THE UNIVERSITY
of LIVERPOOL

THE UNIVERSITY
of MANCHESTER

Fourth Workshop on the Implementation of Logics

Boris Konev
Renate Schmidt (eds.)

Collocated with LPAR 2003
Almaty, Kazakhstan, September 2003

Preface

Following a series of successful workshops held in conjunction with the LPAR conference, the *Fourth Workshop on the Implementation of Logics* was held in conjunction with the *Tenth International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2003)*, in Almaty, Kazakhstan, in September 2003.

Nine submissions were received of which seven were selected for presentation at the workshop. An invited talk was given by Stephan Schulz from the Technische Universität München and RISC Linz.

We thank the program committee who performed the task of reviewing the submissions. We also thank the organisers of LPAR without whom this workshop would certainly not exist.

September 2003

Boris Konev and Renate Schmidt
Liverpool, Manchester

Workshop Organisation

Program Committee

Elvira Albert	Universidad Complutense de Madrid
Bart Demoen	Catholic University of Leuven
Thom Frühwirth	Universität Ulm
Ullrich Hustadt	University of Liverpool
Boris Konev (co-chair)	University of Liverpool
William McCune	Argonne National Laboratory
Gopalan Nadathur	University of Minnesota
Alexandre Riazanov	University of Manchester
Kostis Sagonas	Uppsala University
Renate Schmidt (co-chair)	University of Manchester
Mark Stickel	SRI International
Hantao Zhang	University of Iowa

Previous events

Reunion Workshop (held in conjunction with LPAR'2000 on Reunion Island),
Second Workshop in Cuba (together with LPAR'2001 in Havana, Cuba),
Third workshop in Tbilisi (together with LPAR'2002 in Tbilisi, Georgia).

Table of Contents

Invited talk

Extended abstracts

Simplicity, Measuring, and Good Engineering

One Way to Build a World Class Automated Deduction System

Stephan Schulz[1,2]

[1] Institut für Informatik, Technische Universität München
[2] RISC-Linz, Johannes Kepler Universität Linz*
schulz@informatik.tu-muenchen.de

Abstract

Most published papers on implementation aspects of automated reasoning systems cover only a small set of new techniques. Overview papers are rare, and usually describe the fixed state of a system at a given point in the development process. Moreover, they often have to trade depth for generality. This is particularly true for system descriptions, which often are relegated to second-class status and allowed only a view pages at many major conferences.

In my talk, I will try to shed some lights into the practical aspects of building a complex high-performance theorem prover. I will give an overview on our equational theorem prover E [Sch02]. However, instead of giving a purely static view, I will describe the process that has resulted in a useful and resilient code base which has, up to now, survived at least three major changes without serious problems. I will also discuss some of the design decisions that later turned out to be wrong, and how they have either been fixed or still burden us.

Finally, I will describe some of the engineering tricks and tools we use to make sure that our code remains stable, mostly bug free, and, most of all, maintainable.

References

[Sch02] S. Schulz. E – A Brainiac Theorem Prover. *Journal of AI Communications*, 15(2/3):111–126, 2002.

This talk has been supported by the EU CALCULEMUS Human Potential Programme.

* Currently visiting at the University of Edinburgh.

KAT-ML: An Interactive Theorem Prover for Kleene Algebra with Tests

Kamal Aboul-Hosn and Dexter Kozen

Department of Computer Science
Cornell University
Ithaca, New York 14853-7501, USA
{kamal,kozen}@cs.cornell.edu

Abstract. KAT-ML is an interactive theorem prover for Kleene algebra with tests (KAT). The system is designed to reflect the natural style of reasoning with KAT that one finds in the literature. We describe the main features of the system and illustrate its use with some examples.

1 Introduction

Kleene algebra with tests (KAT), introduced in [13], is an equational system for program verification that combines Kleene algebra (KA) with Boolean algebra. KAT has been applied successfully in various low-level verification tasks involving communication protocols, basic safety analysis, source-to-source program transformation, concurrency control, compiler optimization, and dataflow analysis [1, 3–6, 13, 15]. The system subsumes Hoare logic and is deductively complete for partial correctness over relational models [14].

Much attention has focused on the equational theory of KA and KAT. The axioms of KAT are known to be deductively complete for the equational theory of language-theoretic and relational models, and validity is decidable in *PSPACE* [7, 16]. But because of the practical importance of premises, it is the universal Horn theory that is of more interest; that is, the set of valid sentences of the form

$$p_1 = q_1 \wedge \cdots \wedge p_n = q_n \rightarrow p = q, \tag{1}$$

where the atomic symbols are implicitly universally quantified. Typically, the premises $p_i = q_i$ are basic assumptions regarding the interaction of atomic programs and tests, and the conclusion $p = q$ represents the equivalence of an optimized and unoptimized program, a partial correctness assertion, or the equivalence of an annotated and unannotated program. The necessary premises are obtained by inspection of the program and their validity may depend on properties of the domain of computation, but they are usually quite simple and easy to verify by inspection, since they typically only involve atomic programs and tests. Once the premises are established, the proof of (1) is purely propositional. This ability to introduce premises as needed is one of the features that makes KAT so versatile. By comparison, Hoare logic has only the assignment rule, which is much more limited. In addition, this style of reasoning allows a clean separation between first-order interpreted reasoning to justify the premises

$p_1 = q_1 \wedge \cdots \wedge p_n = q_n$ and purely propositional reasoning to establish that the conclusion $p = q$ follows from the premises.

We have implemented an interactive theorem prover KAT-ML for Kleene algebra with tests. The system is designed to reflect the natural style of reasoning with KAT that one finds in the literature. In this paper we describe the main features of the system and illustrate its use with some examples.

KAT-ML allows the user to develop a proof interactively in a natural human style, keeping track of the details of the proof. An unproven theorem will have a number of outstanding *tasks* in the form of unproven Horn formulas. The initial task is the theorem itself. The user applies axioms and lemmas to simplify the tasks, which may introduce new (presumably simpler) tasks. When all tasks are discharged, the proof is complete.

