m_1, since m_2 is emitted later. The case for $x > y$ is symmetric.

Suppose node p_i has received two generalized duplicates $m_1 =< \psi_1, p_k, a, x >$ and $m_2 =< \psi_2, p_k, a, y >$, such that $y \geq x$. Since ψ_1 and ψ_2 are logically equivalent, it is redundant to perform inferences with both of them. The contents of the messages can be used to decide which one should be discarded as follows. If they carry the same clause, i.e. $\psi_1 = \psi_2$, then we discard m_1, the message carrying an earlier time-stamp. Saving the message with the most recent time-stamp is useful, for instance, to update the last-seen table with the most recent information. Otherwise, i.e. $\psi_1 \neq \psi_2$, we discard m_1 if $\psi_1 \succ \psi_2$ and discard m_2 if $\psi_2 \succ \psi_1$. If ψ_1 and ψ_2 are not comparable, then we discard m_1 since it was emitted earlier. The following inference rule captures these ideas:

Discard Message: Let p_i and p_k be two distinct nodes.

- $\dfrac{M^i \cup \{< \psi_1, p_k, a, x >, < \psi_2, p_k, a, y >\}}{M^i \cup \{< \psi_2, p_k, a, y >\}} \quad \psi_1 \succ \psi_2 \ \vee \ (\psi_2 \not\succ \psi_1 \ \wedge \ y \geq x)$
- $\dfrac{SH^i \cup \{< \psi_1, p_k, a, x >\}; M^i \cup \{< \psi_2, p_k, a, y >\}}{SH^i \cup \{< \psi_2, p_k, a, y >\}; M^i} \quad \psi_1 \succ \psi_2 \vee (\psi_2 \not\succ \psi_1 \wedge y \geq x)$
- $\dfrac{SH^i \cup \{< \psi_2, p_k, a, y >\}; M^i \cup \{< \psi_1, p_k, a, x >\}}{SH^i \cup \{< \psi_2, p_k, a, y >\}; M^i} \quad \psi_1 \succ \psi_2 \vee (\psi_2 \not\succ \psi_1 \wedge y \geq x)$

In addition to comparing two inference messages in the messages set M^i, the Discard Message inference rule can be utilized to compare an incoming message with a

clause in SH^i. The second rule shows how an incoming inference message is used to update SH^i, realizing the mechanism of *update by inference messages* introduced in Section 4.2.2. The third rule shows how an incoming message can be deleted based on the contents of SH^i.

5.3.2 Deletion of redundant wake-up calls

In this section we formulate in terms of contraction inference rules the criteria to delete redundant wake-up calls given in Section 4.7.3:

Discard Wake-up call: Let p_i and p_k be two distinct nodes.

- Let p_i be an intermediate node on the route of a wake-up call $< L, y, p_k, a >$ to its destination p_k:

  $$\frac{WU^i \cup \{< L, y, p_k, a >\}}{WU^i} \quad z > y, \quad < p_k, a, x, z > \in LS^i[k]$$

  The wake-up call is deleted, because the entry for the global identifier $< p_k, a >$ in the last-seen table $LS^i[k]$ shows a time-stamp z which is more recent than the one carried by the wake-up call. The inference message with time-stamp z which has been seen at p_i is guaranteed to reach all nodes in L, making the wake-up call redundant.

- A wake-up call $< L, y, p_k, a >$ is deleted at its destination p_k under two conditions:

  - The first one is that there is no resident with the requested identifier a, because that resident has been deleted:

    $$\frac{S^k; WU^k \cup \{< L, y, p_k, a >\}}{S^k; WU^k} \quad \forall < \psi, b, x > \in S^k, \ b \neq a$$

  - The second one is that the resident with identifier a has a birth-time t, which is more recent than the time-stamp carried by the wake-up call:

$$\frac{S^k \cup \{< \psi, a, t >\}; WU^k \cup \{< L, y, p_k, a >\}}{S^k \cup \{< \psi, a, t >\}; WU^k} \quad t > y$$

This means that the resident has been reduced and thus an inference message has been or will be issued on its behalf. Such inference message is guaranteed to reach all nodes in N and therefore also those in L, making the wake-up call redundant.

5.4 Subsumption in distributed derivations

It is well-known since [91] that the unrestricted application of subsumption, and especially *backward subsumption of variants*, may destroy the completeness of a strategy which is otherwise complete. The reason is that the derivation may generate an infinite sequence of variants of the same clause, each of which subsumes its predecessor and prevents any clause in the sequence from being selected for other inferences necessary to obtain a proof. Technically, the source of the problem is that the subsumption ordering is not well-founded. Because of this difficulty, many inference systems include only proper subsumption since it is well-founded and preserves completeness (e.g. [109, 16]). On the other hand, most existing first-order theorem provers (e.g., [98]) do feature subsumption, not just proper subsumption, because eliminating variants is very useful in practice. Completeness is maintained usually by combining the proper subsumption ordering with some other ordering to sort variants. For instance, the prover may associate to each clause its age for the purpose of subsumption and a clause is forbidden to subsume an older variant. This provision prevents the violation to completeness illustrated in [91].

The problem with subsumption becomes much more serious, however, in a distributed derivation, because clauses may be *duplicated*, i.e. the same clause may appear in different derivations. In this context, two unprecedented difficulties arise. First, it may happen that a combination of concurrent steps of subsumption of

variants deletes all the variants of a clause. This makes the distributed derivation *non-monotonic*, as a theorem of the original theory is lost. Thus, in addition to the possibility of losing completeness, subsumption of variants causes the even more fundamental problem of losing monotonicity. The solutions known for the sequential case cannot be extended straightforwardly to the distributed one. For instance, sorting variants by age is not sufficient, because the concurrent processes are asynchronous, so that there is no general way to establish that a clause has been generated earlier than another.

Second, not only subsumption of variants, but proper subsumption itself may also destroy monotonicity of a distributed derivation. This may happen if a process uses a received inference message ψ to subsume and delete a clause φ from its own data base. In general, there is no guarantee that ψ is stored in the data base of another process. If it is not, both φ and ψ may be lost, violating again monotonicity.

In this section, we analyze these issues and provide a comprehensive solution for both sequential and distributed subsumption. We show that the source of the problem lies in a misconception of the subsumption inference rule. We reason that proper subsumption is not deletion of a clause φ, because of the presence of a more general clause ψ, but rather *replacement* of φ by ψ. This difference cannot be easily appreciated in the sequential case, where φ and ψ are stored in the same data base. But it is significant in a distributed environment, where inferences may involve clauses belonging to remote data bases. Based on this understanding of subsumption, we refine the subsumption inference rule into two rules: *replacement subsumption* and *variant subsumption*. Under this framework, sequential proper subsumption becomes a derived inference rule. In the distributed case, variations of the two inference rules are combined into a *distributed subsumption* inference rule, which has all the desirable properties of subsumption. It allows proper subsumption between clauses stored in remote data bases, without violating monotonicity, and subsumption of variants, while preserving monotonicity and completeness. For the latter, distributed subsumption incorporates a way of achieving well-foundedness for

variant subsumption, without assuming the existence of a global clock.

5.4.1 The problem with subsumption and fairness

A strategy which allows unrestricted application of subsumption is not complete in general. The following example, appeared originally in [87] and reported in [91] (pages 207-208), illustrates this phenomenon. Consider a resolution theorem proving strategy whose search plan uses the *set of support* restriction and sorts the set of support as a first-in-first-out queue. Since resolution is refutationally complete, and the first-in-first-out set of support research plan is fair, i.e. it is guaranteed to select eventually all the steps which are necessary to reach a proof, it follows that the strategy is complete. It is no longer complete, however, if subsumption is added to the inference system and the search plan applies it as soon as possible. Let the input set of clauses be (1) $P(x,a)$, (2) $P(f(x),y) \vee \neg P(x,y)$, (3) $\neg Q(y) \vee \neg P(x,y)$ and (4) $Q(a)$, where (1) $P(x,a)$ is originally in the set of support. The search plan selects clause (1) and resolves it first with (2) to generate (5) $P(f(x),a)$ and then with (3) to yield (6) $\neg Q(a)$. The two new clauses are added in this order to the set of support. Next, the search plan selects (5) and resolves it with (2) and with (3), generating (7) $P(f(f(x)),a)$ and (8) $\neg Q(a)$ respectively. Clause (8) back-subsumes (6), so that the search plan selects (7) next and generates (9) $P((f(f(fx))),a)$ and (10) $\neg Q(a)$. Again, (10) back-subsumes (8) and (9) is selected. The derivation proceeds indefinitely, always missing to use $\neg Q(a)$ to resolve with $Q(a)$ and obtain the empty clause.

As remarked in [91], the root of the problem is that the subsumption ordering is *not* well-founded. In the above example, the derivation generates an infinite sequence of clauses, $\neg Q(a) \succeq \neg Q(a) \succeq \ldots$, where each element in the sequence subsumes its predecessor. The inference step needed for the refutation, the resolution of $\neg Q(a)$ and $Q(a)$, is never selected. Thus, a search plan which applies eagerly a

contraction rule such as subsumption, controlled by an ordering which is not well-founded, is not fair. Consequently, the strategy is not complete. The usual solution to this problem is to label the clauses in some well-founded ordering and perform subsumption of variants according to the ordering. For instance, the retriction of *forward subsumption before backward subsumption* [91] is obtained by labeling the clauses according to the time they are generated.

5.4.2 The problem with subsumption and monotonicity

The problem with subsumption becomes much more involved in a distributed derivation. Unrestricted application of subsumption may violate not only fairness, but also monotonicity of the derivation. Intuitively, the reason is that in a distributed derivation it is possible to have *duplicated data*. That is, the same clause φ may be present at the same time in the derivation both as resident at one node and as inference message at another.

First, we see how subsumption of variants may be harmful. Envision two variants φ and ψ residing at nodes p_i and p_j, respectively. Assume also that p_i and p_j have received, respectively, a copy of ψ and φ as inference messages. There are several possible scenarios on how subsumption may be done, but none of them is satisfactory. If the theorem proving strategy requires that the residents subsume the inference messages first, then both messages will be subsumed, while both residents will remain intact. This defeats the purpose of subsumption since both variants are still in the global data base. If the inference messages are used to subsume residents first, then *both* residents, φ at p_i and ψ at p_j, will be deleted. If the nodes do not save the inference messages they receive, the subsumption of both residents results in a change of the theory, that is, monotonicity is destroyed. If the inference messages are saved in the localized image sets, the message carrying φ is saved at p_j and the one carrying ψ is saved at p_i. However, this is not an acceptable solution, for two reasons. First, both variants are still in the global data

base, so that in actuality subsumption has not been achieved. Second, fairness is lost, because p_i regards $\psi \in SH^i$ as a clause belonging to p_j and therefore it will not perform paramodulation steps into ψ. The situation at p_j is symmetrical.

More surprisingly, even proper subsumption may violate monotonicity: let $P(f(x))$ and $P(g(y))$ be residents at nodes p_i and p_j respectively. Node p_i broadcasts an inference message carrying $P(f(x))$ and similarly does p_j for $P(g(y))$. Assume that p_k is an intermediate node between p_i and p_j. Node p_k receives from p_i the inference message containing $P(f(x))$. A contraction step at p_k, e.g. by $f(x) \simeq x$, replaces $P(f(x))$ in the inference message by $P(x)$. Then $P(x)$ is forwarded to p_j. Similarly, the copy of $P(g(y))$ carried by the inference message may be reduced, so that p_i receives a reduced form $P(y)$ of ψ. The message $P(x)$ subsumes $P(g(y))$ at p_j and the message $P(y)$ subsumes $P(f(x))$ at p_i:

P_i	P_j	time ↓
resident P(f(x))	resident P(g(y))	initial state
resident P(f(x)) message P(y)	resident P(g(y)) message P(x)	messages arriving
message P(y)	message P(x)	after subsumption
empty	empty	messages consumed

Thus, both residents $P(f(x))$ and $P(g(y))$ are deleted, while $P(x)$ and $P(y)$ are not guaranteed to be resident at any node. The theory is modified and monotonicity of the derivation is lost. This phenomenon may happen even if there is no intermediate node p_k, i.e. the inference message $P(f(x))$ is reduced to $P(x)$ at p_j and the inference message $P(g(y))$ is reduced to $P(y)$ at p_i, *before* the subsumption tests are

performed. Thus, the violations of monotonicity may occur also when the broadcasting scheme guarantees that a message issued by a node reaches all the other nodes in one hop. This is the case, for instance, if the graph N is fully connected, i.e. any two nodes are directly connected by a link. Monotonicity is not violated if the received inference messages are stored in the localized image sets. However, as we remarked already for subsumption of variants, fairness may be violated, since the inference messages stored in the localized image sets retain their original ownership.

5.4.3 Sequential subsumption revisited

A satisfactory solution to the above problems requires a re-examination of the concept of subsumption itself. The essence of contraction is basically replacing a clause by an equivalent or more general one in the data base. Thus, *tautological deletion* is replacing a tautology by the clause *true*, which is assumed to be present in any data base of clauses. Similarly, in proper subsumption a clause ψ is deleted if there is a clause φ in the bata base which is more general than ψ ($\psi \succ \varphi$). What is important to observe here is that in the latter, ψ is *replaced* by φ, not by the clause *true*. In other words, a proper subsumption step is in fact composed of two steps: first replacing ψ by a variant $\varphi\prime$ of φ if $\psi \succ \varphi$, then deleting $\varphi\prime$ since its variant φ is already in the data base. In the sequential case, the conventional view of subsumption is sufficient since the inference process assumes one data base. In distributed deduction, however, the separation of the two steps becomes crucial since φ, which subsumes ψ, may not belong to the local data base on which a deduction process is operating. Thus, an outright deletion of ψ at one node may result in the loss of information in the global data base, and the consequent loss of monotonicity.

With this understanding, we refine subsumption into two inference rules, the *replacement of a clause by a more general one* and the *subsumption of variants*:

Replacement Subsumption (R-subsumption):

$$\frac{A \cup \{\varphi_1, \varphi_2\}}{A \cup \{\varphi_1, \varphi_1'\}} \quad \varphi_2 \succcurlyeq \varphi_1,\ \varphi_1' \stackrel{\bullet}{=} \varphi_1$$

Variant Subsumption (V-subsumption):

$$\frac{A \cup \{\varphi_1, \varphi_2\}}{A \cup \{\varphi_1\}} \quad \varphi_2 \stackrel{\bullet}{=} \varphi_1,$$

where the ordering $\succcurlyeq$ is replaced by the ordering $\blacktriangleright$ in case of equations. In replacement-subsumption, a clause is replaced by a variant of the clause which subsumes it, and in variant-subsumption a variant is deleted. With these two basic inference rules the rule of *proper subsumption* becomes a derived inference rule, equivalent to the composition of a replacement subsumption step followed by a variant subsumption step:

Proper Subsumption (P-subsumption):

$$\frac{\dfrac{A \cup \{\varphi_1, \varphi_2\}}{A \cup \{\varphi_1, \varphi_1'\}} \quad \varphi_2 \succcurlyeq \varphi_1,\ \varphi_1' \stackrel{\bullet}{=} \varphi_1}{A \cup \{\varphi_1\}} \quad \varphi_1' \stackrel{\bullet}{=} \varphi_1$$

or, in short:

$$\frac{A \cup \{\varphi_1, \varphi_2\}}{A \cup \{\varphi_1\}} \quad \varphi_2 \succcurlyeq \varphi_1.$$

The conventional approach of sorting variants by age to decide which should be deleted can also fit in this framework by refining variant subsumption into

Ordered variant subsumption

$$\frac{A \cup \{\varphi_1, \varphi_2\}}{A \cup \{\varphi_1\}} \quad \varphi_2 \stackrel{\bullet}{=} \varphi_1, t_2 > t_1,$$

where t_1 and t_2 are the birth-times of φ_1 and φ_2 respectively. One can replace the V-subsumption step in the composition of the proper subsumption inference by an Ordered V-subsumption step and obtain the same proper subsumption inference rule. This is because the variant φ_1' resulting from the R-subsumption step has a

greater birth-time than φ_1, which is already in the data base.

5.4.4 The inference rules for distributed subsumption

In this section we present distributed versions of the variant, replacement, and proper subsumption inference rules. As in the sequential case, distributed proper subsumption is derived as the composition of distributed replacement and variant subsumption rules. If the derivation is sequential, the distributed inference rules reduce to their sequential (ordered) counterparts. Then by *distributed subsumption*, we mean distributed variant subsumption in case of variants and distributed proper subsumption if one clause properly subsumes another. The distributed subsumption rule allows subsumption between clauses at remote data bases while preserving both monotonicity and fairness. It is also not more difficult to implement distributed subsumption than sequential subsumption rules for sequential provers, and it has been already implemented in our distributed prover *Aquarius* (see Chapter 6).

Distributed variant-subsumption

The basic idea is to use extra information on the clauses to establish well-foundedness, even in the absence of a global clock. In addition to the birth-time, assigned to residents, we define the *inference-time* for raw clauses and the *reception-time* for inference-messages and travelling raw clauses. The *inference-time* of a raw clause φ generated at node p_k is the time at p_k's clock, when φ is produced by an inference step. A raw clause is assumed to have a birth-time ∞. The *reception-time* of a message, either inference message or travelling raw clause, received at a node p_k, is defined as the time at p_k's clock, when the message has been received at p_k. It is reasonable to assume that no two residents at the same node receive the same birth-time, no two raw clauses have the same inference-time at the same node and no two messages have the same reception-time at the same node.

We can now associate to every clause a measure, which depends on the attributes of the clause and on the node where the clause is being considered:

Definition 5.4.1 *The measure ds ("distributed subsumption") is defined as follows:*

- *for a resident φ at p_k, with birth-time t, $ds(\varphi, k) = (t, k, \_)$,*
- *for an inference message $< \varphi, p_i, a, x >$ received at node p_k, $ds(\varphi, k) = (x, i, y)$, where y is its reception-time at p_k,*
- *for a raw clause φ at p_k, born at p_k with inference-time s, $ds(\varphi, k) = (\infty, s, \_)$, and*
- *for a travelling raw clause φ, received at p_k with reception-time y, $ds(\varphi, k) = (\infty, y, \_)$.*

The third component is relevant only for inference messages. Next, we define an ordering to compare clauses based on this measure:

Definition 5.4.2 *Let $>$ denote the ordering on natural numbers extended with ∞ as the maximal element. Given two clauses φ and ψ in $S^k \cup M^k \cup CP^k$ at some node p_k, the ordering $\succ_{ds}$ is defined as follows: $\varphi \succ_{ds} \psi$ if and only if $ds(\varphi, k) \succ_{dm} ds(\psi, k)$, where $\succ_{dm}$ is the threefold lexicographic combination of the ordering $>$.*

Then, we define distributed variant-subsumption as follows:

Distributed Variant Subsumption (DV-subsumption):

$$\frac{A \cup \{\varphi_1, \varphi_2\}}{A \cup \{\varphi_1\}} \quad \varphi_2 \stackrel{\bullet}{=} \varphi_1,\ \varphi_2 \succ_{ds} \varphi_1,$$

where A is a set of clauses such as the union $S^k \cup M^k \cup CP^k$ at a node p_k.

Distributed variant-subsumption preserves monotonicity and fairness

We consider two variants φ and ψ. If the two clauses are two residents at the same node p_k, then the birth-time decides and the younger resident (the one whose birth-time is greater) is subsumed. In a sequential derivation, this is equivalent to the criterion of imposing the priority of forward subsumption over backward subsumption, which preserves fairness and thus completeness. Our distributed subsumption naturally incorporates this criterion. If one of the clauses is a (travelling) raw clause and the other one is either a resident or an inference message, then the raw clause is subsumed, since $\infty > t$ for all t. If both clauses are raw clauses or travelling raw clauses, they have the same birth-time ∞ and the ordering on the inference-times and reception-times is used to break the tie.

