Analysis and Representation of Equational Proofs Generated by a Distributed Completion Based Proof System

Jörg Denzinger, Stephan Schulz
{denzinge, s_schulz}@informatik.uni-kl.de

Department of Computer Science
University of Kaiserslautern
Postfach 3049
67653 Kaiserslautern

April 6, 1994

Abstract

Automatic proof systems are becoming more and more powerful. However, the proofs generated by these systems are not met with wide acceptance, because they are presented in a way inappropriate for human understanding.

In this paper we pursue two different, but related, aims. First we describe methods to structure and transform equational proofs in a way that they conform to human reading conventions. We develop algorithms to impose a hierarchical structure on proof protocols from completion based proof systems and to generate equational chains from them.

Our second aim is to demonstrate the difficulties of obtaining such protocols from distributed proof systems and to present our solution to these problems for provers using the TEAMWORK method. We also show that proof systems using this method can give considerable help in structuring the proof listing in a way analogous to human behaviour.

In addition to theoretical results we also include descriptions on algorithms, implementation notes, examples and data on a variety of examples.

Contents

1 Introduction **5**

2 Concepts of equational reasoning **7**

2.1 Annotations and basic definitions 7

2.2 Completion based proof procedures 8

2.2.1 A completion procedure 11

2.3 Distributed deduction - The TEAMWORK method 14

3 Representation of computer generated proofs **17**

3.1 The language PCL 18

3.2 Working with PCL listings 23

3.3 Extracting the proof 25

4 Structuring proof listings **27**

4.1 Partitioning the proof – top-down vs. bottom-up 28

4.2 Frequently used steps 28

4.3 Important intermediate results 29

4.4 Isolated proof segments 29

4.5 Syntactical criteria 31

4.6 Using outside knowledge 33

4.7 Analysing the applied inference rules 34

4.8 Sectioning long proofs 34

4.9 "What to avoid" - not every candidate may be suitable 35

4.10 Combinations of different criteria 36

4.11 An algorithm for the structuring of PCL listings 38

4.12 Evaluating the different criteria for lemma generation 40

5 Proof presentation **43**

5.1 Equational Chains: A calculus for proof presentation 43

5.2 PCL listings and equational chains 45

5.3 An algorithm for proof transformation 49

6 Dealing with a distributed proof system **53**

6.1 Measuring without disturbing 53

6.2 Sequentializing parallel proofs 54
6.2.1 Eliminating redundancies 55
6.3 Handling large amounts of data 57

7 Benefits from going distributed **58**
7.1 Increased power of the proof system 58
7.2 Shorter proof protocols 58
7.3 Easier handling of extreme protocols 59
7.4 Improved lemma recognition 59

8 Implemented Programs **62**
8.1 The DISCOUNT system 64
8.1.1 Changes in the configuration files 64
8.1.2 Logging the proof session 64
8.1.3 Reproducing proofs: The `rdiscount` executable 66
8.1.4 Generating PCL listings: `pcl` and `rpcl` 66
8.2 Programs dealing with PCL protocols 67
8.2.1 Extracting the proof: `extract` and `mextract` 67
8.2.2 Dealing with extreme examples: `revert` and `rextract` 68
8.2.3 Revealing the structure: `lemma` 69
8.2.4 Generating equational chains: `proof` 71

9 Conclusion **73**

A A short log file of a proof session **74**

B Examples **75**
B.1 A ring with $x^2 = x$ is Abelian 75
B.1.1 The problem 75
B.1.2 The proof protocol 76
B.1.3 Lemmata 77
B.1.4 The proof 78
B.2 A ring with $x^3 = x$ is Abelian 81
B.2.1 The Problem 81
B.2.2 The proof 82
B.3 A problem from the domain of lattice ordered groups 88

B.3.1 The problem 88
B.3.2 The proof 89
B.4 Specifications of some other problems 93
B.4.1 The problem SelfInverse 94
B.4.2 The problem Fibgroup 94
B.4.3 The problem BoolAssoc 95
B.4.4 The problem Lattice1 95
B.4.5 The problem DeMorgan 96
B.4.6 The problem Lattice2 97
B.4.7 The problem Z22 98

References **99**

List of Tables

1 Numbers of steps in PCL listings 26
2 Necessary and executed inferences in protocols of distributed proofs . . 60
3 Percentage of arbitrary and selected results needed in the found proof . 61
4 The new DISCOUNT programs and their options 65
5 Constants and options in `lemma` 70
6 Letter codes and corresponding criteria for use with `-criteria` 71

List of Figures

1 Basic structure of a team in the TEAMWORK method 14
2 A cycle between two team meetings 15
3 Proof graph corresponding to the example on page 30 32
4 Team work and name spaces 56
5 A concept for a system generating, analyzing and transforming proofs . 62
6 Structure of the implemented system 63
7 Lemma structure according to B.1.3 78

1 Introduction

Automatic theorem provers have reached a point in their development where they could support human experts in many routine tasks. However, the proofs generated by these systems have not been met with wide acceptance. The main reason for this is the inappropriate way automatically generated proofs are presented. Up to now much work has been invested into developing more powerful provers, but very little effort has been made to present the generated proofs to a user who is not interested in the details of the prover but in the proof itself. This is especially true for equational proof systems, where only very rudimentary suggestions for proof presentation have been brought forward.

In this paper we will address two problems in the field of proof presentation. First we will develop concepts and algorithms to present equational proofs generated by completion based proof systems in a structured format appropriate for human understanding. Our second aim is to generate such proofs from a distributed proof system. Treating these two problems together is justified by two facts: First, distributed theorem provers introduce new problems for proof presentation. These problems have to be dealt with if such provers are to be used by humans. Secondly, we found that distributed proof systems using the TEAMWORK method can give a lot of help in structuring the proofs and often achieve "better" proofs than sequential systems. In particular we can use information gained during the proof process (suggestions by the *referees* used with TEAMWORK) in addition to the more conventional post mortem criteria to generate lemmata. In this way we emulate a human expert, who also bases the structure of his presentation of a proof on information from the proof process *and* on the final proof.

Our approach to proof presentation partitions the procedure into a number of separate phases. In the first phase the proof is found and a step by step listing of each inference done by the prover is generated. In the case of a distributed proof system this listing has to be sequentialized before further analysis. The second main step is the extraction of the inference steps actually used in the proof. In the next phase we try to structure the resulting listing in a hierarchical way and to reveal important intermediate results (or *lemmata*). The final step transforms the structured listing to a calculus conforming to human reading conventions. In our case this calculus uses equational chains.

This report is organized as follows: Section 2 provide a short introduction to the basic concepts of a completion based prover and the TEAMWORK method. Section 3 discusses proof representation, and introduces our protocol language PCL. Additionally we present first results from a very basic analysis of some proof protocols.

Section 4 is dedicated to the structuring of proof protocols. We introduce a couple of criteria for lemma detection and discuss their comparative merits. The next section describes how to transform the proof to equational chains, a calculus acceptable for human digestion. Section 6 describes special problems in dealing with distributed proof systems as well as their solution, and section 7 discusses the beneficial aspects of using a distributed proof system in depth.

The next section describes details of the implemented programs. This includes changes

to the DISCOUNT system, our TEAMWORK based equational prover, and the new programs developed to deal with analysis, structuring and presentation of proofs. The final section concludes the paper with an evaluation and some remarks about our future plans. The appendix provides some examples of different protocols and a couple of automatically generated proofs.

2 Concepts of equational reasoning

An equational theorem prover is a system trying to deal with the following problem:

> Given a set E of equations (over terms), is an equation s=t a logical consequence of E (written as $\mathtt{s} =_{\mathtt{E}} \mathtt{t}$)?

In order to allow a more detailed discussion of the related problems we need some basic definitions. We assume the reader to be familiar with rewriting techniques and use standard notations. For a more in-depth introduction to the field, using similar notations, see [Av91] and [Av90].

2.1 Annotations and basic definitions

- A *term* t is a recursive structure build from a set F of function symbols and a set V of variables. The set of all terms for given sets F and V is called TERM(F, V).A *ground term* is a term not containing any variables, the set of ground terms is denoted as TERM(F).

- We write $t|_p$ to denote the subterm of t at the position p. The top position is written as λ, and $t|_{p.q} \equiv (t|_p)|_q$.

- A substitution σ is a function mapping a finite set of variables into the set of terms. We write $\sigma = \{x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n\}$ if σ maps x_i to t_i for $1 \leq i \leq n$. σ_{id} represents the empty substitution $\{\}$, $mgu(t_1, t_2)$ the most general unifier for t_1 and t_2.

- The *encompassment ordering* $\trianglerighteq$ is defined by $\mathtt{s} \trianglerighteq \mathtt{t}$ iff $\sigma(\mathtt{s}) = \mathtt{t}|_p$ for a substitution σ and a position p. $\triangleright$ is the strict part of $\trianglerighteq$.

- $t[p \leftarrow t']$ denotes the term t with the subterm at position p replaced by t'.

- An equation is a pair $(\mathtt{s}, \mathtt{t}) \in (\mathtt{TERM(F, V)} \times \mathtt{TERM(F, V)})$. We write s=t instead of (s, t). We always regard equations as symmetrical, so that s=t also represents t=s.

- A *reduction ordering* $>$ is a Noetherian ordering compatible with the term structure and stable with respect to substitutions. A reduction ordering total on ground terms is called a *ground reduction ordering.*

- A rule is a pair $(\mathtt{l}, \mathtt{r}) \in (\mathtt{TERM(F, V)} \times \mathtt{TERM(F, V)})$ with $\mathtt{Var(r)} \subset \mathtt{Var(l)}$ (Var(t) denotes the set of variables occurring in t). We write a rule as l→r.

- A rule l→r is *compatible* with a reduction ordering $>$ if $\mathtt{l} > \mathtt{r}$. A system of rules R is compatible with $>$ if all rules in R are compatible.

An equation $s{=}t$ is a logical conclusion form a set of equation E if it is valid in each model of E. According to Birkhoffs theorem $s =_E t$ holds, if and only if we can transform s into t by application of the equations from E. Given this result we can give an *operational* characterization of E-equality.

- Given a set E of equations we define a symmetrical relation $\vdash\!\dashv_E$ as follows:

  $t_1 \vdash\!\dashv_E t_2$ iff there exists an equation $s{=}t \in E$, an position p in t_1 and a substitution σ with $\sigma(s) \equiv t_1|_p$ and $t_2 \equiv t_1[p \leftarrow \sigma(t)]$.

  Then $=_E$ is the reflexive and transitive closure of $\vdash\!\dashv_E$, that is $t_1 =_E t_2$ if and only if $t_1 \overset{*}{\vdash\!\dashv}_E t_2$.

Knuth-Bendix completion (see [KB70]) tries to substitute the application of equations with the application of rules (oriented equations). It tries to generate a confluent rule system by orienting equations according to a reduction ordering, generating new equations from *critical pairs* and using rules to simplify the knowledge base.

In order to handle unorientable equations the calculus introduced by Knuth and Bendix has been extended in [BDP89]. The extended calculus tries to generate only a *ground confluent* system, and orientable instances of equations can be used for simplifications.

- A system R of rules defines a rewriting relation $\Longrightarrow_R$ as follows:

  $t_1 \Longrightarrow_R t_2$ iff there exists a rule $l \to r$, a place p in t_1 and a substitution so that $t_1|p \equiv \sigma(l)$ and $t_2 \equiv t_1[p \leftarrow \sigma(r)]$

- A term that cannot be reduced with $\Longrightarrow_R$ is said to be in *normal form* with respect to R.

- The set $R_E = \{\sigma(s) \to \sigma(t) | \sigma(s) > \sigma(t), s{=}t \in E, \sigma \text{a substitution}\}$ is called the set of *orientable instances* for a set of equations E and a reduction ordering $>$.

- We define $R(E) = R \cup R_E$ for given sets E, R and an ordering $>$.

- *Critical pairs* are defined as follows: Let $l_1 \to r_1$ and $l_2 \to r_2$ be two rules and let $>$ be a (ground) reduction ordering. Let p be a non-variable position in l_1, $\sigma = \mathrm{mgu}(l_1|_p, l_2)$. If $\sigma(l_1) \not< \sigma(r_1)$ and $\sigma(l_2) \not< \sigma(r_2)$ then $\langle \sigma(r_1), \sigma(l_1[p \leftarrow r_2]) \rangle$ is called a critical pair between the two rules at position p.

- Finally, critical pairs between equations can be builtby treating an equation $s{=}t$ as the two rules $s \to t$ and $t \to s$. The set of all critical pairs that can be build for an E and R is called $CP(E, R)$.

2.2 Completion based proof procedures

Input for a completion based proof system are a set of equations E, a (ground) reduction ordering $>$ and a (skolemized) goal $s{=}t$. In order to prove $s =_E t$ the prover

successively orients equations from $\mathtt{E}$, creating a set $\mathtt{R}$ of rules, generates new equations by building critical pairs between rules and equations from $\mathtt{E}$ and $\mathtt{R}$ and uses the rules from $\mathtt{R}$ and orientable instances from equations for simplifications of both the goal and the knowledge base. This process is called *completion*. The goal is proved if the normal forms of both sides (with respect to $\mathtt{R}(\mathtt{E})$) are identical or if an equation subsuming the goal is being generated.

In order to guarantee completeness and correctness of the prover we need to discuss completion more formally. The authors in [BDP89] suggested an inference system describing the operations on the rules and equations. We use this inference system, with an added rule to allow for subsumption. If these inference rules are applied using a *fair* strategy, the resulting prover can be shown to be both complete and correct.

Definition 1 : The inference system U (Unfailing Completion)
Let $>$ be a ground reduction ordering. The inference system works on pairs $(\mathtt{E},\mathtt{R})$, where $\mathtt{E}$ is a set of equations and $\mathtt{R}$ is a set of rules compatible with $>$.

(U1) Orient an equation

$$\frac{(\mathtt{E} \cup \{\mathtt{s}{=}\mathtt{t}\},\mathtt{R})}{(\mathtt{E},\mathtt{R} \cup \{\mathtt{s}{\rightarrow}\mathtt{t}\})} \qquad \text{if } \mathtt{s} > \mathtt{t}$$

(U2) Generate an equation

$$\frac{(\mathtt{E},\mathtt{R})}{(\mathtt{E} \cup \{\mathtt{s}{=}\mathtt{t}\},\mathtt{R})} \qquad \text{if } \mathtt{s}{\vdash\!\dashv}_{\mathtt{E}\cup\mathtt{R}}\mathtt{u}{\vdash\!\dashv}_{\mathtt{E}\cup\mathtt{R}}\mathtt{t},\ \mathtt{s} \not> \mathtt{u},\ \mathtt{t} \not> \mathtt{u}$$

(U3) Simplify an equation

(a) $$\frac{(\mathtt{E} \cup \{\mathtt{s}{=}\mathtt{t}\},\mathtt{R})}{(\mathtt{E} \cup \{\mathtt{u}{=}\mathtt{t}\},\mathtt{R})} \qquad \text{if } \mathtt{s} \Longrightarrow_{\mathtt{R}} \mathtt{u}$$

(b) $$\frac{(\mathtt{E} \cup \{\mathtt{s}{=}\mathtt{t}\},\mathtt{R})}{(\mathtt{E} \cup \{\mathtt{u}{=}\mathtt{t}\},\mathtt{R})} \qquad \text{iff } \mathtt{s} \Longrightarrow_{\mathtt{R}_\mathtt{E}} \mathtt{u} \text{ using a rule } \mathtt{l}{\rightarrow}\mathtt{r} \text{ with } \mathtt{s} \rhd \mathtt{l}$$

(U4a) Delete an equation

$$\frac{(\mathtt{E} \cup \{\mathtt{s}{=}\mathtt{s}\},\mathtt{R})}{(\mathtt{E},\mathtt{R})}$$

(U4b) Subsume an equation

$$\frac{(\mathtt{E} \cup \{\mathtt{s}{=}\mathtt{t},\mathtt{u}{=}\mathtt{v}\},\mathtt{R})}{(\mathtt{E} \cup \{\mathtt{s}{=}\mathtt{t}\},\mathtt{R})} \qquad \text{if } \mathtt{u}|p \equiv \sigma(\mathtt{s}), \mathtt{v} \equiv [p \leftarrow \mathtt{t}],\ \mathtt{u} \rhd \mathtt{s}, \text{ for a position } p \text{ and a substitution } \sigma$$

(S1) Simplify the right side of a rule

$$\frac{(\mathtt{E},\mathtt{R} \cup \{\mathtt{s}{\rightarrow}\mathtt{t}\})}{(\mathtt{E},\mathtt{R} \cup \{\mathtt{s}{\rightarrow}\mathtt{u}\})} \qquad \text{if } \mathtt{t} \Longrightarrow_{\mathtt{R}(\mathtt{E})} \mathtt{u}$$

(S2) Simplify the left side of a rule

$$\frac{(\mathtt{E},\mathtt{R} \cup \{\mathtt{s}{\rightarrow}\mathtt{t}\})}{(\mathtt{E} \cup \{\mathtt{u}{=}\mathtt{t}\},\mathtt{R})} \qquad \text{if } \mathtt{s} \Longrightarrow_{\mathtt{R}(\mathtt{E})} \mathtt{t} \text{ using a rule } \mathtt{l}{\rightarrow}\mathtt{r} \text{ with } \mathtt{s} \rhd \mathtt{l}$$

We write $(\mathtt{E},\mathtt{R}) \vdash_{\mathtt{U}} (\mathtt{E}',\mathtt{R}')$ if $(\mathtt{E},\mathtt{R})$ can be transformed into $(\mathtt{E}',\mathtt{R}')$ using one inference rule from **U**. An application of an inference rule from **U** does not change the equality relation described by the system:

Theorem 1 : Correctness of U
Assume $(\mathtt{E},\mathtt{R}) \vdash_{\mathtt{U}} (\mathtt{E}',\mathtt{R}')$ and consider $\mathtt{R}$ to be compatible with a reduction ordering $>$. The the following facts hold true:

- $\mathtt{R}'$ is compatible with $>$.
- $=_{\mathtt{E}\cup\mathtt{R}} = =_{\mathtt{E}'\cup\mathtt{R}'}$.

A proof for an equation $\mathtt{s}=\mathtt{t}$ in $\mathtt{E}\cup\mathtt{R}$ is a chain of terms connected by one application of a rule or equation from $\mathtt{E}\cup\mathtt{R}$. Using the ground reduction ordering $>$ and the encompassment ordering $\unrhd$ a well founded proof ordering $>_p$ that is compatible with the proof structure can be constructed (see [BDP89]), such that the following theorem holds:

Theorem 2 : Proof orderings
Let $>_p$ be a proof ordering, and assume $(\mathtt{E},\mathtt{R}) \vdash_{\mathtt{U}} (\mathtt{E}',\mathtt{R}')$. Let $\mathtt{B}$ be a proof for $\mathtt{s}=\mathtt{t}$ in $\mathtt{E}\cup\mathtt{R}$. Then there exists a proof $\mathtt{B}'$ be a proof for $\mathtt{s}=\mathtt{t}$ in $\mathtt{E}'\cup\mathtt{R}'$ with $\mathtt{B} \geq_{\mathtt{p}} \mathtt{B}'$. In particular, if $\mathtt{B}$ contains a peak $\mathtt{s} \Longleftarrow_{\mathtt{R(E)}} \mathtt{u} \Longrightarrow_{\mathtt{R(E)}} \mathtt{t}$ and $\mathtt{B'}$ is constructed from $\mathtt{B}$ by replacing this peak with a new equation generated using rule (U2), then $\mathtt{B} >_{\mathtt{p}} \mathtt{B}'$.

If only minimal (with respect to $>_p$) proofs are considered and any peak in these is eventually eliminated by generating new equations, it can be assured that an equivalent proof containing no peaks (a so called V-proof) does exist for every ground proof of a valid equation. To ensure that every peak is eliminated we have to make certain demands on the sequence of the inferences.

Definition 2 : U-fairness

- A sequence $(\mathtt{E_i},\mathtt{R_i})_{\mathtt{i}\in\mathbb{N}}$ is called an **U**-derivation if $(\mathtt{E_i},\mathtt{R_i}) \vdash_{\mathtt{U}} (\mathtt{E_{i+1}},\mathtt{R_{i+1}})$ for all $\mathtt{i} \in \mathbb{N}$.
- The system $(\mathtt{R}^\infty,\mathtt{E}^\infty)$ of *persistent* rules and equations (for a given **U**-derivation) is defined by by:
$$\mathtt{E}^\infty = \bigcup_{\mathtt{i}\geq 0} \bigcap_{\mathtt{j}\geq \mathtt{i}} \mathtt{E_j} \quad \text{and} \quad \mathtt{R}^\infty = \bigcup_{\mathtt{i}\geq 0} \bigcap_{\mathtt{j}\geq \mathtt{i}} \mathtt{R_j}$$
- An **U**-derivation $(\mathtt{E_i},\mathtt{R_i})_{\mathtt{i}\in\mathbb{N}}$ is fair if
$$\mathtt{CP}(\mathtt{E}^\infty,\mathtt{R}^\infty) \subseteq \bigcup_{\mathtt{i}\geq 0} \mathtt{E_i}$$

An derivation is fair, if every critical pair between persistent rules will eventually be built and added to the set of equations. It can be shown that the following theorem holds for **U**-fair derivations.

Theorem 3 : Completeness of fair U-derivations
Let $(\mathtt{E_i}, \mathtt{R_i})_{\mathtt{i}\in\mathbb{N}}$ be a fair **U**-derivation.

- The final system of rules and equations describes the same equality as the initial system: $=_{\mathtt{E}^\infty \cup \mathtt{R}^\infty} = =_{\mathtt{E_0} \cup \mathtt{R_0}}$.
- If $\mathtt{s} =_{\mathtt{E}} \mathtt{t}$ holds,then there is an `i` such that the normal forms of `s` and `t` with respect to $(\mathtt{E_i}, \mathtt{R_i})$ are identical.

This result can be used to build completion based provers for equational logic. The following algorithm describes a possible implementation of such a proof system. It will either proof a given goal or it will try to generate a (possibly infinite) ground confluent system of rules and equations that can be used for calculations in equationally specified algebraic structures.

In order to keep control over the critical pairs already considered it uses not two, but three sets of term pairs to represent the current state of a completion process: A set `E` of processed, but unorientable equations, a set `R` of rules (processed and oriented equations) and a set `CP` of unprocessed equations.

The completion algorithm starts out with empty sets `R` and `E`, and with the initial axioms in `CP`. It examines each equation in `CP`, reduces it to normal form with respect to `E` and `R`, uses it to build new critical pairs (to be added to `CP`) and to eliminate redundancies from `R` and `E` by simplification (this process is known as *interreduction*). It is then added to either `R` (if it can be oriented according to $>$) or `E`.

To build a prover on top of the completion algorithm the goal is brought to normal form with respect to each successive `E` and `R`. If these normal forms are identical or if an equation from `E` subsumes the goal, the goal is proved. Please note that both, completeness and efficiency of the proof process, depend on the order in which the equations from `CP` will be considered, with both goals often conflicting.

2.2.1 A completion procedure

Input:	`M`	A set of equations
	$>$	A ground reduction ordering
	`PROOFMODE`	A boolean value, `TRUE`, if a single goal is to be proved, `FALSE` if a ground confluent end system is desired
	`(gs,gt)`	The goal to be proved (only if `PROOFMODE` has the value `TRUE`

Variables:	`R`	The set of processed rules
	`E`	The set of processed equations
	`CP`	The set of unprocessed equations
	`s,t`	The sides of the unprocessed term pair considered at a given moment
	`l,r`	The terms of a newly generated rule
	`u,v`	The sides of a processed term pair considered again
	`u',v'`	Possibly simplified descendants from `u` and `v`
Functions:	`NOTEMPTY(list)`	FALSE, if `list` is empty, TRUE otherwise
	`FIRST(list)`	First element of `list`
	`EXCEPTFIRST(list)`	`list` without it's first element
	`NORMALFORM(t,R)`	Calculates a normal form for `t` with respect to `R`
	`SUBSUM(e,E)`	Tests, whether the equation `e` is subsumed by an equation in `E` or whether an equivalent equation already exists in `E`
	`INSERT(list,e)`	Inserts the term pair `e` into `list` in a way that ensures that for any given pair in `list` only finitely many pairs will be inserted in front of it (this ensures fairness)
	`CPS(e,E)`	Returns all critical pairs that can be build between `e` and term pairs from `E`

```
CP := M;
E := {};
R := {};
WHILE NOTEMPTY(CP)
  IF PROOFMODE = TRUE THEN
    gs := NORMALFORM(gs,R(E));
    gt := NORMALFORM(gt,R(E));
    IF gs = gt THEN END; A proof has been found
  ENDIF
  (s,t) := FIRST(CP);
  CP := EXCEPTFIRST(CP);
  s := NORMALFORM(s,R(E));
  t := NORMALFORM(t,R(E));
  IF s≠t AND NOT(SUBSUM(s=t,E)) THEN
    FOREACH (u,v)∈ E
```

```
      u' := NORMALFORM(u,R_{s=t});
      v' := NORMALFORM(v,R_{s=t});
      IF u≠u' OR v≠v' THEN
        E := E \ {u=v};
        CP:= INSERT(CP,u'=v');
      ENDIF
    ENDFOREACH
    FOREACH (u,v)∈ R
      v := NORMALFORM(v,R(E∪{s = t}));
      u' := NORMALFORM(u,R_{s=t});
      IF u'≠u THEN
        R := R\{(u, v)};
        CP := INSERT(CP,u'=v);
      ENDIF
    ENDFOREACH
    FOREACH (u,v) ∈ CPS(s=t,E ∪ R)
      CP := INSERT(CP,u=v);
    IF s>t THEN
      R := R ∪ {s→t};
    ELSE IF t>s THEN
      R := R ∪ {t→s};
    ELSE
      E:= E ∪ {s=t};
    ENDIF
  ENDIF
ENDWHILE
```

(E,R) represents the ground confluent endsystem

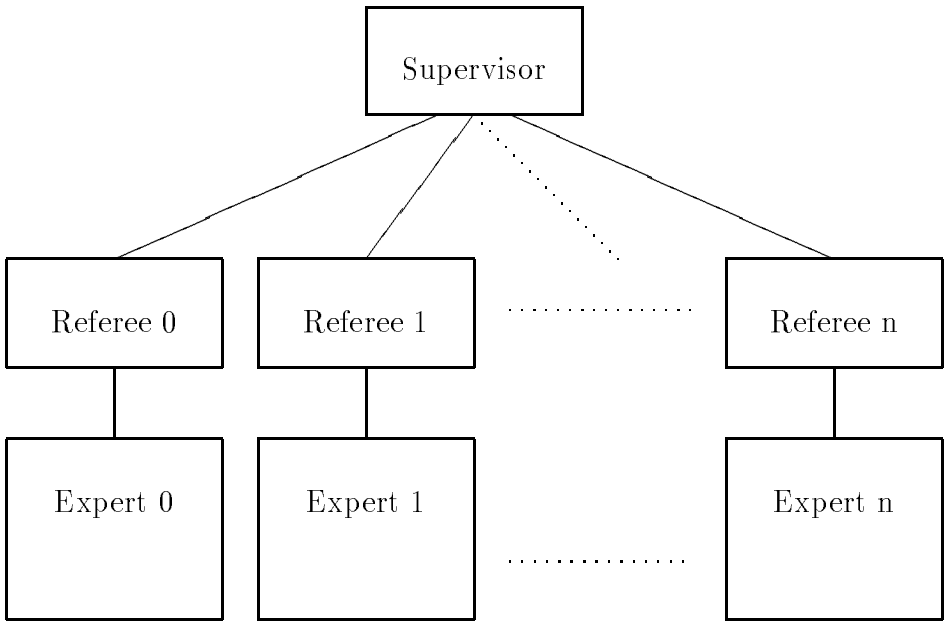

Figure 1: Basic structure of a team in the TEAMWORK method

2.3 Distributed deduction - The TEAMWORK method

The TEAMWORK method (see [De93] or [AD93]) is an approach to distribute theorem proving procedures. It has been inspired by human project teams and has been implemented quite successfully. A TEAMWORK based proof system models human project teams by use of multiple processes running on different processors.