As the user applies proof rules, the system constructs an independently verifiable proof object in the form of a λ-term. The proof term of an unproven theorem has free task variables corresponding to the undischarged tasks. The system can import and export proofs in XML format.

We have used KAT-ML to verify formally several known results in the literature, some of which had previously been verified only by hand, including the KAT translation of the Hoare partial correctness rules [14], a verification problem involving a Windows device driver [2], and an intricate scheme equivalence problem [1].

The system is implemented in Standard ML and is easy to install and use. Source code and executable images for various platforms can be downloaded from [10]. Several tutorial examples are also provided with the distribution.

The *PSPACE* decision procedure for the equational theory has been implemented by Cohen [4–6]. Cohen's approach is to try to reduce a Horn formula to an equivalent equation, then apply the *PSPACE* decision procedure to automatically verify the resulting equation. This reduction is possible in many cases, but not always. Moreover, the decision procedure does not produce an independently verifiable proof object.

2 Preliminary Definitions

2.1 Kleene Algebra

Kleene algebra (KA) is the algebra of regular expressions [11, 8]. The axiomatization used here is from [12]. A *Kleene algebra* is an algebraic structure $(K, +, \cdot, {}^*, 0, 1)$ that satisfies the following axioms:

$$(p+q)+r = p+(q+r) \quad (2) \qquad (pq)r = p(qr) \quad (3)$$
$$p+q = q+p \quad (4) \qquad p1 = 1p = p \quad (5)$$
$$p+0 = p+p = p \quad (6) \qquad 0p = p0 = 0 \quad (7)$$
$$p(q+r) = pq+pr \quad (8) \qquad (p+q)r = pr+qr \quad (9)$$
$$1+pp^* \leq p^* \quad (10) \qquad q+pr \leq r \rightarrow p^*q \leq r \quad (11)$$
$$1+p^*p \leq p^* \quad (12) \qquad q+rp \leq r \rightarrow qp^* \leq r \quad (13)$$

This a universal Horn axiomatization. Axioms (2)–(9) say that K is an *idempotent semiring* under $+, \cdot, 0, 1$. The adjective *idempotent* refers to (6). Axioms (10)–(13)

say that p^*q is the $\leq$-least solution to $q + px \leq x$ and qp^* is the $\leq$-least solution to $q + xp \leq x$, where $\leq$ refers to the natural partial order on K defined by $p \leq q \stackrel{\text{def}}{\iff} p + q = q$.

Standard models include the family of regular sets over a finite alphabet, the family of binary relations on a set, and the family of $n \times n$ matrices over another Kleene algebra. Other more unusual interpretations include the min,+ algebra, also known as the *tropical semiring*, used in shortest path algorithms, and models consisting of convex polyhedra used in computational geometry.

There are several alternative axiomatizations in the literature, most of them infinitary. For example, a Kleene algebra is called *star-continuous* if it satisfies the infinitary property $pq^*r = \sup_n pq^n r$. This is equivalent to infinitely many equations

$$pq^n r \leq pq^* r, \quad n \geq 0 \tag{14}$$

and the infinitary Horn formula

$$(\bigwedge_{n \geq 0} pq^n r \leq s) \rightarrow pq^* r \leq s. \tag{15}$$

All natural models are star-continuous. However, this axiom is much stronger than the finitary Horn axiomatization given above and would be more difficult to implement, since it would require meta-rules to handle the induction needed to establish (14) and (15).

The completeness result of [12] says that all true identities between regular expressions interpreted as regular sets of strings are derivable from the axioms. In other words, the algebra of regular sets of strings over the finite alphabet P is the free Kleene algebra on generators P. The axioms are also complete for the equational theory of relational models.

See [12] for a more thorough introduction.

2.2 Kleene Algebra with Tests

A *Kleene algebra with tests* (KAT) [13] is just a Kleene algebra with an embedded Boolean subalgebra. That is, it is a two-sorted structure $(K,\ B,\ +,\ \cdot,\ ^*,\ \bar{}\ ,\ 0,\ 1)$ such that

- $(K,\ +,\ \cdot,\ ^*,\ 0,\ 1)$ is a Kleene algebra,
- $(B,\ +,\ \cdot,\ \bar{}\ ,\ 0,\ 1)$ is a Boolean algebra, and
- $B \subseteq K$.

Elements of B are called *tests*. The Boolean complementation operator $\bar{}$ is defined only on tests. In KAT-ML, variables beginning with an upper-case character denote tests, and those beginning with a lower-case character denote arbitrary Kleene elements.

The axioms of Boolean algebra are purely equational. In addition to the Kleene algebra axioms above, tests satisfy the equations

$$\begin{aligned} BC &= CB & BB &= B \\ B + CD &= (B + C)(B + D) & B + 1 &= 1 \\ \overline{B + C} &= \overline{B} + \overline{C} & \overline{BC} &= \overline{B} + \overline{C} \\ B + \overline{B} &= 1 & B\overline{B} &= 0 \\ \overline{\overline{B}} &= B \end{aligned}$$

The **while** program constructs are encoded as in propositional Dynamic Logic [9]:

$$\begin{aligned} p\,;\, q &\stackrel{\text{def}}{=} pq \\ \textbf{if } B \textbf{ then } p \textbf{ else } q &\stackrel{\text{def}}{=} Bp + \overline{B}q \\ \textbf{while } B \textbf{ do } p &\stackrel{\text{def}}{=} (Bp)^*\overline{B}. \end{aligned}$$

The Hoare partial correctness assertion $\{B\}\, p\, \{C\}$ is expressed as an equation $Bp\overline{C} = 0$, or equivalently, $Bp = BpC$. All Hoare rules are derivable in KAT; indeed, KAT is deductively complete for relationally valid propositional Hoare-style rules involving partial correctness assertions [14] (propositional Hoare logic is not).

The following simple example illustrates how equational reasoning with Horn formulas proceeds in KAT. To illustrate the use of our system, we will give a mechanical derivation of this lemma in Section 3.4.