If the two clauses are both inference messages, we compare the time-stamps, so that the treatment of inference messages is consistent with the treatment of residents. But two inference messages may have the same time-stamps, if they are taken from clocks of different nodes. When this happens, we simply compare the second component of the measure ds, the indices of the nodes, and allow the one with the larger index to be subsumed. This does not cover still all possibilities. A node may receive two inference messages with different clauses, which originated from the same source. That is, a node may receive $m_1 =< \psi, p_i, a, x >$ and $m_2 =< \psi', p_i, a, x >$, both originated from the same node p_i with the same time-stamp. The two messages originally carried the same clause φ, which has been reduced by contraction inference rules to ψ in m_1 and to ψ' in m_2. In this case, the reception-times are compared and the message received first subsumes the other one.

The last and most interesting case is when one of the variants is an inference message and the other one a resident. Suppose two variants φ and ψ are residents at p_i and p_j respectively, and inference messages carrying φ and ψ arrive at p_j and p_i, respectively. We prove that distributed subsumption *deletes exactly one of the two variants*, thereby preventing the violation of monotonicity and ensuring the

maximal removal of redundancy.

Theorem 5.4.1 *Let variants φ and ψ be two residents at p_i and p_j respectively. Under distributed subsumption, exactly one of them will be subsumed.*

Proof: let φ be a resident at p_i with birth-time t_1 and ψ be a resident at p_j with birth-time t_2, such that φ and ψ are variants. If $i = j$, i.e., the two clauses are residents at the same node, we have $ds(\varphi, i) = (t_1, i, \_)$ and $ds(\psi, i) = (t_2, i, \_)$. Since t_1 and t_2 are taken at the same clock, it is $t_1 \neq t_2$. Therefore, the first component of the measure decides which one is subsumed: φ subsumes ψ if $t_1 < t_2$ and ψ subsumes φ if $t_2 < t_1$. If $i \neq j$, let $m_1 =< \varphi, p_i, a, t_1 >$ and $m_2 =< \psi, p_j, b, t_2 >$ be two inference messages from the two residents: m_1 is received at p_j at time y_1 of p_j's clock, while m_2 is received at p_i at time y_2 of p_i's clock. At node p_i, we have $ds(\varphi, i) = (t_1, i, \_)$ and $ds(\psi, i) = (t_2, j, y_2)$. At node p_j, we have $ds(\varphi, j) = (t_1, i, y_1)$ and $ds(\psi, j) = (t_2, j, \_)$. If $t_1 \neq t_2$, then either $t_2 < t_1$, and $< \psi, p_j, b, t_2 >$ subsumes φ at p_i, or $t_1 < t_2$, and $< \varphi, p_i, a, t_1 >$ subsumes ψ at p_j. Therefore, one and only one of the two residents is deleted. If $t_1 = t_2$, the indices i and j are compared. Since $i \neq j$, we have either $i < j$ or $j < i$, so that again one and only one of the two residents is deleted. □

This proves that distributed subsumption does not harm monotonicity by deleting all variants of a clause. The ordering $\succ_{ds}$ is well-founded since the ordering $\succ_{dm}$ is a lexicographic combination of well-founded orderings. Therefore, the violation of fairness showed at the beginning of this section (see Section 5.4.1) will not happen either.

Distributed proper subsumption

Similar to the sequential case, distributed replacement subsumption replaces a clause by a variant of a more general clause *at a local node*. Furthermore, since the data

involved include more than just the clauses themselves, other information such as birth-time, inference-time, etc, also need to be updated.

Distributed Replacement Subsumption (DR-subsumption):

$$\frac{A \cup \{\varphi_1, \varphi_2\}}{A \cup \{\varphi_1, \varphi_1'\}} \quad \varphi_2 \succcurlyeq \varphi_1,\ \varphi_1' \doteq \varphi_1.$$

Let z be the time at p_k's clock when the step is performed:

- if φ_2 is a resident with birth-time t and identifier a, $< \varphi_2, a, t > \in S_k$, it is replaced by $< \varphi_1', a, z > \in S_k$. Furthermore, if φ_1 is an inference message $< \varphi_1, p_i, b, x > \in M^k$, then its time-stamp is reset to ∞, i.e. we update it into $< \varphi_1, p_i, b, \infty > \in M^k$.
- If φ_2 is an inference message $< \varphi_2, p_i, a, x > \in M^k$, then it is replaced by $< \varphi_1', p_i, a, z > \in M^k$. The reception-time of $< \varphi_1', p_i, a, z >$ at p_k is also updated to z.
- If φ_2 is a raw clause with inference-time s, $< \varphi_2, s > \in CP_k$, it is replaced by $< \varphi_1', z > \in CP_k$.
- If φ_2 is a travelling raw clause with reception-time y, $< \varphi_2, y > \in CP_k$, it is replaced by $< \varphi_1', z > \in CP_k$.

Distributed proper subsumption is the composition of DR-subsumption and DV-subsumption:

Distributed Proper Subsumption (DP-subsumption):

$$\frac{\dfrac{A \cup \{\varphi_1, \varphi_2\}}{A \cup \{\varphi_1, \varphi_1'\}} \quad \varphi_2 \succcurlyeq \varphi_1,\ \varphi_1' \doteq \varphi_1}{A \cup \{\varphi_1''\}} \quad \varphi_1'' \; is \; \varphi_1 \; if \; \varphi_1' \succ_{ds} \varphi_1,\ \varphi_1'' \; is \; \varphi_1' \; if \; \varphi_1 \succ_{ds} \varphi_1'.$$

The distributed proper subsumption rule needs to fulfill two requirements:

1. Noting that the DR-rule may create copies of the same variant, the DP-rule must ensure that either $\varphi_1' \succ_{ds} \varphi_1$ or $\varphi_1 \succ_{ds} \varphi_1'$ is true. This will not only secure fairness, but also guarantees maximal removal of redundancy.

2. If φ_2 in the above inference step is a resident at p_k, then the resulting clause φ_1'' must also be a resident at p_k. This results in the preservation of monotonicity.

To demonstrate that DP-subsumption indeed satisfies these conditions, we analyze each case based on the type of the clauses:

- φ_2 was a resident and thus φ_1' is a resident $< \varphi_1', a, z > \in S_k$:
  - if φ_1 is also a resident at p_k, φ_1' is deleted, because its birth-time z is more recent than the birth-time of φ_1; (thus, φ_1'' is φ_1, which is already a resident)
  - if φ_1 is an inference message received at p_k, φ_1 is deleted, because its time-stamp has been reset by the DR-subsumption step to ∞; (thus, φ_1'' is $\varphi_1\prime$)
  - if φ_1 is a raw clause or travelling raw clause, φ_1 is deleted, because its birth-time is ∞, which is greater than z (thus, φ_1'' is $\varphi_1\prime$).
- φ_2 was an inference message $< \varphi_2, p_i, a, x > \in M^k$, and thus φ_1' is an inference message $< \varphi_1', p_i, a, z > \in M^k$ with reception-time also equal to z:
  - if φ_1 is a resident at p_k, φ_1' is deleted, because its time-stamp z is more recent than the birth-time of φ_1;
  - if φ_1 is another inference message received at p_k, their measures are compared and one of them will be deleted, as explained above, when discussing DV-subsumption;
  - if φ_1 is a raw clause or travelling raw clause, φ_1 is deleted, because its birth-time is ∞, which is greater than z.

- φ_2 was a raw clause (travelling raw clause), and thus φ_1' is a raw clause (travelling raw clause) with birth-time ∞ and inference-time (reception-time) z:
  - if φ_1 is a resident at p_k or an inference message received at p_k, φ_1' is deleted, because its birth-time is ∞;
  - if φ_1 is another raw clause (travelling raw clause), φ_1' is deleted, because its inference-time (reception-time) z is more recent than the inference-time (reception-time) of φ_1.

It follows that the definition of DP-subsumption can be rewritten as follows:

Distributed Proper Subsumption (DP-subsumption)

$$\frac{A \cup \{\varphi_1, \varphi_2\}}{A \cup \{\varphi_1'\}} \quad \varphi_2 \succ\!\!\!\bullet\, \varphi_1,\ \varphi_1' \stackrel{\bullet}{=} \varphi_1,\ \textit{if}\ \varphi_2 \in S^k,\ \textit{then}\ \varphi_1' \in S^k.$$

Finally, we define **Distributed subsumption**:

Definition 5.4.3 *Given two clauses (or equations) φ and ψ in $S^k \cup M^k \cup CP^k$ at some node p_k,* **D-subsumption** *is DV-subsumption ,if $\varphi \stackrel{\bullet}{=} \psi$, DP-subsumption, if $\varphi \succ\!\!\!\bullet\, \psi$ (or $\varphi \blacktriangleright\!\!\!\bullet \psi$).*

Distributed proper subsumption preserves monotonicity

We consider now the problem with monotonicity due to proper subsumption of residents by inference messages (see Section 5.4.2). When an incoming inference message properly subsumes a resident, DP-subsumption prescribes to replace the resident by the message. Therefore, no resident is lost, but rather replaced by a more general clause. For instance, we re-consider the example given in Section 5.4.2: let $P(f(x))$ and $P(g(y))$ be residents at nodes p_i and p_j respectively. A message carrying $P(f(x))$ is reduced to $P(x)$, e.g. by $f(x) \simeq x$, at some intermediate node, and it arrives to p_j as $P(x)$. Symmetrically, a message originally carrying $P(g(y))$

arrives to p_i as $P(y)$. If proper subsumption were applied as in sequential derivations, both residents $P(f(x))$ and $P(g(y))$ would be eliminated, with no guarantee that $P(x)$ is stored somewhere as resident. Under distributed subsumption, first DR-subsumption is applied and a variant $P(y')$ of $P(y)$ replaces $P(f(x))$ at p_i. Then, $P(y)$ is deleted by DV-subsumption. The overall effect of the DP-subsumption step at p_i is to replace $P(f(x))$ by $P(y')$. Similarly, at p_j, $P(g(y))$ is replaced by a variant $P(x')$ of the message $P(x)$, which is deleted. At a subsequent stage, another DV-subsumption step may delete one of the two variants $P(x')$ and $P(y')$:

P_i	P_j	time ↓
resident P(f(x))	resident P(g(y))	initial state
resident P(f(x)) message P(y)	resident P(g(y)) message P(x)	messages arriving
resident P(y')	resident P(x')	after distributed proper subsumption
... ...	... ...	later ...
resident P(y') message P(x')	resident P(x') message P(y')	messages arriving
resident P(y')	empty	after distributed variant subsumption

Remark that since DP-subsumption is regarded as replacement, when $P(y')$ replaces $P(f(x))$ at p_i, $P(y')$ gets the identifier of $P(f(x))$ and the current time at p_i's clock as birth-time. The same happens at node p_j. Thus $P(y')$ belongs to p_i and $P(x')$ belongs to p_j. This shows the difference between performing R-subsumption by an inference message on a resident and storing the inference message in the localized image set. If $P(y)$ were simply saved in the localized image set SH^i, it would be stored as clause belonging to p_j. Similarly, $P(x)$ would be stored in SH^j as clause belonging to p_i. However, there is no guarantee that $P(x) \in S^i$ and $P(y) \in S^j$. In other words, we may have a situation where the clauses $P(x)$ and $P(y)$, albeit stored, do not belong to any node and thus no node takes care of them,

e.g paramodulating into them. For this reason, we observed in Section 5.4.2 that saving received inference messages in the localized image sets is not a true solution to the subsumption problem. D-subumption, instead, provides a clean solution at the inference level.

These phenomena do not occur in a sequential derivation, where there is only one data base. But they may happen in a distributed derivation, where remote data bases interact through messages. The distributed scenario shows very effectively why viewing proper subsumption as replacement is preferable to viewing subsumption as deletion. Operationally, node p_i should not simply delete $P(f(x))$ upon receipt of $P(y)$, because p_i has not sufficient knowledge of the derivations at other nodes to establish that $P(y)$ is stored somewhere else. Conceptually, node p_i should not delete $P(f(x))$, because $P(f(x))$ is not trivially true. Rather, node p_i replaces $P(f(x))$ by $P(y)$, because $P(y)$ is more general than $P(f(x))$. Replacement-subsumption captures this concept. Also, our view of a proper subsumption step as the composition of a replacement-subsumption step and a variant-subsumption step appear very natural in the distributed context. Indeed, in the above example, global proper subsumption is achieved through two stages: first, $P(f(x))$ and $P(g(y))$ are replaced by $P(y')$ and $P(x')$, then, one of $P(x')$ and $P(y')$ is deleted. In the sequential case, it is not possible to appreciate the difference between proper subsumption as atomic deletion and proper subsumption as composition of replacement and deletion of a variant. In the distributed case, the difference is striking. Proper subsumption as atomic deletion violates monotonicity, whereas proper subsumption as composition does not and it mimics effectively the interaction of remote residents through messages.

We conclude by noticing that the problems with monotonicity may not occur in purely shared memory implementations of parallel theorem proving strategies, such as [94, 128]. In a purely shared memory approach, there is just one data base, as all the clauses are held in the shared memory. Each deduction process has direct access to the global data base. In this respect, parallel deduction in shared memory is much

closer to sequential deduction than distributed deduction. Indeed, the problem with proper subsumption as deletion is not exposed in a purely shared memory approach, because the data base is not distributed. Even the problem with subsumption of variants may not appear, because shared memory may force the sequentialization of certain steps, e.g. variant subsumption steps, which are concurrent in a distributed environment. For instance, if ψ and φ be variants, the step *"ψ subsumes φ"* needs write-access to φ and read-access to ψ, while the step *"φ subsumes ψ"* needs write-access to ψ and read-access to φ. The two steps are in read-write conflict on both premises. If the clauses are stored in a shared memory which does not allow two concurrent processes to have read-access and write-access respectively to a same clause, the two steps cannot be executed concurrently.

The restrictions to write-access that prevent the violations to monotonicity in shared memory, cause sequentializations of contraction steps, many of which do not affect monotonicity. The sequentialization of contraction steps determines the backward contraction bottleneck phenomenon, as we have already discussed in previous chapters. In summary, shared memory is safer than distributed memory with respect to subsumption and monotonicity, at the expense of concurrency. We have shown, however, that it is possible to perform subsumption and preserve monotonicity, while exploiting the larger degree of concurrency offered by distributed memory.

Chapter 6

The Aquarius prototype

In this chapter we describe our prototype distributed theorem prover **Aquarius** and some experiments. Aquarius implements a version of the Clause-Diffusion methodology with global contraction at the source by localized image sets. Neither travelling raw clauses nor wake-up calls are used. Each of the concurrent deduction processes in Aquarius executes an adapted version, called *Penguin*, of the code of the theorem prover *Otter* [98, 99]. Thus, Aquarius realizes the distributed version, according to the Clause-Diffusion methodology, of all the theorem proving strategies offered by Otter. Aquarius has been written in C [80] and PCN [27, 47], under the Unix operating system, for a network of Sun workstations. In such an environment, each Penguin process, or simply Penguin, runs on a different node of the network. Since Aquarius does not rely on any specific machine-dependent feature, it may be ported to any machine where C and PCN are available (see [47] for a list of the architectures currently supporting PCN).

In the following we illustrate first the *communication layer* and then the *deduction layer* of Aquarius. By communication layer, we mean the procedures, mostly written in PCN, which handle the communication between the Penguin processes in Aquarius. The communication layer forms the outermost shell of the code of

Aquarius. By deduction layer, we mean the Penguin program, i.e. the procedures, written in C and incorporating the code of Otter, that execute the selected theorem proving strategy at each node. The third section of the chapter is devoted to the user interface, i.e. the flags and parameters that the user may set to drive the working of the prover. We conclude with some analysis of the experiments conducted so far.

6.1 The communication layer

Communication among the Penguin processes is realized by using *streams*. A stream is a data structure that permits communication of messages from a producer to one or more consumers. We refer to [47] for the implementation of streams in PCN. In Aquarius, streams are used to form a *fully connected virtual topology*, where for any two Penguins i and j there is a stream with producer i and consumer j and a stream with producer j and consumer i. In the following we list first the streams defined in Aquarius and then we describe how communication is handled during a theorem proving session.

6.1.1 The streams for inter-process communication

We assume that n is the number of Penguin processes. Since each such process runs on a different computer, this parameters also tells the number of nodes involved. The initialization phase of Aquarius sets up the following streams:

- an $n \times n$ matrix of streams C for data messages, e.g. inference messages and new settlers,
- an $n \times n$ matrix of streams D for control messages, e.g. termination messages:
  - $<$ HALT$, i >$ meaning the successful termination of Penguin i,

  - $<$ TROUBLE, $i>$ meaning that Penguin i ran out of memory and
  - $<$ EARLY_STOP, $i>$ meaning that Penguin i halted upon hitting some threshold set by the user, e.g. the maximum number of clauses which may be generated,

- an $n \times 1$ array of status streams, one per Penguin, used to represent the status of the Penguin processes.

The entry $C[i, j]$ is the stream used for data messages from node i to node j. Similarly, $D[i, j]$ is reserved to control messages from node i to node j. The decision of defining two separate matrices for data messages and control messages stems from the consideration that PCN streams are first-in-first-out queues. If data messages and control messages were sent on the same streams, it would not be possible to give to control messages a higher priority than data messages, as it is desirable for termination messages, such as those listed above. A simple solution is to reserve separate streams for control messages.

Streams are connected by means of two primitives defined in PCN: the *merger* and the *distributor* [47]. A merger implements many-to-one communication by merging many input streams into one output streams. A distributor implements one-to-many communication by placing the contents of one input stream onto many output streams. Each Penguin process is equipped with a merger and a distributor: the Penguin process reads messages from the output stream of the merger and it writes messages to the input stream of the distributor. Then, for node i, all the streams $C[k, i]$, $0 \leq k \leq n-1$, are connected as input streams to the merger of Penguin i, so that all the streams $C[k, i]$ feed messages to Penguin i. All the streams $C[i, k]$, $0 \leq k \leq n-1$, are connected as output streams to the distributor of Penguin i, so that all the streams $C[i, k]$ receive the messages from Penguin i. The same solution, with separate merger and distributor at each node, is applied to connect the streams for control messages in matrix D. In this way, each Penguin process receives input messages from and sends output messages to all the other Penguin processes.

Each Penguin process is also associated with a status stream. Upon termination, a Penguin closes its status stream. All the status streams are in turn connected to a merger: when all the status streams are closed, this merger closes its output and this event signals to the user level the end of the whole computation. For instance, if Penguin i succeeds in finding a proof for the given problem, it broadcasts a message $<$ HALT$, i >$ and closes its status stream. When receiving $<$ HALT$, i >$ all the other Penguins close their status streams and Aquarius halts.

6.1.2 Communication during a theorem proving session

The main PCN module of the Penguin program invokes in parallel two procedures, called $receive()$ and $main\_infer()$:

$$receive() \;||\; main\_infer().$$

The $receive()$ procedure acts as a consumer on the input streams of the Penguin, i.e. the output of the merger which merges the $C[k, i]$'s, $0 \leq k \leq n-1$, and the output of the merger which merges the $D[k, i]$'s, $0 \leq k \leq n-1$. A consumer is a process which remains suspended until a message appears on its stream. Then, it gets the message, processes it and suspends again until a new message comes in. It halts when the stream is closed by the corresponding producer. Also, it may halt upon receiving special messages. In addition, $receive()$ acts as a producer on another stream, called the *inner stream*, which is introduced for the purpose of communication between the $receive()$ and the $main\_infer()$ procedures of the same Penguin. *Receive*() and $main\_infer()$ act as a producer and a consumer respectively on the inner stream. The following messages may be received by the $receive()$ module of Penguin i:

- $<$ HALT$, j >$: it means that Penguin j has halted successfully. *Receive*() calls the C function *store_special*(), puts a message HALT_RECEIVED on the inner stream and halts. The task of *store_special*() is to interrupt the work of the deduction layer at Penguin i.