A *team* consists of a single *supervisor* and a number of *experts*, each accompanied by a *referee* evaluating his work. Figure 1 shows the structure of a team. Usually each expert is working on a problem without communication with the other team members. Only at *team meetings* scheduled by the supervisor results are exchanged.

The supervisor is selecting the experts to work on a specific task, initially by judging their previous success on related problems, later by using the referees' evaluation of their performance in dealing with the problem at hand.

The referees are evaluating the work of the different experts. Their conclusions are used in selecting a new team and important results (from their respective experts) at the team meetings.

The experts are the members of the team working directly on the problem. In our case each of them is using an unfailing completion algorithm[1] as described in the above section. They differ only in the criteria used to select inferences. At the team meetings the system of the best expert is chosen as the base for further work. As only one system

[1]This is a slight simplification. We can also add *specialists* using other algorithms. One example is a *reduction specialist* trying to eliminate critical pairs before they are selected by the other experts.

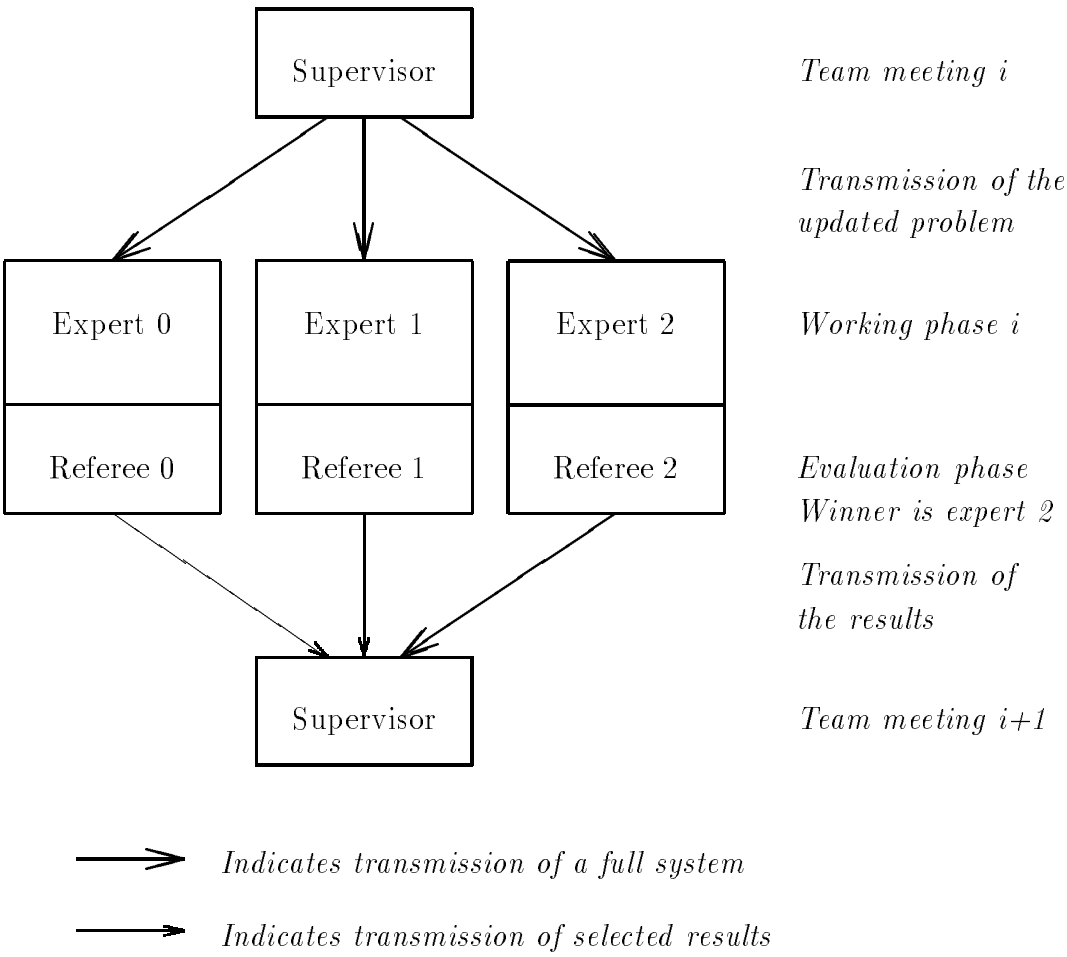

Figure 2: A cycle between two team meetings

survives completely, the experts are competing for the best result.

However, competition is only one aspect of the TEAMWORK method. The second important element is the *cooperation* between the experts. Cooperation is achieved by integrating outstanding results from inferior experts (as chosen by their referees) into the system of the best expert. This is done by the supervisor during the team meetings, before he presents a new and updated problem description to the experts for the next working phase (In our case of a completion based theorem prover the results – rules and equations – are handled as new critical pairs to be processed immediately).

It is important to note that most of the results generated by the inferior experts are dropped or *forgotten*. We believe that one of the reasons for the success of TEAMWORK is this feature of forgetting, because it alleviates the usual explosion of the search space.

A complete cycle between two team meetings is shown in figure 2. It can be described as follows:

- The supervisor selects a team of experts with their respective referees. Each of the experts is given the problem description. The supervisor schedules the next team meeting.
- The experts start working on the problem. If none of them can solve it until the scheduled meeting their progress is evaluated by the respective referees. The referees also selects outstanding single results and report these results and their overall evaluation of the experts progress to the supervisor.
- The supervisor chooses the best expert and integrates the results from the other experts into his system. He chooses a new team based on the success of the experts in the previous working phases. The experts are handed the updated problem description and the cycle starts again.

In [De93] and [AD93] the authors proved that a completion based prover using TEAM-WORK is complete if certain (weak) fairness criteria are fulfilled.

3 Representation of computer generated proofs

The first problem encountered when dealing with computer generated proofs based on inference mechanisms is the representation of the proof. In many cases the proof does not exist in a presentable form, but is represented only by the internal state, or, even worse, the dynamic processes of the program generating this proof. There are two basic ways to get information about the proof process. First, it is possible to build internal data structures representing the proof process. This is done in many proof system for first order predicate logic, which build refutation graphs containing enough information to reproduce the proof.

While this procedure is successful for predicate logic it is not really suitable for completion based rewriting systems. The great strength of rewriting systems is the fact that they can cut down on the information base using simplification rules and thereby keep the size of this data base relatively modest. They have to deal with large amounts of intermediate results (critical pairs are one example). These intermediate facts will usually be simplified extensively before they are used to generate new facts or can be proven trivial. Storing all the intermediate results and the simplifications done on them would nullify the main advantage of rewriting systems and seriously impair their power. For distributed proof systems this effect becomes still more pronounced, as it blows up on the communication between the different components - which already is a well known bottleneck even for systems refraining from proof analysis.

The second approach to get the desired information about the proof process is to generate a complete external listing of all the inferences and generated facts. This results in some problems too, but they can be overcome much easier. Section 6 deals with them for the special case of a distributed proof system (using the TEAMWORK method). Most of the problems encountered in sequential proof systems can be solved using the same techniques. This second approach has a number of advantages, namely:

- Changes in the proof system are kept to a minimum. No changes have to be made to the basic proof algorithm, all that is needed is the addition of routines to produce the relevant output. After this the proof system and the programs working on the proofs can be maintained separately. Our work with the DISCOUNT system has shown that these routines can be integrated into existing, complex proof systems quite easily (see section 6).

- Changing the underlying proof system will not affect the programs working on the generated proof. Building a new proof system which can produce the appropriate output does not significantly change the complexity of the system.

- As the information about the proof is stored on external media, the power of the proof system is not affected in any significant way. While the speed of the proof system may suffer, the class of theorems provable under fixed memory constraints is not usually affected[2].

[2]Lack of memory is at the moment the main restriction for finding proofs with the DISCOUNT system, while lack of time plays no role at all.

- Quite a few operations can be done on the proof listing. This operations are independent of the used calculus and can easily be extended to cover most inference based reasoning processes.
- As an added benefit the complete listings allow a very close and detailed analysis of the work done and problems encountered by the proof system. The knowledge gathered by this analysis can be (and has been) used to improve the heuristics of the prover and to get better insight into the inference mechanism.

Of course there are some problems associated with generating a complete listing of the proof process. They mostly stem from the enormous amount of data processed by a powerful proof system.

- Proof listings can become very large. The sheer amount of proof steps done can overwhelm most people and even programs used to analyze the proof. We cope with this problem by basing our analysis on the necessary steps only and by discarding steps that did not contribute to the proof process. Section 3.3 describes this solution in more detail.
- Producing the proof protocol is an output intensive task and can slow down the proof system significantly. This is particularly grave for distributed systems relying on cooperation at specific times. This problem is examined in more detail in section 6.1, where we also offer a solution for distributed proof systems based on the TEAMWORK method.

Obviously a consistent and general description of the proof process will be of much more use then a specialized format - some of the benefits above do not even apply if a less general description is used.

3.1 The language PCL

To achieve the goals stated above we developed a language for the description of completion based proofs. This language is PCL (*proof communication language*). We believe that it can be easily extended to cover most inference based reasoning processes.

PCL describes the proof process as a pure ASCII listing of single steps, representing the equations and rules (or, to be more general, the *facts*) generated during the completion (or reasoning) process. Connection between different steps are represented by *justifications*, giving the inference type, the facts used in the inference and additional information sufficient to allow a unique reproduction of the inference. Because PCL uses a rather intuitive description of the inferences a proof description in PCL can be read and analyzed by humans. On the other hand the complete description of every single inference makes it possible to use PCL protocols as the base for computer analysis. Some of our programs dealing with PCL will be described later.

As we stated before, a PCL protocol of a proof session is a list of single PCL steps.

```
<pcl-list> ::= <pcl-step>*;;
```

A single PCL step consists of a unique PCL identifier used to reference the step, a designator describing the type of the fact, the fact and finally the justification for the step.

```
<pcl-step> ::= <pcl-id>':'<step-type>':'<fact>':'<pcl-initexpr>;;
```

PCL identifiers are lists of positive integers. On the one hand this allows us to express technical information – for example the origin of the step in distributed environments or the heuristics used in generating the step – by using appropriate name spaces. On the other hand complex proof steps can be broken down into simpler inferences and inserted into the listing without requiring a global renaming. We will go into more depth about name spaces and distributed proof systems in section 6.2.

```
<pcl-id> ::= <int>['.'<int>]*;;

<int> ::= <digit>+;;

<digit> ::= '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9';;
```

PCL descriptions of proof sessions tend to become very long. To allow an efficient working with them we demand that the PCL identifiers are used in a monotonically ascending order. Identifier of later steps have to be larger (in the lexicographic extension over the usual ordering on natural numbers) than earlier steps.

The next element of a PCL step is the type designator. This designator is mainly used to mark the role of the fact in the proof process. Important steps can be marked as lemmata, different types of goals can be distinguished from normal facts and the proof system can mark steps with a special importance for the proof process.

```
<step-type> ::= 'tes-rule'|'tes-eqn'|'tes-goal'|'crit-goal'|
                'tes-intermed'|'tes-intermedgoal'|
                'crit-intermedgoal'|'tes-lemma'|'tes-final;;
```

The different step types are described below:

`tes-rule`	A rewriting rule generated during completion.
`tes-eqn`	An equation yielded by the completion process.
`tes-goal`	One of the hypothesises or a new goal generated (by rewriting) from a hypothesis. Goals of this type usually contain no variables, universally quantified variables have been replaced by skolem constants.

`crit-goal`	A *critical goal*, containing variables that are existentially quantified.
`tes-intermed`	A intermediate fact, either an equation or a rule, deemed particularly important by the proof system. This information can, for example, be used in the generation of lemmata. In the DISCOUNT system this type and the following two types designate the intermediate "good" results gained from unsuccessful experts.
`tes-intermedgoal`	
`crit-intermedgoal`	The same as `tes-goal` and `crit-goal`, respectively, but deemed especially important by the proof system.
`tes-lemma`	A rule or equation generated during completion and awarded a lemma status.
`tes-final`	The last step in a reasoning sequence, usually concluding the proof of at least one hypothesis. If a completion is desired, all facts of the final, ground convergent system are marked as `tes-final`.

Each PCL step represents a single fact. These can either be initial facts or facts generated by the reasoning process. In the case of a completion based proof system for equational reasoning the facts are either rules or equations. They either are a conclusion from the initial axioms or a (probably rewritten) goal. Equations consist of two terms connected by an equal sign, rules consist of two terms and an arrow.

```
<fact> ::= <rule>|<eqn>;;

<rule> ::= <term> '->' <term>;;

<eqn> ::= <term> '=' <term>;;

<term> ::= <ident>|<ident>'('<arglist>');;

<arglist> ::= <term>[','<term>]*;;

<ident> ::= [<letter>|<digit>]+;;

<letter> ::= 'a'|...|'z'|'A'|...|'Z';;
```

The final part of a PCL step is an expression describing the origin of the fact in the step. Facts are either initial axioms or goals without further justification or they have been derived during the inference process. While initial facts need no additional justification, derived facts usually need a description of the actual inference step.

```
<pcl-initexpr> ::= 'initial'|'hypothesis'|<pcl-expr>;;
```

In PCL annotation a step with the justification `initial` represents an axiom and a step with justification `hypothesis` represents a goal to be proved. For consistency reasons these expressions are assigned the fact in their PCL step as a value of the justification.

The justifications for the derived facts either describe the generation of a new fact or simply reference an already known fact.

```
<pcl-expr> ::= <quote-expr>|<orient-expr>|<cp-expr>
               <tes-red-expr>|<instance-expr>;;
```

The most obvious justification for a fact is a reference to an earlier step with the same fact. This is represented by a `<quote-expr>`, which simply consists of the identifier belonging to the earlier PCL step. The value associated with a `<quote-expr>` is the value of the referenced step.

```
<quote-expr> ::= <pcl-id>;;
```

During completion equations can be oriented using a reduction ordering. PCL describes this operation as a `<orient-expr>`. Arguments of an `<orient-expr>` are a `<pcl-expr>`, whose value is the equation to be oriented, and a direction. Directions can be either `u`, designating a rule in which the terms appear in the same order as in the original equation, or `x`, designating a rule in which these terms have been reversed.

```
<orient-expr> ::= 'orient('<pcl-expr>','['u'|'x']')';;
```

New equations are generated from critical pairs. In a critical pair inference one side of a rule or an equation is superposed into a subterm of a side of another term pair (or another instance of the same term pair). This is expressed as a `<cp-expr>`. Arguments are a `<pcl-expr>`, describing the term pair to be superposed into, a side and place descriptor designating the subterm to overlap, another `<pcl-expr>` describing the superposing term pair and finally a side descriptor marking the overlapping term of the second term pair. The value of a `<cp-expr>` is the critical pair resulting from the described superposition.

```
<cp-expr> ::= 'cp('<pcl-expr>','<place>','<pcl-expr>','<side>')';;
```

A place designator consists of two elements, a side descriptor selecting either the left (`L`) or the right (`R`) term in a term pair, followed by a list of integers describing the specific subterm in this term.

```
<place> ::= <side>[.<int>]*;;

<side> ::= 'L'|'R';;
```

Another way to generate new facts is the simplification of existing term pairs. This operation is described by a `<tes-red-expr>`. Like a `<cp-expr>` it takes four arguments, two PCL expressions describing term pairs, a place designator and a side designator. The value of a `<tes-red-expr>` is the fact created by the simplification in which the subterm of the first term pair described by the place designator is being matched with the selected side of the second term pair.

```
<tes-red-expr> ::= 'tes-red('<pcl-expr>','<place>',
                   '<pcl-expr>','<side>')';;
```

While the above expressions cover all inferences usually done by a completion procedure there is another mechanism heavily used by completion based *provers*. This is the instantiation of equations, and is often used to prove that a (skolemized) goal is a consequence of an already known fact. An `<instance-expr>` in PCL describes exactly this operation. Arguments are two PCL expressions, the first one describing a goal and the second one describing the term pair subsuming this goal.

```
<instance-expr> ::= 'instance('<pcl-expr>','<pcl-expr>')';;
```

Additional to the bare bones of a proof process other informations may be of use. To allow inserted text without syntactical structure PCL supports two different comment formats. One format is used to conveniently append comments to the end of an arbitrary line in the protocol file (please note that this includes empty lines). This kind of comments starts with a hash (`#`) and is terminated by the next `newline` character. The second kind of comments is included in C-style delimiters (`\*` and `*\`) and can be inserted between any two syntactical elements.

```
<line-comment> ::= '#'<comment>;;

<insert-comment> ::= '/*'<comment>'/*';;

<comment> ::= Any text without termination symbols;;
```

The following is a very short example of a PCL listing.

```
0 : tes-eqn : f(e(),x) = x : initial
    # e() is a left neutral for f
1 : tes-eqn : f(x,g(x)) = e() : initial
    # g(x) is a right inverse for f
2 : tes-eqn : f(f(x,y),z) = f(x,f(y,z)) : initial
    # f is associative
3 : tes-goal : g(f(e(),x)) = f(e(),g(x)) : hypothesis
    # Hypothesis
4 : tes-rule : f(e(),x) -> x : orient(0,u)
    # Orient the equation in step 0 without swapping the sides
5 : tes-goal : g(f(e(),x)) = g(x) : tes-red(3,R,4,L)
    # Simplify the right side of the fact from step 3 with the rule
    # from step 4
6 : tes-final : g(x) = g(x) : tes-red(5,L.1,4,L)
    # Simplify the subterm f(e(),x) from the left side of the fact
    # from step 5, using the rule from step 4
    # As the result is trivial the goal has been proved
```

3.2 Working with PCL listings

The aim of this section is to supply some basic functions and concepts to ease working with PCL steps and listings. These functions will be used in the description of more complex algorithms. The first group of functions allows access to PCL steps and their basic components.

Definition 3 : The functions `ID`, `TYPE`, `FACT`, `EXPR` and `STEP`
Let `<step>:=pcl-id>:<step-type>:<fact>:<pcl-initexpr>` be a PCL step. The projection functions `ID`, `TYPE`,`FACT` and `EXPR` are defined as follows:

- `ID(step) = <pcl-id>`
- `TYPE(step) = <step-type>`
- `FACT(step) = <fact>`
- `EXPR(step) = <pcl-initexpr>`

The function `STEP` is designed to reference a PCL step using a given identifier, therefore:

- `STEP(<pcl-id>) = <step>`

In most cases we use `ID(step)` instead of `step` when discussing PCL listings. We use the more explicit form in further definitions and in algorithms dealing with PCL listings, however.

The following functions return multisets of PCL steps (in the implementation these are represented as ordered lists). Multisets are basically finite sets which can contain the same element more then once. This can be represented by defining a multiset as a function over a set (the *base set*). This function returns the number of times the element is contained in the multiset for each element of the set. The usual set operations $\in$, $\cup$, $\cap$ and $\setminus$ are extended to multisets in the straightforward way. For more detailed definitions, see [Av91].

We need another operator related to $\setminus$. An additional concept used is the cardinality of a multiset.

Definition 4 : The $\setminus\!\setminus$ operator and the cardinality of multisets

- Let `A` and `B` be multisets over a set `M`. The $\setminus\!\setminus$ operator is defined as follows:

$$(\mathtt{A} \setminus\!\setminus \mathtt{B})(\mathtt{m}) = \begin{cases} 0 & \text{if } \mathtt{B}(\mathtt{m}) \neq 0 \\ \mathtt{A}(\mathtt{m}) & \text{otherwise} \end{cases} \quad \text{for all } \mathtt{m} \in \mathtt{M}$$

- The cardinality of a multiset `A` over `M` is

$$|\mathtt{A}| = \sum_{\mathtt{m} \in \mathtt{M}} \mathtt{A}(\mathtt{m})$$

Using this concepts we can now specify some functions dealing with the relationship of various PCL steps in a protocol. We call a step *parent* of another step if it is referenced in it's justification. The second step is then called a *child* of the first one.

Definition 5 : The functions PARENTS, PARENTS* and CHILDREN
Let step be a PCL step.

- PARENTS(step) is the multiset of the *direct* predecessors of step. Therefore it is the multiset of all steps cited in EXPR(step).

- PARENTS*(step) is the multiset of *all* predecessors of step. More formally, the following equation holds:

$$\texttt{PARENTS}^*(\texttt{step}) = \bigcup_{\texttt{s} \in \texttt{PARENTS(step)}} \texttt{PARENTS}^*(\texttt{s}) \cup \texttt{PARENTS}(\texttt{step})$$

- CHILDREN(step) is the multiset of all steps citing step:

$$\texttt{CHILDREN}(\texttt{step}) = \{\texttt{s} | \texttt{step} \in \texttt{PARENTS}(\texttt{s})\}$$

For partial proofs involving lemmata it is often useful to treat a lemma like an axiom by taking it as proven and ignoring its proof. To this end we define some new functions in a way analogous to the previous ones.

Definition 6 : The functions LPARENTS, LPARENTS* and LCHILDREN
Let step be a PCL step.

- LPARENTS(step) is the multiset of steps cited in EXPR(step) if step is *not* a lemma. It is the empty multiset otherwise.

$$\texttt{LPARENTS}(\texttt{step}) = \begin{cases} \{\} & \text{if } \texttt{TYPE}(\texttt{step}) = \texttt{tes-lemma} \\ \texttt{PARENTS}(\texttt{step}) & \text{otherwise} \end{cases}$$

- LPARENTS*(step) is the multiset of predecessors of step directly used in that proof. More formally:

$$\texttt{LPARENTS}^*(\texttt{step}) = \bigcup_{\texttt{s} \in \texttt{LPARENTS(step)}} \texttt{LPARENTS}^*(\texttt{s}) \cup \texttt{LPARENTS}(\texttt{step})$$

- LCHILDREN(step) is the multiset of all steps citing step if step is neither an initial fact nor a lemma. It is the empty multiset otherwise.

$$\texttt{LCHILDREN}(\texttt{step}) = \begin{cases} \{\} & \text{if } \texttt{TYPE}(\texttt{step}) = \texttt{tes-lemma} \\ \{\} & \text{if } \texttt{EXPR}(\texttt{step}) \in \{\texttt{initial}, \texttt{hypothesis}\} \\ \texttt{CHILDREN}(\texttt{step}) & \text{otherwise} \end{cases}$$

3.3 Extracting the proof

As has been stated above the main problem with an extensive step by step listing of the proof is the overwhelming amount of data produced. Of this data, however, only a tiny fraction refers to facts actually used in the proof. Most of the steps represent unsuccessful paths and dead ends in the search space traversed by the proof system. While this data allows an exact analysis of the strategy, the sheer mass of it tends to hide more than it reveals. Proofs of interesting theorems can easily produce some hundreds of thousands proof steps, of which only a very small fraction – a few permille for large examples – is necessary for the proof.

We noted that an analysis based only on the useful steps is much more helpful than wading through lots of misleading data. Even most programs working with PCL listings cannot cope with the immense amount of data. Therefore we devised a simple algorithm to extract the needed steps from the complete description.

These steps can be identified very easily in a post mortem analysis. Necessary facts are the ones reducing or subsuming a goal in the final proof, and, recursively, steps needed to generate these. They can be found by considering the final proof steps (usually marked as `tes-final` by the proof system) and, while scanning the listing backwards, discarding all steps not cited by steps already known as necessary.

We have implemented some variants of a simple algorithm dealing with PCL listings of various sizes. All use the same principles and differ only by technical considerations. For more detailed informations see [Sch93].

We will now present some results from the extraction process to support our statement about the relation between used and unused steps. As even protocols for simple examples are to long to be reproduced here we limit ourselves to a table with numerical data. Even so the large differences should become obvious. Table 1 shows the numbers of steps in some PCL listings. Please note that the first 12 examples are proof problems while the last 3 are completions. [Sch93] discusses a concrete example in more detail. A number of problems also appears in table 2 on page 60, these problems are commented there.

Two trends become obvious in the data. First, the larger the example the smaller the percentage of used steps. This of course is a result from the larger search space the prover has to handle for more complex problems. Secondly, completions use relatively more steps than proof problems. This is easily explained by taking into account that in a completion each fact in the final system can be considered as a separate theorem, while in the proof problems above only one result is shown.

Problem	**Complete**	**Extracted**	**Comments**
Lusk2	54	16	Group axioms imply $(x^{-1})^{-1} = x$. This is a very simple problem from [LO82].
SelfInverse	1703	69	In a ring with $x^2 = x$ every element is self inverse, that is $x^{-1} = x$. This is related to the next problem. See appendix B.4 for details.
Lusk3	5009	83	In a ring with $x^2 = x$ the multiplicative operation is Abelian. See [LO82] and appendix B.1 for a discussion.
Cooperation	98532	64	See table 2.
DeMorgan	238706	151	ditto
Luka1	145078	22	ditto
Luka2	85934	58	ditto
Luka3	322001	79	ditto
Lusk4	13420	95	In a group with $x^3 = e$ the equation $h(h(x, y), y) = e$ holds for the commutator h. See [LO82].
Lusk5	46477	45	See table 2.
Lusk6	387273	190	ditto
Lattice3	485010	139	ditto
Group	414	44	Completion of the group axioms. This problem was used as an example for the completion procedure in [KB70].
Fibgroup	1610	137	Completion of a cyclic group. See appendix B.4.
Z22	18254	711	See table 2.

Table 1: Numbers of steps in PCL listings

4 Structuring proof listings

Even relatively simple proofs can become overwhelmingly complex if they are presented in an unstructured way. While this statement holds true for every reasonable calculus the situation for completion based proofs is particularly grave, as the proof is usually found in tiny, often unrelated fragments. To make such proofs more accessible they need to be segmented into a number of subproofs. This is done by using selected sub-results as lemmata and, using them, building a hierarchical proof structure.

In our approach to this issue the basic structure of the proof as delivered by the proof system is unchanged, the lemmata only serve to break it into more manageable parts. In particular, steps are usually considered one after the other and with earlier lemmata taken into account. As a result the reasoning process of the proof system can still be studied from the structured and possibly transformed proof. This is a marked difference from the (superficial) approach of the authors in [LP90]. In this paper the authors try to transfer Lingenfelders results (see [Li90]) from restructuring proofs in first order predicate logic to equational reasoning. However, concrete suggestions are lacking from the paper, and the straightforward transfer to a completely different calculus is not very convincing in our opinion.

Structuring the proof "as is" by selecting certain steps as lemmata has some easily identified merits:

- The proof is broken up into smaller, more easily accessible parts.
- A lemma needs to be proofed just once. If it is then used more then once the size of the overall proof can be reduced significantly. This does not apply to the PCL proof, however, because a completion based prover reuses each fact without generating a new proof. It does apply to the more readable formats generally used by human mathematicians, which we try to emulate in section 5.
- By searching the proof for viable lemmata a lot of insight can be won regarding the processes by which the proof has been generated. This knowledge can be used to improve the heuristics used in the prover, probably by a guided search for viable lemmata.

The problem is, of course, to identify suitable candidates for lemmata. Unfortunately, even humans usually do not present exact reasons for their decisions in selecting lemmata. More often than not a lemma is chosen because it is intuitively "important" or "aesthetically satisfying". Our aim in this section is to find some objective criteria applicable to a proof generated by an automatic proof system. As above we restrict ourself to completion based equational proofs. As lemmata are used to structure large proofs the criteria developed here usually have to take into account rather large parts of the proof tree. We therefore give only one example for the most complex criterion. See [Sch93] to find examples for the other criteria.

It should be noted that structuring the proof is in this section viewed purely as a post mortem process. This is not fully in accordance with the behavior of humans, who

usually use lemmata in both proof presentation and proof generation. We will later show that some of the intermediate results generated by the teamwork method can aid in the generation of lemmata. These results can be considered as lemmata found during the proof process itself.

4.1 Partitioning the proof – top-down vs. bottom-up

We considered two different ways to partition a proof. One approach is a recursive algorithm, starting at the goal and working in a top-down partitioning scheme. While this may seem a natural way we found that it is not really possible for equational proofs. The proofs typically possess a large number of steps referenced from different parts of the proof. Selecting a lemma because of its position in one part of the proof may invalidate conditions used to select lemmata in other parts. This is due to the fact that it can be inserted previous to existing ones, thus rendering them unnecessary. Because of the high degree of interconnectedness in the proof we were forced to choose a constructive, bottom-up approach.