Lemma 1. *The following equations are equivalent in* KAT*:*

(i) $Cp = C$
(ii) $Cp + \overline{C} = 1$
(iii) $p = \overline{C}p + C$.

Proof. We prove separately the four Horn formulas (i) → (ii), (i) → (iii), (ii) → (i), and (iii) → (i).

For the first, assume that (i) holds. Replace Cp by C on the left-hand side of (ii) and use the Boolean algebra axiom $C + \overline{C} = 1$.

For the second, assume again that (i) holds. Replace the second occurrence of C on the right-hand side of (iii) by Cp and use distributivity law $\overline{C}p + Cp = (\overline{C} + C)p$, the Boolean algebra axiom $\overline{C} + C = 1$, and the multiplicative identity axiom $1p = p$.

Finally, for (ii) → (i) and (iii) → (i), multiply both sides of (ii) or (iii) on the left by C and use distributivity and the Boolean algebra axioms $C\overline{C} = 0$ and $CC = C$.

See [13, 14, 17] for a more detailed introduction to KAT.

3 Description of the System

KAT-ML is an interactive theorem prover for Kleene algebra with tests. It is written in Standard ML. The system has a command-line interface that works on any platform and a graphical user interface that works on any UNIX-based operating system. A user

can create and manage libraries of KAT theorems that can be proved and cited by name in later proofs. A few standard libraries containing the axioms of KAT and commonly used lemmas are provided.

At the core of the KAT theorem prover are the commands *publish* and *cite*. Publication is a mechanism for making previous constructions available in an abbreviated form. Citation incorporates previously-constructed objects in a proof without having to reconstruct them. All other commands relate to these two in some way.

3.1 Representation of Proofs

KAT-ML represents a proof as a λ-term abstracted over the individual variables $p, q, \ldots$ and test variables $B, C, \ldots$ that appear in the theorem and proof variables $P_0, P_1, \ldots$ for the premises. If the proof is not complete, the proof term will also contain free task variables $T_0, T_1, \ldots$ for the undischarged tasks. All proof terms are well-typed, and the type is the theorem, according to the Curry-Howard isomorphism [18]. The theorem and its proof can be reconstructed from the proof term.

For instance, consider a universal Horn formula

$$\forall x_1 \ldots \forall x_m \;\; \varphi_1 \to \varphi_2 \to \cdots \to \varphi_n \to \psi,$$

where $\varphi_1, \ldots, \varphi_n$ are the premises, ψ is the conclusion, and $x_1, \ldots, x_m$ are all of the individual variables that appear in the φ_i or ψ. Viewed as a type, this theorem would be realized by a proof term representing a function that takes an arbitrary substitution for the variables x_i and proofs of the premises φ_j and returns a proof of the conclusion ψ. Initially, the proof is represented as the λ-term

$$\lambda x_1 \ldots \lambda x_m . \lambda P_1 \ldots \lambda P_n . (T P_1 \cdots P_n),$$

where T is a free variable of type $\varphi_1 \to \varphi_2 \to \cdots \to \varphi_n \to \psi$ representing the main task. Publishing the theorem results in the creation of this initial proof term; as proof rules are applied, the proof term is expanded accordingly. Citing a theorem φ as a lemma in the proof of another theorem ψ is equivalent to substituting the proof term of φ for a free task variable in the proof term of ψ. The proof of φ need not be complete for this to happen; any undischarged tasks of φ become undischarged tasks of ψ.

3.2 Citation

The system allows two forms of citation, *focused* and *unfocused*. Citations are applied to the current task. One may cite a published theorem with the command *cite* or a premise of the current task with the command *use*.

In unfocused citation, the conclusion of the cited theorem is unified with the conclusion of the current task, giving a substitution of terms for the individual variables. This substitution is then applied to the premises of the cited theorem, and the current task is replaced with several new (presumably simpler) tasks, one for each premise of the cited theorem. Each specialized premise of the cited theorem must now be proved under the premises of the original task.

For example, suppose the current task is

```
T6: p < r, q < r, r;r < r |- p;q + q;p < r
```

indicating that one must prove the conclusion $pq + qp \leq r$ under the three premises $p \leq r, q \leq r$, and $rr \leq r$ (in the display, the symbol < denotes less-than-or-equal-to $\leq$). The proof term at this point is

```
\p,q,r.\P0,P1,P2.(T6 (P0,P1,P2))
```

(in the display, \ represents λ). Here `T6` is a task variable representing a function that returns a proof of $pq + qp \leq r$ when given proofs P_0, P_1, P_2 for the three premises.

To prove $pq + qp \leq r$, it suffices to prove $pq \leq r$ and $qp \leq r$ separately. Thus an appropriate citation at this point would be the lemma

```
sup: x < z -> y < z -> x + y < z
```

The conclusion of `sup`, namely $x + y \leq z$, is unified with the conclusion of the task `T6`, giving the substitution $x = pq,\ y = qp,\ z = r$. This substitution is then applied to the premises of `sup`, and the old task `T6` is replaced by two new tasks

```
T7: p < r, q < r, r;r < r |- p;q < r
T8: p < r, q < r, r;r < r |- q;p < r
```

This operation is reflected in the proof term as follows:

```
\p,q,r.\P0,P1,P2.(sup [x=p;q y=q;p z=r] (T7 (P0,P1,P2),
                                         T8 (P0,P1,P2)))
```

This new proof term is a function that returns a proof of $pq + qp \leq r$ when `sup` is provided with proofs of its premises, which are the incomplete proofs `T7(P0,P1,P2)` and `T8(P0,P1,P2)` of $pq \leq r$ and $qp \leq r$, respectively. The arguments of `T7` and `T8` are the proofs `P0,P1,P2` of the premises of the original task `T6`.

A premise can be cited with the command *use* just when the conclusion is identical to that premise, in which case the corresponding task variable is replaced with the proof variable of the cited premise.