- $<$ TROUBLE, $j>$: Penguin j has run out of memory. Each Penguin maintains an array with the current status of all the other Penguins. Penguin i resets to 0 the status of Penguin j, meaning that Penguin j halted unsuccessfully. If all the Penguins but Penguin i itself have halted unsuccessfully, $receive()$ halts, because Penguin i will not receive any more messages.

- $<$ EARLY_STOP, $j>$: it means that Penguin j has halted on a threshold. The treatment is the same as in the previous case.

- A data message $m =< c, j, a, t, d >$: c is a clause with identifier a and birth-time t, sent by Penguin j to destination d. The latter may be either the code ALL_PENGUINS, meaning that m is being broadcast (e.g. m is an inference message) or i, meaning m has been routed from j to i (e.g. m is a new settler). $Receive()$ invokes the C function $store\_mesg()$ to store m in the list $Inbound\_msgs$, where all the incoming data messages are stored at the level of the deduction layer. $Inbound\_msgs$ is a list of clauses, defined as C structures or records. The PCN message m is a string: in this phase, the contents of m is translated into the internal representation of clauses. Also, $receive()$ places a control message RECEIVED on the inner stream to $main\_infer()$. In this way, if $main\_infer()$ had suspended for need of clauses, it is informed that it can resume. It may happen that the node runs out of memory while the incoming message m is being stored in $Inbound\_msgs$: in such case, $receive()$ puts a control message TROUBLE on the inner stream and halts.

The $main\_infer()$ procedure passes control to the deduction layer, handles the information returned by the deduction layer and sends messages according to the requests from the deduction layer. The latter is invoked by calling the procedure $infer()$, which represents the outermost level of C code and corresponds to the $main$ of the Otter program. $Infer()$ may return one of the following codes, where we assume we are considering Penguin i:

- PROOF: it means that the $infer()$ process has found a proof. $Main\_infer()$

broadcasts the control message $<$ HALT, $i>$ and then halts.

- EARLY_STOP: it means that the $infer()$ process has terminated on a threshold. $Main\_infer()$ broadcasts the control message $<$ EARLY_STOP, $i>$ and then halts.

- HALT_RECEIVED: it means that the $infer()$ process has been interrupted by *store_special*(), because another Penguin j has halted successfully. Thus $main\_infer()$ halts.

- TROUBLE: it means that the $infer()$ process has run out of memory. $Main\_infer()$ broadcasts the control message $<$ TROUBLE, $i>$ and then halts.

- WAIT_TO_RECEIVE: it means that the $infer()$ process has terminated, because its *Sos* (Set of Support) list is empty, i.e. $infer()$ has processed all the clauses it has. $Main\_infer()$ suspends until it receives a message on the inner stream. As soon as it gets one, it proceeds as follows, depending on the type of the message:

  - RECEIVED: it means that *receive*() has actually received data messages, i.e. clauses, and appended them to $Inbound\_msgs$. Thus $main\_infer()$ resumes and it will call $infer()$ again. Those clauses will be moved from $Inbound\_msgs$ to *Sos* and the inference mechanism will restart.

  - HALT_RECEIVED: it means that *receive*() has received a message $<$ HALT, $j>$, saying that another Penguin j has halted successfully. Thus $main\_infer()$ halts.

  - TROUBLE: it means that the *receive*() process has run out of memory. $Main\_infer()$ broadcasts the control message $<$ TROUBLE, $i>$ and then halts.

- A number m, $m > 0$: it means that the $infer()$ process has terminated, because it needs to send m clauses in form of data messages. The clauses to

be sent are stored in the list *Outbound_msgs*, which is used to store all the outgoing messages at the level of the C data structures. $Main\_infer()$ invokes the procedure *send_msgs*(), which extracts m clauses from *Outbound_msgs* and sends them as messages on the matrix C. The destination field of a clause extracted from *Outbound_msgs* tells whether the clause is to be broadcast or routed to a specific node. Then, *send_msgs*() calls recursively $main\_infer()$, which will call $infer()$ again and the inference process will restart.

In summary, if $main\_infer()$ invokes $infer()$, the deduction layer does not return control to the communication layer until it needs to do so, either because the *Sos* is empty, or because the deduction layer needs to send data messages or upon some termination signal. This organization of the interaction between the communication layer and the deduction layer, although satisfactory in principle, may turn out to be disappointing in practice. The reason is that the current implementation of PCN gives priority to the execution of C code over the execution of PCN code [120]. The Penguin program invokes $receive()$ and $main\_infer()$ by a call $receive()$ || $main\_infer()$. In turn $main\_infer()$ calls the C module $infer()$. The run time of PCN initiates both processes $receive()$ and $main\_infer()$ and they are executed at the same node by interleaving them. The problem is that in the current run time of PCN this interleaving is not as fair as one may expect: whenever C code is enabled to run, the run time of PCN gives priority to the C code. Therefore, since $infer()$ is written in C, $receive()$ is executed only when $infer()$ relinquishes control. It follows that the activity of receiving messages may be too slow with respect to the inference mechanism. To compensate, at least partially, for this unbalance, we have defined another procedure, called $small\_infer()$, to be invoked by $main\_infer()$ in alternative to $infer()$. $Small\_infer()$ differs from $infer()$ because it returns control to the communication layer much more often. Namely, $small\_infer()$ returns whenever it has completed the processing of a *given_clause*. (See Section 3.5.2 for the definition of *given_clause* in Otter. The treatment of the *given_clause* in Penguin will be described in the next section.) In addition to all the codes returned

by $infer()$, $small\_infer()$ may return CONTINUATION: then $main\_infer()$ simply calls itself recursively, determining a new call to $small\_infer()$ and thus the processing of the next *given_clause*.

6.2 The deduction layer

The basic deduction cycle in Penguin is very similar to that of Otter, which was described in Section 3.5.2. In this section we focus mostly on some of the main differences between Penguin and Otter.

The Aquarius program is invoked with two main parameters: the name of the input problem, e.g. *sam* (from the SAM Lemma), and the number of requested Penguin processes. If n processes are requested, it is expected that n input files, e.g. $sam.in.0$, $sam.in.1 \ldots sam.in.n-1$ are in the current directory, with the meaning that $sam.in.i$ is the input file for the i-th Penguin. The program itself will create n output files, e.g. $sam.out.0$, $sam.out.1 \ldots sam.out.n-1$ and as many log files, e.g. $sam.err.0$, $sam.err.1 \ldots sam.err.n$–1: $sam.out.i$ contains the trace of the execution of the i-th Penguin, while $sam.err.i$ contains errors and exceptions raised at the i-th Penguin.

The format of an input file is the same as in Otter. A typical input file contains up to four list of clauses, i.e. the lists $Usable$, Sos, $Demodulators$ and $Passive$ introduced in Section 3.5.2, and commands to set options. In general, it is expected that only one input file, e.g. $sam.in.0$, contains the complete input, comprising both commands and clauses. All the remaining input files include the commands but not the clauses. The Penguin which reads the input clauses, e.g. Penguin 0, broadcasts them to all the other Penguins. The reason for this is that Penguin 0 decides for each input clause which Penguin it belongs to. If more than one Penguin, e.g. both Penguin 0 and Penguin 1, read the input clauses, it may happen that Penguin 0 assigns a certain input clause ψ as resident to Penguin 0 itself, and, by the same

algorithm, Penguin 1 assigns ψ to Penguin 1. This would defeat the rule that each clause is a resident at only one node. Another advantage of having just Penguin 0 reading the input is that it may inter-reduce the input clauses and broadcast them in reduced form. Thus the task of inter-reducing the input is performed at just one node. We emphasize, though, that Penguin 0 broadcasts all the input clauses to all the nodes right after inter-reducing them. This has two consequences. First, all the input clauses are physically allocated at all the nodes, even if they may be logically partitioned by ownership. Second, when an input clause is selected as *given_clause* it is not broadcast as inference message, because it has been already broadcast in the input phase.

The general rule that each clause belongs to only one process has exceptions. For instance, the reflexivity axiom $x = x$, which the inference mechanism of Otter and hence Penguin needs be given explicitly in the input, is set to be a resident at all the nodes. Also, in equational problems the targets are treated as residents at all the nodes.

In addition to $Usable$, Sos, $Demodulators$ and $Passive$, each Penguin process maintains two more lists of clauses: $Inbound\_msgs$ and $Outbound\_msgs$, which contain respectively received clauses and clauses to be sent as messages. At each iteration of the main loop in $infer()$ or at each execution of $small\_infer()$, Penguin first moves to Sos all the clauses in $Inbound\_msgs$, if there are any. Then, it checks whether there are any clauses in $Outbound\_msgs$ and if so, it returns to the communication layer the number of messages pending in $Outbound\_msgs$ to be sent. Otherwise, it extracts a *given_clause* from Sos. If the *given_clause* is a resident, it appends a copy of it to $Outbound\_msgs$, so that it will be broadcast as inference message. Each Penguin broadcasts only its own residents, because the virtual topology in Aquarius is fully connected, and therefore there is no need for a Penguin to forward received inference messages. No clause is broadcast twice: for instance, an input clause is not broadcast when selected as *given_clause*, because it has been already broadcast in the input phase. Special clauses which are set to be

residents of all Penguins, such as the reflexivity axiom and equational targets, are also not broadcast when selected as *given_clause*. Next, it appends the *given_clause* to $Usable$ and uses it to perform all possible expansion inference steps with other clauses in $Usable$. All the generated raw clauses are first pre-processed, i.e. subject to forward contraction, then appended to *Sos* and post-processed, i.e. used for backward contraction.

The *given_clause* is appended to $Usable$, regardless of whether it is a resident or not: in this way each Penguin saves the received inference messages and forms its own *localized image set*. There is no ad hoc data structure to implement the localized image set: it is simply realized within the local data base of the Penguin. The received inference messages, just like residents, are stored in *Sos* before being selected as *given_clause* and in $Usable$ afterwards. They also appear in $Demodulators$ if they are simplifiers. The union of these lists forms the local data base. Since they store both residents and clauses brought in by inference messages, they also realize the localized image set.

Expansion inference steps are subdivided among the Penguin processes based on ownership, according to the Clause-Diffusion methodology (see Section 4.6.3). In order to describe how this is done, we recall that a significant feature that Penguin inherits from Otter is the usage of indexing techniques for fast retrievals of terms (atoms and thus literals are treated as terms). When performing inferences, we need to retrieve all the terms with a certain property. For instance, expansion inferences require to retrieve terms in the data base, which are unifiable with a given term. Forward contraction inferences require to retrieve anti-instances: when trying to reduce or subsume a given term, we look for more general terms in the data base. Backward contraction inferences require to retrieve instances: when trying to apply the *given_clause* to reduce or subsume the clauses in a data base, we look for instances of the terms of the *given_clause* in the data base. Otter employs *path-indexing* [118] for retrieval of unifiable terms and instances, and *discrimination-net-indexing* [30, 118] for retrieval of anti-instances [96]. (This is the basic setting, which the user

may modify by setting specific options.) All clauses are indexed, i.e. their terms are inserted in path-indexes or discrimination nets. Then, when trying inference steps, the appropriate indexes are consulted: for instance, for resolving with a positive literal A, one consults a path-index to retrieve all negative literals unifiable with A. Back-pointers from literals to clauses allow one to know to which clauses a retrieved literal belongs to. In the following brief description of the subdivision of inferences, "retrieval" refers to this kind of mechanism:

- **Paramodulation** inferences are either $para\_into(given\_clause)$ inferences, where the $given\_clause$ is paramodulated into or $para\_from(given\_clause)$ inferences, where the $given\_clause$ paramodulates into other clauses. In Penguin, $para\_into(given\_clause)$ is invoked only if the $given\_clause$ is a resident. Also, when the $given\_clause$ is indexed, its terms are inserted in the index of the terms to be paramodulated into only if $given\_clause$ is a resident. Therefore, when $para\_from()$ will be called with another $given\_clause$, the latter will be used to paramodulate into terms from residents only.

- **Binary resolution**: for each literal in $given\_clause$, $bin\_res(given\_clause)$ retrieves the literals of opposite sign that unify with it. In Penguin, negative literals in the $given\_clause$ and negative literals retrieved from the indexes are considered only if they belong to residents.

- The implementation of **hyperresolution, negative hyperresolution and unit-resulting resolution** (see Appendix A.1) in Otter proceeds in two phases. First, it assumes that the $given\_clause$ will play the role of the nucleus. It retrieves literals from other clauses, i.e. the potential satellites, which unify with the literals in the $given\_clause$. If it finds sufficiently many satellites, the step commits. In Penguin, this phase is performed only if the $given\_clause$ is a resident. In the second phase, it assumes that the $given\_clause$ will be a satellite. It retrieves a literal from another clause, i.e. the potential nucleus, which unifies with a literal in the $given\_clause$. In Penguin, literals retrieved to be literals of a nucleus are considered only if they occur in a resident. If it

finds a resident nucleus, it looks for the other satellites and the step commits upon finding them.

These subdivisions represent the main modifications to the expansion part of the inference mechanism. For the contraction part, since each Penguin accumulates a localized image set, each Penguin basically treats its local data base in the same way as Otter treats its single data base. Thus, the two main modifications with respect to Otter are the implementation of *distributed subsumption* (see Section 5.4), and *Discard Messages* (see Section 5.3.1), the inference rule which discards redundant inference messages and utilizes inference messages to update localized image sets. Penguin maintains its localized image set by using both *direct contraction* and *update by inference messages* (see Section 4.2.2).

Otter already distinguishes forward and backward subsumption and implements the criterion that forward subsumption has priority over backward subsumption. Thus, no modification has been necessary for subsumption between residents. Second, since in Otter each raw clause is pre-processed, i.e. forward-contracted, right after generation, it is never necessary to compare for subsumption two raw clauses. Third, Aquarius does not use travelling raw clauses. Therefore, the modifications have been concentrated on subsumption between residents and inference messages, which is realized according to our treatment in Section 5.4.

During the pre-processing of a received inference message, we apply forward subsumption *before demodulation*, i.e. simplification, in order to prevent the following phenomenon. We consider the situation where the local data base at Penguin i stores $< l \simeq r, p_i, a, t >$, and an inference message $< l' \simeq r', p_j, b, t' >$, with $l' \simeq r'$ a variant of $l \simeq r$, is received at Penguin i. The abstract definition of simplification (see for instance Section 2.9) does not allow simplification between variants. For $l \simeq r$ to simplify $p \simeq q$ into $p[r\sigma]_u \simeq q$, it is required that either $p \triangleright l$ or $q \succ p[r\sigma]_u$: neither of

these two conditions is satisfied, if $l \simeq r$ and $p \simeq q$ are variants [1]. Since the definition of simplification does not allow deletion of variants, we have treated the problem of how to eliminate variants correctly in a distributed derivation in our treatment of distributed subsumption (see Section 5.4). Accordingly, we have implemented deletion of variants within distributed subsumption. However, demodulation in Otter, and hence in Penguin, includes simplification between variants. It follows that if demodulation is applied before forward subsumption, during the pre-processing of the message $< l' \simeq r', j, b, t' >$, $l \simeq r$ reduces $l' \simeq r'$ to a trivial equation, regardless of birth-times. In other words, the resident variant always deletes the message variant. This may prevent the effective deletion of variants, as we have explained in Section 5.4.2. One possible approach to avoid this problem is to modify the code for demodulation. However, this should be done without embedding a duplication of the test for variants in the simplification process. Since we had already implemented subsumption of variants as part of distributed subsumption, we have preferred the simpler solution consisting in applying forward subsumption before demodulation: this ensures that if an incoming message is a variant of an already stored clause, the two will be compared according to the definition of distributed subsumption. Also, if the incoming message subsumes as variant a clause in $Usable$, the clause in the message will be appended directly to $Usable$, and not to Sos, on the ground that its variant had already been extracted from Sos. The drawback of performing forward subsumption before demodulation is that if the clause in the message is not forward-subsumed, and demodulation reduces it afterwards, it may be necessary to repeat the forward subsumption test on the reduced form.

A similar concern is posed by the order of application of Discard Messages and demodulation. Discard Messages is also applied as part of the pre-processing of a received inference message. It checks whether the localized image set contains a clause with the same global identifier as the clause in the message: if so, the two clauses and their birth-times are compared, as explained in Section 5.3.1, and

[1] Indeed, simplification between variants is not proof-reducing. Deletion of variants by subsumption is justified in terms of elimination of redundant clauses.

the greater clause is deleted. Thus, the rule is applied in both directions: to use an incoming message to delete a clause in the localized image set and vice versa. We apply Discard Messages before all the other contraction rules, especially *before demodulation*. The reason is the following. Assume that Penguin i has the equation $< l \simeq r, p_j, a, t >$, a resident of Penguin j. Next, Penguin i receives and pre-process an inference message $< l \simeq r', p_j, a, t' >$ from Penguin j, where r' is the reduced form of r. If demodulation were applied before Discard Messages, $l \simeq r$ would reduce $l \simeq r'$ to $r \simeq r'$ and then $l \simeq r$ would be deleted by Discard Messages because of $r \simeq r'$. The result is non-monotonic, since both $l \simeq r$ and its reduced form $l \simeq r'$ are lost. The cause of the problem is that $l \simeq r$ is applied to simplify its reduced form $l \simeq r'$. Such a phenomenon could never happen in a sequential computation, where the unique copy of $l \simeq r$ would have been deleted upon creation of $l \simeq r'$. In a distributed derivation, it is possible, and it can be prevented by simply applying Discard Messages before demodulation, so that $l \simeq r'$ correctly eliminates $l \simeq r$.

More features of the Penguin program will be described in the next section as part of the user options.

6.3 The user interface

The user may set a very high number of options, which affect the inference mechanism, e.g. by determining which inference rules are active, the search plan at each node and the communication layer. Boolean-valued options are called *flags* and integer-valued options are called *parameters*. Flags are turned on and off by inserting in the input files the commands $set(flag\_name)$ and $clear(flag\_name)$ respectively. Integer values are assigned to parameters by the command $assign(parameter\_name, value)$. The syntax of the commands and all the flags and parameters of Otter are inherited by Aquarius. We refer to [98, 99] for the flags and parameters of Otter. An important feature of Aquarius is that the user may give different options patterns to different Penguin processes, because each

Penguin process reads its own input file. This gives to the user extra flexibility.

Currently, Aquarius has 99 flags and 22 parameters, for a total of 121 options. The following new options have been added in Aquarius:

- PRIORITY_MSGS

  If it is on, the PCN module $main\_infer()$ invokes $small\_infer()$, otherwise it invokes $infer()$. (See Section 6.1.2.) Default is off.

- POST_PROC_NS_BEFORE_SEND

  Assume that Penguin i has generated a raw clause, and, after pre-processing it, decides to send it as a new settler ψ to Penguin j. If this option is on, Penguin i appends temporarily ψ to its Sos and post-process it. Clause ψ is appended to Sos for the purpose of post-processing only. When Penguin i selects ψ as $given\_clause$ from Sos, it does not use it for inferences as a standard $given\_clause$, but simply sends it as new settler to Penguin j. If this option is off, a new settler is sent right after having been pre-processed, without post-processing. Default is off. If KNUTH_BENDIX (an option inherited from Otter) is on, the selected inference mechanism is that of Unfailing Knuth-Bendix equational completion. In this context, POST_PROC_NS_BEFORE_SEND is turned on automatically, because it has turned out experimentally that in equational problems it is better to post-process new settlers (i.e. use them for backward contraction) before sending them.

- EAGER_BD_INF_MSGS

  If it is on, a resident $given\_clause$ is broadcast as inference message before being used for expansion. If it is off, it is broadcast after the expansion phase. Default is on.

- STAND_ALONE

  If it is on, each Penguin finds in its input file the complete input, both commands and clauses, and then it works by itself as a sequential prover; no

messages are sent and received. One purpose of this option is to allow the user to try in parallel different strategies on a given problem. Another application of this mode is to give to each Penguin a different input and have the n processes working in parallel on different problems. For instance, one may want to give to each Penguin a different lemma from a large problem and have the lemmas proved independently. Default is off: in standard mode the Penguins work cooperatively.