In this second approach the proof listing is considered one step after the other. For each step under consideration all predecessors have already been processed and all previous lemmata are known. A newly introduced lemma cannot influence subproofs of previous lemma, and no old lemma is invalidated.

The resulting algorithm can be summed up as follows: Starting at the beginning of a proof listing, each step is evaluated, using one or more of the criteria suggested below. In this evaluation previous lemmata are treated just like axioms. If a step is suitable, its status is set to *lemma*. The process is then repeated for subsequent steps. Please note that a lemma in a PCL listing is just an ordinary step with a special status. The subproof for a lemma consists of the PCL steps describing its derivation. Only when transforming the proof to human readable form (see section 5) this subproof is separated from the main proof (but will still use previous lemmata).

4.2 Frequently used steps

A proof step that is referenced frequently in the proof is an obvious candidate for a lemma. The fact that it is used relatively often indicates its importance for the whole proof. Additionally the total proof is reduced in size if this step is only proved once and not on every occurrence.

Formalizing this criteria a step becomes a lemma if

$$|\texttt{CHILDREN(step)}| \geq \texttt{MINUSED}$$

holds. Here `MINUSED` is a constant designating the minimum number of times a step has to be referenced in order to satisfy this criterion.

4.3 Important intermediate results

As the whole point in finding lemmata is isolating "important" steps the above title may seem a little bit preposterous. However, in this section we are addressing one objective measure for the importance of a proof step, namely how many applications of initial axioms or lemmata it represents. This can be easily calculated as the product of the number of applications of axioms or lemmata necessary to proof the step (the *length of its proof chain*) and the number of references to this step in the remaining proof.

Definition 7 : The function `CHAINLEN`
Let `step` be a PCL step. `CHAINLEN(step)` calculates the number of applications of axioms or lemmata used to proof the fact from `step`. Therefore

$$\texttt{CHAINLEN(step)} = \left| \left\{ \texttt{s} \in \texttt{LPARENTS}^*\texttt{(step)} \;\middle|\; \begin{array}{l} \texttt{TYPE(s)} = \texttt{tes-lemma or} \\ \texttt{EXPR(s)} \in \{\texttt{initial},\texttt{hypothesis}\} \end{array} \right\} \right|$$

`CHAINLEN` can be implemented quite efficiently by recursion over the arguments of `EXPR(step)`. Given the above definition, a step becomes a lemma by this criterion if

$$|\texttt{CHILDREN(step)}| \times \texttt{CHAINLEN(step)} \geq \texttt{MINWEIGHT}$$

The constant `MINWEIGHT` defines the minimum weight a step has to achieve in order to become a lemma.

4.4 Isolated proof segments

The results of a (relatively) isolated subproof are good candidates for lemmata. They usually represent important implications from a subset of the axioms and tend to partition the proof into segments connected by a common subject. This criterion uses a rather large part of the proof listing and relies on the overall structure of the proof.

Note that in [Li90] the author suggests using isolated subgraphs as a criterion to restructure refutation based predicate logic proofs. In [LP90] the authors suggest to transfer this principle to equational proofs. They do however use *Equation Solution Graphs* as the base for their ideas, while our approach works directly with a step-by-step (PCL) listing of the proof.

Our basic measure for the degree of isolation is the number of steps in the subproof for a potential lemma that are cited outside this subproof. In order to get a more exact impression of the weight of this citations we calculate an "importance" analog to 4.3 for each of them. This value is compared to the weight for the potential lemma to reach a decision.

For a more detailed discussion we need the following function:

Definition 8 : The function `EXITS`
`EXITS(step)` is the multiset of all steps outside the (sub-)proof for `step` but referencing

a step in this proof. Therefore

$$\texttt{EXITS}(\texttt{step}) = \left(\bigcup_{\texttt{s} \in \texttt{LPARENTS}^*(\texttt{step})} \texttt{LCHILDREN}(\texttt{s}) \right) \setminus\setminus (\texttt{LPARENTS}^*(\texttt{step}) \cup \{\texttt{step}\})$$

This function can be easily implemented recursively, too. Given this function the required condition for a lemma can be written as follows:

$$\left(\sum_{\texttt{s} \in \texttt{LPARENTS}^*(\texttt{step})} |\texttt{CHAINLEN}(\texttt{s})| \times |\texttt{CHILDREN}(\texttt{s}) \cap \texttt{EXITS}(\texttt{step})| \right) \times \texttt{WEIGHTFACTOR} + \texttt{OFFSET}$$
$$< |\texttt{CHILDREN}(\texttt{step})| \times \texttt{CHAINLEN}(\texttt{step})$$

Please note that $\texttt{CHILDREN}(\texttt{s}) \cap \texttt{EXITS}(\texttt{step})$ is the multiset of all descendents of `s` that are *not* part of the proof for `step`. `WEIGHTFACTOR` and `OFFSET` are two constants determining the requirements for a lemma with this criterion.

As this criterion is rather more complex then the preceding ones we will illustrate it with the following example:

Example: Consider the following excerpt from a longer PCL listing:

```
...
  1 : tes-eqn : j(0(),x) = x : initial
...
  4 : tes-eqn : j(x,g(x)) = 0() : initial
...
  6 : tes-eqn : j(j(x,y),z) = j(x,j(y,z)) : initial
...
 12 : tes-rule : j(0(),x) -> x : orient(1,u)
...
 20 : tes-rule : j(x,g(x)) -> 0() : orient(4,u)
 45 : tes-rule : j(j(x,y),z) -> j(x,j(y,z)) : orient(6,u)
 53 : tes-eqn : j(x,j(g(x),y)) = j(0(),y) : cp(45,L.1,20,L)
 54 : tes-eqn : j(x,j(g(x),y)) = y : tes-red(53,R,12,L)
 59 : tes-eqn : j(x,j(y,z)) = j(y,j(z,x)) : cp(5,L,45,L)
 65 : tes-rule : j(x,j(g(x),y)) -> y : orient(54,u)
 71 : tes-eqn : g(g(x)) = j(x,0()) : cp(65,L.2,20,L)
...
 87 : tes-eqn : x = j(y,j(x,g(y))) : cp(65,L.2,5,L)
...
 93 : tes-eqn : x = j(g(y),j(y,x)) : cp(65,L.2.1,89,L)
...
2721 : tes-eqn : f(g(x),0()) = j(x,g(x)) : cp(65,L.2,2707,L)
2722 : tes-eqn : f(g(x),0()) = 0() : tes-red(2721,R,20,L)
...
```

```
2883 : tes-eqn : j(g(x),f(g(x),x)) = f(g(x),0()) : cp(2859,L.2,20,L)
 ...
2961 : tes-eqn : f(g(x),x) = j(x,0()) : cp(65,L.2,2944,L)
 ...
3632 : tes-eqn : j(x,f(g(x),x)) = f(0(),x) : cp(3610,L.1,20,L)
 ...
3986 : tes-eqn : g(x) = j(x,0()) : tes-red(3985,R.2,20,L)
 ...
4132 : tes-eqn : j(x,j(x,y)) = y : tes-red(65,L.2.1,4117,L)
 ...
4267 : tes-eqn : j(x,j(x,j(y,f(y,x)))) = f(j(x,y),y) :
                  tes-red(4266,L.2,45,L)
 ...
```

The graph expressing the relations in this listing is drawn in figure 3. Note that step 65 is being referenced 7 times. The length of the proof chain for this step is 3, and only a few references to steps with a trivial proof chain (oriented axioms in this case) are being made outside the subproof for step 65. Therefore step 65 is relatively isolated and a viable lemma.

4.5 Syntactical criteria

Humans rather seldom view the proof as a tree or graph. They are much more concerned with the facts themselves, and less concerned with the structure of the reasoning process. The proof is perceived as a (mostly linear) sequence of results, and results are often selected as lemmata for their own appearance. To emulate this behaviour we developed some simple criteria comparing the size of the two terms in a fact. We mainly aimed at "small" terms for the sides of the potential lemma, because they represent more general concepts. However, as there is only little effort involved in extending this principles, we tried some variations. Be forewarned that this criteria have proven to be very weak compared to the more global ones.

As a measure for the size of a term we use the number of variables and function symbols contained in it.

Definition 9 : The size of a term
The size $|\mathtt{t}|$ of a term $\mathtt{t}$ is recursively defined by the following equation:

$$|\mathtt{t}| = \begin{cases} 1 & \text{falls } \mathtt{t} \in \mathtt{V} \\ 1 + \sum_{1 \le \mathtt{i} \le \mathtt{n}} |\mathtt{t_i}| & \text{falls } \mathtt{t} \equiv \mathtt{f}(\mathtt{t_1}, \ldots, \mathtt{t_n}) \end{cases}$$

Given this information we can easily formulate two criteria checking for either a small size of the larger term or a small average size of the two terms in a fact. Let `step` be

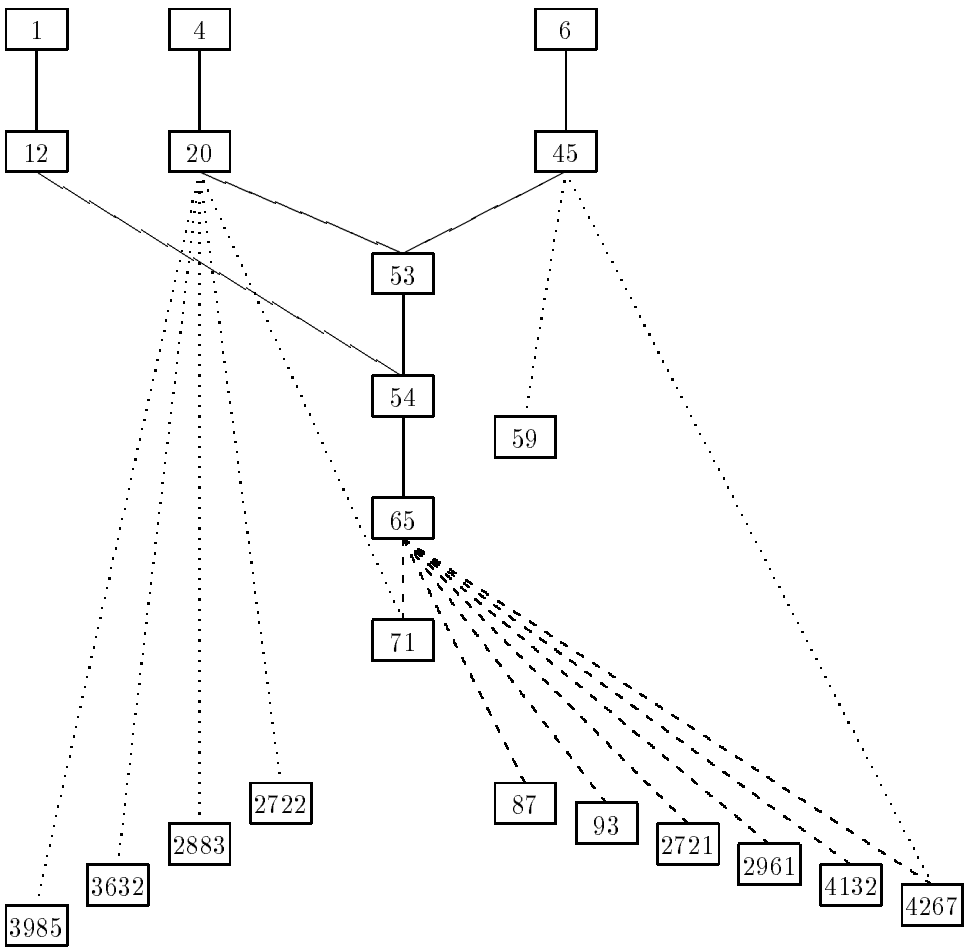

Figure 3: Proof graph corresponding to the example on page 30

a PCL step, $\texttt{FACT(step)} = \texttt{t}_1\texttt{=t}_2$ or $\texttt{FACT(step)} = \texttt{t}_1 \rightarrow \texttt{t}_2$. The criteria for step then become either

$$\texttt{max}(|\texttt{s}|, |\texttt{t}|) \leq \texttt{MAXSIZE}$$

for the maximum size or

$$\frac{|\texttt{t}_1| + |\texttt{t}_2|}{2} \leq \texttt{AVERAGESIZE}$$

for the average size.

Now remember that we are at the moment working on proofs generated by a rewriting system. Important facts are then rules which allow "strong" simplifications. However, most reduction orderings do not support the concept of a measure of the difference between two terms, and PCL listings do not contain informations about orderings, anyway. As a simple and at least heuristically sound measure we utilize the difference between the terms sizes. If the difference between the two sides of a rule is large it will alter a reduced term significantly. Therefore such a rule can probably be an important step, or potential lemma. More exactly this requirement becomes

$$\frac{\texttt{max}(|\texttt{s}|, |\texttt{t}|)}{\texttt{min}(|\texttt{s}|, |\texttt{t}|)} \geq \texttt{MINFAK}$$

In some equations valid in many domains the size of the two terms is exactly the same or differs very little. Examples are the axioms of commutativity and associativity. Steps with equal term sizes may be good lemmata, two. Stating this more exactly yields

$$\frac{\texttt{max}(|\texttt{s}|, |\texttt{t}|)}{\texttt{min}(|\texttt{s}|, |\texttt{t}|)} \leq \texttt{MAXFAK}$$

Obviously there are much more possibilities for a purely syntactical analysis of the proof listing. The success or better lack of success we had with this criteria did not encourage more research. More about this can be found in section 4.12, which compares the merits and weaknesses of the various criteria.

4.6 Using outside knowledge

Another way to select lemmata using only data from a single step is to apply outside information. Many problems come from well known domains, and important results from these domains can be used as lemmata in more special domains. Examples are equations of the form $\texttt{f(x,f(y,z))=f(f(x,y),z)}$ (describing associativity) or $\texttt{f(x,y)=f(y,x)}$ (describing commutativity).

This criterion is simply applied by comparing the facts of steps in a protocol with a data base of known good lemmata. A step becomes a lemma if it matches one of the patterns in this data base.

A human selecting the lemmata can be viewed as a special case of this criterion, employing his "biological data base".

4.7 Analysing the applied inference rules

In this section we again utilize the knowledge about the inference processes used to generate a PCL listing. Currently we only deal with protocols from completion based proof systems. These systems use three major inference mechanisms: orientation, simplification and critical pair generation. While all three mechanisms are useful in finding a proof they obviously have different impacts on the data base. An orientation is a purely technical process, a simplification is actually changing the data base and a new critical pair is a real addition to the data base of the proof system.

Using this knowledge we can assign different values to the inference mechanisms. The sum for all the inferences necessary to generate a given step can then, as above, be used to base the lemma decision on.

Definition 10 : The function `IWEIGHT`
The function `IWEIGHT` can be applied to PCL steps and PCL expressions and assigns them a weight representing the inference steps used to generate the fact of the step or the *value* (see 3.1) of the expression. Let `step` be a PCL step and let `INITW`, `HYPOW`, `QUOTEW`, `ORIENTW`, `CPW`, `REDW` and `INSW` be constants.

- $\texttt{IWEIGHT(step)} = \begin{cases} \texttt{INITW} & \text{if } \texttt{TYPE(step)} = \texttt{tes-lemma} \\ \texttt{IWEIGHT(EXPR(step))} & \text{otherwise} \end{cases}$
- $\texttt{IWEIGHT(expr)} =$

$$\begin{cases} \texttt{INITW} & \text{if } \texttt{expr} = \texttt{initial} \\ \texttt{HYPOW} & \text{if } \texttt{expr} = \texttt{hypothesis} \\ \texttt{QUOTEW} + \texttt{IWEIGHT(STEP(<id>))} & \text{if } \texttt{expr} = \texttt{<id>} \\ \texttt{ORIENTW} + \texttt{IWEIGHT}(\texttt{expr}_1) & \text{if } \texttt{expr} = \texttt{orient}(\texttt{expr}_1, \texttt{d}) \\ \texttt{CPW} + \texttt{IWEIGHT}(\texttt{expr}_1) + \texttt{IWEIGHT}(\texttt{expr}_2) & \text{if } \texttt{expr} = \texttt{cp}(\texttt{expr}_1, \texttt{p}, \texttt{expr}_2, \texttt{s}) \\ \texttt{REDW} + \texttt{IWEIGHT}(\texttt{expr}_1) + \texttt{IWEIGHT}(\texttt{expr}_2) & \text{if } \texttt{expr} = \texttt{tes-red}(\texttt{expr}_1, \texttt{p}, \texttt{expr}_2, \texttt{s}) \\ \texttt{INSW} + \texttt{IWEIGHT}(\texttt{expr}_1) + \texttt{IWEIGHT}(\texttt{expr}_2) & \text{if } \texttt{expr} = \texttt{instance}(\texttt{expr}_1, \texttt{expr}_2) \end{cases}$$

To decide wether a step should become a lemma we only need to compare the weight calculated using `IWEIGHT` with a predetermined boundary. Therefore our criterion becomes

$$\texttt{IWEIGHT(step)} \geq \texttt{MINWEIGHT}$$

with the boundary constant `MINWEIGHT`.

4.8 Sectioning long proofs

The purpose of using lemmata is of course sectioning a long proof into smaller, more easily digestable proofs by selecting *meaningful* intermediate results and separating their proofs. However, sometimes no important fact can be found - either because our criteria are to weak or simply because no outstanding result is generated in a large part of the proof. In this case the proof should still be broken up into smaller segments for easier understanding. As there are no particulary suitable steps we select a lemma purely on the length of its proof chain.

This length can be easily computed using the function CHAINLEN introduced in section 4.3. A step becomes a lemma if the relation

$$\texttt{CHAINLEN}(\texttt{step}) > \texttt{MAXLEN}$$

holds for step. Please keep in mind that for a fact generated by a reduction or a critical pair inference the lenght of the new proof chain is the sum of the lenghts of the two old chains. Therefore chains up to 2 times MAXLENGHT can be generated.

Obviously this simple implementation does not reach an optimal division. It does reach relativly good results, however. A perfectly regular division of the proof is generally impossible to achieve and even small improvements lead to much more complex algorithms. As the importance of this criterion is rather small we consider the simple solution sufficent in this context.

4.9 "What to avoid" - not every candidate may be suitable

Up to now we only considered conditions *for* selecting a step as a lemma. However, some steps, albeit they fulfill one or more of the above criteria, are not really suitable as lemmata. In this section we try to develop some *necessary* conditions for lemmta. A step not satisfying these criteria cannot become a lemma even if it satisfies other criteria.

Apart from certain limitations in our current implementation of PCL we found the criteria from sections 4.2 and 4.8 to be particulary useful in designing these necessary conditions for lemmata. Of course a lemma should be used at least once in a proof. This condition is guaranteed if lemmata are generated from extracted listings or, as in our implementation, a proof graph is build that contains only used steps. In many cases it can be useful to demand that a lemma is used more then once in the proof. This condition can be stated exactly as in section 4.2:

$$|\texttt{CHILDREN}(\texttt{step})| \geq \texttt{MINUSED}$$

It should be kept in mind that this condition now describes a *necessary* condition, while above it described a *sufficient* condition.

Of course demanding a certain minimum of *new* information in a lemma is reasonable. A step only citing an already known fact does not deserve lemma status. The length of a step's proof chain represents the number of applications of other facts necessary to generate it. It is therefore a reasonable measure of the amount of new knowledge represented by the step. Stating this as a formal condition yields

$$\texttt{CHAINLEN}(\texttt{step}) \geq \texttt{MINLEN}$$

A more specialized criterion can be used to enhance the performance of the criterion from section 4.4. If a step with an isolated proof has but a single successor generated within the same subproof as the step this successor is usually the better lemma. Checking this condition for steps with only a single successor can be done by comparing the

exits from the step and its sucessor. Let `step` be a potential lemma according to the criterion from section 4.4 and let `succ` be its single successor. `step` should only become a lemma if either another criterion suggests this or if

$$\texttt{EXITS(step)} \neq \texttt{EXITS(succ)}$$

Note that this criterion is not perfect because not the whole sproof graph of `succ` has been considered during the analysis for `step`.

4.10 Combinations of different criteria

Many of the criteria mentioned so far are able to structure a PCL proof without further asistance. However, a combination of several criteria usually achieves a much better result then any single one. For example, consider the lemmata resulting from the isolated subgraph criterion (section 4.4). Every single lemma is usally of superior quality, however it is impossible to structure the complete proof using only this criterion, because the lemmata are generated at more or less random intervalls, giving lemmata with very long or very short subproofs. It is therefore not very well suited as a stand alone criterion, but can drastically improve the sectioning if used as a supplementary criterion. Similar effects can be noted for other criteria.

There are several different ways to combine multiple criteria. They will be discussed in the following paragraphs. Note that some of them need a completly separate treatment of the negative criteria form section 4.9.

(1) Every step in the PCL listing is tested exactly once for each of the used criteria. It becomes a lemma if one of the conditions is fulfilled. This simple approach has several advantages:

- The implementation can be kept very simple.
- When considering a step all previous steps have been analysed and the selected lemmata among them can be taken into account.
- The resulting implementation is very efficient as every step has to be considered only once.

There are some disadvantages, too:

- Each criterion is given the same weight. In particular, a rather weak criterion can produce lemmata hindering a better structuring by other criteria.
- Steps conforming to only one criterion are handled exactly as steps fulfilling two ore more conditions.
- The criteria from section 4.9 can only be incorporated as necessary conditions for a lemma. A weighted decision is not possible.

(2) The criteria are ordered according to (perceived) quality. The complete PCL listing is then analysed using one of this criteria at a time. As above only one condition needs to be fulfilled for a step to become a lemma. The advantages:

- Criteria producing "better" lemmata have the first chance to evaluate the listing. It is therefore less likely that an inferior lemma is generated instead of a better one.
- The implementation is still very managable.

However, some new disadvantages appear together with the advantages:

- The lemmata are generally *not* generated in their order of appearance in the PCL listing. As the criteria only take earlier lemmata into account a newly created lemma can make an old one superfluous.
- The PCL listing has to be traversed once for every single criterion, the cost for the calculation rises by a small factor. However, this has not caused any problems for the examples analysed so far.
- As above the criteria from section 4.9 can only be incorporated as necessary conditions

(3) The different criteria are weighted numerically, and the sum over the fulfilled conditons is compared to a limit. This combines most of the advantages from (1) and (2) above.

- An implementation can still be kept quite simple.
- At the evaluation of a step all earlier lemmata are already known and can be taken into account.
- The efficiency is rather high as the listing has to be analyzed but once.
- For the first time the (negative) criteria from 4.9 can be incorporated in a weighted way. Even steps that, according to one of these criteria, are not suitable can become lemmata if a lot of other conditions are fulfilled.
- Based on this approach other criteria searching for a globaly optimal sectioning can be constructed. They would be extremly costly to evaluate, though.

Of course there still remain some disadvantages:

- Analogous to (1) the best lemmata are generally not guaranteed to be found.
- As the field is a rather new one we do not yet know enough about the quality of the different criteria to assign meaningful weights to the different ones. This disadvantage will hopefully vanish with time.

(4) When analyzing the different criteria for lemmata it can be noted that the decission is usually based on the comparison of numerical values. This can be used to switch from the purely binary approach described so far (each criteria either recommends a step or it does not) to a more subtle procedure. In this each criteria evaluates a step on a continuous scale and gives its recomendation in the form of a single numerical value. The (weighted) sum of this values is then – as above – compared to a limit.

- The concept uses the data collected for the criteria in a more flexible way, probably achieving a more just verdict.
- The negative criteria can be smoothly integrated. In particular the first two conditions from section 4.9 can be completly merged with the criteria from 4.2 and 4.8 respectively.

Two important disadvantages have hindered an implementation so far:

- The cost for an implementation based on our current programs is comparatively high.
- In a much stronger sense than above our limited knowledge about the behaviour of the different criteria makes it difficult to map the data onto a continuous scale. However, we do hope to overcome this problem in the future.

(5) The last and potentially most powerful possibility is to constuct a complex language supporting arbitrary expressions and access to the functions described in the various criteria and possibly to the PCL listing itself. This has an immediately obvious advatage:

- The user enjoys maximum flexiblility and power in designing his own criteria.

There are however a number of disadvantages to consider, too.

- An implementation is comparatively very costly in terms of manpower.
- The efficency of an implementation will suffer because of the need to interpret an additional language.
- The user has to invest a lot of effort if he wants to gain any benefits from this approach.

Up to now we only implemented the methods (1) and (2). The algorithm used will be presented in the next section. We found that most proofs could be structured quite well using this algorithm. While there is still interest in the approaches labeled (3) and (4) the last one (labeld (5)) will probably never be realised. For the few cases where this flexibility is needed the criteria can probably be coded much easier directly into the C sources of the program.

4.11 An algorithm for the structuring of PCL listings

This section describes the basic algorithm used by our program for the structuring of lemmata in PCL listings. It uses all the criteria presented so far. Additionally several of these critera can be combined using the methods (1) and (2) from section 4.10.

Input and output are (extracted) PCL listings. The output listing contains all the used steps from the input listings. Recognized lemmata will be of the type `tes-lemma`.

Input:	`in`	A list of PCL steps.
	`criteria`	A list of (boolean) functions for the evaluation of a PCL step (see below).
	`iterate`	TRUE, if the criteria to be used should be combined according to the second approach from 4.10, FALSE otherwise.
Output:	`out`	A list of PCL steps with the lemmata markes as `tes-lemma`.
Variables:	`store`	A list for the intermediate storage of the PCL listing.
	`step`	The PCL step considered at the moment.
	`IsLemma(step)`	The momentarily active evaluation function.
Functions:	`NOTEMPTY(list)`	FALSE, if `list` is empty, TRUE if not.
	`FIRST(list)`	First entry in `list`.
	`EXCEPTFIRST(list)`	`list` without its first entry.
	`TYPE(step)`	Type of `step` (compare 3.2).
	`APPEND(list,step)`	List generated by appending `step` as the last element to `list`.
	`IsOftenLemma(step)`	Realizes the search for frequently used steps (section 4.2).
	`IsImportantLemma(step)`	Realizes the search for important steps from section 4.3.
	`IsTreeLemma(step)`	Implements the search for isolated segments of the proof (see 4.4). This may take into account the third criterion from section 4.9, although our current implementation does not yet check this.
	`IsSyntaxLemma(step)`	Realizes the different syntactical criteria from section 4.5.
	`IsKnownLemma(step)`	Checks the step by comparing its fact to known important results from the domain.
	`IsCompletionLemma(step)`	Evaluates lemmata using weighted inference steps (see 4.7).

`IsPartLemma(step)`	Evaluates steps according to the criterion from section 4.8 (breaking up long proof chains).
`IsNoLemma(step)`	Checks for the first two negative criteria from section 4.9.

Remark: `IsNoLemma(step)` = TRUE holds if the step is *not* acceptable as a lemma, FALSE otherwise. The other evaluation functions are TRUE if the step should become a lemma, FALSE otherwise. The different constants needed for the evaluation of the criteria are not listed as separate parameters. Please note that by suitable selection of these constants it can be guaranteed that `IsNoLemma(step)` = FALSE does always hold.

```
IF iterate = TRUE THEN
  store := in;
  WHILE NOTEMPTY(criteria)
    Islemma := FIRST(criteria);
    criteria := EXCEPTFIRST(criteria);
    WHILE NOTEMPTY(store)
      step := FIRST(store);
      store := EXCEPTFIRST(store);
      IF NOT(IsNoLemma(step)) THEN
        IF IsLemma(step) THEN TYPE(step) := tes-lemma;
      out := APPEND(out,step);
    ENDWHILE
    store := out;
  ENDWHILE
ELSE
  WHILE NOTEMPTY(in)
    step := FIRST(in);
    in := EXCEPTFIRST(in);
    IF NOT(IsNoLemma(step)) THEN
      FOREACH IsLemma ∈ criteria
        IF IsLemma(step) THEN TYPE(step) := tes-lemma;
    ENDIF
    out := APPEND(out,step);
  ENDWHILE
ENDIF
```

4.12 Evaluating the different criteria for lemma generation

The lack of objective criteria for lemma selection hinders not only the generation but also the evaluation of structured proofs. Any assessment therefore has to be highly subjective. As PCL listings are not particulary suitable for human interpretation (they

work well for analyzing the proof process, but not for proof presentation), we used proofs transformed into the more friendly format presented in chapter 5.