Focused citation is used to implement the proof rule of substitution of equals for equals. In focused citation, a subterm of the conclusion of the current task is specified; this subterm is called the *focus*. The system provides a set of navigation commands to allow the user to focus on any subterm. When there is a current focus, any citation will attempt to unify either the left- or the right-hand side of the conclusion of the cited theorem with the focus, then replace it with the specialized other side. As with unfocused citation, new tasks are introduced for the premises of the cited theorem. A corresponding substitution is also made in the proof term. In the event that multiple substitutions are possible, the system prompts the user with the options and applies the one selected.

For example, suppose that the current task is

```
T0: p;q = 0 |- (p + q)* < q*;p*
```

The axiom

```
*R: x;z + y < z -> x*;y < z
```

is a good one to apply. However, the system will not allow the citation yet, since there is nothing to unify with y. If the task were

```
T1: p;q = 0 |- (p + q)*;1 < q*;p*
```

then y would unify with 1. We can make this change by focusing on the left-hand side of the conclusion of `T0` and citing the axiom

```
id.R: x;1 = x
```

Focusing on the desired subterm gives

```
T0: p;q = 0 |- (p + q)* < q*;p*
               --------
```

where the focus is underlined. Now citing `id.R` unifies the right-hand side with the focus and replaces it with the specialized left-hand side of `id.R`, yielding

```
T1: p;q = 0 |- (p + q)*;1 < q*;p*
               ----------
```

Many other commands exist to facilitate the proving of theorems. The `cut` rule adds a new premise σ to the list of premises of the current task and adds a second task to prove σ under the original premises. Starting from the task $\varphi_1, \ldots, \varphi_n \vdash \psi$, the command `cut` σ yields the two new tasks

$$\varphi_1, \ldots, \varphi_n, \sigma \vdash \psi \qquad \varphi_1, \ldots, \varphi_n \vdash \sigma.$$

For a list of other commands, see the README file in the KAT-ML distribution [10].

3.3 Heuristics

KAT-ML has a simple set of heuristics to aid in proving theorems. The heuristics can automatically perform unfocused citation with premises or theorems in the library that have no premises (such as reflexivity) that unify with the current task.

The system also provides a list of suggested citations from the library, both focused and unfocused, that unify with the current task and focus. Currently, the system does not attempt to order the suggestions, but only provides a list of all possible citations. Eventually, the system will attempt to order the list of suggested citations according to some learned priority determined by usage statistics.

3.4 An Extended Example

The following is an example of the system in use. It is the proof of the first and last Horn formulas in Lemma 1. The proof demonstrates basic publication and citation, focus, and navigation. For more examples of varying complexity, see the Examples directory in the KAT-ML distribution [10].

```
>pub C p = C -> C p + ~C = 1
L0: C;p = C -> C;p + ~C = 1  (1 task)

current task:
T0: C;p = C |- C;p + ~C = 1

>proof
\C,p.\P0.(T0 P0)

current task:
T0: C;p = C |- C;p + ~C = 1

>focus

current task:
T0: C;p = C |- C;p + ~C = 1

C;p + ~C = 1
--------

>down

current task:
T0: C;p = C |- C;p + ~C = 1

C;p + ~C = 1
---

>use A0 l
cite A0

current task:
T1: C;p = C |- C + ~C = 1

C + ~C = 1
-

>unfocus

current task:
T1: C;p = C |- C + ~C = 1

>cite compl+
cite compl+
task completed

no tasks

>proof
\C,p.\P0.(subst [0,0,1] (C;p + ~C = 1)
   L P0 (compl+ [B=C]))

no tasks

>heuristics theorem on

no tasks

>heuristics prem on

no tasks

>pub p = ~C p + C -> C p = C
L1: p = ~C;p + C -> C;p = C  (1 task)

current task:
T2: p = ~C;p + C |- C;p = C

>proof
\C,p.\P1.(T2 P1)

current task:
T2: p = ~C;p + C |- C;p = C

>focus

current task:
T2: p = ~C;p + C |- C;p = C

C;p = C
---

>r

current task:
T2: p = ~C;p + C |- C;p = C

C;p = C
      -

>cite id+L r
cite id+L

current task:
T3: p = ~C;p + C |- C;p = 0 + C

C;p = 0 + C
      -----

>d

current task:
T3: p = ~C;p + C |- C;p = 0 + C

C;p = 0 + C
      -

>cite annihL r
cite annihL
x=? p

current task:
T4: p = ~C;p + C |- C;p = 0;p + C

C;p = 0;p + C
      ---

>d

current task:
T4: p = ~C;p + C |- C;p = 0;p + C

C;p = 0;p + C
      -

>cite compl. r
cite compl.
B=? C

current task:
T5: p = ~C;p + C |- C;p = C;~C;p + C

C;p = C;~C;p + C
      ----
```

```
>u r

current task:
T5: p = ~C;p + C |- C;p = C;~C;p + C

C;p = C;~C;p + C
                -

>cite idemp. r
cite idemp.

current task:
T6: p = ~C;p + C |- C;p = C;~C;p + C;C

C;p = C;~C;p + C;C
                ---

>u

current task:
T6: p = ~C;p + C |- C;p = C;~C;p + C;C

C;p = C;~C;p + C;C
      ------------

>cite distrL r
cite distrL

current task:
T7: p = ~C;p + C |- C;p = C;(~C;p + C)

C;p = C;(~C;p + C)
      ------------

>unfocus

current task:
T7: p = ~C;p + C |- C;p = C;(~C;p + C)

>cite cong.L
cite cong.L
cite A0
task completed

no tasks

>proof
\C,p.\P1.(subst [1,1] (C;p = C) R (id+L [x=C]) (subst [1,0,1] (C;p = 0 + C) R
(annihL [x=p]) (subst [1,0,0,1] (C;p = 0;p + C) R (compl. [B=C])
(subst [1,1,1] (C;p = C;~C;p + C) R (idemp. [B=C])
(subst [1,1] (C;p = C;~C;p + C;C) R (distrL [x=C y=~C;p z=C])
(cong.L [x=C y=p z=~C;p + C] P1))))))

no tasks

>
```

4 Conclusions and Future Work

We have described an interactive theorem prover for Kleene algebra with tests (KAT). We feel that the most interesting part of this work is not the particular data structures or algorithms we have chosen—these are fairly standard—but rather the design of the mode of interaction between the user and the system. Our main goal was not to automate as much of the reasoning process as possible, but rather to provide support to the

user for developing proofs in a natural human style, similar to proofs in KAT found in the literature. KAT is naturally equational, and equational reasoning pervades every aspect of reasoning with KAT. Our system is true to that style. The user can introduce self-evident equational premises describing the interaction of atomic programs and tests and reason under those assumptions to derive the equivalence of more complicated programs. The system performs low-level reasoning tasks and bookkeeping and facilitates sharing of lemmas, but it is up to the user to develop the main proof strategies.