- Flags related to the inference mechanism:

  - SATURATION

    It is basically a renaming of the KNUTH_BENDIX option of Otter (which is turned on automatically if SATURATION is on). If it is on, the prover uses the Unfailing Knuth-Bendix strategy to try to generate a saturated set of equations. Default is off.

  - PART_FACTORS

    If it is on, each Penguin generates only the factors of its own residents, while it does not generate factors of received inference messages. The default is on, according to the general rule that expansion inference steps, which include factoring steps, are subdivided among the processes based on ownership of clauses. On the other hand, it is better for some problems to let each process generate all the factors. Therefore, this restriction has been made into an option rather than being built-in in the prover.

- Flags related to the allocation of clauses: in this context, "allocation" means "logical allocation" rather than "physical allocation". The "allocation algorithm" is the algorithm which takes as input a clause and returns as output the number of the node the clause will be a resident of.

  - ALTERNATE_FIT, HALF_ALT_FIT, ALT_FIRST_FIT and FIRST_FIT

    See Section 4.4. The default is ALTERNATE_FIT. In addition, the parameter MAX_SC_W_ALT represents the maximum number of generated

clauses which a Penguin may allocate to itself by using ALT_FIRST_FIT or FIRST_FIT. If either ALT_FIRST_FIT or FIRST_FIT is on, they are turned off automatically and replaced by ALTERNATE_FIT as soon as the number of clauses settled at the node so far exceeds the value of the parameter MAX_SC_W_ALT. The purpose of this parameter is to prevent the situation where each Penguin keeps all its generated clauses. However, if such a situation is deemed desirable, the user may simply turn on an appropriate flag, e.g. ALT_FIRST_FIT, and set the value of the parameter MAX_SC_W_ALT to MAX_INT, i.e. the maximum integer defined in the prover.

– OWN_IN_USABLE (OWN_IN_SOS)

  If it is on, each Penguin keeps as residents all the input clauses in $Usable$ (Sos), regardless of the allocation policy. Default is off. These two options have proved to be very valuable experimentally, as was largely expected. For instance, basic axioms, such as associativity, commutativity or distributivity, are generally given in the input $Usable$ and it is clearly very useful, sometimes necessary to obtain a proof in reasonable time, that such axioms are owned by all the Penguins.

  Similar options for input $Passive$ and input $Demodulators$ have not been inserted so far, because ownership does not affect the usage of clauses in those lists. Clauses in the $Passive$ list are used only for forward subsumption and unit-conflict (that is a resolution step between two unit clauses, which generates the empty clause). Clauses in the $Demodulators$ list are used only for demodulation, i.e. simplification. Forward subsumption, unit-conflict and demodulation are not restricted based on ownership of clauses: each Penguin tries to use as subsumers, as demodulators and for unit-conflict all the clauses it has, regardless of whether they are residents or not.

– OWN_FACTORS

  If it is on, each Penguin keeps as residents the factors of its residents

regardless of the allocation policy. Default is on.

- OWN_NFR

  If it is on, each Penguin keeps as residents the equations generated from its residents by the $new\_function\_rule$ (i.e. "splitting" as defined in the original paper by Knuth and Bendix [84]), regardless of the allocation policy. Default is on.

- REALLY_DELETE_MSGS

  In Otter, a clause subject to a contraction step is not physically removed as part of the contraction step. Rather, it is appended to a list of $Hidden\_clauses$, i.e. clauses that are logically, but not physically deleted. One reason to do this is that it makes possible, after a refutation has been obtained, to print a full trace of the proof, including also clauses which contributed to the proof and later contracted. Since the prover may run out of memory, by not deleting clauses physically, Otter features the option REALLY_DELETE_CLAUSES: if it is on, all the clauses in $Hidden\_clauses$ are deleted at the end of each post-processing phase. Default is off. REALLY_DELETE_MSGS is modeled on REALLY_DELETE_CLAUSES. Sending a clause c as a message comprises the following steps: extracting c, which is a C structure or record, from the list $Outbound\_msgs$, translating it into a string, since messages at the PCN level are tuples of strings, and appending the structure c to a list $Hidden\_msgs$. If REALLY_DELETE_MSGS is on, the clauses in $Hidden\_msgs$ are deleted at the end of each post-processing phase, otherwise they are kept. As a default REALLY_DELETE_MSGS is on, because it turns out that sent messages are rarely needed to trace the proof. The reason is the following. Clauses in $Outbound\_msgs$ are either inference messages or new settlers. Each Penguin sends as inference messages only its own residents. When a Penguin processes one of its residents c as $given\_clause$, it makes a copy of c, appends it to $Outbound\_msgs$ for sending, appends c to $Usable$ and uses it for expansion inferences. Thus, all the inference messages in $Outbound\_msgs$ are in actuality

copies of clauses in $Usable$. On the other hand, when a Penguin sends a new settler to be a resident at another node, it does not keep a copy of it. However, a new settler is not used for inferences at the node where it is generated but not kept as resident, except for its own pre-processing. The new settler will be used, to contract other clauses and for expansion, at the node where it is a resident. Therefore, it is rare that a new settler as such appears in the trace of a proof. It may happen, though, if POST_PROC_NS_BEFORE_SEND is on, because in such case new settlers are used for post-processing before being sent. If it happens, the user may still reconstruct the complete trace of the proof by turning REALLY_DELETE_MSGS off.

- In Otter, and hence in Penguin, the default way to select the *given_clause* in Sos is to pick the clause with minimum weight, i.e. the smallest. The weight of a clause is the number of its symbols, unless the user modifies it by giving *weight templates* [98, 99]. Other options in Otter allow the user to adopt different criteria, e.g. the flags SOS_QUEUE and SOS_STACK or the parameter PICK_GIVEN_RATIO: if the latter is set to n, the prover picks n times the lightest clause, then the first one, then again n times the lightest clause, then the first one and so on. Two more flags are added in Aquarius for this purpose:

  – SOS_QUEUE_MOD

    If this option is on, the Sos is treated as a queue, sorted by an ordering on the identifiers of clauses. More precisely, for a resident c, we consider the pair (lid, id), where lid is the "logical identifier" of c, i.e. what we have called the "identifier" throughout our treatment of the Clause-Diffusion methodology, and id is the "inner identifier" of c, i.e. the identifier defined in Otter and used by the low level procedures of the prover. If c is not a resident, we consider the pair $(lid, id/No\_of\_nodes)$, where No_of_nodes is the number of active nodes. The reason for dividing id by No_of_nodes is that a clause c resident at Penguin i and received at Penguin j as

inference message, gets at Penguin j an id which is in general much greater than the one it got at Penguin i, because of the communication delay. Then, Sos is sorted by the two-fold lexicographic combination of the ordering on integers applied to these pairs of numbers.

- SOS_D_LIGHT

  If this option is on, the $given\_clause$ at Penguin i is extracted as follows: select the lightest clause in Sos, which is a resident of Penguin i, select the lightest clause in Sos, which is not a resident of Penguin i, compare them by the lexicographic combination defined above and pick the smallest of them.

- IN_MSG_QUEUE and IN_MSG_STACK

  OUT_MSG_QUEUE and OUT_MSG_STACK

  Handle $Inbound\_msgs$ and $Outbound\_msgs$ as a first-in first-out queue or a last-in first-out stack respectively. The default is off for all of them. The treatment of $Inbound\_msgs$ and $Outbound\_msgs$ is similar to the treatment of Sos, which Aquarius inherits from Otter. As a default, the smallest clause is extracted first. The user may modify this behaviour by setting the above flags.

- PRINT_SENT and PRINT_RECEIVED

  PRINT_ALLOC and PRINT_UPDATES

  cause a Penguin to print a message in the output file for each clause sent, each clause received, each allocation decision and each application of the *Discard Messages* inference rule (see Section 5.3.1). They are all on by default. Also, ifthe Otter option VERY_VERBOSE is on, they are all turned on automatically. For instance, if the user gives $set(very\_verbose)$ and $clear(print\_sent)$, the latter will be on nonetheless.

6.4 Experiments

In Table 1 and Table 2, we show the performances of Aquarius on 33 problems. Aquarius-n is Aquarius with n processors and we give the run times for Aquarius-1, Aquarius-2 and Aquarius-3. The run time of Aquarius is the run time of the fastest Penguin to succeed. In turn, the run time of a Penguin process is the time elapsed since $main\_infer()$ called $infer()$ (or $small\_infer()$) for the first time, until both $main\_infer()$ and $receive()$ have terminated. Thus, it includes both the time spent in inferences and the time spent in communication. However, it does not include the initialization time spent to set up the Penguin processes at the nodes and the streams inter-connecting them. Nor does it include the time spent to close the PCN processes, at the levels higher than $main\_infer()$ and $receive()$, upon termination. Also, when Aquarius, e.g. Aquarius-3, is invoked on a workstation, Penguin2 is executed on the workstation where Aquarius was invoked, while Penguin1 and Penguin0 are executed on two other workstations specified in the command. The user will not see any output from Penguin1 and Penguin0 until Penguin2 terminates. Therefore, if Penguin0 succeeds first, the user will see the program running until Penguin2 has received HALT from Penguin0. For all these reasons, the turn-around time observed by a user is usually longer than the run time.

Aquarius-1 does not perform any communication. However, Aquarius-1 is generally slower than Otter, which indicates that the overhead induced merely by having linked the PCN part with the C part is not irrelevant. Each node is a Sun 4 sparcstation. So far we have been able to experiment with up to 3 of them. They communicate over the Ethernet of the department. The sparcstations used for our experiments were not isolated from the rest of the network, i.e. the network was carrying the usual amount of traffic. Also, other users were logged on the sparcstations involved in the experiments and thus other programs were running when Aquarius was. Therefore, the following run times represent the performances under realistic working conditions. All these problems come from the data base of problems for

Problem	*Aquarius-1*	*Aquarius-2*	*Aquarius-3*
andrews	18.00	25.40	24.39
apabhp	11.86	18.11	14.18
bledsoe	12.29	21.53	23.00
boolean-assoc	18.20	21.05	21.18
cd12	104.18	50.98	47.56
cd13	98.79	45.32	51.07
cd90	3.10	0.63	11.87
cn	5.04	8.63	14.50
ec	3.03	1.96	1.77
grp-div	0.37	0.73	0.58
kbgroup	0.27	0.94	0.89
kb-comm	0.65	0.94	1.92
kb-entropic	0.21	0.24	0.20
kb-x2-r	1.79	2.14	5.50
imp1	6.63	2.64	3.54
imp2	7.25	3.31	7.43
imp3	32.05	17.92	38.89

Table 1: Experiments with Aquarius (all times are expressed in seconds).

Otter. For each of them, we give a short explanation or a reference:

- andrews: it is a famous challenge problem for theorem provers given by Peter Andrews in 1979. It was reported first in [25] and then in [104]. It has been solved by several approaches (e.g. [26, 54, 56, 108]). The inference rules used in Aquarius are binary resolution and factoring with subsumption (forward and backward subsumption are always included, unless otherwise stated).
- apabhp: it is the so-called "blind hand problem", which we found in [94]. The

Problem	*Aquarius-1*	*Aquarius-2*	*Aquarius-3*
lifsch	2.14	4.96	3.22
luka5	844.20	299.24	1079.45
mission	0.33	0.91	1.04
mv	7.24	6.55	3.60
pigeon	8.21	7.66	8.14
robbins	0.68	1.16	1.16
robbins2	21.62	22.91	24.12
salt	3.89	4.45	5.49
sam-hyp	6.35	5.40	3.90
steam	1.01	1.59	0.57
str-bws	2.32	2.29	2.38
subgroup	15.55	9.36	17.40
tba-1	0.23	0.29	0.86
tba-gg	0.55	0.91	0.84
x2-quant	0.25	0.44	0.97
w-sk	3.50	3.52	3.34

Table 2: Experiments with Aquarius (all times are expressed in seconds).

inference rules are hyperresolution with forward and backward subsumption. The PICK_GIVEN_RATIO is 3 for Aquarius-1 and Aquarius-2, while it is 4 for Aquarius-3.

- bledsoe: it is a simple variant of the theorem that the sum of continuous functions is continuous [18, 94]. The inference rules are binary resolution, factoring and forward subsumption, but not backward subsumption. Also, Otter-style heuristics, i.e. discarding generated clauses with more than 40 symbols or containing certain patterns of nested function symbols, were used.

- boolean-assoc: prove that Boolean algebras are associative. Even if the formulation of the problem is naturally equational, here it is presented in clausal form, with the equality axioms, to be a test for hyperresolution.

- cd12, cd13 and cn: these are problems in the two-valued sentential calculus of Lukasiewicz; cd12 and cd13 are taken from [94]. The main inference rule is hyperresolution and the target theorems are put in the *Passive* list. Backward subsumption is not used for cn. Generated clauses with more than 20 symbols (16 for cn) were discarded. PICK_GIVEN_RATIO is 3 in cn, is not used in cd12 and in cd13 it is 3 for Aquarius-3, whereas it is not used for Aquarius-1 and Aquarius-2.

- cd90: prove that one of the axioms in Kalman's original axiomatization of the left group calculus can be derived from the others [73, 94, 100]. It is done by hyperresolution, with PICK_GIVEN_RATIO set to 2 for Aquarius-1 and Aquarius-2, to 4 for Aquarius-3.

- ec: it is a problem in the equivalential calculus of Lesniewski. The strategy is the same as in cn.

- grp-div, kbgroup and kb-entropic: in these problems, Unfailing Knuth-Bendix completion (UKB) is used to generate confluent systems from the given sets of equations; grp-div is an axiomatization of group theory using right division, proposed as a test in [79], kbgroup and kb-entropic are group theory [84] and entropic groupoid [62].

- kb-comm and kb-x2-r: these are two well-known theorem proving problems for UKB; kb-comm is the commutator problem in group theory (e.g. [2]), while kb-x2-r is proving that $x^2 = x$ implies commutativity in ring theory [59, 93].

- imp1, imp2 and imp3: these are problems from a series intended to show that a single axiom axiomatizes the implicational propositional calculus, by showing that the other axioms can be derived from that one [94, 102]. The

main inference rule is hyperresolution, while backward subsumption is not included. Generated clauses with more than 20 symbols are discarded.

- lifsch: it is a challenge problem for resolution-based theorem provers, suggested by Vladimir Lifschitz.

- luka5: prove that the fifth axiom in the axiomatization of the many-valued sentential calculus of Lukasiewicz depends on the remaining four [22]. The problem is formulated as an equational problem and solved by UKB. Clauses with more than 16 symbols are discarded.

- mission: the Missionaries and Cannibals Puzzle, solved by hyperresolution.

- mv: prove three lemmas in the many-valued sentential calculus of Lukasiewicz (see e.g. [22]). The strategy is the same as in cn, except that no limit is set on the length of kept clauses.

- pigeon: this is an instance of the pigeon-hole problem (4 pigeons and 3 holes) [104], done by binary resolution.

- robbins and robbins2: these are two problems done by UKB completion. In robbins, one proves that if a Robbins algebra has an element c, such that $x + c = c$ for all x, then the algebra is Boolean, by proving that the Robbins axiom, associativity, commutativity and $x + c = c$ imply the Huntington's axiom. It is known that associativity, commutativity and the Huntington's axiom axiomatize Boolean algebra. PICK_GIVEN_RATIO is set to 2. In robbins2 [94] the problem is similar, with the hypothesis $x + c = c$ replaced by the hypothesis that there exists an element c, such that $c + c = c$. For the second problem, generated clauses with more than 20 symbols are discarded. Proving that every Robbins algebra, i.e. without additional hypothesis, is a Boolean algebra is still an open problem for theorem provers [124, 126].

- salt: this is a propositional version of the well-known Salt and Mustard puzzle from Lewis Carroll [19, 92]. It is done by binary resolution.

- sam-hyp: SAM's lemma in lattice theory [53] proved by hyperresolution.

- steam: Schubert's Steamroller problem [103, 104] solved by unit-resulting resolution.

- str-bws and w-sk: these are two example from a series of problems in combinatory logic, appeared in form of puzzles in [113] and as problems for theorem provers in several articles, e.g. [20, 97]. Some of these problems consist in determining whether certain fixed point properties hold for given sets of combinators: str-bws is one of these. It is done by unit-resulting resolution. Others consist in showing that a combinator can be expressed in terms of other combinators: w-sk belongs to this second class. The strategy here has paramodulation, demodulation and no backward subsumption.

- subgroup: prove that subgroups of index 2 are normal [125]. A subgroup A of a group G has index 2 if whenever $x \notin A$ and $y \notin A$ there exists a $z \in A$ such that $x * z = y$. A is normal if for all $x, y, z, w \in G$ such that $y \in A$, $y \cdot x^{-1} = z$ and $x \cdot z = w$, also w is in A.

- tba-1 and tba-gg: these are two problems in ternary boolean algebra, whose original axiomatization appears in [52]; tba-1 consists in proving the dependency of certain axioms from others; tba-gg proves a lemma. The first proof for the problem in tba-1 was given in [29] and automated proofs were given in [49, 123]. They are solved by paramodulation and demodulation.

- x2-quant: prove by paramodulation and demodulation that groups such that $xx = e$, where e is the identity, are commutative [65, 93]. The problem is given as a formula in first order logic with equality.

6.4.1 Discussion of experiments

The significance of these experiments is limited by having only up to 3 nodes. Also, some of the problems, e.g. those solved sequentially in a few seconds, are probably too easy for the parallelization to pay off. The results are generally unstable, as repeating the same experiment in apparently similar working conditions may yield different outcomes. While in some cases, e.g. $cd12$ and $cd13$, Aquarius-2 and Aquarius-3 speed-up over Aquarius-1, in several others Aquarius-2 and Aquarius-3 do not speed-up. Furthermore, it happens that Aquarius-2 improves over Aquarius-1, but Aquarius-3 performs much worse, e.g. $luka5$ and $imp3$. These mixed results may be caused by many factors. First, one should keep into account that the current version of Aquarius is a prototype developed in a 5 months period. It is evident that much more work is needed, in order to obtain an advanced implementation of the Clause-Diffusion methodology. Second, we may analyze the performances of Aquarius in terms of *communication*, *duplication* and *distribution of clauses*.

Observations of communication problems in Aquarius

Communication in Aquarius is too slow. An immediate evidence of this is that in many cases, the Penguin which succeeds first in finding a proof is the Penguin which read the input clauses, i.e. Penguin0. In some of these cases, it also happens that the run time of Penguin1 and Penguin2 is shorter than the run time of Penguin0. This indicates that the start of the derivations by Penguin1 and Penguin2 was delayed. These observations suggest that an improvement would be to modify the program in such a way that each Penguin reads the input clauses from its input file, rather than having one Penguin reading the input clauses and the other Penguins receiving them through messages. This would eliminate the need for communication in the input phase. Another evidence that communication is hindering the performances is the following. Let ψ be a clause which can be derived independently at two nodes, e.g. Penguin0 and Penguin1. In most runs, it happens that Penguin0 generates

and broadcasts ψ, but Penguin1 derives it on its own, *before* receiving the inference message from Penguin0. The intuitive idea of inference messages in the Clause-Diffusion methodology is that in general the clause carried by the message is "new" for the receiver. Therefore, when the above phenomenon happens in Aquarius the purpose of the inference messages is sort of defeated.