Apart from the purely subjective impression we use two more objective (albeit quite weak) criteria: The length of the proof chains and the size of the terms in the lemmata. We think proof chains of similar length and lemmata containing small (more general) terms are desirable.

The overall result is quite promising. Comparisons with proof listings structured by humans have shown that our automatic algorithms usually achieve uniformly better results. This mainly stems from the difficulty of considering the global effects of a inserted lemma. However, the automatic proofs does not quite reach the quality of proofs found *and* structured by humans. We also found that these proofs can still be improved by a human working not with a plain listing but with a proof prestructured.

We will now discuss the problems and merits of the different criteria in more detail.

- Frequently used steps (section 4.2) can be quite good individual lemmata. However, while they often contain small, meaningful terms, they do not structure the proof very well. Proof chains vary widely in length and particulary towards the end of the proof grow much to large for easy comprehension. However, if used in conjunction with additional criteria limiting the maximum size of the subproofs, good results can be obtained.

- Searching for "important" steps as detailed in section 4.3 is one of the strongest single criteria developed so far. The lemmata are often intuitively appealing and the length of the proof chains, while quite variable, never becomes overwhelming. In most cases lemmata obtained using only this criterion are sufficient to structure the proof in an agreeable way.

- Using the criterion from section 4.4 yields very good lemmata. However, while the structure of the proof is well reflected and the lemmata are quite intuitive, the subproofs are of very different length. Lemmata are clustered together, with very short subproofs within the clusters and very large subproofs outside of them. Our best results are achieved if this criterion is combined with another criterion limiting the maximum proof size.

- Syntactical criteria based purely on the size of the terms have proven to be very weak. While some of the resulting lemmata look quite reasonable they usually have no special role in the proof. The frequency of the lemmata is extremely random, with very large proof chains towards the end of the proof. Even in conjunction with other criteria no acceptable results have been achieved. All in all these syntactical criteria are only useful in very special cases and not suited for more general purposes.

- Employing outside knowledge to identify good lemmata results in quite intuitive and appealing lemmata. However, as the criterion does not take the global structure of the proof into account, these lemmata suffer from the same problem as

the ones based onto syntactical structure. If used in combination with strong negative criteria and some more global positive ones, this criterion can help to improve the overall proof structure.

- The analysis of the used inference steps yields lemmata of average quality. The lengths of the subproofs are quite regular, and the size of the terms in the lemmata as well as in the proof chains are rather small. Proofs structured with this criterion bear some similarity to proofs structured by simply breaking long proof chains (see section 4.8), but appear more apealing and regular to the viewer. As a possible consequence we consider to use the weight of the inference steps instead of the length of the proof chain in some of the more complex criteria.

- Inserting lemmata whenever the proof chain becomes too long is the weakest of the global criteria. While the structured proofs are much better to follow then monolithic proofs, the lemmata bear no special meaning. However, while the criteria does not produce good results by itself, it is a very good backup criterion for cases in which more powerful criteria leave some large subproofs.

Apart form the different criteria we have to discuss the two implemented ways to combine them. We noted that both approaches have advantages. They can be summed up as follows: Analysing the complete proof with all criteria to be used at once yields less extreme lemmata. Analysing the proof with the different critera one after the other generates some better lemmata. However, it also produces some less than desirable lemmata. Which overall result is considered better has to be decided for each single case, and depends largely on personal preferences.

All in all the lemmata found by our algorithms are sufficient for most problems and usually lead to proofs quite readable for humans.

5 Proof presentation

Completion based proof systems work mainly by applying inference steps to *sets of equations*, thereby deriving more equations. Humans generally use another concept: They apply existing equations to *terms*, building equational chains. New equations are only generated when such a chain becomes unwieldy. The basic difference is that humans are working on terms, using equations as tools, while an automatic proof system works on equations (and rules), using certain inferences as tools.

This difference makes automatic proofs very hard to follow. While every single inference is easily understood and proved to be correct, the complete proof is generated in small, largely independent pieces that arrive in a more or less random order. The original axioms (used heavily by humans) are applied only very occasionally, and their role in the final proof is very hard to perceive.

To make automatic proofs easier to understand we transform them into a calculus employing the same *equational chains* used by human mathematicians. These proofs can be presented in a very natural way, resembling textbook proofs for simple mathematical problems.

5.1 Equational Chains: A calculus for proof presentation

In the calculus presented here each equation is accompanied by an equational chain (called *justification*) describing the applications of (known) equations necessary to transform the two terms of the equation into each other. Therefore an equation is not viewed as a logical consequence of a set of equations, but as the result of the application of these equations to a term. Although the two approaches are equivalent, the users viewpoint has changed.

Definition 11 : Justifications for equations

1) A tuple $(\mathtt{u}, (\mathtt{s{=}t}, \mathtt{p}, \sigma), \mathtt{v})$ is called *justification* for the equation $\mathtt{u{=}v}$, if $\mathtt{u}|_\mathtt{p} = \sigma(\mathtt{s})$ and $\mathtt{u}[\mathtt{p} \leftarrow \sigma(\mathtt{t})] = \mathtt{v}$.

2) A tuple $(\mathtt{u}, (\overline{\mathtt{s{=}t}}, \mathtt{p}, \sigma), \mathtt{v})$ is called *justification* for the equation $\mathtt{u{=}v}$, if $\mathtt{u}|_\mathtt{p} = \sigma(\mathtt{t})$ and $\mathtt{u}[\mathtt{p} \leftarrow \sigma(\mathtt{s})] = \mathtt{v}$.

3) Let $(\mathtt{s}, \mathtt{B}_1, \mathtt{t})$ and $(\mathtt{t}, \mathtt{B}_2, \mathtt{u})$ be justifications for $\mathtt{s{=}t}$ and $\mathtt{t{=}u}$, respectively. Then $(\mathtt{s}, \mathtt{B}_1, \mathtt{t}) \bullet (\mathtt{t}, \mathtt{B}_2, \mathtt{u}) \equiv (\mathtt{s}, \mathtt{B}_1, \mathtt{t}, \mathtt{B}_2, \mathtt{u})$ is a justification for $\mathtt{s{=}u}$.

Given this recursive definition, a justification is a chain of the form

$$(\mathtt{u}_0, (\mathtt{s}_1 \doteq \mathtt{t}_1, \mathtt{p}_1, \sigma_1), \mathtt{u}_1, (\mathtt{s}_2 \doteq \mathtt{t}_2, \mathtt{p}_2, \sigma_2), \mathtt{u}_2, \ldots, \mathtt{u}_{n-1}, (\mathtt{s}_n \doteq \mathtt{t}_n, \mathtt{p}_n, \sigma_n), \mathtt{u}_n)$$

We write $\mathtt{s} \doteq \mathtt{t}$ to denote either $\mathtt{s{=}t}$ or $\overline{\mathtt{s{=}t}}$, and require that $\overline{\overline{\mathtt{s{=}t}}} = \mathtt{s{=}t}$ holds.

We need some additional operation on justifications. The most simple of these operations makes use of the inherent symmetry of equality. It generates a justification for $\mathtt{t{=}s}$ from a justification for the symmetric equation $\mathtt{s{=}t}$.

Definition 12 : The symmetry operator on justifications
The symmetry operator $\overline{}$ is defined as follows:

1) $\overline{(\mathtt{u}, (\mathtt{s} \doteq \mathtt{t}, \mathtt{p}, \sigma), \mathtt{v})} = (\mathtt{v}, (\overline{\mathtt{s} \doteq \mathtt{t}}, \mathtt{p}, \sigma), \mathtt{u})$

2) $\overline{(\mathtt{u}, \mathtt{B}_1, \mathtt{t}, \mathtt{B}_2, \mathtt{v})} = \overline{(\mathtt{t}, \mathtt{B}_2, \mathtt{v})} \bullet \overline{(\mathtt{u}, \mathtt{B}_1, \mathtt{t})}$

Repeated application of 1) and 2) yields

$$\overline{(\mathtt{u}_0, (\mathtt{s}_1 \doteq \mathtt{t}_1, \mathtt{p}_1, \sigma_1), \mathtt{u}_1, (\mathtt{s}_2 \doteq \mathtt{t}_2, \mathtt{p}_2, \sigma_2), \mathtt{u}_2, \ldots, \mathtt{u}_{n-1}, (\mathtt{s}_n \doteq \mathtt{t}_n, \mathtt{p}_n, \sigma_n), \mathtt{u}_n)} =$$
$$(\mathtt{u}_n, (\overline{\mathtt{s}_n \doteq \mathtt{t}_n}, \mathtt{p}_n, \sigma_n), \mathtt{u}_{n-1}, \ldots, \mathtt{u}_2, (\overline{\mathtt{s}_2 \doteq \mathtt{t}_2}, \mathtt{p}_2, \sigma_2), \mathtt{u}_1, (\overline{\mathtt{s}_1 \doteq \mathtt{t}_1}, \mathtt{p}_1, \sigma_1), \mathtt{u}_0)$$

As one of our goals is to eliminate unnecessary intermediate results from the reasoning chain we need a mechanism to replace an application of such a fact by "simpler" facts. To this end we introduce a *flattening operator*. Using this operator we can replace an equation in a proof chain with the equations in its justification.

Definition 13 : The flattening operator S
The flattening operator S is defined as follows:

1) $\mathtt{S}(\mathtt{w}, \mathtt{q}, \tau, (\mathtt{u}, (\mathtt{s} \doteq \mathtt{t}, \mathtt{p}, \sigma), \mathtt{v})) = (\mathtt{w}[\mathtt{q} \leftarrow \tau(\mathtt{u})], (\mathtt{s} \doteq \mathtt{t}, \mathtt{q}.\mathtt{p}, \tau \circ \sigma), \mathtt{w}[\mathtt{q} \leftarrow \tau(\mathtt{v})])$

2) $\mathtt{S}(\mathtt{w}, \mathtt{q}, \tau, (\mathtt{s}, \mathtt{B}_1, \mathtt{t}, \mathtt{B}_2, \mathtt{u})) = \mathtt{S}(\mathtt{w}, \mathtt{q}, \tau, (\mathtt{s}, \mathtt{B}_1, \mathtt{t})) \bullet \mathtt{S}(\mathtt{w}, \mathtt{q}, \tau, (\mathtt{t}, \mathtt{B}_2, \mathtt{u}))$

Using these operators we can build new justifications from old ones. The following theorem shows the correctness of the transformations.

Theorem 4 : Symmetry and flattening

- Let B be a justification for u=v. Then $\overline{\mathtt{B}}$ is a justification for v=u.
- Let $(\mathtt{u}, \mathtt{B}_1, \mathtt{s}', (\mathtt{s}{=}\mathtt{t}, \mathtt{p}, \sigma), \mathtt{t}', \mathtt{B}_2, \mathtt{v})$ be a justification for u=v and let B be a justification for s=t. Then $(\mathtt{u}, \mathtt{B}_1, \mathtt{S}(\mathtt{s}', \mathtt{p}, \sigma, \mathtt{B}), \mathtt{B}_2, \mathtt{v})$ is another justification for u=v.
- Let $(\mathtt{u}, \mathtt{B}_1, \mathtt{s}', (\overline{\mathtt{s}{=}\mathtt{t}}, \mathtt{p}, \sigma), \mathtt{t}', \mathtt{B}_2, \mathtt{v})$ be a justification for u=v and let B be a justification for s=t. Then $(\mathtt{u}, \mathtt{B}_1, \mathtt{S}(\mathtt{s}', \mathtt{p}, \sigma, \overline{\mathtt{B}}), \mathtt{B}_2, \mathtt{v})$ is another justification for u=v.

The following short example will illustrate the use of the flattening operator:

Example: Let $(\mathtt{g}(\mathtt{a}, \mathtt{b}), (\mathtt{g}(\mathtt{x}, \mathtt{b}){=}\mathtt{g}(\mathtt{x}, \mathtt{c}), \lambda, \{\mathtt{x} \leftarrow \mathtt{a}\}), \mathtt{g}(\mathtt{a}, \mathtt{c}))$ be a justification for $\mathtt{g}(\mathtt{a}, \mathtt{b}){=}\mathtt{g}(\mathtt{a}, \mathtt{c})$, and let $(\mathtt{g}(\mathtt{x}, \mathtt{b}), (\mathtt{b}{=}\mathtt{c}, 2, \{\}), \mathtt{g}(\mathtt{x}, \mathtt{c}))$ be a justification for $\mathtt{g}(\mathtt{x}, \mathtt{b}){=}\mathtt{g}(\mathtt{x}, \mathtt{c})$.

Then

$\mathtt{S}(\mathtt{g}(\mathtt{a}, \mathtt{b}), \lambda, \{\mathtt{x} \leftarrow \mathtt{a}\}, (\mathtt{g}(\mathtt{x}, \mathtt{b}), (\mathtt{b}{=}\mathtt{d}, 2, \{\}), \mathtt{g}(\mathtt{x}, \mathtt{c})))$
$= (\mathtt{g}(\mathtt{a}, \mathtt{b})[\lambda \leftarrow \{\mathtt{x} \leftarrow \mathtt{a}\}(\mathtt{g}(\mathtt{x}, \mathtt{b}))], (\mathtt{b}{=}\mathtt{d}, \lambda.2, \{\mathtt{x} \leftarrow \mathtt{a}\} \circ \{\}),$
$\quad \mathtt{g}(\mathtt{a}, \mathtt{b})[\lambda \leftarrow \{\mathtt{x} \leftarrow \mathtt{a}\}(\mathtt{g}(\mathtt{x}, \mathtt{c}))])$
$= (\mathtt{g}(\mathtt{a}, \mathtt{b}), (\mathtt{b}{=}\mathtt{c}, 2, \{\mathtt{x} \leftarrow \mathtt{a}\}), \mathtt{g}(\mathtt{a}, \mathtt{c}))$

is an alternative justification for $\mathtt{g}(\mathtt{a}, \mathtt{b}){=}\mathtt{g}(\mathtt{a}, \mathtt{c})$. Here we replaced the application of $\mathtt{g}(\mathtt{x}, \mathtt{b}){=}\mathtt{g}(\mathtt{x}, \mathtt{c})$ by an application of the equation b=c.

As the calculus of equational chain already emulates the definition of operational E-equality rather closely we can easily use justified equations to express proofs for equality. This is achieved by considering only equations valid in E for use in justifications.

Definition 14 : E-justified equations
Let E be a set of equations, the *axioms* of the set of E-justified equations. An equation $s{=}t$ is called *justified in* E, if one of the following conditions is met:

1) $s{=}t \in E$. In this case $B = (s, (s{=}t, \lambda, \sigma_{id}), t)$ is the justification for $s{=}t$.

2) There exist E-justified equations $s_0{=}t_0$, $s_1{=}t_1$, ..., $s_n{=}t_n$ and a justification $(u_0, (s_1 \doteq t_1, p_1, \sigma_1), u_1, (s_2 \doteq t_2, p_2, \sigma_2), u_2, \ldots, u_{n-1}, (s_n \doteq t_n, p_n, \sigma_n), u_n)$ with $u_0 \equiv s$ and $u_n \equiv t$.

In the above definition we allow any E-justified equation in the justifications. However, as stated earlier, we want to use only axioms and selected lemmata in equational chains. Therefore we will introduce the concept of *flat justifications*.

Definition 15 : Flat justifications
Let E be a set of equations, **L** a set of equations justified in E. A justification

$$(u_0, (s_1 \doteq t_1, p_1, \sigma_1), u_1, (s_2 \doteq t_2, p_2, \sigma_2), u_2, \ldots, u_{n-1}, (s_n \doteq t_n, p_n, \sigma_n), u_n)$$

is called a *flat justification with respect to* E *and* **L** if $s_i{=}t_i \in (E \cup \mathbf{L})$ holds for all i.

These flat justifications are sufficient to justify any equation valid in E.

Theorem 5 : Flat justifications and E-equality
Let E be a set of equations.

- Let **L** be an arbitrary set of equations justified in E. The equation $s{=}t$ is justified in E if and only if a flat justification with respect to E and **L** exists for it.
- $s =_E t$ holds if and only if $s{=}t$ is justified in E.

Please note that the first part of the above theorem applies even for **L**= {}. Lemmata are not necessary in any proof. They only allow a proof to be followed more conveniently.

5.2 PCL listings and equational chains

A proof in the form of equational chains is very similar to an equational proof done by a human. The remaining differences are only superficial details. Humans usually don't mention places or substitutions, and use a slightly different display format. However, these problems can be solved easily by projecting only the wanted parts of the proof in any desired format.

Of course a proof is not usually generated in the form of equational chains. Therefore we need to transform the proofs delivered by the proof system to the new form. We will now show how this can be achieved.

The foundation of our algorithm is the following transformation system. It will generate (flat) justifications for simple PCL expressions if such justifications exist for the arguments of these expressions.

Definition 16 : The transformation system JE (*justified equations*)
In the following we write $\mathtt{s} \simeq \mathtt{t}$ to denote either $\mathtt{s}{=}\mathtt{t}$ or $\mathtt{s} \rightarrow \mathtt{t}$. The function `eval` yields the *value* of a PCL expression (compare 3.1). The following conventions are used in the transformation rules:

- $\mathtt{eval}(\texttt{<expr}_1\texttt{>}) = \mathtt{s}_1 \simeq \mathtt{t}_1$ and $\mathtt{eval}(\texttt{<expr}_2\texttt{>}) = \mathtt{s}_2 \simeq \mathtt{t}_2$.
- $\mathtt{B}_1$ is a (flat) justification for $\mathtt{s}_1 \simeq \mathtt{t}_1$, $\mathtt{B}_2$ is a (flat) justification for $\mathtt{s}_2 \simeq \mathtt{t}_2$.

Input for a transformation step is a PCL expression with value $\mathtt{s} \simeq \mathtt{t}$, output is a (flat) justification for the value of this expression.

(1) Axioms

$$\frac{\mathtt{initial}}{(\mathtt{s}, (\mathtt{s}{=}\mathtt{t}, \lambda, \sigma_{\mathtt{id}}), \mathtt{t})}$$

Compare the definition of E-justified equations (Definition 14 on page 45).

(2) Quotes

$$\frac{\texttt{<expr}_1\texttt{>}}{\mathtt{B}_1}$$

$\texttt{<expr}_1\texttt{>}$ here is just an identifier referencing another PCL step.

(3) Orienting

$$\frac{\mathtt{orient}(\texttt{<expr}_1\texttt{>}, \mathtt{u})}{\mathtt{B}_1}$$

$$\frac{\mathtt{orient}(\texttt{<expr}_1\texttt{>}, \mathtt{x})}{\overline{\mathtt{B}_1}}$$

(4) Critical pairs

$$\frac{\mathtt{cp}(\texttt{<expr}_1\texttt{>}, \mathtt{L.p}, \texttt{<expr}_2\texttt{>}, \mathtt{L})}{\mathtt{S}(\sigma(\mathtt{s}_1), \mathtt{p}, \sigma, \overline{\mathtt{B}_2}) \bullet \mathtt{S}(\sigma(\mathtt{s}_1), \lambda, \sigma, \mathtt{B}_1)} \quad \text{with } \sigma = \mathtt{mgu}(\mathtt{s}_1|_{\mathtt{p}}, \mathtt{s}_2).$$

$$\frac{\mathtt{cp}(\texttt{<expr}_1\texttt{>}, \mathtt{L.p}, \texttt{<expr}_2\texttt{>}, \mathtt{R})}{\mathtt{S}(\sigma(\mathtt{s}_1), \mathtt{p}, \sigma, \mathtt{B}_2) \bullet \mathtt{S}(\sigma(\mathtt{s}_1), \lambda, \sigma, \mathtt{B}_1)} \quad \text{with } \sigma = \mathtt{mgu}(\mathtt{s}_1|_{\mathtt{p}}, \mathtt{t}_2).$$

$$\frac{\mathtt{cp}(\texttt{<expr}_1\texttt{>}, \mathtt{R.p}, \texttt{<expr}_2\texttt{>}, \mathtt{L})}{\mathtt{S}(\sigma(\mathtt{t}_1), \mathtt{p}, \sigma, \overline{\mathtt{B}_2}) \bullet \mathtt{S}(\sigma(\mathtt{t}_1), \lambda, \sigma, \overline{\mathtt{B}_1})} \quad \text{with } \sigma = \mathtt{mgu}(\mathtt{t}_1|_{\mathtt{p}}, \mathtt{s}_2).$$

$$\frac{\mathtt{cp}(\texttt{<expr}_1\texttt{>}, \mathtt{R.p}, \texttt{<expr}_2\texttt{>}, \mathtt{R})}{\mathtt{S}(\sigma(\mathtt{t}_1), \mathtt{p}, \sigma, \mathtt{B}_2) \bullet \mathtt{S}(\sigma(\mathtt{t}_1), \lambda, \sigma, \overline{\mathtt{B}_1})} \quad \text{with } \sigma = \mathtt{mgu}(\mathtt{t}_1|_{\mathtt{p}}, \mathtt{t}_2).$$

(5) Simplifications

$$\frac{\texttt{tes-red(<expr}_1\texttt{>, L.p, <expr}_2\texttt{>, L)}}{\texttt{S}(\texttt{s}_1, \texttt{p}, \sigma, \overline{\texttt{B}_2}) \bullet \texttt{B}_1} \quad \text{with } \sigma = \texttt{match}(\texttt{s}_2, \texttt{s}_1|_\texttt{p}).$$

$$\frac{\texttt{tes-red(<expr}_1\texttt{>, L.p, <expr}_2\texttt{>, R)}}{\texttt{S}(\texttt{s}_1, \texttt{p}, \sigma, \texttt{B}_2) \bullet \texttt{B}_1} \quad \text{with } \sigma = \texttt{match}(\texttt{t}_2, \texttt{s}_1|_\texttt{p}).$$

$$\frac{\texttt{tes-red(<expr}_1\texttt{>, R.p, <expr}_2\texttt{>, L)}}{\texttt{B}_1 \bullet \texttt{S}(\texttt{t}_1, \texttt{p}, \sigma, \texttt{B}_2)} \quad \text{with } \sigma = \texttt{match}(\texttt{s}_2, \texttt{t}_1|_\texttt{p}).$$

$$\frac{\texttt{tes-red(<expr}_1\texttt{>, R.p, <expr}_2\texttt{>, R)}}{\texttt{B}_1 \bullet \texttt{S}(\texttt{t}_1, \texttt{p}, \sigma, \overline{\texttt{B}_2})} \quad \text{with } \sigma = \texttt{match}(\texttt{t}_2, \texttt{t}_1|_\texttt{p}).$$

The system **JE** is indeed capable of generating correct justifications:

Theorem 6 : Correctness of JE
The transformation system **JE** is correct. If the requirements from the definition are fulfilled, the generated justification will be a (flat) justification for the value of the expression. It will only use facts from the arguments justifications.

The proof system **JE** up to now offers only support for new, valid equations generated from the axioms. However, in many proofs the goal is not proofed constructively (by generating an equation subsuming the goal), but destructively by reducing both sides of the goal to common normal forms. The following theorem serves to generate justifications for such goals. `u=v` is the goal to be proved.

Theorem 7 : Justifications for destructively proofed equations

- Let `B` be a justification for the trivial equation `s=s`. Let `u=v` appear in `B` exactly once, at top level and instantiated with the empty substitution, that is $\texttt{B} = (\texttt{s}, \texttt{B}_1, \texttt{u}, (\texttt{u=v}, \lambda, \sigma_{\texttt{id}}), \texttt{v}, \texttt{B}_2, \texttt{s})$. Then $\overline{(\texttt{s}, \texttt{B}_1, \texttt{u})} \bullet \overline{(\texttt{v}, \texttt{B}_2, \texttt{s})}$ is a justification for `u=v`.

- Let `B` be an justification for an equation `s=t` not containing `u=v`. Let `B'` be another justification for `s=t` with `u=v` appearing exactly once, at top level and instantiated with the empty substitution, that is $\texttt{B}' = (\texttt{s}, \texttt{B}_1, \texttt{u}, (\texttt{u=v}, \lambda, \sigma_{\texttt{id}}), \texttt{v}, \texttt{B}_2, \texttt{s})$. Then $\overline{(\texttt{s}, \texttt{B}_1, \texttt{u})} \bullet \texttt{B} \bullet \overline{(\texttt{v}, \texttt{B}_2, \texttt{t})}$ is a justification for `u=v`.

Using this theorem and treating the hypotheses as axioms we can generate justifications for goals proved with any combinations of constructive and destructive inferences. Please note that the intermediate justifications are not valid justifications in `E`. Only by applying theorem 7 the final justifications are arrived at. To generate the intermediate justifications we need to expand the system **JE** by adding a rule dealing with hypotheses.

Definition 17 : The transformation system JE'
Using the same postulates as in definition 16, the system **JE'** consists of the rules from **JE** and the following new rule:

(1') Hypotheses

$$\frac{\texttt{hypothesis}}{(\texttt{s}, (\texttt{s=t}, \lambda, \sigma_{\texttt{id}}), \texttt{t})}$$

Using the system **JE'** we can generate justifications for all facts in a correct PCL listing. The following example will demonstrate this:

Example: We are presenting a very short PCL listing, accompanied by the justifications generated using **JE'**.

```
0 : tes-eqn : f(e(),x) = x : initial
```

$\texttt{B}_0 = (\texttt{f(e(), x)}, (\texttt{f(e(), x)=x}, \lambda, \{\}), \texttt{x})$ is a justification for the fact of step 0 by rule (1).

```
3 : tes-goal : g(f(e(),x)) = f(e(),g(x)) : hypothesis
```

$\texttt{B}_3 = (\texttt{g(f(e(), x))}, (\texttt{g(f(e(), x))=f(e(), g(x))}, \lambda, \{\}), \texttt{f(e(), g(x))})$ using rule (1').

```
4 : tes-rule : f(e(),x) -> x : orient(0,u)
```

$\texttt{B}_4 = \texttt{B}_0 = (\texttt{f(e(), x)}, (\texttt{f(e(), x)=x}, \lambda, \{\}), \texttt{x})$ according to rule (2), first case.

```
5 : tes-goal : g(f(e(),x)) = g(x) : tes-red(3,R,4,L)
```

$$\begin{aligned}\texttt{B}_5 &= \texttt{B}_3 \bullet \texttt{S}(\texttt{f(e(), g(x))}, \lambda, \{\texttt{x} \leftarrow \texttt{g(x)}\}, \texttt{B}_4)\\ &= (\texttt{g(f(e(), x))}, (\texttt{g(f(e(), x))=f(e(), g(x))}, \lambda, \{\}), \texttt{f(e(), g(x))},\\ &\qquad (\texttt{f(e(), x)=x}, \lambda, \{\texttt{x} \leftarrow \texttt{g(x)}\}), \texttt{g(x)})\end{aligned}$$

according to rule (5), third case.

```
6 : tes-final : g(x) = g(x) : tes-red(5,L.1,4,L)
```

$$\begin{aligned}\texttt{B}_6 &= \texttt{S}(\texttt{g(f(e(), x))}, 1, \{\}, \overline{\texttt{B}_4}) \bullet \texttt{B}_5\\ &= (\texttt{g(x)}, (\overline{\texttt{f(e(), x)=x}}, 1, \{\}), \texttt{g(f(e(), x))})\\ &\quad \bullet (\texttt{g(f(e(), x))}, (\texttt{g(f(e(), x))=f(e(), g(x))}, \lambda, \{\}), \texttt{f(e(), g(x))},\\ &\qquad (\texttt{f(e(), x)=x}, \lambda, \{\texttt{x} \leftarrow \texttt{g(x)}\}), \texttt{g(x)})\end{aligned}$$

using rule (5), first case.