Our current focus is to extend the system with first-order constructs, including arrays. Here atomic programs are assignments $x := t$, where x is a program variable and t a first-order term ranging over a domain of computation of a particular first-order signature. There are only a few extra equational axioms needed for most schematic (uninterpreted) first-order reasoning and a single rule for introducing properties of the domain of computation [1, 3]. The first-order axioms are typically used to establish the correctness of premises; once this is done, reasoning reverts to the purely propositional level. A short-term goal is to implement enough first-order infrastructure to support the mechanical derivation of various proofs in first-order KAT appearing in the literature [1, 3].

Acknowledgments

This work was supported in part by NSF grant CCR-0105586 and ONR Grant N00014-01-1-0968. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of these organizations or the US Government.

References

1. Allegra Angus and Dexter Kozen. Kleene algebra with tests and program schematology. Technical Report 2001-1844, Computer Science Department, Cornell University, July 2001.
2. Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proceedings of the 8th International SPIN Workshop on Model Checking of Software (SPIN 2001)*, volume 2057 of *Lecture Notes in Computer Science*, pages 103–122. Springer-Verlag, May 2001.
3. Adam Barth and Dexter Kozen. Equational verification of cache blocking in LU decomposition using Kleene algebra with tests. Technical Report 2002-1865, Computer Science Department, Cornell University, June 2002.
4. Ernie Cohen. Lazy caching in Kleene algebra. `http://citeseer.nj.nec.com/22581.html`.
5. Ernie Cohen. Hypotheses in Kleene algebra. Technical Report TM-ARH-023814, Bellcore, 1993. `http://citeseer.nj.nec.com/1688.html`.
6. Ernie Cohen. Using Kleene algebra to reason about concurrency control. Technical report, Telcordia, Morristown, N.J., 1994.
7. Ernie Cohen, Dexter Kozen, and Frederick Smith. The complexity of Kleene algebra with tests. Technical Report 96-1598, Computer Science Department, Cornell University, July 1996.
8. John Horton Conway. *Regular Algebra and Finite Machines*. Chapman and Hall, London, 1971.

9. Michael J. Fischer and Richard E. Ladner. Propositional dynamic logic of regular programs. *J. Comput. Syst. Sci.*, 18(2):194–211, 1979.
10. `http://www.cs.cornell.edu/kozen/KAT-ML.zip`.
11. Stephen C. Kleene. Representation of events in nerve nets and finite automata. In C. E. Shannon and J. McCarthy, editors, *Automata Studies*, pages 3–41. Princeton University Press, Princeton, N.J., 1956.
12. Dexter Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Infor. and Comput.*, 110(2):366–390, May 1994.
13. Dexter Kozen. Kleene algebra with tests. *Transactions on Programming Languages and Systems*, 19(3):427–443, May 1997.
14. Dexter Kozen. On Hoare logic and Kleene algebra with tests. *Trans. Computational Logic*, 1(1):60–76, July 2000.
15. Dexter Kozen and Maria-Cristina Patron. Certification of compiler optimizations using Kleene algebra with tests. In John Lloyd, Veronica Dahl, Ulrich Furbach, Manfred Kerber, Kung-Kiu Lau, Catuscia Palamidessi, Luis Moniz Pereira, Yehoshua Sagiv, and Peter J. Stuckey, editors, *Proc. 1st Int. Conf. Computational Logic (CL2000)*, volume 1861 of *Lecture Notes in Artificial Intelligence*, pages 568–582, London, July 2000. Springer-Verlag.
16. Dexter Kozen and Frederick Smith. Kleene algebra with tests: Completeness and decidability. In D. van Dalen and M. Bezem, editors, *Proc. 10th Int. Workshop Computer Science Logic (CSL'96)*, volume 1258 of *Lecture Notes in Computer Science*, pages 244–259, Utrecht, The Netherlands, September 1996. Springer-Verlag.
17. Dexter Kozen and Jerzy Tiuryn. Substructural logic and partial correctness. *Trans. Computational Logic*, 4(3):355–378, July 2003.
18. Morten Heine Sørensen and Pawel Urzyczyn. Lectures on the Curry–Howard isomorphism. Available as DIKU Rapport 98/14, 1998.

MUltlog and MUltseq Reanimated and Married *

M. Baaz[1] C.G. Fermüller[1] A. Gil[2] G. Salzer[1] N. Preining[1]

[1]Technische Universität Wien, Vienna, Austria
[2]Universitat Pompeu Fabra, Barcelona, Spain

1 Introduction

MUltlog is a logic engineering tool that produces descriptions of various sound and complete logical calculi for an arbitrary finite-valued first-order logic from a given specification of the semantics of such a logic (see [1]). MUltseq, on the other hand, is a simple, generic, sequent based theorem prover for propositional finite-valued logics (see [6]). From its very beginning, MUltseq was intended to be a 'companion' to MUltlog. So far, however, MUltseq does not directly use the representation of sequent rules as generated by MUltlog. Moreover (due to lack of funding, personnel and time), further development and maintenance of both system has been stalled for some time now. It is the purpose of this abstract to shortly describe the two systems and the current efforts to integrate them.