How communication is implemented in PCN

In order to understand the behaviour of Aquarius, we recall briefly how communication is implemented in PCN [120]. Let A and B be two processes executed at the same node and S be the stream for communication from A to B. In other words, A is the producer and B is the consumer. The consumer B is suspended, waiting to receive something on S. Thus, B is on the "suspension queue" associated to S. As soon as A sends a message m over S, B wakes up, i.e. it is moved to the "active queue" of the ready processes. When B is scheduled for execution, it will read m and then suspend again. We assume next that process A and process B are executed at two different nodes, node 1 and node 2 respectively. In this case, the suspension queue on S is held at node 1 and process B is suspended on a "remote reference" S' to S on node 2. The exchange of program-level messages, i.e. the messages defined in the specific PCN program, is implemented in terms of the runtime-level messages READ and VALUE, that are messages defined in the runtime of PCN. As B is being queued onto the S' suspension queue, a READ message will be generated from node 2 to node 1, asking for the value at S. If a program-level message m has been sent on S, then a VALUE message with m will be returned immediately from node 1 to node 2. Otherwise, a "value-note" will be enqueued on the S suspension queue. When A sends a message m on S, that value-note will cause a VALUE message to be sent from node 1 to node 2 with the content of m. Upon receipt of the VALUE message on node 2, B will wake-up and move to the active queue. For instance, in Aquarius, the producer A is the $main\_infer()$ procedure at Penguin0, the consumer B is the $receive()$ procedure at Penguin1 and the stream S is $C[0,1]$.

How the implementation of communication in PCN affects Aquarius

An issue which affects the performance of such a communication mechanism is the scheduling of the processes. The question is how soon a consumer process, which has been moved from a suspended queue to the active queue, will be selected from the active queue and enabled to read its message. The current implementation of PCN gives priority to the execution of C code over the execution of PCN code: when C code is running, no PCN code will run until that C code completes [120]. Also, no PCN message-passing will take place until the C code completes. It follows that a consumer, e.g. a PCN process *receive*() at Penguin1, may not be scheduled from the active queue to get its pending messages, because C code is being executed at the node. Therefore, communication, which is already likely to be the potential bottleneck in a distributed implementation, is at a strong disadvantage with respect to inference. Second, in Aquarius the consumers, i.e. *receive*(), are written in PCN. On the other hand, the producers are written in C, as the PCN procedure $main\_infer()$ immediately calls the C function $infer()$ and the inference message are generated by $infer()$. Thus, the producers generate messages at a much faster pace than the consumers may consume them. Indeed, we observed executions, where the inference part of the computation halts upon finding a proof and then several pending messages are delivered all together. Third, the runtime of PCN implements tail recursion optimization. Roughly speaking, this works as follows. Let A be a process executing a tail recursive procedure. When A invokes a sub-procedure, including another instance of A itself, A is not put back in the pool of processes, but it maintains the status of running process, so that it will be possible to resume it without the overhead of re-selecting it from the pool of processes. The procedure $main\_infer()$ is tail recursive. Therefore, it may happen that after $main\_infer()$ has produced a first bunch of messages, $main\_infer()$ is re-scheduled (with the possibility of generating a second group of messages) before *receive*() is allowed to see the messages in the first bunch.

These phenomena were apparent since the very first experimental runs of Aquarius. As we already mentioned, we tried to counter them by reducing the size of the C processes. Namely, we provided the possibility of invoking $small\_infer()$ rather than $infer()$ (see Section 6.1.2). Indeed, in almost all the listed problems Aquarius fares better if the flag $PRIORITY\_MSGS$ is on, which means $small\_infer()$ is used. However, this does not seem to have been sufficient, i.e. the selection and processing of a *given_clause* (including pre-processing and post-processing of all its children) may still represent a too large C task, which overwhelms the $receive()$ part. Thus, it may be necessary to break down further the C code. An alternative approach is to synchronize $receive()$ and $main\_infer()$ within each Penguin. Currently, $receive()$ and $main\_infer()$ are largely asynchronous. The only exception is when the *Sos* list is empty and $main\_infer()$ waits for $receive()$ to have received something. One possibility is to have $receive()$ telling $main\_infer()$ when it may proceed, e.g. when all the messages have been received. This approach, however, is less satisfactory from the point of view of the logic of the program, because $main\_infer()$ does not depend on $receive()$ unless the *Sos* is empty.

Another factor is that the communication done through PCN and Unix over the network may be hampered by too many layers of software, which mean for instance too much copying for each message. More precisely, in each Penguin process, a clause represented as a C structure, or record, at the C level, is copied into a string to become a message at the PCN level. This is necessary, because PCN does not feature structures and pointers. Then in the runtime of PCN, whenever a VALUE message is sent, it will be copied once at the sender from the heap into a memory buffer. On the receiving side, it may also be copied once from the message buffer to the heap [120]. More copying may be done at the Unix level. Thus each message is copied several times between the sender and the receiver. Aquarius would benefit if it were possible to bypass some layers and reduce the number of copies per message at each node.

Duplication

After having experienced the problem with communication, we resorted to try to reduce the amount of communication by empowering the single nodes. Because communication is so slow, it is better that all nodes are able to work as independently as possible. Some of the reported experiments have been done by turning on the flags OWN_IN_USABLE and/or OWN_IN_SOS, so that each node owns most of the input clauses. In other experiments, the default $ALTERNATE\_FIT$ has been turned off and replaced by $HALF\_ALT\_FIT$ or ALT_FIRST_FIT, so that each node retains most of its raw clauses as residents. None of the reported results, however, refer to executions under a combination of flags equivalent to the $STAND\_ALONE$ mode. In other words, in all the listed experiments, there is some partitioning of the search space.

The version of the Clause-Diffusion methodology implemented in Aquarius, already empowers the nodes significantly, by using the localized image sets. If, in addition, the above flags are turned on, the behaviour of each Penguin becomes very close to that of the sequential prover. While reducing communication, these settings of flags induce a strong increase in duplication. It appears from the trace files of the experiments, that often most of the clauses needed in the proof are generated independently at all nodes. This causes an incredible amount of duplication. For instance, in one run of the problem *cd*90, the clause $P(e(e(x,y),e(x,y)))$ appeared in the trace of the execution at Penguin2 as follows: first, it is generated and sent as new settler to Penguin0; second, it is generated again and kept as resident; third, it is received as inference message from Penguin0; fourth, it is generated one more time and sent as new settler to Penguin1; fifth, it is received as new settler. Finally, $P(e(e(x,y),e(x,y)))$ is subsumed by $P(e(x,x))$.

This amount of duplication may explain the lack of speed-up in many experiments. Duplication was increased to reduce communication. In fact, the problem with comunication is so strong, that Aquarius performs better in conditions

of very high duplication than in conditions where less duplication is present and more communication is necessary. The Clause-Diffusion methodology and Aquarius are sufficiently general and flexible to provide combinations of different degrees of communication and duplication. However, this highly duplication-oriented version of the Clause-Diffusion methodology was not intended to be the main one, since it reduces the significance of partitioning the search space. The basic idea in the Clause-Diffusion methodology is to partition the search space. Indeed, the cases where Aquarius-2 speeds-up significantly over Aquarius-1 are exactly those where partitioning the search space helps. More precisely, in most of the positive results, one Penguin, e.g. Penguin0 of Aquarius-2, finds a different and shorter proof than the proof found by Aquarius-1, because some clauses are not retained by Penguin0. An example is *cd*90, where Aquarius-2 has super-linear speed-up over Aquarius-1. The latter finds an 8-steps proof, which uses first $P(e(e(x,y),e(x,y)))$ and then $P(e(x,x))$. Aquarius-2 finds a 5-steps proof, which uses $P(e(e(x,y),e(x,y)))$, but does not even need the generation of $P(e(x,x))$.

Distribution of clauses

The third issue, i.e. the distribution of clauses, is more of a conceptual nature. None of the criteria for distributed allocation of clauses implemented in Aquarius keep into account the contents of a message, i.e. the clause, in order to decide its destination. The allocation policies also ignore the history of the derivation. In other words, all clauses are treated in the same way at all the stages of a derivation. The design of more informed allocation policies, e.g. policies which keep into account informations about the clause being allocated and the history of the derivation, may be an important progress. As an example, one may think of heuristics of the form: if more than n clauses with property Q have been allocated to node p_i, then the next clause with property Q will also be allocated to node p_i. Such criteria, however, will be more expensive to compute and it may not be simple to devise them. More generally, the question is how to find better ways to partition the search space of a

theorem proving problem by partitioning its data base.

Discussion

Other parallel theorem provers have obtained better experimental results than Aquarius. For instance, ROO shows linear speed-up on most non-equational problems, while its performances on equational problems suffer from the backward contraction bottleneck. Also, ROO is not very scalable, due to the hardware limitations of shared memory machines.

ROO uses parallelism at the clause level in shared memory. The data base of clauses and thus the search space is not partitioned. Under these conditions, ROO explores a search space which is very similar to the search space of its sequential counterpart, Otter. In general, a purely shared approach to parallel theorem proving, with parallelism at the term/clause level, does not modify the search space (and does not intend to). Thus, the parallel prover works on a search space which is basically the same as in the sequential case and it is likely that it finds a similar proof. The parallel prover speeds-up over the sequential one by generating faster the same proof and the results are rather regular.

Our philosophy is very different, because we aim at parallelism at the search level. Because the search space is partitioned, the concurrent processes deals with search spaces that may be radically different from that of the sequential prover. For instance, in Aquarius, it is sufficient that a Penguin does not retain a certain clause and sends it to settle at another node to change dramatically the search space for that Penguin. By considering a different portion of the search space, a shorter proof may be found. In such cases, the distributed theorem prover speeds-up considerably. On the other hand, if the search space turns out to be partitioned in a way that does not reveal a shorter proof, the distributed prover is at a strong disadvantage, as it may be basically generating the sequential proof from a fragmented search space. The irregular results are the consequence of this kind of phenomena.

In summary, at the operational level, the main cause for the mixed results of Aquarius is the inefficiency of communication. At least part of the problem seems to be related to the choice of the PCN language, which perhaps was not designed for the parallelization of a large, computation-bound C program, such as Otter. The problem with communication may represent evidence in favor of a less distributed approach, such as the mixed shared-distributed approach of the Clause-Diffusion methodology. At the conceptual level, the issue is that Aquarius probes a radically new approach to parallelization, whose success will require a better understanding of the parallelization of search.

Chapter 7

Summary and directions for future research

In this thesis we have presented a new abstract framework for completion-based theorem proving, the Clause-Diffusion methodology for distributed deduction and its implementation in the Aquarius theorem prover. In the following we summarize our results and we suggest directions to continue this work.

7.1 A new abstract framework for completion procedures

The most well-known application of completion procedures is the generation of *saturated* presentations, e.g. confluent sets of equations, which may act as *decision procedures* for the validity of the theorems in the theory. Theorem proving is then regarded as a two-phase process: first compile the given presentation into a finite saturated one and next prove theorems in the saturated presentation. This approach does not apply whenever the saturated presentation is infinite, as it happens in most

cases. In all such cases completion procedures have been used as *semidecision procedures*. The procedure takes in input both the given non-saturated presentation and the target theorem: if the latter is indeed a theorem, it will be proved as a side-effect of the construction of the infinite saturated presentation.

Given this situation, the problem is to improve the efficiency of completion procedures as theorem provers, in order to obtain faster proofs for more theorems. Several authors have worked on the inference systems of completion procedures, with the purpose of making the saturation process more efficient. As a side-effect, more theorems may be proved in reasonable time.

Our approach is intrinsically different. We observed that the inference system is not the only component of a completion procedure. A completion procedure is given by an inference system and a search plan. While this is well-known in principle, the role of the search plan is generally overlooked. Most theorem proving strategies are presented by giving the set of inference rules only and leaving the task of designing a suitable search plan to the implementation phase. This is not satisfactory, since the search plan is what ultimately turns a set of inference rules into a procedure. The actual performance of a prover depends heavily on the search plan. Therefore, we decided to start our study of completion-based theorem proving at the search plan level.

The key property of a search plan is *fairness*. Intuitively, fairness of a search plan means that every inference step which needs to be considered will eventually be considered. In completion-based methods, this usually has meant resolving all potential critical pairs. We termed this notion of fairness *uniform fairness*. Indeed, uniform fairness is sufficient to generate saturated presentations and therefore it is also sufficient for theorem proving. Since theorem proving by completion has been traditionally regarded as a side-effect of saturation, uniform fairness is the standard requirement on the search plan of a completion procedure.

In theorem proving, on the other hand, one is not interested in critical pairs

which may not contribute to a proof of the target theorem. Thus, in theorem proving applications fairness should not require considering all possibile critical pairs, but only those which may lead to a proof. With this intuition in mind, we conjectured that uniform fairness may not be necessary for theorem proving and may in fact represent a too strong requirement. Therefore, we worked to weaken the fairness requirement on search plans for theorem proving. A weaker fairness requirement means that less work, i.e. fewer inferences, is necessary, and more efficient procedures may be possible.

In order to pursue this investigation, we analyzed the nature of theorem proving derivations. We discovered that a successful theorem proving derivation may be conceived as a process of *reduction of a minimal proof of the target theorem in the given presentation of the theory*. Accordingly, all the fundamental concepts in theorem proving, at both the inference and search levels, such as *contraction*, *redundancy*, *refutational completeness* and *fairness*, have been uniformly defined in terms of target-oriented proof reduction with respect to a well-founded ordering on proofs. We called "fairness" tout court, as opposed to "uniform fairness", our target-oriented notion.

We proved that if the inference rules are *refutationally complete* and the search plan is *fair* according to these definitions, a completion procedure is a semidecision procedure for theorem proving. This result makes the interpretation of completion procedures as semidecision procedures independent from the interpretation as generators of confluent systems. Confluence of the limit is not necessary for theorem proving and therefore a completion procedure can be a semidecision procedure without being a generator of confluent systems.

Our definition of fairness is the first attempt to give a definition of fairness for completion procedures, which is weaker than uniform fairness and still sufficient for theorem proving. By focusing on the given target, it makes possible to design fair search plans which ignore the majority of possible critical pairs. Coherently,

our notions of contraction and redundancy are also target-oriented and therefore may induce more powerful contraction rules than those admissible in a generator of saturated sets.

In addition to opening new perspectives, our approach is compatible with the pre-existing results. The classical theorems on uniformly fair completion in equational logic and their extensions to Horn logic with equality fit in our framework, if uniform fairness, rather than fairness, is assumed. We presented some equational completion procedures based on Unfailing Knuth-Bendix completion according to our notions. They include the AC-UKB procedure with Cancellation laws, the S-strategy and the Inequality Ordered Saturation strategy. These extensions of UKB had not been presented in a unified framework for completion before. We also showed that the process of disproving inductive theorems by the so called *inductionless induction* method is a semidecision process.

7.2 The Clause-Diffusion methodology for distributed deduction

On a more practical side, we studied the problem of how to execute theorem proving strategies in a distributed environment. The basic motivation for studying distributed theorem proving is to improve the efficiency of theorem proving strategies by exploiting parallelism. The result of this study is a methodology for distributed deduction by *Clause-Diffusion*, whose basic principles and many variations we described in this thesis.

7.2.1 An analysis of the parallelization of theorem proving strategies

Although our approach to parallelization applies to theorem proving in general, we focused on *contraction-based strategies*. This choice has two reasons. First, these strategies proved to be more powerful than strategies without contraction on significant classes of problems. This experimental evidence is accompanied by several theoretical studies, including our framework for completion-based theorem proving, which all point to the important role of contraction rules in trimming the generated search space. Second, the presence of contraction rules makes the parallelization more difficult. As it has been often observed, those features which make a sequential algorithm or procedure more efficient than others, are also those which may make its parallelization harder. Thus, the study of strategies without contraction can be regarded as a special case of the study of strategies with contraction.

This conclusion is one of the results of our analysis of the parallelizability of theorem proving strategies. For the purposes of this analysis, we classified deduction methods in three categories: *subgoal-reduction strategies*, *expansion-oriented strategies* and *contraction-based strategies*. Examples of subgoal-reduction methods are Prolog technology theorem proving strategies, logic and equational programming. The distinction between expansion-oriented and contraction-based strategies is drawn based on the application of contraction: the former use expansion and forward contraction only, while the latter allow also backward contraction. *Forward contraction* is the process of reducing a newly generated clause with respect to the clauses existing at the time of its generation. *Backward contraction* is the process of maintaining a clause reduced with respect to the clauses generated afterwards. Next, we defined three types of parallelism in logical inferences: *parallelism at the term level* (fine grain), *parallelism at the clause level* (medium grain) and *parallelism at the search level* (coarse grain). Most of the existing works in parallel deduction exploit parallelism at the term or clause level. At these levels, two concurrent inference steps may be in *conflict* to access common premises. There may be *write-write*

conflicts or *read-write conflicts* between contraction steps and *read-write conflicts* between an expansion step and a contraction step.

In derivations by subgoal-reduction strategies, the data base containing the presentation, e.g. a logic program, is essentially *static*, because all steps consist in reducing a goal to subgoals. Conflicts do not arise or may be prevented by preprocessing the clauses in the data base at "compile-time", i.e. before the derivation. Thanks to these conditions, subgoal-reduction strategies are amenable to all three types of parallelism. In derivations by expansion-oriented strategies, the data base is *monotonically increasing*, because of expansion. Thus, it is not possible to prevent conflicts by pre-processing all clauses at "compile-time" and a limited amount of conflicts, i.e. write-write conflicts in forward contraction, appears. Parallelism at the term level, e.g. parallel rewriting in forward contraction, is no longer appealing. In derivations by contraction-based strategies, the data base is *highly dynamic* and *not monotonic*, because of backward contraction. Also, backward contraction applies to clauses which are already being used as parents of expansion steps, causing read-write conflicts between the backward contraction steps and the expansion steps. Under these conditions, also parallelism at the clause level is likely to cause too much overhead and only parallelism at the search level remains suitable.

In summary, as we move from subgoal-reduction strategies through expansion-oriented strategies to contraction-based strategies, the data base becomes more and more dynamic, the degree of inter-dependence of inference steps increases and correspondingly the appropriate granularity of parallelism grows. In our survey, we considered some existing parallel theorem provers, which execute contraction-based strategies with parallelism at the clause level. According to our analysis, these approaches seem to suffer from a negative phenomenon, which epitomizes the drawbacks of a too fine-grained parallelism in the presence of backward contraction. We termed this problem the **backward contraction bottleneck**. In shared memory, it appears as a write bottleneck, due to the avalanche growth of write-access requests from backward-contraction processes.

7.2.2 A notion of parallelism at the search level

In parallelism at the search level, concurrent, asynchronous processes search in parallel for a solution. As soon as one of them succeeds, the whole process succeeds. The search space is partitioned among the processes by distributing the clauses. Unlike in fine and medium grain parallelism, each process is given a large portion of the data, e.g. a fairly large set of clauses, and can develop its own derivation while communicating with the other processes.

The data bases of the concurrent processes are physically separated. Therefore, no two concurrent inference steps are physically in conflict: two concurrent inference steps which use the same clause logically, access two distinct physical copies of the clause. The cost of preventing physical conflicts by using separated data bases is the generation of redundant clauses, which would not be generated by a sequential derivation. Our estimate is that it is better to let the processes proceed eagerly in parallel, generating additional redundant clauses and deleting them afterwards, rather than synchronize the processes, thus forcing them to wait, in order to avoid conflicts.

While parallelism at the term and clause level are generally implemented in *shared memory*, parallelism at the search level leads toward *distributed memory*. We weighed the advantages and disadvantages of shared and distributed memory. We preferred not to adopt a purely shared memory solution for several reasons, most of which are related to backward contraction and are indeed the same reasons in favor of coarse grain parallelism. Thus, we settled for an environment with mostly distributed memory and possibly a shared memory component, such as a loosely coupled, asynchronous multiprocessor or a network of computers.

7.2.3 The Clause-Diffusion methodology

Parallelism at the search level requires to partition the search space among the deductive processes. The search space is determined by the input problem, i.e. the clauses, and the inference rules. The Clause-Diffusion methodology partitions the clauses among the processes, so that each one of them owns a portion (its *residents*) of the data base of clauses. The expansion inferences are subdivided based on the ownership of the clauses. Contraction inferences are not subdivided, so that each process can perform as much contraction as possible. Exactly because each process is assigned just a portion of the search space, the processes need to communicate, in order to find a proof whenever there exists one. Communication is realized via *message-passing*: the processes send their residents to the other processes in form of messages, termed *inference messages*. In a distributed environment each deduction process runs at a different node and they exchange messages over the links interconnecting the nodes.