$\texttt{B}_6$ now is a justification for the trivial equation $\texttt{g(x)=g(x)}$, and fulfilles the criteria from theorem 7 with respect to $\texttt{g(f(e(), x))=f(e(), g(x))}$. By appying this theorem we arrive at

$$\begin{aligned}\texttt{B} &= \overline{(\texttt{g(x)}, (\overline{\texttt{f(e(), x)=x}}, 1, \{\}), \texttt{g(f(e(), x))})}\\ &\quad \bullet \overline{(\texttt{f(e(), g(x))}, (\texttt{f(e(), x)=x}, \lambda, \{\texttt{x} \leftarrow \texttt{g(x)}\}), \texttt{g(x)})}\\ &= (\texttt{g(f(e(), x))}, (\texttt{f(e(), x)=x}, 1, \{\}), \texttt{g(x)}, (\overline{\texttt{f(e(), x)=x}}, \lambda, \{\}), \texttt{f(e(), g(x))})\end{aligned}$$

$\texttt{B}$ now is a valid justification in $\texttt{E}$ for $\texttt{g(f(e(), x))=f(e(), g(x))}$.

The transformation rules in **JE** can for the most part be viewed as dual (or reverse) to the operations of *orientation*, *simplification* and *critical pair building* used in an unfailing completion algorithm. They are not exactly dual to the inference rules for unfailing completion because they are acting on a more concrete level. However, they can be easily generalized to match these original rules.

5.3 An algorithm for proof transformation

We will now develop an algorithm for generating equational chains from PCL listings. To keep the description of this algorithm compact and understandable we need a set of auxilliary functions. The first two functions employ **JE'** recursively to generate justifications for arbitrary PCL steps and expressions.

Definition 18 : The functions `JUST` and `LJUST`
Let `step` be a PCL step and `expr` be a PCL expression.

- The function `JUST` is defined as follows:
  - $\texttt{JUST}(\texttt{step}) = \texttt{JUST}(\texttt{EXPR}(\texttt{step}))$
  - $\texttt{JUST}(\texttt{expr})$ is calculated recursively:
    Assign $\texttt{JUST}(\texttt{expr}_\texttt{i})$ to $\texttt{B}_\texttt{i}$ and $\texttt{eval}(\texttt{expr}_\texttt{i})$ to $(\texttt{s}_\texttt{i}, \texttt{t}_\texttt{i})$ for all direct arguments $\texttt{expr}_\texttt{i}$ of `expr`. Use the $\texttt{B}_\texttt{i}$ and $(\texttt{s}_\texttt{i}, \texttt{t}_\texttt{i})$ as input and apply the appropriate rule from **JE'** to it. Let `B` be the result of this rule, then $\texttt{JUST}(\texttt{expr}) = \texttt{B}$.
- The function `LJUST` is defined quite analogous to `JUST`. It does, however, employ lemmata to keep the justifications shorter.
  - $\texttt{LJUST}(\texttt{step}) = \begin{cases} \texttt{LJUST}(\texttt{EXPR}(\texttt{step})) & \text{if } \texttt{TYPE}(\texttt{step}) \neq \texttt{tes-lemma} \\ (\texttt{s}, (\texttt{s=t}, \lambda, \sigma_{\texttt{id}}), \texttt{t}) & \text{otherwise} \end{cases}$

    with $\texttt{FACT}(\texttt{step}) = \texttt{s} \simeq \texttt{t}$.
  - $\texttt{LJUST}(\texttt{expr})$ is calculated recursively as above:
    Assign $\texttt{LJUST}(\texttt{expr}_\texttt{i})$ to $\texttt{B}_\texttt{i}$ and $\texttt{eval}(\texttt{expr}_\texttt{i})$ to $(\texttt{s}_\texttt{i}, \texttt{t}_\texttt{i})$ for all direct arguments $\texttt{expr}_\texttt{i}$ of `expr`. Use the $\texttt{B}_\texttt{i}$ and $(\texttt{s}_\texttt{i}, \texttt{t}_\texttt{i})$ as input and apply the appropriate rule from **JE'** to it. Let `B` be the result of this rule, then $\texttt{LJUST}(\texttt{expr}) = \texttt{B}$.

The functions just presented enable us to handle all kinds of generated equations. However, we still need tools to deal with equations proved destructively. The following functions help us to make full use of theorem 7.

Definition 19 : The functions `UsesHypothesis` and `SplitChain`
The function `UsesHypothesis` checks a justification for an occurance of a hypothesis at top level and instanciated with the empty substitution. If such an occurance exists,

the function SplitChain will split the justification at this point. More formaly the following holds:

- UsesHypothesis(B, list) = TRUE if and only if FACT(step) is used in B, at top level and instanciated with the empty substitution, for a step in list with EXPR(step) = hypothesis. If this is not the case, UsesHypothesis(B, list) yields FALSE.
- Now consider a B and a list with UsesHypothesis(B, list) = TRUE. In this case B can be written in the form $(\mathtt{s}, \mathtt{B}_1, \mathtt{u}, (\mathtt{u{=}v}, \lambda, \sigma_{\mathtt{id}}), \mathtt{v}, \mathtt{B}_2, \mathtt{t})$, with u=v being the fact of a step with EXPR(step) = hypothesis. Then the function is defined by SplitChain(B, list) = $(\mathtt{u{=}v}, (\mathtt{s}, \mathtt{B}_1, \mathtt{u}), (\mathtt{v}, \mathtt{B}_2, \mathtt{t}))$. In any other case SplitChain(B, list) is undefined.

We will now use these auxilliary functions to present a compact algorithm for the transformation of PCL listings to equational chains. It will generate a list of justified equations with the following properties:

- All facts of PCL steps with TYPE(step) = tes-lemma will be represented in the list with their fact.
- All PCL steps with TYPE(step) = tes-final concluding a proof of an original goal will be represented in the list with the fact of this goal.
- All other PCL steps with TYPE(step) = tes-final will be represented in the list with their own fact.
- All justifications appearing in the list will only use earlier equations from the lists or facts from steps with EXPR(step) = initial. The justifications therefore are flat justifications with respect to {FACT(step)|EXPR(step) = initial} and {FACT(step)|TYPE(step) = tes-lemma}.

Given this properties, the generated list represents a proof (in equational chains) for the original hypotheses and eventual new final equations.

Input:	`in`	A list of PCL steps.
Output:	`out`	A list of justified equations.
Variables	`step`	A single PCL step.
	`u, v`	Terms.
	B_1, B_2	Justifications.
	$expr_1, expr_2$	PCL-expressions.
Functions:	`NOTEMPTY(list)`	FALSE, if `list` is empty, TRUE if not.
	`FIRST(list)`	First entry in `list`.
	`EXCEPTFIRST(list)`	`list` without its first entry.
	`APPEND(list,B)`	List generated by appending `B` as the last element to `list`.
	`TYPE(step)`	Type of `step` (compare 3.2).
	`EXPR(step)`	Expression of a PCL step.
	`TOP(expr)`	Top symbol of a PCL expression.
	`SUBEXPRS(expr)`	The direct subexpressions of `expr` that are PCL expressions (as opposed to place designators or directional arguments).

```
store := in;
WHILE NOTEMPTY(in)
  step := FIRST(in);
  in := EXCEPTFIRST(in);
  IF TYPE(step) = tes-lemma THEN
    out := APPEND(out,(FACT(step),LJUST(step)));
  IF TYPE(step) = tes-final THEN
    IF TOP(EXPR(step)) = instance THEN
      A (sub-) goal is being proven by instanciation
      (expr1,expr2) := SUBEXPRS(EXPR(step));
      (u=v,B1, B2) := SplitChain(LJUST(expr1),store);
      out := APPEND(out,(u=v, B1 • LJUST(step) • B2));
    ELSE IF NOT(UsesHypothesis(LJUST(step),store)) THEN
      An interesting constructive fact is made a theorem
      out := APPEND(out,(FACT(step),LJUST(step)));
    ELSE
      A goal is being proved by reduction
      (u=v,B1, B2) := SplitChain(LJUST(step),store);
      out := APPEND(out,(u=v,B1 • B2);
    ENDIF
  ENDIF
ENDWHILE
```

Appendix B provides two examples for proofs transfomed using our implementation of the algoritm above.

6 Dealing with a distributed proof system

In analyzing completion based proofs we have developed tools to deal with a verbose listing of the proof steps. The language used to describe the proof is PCL, which has been introduced in [Sch93]. It represents the proof in a generic format independent of the internals of a specific prover. Obtaining this listing from a sequential (as opposed to parallel) program is rather straightforward, but there are a number of problems to consider when dealing with a parallel proof system.

6.1 Measuring without disturbing

Given todays computer systems, input and output operations are usually very time expensive compared to computation and symbol manipulation. Generating a complete listing of the inference steps during a proof session does therefore significantly increase the time needed to find the proof. This does not matter in sequential provers, which usually show a deterministic behaviour. Given a specific input they proceed always through the same states and eventually arrive at the same proof. The process is independent of outside events and of the elapsed time.

This is not true for parallel programs, which usually rely heavily on cooperation and whose behaviour is largely influenced by the timing of both the complete system and the single components. Our experiments have shown that producing a complete PCL-listing during the generation of a proof significantly alters the behaviour of the proof system. In particular, results found without extensive documentation of the proof run are not directly reproducible under the altered circumstances.

However, generating full documentation of every proof run is not a desirable option. Using the prover without the protocol is usually much more convenient, especially while searching for a new proof. The significant advantage in speed does allow deeper searches within the same time limits. The protocol is only needed when a already existing proof is to be analyzed in detail, or when the proof has to be presented to humans. We therefore chose another way to generate the data necessary for this analysis.

During the initial phase of the proof generation no verbose listing is written. Instead we use a very short, specialized protocol written only at some crucial points in the process. For the teamwork method these points are the team meetings, during which the different processes exchange their information. The use of this protocol allows us to reproduce the proof without any further dependency on the elapsed time. We do no longer use the elapsed time but a comparison with the protocol data as a measure for the progress of the proof process. During this time independent reproduction a verbose listing in PCL is generated, which can then be used for further analysis.

Note that while this concept can be implemented very easily for the teamwork method it is not viable for most other approaches to parallel processing. It works only with algorithms using short periods of intensive communication and longer periods of sequential and deterministic work not interrupted by interprocess communication. This property is inherent in the teamwork method but not for example in the concepts

presented in [BH92].

6.2 Sequentializing parallel proofs

Humans are only capable of following one thread at a time. To prepare proofs in a form suitable for human reading the proof has to be a single chain of logical arguments. A conventional proof system generates such a chain automatically, but a parallel proof system without shared memory can (and will) analyze more than one thread at a time. These different threads have to be integrated into a single proof chain, preferably in a way that keeps the context of each proof step as intact as possible.

Additionally, in a complete recording of the reasoning process the facts have to be labeled so that a justification for a new fact can point at the facts it has been derived from. When integrating the different threads these labels have to be unique to avoid problems with ambiguous references.

Another, more practical, consideration is that we deal with very large amounts of data. For a detailed analysis of the proof we use a protocol which contains only the facts actually used in generating the final proof, not the (usually much bigger) set of unused facts. This extraction of the important facts can be done rather efficiently because we use a total ordering on the labels and demand that labels in a protocol are used in ascending order only. We want to maintain this property for protocols generated from a parallel proof session.

For the teamwork method the different threads are generated by the different experts. During the working phases each expert is working on a single thread. At the team meetings these different threads are integrated into a common knowledge base. This base is then used as the single starting point for the threads of the next working phase. An important feature of this approach is that an inference process can only access informations from his own thread and from the common data base generated during the last team meeting.

PCL represents the inference processes as a list of steps labeled by an identifier and containing the result of a single inference as well as a description of the inference and the labels of the parent steps. The identifier consists of a list of arbitrary length, containing positive integers. The ordering defined on this identifiers is the lexicographical extension of the standard ordering on the natural numbers. This allows us to split the set of all possible identifiers into a hierarchical structure of infinitely many different *name spaces*. If carefully chosen this name spaces can be ordered in a way that the ordering on the name spaces is total and compatible with the ordering on the identifiers. In this case every two name spaces can be compared and identifiers from a "greater" name space are always larger than identifiers from a "smaller" name space.

To get more specific we use the first n elements in the list that constitutes an identifier as the criterion to distinguish different name spaces. For protocols of teamwork sessions the identifiers are composed of at least three integers. The first one designates the working phase or team meeting the inference happened in, the second element distinguishes between the different experts and the rest of the identifier is used to label

the different steps within a single thread.

The rules for identifiers and name spaces can be summarized as follows: Identifiers consist of at least 3 integers, they are of the form *cycle.expert.count*. The first integer, *cycle*, numbers (in ascending order) the team meetings and working phases. Identifiers from team meeting n start with $cycle = 2 \times n$, identifiers from working phase n start with $cycle = (2 \times n) + 1$. The second element, *expert*, is the number of the processor running the expert that generated the proof step. The rest of the identifier, *count*, is usually just another simple integer which numbers all the proof steps done by that particular expert. In the general case it can be an arbitrary list that distinguishes among the proof steps done by a single expert. Figure 4 shows this arrangement for a simple example.

This concept does achieve most of the goals set above. By simply concatenating the protocols of the different experts in a single working phase a protocol for that working phase can be generated. It exhibits three important properties:

- The identifiers of the proof steps are used in an ascending order.
- Every proof step references only steps with smaller identifiers.
- As the entire thread is kept in one part the proof steps usually do not loose the context they were generated in.

These protocols can then be concatenated with the protocols generated from the team meetings in the order indicated by there respective name spaces. The resulting list constitutes a valid sequential PCL protocol of the complete proof session. This shares most of the above characteristics and additionally (if the experts work correctly) contains every step referenced in an inference.

6.2.1 Eliminating redundancies

A second, minor problem with the distributed proof system is that some facts might be found independently by more then one expert. This is not a problem for the validity of the generated proof, however, it is not a feature desirable in proofs presented to humans. The redundant data does not add any new information but merely confuses the reader.

We cope with this problem by eliminating these steps, using two different approaches. In a first step we take the complete protocol and eliminate all steps proved by a simple reference to a previous step. This is achieved by replacing all references to the redundant step with references to the prior step and by (possibly) changing the previous steps type to reflect the additional information that might have been incorporated into the redundant step (the redundant step might have been of type `tes-final` or `tes-intermed`, while the previous step might have been an ordinary result).

It should be noted that this first handling of the protocol is not primarily done for the elimination of redundancies, but to move the more detailed types to the first occurence

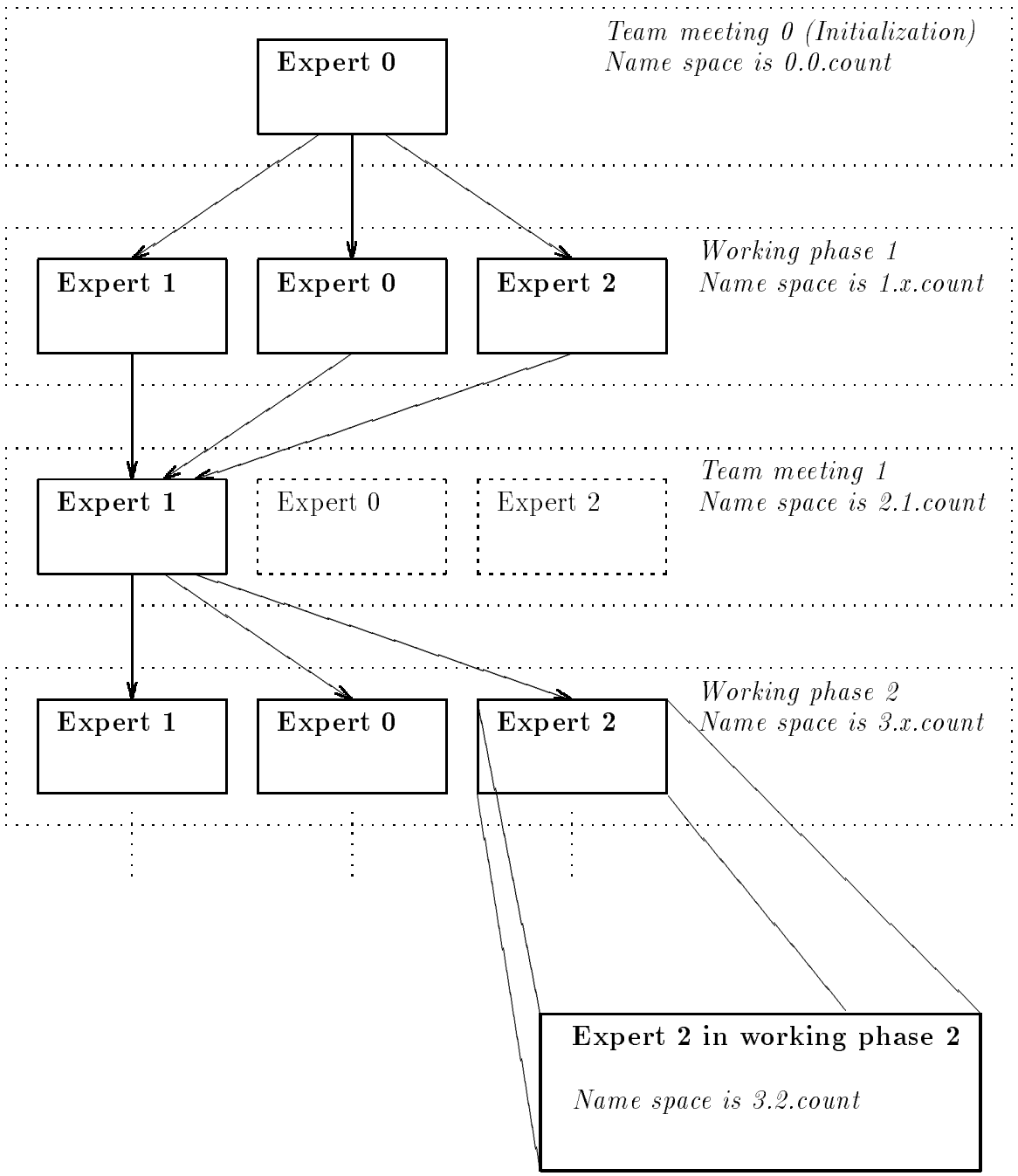

Figure 4: Team work and name spaces

of a fact, thus ensuring that this information is not lost during the extraction phase. It does, however, eliminate a couple of redundant steps, too.

The second elimination phase does check all steps with compatible types for identical facts. If two steps with identical facts (and compatible types) are found the listing

is treated as mentioned above: All references to the second step are changed into references to the first step. This operation is usually performed after a first extraction, because the operation tends to be quite costly (in terms of CPU time) for large listings. A final extraction then removes all steps that became redundant during this phase.

6.3 Handling large amounts of data

Even protocols of sequential proofs can become very large, as section 3.3 demonstrated. This leads to serious practical problems in handling the files. For a distributed proof system these problems are further aggravated. First, the same protocol will grow about 20 %, because of the necessarily more complex identifier structure reflecting the name spaces introduced in the previous section.

The real increase in the amount of data is the result of multiple processes generating inferences. Given the same amount of time a distributed system with n processors will generate approximately n times the inferences of a system with one processor. As PCL depends on a complete protocol of all inference steps the size of the PCL listings will reflect this fact.

We have not yet encountered a situation our system could not master. This is, however, a result of the success of the *teamwork method*, which generated much shorter proofs. We expect PCL protocols for more difficult problem to quickly become unmanageable.

To handle this problem we can again use a feature of TEAMWORK. As has been stated in section 2.3 only a few results of inferior experts are selected at the team meetings, the other results are dropped from the data base. We can transfer this process of *forgetting* to the PCL layer by interleaving proving phases and extraction.

To be more specific, the exceptional results handed to the master of the team are marked in the PCL protocol by type identifiers (`tes-intermed, tes-intermedgoal` and `crit-intermedgoal`, compare section 3.1) expressing their special status.

The PCL listings produced by the inferior experts are then extracted using the algorithm described in section 3.3, treating the intermediate results as finals. All facts needed for the generation of the intermediate results are preserved, all other generated facts are discarded. As these other results cannot be referenced by the prover any more, we will still get a valid PCL listing describing the proof.

The final PCL listing is usually much smaller this way, and can be handled quite easily in most cases. However, there is a price to pay. If this intermediate extraction is used no complete listing of *all* inferences is generated. As we are interested much more in the steps necessary for the proof we think this is acceptable.

7 Benefits from going distributed

Using a distributed, TEAMWORK based proof system has a number of advantages in comparison to a sequential prover, ranging from the obvious advantage of increased speed to some more subtle aspects in proof analysis and presentation. This chapter will explain the benefits gained.

7.1 Increased power of the proof system

Of course the original reason for choosing a distributed system are the gains in speed and power. These reasons still hold. The increase in the provers capabilities allow access to a broader class of problems, including more difficult and general examples.

Our equational prover, the DISCOUNT system, achieves this goal of more power in an admirable way. It has often demonstrated superior strategies when running in distributed mode. This manifests in super-linear speedups for many problems (see [AD93] for a detailed discussion of some examples). Proof sessions occupying a single machine for hours can now often be solved in a couple of minutes, using a cluster of workstations. We also found a couple of problems we could prove using the distributed mode of DISCOUNT, but not with any single strategy in sequential mode. One example is shown in table 2.

Using the methods detailed in section 6 we were able to maintain the increased power even for proof processes documented in a way that allows a exact analysis of the proof. Additionally, as we can now reproduce proof runs, there is no need to generate a full protocol of each proof process. Only successful proof runs need to be reconstructed, yielding a complete protocol only at reproduction time.

7.2 Shorter proof protocols

The superior behaviour of DISCOUNT in distributed mode also shows in the better ratio of necessary inferences to executed inferences. The prover does behave more goal oriented, a larger part of his work does actually further the proof. Many important results are generated more directly and earlier, because the different experts arrive at important results in their part of the search space. These results are usually recognized by the referees and integrated into the prover's system of rules and equations, thus forming a strong reasoning base for the prover quite early.

The total number of steps in an extracted listing is roughly the same for the distributed and sequential system. However, due to the improved ratio of necessary to executed steps we generally get much smaller complete protocols for the same problem. For large examples the proof listing typically shrinks by at least 50 %, quite often the reduction is even more dramatic. These protocols are easier to handle than the larger, monolithic protocols of sequential proofs.

To show this trend we have included data from distributed proofs for some of the more

difficult examples[3] from Table 1 to demonstrate the improvement. Table 2 shows the data for proofs generated in distributed mode.

7.3 Easier handling of extreme protocols

The faster and more efficient proofs possible with the TEAMWORK method usually result in much smaller protocols. However, for more difficult examples, especially for examples challenging for even the distributed system, the protocols can still become overwhelmingly large.

However, by transferring the process of "forgetting" introduced by TEAMWORK to the level of proof analysis we can alleviate this task. Section 6.3 describes how we can use this feature to ease the handling of large protocols to a point where even the most complex examples we encountered so far can be analyzed quite casually.

7.4 Improved lemma recognition

Most of the advantages listed so far are only visible to the operator of the proof system, not to the recipient of the proof. The important advantage of TEAMWORK with respect to proof presentation, however, is the use of the referees in lemma detection. The outstanding results selected by the referees prior to the team meetings make very good and quite often superior lemmata.

Table 3 shows the percentage of arbitrary steps necessary for the final proof and, in contrast, the percentage of steps selected by the referees as important intermediate results used in this proof. The chosen examples come from a wide variety of domains, but in all cases the selected results are much more likely to be needed for the final proof. Averaged over all example the probability for a selected result to be of use for the proof is more then 160 times higher then for an arbitrary step. In addition to demonstrating the quality of DISCOUNT's referees this result also is a strong indication that the intermediate results play a special role in the proof process and thus are probably suitable as lemmata.

The referees select these results because of their good performance during the completion. Thus a result is chosen on its importance for the complete equational domain. The post mortem criteria from section 4, on the other hand, choose lemmata purely by judging their relevance to the proof at hand. Especially, the more powerful of these criteria use quite complex algorithms than cannot easily be applied to the complete listings.

We can, however, use the suggested intermediate results of the experts to generate lemmata with a more global perspective. While a small number of these results are not suitable as lemmata at all (usually because they represent very simple consequences of the axioms), a larger part does indeed make superior lemmata. Taking the refer-

[3] We did not include the easier examples because they can be solved very quickly by the sequential prover – generally faster than starting the distributed system over the network.

Example	Complete	Extracted	Comment
BoolAssoc	95788	117	The disjunctive operator (`and`) in a Boolean algebra is associative. See appendix B.4.
Cooperation	6879	58	Calculations in cyclic group. Presented in [Pi92].
DeMorgan	181336	159	Proof of one of DeMorgan's laws in an arbitrary boolean algebra. See appendix B.4.
GT7-3	9638	214	Derivation of the associativity axiom from a single equation axiomatization of a group. See [LW92].
Luka1	139178	22	Derive the first of Lucaciewicz's axioms for propositional calculus from Frege's axiomatization of this calculus. See [Ta56].
Luka2	47898	57	Derive the second of Lucaciewicz's axioms (see above).
Luka3	180278	42	Derive the third of Lucaciewicz's axioms (see above).
Lusk5	24738	48	An example using a ternary Boolean algebra - see [LO82].
Lusk6	144099	227	In a ring with $x^3 = x$ the multiplicative operation is Abelian. This was presented as a challenging example in [LO82]. See also appendix B.2.
Sims2	283473	936	Completion of a finite group. See [Si92].
Lattice1	324019	73	An example from the field of lattice ordered groups. See appendix B.4
Lattice2	28124	103	Another problem from the domain of lattice ordered groups. See appendix B.4.
Lattice3	63860	133	Show that each element of a lattice ordered group can be expressed as the product of it's positive and it's negative part. See appendix B.3.
Z22	25508	646	Completion of a large cyclic group. See appendix B.4.

Table 2: Necessary and executed inferences in protocols of distributed proofs

Remark: For examples present in this table but not in table 1 there does not at the moment exist a sequential proof with DISCOUNT. For BoolAssoc, as an example, the sequential prover did 879594 inferences (generating a protocol of 73 Megabytes) before terminating unsuccessfully due to lack of memory.

Example	Protocol size	Arbitrary facts	Referees choice
BoolAssoc	95788	0.12 %	43.5 %
Cooperation	6879	0.8 %	40.0 %
DeMorgan	181336	0.08 %	20.9 %
GT7-3	9638	2.2 %	83.3 %
Luka1	139178	0.015 %	20.0 %
Luka2	47898	0.01 %	33.3 %
Luka3	180278	0.023 %	23.1 %
Lusk5	24738	0.19 %	50.0 %
Lusk6	144099	0.16 %	20.5 %
Sims2	283473	0.33 %	50.0 %
Lattice1	324019	0.01 %	8.2 %
Lattice2	28124	0.37 %	50.0 %
Lattice3	63860	0.2 %	15.1 %
Z22	25508	2.5 %	75.1 %
Weighted Sum	1582627	0.18%	29.6 %

Table 3: Percentage of arbitrary and selected results needed in the found proof

ees suggestions into account does also emulate human behaviour quite well, because humans often base their lemma selection on the global performance of a fact in the respective field, too.

We apply this criterion by simply marking the selected results in the proof protocol. This information can be used as a constant element in a weighted decision, as described in section 4.10, or we can use these results (if they conform to the most basic requirements for lemmata) as the basic building blocks and generate (using other criteria) more lemmata to fill in this skeleton. The results are quite promising, especially if more than one example from the same domain are to be presented. For an example see the proof in appendix B.3.

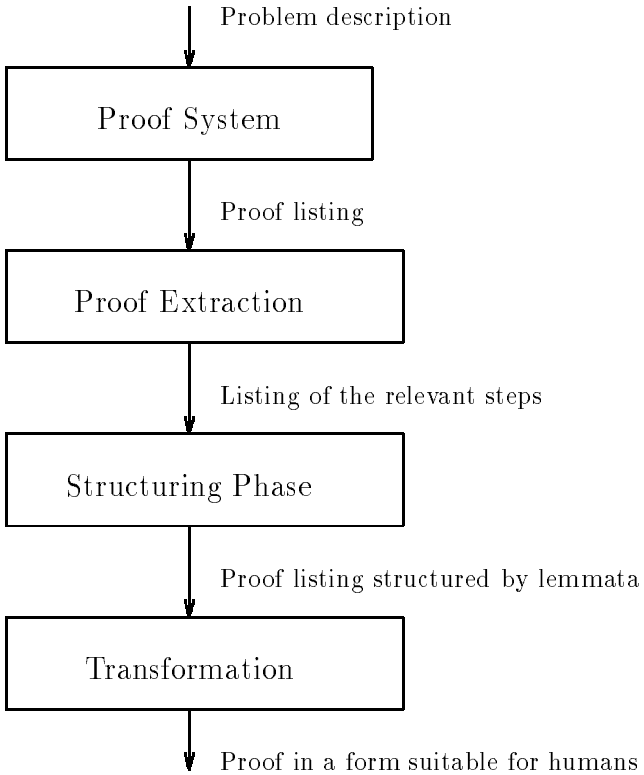

Figure 5: A concept for a system generating, analyzing and transforming proofs

8 Implemented Programs

To achieve our goals with respect to proof analysis and representation we implemented a number of programs. They have first been described in [Sch93] and have been extended to cover some special situations arising with the use of the distributed version of the DISCOUNT system. Additional changes have been made to the proof system itself. This chapter deals with both the extensions to DISCOUNT and the programs dealing with PCL listings. For more information about DISCOUNT see [DP92], for a more extensive description of the programs for proof analysis and transformation consult [Sch93].