2 A short description of MUltlog

A many-valued logic is characterized by the truth functions associated with its propositional operators and quantifiers. More precisely, if W denotes the set of truth values, then a total function $\tilde{\theta}\colon W^n \mapsto W$ is associated with each n-ary operator θ, and a total function $\tilde{\lambda}\colon (2^W - \{\emptyset\}) \mapsto W$ with each quantifier λ.[1]

For finitely-valued logics, $\tilde{\theta}$ and $\tilde{\lambda}$ can be specified by finite tables. The size of quantifier tables, however, grows exponentially with the number of truth values. Fortunately, many operators and quantifiers are defined implicitly as greatest lower or least upper bounds with respect to some (semi-)lattice ordering on the truth values; conjunction and disjunction as well as universal and existential quantification fall into this class. For this reason MUltlog supports several possibilities for specifying operators and quantifiers.

The kernel of MUltlog is written in Prolog. Its main task is to compute a certain conjunctive normal form (CNF) for each combination of operators or quantifiers with truth values. Once given the CNF, all calculi can be obtained more or less by syntactic transformations. The problem is not to find any such CNFs: one particular kind can be immediately obtained from the definition of

* Partially supported by the Austrian science foundation FWF, project P16539-N04.

[1] Quantifiers defined this way are called *distribution quantifiers*. The intuitive meaning is that a quantified formula $(\lambda x)A(x)$ takes the value $\tilde{\lambda}(U)$ if the instances $A(d)$ take exactly the elements of U as their values. E.g., the universal quantifier in classical logic can be defined as $\tilde{\forall}(\{t\}) = t$ and $\tilde{\forall}(\{f\}) = \tilde{\forall}(\{t, f\}) = f$.

operators and quantifiers. However, these CNFs are of a maximal branching degree and therefore do not lead to feasible deduction systems. `MUltlog` computes CNFs that are *optimal* regarding the number of conjuncts. For operators and quantifiers referring to an ordering the matter is easy: provably optimal CNFs are obtained by instantiating a schema. For all other operators and quantifiers more complex computations are needed, which involve resolution and a special inference rule called combination (for a detailed description and correctness proofs of the employed algorithms see [8]).

The output consists of a style file containing LaTeX definitions specific to the input logic, which is included by a generic document when compiled with TeX. The style file is generated by DCGs (definite clause grammars) on the basis of the specification read by `MUltlog` and the minimized CNFs computed by `MUltlog`.

Users of `MUltlog` can choose among different interfaces. One is written in Tcl/Tk and runs under Unix and X-Windows. A second one is written in C for PCs under DOS. A third one is written in HTML and Perl, providing access to `MUltlog` via WWW: the user fills in some HTML forms and gets the output of `MUltlog` as a Postscript file, obviating the need to install it on her own machine. All three interfaces communicate with `MUltlog` by an ordinary text file, which can be viewed as a fourth interface. Moreover there exists `JMUltlog`, a Java applet serving roughly the same purpose as the HTML/Perl interface.

3 A short description of `MUltseq`

In its core, `MUltseq` is a generic sequent prover for propositional finitely-valued logics. This means that it takes as input the rules of a many-valued sequent calculus as well as a many-sided sequent and searches – automatically or interactively – for a proof of the latter. For the sake of readability, the output of `MUltseq` is typeset as a LaTeX document.

Though the sequent rules can be entered by hand, `MUltseq` is primarily intended as a companion for `MUltlog`. Provided the input sequent calculus is sound and complete for the logic under consideration – which is always the case when the rules were computed by `MUltlog` – `MUltseq` serves as a decision procedure for the validity of *sequents* and *formulas*. More interestingly, `MUltseq` can also be used to decide the *consequence relations* associated with the logic and the sequent calculus. The problem of deciding whether a particular formula ϕ is true in all models satisfying a given set of formulas Δ, i.e., whether ϕ logically follows from Δ, can be reduced to the problem of proving that certain sequent that depends only on ϕ and Δ is true. Similarly, as a consequence of the *Deduction Detachment Theorem for many-valued sequents* [5, 7], the problem of finding a derivation of a sequent σ from hypotheses Σ can be reduced to proving a particular set of sequents.

From the algebraic point of view, it is an interesting problem to determine whether an *equation* or a *quasi-equation* is valid in a finite algebra. If we consider the algebra as a set of truth values and a collection of finitely-valued connectives,

and use an appropriate translation of equations and quasi-equations to sequents, the problem again reduces to the provability of many-valued sequents [4].

The decision procedures implemented in `MUltseq` help to get a better intuition and understanding of some theoretical problems. For instance, it is known that each propositional logic between the implication-less fragment of Intuitionistic Propositional Calculus and Classical Propositional Calculus has an algebraic semantics. If we consider the algebraic semantics of all these logics, we obtain a denumerable chain which corresponds to the chain of all subvarieties of the variety of Pseudo-complemented Distributive Lattices [7]. Each of these subvarieties is generated by a finite algebra, so the study of the sequent calculi obtained by `MUltlog` for each of these algebras and the decision procedures in `MUltseq` might help to find algebraizable Gentzen systems for the original logics.

4 Availability

Further information on `MUltlog` as well as the latest version of the system (version 1.10, dated 11/07/2001) is available at

`http://www.logic.at/multlog` .

`MUltseq` is currently is at version 0.6 (dated 13/09/2002). It is available at

`http://www.logic.at/multseq` .

5 The marriage agenda

The input for `MUltseq`, i.e. the description of sequent rules for the introduction of connectives at the sequent-positions corresponding to the truth values, is currently prepared by hand. In principle, such a description could and should be extracted from the output of `MUltlog`. Moreover, the intended use of the systems is to investigate and compare the forms of logical rules that can be computed from truth tables and to check simple logical statements by using these rules. This calls for an explicit integration of `MUltlog` and `MUltseq`. The corresponding agenda is as follows:

1. Write a conversion program that takes the output of `MUltlog`, as described above, as input and generates the corresponding sequent rules in the format used for the input of `MUltseq`.
2. Prepare an integrated distribution package that contains properly updated versions of `MUltlog`, `MUltseq` and the conversion tool just described.
3. Design and maintain a joint internet page, that not only just refers to the already available seperate pages for the two systems, but describes and illustrates the intended use of the integrated system.