Specific Clause-Diffusion strategies are obtained by selecting several components, beside the inference rules and the search plan, such as the distributed global contraction scheme, the distributed allocation algorithm and the algorithms for message-passing. Also the search plan to be executed by each deduction process needs to schedule both the inference and the communication steps. For most of these components we analyzed the underlying problems and proposed several solutions:

- Schemes for distributed global contraction:
  - Global contraction by travelling,
  - Global contraction by travelling with selected simplifiers,
  - Global contraction at the source by localized image sets or
  - Global contraction at the source by global image set in shared memory.
- Techniques to generate a data base of selected simplifiers:

  - Centralized selection with or without simplification or
  - Distributed selection with simplification.

- Policies to maintain the localized image sets with respect to contraction:
  - Direct contraction,
  - No contraction or
  - Update by inference messages.

- Policies to maintain the global image set in shared memory with respect to contraction:
  - No contraction,
  - Direct update or
  - Delayed update with garbage collection.

- Criteria for distributed allocation: best-fit, alternate-fit, half-alternate-fit, first-fit, alternate-first-fit or next-fit.

- Algorithms for routing and broadcasting: we considered the (virtual) ring topology, the cube-connected topology, a fully connected virtual topology, such as in Aquarius, and flooding on a random topology.

Perhaps the most influential choice, from the point of view of parallelization, is the choice of the scheme for distributed global contraction.

7.2.4 Distributed global contraction

Distributed global contraction and, more concretely, the schemes which realize it, is our solution to the backward contraction bottleneck problem. Depending on how global contraction is done, the Clause-Diffusion methodology comprises instances of the following approaches:

- Global contraction by travelling (which also involves the use of wake-up calls) represents a **distributed, communication-oriented approach**, which may be feasible only if very fast communication is available. This approach is communication-oriented, because it relies primarily on the exchange of clauses between the processes to allow parallel inferences.

- Global contraction at the source with localized image sets represents a **distributed, duplication-oriented approach**, which requires sufficiently large memories at the nodes. It is duplication-oriented, because parallel inferences are made possible by forming the localized image sets, i.e. by duplicating the clauses.

- Global contraction at the source with shared memory represents a **mixed shared-distributed approach**, which relies on a fast shared memory and also sufficiently large memories at the nodes. This is a hybrid approach. It reduces the amount of communication, because exchange of messages may be replaced in part by access to the shared memory. It still involves duplication, because all the clauses in the shared memory are duplicates of the residents at the nodes. But duplication is reduced with respect to the previous approach, because just one shared global image set, rather than many localized image sets, is maintained.

In all these approaches, the clauses to be rewritten by contraction are held in the local memories of the nodes. This is the key feature which prevents the backward contraction bottleneck. The main costs of our solutions are represented by memory and/or communication requirements as indicated above. In particular in the mixed approach, the backward contraction bottleneck does not appear, even if a shared memory is used, because the clauses in shared memory are used as simplifiers only. They are not subject themselves to contraction, while the residents at the node are. The negative sides are the duplication of clauses and the delay in updating the shared memory with respect to the nodes.

This mixed approach seems to be especially appealing. The general wisdom is that distributed memory may be more convenient for intrinsically independent, asynchronous activities, while shared memory may be more appropriate for concurrent activities which cooperate closely and synchronously. We feel that parallel deduction processes with contraction involve activities of both types. Expansion and local contraction may be considered as belonging to the first category, while global contraction may fall in the second one. Indeed, the repeated contraction of a clause with respect to a given set of simplifiers may be conceived as a single inference step, rather than a combination of steps. Under such view, it becomes apparent that it may be convenient to have all the simplifiers in one place. Our scheme for global contraction at the source with shared memory seems promising because it allows to share the simplifiers, without incurring in the backward contraction bottleneck of other purely shared memory approaches.

7.2.5 A study of fairness in distributed derivations

We defined the distributed derivations generated by Clause-Diffusion strategies and studied problems related to the *(uniform) fairness* and *monotonicity* of distributed derivations. Intuitively, the (uniform) fairness of a distributed derivation requires the local (uniform) fairness of the search plan Σ executed at the nodes and the fairness of the mechanisms to generate and pass the messages. We gave an abstract definition of distributed uniform fairness and reduced it to a set of sufficient conditions, which formalize the fairness requirements on the communication part. The Clause-Diffusion strategies satisfy the fairness requirements by using either the localized image sets (distributed duplication-oriented approach) or the global image set in shared memory (mixed approach) or the wake-up calls (distributed communication-oriented approach).

Our study of fairness in distributed deduction may be regarded as a special case of the more general problem of fairness in distributed data bases. One of the

important problems in distributed data bases is to maintain the global *consistency* of data in the presence of updates. Depending on the applications, consistency may also be called *agreement* or *coherence*. The same problem also appears in distributed deduction, although in a weaker form. Contraction inferences are in fact a form of update, since they replace data by others. Our counterpart to the notion of consistency, on the other hand, is not as rigid. The main difference is that once a clause is modified through contraction, it is not necessary to require that all of its copies be updated immediately into identical form. This is because a contracted clause is still *logically equivalent* to the original one, which is all that is required in automated deduction. What is lost by not keeping copies of the same datum identical is not consistency, but minimality: there is a temporary increase of redundancy which, although undesirable, does not disturb the global integrity of the system. This is why it is relatively easier to come up with a reasonable solution for our problem than for the general problem for distributed data bases. Some of our work may be useful to distributed data bases applications with less stringent requirements of consistency.

7.2.6 Distributed subsumption

We observed that the unrestricted application of subsumption, may thwart the monotonicity and the fairness of a distributed derivation. The restriction to proper subsumption, commonly used in the sequential case to prevent violations of fairness by subsumption of variants, does not solve the problem, since even proper subsumption may harm the monotonicity of a distributed derivation. The observation of the latter, surprising phenomenon, led us to re-think the meaning of the subsumption inference rule. We reasoned that subsumption is replacement of a clause by a more general one, rather than mere deletion of the less general clause. We distinguished between *replacement-subsumption* and *variant-subsumption*, and derived proper subsumption as their composition. We extended this three-rules framework to the distributed case and defined an inference rule of *distributed subsumption*

which embeds distributed variant-subsumption and distributed proper subsumption as subcases. We proved that distributed subsumption has all the desirable properties. It prevents violations to monotonicity, by applying replacement-subsumption (as embedded in distributed proper subsumption) to subsume a resident clause by an inference message. It prevents the problems with subsumption between variants, by establishing a global well-founded ordering between variants, even in the absence of a global clock. Thus, subsumption can be realized in distributed theorem proving safely and with negligible overhead.

7.2.7 The Aquarius theorem prover

Our prototype Aquarius implements the Clause-Diffusion methodology on a network of Sun sparcstations. The theorem proving program executed at each node embeds the code of the Otter theorem prover [98, 99], modified and enriched with communication capabilities. In other words, Aquarius implements the Clause-Diffusion strategies based on the sequential strategies provided by Otter. Aquarius adopts global contraction at the source with localized image sets. The localized image sets are formed by saving the received inference messages and maintained by using both direct contraction and update by inference messages. Since all simplifiers are available at each node, there is no mechanism to choose selected simplifiers. For the same reason, Aquarius does not use wake-up calls. The distributed allocation algorithm features the alternate-fit, half-alternate-fit, alternate-first-fit and first-fit policies. Most of the code is written in C. The communication part is written in PCN [27, 47], a parallel programming language compatible with C, which provides primitives for communication through streams. Therefore, routing and broadcasting are done on a fully connected virtual topology, where any two nodes are directly connected by a stream. Aquarius inherits from Otter its efficiency of basic operations and data structures. The portability of Otter is also maintained, since PCN is almost as portable as C. Aquarius has all the options of Otter and it adds new ones related to the distributed nature of the execution. By setting and combining

parameters and flags, the user may experiment with different strategies within the same theorem prover.

7.2.8 The experiments with Aquarius

We have described the experiments with Aquarius with up to three nodes on a set of 33 examples. The results are very mixed, ranging from encouraging to outright disappointing. We have discussed several factors, related to the implementation and to the software environment, which may contribute to cause this outcome. The main source of difficulty, at least in the present implementation, is communication. Therefore, Aquarius has been geared toward a mostly duplication-oriented approach. This is not entirely satisfactory, though, because the high degree of duplication means that the concurrent processes overlap to a large extent, thereby wasting the advantage of having concurrent processes in the first place. As a consequence, the performances of Aquarius suffer from a combination of slow communication and excessive duplication.

More generally, the experience with Aquarius shows that partitioning the search space may change radically, with respect to the sequential case, the search space that each deductive process is dealing with. Intuitively, as we go from a purely shared approach to a mixed shared-distributed approach, through a distributed duplication-oriented approach and finally a distributed communication-oriented approach, the degree of overlap among the search spaces for the concurrent processes decreases. Therefore, we obtain search spaces that are more and more different from the search space in the sequential case. It is probably easier to obtain better behaved experimental results by parallelizing a strategy without changing the underlying search space too radically. As the search space does not change dramatically, it is more likely that the sequential prover and the parallel prover find the same proof or very similar proofs. The parallel prover speeds-up by generating faster the same proof. On the other hand, an approach which partitions the search space effectively and

has search processes with little overlap is likely to yield irregular results. Because the search space is different, the distributed prover may find a shorter proof than the sequential one. If this happens, the result is very good. Otherwise, the result may be very poor, as the distributed prover generates a proof similar to the sequential one by searching different fragments of search space, rather than the whole search space seen by the sequential prover.

7.3 Directions for future research

Many directions for ongoing and future work can be envisioned. On the practical side, we plan to continue the development and fine-tuning of our prototype Aquarius. The experimentation with the prover is leading already to variations and enhancements. For instance, in Aquarius the nodes may execute different search plans and select different subsets of the available inference rules. Indeed, the Clause-Diffusion method do not require to have the same strategy at all sites. We expect that most of the room for improvement is at the level of the search plans featured by Aquarius, i.e. the criteria to allocate and sort clauses and inference/communication steps. The experimentation done so far has shown that the allocation algorithm plays an especially important role in determining the performances of Aquarius. Therefore, we shall consider the problem of designing better criteria to decide where to allocate a given clause during the derivation. Such criteria may keep into account statistics about the derivations developed so far at the nodes. Examples of statistics are the number of clauses stored at a node, the number of clauses of a certain type, e.g. simplifiers, the time spent so far in performing a certain type of inference and so on. Another possibility might be to design criteria based on the syntax of the clauses, e.g. distributing the clauses according to some partition of the signature.

Aquarius implements a distributed duplication oriented approach, since global contraction is done by localized image sets. Another direction is to develop in detail and implement the mixed shared-distributed approach, with a global image set in

shared memory. This will require an environment where each node, while still having a fairly large local memory, may access a shared memory component.

On the theoretical side, we may work on the design of *parallel search plans*. In the present work, we assumed a sequential search plan for the local inferences at a single node. Such a search plan is simply extended to incorporate communication according to the priorities dictated by the Clause-Diffusion methodology. The next phase is to study how to develop intrinsically parallel search plans, i.e. search plans which keep into account that the strategy will be executed in a distributed environment.

Bibliography

[1] S.G.Akl, *Parallel Sorting Algorithms*, Notes and Reports in Computer Science and Applied Mathematics, Academic Press, 1985.

[2] S.Anantharaman and J.Mzali, Unfailing Completion modulo a set of equations, Technical Report, LRI, Université de Paris Sud, 1989.

[3] S.Anantharaman, J.Hsiang and J.Mzali, SbReve2: A Term Rewriting Laboratory with (AC)-Unfailing Completion, in N.Dershowitz (ed.), *Proceedings of the Third International Conference on Rewriting Techniques and Applications*, Chapel Hill, NC, USA, April 1989, Springer Verlag, Lecture Notes in Computer Science 355, 533–537, 1989.

[4] S.Anantharaman and J.Hsiang, Automated Proofs of the Moufang Identities in Alternative Rings, *Journal of Automated Reasoning*, Vol. 6, No. 1, 76–109, 1990.

[5] S.Anantharaman and N.Andrianarivelo, Heuristical Criteria in Refutational Theorem Proving, in A.Miola (ed.), *Proceedings of the Symposium on the Design and Implementation of Systems for Symbolic Computation*, Capri, Italy, April 1990, Springer Verlag, Lecture Notes in Computer Science 429, 184–193, 1990.

[6] S.Anantharaman and M.P.Bonacina, An Application of the Theorem Prover SBR3 to Many-valued Logic, in M.Okada and S.Kaplan (eds.), *Proceedings of*

the Second International Workshop on Conditional and Typed Term Rewriting Systems, Montréal, Canada, June 1990, Springer Verlag, Lecture Notes in Computer Science 516, 156–161, 1991.

[7] L.Bachmair, N.Dershowitz and J.Hsiang, Orderings for Equational Proofs, in *Proceedings of the First Annual IEEE Symposium on Logic in Computer Science*, 346–357, Cambridge, Massachussets, June 1986.

[8] L.Bachmair and N.Dershowitz, Inference Rules for Rewrite-Based First-Order Theorem Proving, in *Proceedings of the Second Annual IEEE Symposium on Logic in Computer Science*, Ithaca, New York, June 1987.

[9] L.Bachmair, Proofs Methods for Equational Theories, Ph.D. thesis, Department of Computer Science, University of Illinois, Urbana, Illinois, 1987.

[10] L.Bachmair, Proof by consistency in equational theories, in *Proceedings of the Third Annual IEEE Symposium on Logic in Computer Science*, 228–233, Edinburgh, Scotland, July 1988.

[11] L.Bachmair and N.Dershowitz, Critical Pair Criteria for Completion, *Journal of Symbolic Computation*, Vol. 6, No. 1, 1–18, August 1988.

[12] L.Bachmair and N.Dershowitz, Completion for rewriting modulo a congruence, *Theoretical Computer Science*, Vol. 67, No. 2 & 3, 173–202, October 1989.

[13] L.Bachmair, N.Dershowitz and D.A.Plaisted, Completion without failure, in H.Aït-Kaci, M.Nivat (eds.), *Resolution of Equations in Algebraic Structures*, Vol. II: Rewriting Techniques, 1–30, Academic Press, New York, 1989.

[14] L.Bachmair and N.Dershowitz, Equational inference, canonical proofs and proof orderings, *Journal of the ACM*, to appear.

[15] L.Bachmair and H.Ganzinger, On Restrictions of Ordered Paramodulation with Simplification, in M.E.Stickel (ed.), *Proceedings of the Tenth International Conference on Automated Deduction*, Kaiserslautern, Germany, July

1990, Springer Verlag, Lecture Notes in Artificial Intelligence 449, 427–441, 1990.

[16] L.Bachmair and H.Ganzinger, Completion of First-Order Clauses with Equality by Strict Superposition, in M.Okada and S.Kaplan (eds.), *Proceedings of the Second International Workshop on Conditional and Typed Term Rewriting Systems*, Montréal, Canada, June 1990, Springer Verlag, Lecture Notes in Computer Science 516, 162–180, 1991.

[17] L.Bachmair and H.Ganzinger, Non-Clausal Resolution and Superposition with Selection and Redundancy Criteria, in *Proceedings of Logic Programming and Automated Reasoning*, Springer Verlag, Lecture Notes in Artificial Intelligence 624, 273–284, 1992.

[18] W.Bledsoe, Challenge problems in elementary calculus, in *Journal of Automated Reasoning*, Vol. 6, No. 3, 341-359, 1990.

[19] W.W.Bartley III (ed.), *Lewis Carroll's Symbolic Logic*, Clarkson N.Potter Inc.

[20] M.P.Bonacina and G.Sanna, KBlab: An Equational Theorem Prover for the Macintosh, in N.Dershowitz (ed.), *Proceedings of the Third International Conference on Rewriting Techniques and Applications*, Chapel Hill, NC, USA, April 1989, Springer Verlag, Lecture Notes in Computer Science 355, 548–550, 1989.

[21] M.P.Bonacina and J.Hsiang, On Rewrite Programs: Semantics and Relationship with Prolog, *Journal of Logic Programming*, Vol. 14, No. 1&2, 155–180, October 1992.

[22] M.P.Bonacina, Problems in Lukasiewicz logic, *Newsletter of the Association for Automated Reasoning*, No. 18, 5-12, June 1991.

[23] S.Bose, E.M.Clarke, D.E.Long and S.Michaylov, Parthenon: A parallel theorem prover for non-Horn clauses, *Journal of Automated Reasoning*, Vol. 8, N. 2, 153–182, April 1992.

[24] B.Buchberger, An Algorithm for Finding a Basis for the Residue Class Ring of a Zero-dimensional Polynomial Ideal, (in German), PhD thesis, Department of Mathematics, University of Innsbruck, Austria, 1965.

[25] D.Champeaux, Sub-problem finder and instance checker: Two cooperating pre-processors for theorem provers, in *Proceedings of the Sixth International Joint Conference on Artificial Intelligence*, 191–196, 1979.

[26] D.Champeaux, Subproblem Finder and Instance Checker, *Journal of the ACM*, Vol. 33, No. 4, 633–657, 1986.

[27] K.M.Chandy and S.Taylor, An Introduction to Parallel Programming, Jones and Bartlett, 1991.

[28] C.L.Chang and R.C.Lee, *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, New York, 1973.

[29] Chinthayamma, Sets of independent axioms for a ternary boolean algebra, Notices of the American Mathematical Society, No. 164, 654, June 1969.

[30] J.D.Christian, High-Performance Permutative Completion, Ph.D. thesis, The University of Texas at Austin, available as MCC Technical Report ACT-AI-303-89, August 1989.

[31] E.M.Clarke, D.E.Long, S.Michaylov, S.A.Schwab, J.-P.Vidal and S.Kimura, Parallel Symbolic Computation Algorithms, Technical Report CMU-CS-90-182, School of Computer Science, Carnegie Mellon University, October 1990.

[32] S.E.Conry, D.J.MacIntosh and R.A.Meyer, DARES: A Distributed Automated REasoning System, in *Proceedings of the 11th Conference of the American Association for Artificial Intelligence*, 78–85, 1990.

[33] J.Denzinger, Distributed knowledge-based deduction using the team work method, Technical report, Department of Computer Science, University of Kaiserslautern, 1991.

[34] N.Dershowitz and Z.Manna, Proving termination with multisets orderings, *Communications of the ACM*, Vol. 22, N. 8, 465–476, August 1979.

[35] N.Dershowitz, Orderings for term-rewriting systems, *Theoretical Computer Science*, Vol. 17, No. 3, 279–301, 1982.

[36] N.Dershowitz and N.A.Josephson, Logic Programming by Completion, in *Proceedings of the Second International Conference on Logic Programming*, 313–320, Uppsala, Sweden, 1984.

[37] N.Dershowitz, Computing with Rewrite Systems, *Information and Control*, Vol. 65, 122–157, 1985.

[38] N.Dershowitz, Termination of Rewriting, *Journal of Symbolic Computation*, Vol. 3, No. 1 & 2, 69–116, February/April 1987.

[39] N.Dershowitz, Completion and its Applications, in H.Aït-Kaci, M.Nivat (eds.), *Resolution of Equations in Algebraic Structures*, Vol. II: Rewriting Techniques, 31–86, Academic Press, New York, 1989.

[40] N.Dershowitz and D.A.Plaisted, Equational Programming, in J.E.Hayes, D.Michie and J.Richards (eds.), *Machine Intelligence 11: The logic and acquisition of knowledge*, Chapter 2, 21-56, Oxford Press, 1988.

[41] N.Dershowitz and J.-P.Jouannaud, Rewrite Systems, Chapter 15, Volume B, *Handbook of Theoretical Computer Science*, North-Holland, 1989.

[42] N.Dershowitz and J.-P.Jouannaud, Notations for Rewriting, Rapport de Recherche 478, LRI, Université de Paris Sud, January 1990.

[43] N.Dershowitz and N.Lindenstrauss, An Abstract Concurrent Machine for Rewriting, in H.Kirchner, W.Wechler (eds.) *Proceedings of the Second Conference on Algebraic and Logic Programming*, Nancy, France, October 1990, Springer Verlag, Lecture Notes in Computer Science 463, 318–331 1990.