Figure 5 shows the basic framework for a system generating and transforming proofs using the concepts described in chapters 3 to 5. A proof is generated from a problem description, yielding a listing of all inference steps. This listing is analyzed and the relevant steps are extracted. Then the proof is structured and finally transformed to a calculus suitable for human consumption.

However, figure 5 is quite abstract and does not take into account the special problems arising with the use of distributed systems. Taking into account the results from chapter 6 we arrive at the structure depicted in figure 6.

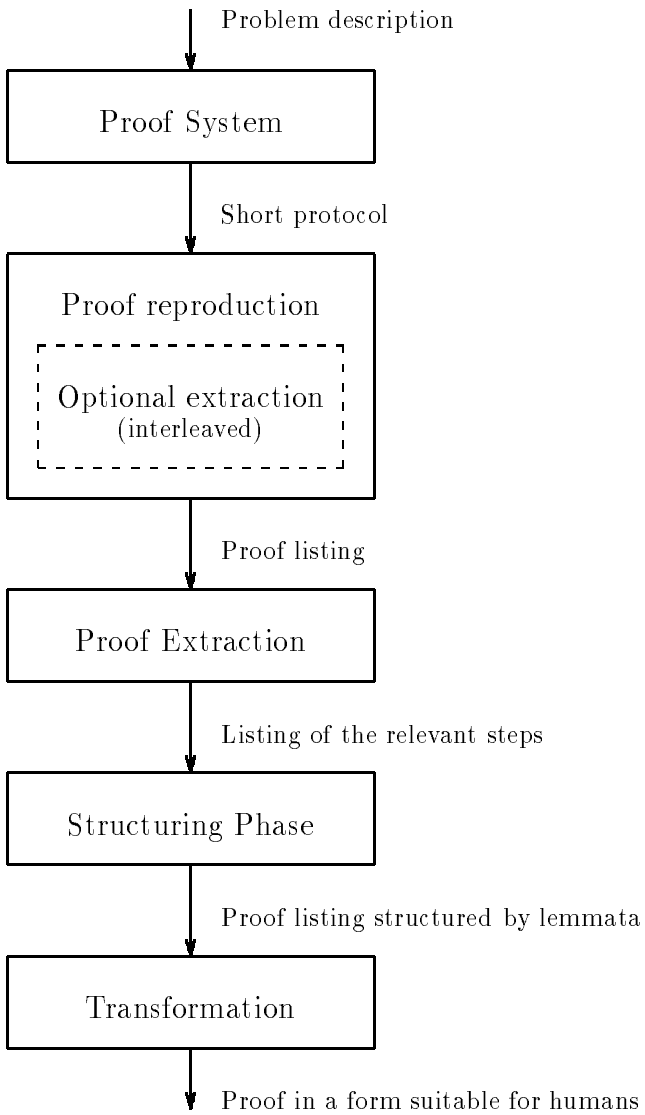

Figure 6: Structure of the implemented system

Our implementation fills the roles in this diagram as follows: The proof (and the accompanying short protocol) is generated by the basic DISCOUNT prover using the executable `discount`. Closely related to this is the program `rpcl` which takes the short protocol (and problem description) of a proof and reproduces it, yielding a PCL protocol of the process.

This PCL listing is extracted by one of a number of programs or program combinations, with `mextract` being the most often used default. Section 8.2 describes these programs in some detail. The programs `lemma` and `proof` deal with proof structuring and transformation, respectively.

8.1 The DISCOUNT system

The DISCOUNT system is a parallel proof system for equational reasoning based on the TEAMWORK method (see [De93]). It is described in detail in [Pi92], the user interface has been documented in [DP92]. An important enhancement of the system is the incorporation of a true broadcast for the transfer of large amounts of data to all members of a team. It lead to a much better performance for configurations with more then two experts and will be described in [Lin93].

The latest improvement so far is the implementation of the protocols necessary to analyze and reproduce the proofs generated by the system. This implementation follows the concepts developed in chapter 6.

It resulted in some small changes to the basic proof system and three new variant programs implementing features not necessary in the plain prover. For consistency reasons and easy maintenance the different versions are build from a common source code using the conditional compilation supported by the C language preprocessor. Which of the four executables shall be build can thus be determined using compile time switches.

The changes in the behaviour of the programs are described in detail in the following sections and will be integrated into upcoming versions of the *DISCOUNT User Guide*. For easy reference the new options available with the different program versions are listed in table 8.1.

8.1.1 Changes in the configuration files

The DISCOUNT system can be tailored to specific problems by the use of configuration files describing the number of processors to use, the experts and specialists to run on them and some additional data of less importance. The `EXEC` keyword was used in the configuration files to specify the name of the executable to be started on the remote processors.

The new versions of the DISCOUNT system discard the `EXEC` entry in the configuration files, although for compatibility reasons it can still be given. Now the master process queries the command line for its own name and uses this as the name of the executable to be started on the remote hosts. This allows the use of the same configuration files with different versions of the DISCOUNT system.

8.1.2 Logging the proof session

The `discount` executable implements our basic proof system for equational proofs based on unfailing completion. There has been only one major change necessary to allow the analysis of proofs generated with the DISCOUNT system.

This is the incorporation of the ability to generate the short protocol mentioned in section 6.1. By default the prover now uses the team meetings to write the data necessary for a later reproduction of the proof. The default filename for this protocol

Shortcut	Option	Effect
	discount	
-l	-no_log	Suppress the production of the short protocol generated by default during the proof session.
-L <name>	-log_file <name>	Select <name> as the filename for the short protocol to be written.
	rdiscount	
-L <name>	-log_file <name>	Select <name> as the filename for the short protocol to be read for the reproduction.
	pcl and rpcl	
-l	-no_log	Suppress the production of a short protocol generated by default during the proof session (only for pcl).
-L <name>	-log_file <name>	Select <name> as the filename for the short protocol to be written or to be read.
-X <method>	-extract <method>	Select the programs to be used for the intermediate extraction of the PCL files. <method> can be one of the following: none — No intermediate extraction at all. mextract — Use mextract for the intermediate extractions. revert — Use revert and rextract. tac — Use tac and rextract.
-F <method>	-fextract <method>	Select the programs to be used for the final extraction of the PCL files. <method> can have the same values as for the preceding option.
-a	-async	Start the final extraction asynchronously. This allows the proof system to terminate and free its resources for the extracting process.

Table 4: The new DISCOUNT programs and their options

is the name of the problem file with the added suffix `.prk`. This can be changed by specifying the option `-log_file <filename>`. The new option `-no_log` suppresses the protocol altogether.

The symmetric design of the DISCOUNT system has been preserved for the task of writing this protocol. As a consequence all `discount` processes in a parallel proof session have to access a single file system, because every process can be responsible for part of this protocol. As the system typically runs on a cluster of workstations with a common NFS *(Network File System)* this assumption is usually fulfilled.

In the other case the protocol file may be shattered among the different file systems. The final version then has to be assembled by the user from the files generated by the different processes. The format of the protocol file allows for this by explicitly stating cycle numbers.

An example of a logged proof session can be found in appendix A.

8.1.3 Reproducing proofs: The `rdiscount` executable

The simple reproduction of a once generated proof session was not one of our main objectives. However, we managed to solve the two problems of reproducing a proof session and generating a PCL listing quite independently. The result of our solution for the reproduction problem is the executable `rdiscount`.

This program reads the short format file described in 8.1.2 and uses the data to reproduce the proof session. It supports the same syntax for configuration and problem files and accepts the same `-log_file` option as `discount`. The same conventions for the names of the protocol files as above apply, however the files are of course only read, not written.

Although `rdiscount` is only a byproduct of our development it still proved to be useful for demonstration and test purposes.

8.1.4 Generating PCL listings: `pcl` and `rpcl`

`pcl` is basically the `discount` executable with added routines for the PCL output. It is, therefore, a full proof system for equational reasoning and implements – except for the additional generation of PCL – the same functionality as `discount`. This does extend to the new features dealing with the short protocol and the options `-log_file <filename>` and `-no_log`. Although it is a pretty powerful system in itself it lacks the full power of the plain `discount` executable for the reasons stated in 6.1. It can be used for small examples, but even for them it is usually more feasible to generate the proof and a short protocol using `discount`. The PCL listing can then be generated using `rpcl`.

`rpcl` combines the new features from `rdiscount` and `pcl`. It can reproduce proofs using only the short protocol from section 8.1.2 and it can produce pcl listings documenting these proofs. As both `pcl` and `rpcl` use the same principles for dealing with the PCL

output they are described together.

The programs use the multidimensional name space conventions described in 6.2. They create one PCL file for each interreduction phase and one PCL file per expert for the working phases. By default they try an extraction on the fly (see 6.3) using `mextract` as the extraction program. The user can select other extraction programs or no extraction at all for large or particularly interesting examples by specifying the option `-extract <method>`. `<method>` can be `none`, indicating that no intermediate extraction has to take place, it can be `mextract` to designate `mextract` as the extraction program to use or it can be one of `revert` and `tac`, designating `rextract` with preprocessing by either `revert` or `tac`.

As the protocol files can become rather large we encountered a few memory problems especially in the final extraction where more then one file have to be extracted. Up to now these have not been solved completely satisfactory. An improvement is the additional option `-async`, which starts the final extraction as an independent background process and thus allows the proof system to terminate and free its resources for the extraction program. We do need to manipulate extreme examples manually, though.

8.2 Programs dealing with PCL protocols

As the final proof description generated by the DISCOUNT system is still a sequential PCL listing of all executed inferences, our tools for proof analysis and transformation were able to deal with distributed proofs immediately. However, to use all the features described in chapter 6 we needed some minor extensions.

The programs share some common features. They have been implemented in ANSI-C, using the GNU-C-Compiler. This results in efficient code and high portability. All of them can deal with input from files (named in the command line) or can be used as a UNIX style filter, reading from `stdin`, possibly connected to a pipeline. Output can be either to a file or to `stdout`.

8.2.1 Extracting the proof: `extract` and `mextract`

The programs `extract` and `mextract` deal with the straightforward extraction of the necessary proof steps. Both implement the algorithm described in section 3.3. In fact, both programs behave exactly alike with only two exceptions. In the design of `mextract`, the younger of the two programs, we could utilize experience made with the earlier version. We therefore used more specialized data structure, resulting in a much faster execution and the capability to deal with examples approximately one order of magnitude larger.

The success of `mextract` made `extract` obsolete and we do no longer develop it. In particularly, it does not support some of the newer options of `mextract`, dealing with statistics and the redundancy elimination process described in section 6.2.1. `extract`'s main value nowadays lies in its role as a simple example for programs using the PCL data structures and parsing routines.

Both programs take a PCL protocol and return a PCL file containing only steps used to arrive at certain results. By default they take only steps with type `tes-final` as anchors for the extraction, but runtime options can change this behaviour.

The programs support the following set of options:

Option	Semantics
`-v`	Give a short explanation of the program and information on the extraction phases.
`-h`	Terminate the program after producing a short description.
`-s`	Print a block of statistic information (steps read, steps extracted, etc..) to `stderr` after termination. Implemented in `mextract` only.
`-c`	Preserve comments in the input file (if possible).
`-i`	Use not only steps with type `tes-final` but also steps with types `tes-intermed`, `tes-intermedgoal` and `crit-intermedgoal` as anchors for the extraction. This option is used for the interleaved extraction described in section 6.3.
`-l`	Use the last step, regardless of type, as an anchor.
`-n`	Eliminate steps proved by a simple reference to an earlier step. Implemented in `mextract` only.
`-n2`	Eliminate redundant results generated more then once during the proof process. Implemented in `mextract` only.
`-o outfile`	Select the output file `outfile`. If no `-o` option is given, output is directed to `stdout`.

8.2.2 Dealing with extreme examples: `revert` and `rextract`

As we already mentioned, proof listings can become overwhelmingly large. For really challenging examples the protocol of proof generated by the sequential system can easily exceed 50 Megabytes. This problem has become less serious with the distributed system because of the interleaved extraction described in section 6.3. However, it is only a matter of time until new examples will yield even greater protocols.

Our extraction algorithm inspects the complete listing, starting with the *final* proof step. The straightforward programs therefore have to handle the complete listing - a task that quickly becomes impossible for extreme examples. We therefore developed the program `rextract`, an alternative implementation of the extraction algorithm expecting the PCL steps in reverse order. As no standard UNIX utility was able to reorder the steps of a large PCL listing at that time (in the meantime the GNU program `tac` has become widely available, solving the same problem) we also implemented `revert`, a program designed to reverse the order of lines in an ASCII text.

`revert` does support the following options:

Option	Semantics
`-v`	Give a short explanation of the program and information on the different phases during the program execution.
`-h`	Terminate the program after producing a short description.
`-o outfile`	Select the output file `outfile`. If no `-o` option is given, output is directed to `stdout`. If the virtual memory of the machine does not suffice for execution `revert` will write a number of files called `outfile.rev*` or `stdout.rev*` which have to concatenated for the final result.

The program `rextract`, serving a very similar purpose to `mextract`, does feature nearly the same options:

Option	Semantics
`-v`	Give a short explanation of the program and information on the extraction phases.
`-h`	Terminate the program after producing a short description.
`-c`	Preserve comments in the input file (if possible).
`-r`	Do immediately print steps recognized as necessary, yielding a reversed extracted listing. By default the program will store these steps and produce an extracted listing with the steps already in the correct sequence.
`-i`	Use not only steps with type `tes-final` but also steps with types `tes-intermed`, `tes-intermedgoal` and `crit-intermedgoal` as anchors for the extraction. This option is used for the interleaved extraction described in section 6.3.
`-l`	Use the last step, regardless of type, as an anchor.
`-o outfile`	Select the output file `outfile`. If no `-o` option is given, output is directed to `stdout`.

8.2.3 Revealing the structure: `lemma`

The program `lemma` implements the structuring algorithm described in section 4.11. Input is a PCL listing[4], output is a listing with important steps marked as `tes-lemma`. It realizes most of the criteria mentioned in chapter 4. Consequently, its configuration can be quite complex. The ability to choose any combination of criteria and influence all constant values in the evaluation function required a lot of different options. They can be split into two classes: Options necessary to set values for the criteria and options used to influence the programs behavior in other ways. The first group of options is displayed in table 5.

[4] `lemma` can handle all kinds of listings, however, extracted listings, being smaller, are analyzed much faster and can be handled for much more difficult problems.

Constant	Default	Option
Constants from section 4.2 (Frequently used steps)		
MINUSED	4	-o_min_used
Constants from section 4.3 (Important intermediate results)		
MINWEIGHT	11	-i_lemma_weight
Constants from section 4.4 (Isolated proof segments)		
WEIGHTFACTOR	0.5	-t_weight_factor
OFFSET	2	-t_offset
Constants from section 4.5 (Syntactical criteria)		
MAXSIZE	1	-s_average_size
AVERAGESIZE	2	-s_max_size
MINFAK	5	-s_min_fak
MAXFAK	0	-s_max_fak
Constants from section 4.7 (Analyzing the applied inference rules)		
INITW	1	-c_init_weight
HYPOW	0	-c_hypo_weight
QUOTEW	0	-c_quot_weight
ORIENTW	0	-c_orient_weight
CPW	3	-c_cp_weight
REDW	2	-c_redu_weight
INSW	0	-c_inst_weight
MINWEIGHT	15	-c_lemma_weight
Constants from section 4.8 (Sectioning long proofs)		
MAXLEN	10	-p_max_length
Constants from section 4.9 (Avoiding unsuitable lemmata)		
MINUSED	1	-u_min_used
MINLEN	2	-u_min_length

Table 5: Constants and options in `lemma`

The second class of options determines more general parts of the program's behaviour. Many of these options are familiar from the other programs. There are, however, two important new options determining the lemma criteria to use and the way to combine them:

Option	Semantics
`-v`	Give a short explanation of the program and information on the extraction phases.

Letter	Criterion / Criteria
o	Frequently used steps (4.2)
i	Important intermediate results (4.3)
t	Isolated proof segments (4.4)
s	Syntactical criteria (4.5)
c	Analyzing the applied inference rules (4.7)
p	Sectioning long proofs (4.8)

Table 6: Letter codes and corresponding criteria for use with `-criteria`

`-h`	Terminate the program after producing a short description.
`-c`	Preserve comments in the input file (if possible). This option will also generate new comments, documenting the lemma evaluation process and the reasons for the final decision.
`-criteria w`	The option `-criteria` selects the criteria to be used in lemma evaluation. The argument `w` is a word containing letters from $\{\mathtt{s}, \mathtt{o}, \mathtt{i}, \mathtt{t}, \mathtt{c}, \mathtt{p}\}$, with each letter denoting on criterion. The correspondence can be found in table 6.
`-iterate`	By default the program will combine different criteria according to method (1) from section 4.10. If this option is set it will use method (2) instead.
`-o outfile`	Select the output file `outfile`. If no `-o` option is given, output is directed to `stdout`.

8.2.4 Generating equational chains: `proof`

The transformation algorithm from section 5.3 has been implemented in the program `proof`. It reads a (possibly prestructured) PCL listing and generates an equivalent proof to the one described in the PCL listing, but using equational chains. These chains can be printed in a variety of styles with different levels of detail. For convenience the program usually ignores lemmata in the input file and generates lemmata by itself (using the default strategy employed by `lemma`).

The output of the program contains a list of axioms and a list of propositions, followed by proof chains for a hierarchy of lemmata and finally the theorems. The proof chains can contain places and substitutions and are either formatted for easy readability of a pure ASCII description or typeset in LaTeX. The behaviour of the program is determined by the following options:

Option	Semantics
`-v`	Give a short explanation of the program and information on the extraction phases.
`-h`	Terminate the program after producing a short description.
`-o outfile`	Select the output file `outfile`. If no `-o` option is given, output is directed to `stdout`.
`-nobrackets`	Print constants in the output without a pair of brackets. Proofs look much cleaner this way, but only naming conventions can be used to distinguish variables and constant. `proof` supports this by using only variables from the set $\{$`x`, `y`, `z`, `u`, `v`, `w`, `p`, `q`, `x0`, `x1`, ...$\}$.
`-nolemmas`	If this option is set, `proof` will refrain from generating new lemmata and will use lemmata in the input file instead.
`-noplace`	Suppress the place designator in the output, yielding a more natural looking proof, but at the cost of loosing some accuracy.
`-nosubst`	Refrain from printing the substitutions necessary for applying equations, with the same effect as above.
`-latex`	Generate a LaTeX description of the proof, suitable for inclusion in a document.

9 Conclusion

Our results show that proof protocols are a suitable base for proof analysis and transformation. The language PCL provides a flexible tool for the description of inference based proofs. Using short, specialized protocols and a reproduction mode we can produce protocols without measurably influencing the proof system even for distributed proof systems using TEAMWORK.

The structuring algorithms developed for use with these protocols are capable of recognizing many important intermediate results. Important results with respect to the proof at hand can be found by a post mortem analysis using only the inferences relevant to the proof, while TEAMWORK's referees can add a more global perspective, judging facts on their performance in the equational domain. The resulting proofs are comparable with proofs structured by humans.

The transformation of structured proof protocols into a hierarchical proof using equational chains yield a proof representation fully adequate for human understanding. Equational proofs represented in this calculus resemble textbook proofs of equational problems.

However, while we are quite satisfied with the results up to now, there still remain some paths for further investigation.

First the proof presentation can be improved in some simple, but significant details. This includes changes to the term representation (using infix notation and dropping some brackets), naming of lemmata and axioms, and merging of repeated applications of "well known" theorems like associativity or commutativity.

Finally we already have transferred back to the prover some knowledge gained from the proof analysis. We hope to gain strong heuristics and significant performance improvements for the prover by further following this path, using either manual analysis or automatic learning procedures. The future may actually see automatic provers used by mathematicians for routine tasks.

A A short log file of a proof session

Here we present a short example of the protocol format used to log proof sessions. Please note that the format is rather compact and depends heavily on the implementation of a given proof system. It does not contain any data on the actual proof but only information on the number of steps and the configurations used by the proof system.

Obviously it is not very useful in analyzing the proof without further information...

```
############################################################
##      DISCOUNT
##
##      Aufgabenstellung:       luka1
##      Konfigurationsdatei:    luka1.cfg.gut
##
##  Hashs are used to mark comments...

cycle 0

master:0
process 0 using configuration 0 (ADD_WEIGHT) did 68 steps
process 1 using configuration 1 (GOALMATCH) did 53 steps


cycle 1

master:0
process 0 using configuration 0 (ADD_WEIGHT) did 48 steps
process 1 using configuration 1 (GOALMATCH) did 16 steps


cycle 2

master:0
process 0 using configuration 0 (ADD_WEIGHT) did 58 steps
process 1 using configuration 1 (GOALMATCH) did 19 steps


cycle 3

master:0
process 0 using configuration 0 (ADD_WEIGHT) did 81 steps
process 1 using configuration 1 (GOALMATCH) did 10 steps


cycle 4
```

```
master:0
process 0 using configuration -1 (NO_CONFIG) did 0 steps
process 1 using configuration 1 (GOALMATCH) did 1 steps

team terminated by process 1 during completion
```

B Examples

In this section we will represent two related problems from the theory of rings and one example from the domain of lattice ordered groups. The first example is of medium difficulty and will be presented in some detail. The second one is a challenging example for equational provers. To our knowledge the DISCOUNT system is the only existing automatic prover capable of generating a proof for this example using pure equational logic without underlying AC-theory. As this example is quite large we will only include the final proof. Both examples have been suggested in [LO82]. Ring theory is a field where humans quite often reason using a (semi-)formal equational calculus. Therefore automatically generated proofs in this domain are comparable to proofs found by humans. This is also true for the last example, a hard problem from the domain of lattice ordered groups.

B.1 A ring with $x^2 = x$ is Abelian

B.1.1 The problem

The problem description printed below provides the specification in the format used by DISCOUNT.

```
MODE        PROOF

NAME        Lusk3

ORDERING    XKBO
            f:5 > j:4 > g:3 > 0:1 > b:1 > a:1

EQUATIONS   j (0,x)       = x                  # 0 is a left identity
            j (x,0)       = x                  # 0 is a right identity
            j (g (x),x)   = 0                  # There is a left inverse
            j (x,g (x))   = 0                  # There is a right inverse
            j (j (x,y),z) = j (x,j (y,z))      # Addition is associative
            j (x,y)       = j(y,x)             # Addition is Abelian
            f (f (x,y),z) = f (x,f (y,z))      # Multiplication is
                                               # associative
            f (x,j (y,z)) = j (f (x,y),f (x,z)) # Distributive axioms
```

```
            f (j (x,y),z) = j (f (x,z),f (y,z)) #
            f (x,x)       = x                   # Special axiom: x*x = x

CONCLUSION  f (a,b) = f (b,a)                   # Theorem
```

B.1.2 The proof protocol

The prover generates a protocol of 5009 steps, using about 360 Kilobytes. For obvious reasons we print only the extracted version. It contains only 83 steps.

```
    0 : tes-eqn : f(x,x) = x : initial
    1 : tes-eqn : j(0(),x) = x : initial
    2 : tes-eqn : j(x,0()) = x : initial
    4 : tes-eqn : j(x,g(x)) = 0() : initial
    5 : tes-eqn : j(x,y) = j(y,x) : initial
    6 : tes-eqn : j(j(x,y),z) = j(x,j(y,z)) : initial
    8 : tes-eqn : f(x,j(y,z)) = j(f(x,y),f(x,z)) : initial
    9 : tes-eqn : f(j(x,y),z) = j(f(x,z),f(y,z)) : initial
   10 : tes-goal : f(a(),b()) = f(b(),a()) : hypothesis
   11 : tes-rule : f(x,x) -> x : orient(0,u)
   12 : tes-rule : j(0(),x) -> x : orient(1,u)
   13 : tes-rule : j(x,0()) -> x : orient(2,u)
   20 : tes-rule : j(x,g(x)) -> 0() : orient(4,u)
   45 : tes-rule : j(j(x,y),z) -> j(x,j(y,z)) : orient(6,u)
   53 : tes-eqn : j(x,j(g(x),y)) = j(0(),y) : cp(45,L.1,20,L)
   54 : tes-eqn : j(x,j(g(x),y)) = y : tes-red(53,R,12,L)
   59 : tes-eqn : j(x,j(y,z)) = j(y,j(z,x)) : cp(5,L,45,L)
   65 : tes-rule : j(x,j(g(x),y)) -> y : orient(54,u)
   71 : tes-eqn : g(g(x)) = j(x,0()) : cp(65,L.2,20,L)
   72 : tes-eqn : g(g(x)) = x : tes-red(71,R,13,L)
   87 : tes-eqn : x = j(y,j(x,g(y))) : cp(65,L.2,5,L)
   89 : tes-rule : g(g(x)) -> x : orient(72,u)
   93 : tes-eqn : x = j(g(y),j(y,x)) : cp(65,L.2.1,89,L)
   97 : tes-rule : j(x,j(y,g(x))) -> y : orient(87,x)
  115 : tes-eqn : x = j(g(y),j(x,y)) : cp(97,L.2.2,89,L)
  126 : tes-rule : j(g(x),j(x,y)) -> y : orient(93,x)
  160 : tes-rule : j(g(x),j(y,x)) -> y : orient(115,x)
  181 : tes-eqn : g(x) = j(g(j(x,y)),y) : cp(160,L.2,126,L)
  182 : tes-eqn : g(x) = j(y,g(j(x,y))) : tes-red(181,R,5,L)
  298 : tes-rule : j(x,g(j(y,x))) -> g(y) : orient(182,x)
 2615 : tes-rule : j(f(x,y),f(x,z)) -> f(x,j(y,z)) : orient(8,x)
 2616 : tes-eqn : f(x,j(x,y)) = j(x,f(x,y)) : cp(2615,L.1,11,L)
 2617 : tes-eqn : f(x,j(y,x)) = j(f(x,y),x) : cp(2615,L.2,11,L)
 2618 : tes-eqn : f(x,j(y,x)) = j(x,f(x,y)) : tes-red(2617,R,5,L)
 2656 : tes-rule : f(x,j(x,y)) -> j(x,f(x,y)) : orient(2616,u)
 2669 : tes-eqn : j(x,f(x,0())) = f(x,x) : cp(2656,L.2,13,L)
 2670 : tes-eqn : j(x,f(x,0())) = x : tes-red(2669,R,11,L)
 2707 : tes-rule : j(x,f(x,0())) -> x : orient(2670,u)
 2721 : tes-eqn : f(g(x),0()) = j(x,g(x)) : cp(65,L.2,2707,L)
 2722 : tes-eqn : f(g(x),0()) = 0() : tes-red(2721,R,20,L)
 2767 : tes-rule : f(g(x),0()) -> 0() : orient(2722,u)
 2788 : tes-eqn : 0() = f(x,0()) : cp(2767,L.1,89,L)
```

```
2791 : tes-rule : f(x,0()) -> 0() : orient(2788,x)
2859 : tes-rule : f(x,j(y,x)) -> j(x,f(x,y)) : orient(2618,u)
2883 : tes-eqn : j(g(x),f(g(x),x)) = f(g(x),0()) : cp(2859,L.2,20,L)
2884 : tes-eqn : j(g(x),f(g(x),x)) = 0() : tes-red(2883,R,2791,L)
2944 : tes-rule : j(g(x),f(g(x),x)) -> 0() : orient(2884,u)
2961 : tes-eqn : f(g(x),x) = j(x,0()) : cp(65,L.2,2944,L)
2962 : tes-eqn : f(g(x),x) = x : tes-red(2961,R,13,L)
3044 : tes-rule : f(g(x),x) -> x : orient(2962,u)
3464 : tes-rule : j(f(x,y),f(z,y)) -> f(j(x,z),y) : orient(9,x)
3465 : tes-eqn : f(j(x,y),x) = j(x,f(y,x)) : cp(3464,L.1,11,L)
3466 : tes-eqn : f(j(x,y),y) = j(f(x,y),y) : cp(3464,L.2,11,L)
3467 : tes-eqn : f(j(x,y),y) = j(y,f(x,y)) : tes-red(3466,R,5,L)
3610 : tes-rule : f(j(x,y),x) -> j(x,f(y,x)) : orient(3465,u)
3630 : tes-eqn : j(x,f(0(),x)) = f(x,x) : cp(3610,L.1,13,L)
3631 : tes-eqn : j(x,f(0(),x)) = x : tes-red(3630,R,11,L)
3632 : tes-eqn : j(x,f(g(x),x)) = f(0(),x) : cp(3610,L.1,20,L)
3633 : tes-eqn : j(x,x) = f(0(),x) : tes-red(3632,L.2,3044,L)
3682 : tes-rule : j(x,f(0(),x)) -> x : orient(3631,u)
3867 : tes-eqn : x = j(x,j(x,x)) : cp(3682,L.2,3633,R)
3943 : tes-rule : j(x,j(x,x)) -> x : orient(3867,x)
3983 : tes-eqn : g(x) = j(j(x,x),g(x)) : cp(298,L.2.1,3943,L)
3984 : tes-eqn : g(x) = j(g(x),j(x,x)) : tes-red(3983,R,5,L)
3985 : tes-eqn : g(x) = j(x,j(x,g(x))) : tes-red(3984,R,59,L)
3986 : tes-eqn : g(x) = j(x,0()) : tes-red(3985,R.2,20,L)
3987 : tes-eqn : g(x) = x : tes-red(3986,R,13,L)
4117 : tes-rule : g(x) -> x : orient(3987,u)
4132 : tes-eqn : j(x,j(x,y)) = y : tes-red(65,L.2.1,4117,L)
4259 : tes-rule : j(x,j(x,y)) -> y : orient(4132,u)
4263 : tes-eqn : j(j(x,y),f(j(x,y),x)) = f(j(x,y),y) : cp(2859,L.2,4259,L)
4264 : tes-eqn : j(j(x,y),j(x,f(y,x))) = f(j(x,y),y) : tes-red(4263,L.2,3610,L)
4265 : tes-eqn : j(x,j(f(y,x),j(x,y))) = f(j(x,y),y) : tes-red(4264,L,59,L)
4266 : tes-eqn : j(x,j(j(x,y),f(y,x))) = f(j(x,y),y) : tes-red(4265,L.2,5,L)
4267 : tes-eqn : j(x,j(x,j(y,f(y,x)))) = f(j(x,y),y) : tes-red(4266,L.2,45,L)
4268 : tes-eqn : j(x,f(x,y)) = f(j(y,x),x) : tes-red(4267,L,4259,L)
4435 : tes-rule : f(j(x,y),y) -> j(y,f(x,y)) : orient(3467,u)
4800 : tes-eqn : j(x,f(x,y)) = j(x,f(y,x)) : tes-red(4268,R,4435,L)
4834 : tes-eqn : j(x,f(j(x,y),x)) = j(x,j(x,f(x,y))) : cp(4800,L.2,2656,L)
4835 : tes-eqn : j(x,j(x,f(y,x))) = j(x,j(x,f(x,y))) : tes-red(4834,L.2,3610,L)
4836 : tes-eqn : f(x,y) = j(y,j(y,f(y,x))) : tes-red(4835,L,4259,L)
4837 : tes-eqn : f(x,y) = f(y,x) : tes-red(4836,R,4259,L)
5008 : tes-final : f(a(),b()) = f(b(),a()) : instance(10,4837)
```