6 Future developments

Argueably, a happy marriage should result in common offspring. We list some goals for future developments of `MUltlog` and `MUltseq`; in particular ones that serve the aim of a better integration of the two systems.

- *First order theorem proving:* `MUltseq` should be extended to include the application of rules for distribution quantifiers as computed by `MUltlog`.
- *Model construction:* Augmentation of `MUltseq` with features for the explicit construction of (descriptions of) counter models for non-valid formulas and invalid statements involving different versions of consequence relations.
- *Extension to projective logics:* In [2] the systematic construction of special sequent calculi for projective logics, an extension of the class of finite valued logics, has been described. We plan to integrate these algorithms into `MUltlog` and, correspondingly, to enhance `MUltseq` to allow for the use of the resulting sequent calculi in proof search.
- *Cut elimination:* A future version of `MUltlog` should construct specifications of cut elimination algorithms for finite-valued logics as described in [3]. The corresponding cut-reduction operators should then be integrated into `MUltseq`, together with the possibility to apply appropriate cut rules, at least in an interactive fashion.

References

1. M. Baaz, C. G. Fermüller, G. Salzer, and R. Zach. MUltlog 1.0: Towards an expert system for many-valued logics. In M. A. McRobbie and J. K. Slaney, editors, *13th Int. Conf. on Automated Deduction (CADE'96)*, LNCS 1104 (LNAI), pp. 226–230. Springer-Verlag, 1996.
2. M. Baaz and C. G. Fermüller. Analytic Calculi for Projective Logics. In Neil V. Murray (Ed.), *Automated Reasoning with Analytic Tableaux and Related Methods*, TABLEAUX'99, Saratoga Springs, NY, USA, June 1999, LNAI 1617, Springer-Verlag, 1999, pp. 36–50.
3. M. Baaz, C. G. Fermüller, G. Salzer, and R. Zach. Elimination of Cuts in First-Order Finite-Valued Logics. *Journal of Information Processing and Cybernetics*, EIK 29 (1993) 6, pp. 333-355.
4. A.J. Gil, J. Rebagliato, and V. Verdú. A strong completeness theorem for the Gentzen systems associated with finite algebras. *Journal of Applied non-Classical Logics*, vol. 9-1:9–36, 1999.
5. A.J. Gil, A. Torrens, and V. Verdú. On Gentzen Systems Associated with the Finite Linear MV-algebras. *Journal of Logic and Computation*, 7:1–28, 1997.
6. A.J. Gil, G. Salzer. `MUltseq`: Sequents, Equations, and Beyond. Extended version of an abstract presented at the *Joint conference of the 5th Barcelona Logic Meeting and the 6th Kurt Gödel Colloquium*, June 1999; available at `http://www.logic.at/multseq`
7. J. Rebagliato and V. Verdú. Algebraizable Gentzen systems and the Deduction Theorem for Gentzen systems. Mathematics Preprint Series 175, Universitat de Barcelona, June 1995.

8. G. Salzer. Optimal axiomatizations for multiple-valued operators and quantifiers based on semi-lattices. In M. A. McRobbie and J. K. Slaney, editors, *13th Int. Conf. on Automated Deduction (CADE'96)*, LNCS 1104 (LNAI), pages 688–702. Springer-Verlag, 1996.
9. G. Salzer. Optimal Axiomatizations of Finitely-valued Logics. *Information and Computation*, 162:185–205, 2000.

A Syntactic Approach to Satisfaction

Guilherme Bittencourt, Jerusa Marchi, and Régis S. Padilha

Departamento de Automação e Sistemas
Universidade Federal de Santa Catarina
88040-900 - Florianópolis - SC - Brazil
{ gb | jerusa | regis }@das.ufsc.br

Abstract. Most of the research on propositional logic satisfiability follows the Davis-Putnam approach, which is based on a semantic view that all the possible assignments of true values to propositional symbols should be tested. This paper proposes an algorithm that is based on a syntactic view, that explores the properties of the normal forms of a given theory to verify its satisfiability. Any propositional theory can be represented either by its conjunctive normal form (CNF) or by its disjunctive normal form (DNF). The proposed algorithm, given a propositional theory represented by a CNF, calculates, using a specially designed representation, the minimal DNF, where minimal is defined as the smallest set of non contradictory, non subsumed dual clauses. Each one of the minimal dual clauses represents (minimally) a set of semantic assignments that satisfy the theory. Therefore, if we generate all minimal dual clauses, we have a syntactic description of all possible assignments. The main idea is that the number of minimal dual clauses is always less (or in the worst case equal) than the number of assignments and this is especially true for difficult theories. The paper also presents some preliminary experimental results, obtained with a Common Lisp implementation.

1 Introduction

The importance of the propositional logic satisfiability problem (SAT) can be hardly overemphasized: it is the first (and the prototype) NP-complete problem [5], it presents very interesting properties with respect to its complexity behavior [14], analogous, in mathematical terms, to phase transitions in physical systems [11], and it has a wide range of applications, e.g., computer aided design of integrated circuits, logic verification, timing analysis. One of the first Artificial Intelligence problems [22], it has deserved increasing interest in recent years, from science magazines [10] to the most important scientific journals [21].

Most of the research on SAT solving algorithms follows the Davis-Putnam [6] approach, which is based on a *semantic* view that all the possible *assignments* of truth values to propositional symbols should be tested. In this paper, on the other hand, an algorithm is proposed which is based on a *syntactic* view that explores the properties of the normal forms of a given theory to verify its satisfiability. Any propositional theory can be represented either by its *conjunctive normal form (CNF)* or by its *disjunctive normal form (DNF)*. Given an *ordinary formula*

W, i.e., a well-formed expression of the full propositional logic syntax, there are algorithms for converting it into a formula W_c, in CNF, and into a formula W_d, in DNF, such that $W \Leftrightarrow W_c \Leftrightarrow W_d$ (e.g., [25], [26], [27]). To transform a formula from one clause form to the other, only the distributivity of the logical operators $\vee$ and $\wedge$ is needed.