[44] N.Dershowitz, A Maximal-Literal Unit Strategy for Horn Clauses, in M.Okada and S.Kaplan (eds.), *Proceedings of the Second International Workshop on Conditional and Typed Rewriting Systems*, Montréal, Canada, June 1990, Springer Verlag, Lecture Notes in Computer Science 516, 14–25, 1991.

[45] N.Dershowitz, Canonical Sets of Horn Clauses, in J.Leach Albert, B.Monien, M.Rodríguez Artalejo (eds.), *Proceedings of the Eighteenth International Conference on Automata, Languages and Programming*, Madrid, Spain, July 1991, Springer Verlag, Lecture Notes in Computer Science 510, 267–278, 1991.

[46] F.Fages, Associative-commutative unification, in R.Shostak (ed.), *Proceedings of the Seventh International Conference on Automated Deduction*, Napa Valley, CA, USA, 1984, Springer Verlag, Lecture Notes in Computer Science 170, 1984.

[47] I.Foster and S.Tuecke, Parallel Programming with PCN, Technical Report ANL-91/32, Version 1.2, December 1991.

[48] L.Fribourg, A Strong Restriction to the Inductive Completion Procedure, *Journal of Symbolic Computation*, Vol. 8, No. 3, 253–276, September 1989.

[49] L.Fribourg, A superposition oriented theorem prover, *Journal of Theoretical Computer Science*, Vol. 35, 129–166, 1985.

[50] J.A.Goguen, How to prove algebraic inductive hypotheses without induction, in W.Bibel and R.Kowalski (eds.), *Proceedings of the Fifth International Conference on Automated Deduction*, Les Arcs, France, 1980, Springer Verlag, Lecture Notes in Computer Science 87, 356–373, 1980.

[51] J.A.Goguen, S.Leinwand, J.Meseguer and T.Winkler, The Rewrite Rule Machine, 1988, Technical Monograph PRG-76, Oxford University Computing Laboratory, August 1989.

[52] A.A.Grau, Ternary Boolean Algebra, Bulletin of the American Mathematical Society, Vol. 53, No. 6, 567–572, June 1947.

[53] J.R.Guard, F.C.Oglesby, J.H.Bennett and L.G.Settle, Semi-Automated Mathematics, *Journal of the ACM*, Vol. 16, No. 1, 1969.

[54] A.Guha and H.Zhang, Andrews' Challenge Problem: Clause Conversion and Solutions, *Newsletter of the Association for Automated Reasoning*, No. 14, 5-8, December 1989.

[55] D.J.Hawley, A Buchberger Algorithm for Distributed Memory Multi-Processors, in *Proceedings of the International Conference of the Austrian Center for Parallel Computation*, Linz, Austria, October 1991, Springer Verlag, Lecture Notes in Computer Science, to appear.

[56] L.Henschen et al., Challenge Problem 1, *SIGART Newsletter*, No. 72, 30–31, July 1980.

[57] C.M.Hoffmann and M.J.O'Donnell, Programming with Equations, *ACM Transactions on Programming Languages and Systems*, Vol. 4, N. 1, 83–112, January 1982.

[58] J.Hsiang and N.Dershowitz, Rewrite Methods for Clausal and Nonclausal Theorem Proving, in *Proceedings of the Tenth International Conference on Automata, Languages and Programming*, Barcelona, Spain, July 1983, Springer Verlag, Lecture Notes in Computer Science 154, 1983.

[59] J.Hsiang, Refutational Theorem Proving Using Term Rewriting Systems, *Artificial Intelligence*, Vol. 25, 255–300, 1985.

[60] J.Hsiang and M.Rusinowitch, A New Method for Establishing Refutational Completeness in Theorem Proving, in J.Siekmann (ed.), *Proceedings of the Eighth Conference on Automated Deduction*, Oxford, England, July 1986, Springer Verlag, Lecture Notes in Computer Science 230, 141–152, 1986.

[61] J.Hsiang, Rewrite Method for Theorem Proving in First Order Theories with Equality, *Journal of Symbolic Computation*, Vol. 3, 133–151, 1987.

[62] J.Hsiang and M.Rusinowitch, On word problems in equational theories, in Th.Ottman (ed.), *Proceedings of the Fourteenth International Conference on Automata, Languages and Programming*, Karlsruhe, Germany, July 1987, Springer Verlag, Lecture Notes in Computer Science 267, 54–71, 1987.

[63] J.Hsiang, M.Rusinowitch and K. Sakai, Complete Inference Rules for the Cancellation Laws, in *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, Milano, Italy, August 1987, 990–992, 1987.

[64] J.Hsiang and M.Rusinowitch, Proving Refutational Completeness of Theorem Proving Strategies: the Transfinite Semantic Tree Method, *Journal of the ACM*, Vol. 38, No. 3, 559–587, July 1991.

[65] G.Huet, Experiments with an interactive prover for Logic with Equality, Report 1106, Jenning Computing Center, Case Western Reserve University , Cleveland, Ohio, 1971.

[66] G.Huet, Confluent reductions: abstract properties and applications to term rewriting systems, *Journal of the ACM*, Vol. 27, 797–821, 1980.

[67] G.Huet, A Complete Proof of Correctness of the Knuth-Bendix Completion Algorithm, *Journal of Computer and System Sciences*, Vol. 23, 11–21, 1981.

[68] G.Huet and J.M.Hullot, Proofs by Induction in Equational Theories with Constructors, *Journal of Computer and System Sciences*, Vol. 25, 239–266, 1982.

[69] A.Jindal, R.Overbeek and W.Kabat, Exploitation of parallel processing for implementing high-performance deduction systems, *Journal of Automated Reasoning*, Vol. 8, 23–38, 1992.

[70] J.-P.Jouannaud and H.Kirchner, Completion of a set of rules modulo a set of equations, *SIAM Journal of Computing*, Vol. 15, 1155–1194, November 1986.

[71] J.-P.Jouannaud and E.Kounalis, Automatic proofs by induction in equational theories without constructors, *Information and Computation*, Vol. 82, No. 1, 1–33, July 1989.

[72] J.-P.Jouannaud and C.Kirchner, Solving Equations in Abstract Algebras: A Rule-Based Survey of Unification, Technical Report, LRI, Université de Paris Sud, November 1989.

[73] J.A.Kalman, Axiomatizations of logics with values in groups, in *Journal of the London Mathematical Society*, Vol. 2, No. 14, 193–199, 1975.

[74] S.Kamin and J.-J.Lévy, Two generalizations of the recursive path ordering, Unpublished note, Department of Computer Science, University of Illinois, Urbana, Illinois, February 1980.

[75] D.Kapur and P.Narendran, An equational approach to theorem proving in first order predicate calculus, in *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, 1146–1153, Los Angeles, California, August 1985.

[76] D.Kapur, P.Narendran and H.Zhang, Proof by induction using test sets, in J.Siekmann (ed.), *Proceedings of the Eighth Conference on Automated Deduction*, Oxford, England, July 1986, Springer Verlag, Lecture Notes in Computer Science 230, 99–117, 1986.

[77] D.Kapur and D.R.Musser, Proof by consistency, *Artificial Intelligence*, Vol. 31, No. 2, 125–157, February 1987.

[78] D.Kapur and H.Zhang, RRL: a Rewrite Rule Laboratory, in E.Lusk, R.Overbeek (eds.), *Proceedings of the Nineth International Conference on Automated Deduction*, Argonne, Illinois, May 1988, Springer Verlag, Lecture Notes in Computer Science 310, 768–770, 1988.

[79] D.Kapur and H.Zhang, Axiomatizations of free groups, in *Journal of Automated Reasoning*, Vol. 4, No. 3, Problem corner.

[80] B.W.Kernighan and D.M.Ritchie, The C Programming Language, second edition, Prentice Hall Software Series, 1988.

[81] C.Kirchner, H.Kirchner and J.Meseguer, Operational semantics of OBJ3, in *Proceedings of the 9th International Conference on Automata, Languages and Programming*, Springer Verlag, Lecture Notes in Computer Science 241, 1988.

[82] C.Kirchner and P.Viry, Implementing Parallel Rewriting, in B.Fronhöfer and G.Wrightson (eds.), *Parallelization in Inference Systems*, Springer Verlag, Lecture Notes in Artificial Intelligence 590, 123–138, 1992.

[83] C.Kirchner, Personal communication, January 1992.

[84] D.E.Knuth and P.B.Bendix, Simple Word Problems in Universal Algebras, in J.Leech (ed.), *Proceedings of the Conference on Computational Problems in Abstract Algebras*, Oxford, England, 1967, Pergamon Press, Oxford, 263–298, 1970.

[85] R.E.Korf, Depth-first iterative deepening: an optimal admissible tree search, *Artificial Intelligence*, Vol. 27, No. 1, 97–109, September 1985.

[86] E.Kounalis and M.Rusinowitch, On Word Problems in Horn Theories, *Journal of Symbolic Computation*, Vol. 11, No. 1 & 2, 113–128, January/February 1991.

[87] R.Kowalski, Studies in the completeness and efficiency of theorem proving by resolution, Ph.D. Thesis, University of Edinburgh, 1970.

[88] D.S.Lankford, Canonical inference, Memo ATP-32, Automatic Theorem Proving Project, University of Texas, Austin, Texas, May 1975.

[89] D.S.Lankford, A simple explanation of inductionless induction, Technical report MTP-14, Mathematics Department, Louisiana Technical University, Ruston, Louisiana, 1981.

[90] D.W.Loveland, A simplified format for the model elimination procedure, *Journal of the ACM*, Vol. 16, N. 3, 349–363, July 1969.

[91] D.W.Loveland, *Automated Theorem Proving: A Logical Basis*, North-Holland, Amsterdam, 1978.

[92] E.L.Lusk and R.A.Overbeek, Non-Horn Problems, *Journal of Automated Reasoning*, Vol. 1, No. 1, 1985.

[93] E.L.Lusk and R.A.Overbeek, Reasoning about Equality, *Journal of Automated Reasoning*, Vol. 1, 209–228, 1985.

[94] E.L.Lusk and W.W.McCune, Experiments with ROO: a Parallel Automated Deduction System, in B.Fronhöfer and G.Wrightson (eds.), *Parallelization in Inference Systems*, Springer Verlag, Lecture Notes in Artificial Intelligence 590, 139–162, 1992.

[95] D.J.MacIntosh, Distributed Automated Reasoning: The Role of Knowledge in Distributed Problem Solving, PhD thesis, Clarkson University, Potsdam, New York, December 1989.

[96] W.W.McCune, An indexing mechanism for finding more general formulas, *Newsletter of the Association for Automated Reasoning*, No. 9, 7–8, January 1988.

[97] W.W.McCune and L.Wos, Some Fixed Point Problems in Combinatory Logic, *Newsletter of the Association for Automated Reasoning*, No. 10, 7–8, April 1988.

[98] W.W.McCune, OTTER 2.0 Users Guide, Technical Report ANL-90/9, Argonne National Laboratory, Argonne, Illinois, March 1990.

[99] W.W.McCune, What's New in OTTER 2.2, Technical Memorandum ANL/MCS-TM-153, Argonne National Laboratory, Argonne, Illinois, July 1991.

[100] W.W.McCune, Single axioms for the left group and right group calculi, Preprint MCS-P219-0391, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Illinois, July 1991.

[101] D.Musser, On proving inductive properties of abstract data types, in *Proceedings of the Seventh ACM Symposium on Principles of Programming Languages*, 154–162, Las Vegas, Nevada, 1980.

[102] F.Pfenning, Single axioms in the implicational propositional calculus, in in E.Lusk, R.Overbeek (eds.), *Proceedings of the Ninth International Conference on Automated Deduction*, 710–713, Argonne, Illinois, May 1988, Springer Verlag, Lecture Notes in Computer Science 310, 1988.

[103] F.J.Pelletier, Completely nonclausal, completely heuristically driven automated theorem proving, Technical Report TR82-7, Department of Computer Science, Edmonton, Alberta, Canada, 1982.

[104] F.J.Pelletier, Seventy-five problems for testing automatic theorem provers, in *Journal of Automated Reasoning*, Vol. 2, 191-216, 1986.

[105] G.E.Peterson and M.E.Stickel, Complete sets of reductions for some equational theories, *Journal of the ACM*, Vol. 28, No. 2, 233–264, 1981.

[106] G.E.Peterson, A Technique for Establishing Completeness Results in Theorem proving with Equality, *SIAM Journal of Computing*, Vol. 12, No. 1, 82–100, 1983.

[107] D.A.Plaisted, Semantic confluence tests and completion methods, *Information and Control*, Vol. 65, 182–215, 1985.

[108] A.Quaife, Andrews' Challenge Problem Revisited, *Newsletter of the Association for Automated Reasoning*, No. 15, 3–7, May 1990.

[109] M.Rusinowitch, Theorem-proving with Resolution and Superposition, *Journal of Symbolic Computation*, Vol. 11, N. 1 & 2, 21–50, January/February 1991.

[110] J.Schumann and R.Letz, PARTHEO: A High-Performance Parallel Theorem Prover, in M.E.Stickel (ed.), *Proceedings of the Tenth International Conference on Automated Deduction*, Kaiserslautern, Germany, July 1990, Springer Verlag, Lecture Notes in Artificial Intelligence 449, 28–39, 1990.

[111] K.Siegl, Gröbner Bases Computation in STRAND: A Case Study for Concurrent Symbolic Computation in Logic Programming Languages, Master thesis and Technical Report N. 90-54.0, RISC-LINZ, November 1990.

[112] R.Socher-Ambrosius, How to Avoid the Derivation of Redundant Clauses in Reasoning Systems, *Journal of Automated Reasoning*, to appear.

[113] R.Smullyan, *To Mock a Mockingbird*, Knopf, 1985.

[114] M.E.Stickel, A unification algorithm for associative-commutative functions, *Journal of the ACM*, Vol. 28, No. 3, 423–434, 1981.

[115] M.E.Stickel, A Prolog technology theorem prover, *New Generation Computing*, Vol. 2, N. 4, 371–383, 1984.

[116] M.E.Stickel and W.M.Tyson, An analysis of consecutively bounded depth-first search with applications in automated deduction, *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, Los Angeles, California, August 1985, 1073–1075.

[117] M.E.Stickel, A Prolog Technology Theorem Prover: Implementation by an Extended Prolog Compiler, *Journal of Automated Reasoning*, Vol. 4, 353–380, 1988.

[118] M.E.Stickel, The Path-Indexing Method for Indexing Terms, Technical Note 473, SRI International, October 1989.

[119] A.S.Tanenbaum, *Computer Networks*, Second Edition, Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[120] S.Tuecke, Personal communication, May 1992 and December 1992.

[121] J.-P.Vidal, The Computation of Gröbner Bases on A Shared Memory Multiprocessor, in A.Miola (ed.), *Proceedings of International Symposium on the Design and Implementation of Symbolic Computation Systems*, Capri, Italy, April 1990, Springer Verlag, Lecture Notes in Computer Science 429, 81–90, 1990. Full version available as Technical Report CMU-CS-90-163, School of Computer Science, Carnegie Mellon University, August 1990.

[122] D.H.D.Warren, An abstract Prolog instruction set, Technical Note 309, Artificial Intelligence Center, SRI International, Menlo Park, California, October 1983.

[123] S.Winker and L.Wos, Automated generation of models and counter-examples and its application to open questions in ternary Boolean algebra, in *Proceedings of the Eighth Symposium on Multiple-Valued Logic*, 251–256, IEEE, New York, 1978.

[124] L.Wos, R.Overbeek and L.Henschen, Hyperparamodulation: a refinement of paramodulation, in W.Bibel, R.Kowalski (eds.), *Proceedings of the Fifth Conference on Automated Deduction*, Les Arcs, France, 1980, Springer Verlag, Lecture Notes in Computer Science 87, 1980.

[125] L.Wos, *Automated Reasoning: 33 Basic Research Problems*, Prentice Hall, 1988.

[126] L.Wos, Searching for Open Questions, *Newsletter of the Association for Automated Reasoning*, No. 15, 8, May 1990.

[127] K.A.Yelick, Using abstraction in explicitly parallel programs, Ph.D. Thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, available as Technical Report MIT/LCS/TR-507, July 1991.

[128] K.A.Yelick and S.J.Garland, A Parallel Completion Procedure for Term Rewriting Systems, in D.Kapur (ed.), *Proceedings of the Eleventh International Conference on Automated Deduction*, Saratoga Springs, New York, June

1992, Springer Verlag, Lecture Notes in Artificial Intelligence 607, 109–123, 1992.

[129] K.A.Yelick, Personal communication, June 1992.

[130] H.Zhang and D.Kapur, First Order Theorem Proving Using Conditional Rewrite Rules, in E.Lusk, R.Overbeek (eds.), *Proceedings of the Ninth International Conference on Automated Deduction*, 1–20, Argonne, Illinois, May 1988, Springer Verlag, Lecture Notes in Computer Science 310, 1988.

Appendix A

Parallel inferences

A.1 Inference rules

We use l, p, q, r for terms, A, B, L for literals, C, D for clauses and $\succ$ is a complete simplification ordering on terms and atoms. The following inference rules, or variations thereof, have appeared in several works, e.g. in [28, 64, 109, 15, 44]:

Contraction inference rules:

Simplification:

$$\frac{S \cup \{p \simeq q, l \simeq r\}}{S \cup \{p[r\sigma]_u \simeq q, l \simeq r\}} \quad p|u = l\sigma, \quad p \succ p[r\sigma]_u, \quad p \triangleright l \vee q \succ p[r\sigma]_u$$

$$\frac{S \cup \{A \vee C, l \simeq r\}}{S \cup \{A[r\sigma]_u \vee C, l \simeq r\}} \quad A|u = l\sigma, \quad A \succ A[r\sigma]_u$$

Clausal Simplification:

$$\frac{S \cup \{L_1 \vee \ldots \vee L_n, L\}}{S \cup \{L_2 \vee \ldots \vee L_n, L\}} \quad L_1 = \neg L\sigma$$

where the literals in $L_1 \vee \ldots \vee L_n$ may be permuted, so that the one which is deleted

is the first one.

Functional Subsumption

$$\frac{S \cup \{p \simeq q, l \simeq r\}}{S \cup \{l \simeq r\}} \; (p \simeq q) \trianglerighteq (l \simeq r)$$

Clausal Subsumption

$$\frac{S \cup \{C, D\}}{S \cup \{C\}} \; D \trianglerighteq C$$

where $\trianglerighteq$ is the encompassment ordering and $\trianglerighteq$ is the subsumption ordering (see Section 2.1). In the next section we may use "simplification" for both simplification and clausal simplification and "subsumption" to mean functional subsumption of equations and clausal subsumption of clauses.

Expansion inference rules:

Superposition:

$$\frac{S \cup \{p \simeq q \vee D, l \simeq r \vee C\}}{S \cup \{p \simeq q \vee D, l \simeq r \vee C, p[r]_u\sigma \simeq q\sigma \vee C\sigma \vee D\sigma\}}$$

$$p|u \notin X,\ (p|u)\sigma = l\sigma,\ p\sigma \not\preceq q\sigma,\ p\sigma \not\preceq p[r]_u\sigma,\ (p \simeq q)\sigma \not\preceq A\sigma, \forall A \in D$$

(Ordered) Paramodulation:

$$\frac{S \cup \{A \vee D, l \simeq r \vee C\}}{S \cup \{A \vee D, l \simeq r \vee C, A[r]_u\sigma \vee C\sigma \vee D\sigma\}}$$

$$A|u \notin X,\ (A|u)\sigma = l\sigma,\ l\sigma \not\preceq r\sigma,\ (l \simeq r)\sigma \not\preceq B\sigma, \forall B \in C,\ A\sigma \not\preceq B\sigma, \forall B \in D$$

where the clause $l \simeq r \vee C$, called *paramodulant*, *paramodulates into* the clause $p \simeq q \vee D$ or $A \vee D$.