B.1.3 Lemmata

If we use the program `lemma` with the default settings (or directly pipe the protocol into `proof`) the following steps are recognized as important lemmata:

```
  65 : tes-lemma : j(x,j(g(x),y)) -> y : orient(54,u)
  89 : tes-lemma : g(g(x)) -> x : orient(72,u)
2884 : tes-lemma : j(g(x),f(g(x),x)) = 0() : tes-red(2883,R,2791,L)
3610 : tes-lemma : f(j(x,y),x) -> j(x,f(y,x)) : orient(3465,u)
3867 : tes-lemma : x = j(x,j(x,x)) : cp(3682,L.2,3633,R)
3985 : tes-lemma : g(x) = j(x,j(x,g(x))) : tes-red(3984,R,59,L)
```

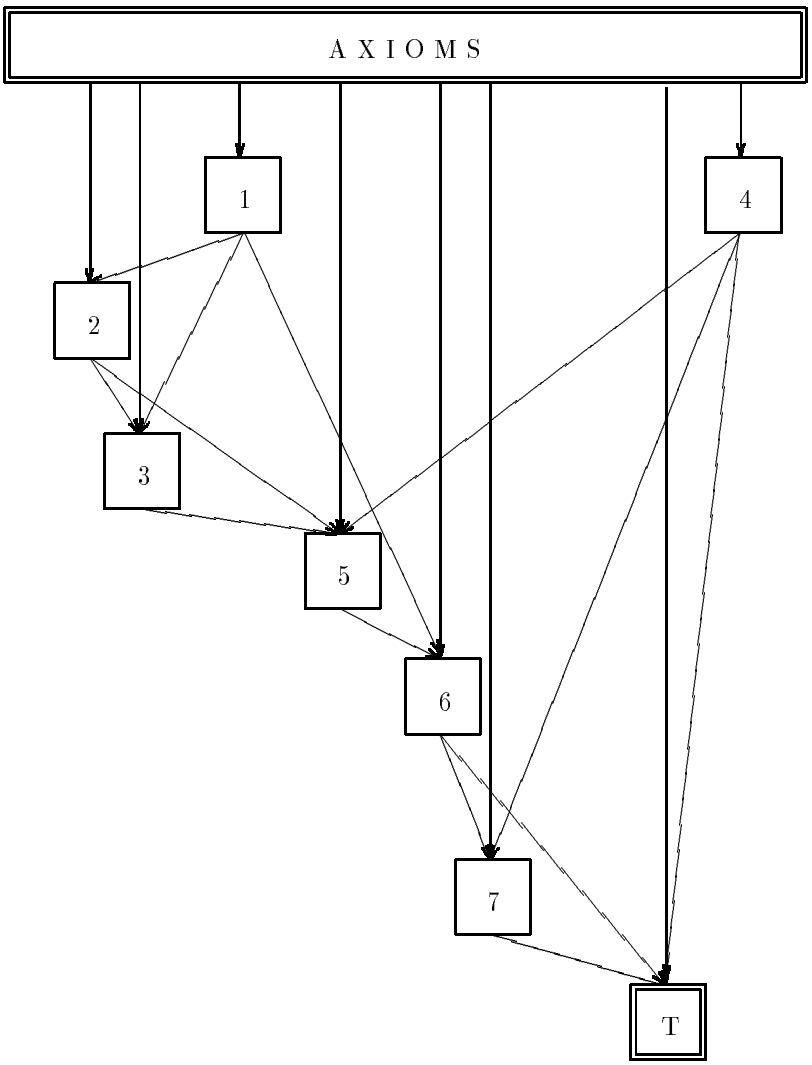

Figure 7: Lemma structure according to B.1.3

```
4259 : tes-lemma : j(x,j(x,y)) -> y : orient(4132,u)
4267 : tes-lemma : j(x,j(x,j(y,f(y,x)))) = f(j(x,y),y) : tes-red(4266,L.2,45,L)
4836 : tes-lemma : f(x,y) = j(y,j(y,f(y,x))) : tes-red(4835,L,4259,L)
```

The dependencies of the lemmata are depicted in figure 7. The numbers correspond to the proof as printed in the next section, the theorem is marked "T".

B.1.4 The proof

To arrive at a readable proof we have used the program `proof` on the above proof listing. We used the `-latex` option to produce output in LaTeX. Please note that the terms going to be replaced (affected by the next operation) are marked by underlining,

while inserted terms (affected by the last operation) are set in bold face. We refrained from printing the substitutions to keep the proof more readable.

Consider the following set of axioms:

Axiom 1: f(x, x) = x
Axiom 2: j(0, x) = x
Axiom 3: j(x, 0) = x
Axiom 4: j(x, g(x)) = 0
Axiom 5: j(x, y) = j(y, x)
Axiom 6: j(j(x, y), z) = j(x, j(y, z))
Axiom 7: f(x, j(y, z)) = j(f(x, y), f(x, z))
Axiom 8: f(j(x, y), z) = j(f(x, z), f(y, z))

This theorem holds true:

Theorem 1: f(a, b) = f(b, a)

Proof:

Lemma 1: j(u, j(g(u), z)) = z

j(u, j(g(u), z))	=	**j(j(u, g(u)), z)**	by Axiom 6 RL
	=	j(**0**, z)	by Axiom 4 LR
	=	**z**	by Axiom 2 LR

Lemma 2: g(g(u)) = u

g(g(u))	=	**j(u, j(g(u), g(g(u))))**	by Lemma 1 RL
	=	j(u, **0**)	by Axiom 4 LR
	=	**u**	by Axiom 3 LR

Lemma 3: j(g(z), f(g(z), z)) = 0

j(g(z), f(g(z), z))	=	**j(f(g(z), z), g(z))**	by Axiom 5 RL
	=	j(f(g(z), z), **f(g(z), g(z)))**	by Axiom 1 RL
	=	**f(g(z), j(z, g(z)))**	by Axiom 7 RL
	=	f(g(z), **0**)	by Axiom 4 LR
	=	f(**g(g(g(z)))**, 0)	by Lemma 2 RL
	=	**j(g(g(z)), j(g(g(g(z))), f(g(g(g(z))), 0)))**	by Lemma 1 RL
	=	j(g(g(z)), j(**f(g(g(g(z)))), g(g(g(z))))**, f(g(g(g(z))), 0)))	by Axiom 1 RL
	=	j(g(g(z)), **f(g(g(g(z))), j(g(g(g(z))), 0)**))	by Axiom 7 RL
	=	j(g(g(z)), f(g(g(g(z))), **g(g(g(z))))**)	by Axiom 3 LR
	=	j(g(g(z)), **g(g(g(z))))**	by Axiom 1 LR
	=	**0**	by Axiom 4 LR

Lemma 4: f(j(v, y), v) = j(v, f(y, v))

f(j(v, y), v)	=	**j(f(v, v), f(y, v))**	by Axiom 8 LR
	=	j(**v**, f(y, v))	by Axiom 1 LR

Lemma 5: g(y) = j(j(y, y), g(y))

g(y)	=	**j(g(j(y, j(y, y))), j(g(g(j(y, j(y, y)))), g(y)))**	by Lemma 1 RL
	=	j(g(j(y, j(y, y))), **j(g(y), g(g(j(y, j(y, y))))))**	by Axiom 5 LR
	=	j(g(j(y, j(y, y))), j(g(y), **j(y, j(y, y))**))	by Lemma 2 LR
	=	j(g(j(y, j(y, y))), j(g(y), j(**g(g(y))**, j(y, y))))	by Lemma 2 RL
	=	j(g(j(y, j(y, y))), **j(y, y)**)	by Lemma 1 LR
	=	**j(j(y, y), g(j(y, j(y, y))))**	by Axiom 5 LR
	=	j(j(y, y), g(j(y, j(y, **j(y, 0)**))))	by Axiom 3 RL
	=	j(j(y, y), g(j(y, j(y, j(y, **j(g(y), f(g(y), y))**))))	by Lemma 3 RL
	=	j(j(y, y), g(j(y, j(y, **f(g(y), y)**))))	by Lemma 1 LR
	=	j(j(y, y), g(j(y, **f(j(y, g(y)), y)**)))	by Lemma 4 RL
	=	j(j(y, y), g(j(y, f(**0**, y))))	by Axiom 4 LR
	=	j(j(y, y), g(**f(j(y, 0), y)**))	by Lemma 4 RL
	=	j(j(y, y), g(f(**y**, y)))	by Axiom 3 LR
v	=	j(j(y, y), g(**y**))	by Axiom 1 LR

Lemma 6: j(u, j(u, z)) = z

j(u, j(u, z))	=	j(u, j(**j(u, 0)**, z))	by Axiom 3 RL
	=	j(u, j(j(u, **j(u, g(u))**), z))	by Axiom 4 RL
	=	j(u, j(**j(j(u, u), g(u))**, z))	by Axiom 6 RL
	=	j(u, j(**g(u)**, z))	by Lemma 5 RL
	=	**z**	by Lemma 1 LR

Lemma 7: j(z, f(z, w)) = j(z, f(w, z))

j(z, f(z, w))	=	**j(w, j(w, j(z, f(z, w))))**	by Lemma 6 RL
	=	j(w, **j(j(w, z), f(z, w))**)	by Axiom 6 RL
	=	j(w, **j(f(z, w), j(w, z))**)	by Axiom 5 RL
	=	**j(j(w, f(z, w)), j(w, z))**	by Axiom 6 RL
	=	**j(j(w, z), j(w, f(z, w)))**	by Axiom 5 LR
	=	j(j(w, z), **f(j(w, z), w)**)	by Lemma 4 RL
	=	**j(f(j(w, z), w), j(w, z))**	by Axiom 5 RL
	=	j(f(j(w, z), w), **f(j(w, z), j(w, z))**)	by Axiom 1 RL
	=	**f(j(w, z), j(w, j(w, z)))**	by Axiom 7 RL
	=	f(j(w, z), **z**)	by Lemma 6 LR
	=	**j(f(w, z), f(z, z))**	by Axiom 8 LR
	=	j(f(w, z), **z**)	by Axiom 1 LR
	=	**j(z, f(w, z))**	by Axiom 5 LR

Theorem 1: f(a, b) = f(b, a)

f(a, b)	=	**j(b, j(b, f(a, b)))**	by Lemma 6 RL
	=	j(b, **f(j(b, a), b)**)	by Lemma 4 RL
	=	**j(b, f(b, j(b, a)))**	by Lemma 7 RL
	=	j(b, **j(f(b, b), f(b, a))**)	by Axiom 7 LR
	=	j(b, j(**b**, f(b, a)))	by Axiom 1 LR
	=	**f(b, a)**	by Lemma 6 LR

B.2 A ring with $x^3 = x$ is Abelian

This problem is very similar to the last one. However, albeit the specifications seem to be nearly identical, this example is considered as a particularly challenging example for pure equational reasoning. The first known automatic proof for this problem by a prover *not* employing a build-in AC-theory was found by the DISCOUNT system in a sequential run on a SUN4/370 server. It took 8188 seconds (roughly two and one quarter hours) and was published in [Pi92]. In the meantime this problem can be solved by teams using two or three SUN ELC workstations[5] in approximately 300 seconds.

B.2.1 The Problem

```
MODE        PROOF

NAME        lusk6

ORDERING    XKBO
            f:5 > j:4 > g:3 > 0:1 > b:1 > a:1

EQUATIONS   j (0,x)       = x                  # 0 is a left identity
            j (x,0)       = x                  # 0 is a right identity
            j (g (x),x)   = 0                  # there is a left inverse
            j (x,g (x))   = 0                  # there is a right inverse
            j (j (x,y),z) = j (x,j (y,z))      # associativity of addition
            j (x,y)       = j(y,x)             # commutativity of addition
            f (f (x,y),z) = f (x,f (y,z))      # associativity of
                                               # multiplication
            f (x,j (y,z)) = j (f (x,y),f (x,z)) # distributivity axioms
            f (j (x,y),z) = j (f (x,z),f (y,z)) #
            f (f(x,x),x) = x                   # special hypothesis

CONCLUSION  f (a,b) = f (b,a)                  # theorem
```

[5] For better comparison: We reproduced the original sequential proof on a SUN ELC workstation in 5153 seconds.

B.2.2 The proof

As we already mentioned above we will only print the final proof. The proof listing for this problem can be very long - the sequential version, which we present here, contains nearly 400000 steps and has a size of nearly 50 Megabytes. However, the extracted listing uses only 190 steps.

Consider the following set of axioms:

Axiom 1: j(0, x) = x
Axiom 2: j(x, 0) = x
Axiom 3: j(x, g(x)) = 0
Axiom 4: j(j(x, y), z) = j(x, j(y, z))
Axiom 5: j(x, y) = j(y, x)
Axiom 6: f(f(x, y), z) = f(x, f(y, z))
Axiom 7: f(x, j(y, z)) = j(f(x, y), f(x, z))
Axiom 8: f(j(x, y), z) = j(f(x, z), f(y, z))
Axiom 9: f(f(x, x), x) = x

This theorem holds true:

Theorem 1: f(a, b) = f(b, a)

Proof:

Lemma 1: j(u, j(g(u), z)) = z

j(u, j(g(u), z))	=	j(j(u, g(u)), z)	by Axiom 4 RL
	=	j(0, z)	by Axiom 3 LR
	=	z	by Axiom 1 LR

Lemma 2: g(g(u)) = u

g(g(u))	=	j(u, j(g(u), g(g(u))))	by Lemma 1 RL
	=	j(u, 0)	by Axiom 3 LR
	=	u	by Axiom 2 LR

Lemma 3: j(v, g(j(w, v))) = g(w)

j(v, g(j(w, v)))	=	j(g(j(w, v)), v)	by Axiom 5 RL
	=	j(g(j(w, v)), j(g(w), j(g(g(w)), v)))	by Lemma 1 RL
	=	j(g(j(w, v)), j(g(w), j(w, v)))	by Lemma 2 LR
	=	j(g(j(w, v)), j(g(w), g(g(j(w, v)))))	by Lemma 2 RL
	=	j(g(j(w, v)), j(g(g(j(w, v))), g(w)))	by Axiom 5 RL
	=	g(w)	by Lemma 1 LR

Lemma 4: f(y, 0) = 0

f(y, 0)	=	f(g(g(y)), 0)	by Lemma 2 RL
	=	j(g(y), j(g(g(y)), f(g(g(y)), 0)))	by Lemma 1 RL
	=	j(g(y), j(f(f(g(g(y)), g(g(y))), g(g(y))), f(g(g(y)), 0)))	by Axiom 9 RL
	=	j(g(y), j(f(g(g(y)), f(g(g(y)), g(g(y)))), f(g(g(y)), 0)))	by Axiom 6 LR
	=	j(g(y), f(g(g(y)), j(f(g(g(y)), g(g(y))), 0)))	by Axiom 7 RL
	=	j(g(y), f(g(g(y)), f(g(g(y)), g(g(y)))))	by Axiom 2 LR
	=	j(g(y), f(f(g(g(y)), g(g(y))), g(g(y))))	by Axiom 6 RL
	=	j(g(y), g(g(y)))	by Axiom 9 LR
	=	0	by Axiom 3 LR

Lemma 5: f(0, y) = 0

f(0, y)	=	f(0, j(y, 0))	by Axiom 2 RL
	=	f(0, j(y, j(y, g(y))))	by Axiom 3 RL
	=	f(0, j(j(y, y), g(y)))	by Axiom 4 RL
	=	j(f(0, j(y, y)), f(0, g(y)))	by Axiom 7 LR
	=	j(j(f(0, y), f(0, y)), f(0, g(y)))	by Axiom 7 LR
	=	j(f(j(0, 0), y), f(0, g(y)))	by Axiom 8 RL
	=	j(f(0, y), f(0, g(y)))	by Axiom 1 LR
	=	f(0, j(y, g(y)))	by Axiom 7 RL
	=	f(0, 0)	by Axiom 3 LR
	=	0	by Lemma 4 LR

Lemma 6: f(z, f(y, f(z, f(y, f(z, y))))) = f(z, y)

f(z, f(y, f(z, f(y, f(z, y)))))	=	f(z, f(y, f(f(z, y), f(z, y))))	by Axiom 6 RL
	=	f(f(z, y), f(f(z, y), f(z, y)))	by Axiom 6 RL
	=	f(f(f(z, y), f(z, y)), f(z, y))	by Axiom 6 RL
	=	f(z, y)	by Axiom 9 LR

Lemma 7: f(p, j(f(p, f(p, v)), z)) = f(p, j(v, z))

f(p, j(f(p, f(p, v)), z))	=	j(f(p, f(p, f(p, v))), f(p, z))	by Axiom 7 LR
	=	j(f(p, f(f(p, p), v)), f(p, z))	by Axiom 6 RL
	=	j(f(f(p, f(p, p)), v), f(p, z))	by Axiom 6 RL
	=	j(f(f(f(p, p), p), v), f(p, z))	by Axiom 6 RL
	=	j(f(p, v), f(p, z))	by Axiom 9 LR
	=	f(p, j(v, z))	by Axiom 7 RL

Lemma 8: f(j(v, g(f(x, f(x, v)))), x) = 0

f(j(v, g(f(x, f(x, v)))), x)

= f(j(v, g(f(x, f(x, v)))), f(x, f(j(v, g(f(x, f(x, v)))), f(x, f(j(v, g(f(x, f(x, v)))), x)))))
by Lemma 6 RL

= f(j(v, g(f(x, f(x, v)))), f(f(x, j(v, g(f(x, f(x, v))))), f(x, f(j(v, g(f(x, f(x, v)))), x))))
by Axiom 6 RL

= f(j(v, g(f(x, f(x, v)))), f(f(x, j(f(x, f(x, v)), g(f(x, f(x, v))))), f(x, f(j(v, g(f(x, f(x, v)))), x))))
by Lemma 7 RL

= f(j(v, g(f(x, f(x, v)))), f(f(x, 0), f(x, f(j(v, g(f(x, f(x, v)))), x))))
by Axiom 3 LR

= f(j(v, g(f(x, f(x, v)))), f(0, f(x, f(j(v, g(f(x, f(x, v)))), x))))
by Lemma 4 LR

= f(j(v, g(f(x, f(x, v)))), 0)
by Lemma 5 LR

= 0
by Lemma 4 LR

Lemma 9: f(z, g(u)) = g(f(z, u))

f(z, g(u))	=	j(0, f(z, g(u)))	by Axiom 1 RL
	=	j(f(z, g(u)), 0)	by Axiom 5 RL
	=	j(f(z, g(u)), j(0, g(0)))	by Axiom 3 RL
	=	j(f(z, g(u)), g(0))	by Axiom 1 LR
	=	j(f(z, g(u)), g(f(z, 0)))	by Lemma 4 RL
	=	j(f(z, g(u)), g(f(z, j(u, g(u)))))	by Axiom 3 RL
	=	j(f(z, g(u)), g(j(f(z, u), f(z, g(u)))))	by Axiom 7 LR
	=	g(f(z, u))	by Lemma 3 LR

Lemma 10: f(g(u), y) = g(f(u, y))

f(g(u), y)	=	j(0, f(g(u), y))	by Axiom 1 RL
	=	j(f(g(u), y), 0)	by Axiom 5 RL
	=	j(f(g(u), y), j(0, g(0)))	by Axiom 3 RL
	=	j(f(g(u), y), g(0))	by Axiom 1 LR
	=	j(f(g(u), y), g(f(0, y)))	by Lemma 5 RL
	=	j(f(g(u), y), g(f(j(u, g(u)), y)))	by Axiom 3 RL
	=	j(f(g(u), y), g(j(f(u, y), f(g(u), y))))	by Axiom 8 LR
	=	g(f(u, y))	by Lemma 3 LR

Lemma 11: g(f(u, w)) = f(w, f(w, g(f(u, w))))

g(f(u, w))	=	j(f(g(f(w, f(w, u))), w), g(j(f(u, w), f(g(f(w, f(w, u))), w))))	by Lemma 3 RL
	=	j(f(g(f(w, f(w, u))), w), g(f(j(u, g(f(w, f(w, u)))), w)))	by Axiom 8 RL
	=	j(f(g(f(w, f(w, u))), w), g(0))	by Lemma 8 LR
	=	j(f(f(w, g(f(w, u))), w), g(0))	by Lemma 9 RL
	=	j(f(w, f(g(f(w, u)), w)), g(0))	by Axiom 6 LR
	=	j(f(w, f(f(w, g(u)), w)), g(0))	by Lemma 9 RL
	=	j(f(w, f(w, f(g(u), w))), g(0))	by Axiom 6 LR
	=	j(f(w, f(w, f(g(u), w))), j(0, g(0)))	by Axiom 1 RL
	=	j(f(w, f(w, f(g(u), w))), 0)	by Axiom 3 LR
	=	j(0, f(w, f(w, f(g(u), w))))	by Axiom 5 LR
	=	f(w, f(w, f(g(u), w)))	by Axiom 1 LR
	=	f(w, f(w, g(f(u, w))))	by Lemma 10 LR

Lemma 12: f(v, f(v, f(z, v))) = f(z, v)

f(v, f(v, f(z, v)))	=	g(g(f(v, f(v, f(z, v)))))	by Lemma 2 RL
	=	g(f(v, g(f(v, f(z, v)))))	by Lemma 9 RL
	=	g(f(v, f(v, g(f(z, v)))))	by Lemma 9 RL
	=	g(g(f(z, v)))	by Lemma 11 RL
	=	f(z, v)	by Lemma 2 LR

Lemma 13: f(y, w) = f(w, f(y, f(w, f(y, f(w, f(w, f(y, w)))))))

f(y, w) = f(y, f(w, f(y, f(w, f(y, w)))))
by Lemma 6 RL

= f(f(w, f(y, f(w, f(y, w)))), f(f(w, f(y, f(w, f(y, w)))), f(y, f(w, f(y, f(w, f(y, w)))))))
by Lemma 12 RL

= f(f(w, f(y, f(w, f(y, w)))), f(f(w, f(y, f(w, f(y, w)))), f(y, w)))
by Lemma 6 LR

= f(f(w, f(y, f(w, f(y, w)))), f(w, f(f(y, f(w, f(y, w))), f(y, w))))
by Axiom 6 LR

= f(f(w, f(y, f(w, f(y, w)))), f(w, f(y, f(f(w, f(y, w)), f(y, w)))))
by Axiom 6 LR

= f(f(w, f(y, f(w, f(y, w)))), f(w, f(y, f(w, f(f(y, w), f(y, w))))))
by Axiom 6 LR

= f(f(w, f(y, f(w, f(y, w)))), f(w, f(y, f(w, f(y, f(w, f(y, w)))))))
by Axiom 6 LR

= f(f(w, f(y, f(w, f(y, w)))), f(w, f(y, w)))
by Lemma 6 LR

= f(w, f(f(y, f(w, f(y, w))), f(w, f(y, w))))
by Axiom 6 LR

= f(w, f(y, f(f(w, f(y, w)), f(w, f(y, w)))))
by Axiom 6 LR

= f(w, f(y, f(w, f(f(y, w), f(w, f(y, w))))))
by Axiom 6 LR

= f(w, f(y, f(w, f(y, f(w, f(w, f(y, w)))))))
by Axiom 6 LR

Lemma 14: f(j(p, f(p, g(f(x, x)))), x) = f(p, j(g(x), f(f(p, p), x)))

f(j(p, f(p, g(f(x, x)))), x) = f(j(f(f(p, p), p), f(p, g(f(x, x)))), x)
by Axiom 9 RL
= f(j(f(p, f(p, p)), f(p, g(f(x, x)))), x)
by Axiom 6 LR
= f(f(p, j(f(p, p), g(f(x, x)))), x)
by Axiom 7 RL
= f(p, f(j(f(p, p), g(f(x, x))), x))
by Axiom 6 LR
= f(p, f(j(f(p, p), j(g(f(p, p)), g(j(f(x, x), g(f(p, p)))))), x))
by Lemma 3 RL
= f(p, f(g(j(f(x, x), g(f(p, p)))), x))
by Lemma 1 LR
= f(p, g(f(j(f(x, x), g(f(p, p))), x)))
by Lemma 10 LR
= f(p, g(j(f(f(x, x), x), f(g(f(p, p)), x))))
by Axiom 8 LR
= f(p, g(j(x, f(g(f(p, p)), x))))
by Axiom 9 LR
= f(p, g(j(x, g(f(f(p, p), x)))))
by Lemma 10 LR
= f(p, j(f(f(p, p), x), j(g(f(f(p, p), x)), g(j(x, g(f(f(p, p), x)))))))
by Lemma 1 RL
= f(p, j(f(f(p, p), x), g(x)))
by Lemma 3 LR
= f(p, j(g(x), f(f(p, p), x)))
by Axiom 5 LR

Lemma 15: f(j(p, g(f(p, f(x, x)))), x) = 0

f(j(p, g(f(p, f(x, x)))), x)	=	f(j(p, f(p, g(f(x, x)))), x)	by Lemma 9 RL
	=	f(p, j(g(x), f(f(p, p), x)))	by Lemma 14 LR
	=	f(p, j(g(x), f(p, f(p, x))))	by Axiom 6 LR
	=	j(f(p, g(x)), f(p, f(p, f(p, x))))	by Axiom 7 LR
	=	j(f(p, g(x)), f(p, f(f(p, p), x)))	by Axiom 6 RL
	=	j(f(p, g(x)), f(f(p, f(p, p)), x))	by Axiom 6 RL
	=	j(f(p, g(x)), f(f(f(p, p), p), x))	by Axiom 6 RL
	=	j(f(p, g(x)), f(p, x))	by Axiom 9 LR
	=	f(p, j(g(x), x))	by Axiom 7 RL
	=	f(p, j(x, g(x)))	by Axiom 5 LR
	=	f(p, 0)	by Axiom 3 LR
	=	0	by Lemma 4 LR

Lemma 16: f(w, g(f(w, f(w, g(f(y, f(w, w))))))) = f(w, y)

f(w, g(f(w, f(w, g(f(y, f(w, w)))))))

= f(w, j(0, g(f(w, f(w, g(f(y, f(w, w))))))))
by Axiom 1 RL

= f(w, j(f(w, 0), g(f(w, f(w, g(f(y, f(w, w))))))))
by Lemma 4 RL

= f(w, j(f(w, f(j(w, g(f(j(y, g(f(y, f(w, w))))), f(j(y, g(f(y, f(w, w))))), w)))),
j(y, g(f(y, f(w, w))))), g(f(w, f(w, g(f(y, f(w, w))))))))
by Lemma 8 RL

= f(w, j(f(w, f(j(w, g(f(j(y, g(f(y, f(w, w))))), 0))), j(y, g(f(y, f(w, w))))),
g(f(w, f(w, g(f(y, f(w, w))))))))
by Lemma 15 LR

= f(w, j(f(w, f(j(w, g(0)), j(y, g(f(y, f(w, w))))), g(f(w, f(w, g(f(y, f(w, w))))))))
by Lemma 4 LR

= f(w, j(f(w, f(j(w, j(0, g(0))), j(y, g(f(y, f(w, w))))), g(f(w, f(w, g(f(y, f(w, w))))))))
by Axiom 1 RL

= f(w, j(f(w, f(j(w, 0), j(y, g(f(y, f(w, w))))), g(f(w, f(w, g(f(y, f(w, w))))))))
by Axiom 3 LR

= f(w, j(f(w, f(w, j(y, g(f(y, f(w, w))))), g(f(w, f(w, g(f(y, f(w, w))))))))
by Axiom 2 LR

= f(w, j(j(y, g(f(y, f(w, w)))), g(f(w, f(w, g(f(y, f(w, w))))))))
by Lemma 7 LR

= f(w, j(y, j(g(f(y, f(w, w))), g(f(w, f(w, g(f(y, f(w, w)))))))))
by Axiom 4 LR

= j(f(w, y), f(w, j(g(f(y, f(w, w))), g(f(w, f(w, g(f(y, f(w, w)))))))))
by Axiom 7 LR

= j(f(w, y), f(w, j(f(w, f(w, g(f(y, f(w, w))))), g(f(w, f(w, g(f(y, f(w, w)))))))))
by Lemma 7 RL

= j(f(w, y), f(w, 0))
by Axiom 3 LR

= j(f(w, y), 0)
by Lemma 4 LR

= j(0, f(w, y))
by Axiom 5 LR

= f(w, y)
by Axiom 1 LR

Lemma 17: f(w, f(y, f(w, w))) = f(w, y)

f(w, f(y, f(w, w)))	=	f(w, g(g(f(y, f(w, w))))))	by Lemma 2 RL
	=	f(w, g(g(f(f(y, w), w))))	by Axiom 6 RL
	=	f(w, g(g(f(w, f(w, f(f(y, w), w))))))	by Lemma 12 RL
	=	f(w, g(g(f(w, f(w, f(y, f(w, w)))))))	by Axiom 6 LR
	=	f(w, g(f(w, g(f(w, f(y, f(w, w)))))))	by Lemma 9 RL
	=	f(w, g(f(w, f(w, g(f(y, f(w, w)))))))	by Lemma 9 RL
	=	f(w, y)	by Lemma 16 LR

Lemma 18: f(z, x) = f(f(z, f(x, x)), f(z, f(x, z)))

f(z, x)	=	f(z, f(f(x, x), x))	by Axiom 9 RL
	=	f(f(z, f(x, x)), x)	by Axiom 6 RL
	=	f(f(z, f(x, x)), f(x, f(f(z, f(x, x)), f(z, f(x, x)))))	by Lemma 17 RL
	=	f(f(z, f(x, x)), f(x, f(f(f(z, f(x, x)), z), f(x, x))))	by Axiom 6 RL
	=	f(f(z, f(x, x)), f(x, f(f(z, f(x, x)), z)))	by Lemma 17 LR
	=	f(f(z, f(x, x)), f(x, f(z, f(f(x, x), z))))	by Axiom 6 LR
	=	f(f(z, f(x, x)), f(x, f(z, f(x, f(x, z)))))	by Axiom 6 LR
	=	f(f(z, f(x, x)), f(f(x, z), f(x, f(x, z))))	by Axiom 6 RL
	=	f(f(z, f(x, x)), f(f(z, f(z, f(x, z))), f(x, f(x, z))))	by Lemma 12 RL
	=	f(f(z, f(x, x)), f(z, f(f(z, f(x, z)), f(x, f(x, z)))))	by Axiom 6 LR
	=	f(f(z, f(x, x)), f(z, f(z, f(f(x, z), f(x, f(x, z))))))	by Axiom 6 LR
	=	f(f(z, f(x, x)), f(z, f(z, f(x, f(z, f(x, f(x, z)))))))	by Axiom 6 LR
	=	f(f(z, f(x, x)), f(z, f(z, f(x, f(z, f(x, f(z, f(z, f(x, z))))))))	by Lemma 12 RL
	=	f(f(z, f(x, x)), f(z, f(x, z)))	by Lemma 13 RL

Lemma 19: f(z, x) = f(z, f(x, f(x, f(z, f(x, z)))))

f(z, x)	=	f(f(z, f(x, x)), f(z, f(x, z)))	by Lemma 18 LR
	=	f(z, f(f(x, x), f(z, f(x, z))))	by Axiom 6 LR
	=	f(z, f(x, f(x, f(z, f(x, z)))))	by Axiom 6 LR

Theorem 1: f(a, b) = f(b, a)

f(a, b)	=	f(a, f(b, f(b, f(a, f(b, a)))))	by Lemma 19 LR
	=	f(a, f(b, f(b, f(f(a, b), a))))	by Axiom 6 RL
	=	f(a, f(b, f(f(b, f(a, b)), a)))	by Axiom 6 RL
	=	f(a, f(f(b, f(b, f(a, b))), a))	by Axiom 6 RL
	=	f(a, f(f(a, b), a))	by Lemma 12 LR
	=	f(a, f(a, f(b, a)))	by Axiom 6 LR
	=	f(b, a)	by Lemma 12 LR

B.3 A problem from the domain of lattice ordered groups

The DISCOUNT system is used as a component of the ILF system by the group of B.I. Dahn at the Humbold-University in Berlin (see [Da+94]). From him we received the following problem from the domain of lattice ordered groups:

B.3.1 The problem

For each element in a lattice ordered group show that it can be expressed as the product of its positive part and of its negative part. This is said to be a non-trivial task even for human experts.

```
MODE        PROOF

NAME        lattice3
```

```
ORDERING    LPO
            i > n > u > f > np > pp > 1 > a

EQUATIONS   n(x,y) = n(y,x)
            u(x,y) = u(y,x)
            n(x,n(y,z)) = n(n(x,y),z)
            u(x,u(y,z)) = u(u(x,y),z)
            u(x,x) = x
            n(x,x) = x
            u(x,n(x,y)) = x
            n(x,u(x,y)) = x
            f(x,f(y,z)) = f(f(x,y),z)
            f(1,x) = x
            f(i(x),x) = 1
            i(1) = 1
            i(i(x)) = x
            i(f(x,y)) = f(i(y),i(x))
            f(x,u(y,z)) = u(f(x,y),f(x,z))
            f(x,n(y,z)) = n(f(x,y),f(x,z))
            f(u(y,z),x) = u(f(y,x),f(z,x))
            f(n(y,z),x) = n(f(y,x),f(z,x))
            pp(x) = u(x,1)
            np(x) = n(x,1)
            u(x,n(y,z)) = n(u(x,y),u(x,z))
            n(x,u(y,z)) = u(n(x,y),n(x,z))

CONCLUSION  a = f(pp(a),np(a))
```

B.3.2 The proof

The proof is here reprinted as it has been typeset by our proof transformation system. Except for minor reformatting of the long proof lines in the subproof for lemma 8 nothing has been edited. The extracted PCL protocol for the proof contains 8 results deemed important by the referees. Five of these results are axioms used often in the proof. This effect can be explained by the rich and powerful axiomatization of the problem. The other three selected results became lemmata 3, 7 and 12. Additional lemmata have been selected by the default strategy of the program `lemma`, with the additional restriction that no proof chain should be shorter then 3 steps (option `-u_min_length 3`). See section 8.2.3 for a description of the default criteria of `lemma`.

Consider the following set of axioms:

Axiom 1: n(x, x) = x
Axiom 2: f(1, x) = x
Axiom 3: i(1) = 1
Axiom 4: i(i(x)) = x
Axiom 5: n(x, y) = n(y, x)
Axiom 6: u(x, y) = u(y, x)
Axiom 7: u(x, n(x, y)) = x
Axiom 8: f(i(x), x) = 1
Axiom 9: pp(x) = u(x, 1)
Axiom 10: np(x) = n(x, 1)
Axiom 11: n(x, n(y, z)) = n(n(x, y), z)
Axiom 12: u(x, u(y, z)) = u(u(x, y), z)
Axiom 13: f(x, f(y, z)) = f(f(x, y), z)
Axiom 14: i(f(x, y)) = f(i(y), i(x))
Axiom 15: f(x, u(y, z)) = u(f(x, y), f(x, z))
Axiom 16: f(x, n(y, z)) = n(f(x, y), f(x, z))
Axiom 17: f(u(x, y), z) = u(f(x, z), f(y, z))
Axiom 18: f(n(x, y), z) = n(f(x, z), f(y, z))
Axiom 19: n(x, u(y, z)) = u(n(x, y), n(x, z))

This theorem holds true:

Theorem 1: a = f(pp(a), np(a))

Proof:

Lemma 1: u(v, f(x, v)) = f(pp(x), v)

u(v, f(x, v))	=	**u(f(x, v), v)**	by Axiom 6 RL
	=	u(f(x, v), **f(1, v)**)	by Axiom 2 RL
	=	**f(u(x, 1), v)**	by Axiom 17 RL
	=	f(**pp(x)**, v)	by Axiom 9 RL

Lemma 2: f(pp(i(y)), y) = pp(y)

f(pp(i(y)), y)	=	**u(y, f(i(y), y))**	by Lemma 1 RL
	=	u(y, **1**)	by Axiom 8 LR
	=	**pp(y)**	by Axiom 9 RL

Lemma 3: pp(np(z)) = 1

pp(np(z))	=	**f(pp(i(np(z))), np(z))**	by Lemma 2 RL
	=	f(**f(pp(i(i(np(z)))), i(np(z)))**, np(z))	by Lemma 2 RL
	=	f(f(pp(**np(z)**), i(np(z))), np(z))	by Axiom 4 LR
	=	f(**u(i(np(z)), f(np(z), i(np(z))))**, np(z))	by Lemma 1 RL
	=	f(u(i(np(z)), f(**n(z, 1)**, i(np(z)))), np(z))	by Axiom 10 LR
	=	f(u(i(np(z)), **n(f(z, i(np(z))), f(1, i(np(z))))**), np(z))	by Axiom 18 LR
	=	f(u(i(np(z)), n(f(z, i(np(z))), **i(np(z))**)), np(z))	by Axiom 2 LR
	=	f(u(i(np(z)), **n(i(np(z)), f(z, i(np(z))))**), np(z))	by Axiom 5 LR
	=	f(**i(np(z))**, np(z))	by Axiom 7 LR
	=	**1**	by Axiom 8 LR

Lemma 4: f(x, 1) = x

f(x, 1)	=	f(**i(i(x))**, 1)	by Axiom 4 RL
	=	f(i(i(x)), **i(1)**)	by Axiom 3 RL
	=	**i(f(1, i(x)))**	by Axiom 14 RL
	=	i(**i(x)**)	by Axiom 2 LR
	=	**x**	by Axiom 4 LR

Lemma 5: np(pp(x)) = u(1, np(x))

np(pp(x))	=	np(**u(x, 1)**)	by Axiom 9 LR
	=	np(**u(1, x)**)	by Axiom 6 LR
	=	**n(u(1, x), 1)**	by Axiom 10 LR
	=	**n(1, u(1, x))**	by Axiom 5 LR
	=	**u(n(1, 1), n(1, x))**	by Axiom 19 LR
	=	u(**np(1)**, n(1, x))	by Axiom 10 RL
	=	u(np(1), **n(x, 1)**)	by Axiom 5 RL
	=	u(np(1), **np(x)**)	by Axiom 10 RL
	=	u(**n(1, 1)**, np(x))	by Axiom 10 LR
	=	u(**1**, np(x))	by Axiom 1 LR

Lemma 6: np(pp(x)) = 1

np(pp(x))	=	**u(1, np(x))**	by Lemma 5 LR
	=	**u(np(x), 1)**	by Axiom 6 RL
	=	**pp(np(x))**	by Axiom 9 RL
	=	**1**	by Lemma 3 LR

Lemma 7: np(n(pp(z), y)) = np(y)

np(n(pp(z), y))	=	np(**n(y, pp(z))**)	by Axiom 5 RL
	=	**n(n(y, pp(z)), 1)**	by Axiom 10 LR
	=	**n(y, n(pp(z), 1))**	by Axiom 11 RL
	=	n(y, **np(pp(z))**)	by Axiom 10 RL
	=	**n(np(pp(z)), y)**	by Axiom 5 LR
	=	n(**1**, y)	by Lemma 6 LR
	=	**n(y, 1)**	by Axiom 5 RL
	=	**np(y)**	by Axiom 10 RL

Lemma 8: f(pp(i(y)), np(y)) = u(1, f(pp(i(y)), np(y)))

f(pp(i(y)), np(y))	=	**u(f(pp(i(y)), np(y)), n(f(pp(i(y)), np(y)), 1))** by Axiom 7 RL
	=	u(f(pp(i(y)), np(y)), **np(f(pp(i(y)), np(y)))**) by Axiom 10 RL
	=	u(f(pp(i(y)), np(y)), np(f(pp(i(y)), **n(y, 1)**))) by Axiom 10 LR
	=	u(f(pp(i(y)), np(y)), np(f(pp(i(y)), **n(1, y)**))) by Axiom 5 LR
	=	u(f(pp(i(y)), np(y)), np(**n(f(pp(i(y)), 1), f(pp(i(y)), y)**)) by Axiom 16 LR
	=	u(f(pp(i(y)), np(y)), np(n(**pp(i(y))**, f(pp(i(y)), y)))) by Lemma 4 LR
	=	u(f(pp(i(y)), np(y)), **np(f(pp(i(y)), y))**) by Lemma 7 LR
	=	**u(np(f(pp(i(y)), y)), f(pp(i(y)), np(y)))** by Axiom 6 LR
	=	u(np(**pp(y)**), f(pp(i(y)), np(y))) by Lemma 2 LR
	=	u(**1**, f(pp(i(y)), np(y))) by Lemma 6 LR

Lemma 9: f(pp(i(y)), np(y)) = pp(np(i(y)))

f(pp(i(y)), np(y))	=	**u(1, f(pp(i(y)), np(y)))**	by Lemma 8 LR
	=	**u(f(pp(i(y)), np(y)), 1)**	by Axiom 6 RL
	=	**pp(f(pp(i(y)), np(y)))**	by Axiom 9 RL
	=	pp(**u(np(y), f(i(y), np(y)))**)	by Lemma 1 RL
	=	pp(**u(f(i(y), np(y)), np(y))**)	by Axiom 6 RL
	=	**u(u(f(i(y), np(y)), np(y)), 1)**	by Axiom 9 LR
	=	**u(f(i(y), np(y)), u(np(y), 1))**	by Axiom 12 RL
	=	u(f(i(y), np(y)), **pp(np(y))**)	by Axiom 9 RL
	=	u(f(i(y), np(y)), **1**)	by Lemma 3 LR
	=	**pp(f(i(y), np(y)))**	by Axiom 9 RL
	=	pp(f(i(y), **n(y, 1)**))	by Axiom 10 LR
	=	pp(f(i(y), **n(1, y)**))	by Axiom 5 LR
	=	pp(**n(f(i(y), 1), f(i(y), y)**))	by Axiom 16 LR
	=	pp(n(**i(y)**, f(i(y), y)))	by Lemma 4 LR
	=	pp(n(i(y), **1**))	by Axiom 8 LR
	=	pp(**np(i(y))**)	by Axiom 10 RL

Lemma 10: f(pp(x), np(x)) = x

f(pp(x), np(x))	=	f(**u(x, 1)**, np(x))	by Axiom 9 LR
	=	f(u(x, **f(i(i(x)), i(x))**), np(x))	by Axiom 8 RL
	=	f(u(x, f(**x**, i(x))), np(x))	by Axiom 4 LR
	=	f(u(**f(x, 1)**, f(x, i(x))), np(x))	by Lemma 4 RL
	=	f(**f(x, u(1, i(x)))**, np(x))	by Axiom 15 RL
	=	f(f(x, **u(i(x), 1)**), np(x))	by Axiom 6 RL
	=	f(f(x, **pp(i(x))**), np(x))	by Axiom 9 RL
	=	**f(x, f(pp(i(x)), np(x)))**	by Axiom 13 RL
	=	f(x, **pp(np(i(x)))**)	by Lemma 9 LR
	=	f(x, **1**)	by Lemma 3 LR
	=	**x**	by Lemma 4 LR

Theorem 1: a = f(pp(a), np(a))

a = **f(pp(a), np(a))** by Lemma 10 RL

B.4 Specifications of some other problems

This section contains the problem descriptions of some of the benchmark problems used as examples. Problems not described in the appendix are quoted from literature. See the bibliography for sources.

B.4.1 The problem SelfInverse

Proof that in a ring with $x * x = x$ for all x each element is self inverse with respect to the multiplicative operation, that is $g(x) = x$ for all x.

```
MODE        PROOF

NAME        SelfInverse

ORDERING    XKBO
            f:5 > j:4 > g:3 > 0:1 > a:1

EQUATIONS   j (0,x)       = x
            j (g (x),x)   = 0
            j (j (x,y),z) = j (x,j (y,z))
            j (x,y)       = j(y,x)
            f (f (x,y),z) = f (x,f (y,z))
            f (x,j (y,z)) = j (f (x,y),f (x,z))
            f (j (x,y),z) = j (f (x,z),f (y,z))
            f (x,x)       = x

CONCLUSION  g(a) = a
```

B.4.2 The problem Fibgroup

Show that the following axioms describe a Fibonacci-group of grade 5.

```
MODE        COMPLETION

NAME        FibGroup

ORDERING    KBO
            a      : 6
            b      : 10
            c      : 6
            d      : 6
            e      : 1


EQUATIONS   a (e (x)) = d (x)
            b (a (x)) = e (x)
            c (b (x)) = a (x)
```

```
            d (c (x)) = b (x)
            e (d (x)) = c (x)
```

B.4.3 The problem BoolAssoc

Show that the conjunctive operation (and) in an arbitrary boolean algebra is associative.

```
MODE        PROOF

NAME        BoolAssoc

ORDERING    LPO
            n > a > o > 1 > 0 > x0 > x1 > x2

EQUATIONS   o (x,y) = o (y,x)                    # Commutativity
            a (x,y) = a (y,x)

            a (x,o (y,z)) = o (a (x,y),a (x,z)) # Distributivity
            o (x,a (y,z)) = a (o (x,y),o (x,z))

            o (x,0) = x                          # Neutral elements
            a (x,1) = x

            a (x,n (x)) = 0                      # Complement
            o (x,n (x)) = 1

CONCLUSION  a(a(x0,x1),x2) = a(x0,a(x1,x2))
```

B.4.4 The problem Lattice1

Another problem from the domain of lattice ordered groups.

```
MODE        PROOF

NAME        lattice1

ORDERING    LPO
            n > u > i > f > 1 > a > b > c > d
```

```
EQUATIONS   n(x,y) = n(y,x)
            u(x,y) = u(y,x)
            n(x,n(y,z)) = n(n(x,y),z)
            u(x,u(y,z)) = u(u(x,y),z)
            u(x,x) = x
            n(x,x) = x
            u(x,n(x,y)) = x
            n(x,u(x,y)) = x
            f(x,f(y,z)) = f(f(x,y),z)
            f(1,x) = x
            f(i(x),x) = 1
            i(1) = 1
            i(i(x)) = x
            i(f(x,y)) = f(i(y),i(x))
            f(x,u(y,z)) = u(f(x,y),f(x,z))
            f(x,n(y,z)) = n(f(x,y),f(x,z))
            f(u(y,z),x) = u(f(y,x),f(z,x))
            f(n(y,z),x) = n(f(y,x),f(z,x))
            u(a,b) = b
            u(c,d) = d

CONCLUSION  u(f(a,c),f(b,d)) = f(b,d)
```

B.4.5 The problem DeMorgan

Show one of DeMorgan's laws in a Boolean algebra.

```
MODE        PROOF

NAME        DeMorgan

ORDERING    LPO
            n > a > o > 1 > 0 > y0 > x0

EQUATIONS   o (x,y) = o (y,x)                    # Commutativity
            a (x,y) = a (y,x)

            a (x,o (y,z)) = o (a (x,y),a (x,z)) # Distributivity
            o (x,a (y,z)) = a (o (x,y),o (x,z))

            o (x,0) = x                          # Neutral elements
            a (x,1) = x
```

```
            a (x,n (x)) = 0                      # Complement
            o (x,n (x)) = 1

CONCLUSION  a (n (x0),n (y0)) = n (o (x0,y0))    # De Morgan
```

B.4.6 The problem Lattice2

Yet another (hard) problem from the domain of lattice ordered groups.

```
MODE        PROOF

NAME        lattice2

ORDERING    LPO
            i > f > n > u > 1 > a > b

ORDERING    LPO
            n > u > i > f > 1 > a > b

EQUATIONS   n(x,y) = n(y,x)
            u(x,y) = u(y,x)
            n(x,n(y,z)) = n(n(x,y),z)
            u(x,u(y,z)) = u(u(x,y),z)
            u(x,x) = x
            n(x,x) = x
            u(x,n(x,y)) = x
            n(x,u(x,y)) = x
            f(x,f(y,z)) = f(f(x,y),z)
            f(1,x) = x
            f(i(x),x) = 1
            i(1) = 1
            i(i(x)) = x
            i(f(x,y)) = f(i(y),i(x))
            f(x,u(y,z)) = u(f(x,y),f(x,z))
            f(x,n(y,z)) = n(f(x,y),f(x,z))
            f(u(y,z),x) = u(f(y,x),f(z,x))
            f(n(y,z),x) = n(f(y,x),f(z,x))

CONCLUSION  i(u(a,b)) = n(i(a),i(b))
```

B.4.7 The problem Z22

The axioms describe a large, cyclic group.

```
MODE        COMPLETION

NAME        Z22

ORDERING    LPO
            e1 > e > d1 > d > c1 > c > b1 > b > a1 > a

ORDERING    LPO
            a1 > a > b1 > b > c1 > c > d1 > d > e1 > e

EQUATIONS   a (b (c (x))) = d (x)
            b (c (d (x))) = e (x)
            c (d (e (x))) = a (x)
            d (e (a (x))) = b (x)
            e (a (b (x))) = c (x)
            a (a1 (x)) = x
            a1 (a (x)) = x
            b (b1 (x)) = x
            b1 (b (x)) = x
            c (c1 (x)) = x
            c1 (c (x)) = x
            d (d1 (x)) = x
            d1 (d (x)) = x
            e (e1 (x)) = x
            e1 (e (x)) = x
```

References

[Av91] **Avenhaus, J.:** *Reduktionssysteme I (german language)*, Skript zur Vorlesung im WS91, Fachbereich Informatik der Universität Kaiserslautern

[Av90] **Avenhaus, J.:** *Reduktionssysteme II (german language)*, Skript zur Vorlesung im SS89, Fachbereich Informatik der Universität Kaiserslautern

[AD93] **Avenhaus, J.; Denzinger, J.:** *Distributing equational theorem proving*, Seki-Report SR-93-06, also in Proc. of the RTA-93, Montreal, Springer Verlag, LNCS690, pp. 62-76

[BDP89] **Bachmair, L.; Derschowitz, N.; Plaisted, D.A.:** *Completion without Failure*, Coll. on the Resolution of Equations in Algebraic Structures, Austin 1987, Academic Press 1989

[BH92] **Bonacina, M. P., Hsiang, J.:** *Distributed Deduction by Clause-Diffusion: the Aquarius Prover*, Proc. 3rd DISCO 1993, Gmunden, Springer Verlag, LNCS722, pp. 272-287

[Da+94] **Dahn, B.I.; Gehne, J.; Honigmann, T.; Walther, L.; Wolf, A.:** *Integrating Logical Functions with ILF*, Internal report, Institut für Reine Mathematik, Humbold-University, Berlin, 1994

[De93] **Denzinger, J.:** *Teamwork: Eine Methode zum verteilten, wissensbasierten Entwurf von Theorembeweisern (german language)*, Ph.D.-Thesis, University of Kaiserslautern, 1993

[DP92] **Denzinger, J.; Pitz, W.:** *Das DISCOUNT-System Benutzerhandbuch (german language)*, Seki-Working Paper SWP-92-16

[KB70] **Knuth, D.E.; Bendix, P.B.:** *Simple Word Problems in Universal Algebras*, Computational Problems in Abstract Algebra, ed.: J.Leech, Pergamon Press 1970, pp 263-297

[Li90] **Lingenfelder, C.:** *Structuring Computer generated Proofs*, Ph.D.-Thesis, University of Kaiserslautern, 1990

[LP90] **Lingenfelder,C.; Präcklein, A.:** *Presentation of Proofs in an Equational Calculus*, Seki-Report SR-90-15

[Lin93] **Lind, J.:** *Sicheres Broadcasting für DISCOUNT (german language)*, Project-Report, University of Kaiserslautern, 1993

[LO82] **Lusk, E.L.; Overbeck, R.A.:** *A Short Problem Set for Testing Systems that Include Equality Reasoning*, Argonne National Laboratory, Illinois, 1982

[LW92] **Lusk, E.L.; Wos, L.:** *Benchmark Problems in which equality plays the major role*, Proc. of the CADE-11, 1992, Springer Verlag, LNAI607, pp. 781-785

[Pi92] **Pitz, W.:** *Realisierung eines Systems zum verteilten, wissensbasierten Gleichheitsbeweisen mit Hilfe der Teamwork Methode (german language)*, Diploma thesis, University of Kaiserslautern, 1992

[Sch93] **Schulz, S.:** *Analyse und Transformation von Gleichheitsbeweisen (german language)*, Project-Report, University of Kaiserslautern, 1993

[Si92] **Sims, C.C.:** *The Knuth-Bendix Procedure for Strings as a Substitute for Coset Enumeration*, JSC 12 (1991) pp. 439-442

[Ta56] **Tarski, A:** *Logic, Semantics, Meta mathematics,* Oxford University Press, 1956