Binary (Ordered) Resolution:

$$\frac{S \cup \{L_1 \vee D, \neg L_2 \vee C\}}{S \cup \{L_1 \vee D, \neg L_2 \vee C, C\sigma \vee D\sigma\}} \quad \begin{array}{l} L_1\sigma = L_2\sigma \quad L_1\sigma \not\preceq B\sigma, \quad \forall B \in C \\ L_2\sigma \not\preceq B\sigma, \quad \forall B \in D \end{array}$$

(Ordered) Factoring:

$$\frac{S \cup \{L_1 \vee L_2 \vee C\}}{S \cup \{L_1 \vee L_2 \vee C, L_1\sigma \vee C\sigma\}} \quad L_1\sigma = L_2\sigma \quad L_1\sigma \not\preceq B\sigma, \forall B \in C - \{L_2\}$$

Hyperresolution:

$$\frac{S \cup \{\neg L_1 \vee \ldots \vee \neg L_n \vee C, A_1 \vee D_1, \ldots A_n \vee D_n\}}{S \cup \{\neg L_1 \vee \ldots \vee \neg L_n \vee C, A_1 \vee D_1, \ldots A_n \vee D_n, C\sigma \vee D_1\sigma \ldots \vee D_n\sigma\}}$$

$L_j\sigma = A_j\sigma,\ 1 \leq j \leq n\ C, D_1 \ldots D_n$ positive

An hyperresolution step involves a non-positive clause $\neg L_1 \vee \ldots \vee \neg L_n \vee C$ with n negative literals and n positive clauses $A_1 \vee D_1, \ldots A_n \vee D_n$. The non-positive clause is called *nucleus* and the positive clauses are called *satellites*. By resolving upon the n negative literals of the nucleus, an hyperresolution step generates a positive hyperresolvent $C\sigma \vee D_1\sigma \ldots \vee D_n\sigma$.

Negative Hyperresolution:

$$\frac{S \cup \{L_1 \vee \ldots \vee L_n \vee C, \neg A_1 \vee D_1, \ldots \neg A_n \vee D_n\}}{S \cup \{L_1 \vee \ldots \vee L_n \vee C, \neg A_1 \vee D_1, \ldots \neg A_n \vee D_n, C\sigma \vee D_1\sigma \ldots \vee D_n\sigma\}}$$

$L_j\sigma = A_j\sigma,\ 1 \leq j \leq n\ C, D_1 \ldots D_n$ negative

is the dual of hyperresolution, where a non-negative nucleus with n positive literals and n negative satellites generate a negative hyperresolvent.

Unit Resulting Resolution:

$$\frac{S \cup \{L_1 \vee \ldots \vee L_n, A_1, \ldots A_{n-1}\}}{S \cup \{L_1 \vee \ldots \vee L_n, A_1, \ldots A_{n-1}, L_n\sigma\}} \quad L_j\sigma = \neg A_j\sigma, 1 \leq j \leq n-1, \quad n > 1$$

where the literals in $L_1 \vee \ldots \vee L_n$ may be permuted, so that the only literal which is not resolved upon is the last one. Unit resulting resolution fits in the same pattern as hyperresolution and negative hyperresolution, if the property of being a unit clause

replaces the properties of being a positive or negative clause respectively: a non-unit nucleus $L_1 \vee \ldots \vee L_n$ resolves with $n-1$ unit satellites $A_1, \ldots A_{n-1}$ to generate a unit resolvent $L_n\sigma$.

A.2 Analysis of concurrency of inference steps

We denote an expansion inference step by $\psi_1, \ldots \psi_n \nearrow \varphi$, meaning a step generating φ from the premises $\psi_1 \ldots \psi_n$. We list first the paramodulant(s) and then the clauses paramodulated into. The full notation

$$\psi_1, \ldots \underline{\psi_j}, \ldots \psi_n \overset{\sigma}{\underset{u}{\nearrow}} \varphi$$

indicates also the unifier σ of the inference step and the position u of the term or literal unified in the underlined clause ψ_j. A contraction inference step is denoted by $\psi_1, \psi_2 \searrow \psi_3$, meaning that ψ_2 is replaced by ψ_3. The notation

$$\psi_1, \psi_2 \overset{\sigma}{\underset{u}{\searrow}} \psi_2[s]_u$$

specifies also the matching substitution σ and the position u in ψ_2 which is being rewritten. If u is the top-most position and s is empty, ψ_2 is deleted. This is the case for instance for a subsumption step. In a clausal simplification step s is empty and u is the position of the deleted literal.

We examine all possible combinations of two steps with a common premise φ. Without loss of generality, we consider only binary expansion steps. The following considerations extend naturally to steps with multiple premises.

1. **Expansion - expansion**: two expansion steps

$$\psi_1, \varphi \nearrow \psi_3$$

$$\psi_2, \varphi \nearrow \psi_4$$

are concurrent, if *concurrent-read* of the common premise φ is allowed. Executing concurrently the two steps has the same effect as executing them sequentially in any order: both ψ_3 and ψ_4 are added to the data base.

2. **Contraction - contraction**:

   (a) **Same simplifier/subsumer applied to different objects**: similar to the previous case, two contraction steps

   $$\varphi, \psi_1 \underset{u}{\searrow} \psi_1[s]_u$$

   $$\varphi, \psi_2 \underset{v}{\searrow} \psi_2[t]_v$$

   that reduce two different clauses ψ_1 and ψ_2, are concurrent under concurrent-read and concurrent execution has the same effect as any sequential execution.

   These first two cases are the simplest ones as they require only concurrent-read. Since all the following combinations will require concurrent-read, we assume henceforth that concurrent-read is allowed.

   (b) **Two different simplifier/subsumers applied to the same object**: this is the situation which is considered typically in parallel rewriting for equational languages. Two contraction steps

   $$\psi_1, \varphi \underset{u}{\searrow} \varphi[s]_u$$

   $$\psi_2, \varphi \underset{v}{\searrow} \varphi[t]_v$$

   that reduce the same clause φ require not only read-access, but also write-access to the common premise φ. Therefore, a *write-write conflict* may arise. We consider first write-access at the equation level. If *concurrent-write* is not allowed, just one step, say $\psi_1, \varphi \searrow_u \varphi[s]_u$, will commit. Then, if ψ_2 can reduce $\varphi[s]_u$, a step $\psi_2, \varphi[s]_u \searrow_{v'} \varphi[s]_u[t]_{v'}$ may be executed afterwards. This is not always possible, as the reduction at one position may destroy the redex at the other. If *concurrent-write* is allowed, e.g.

when two distinct copies of φ are stored at two different nodes, the effect of the two steps in parallel is to delete φ and add both $\varphi[s]_u$ and $\varphi[t]_v$.

If we consider write-access at the term level, we observe that if $u|v$, i.e. the positions u and v are disjoint, the two steps can proceeed concurrently. Under this condition, the effect of the concurrent execution is to replace φ by $\varphi[s]_u[t]_v$, achieving in one step the same result obtained by a sequential execution of the two steps. On the other hand, if u is a prefix of v or vice versa, the two steps are in write-write conflict. If concurrent-write is not allowed, only one step commits: if u is a prefix of v, the step at u (at v) is executed, if positions are selected in a top-down (bottom-up) fashion. As above, if concurrent-write is allowed, the combined effect of the two steps is to delete φ and add both $\varphi[s]_u$ and $\varphi[t]_v$.

(c) **A simplifier/subsumer is being simplified/subsumed**: two contraction steps

$$\varphi, \psi_1 \overset{\sigma}{\underset{u}{\searrow}} \psi_1[s]_u$$

$$\psi_2, \varphi \overset{\mu}{\underset{v}{\searrow}} \varphi[t]_v$$

experience a *read-write conflict*, as the first step needs to read the common premise φ, while the second one needs to read and write it. Conflicts of this kind do not occur in parallel rewriting for equational languages, since in those applications the simplifiers, i.e. the equations in the program, are not subject to simplification themselves. However, such a situation does appear in contraction-based theorem proving, mostly because of backward contraction. For instance, if the strategy requires that every newly generated clause undergo forward contraction *before* being cleared for any other inference, the second one of the two steps above must be a backward contraction steps. Indeed, if ψ_2 were reducing φ as part of the forward contraction of the latter, φ would not be considered as simplifier.

If the first step may read φ, both steps may proceed, reducing both ψ_1 and φ: this is the same effect as in a sequential execution, such that

first φ reduces ψ_1 and next ψ_2 reduces φ. If the first step cannot read φ, because of the conflict with the second step, only the second step commits. However, we show that even if the first step is not executed, ψ_1 is guaranteed to be reduced to $\psi_1[s]_u$ or to another reduced form. This shows that *read-write conflict between two contraction steps is not a real conflict*, since both reducible clauses will be reduced regardless of the order of execution of the steps. We distinguish different cases for some specific inference rules:

i. **Subsumption - subsumption**: we have $\psi_1 \succeq \varphi$ and $\varphi \succeq \psi_2$. By transitivity of the ordering $\succeq$, $\psi_1 \succeq \psi_2$ follows, i.e. if ψ_2 subsumes φ preventing φ from subsuming ψ_1, ψ_2 may subsume ψ_1 also. The same holds for functional subsumption.

ii. **Clausal simplification - subsumption**: we have for the first step $\psi_1 = L_1 \dots L_n$, $\varphi = L$ and $\neg L\sigma = L_1$. For the second step it is $\psi_2 = L'$ and $L'\mu = L$. It follows that $\neg L'\mu\sigma = L_1$, i.e. ψ_2 can subsume φ and simplify ψ_1 as well.

iii. **Subsumption - clausal simplification**: we have $\varphi = L_1 \dots L_n$, $\psi_1 = L_1\sigma \dots L_n\sigma \vee C$, $\psi_2 = L$ and $\neg L\mu = L_1$. It follows that $\neg L\mu\sigma = L_1\sigma$ holds. Thus ψ_2 can eliminate the literal L_1 in φ, replacing φ by $\varphi' = L_2 \dots L_n$, and the literal $L_1\sigma$ in ψ_1, replacing ψ_1 by $\psi_1' = L_2\sigma \dots L_n\sigma \vee C$. Then φ' can subsume ψ_1'.

iv. **Clausal simplification/simplification - clausal simplification**: φ and ψ_2 are unit clauses L_1 and L_2 such that $\neg L_2\mu = L_1$. The second step generates the empty clause, so that it becomes irrelevant whether ψ_1 is reduced.

v. **Simplification - subsumption**: for the first step it is $\varphi = (l \simeq r)$, $\psi_1 = c[l\sigma]_u$ and φ reduces ψ_1 to $c[r\sigma]_u$. For the second step we have $\varphi \trianglerighteq \psi_2$, i.e. $\psi_2 = (l' \simeq r')$, $l = d[l'\mu]_v$ and $r = d[r'\mu]_v$. It follows that $\psi_1|uv = l'\mu\sigma$, i.e. $l' \simeq r'$ can simplify ψ_1 to $c[d[r'\mu]_v\sigma]_u = c[r\sigma]_u$: ψ_1 is reduced to the same clause regardless of whether it is simplified

by $l \simeq r$ or by $l' \simeq r'$. The only difference is in the matching substitutions and positions involved in the two steps.

vi. **Subsumption - simplification**: we consider clausal subsumption, as the reasoning for functional subsumption is analogous. We have $\psi_2 = (l \simeq r)$, $\varphi = A[l\mu]_v \dots L_n$ and $\psi_1 = A[l\mu]_v\sigma \dots L_n\sigma \vee C$. The equation ψ_2 reduces φ to $\varphi' = A[r\mu]_v \dots L_n$ and it can reduce also ψ_1 to $\psi_1' = A[r\mu]_v\sigma \dots L_n\sigma \vee C$. In turn, φ' can subsume ψ_1'.

vii. **Clausal simplification - simplification**: we have $\psi_2 = (l \simeq r)$, $\varphi = A[l\mu]_v$ and $\psi_1 = L_1 \dots L_n$, with $\neg A[l\mu]_v\sigma = L_1$. Then ψ_2 can rewrite φ to $\varphi' = A[r\mu]_v$ and ψ_1 to $\psi_1' = \neg A[r\mu]_v\sigma \dots L_n$, so that φ' can simplify ψ_1'.

viii. **Simplification - simplification**: we further distinguish two subcases:

A. **Simplification - simplification of the left hand side**: we have $\psi_2 = (l \simeq r)$, $\varphi = (l' \simeq r') = (d[l\mu]_v \simeq r')$ and $\psi_1 = C[l'\sigma]_u = C[d[l\mu]_v\sigma]_u$. Thus ψ_2 reduces φ to $\varphi' = (d[r\mu]_v \simeq r')$ and ψ_1 to $\psi_1' = C[d[r\mu]_v\sigma]_u$. However, it is not guaranteed that φ' can simplify ψ_1': this requires that $C[d[r\mu]_v\sigma]_u \succ C[r'\sigma]_u$, a condition which is not necessarily implied by the conditions for the above simplification steps by ψ_2. However, regardless of the order of the steps, both φ and ψ_1 are reduced.

B. **Simplification - simplification of the right hand side**: we have $\psi_2 = (l \simeq r)$, $\varphi = (l' \simeq r') = (l' \simeq d[l\mu]_v)$ and $\psi_1 = C[l'\sigma]_u$. In addition, we know that $C[l'\sigma]_u \succ C[r'\sigma]_u$ and $r' = d[l\mu]_v \succ d[r\mu]_v$. The simplifier ψ_2 reduces φ to $\varphi' = (l' \simeq d[r\mu]_v)$ and in turn φ' reduces ψ_1 to $C[d[r\mu]_v\sigma]_u$, since $C[l'\sigma]_u \succ C[d[r\mu]_v\sigma]_u$ follows from $C[l'\sigma]_u \succ C[r'\sigma]_u$ and $r' = d[l\mu]_v \succ d[r\mu]_v$, by monotonicity and transitivity of the ordering.

3. **Expansion - contraction**:

(a) **A parent is also a simplifier/subsumer**: an expansion step

$$\psi_1, \varphi \nearrow \psi_3 \text{ or } \varphi, \psi_1 \nearrow \psi_3$$

and a contraction step

$$\varphi, \psi_2 \underset{v}{\searrow} \psi_2[t]_v$$

are concurrent if concurrent-read is allowed, since the common premise is not the one reduced by the contraction step.

(b) **A parent is being simplified/subsumed**: the last combination is given by an expansion step

$$\psi_1, \underline{\varphi} \overset{\sigma}{\underset{u}{\nearrow}} \psi_3 \text{ or } \underline{\varphi}, \psi_1 \overset{\sigma}{\underset{u}{\nearrow}} \psi_3$$

with mgu σ at position u in φ and a contraction step

$$\psi_2, \varphi \overset{\mu}{\underset{v}{\searrow}} \varphi[t]_v$$

with matching substitution μ at position v in φ, where the common premise φ is reduced. Similar to Case 2c, this combination of steps is largely due to backward contraction: if we assume that φ is not allowed to expand before undergoing forward contraction, the second step must be a backward contraction step.

If we consider access at the equation level, we have a *read-write conflict*, as the expansion step needs to read φ, while the contraction step needs to read it and write it. According to a contraction-first search plan, the conflict should be solved by giving priority to the contraction step, so that the contraction step commits, whereas the expansion step does not. This has the same effect as in a sequential execution controlled by a contraction-first search plan.

As for Case 2b, if we consider access at the term level, a conflict arises only if the two positions u and v are not disjoint. In such a situation the contraction step modifies exactly the literal or term involved in the

unification process in the expansion step. If the two positions are disjoint, there is no real conflict and the two steps may proceed in parallel, with the overall effect of adding ψ_3 and replacing φ by $\varphi[t]_v$. However, this concurrent execution may introduce some extra redundancy with respect to a sequential derivation. Under the hypothesis that the positions u and v are disjoint, the contraction step does not really prevent the expansion step: the latter is still applicable to $\varphi[t]_v$. Therefore, it may seem that the concurrent execution of the two steps does not introduce extra redundancy. Still, if the contraction step is executed first, there are two advantages. First, other contraction steps may intervene between the reduction of φ to $\varphi[t]_v$ and the application of the expansion rule to $\varphi[t]_v$, and the latter may be prevented. Second, $\varphi[t]_v$ is smaller than φ and therefore the clause generated by $\varphi[t]_v$ will be smaller than the one generated by φ. On the other hand, allowing the two steps to commit in parallel may generate earlier some useful equation. This is an instance of the contrast between enforcing a contraction-first search plan and letting inferences proceed eagerly.

These general considerations apply as follows to the specific inference rules:

i. **Expansion - subsumption**: if $\psi_2, \varphi \searrow^{\mu}_{v} \varphi[t]_v$ is a subsumption step, we have $\psi_2\mu \subseteq \varphi$ or $\varphi \blacktriangleright \psi_2$ and v is the top-most position in φ, occupied by φ itself. Then the positions u and v are not disjoint: by deleting φ, the subsumption step inhibits the expansion step.

ii. **Expansion - clausal simplification**: if $\psi_2, \varphi \searrow^{\mu}_{v} \varphi[t]_v$ is a clausal simplification step, we have $\varphi = L_1 \dots L_n$, $\psi_2 = L$, $\neg L\mu = L_1$ and the position v is the position of the literal L_1 in φ, i.e. $v = 1$. Since v is a literal position, the position u cannot be a proper prefix of v. Therefore, v is a proper prefix of u or $v = u$ or $v|u$. For v to be a proper prefix of u, the step $\psi_1, \underline{\varphi} \nearrow^{\sigma}_{u} \psi_3$ needs to be a paramodulation/superposition step, paramodulating into L_1. For the

condition $v = u$ to hold, $\psi_1, \underline{\varphi} \nearrow_u^\sigma \psi_3$ needs to be a resolution step, resolving upon literal L_1. In both cases, the clausal simplification step deletes the literal L_1 resolved upon or paramodulated into and thereby inhibits the expansion step.

If $u|v$, the expansion step involves a literal of φ different from L_1. If this is the case, both steps may proceed, deleting L_1 from φ and adding ψ_3. In addition, ψ_2 may simplify ψ_3 as well, deleting the literal $L_1\sigma$ that ψ_3 has inherited from φ.

iii. **Expansion - simplification**: if $\psi_2, \varphi \searrow_v^\mu \varphi[t]_v$ is a simplification step, we have $\psi_2 = (l \simeq r)$, $\varphi|v = l\mu$, $t = r\mu$ and the position v is a term position in φ. We further distinguish based on the role played by φ in the expansion step:

   A. **Resolution - simplification**: if $\psi_1, \underline{\varphi} \nearrow_u^\sigma \psi_3$ is a resolution step, u is a literal position. Either u is a proper prefix of v, i.e. the simplification step applies inside the literal resolved upon, or $u|v$, i.e. the simplification step applies at another literal. In the first case, the commitment of the simplification step may destroy the resolution step by modifying the literal resolved upon. In the second case, the two steps may proceed and ψ_2 can simplify also the subterm $(\varphi|v)\sigma$ which ψ_3 has inherited from φ.

   B. **Paramodulation/superposition - simplification**: if φ paramodulates into ψ_1, u is a term position: more precisely, it is the root position of a side of an equation in φ. Either u is a prefix of v or $u|v$. In the first case, the simplification step at v applies inside the left hand side of the paramodulating equation and thereby inhibits the paramodulation step. In the second case, the simplification step at v applies either in the right hand side of the paramodulating equation or in another literal of φ. Both steps may proceed and ψ_2 can simplify also the subterm $(\varphi|v)\sigma$ which ψ_3 has inherited from φ.

If φ is paramodulated into by ψ_1, u is any term position, not necessarily a root position. Either u and v are not disjoint, with the consequent inhibition of the paramodulation step, or $u|v$, with possible concurrent execution and the further simplification of ψ_3 by ψ_2. Note that if $u|v$, the two positions may be as far apart as in two different literals or as close to each other as two independent positions in the same term.