The proposed algorithm calculates, given a propositional theory represented by a CNF W_c, the *minimal* representation of its DNF W_d, where minimal is defined as the smallest set of non contradictory, non subsumed dual clauses. In the literature, the non subsumed set is sometimes called *condensed* [9] and, when inference is also taken into account, *prime implicants* [12, 13]. Each one of the minimal dual clauses represents (minimally) a set of semantic assignments that satisfy the theory. Therefore, if we generate all minimal dual clauses, we have a syntactic description of all possible assignments. The main idea is that the number of minimal dual clauses is always less (or in the worst case equal) than the number of assignments and this is specially true for *difficult* theories, i.e., those near the complexity edge.

In particular, the proposed algorithm can be used to solve the satisfiability problem, if it is terminated when the first minimal dual clause is found, but we are also interested in the complete set of minimal dual clauses for knowledge representation purposes [2]. The goal of this paper is to present the algorithm and to analyze its performance properties. The knowledge representation applications are just sketched and will be the subject of a future paper.

The paper is organized as follows. In Section 2, we introduce some notation for normal forms that explicitly represents the relations between them. In Sections 3 and 4, we describe the proposed algorithm and give some examples. In Section 5, we present some preliminary experimental results obtained with a proof-of-concept Common Lisp implementation of the algorithm. In Section 6, the application of the algorithm as a knowledge representation tool in the field of autonomous agents is sketched. Finally, in Section 7, we conclude and comment upon some ongoing and future work.

2 Theory Representation

Let $P = \{P_1, \ldots, P_n\}$ be a set of propositional symbols and $L = \{\phi_1, \ldots, \phi_{2n}\}$ the set of their associated literals, where $\phi_i = P_j$ or $\phi_i = \neg P_j$. A *clause* C is a *generalized disjunction* [8] of literals: $C = [\phi_1, \ldots, \phi_{k_C}] \equiv \phi_1 \vee \ldots \vee \phi_{k_C}$ and a *dual clause* is a *generalized conjunction* of literals: $D = \langle \phi_1, \ldots, \phi_{k_D} \rangle \equiv \phi_1 \wedge \ldots \wedge \phi_{k_D}$.

A propositional theory $\mathcal{L}(P)$ can be represented by its *conjunctive normal form (CNF)*: $W_c = \langle C_1, \ldots, C_m \rangle$ defined as a generalized conjunction of clauses, or by its *disjunctive normal form (DNF)*: $W_d = [D_1, \ldots, D_w]$ defined as a generalized disjunction of dual clauses.

The fundamental element in the algorithm is called a *quantum* and is defined as a pair (ϕ, F), where ϕ is a literal and $F \subseteq W_c$ is its set of *coordinates* that contains the subset of clauses in W_c to which the literal ϕ belongs. A quantum is noted ϕ^F, to remind us that F can be seen as a function $F : L \rightarrow 2^L$.

During the presentation of the algorithm, it is frequently necessary to refer to the set of negated literals, or quanta, of a given set of literals, or quanta[1]. To simplify the notation, we introduce the notion of *mirror*. The mirror of a quantum ϕ^F, noted $\overline{\phi}^{\overline{F}}$, is defined simply as the quantum associated with the negation of its literal: $\overline{\phi} = \neg\phi$. The quantum attribute mirror can also be seen as function: $\overline{\_} : L \to L$ and, from this point of view, $\overline{F}$ is the composition $F \circ \overline{\_} : L \to 2^L$.

This notation is extended to clauses and dual clauses, such that the mirror (dual) clause $\overline{C}$ of (dual) clause C is defined as the set of mirror literals associated with the literals in C[2].

Any dual clause in the DNF W_d is associated with a set of models, i.e., a set of assignments to the propositional symbols in P, that satisfy it. To each dual clause, we can associate a set of quanta: $\Phi = \langle \phi_1^{F_1}, \ldots, \phi_k^{F_k} \rangle$ such that $\cup_{i=1}^{k} F_i = W_c$, i.e., a dual clause is always associated with a set of literals $L_\Phi = \langle \phi_1, \ldots, \phi_k \rangle$ that contains at least one literal that belongs to each clause in W_c, spanning a path through W_c, and no pair of contradictory literals, i.e., if a literal belongs to L_Φ, its negation is excluded. To avoid the introduction of a new name, we call indistinctly the sets Φ and L_Φ dual clauses.

A set Φ represents a minimal dual clause, if the following condition is also satisfied: $\forall i \in \{1, \ldots, k\}, F_i \not\subseteq \cup_{j=1, j\neq i}^{k} F_j$. This condition states that each literal in L_Φ should represent *alone* at least one clause in W_c, otherwise it would be redundant and could be deleted. Given a theory, the set of all minimal Φ's is associated with the minimal representation of its DNF W_d.

Example 1. Consider the theory, whose CNF is given by:

$$
\begin{array}{ll}
0 : [\neg P_4, P_2, \neg P_3] & 5 : [\neg P_3, P_2, \neg P_1] \\
1 : [P_0, \neg P_2, \neg P_3] & 6 : [\neg P_2, \neg P_1, P_0] \\
2 : [P_0, \neg P_3, P_1] & 7 : [P_4, P_0, P_2] \\
3 : [\neg P_0, \neg P_2, P_3] & 8 : [\neg P_1, P_4, P_3] \\
4 : [\neg P_2, \neg P_3, \neg P_1] & 9 : [\neg P_3, \neg P_1, \neg P_4]
\end{array}
$$

Its minimal DNF has seven dual clauses that can be represented by the following sets of quanta, according to the definitions above:

$$
\begin{array}{l}
0 : \langle \neg P_4^{\{0,9\}}, P_0^{\{1,2,6,7\}}, \neg P_1^{\{4,5,6,8,9\}}, \neg P_2^{\{1,3,4,6\}} \rangle \\
1 : \langle \neg P_1^{\{4,5,6,8,9\}}, P_0^{\{1,2,6,7\}}, \neg P_3^{\{0,1,2,4,5,9\}}, \neg P_2^{\{1,3,4,6\}} \rangle \\
2 : \langle P_4^{\{7,8\}}, \neg P_3^{\{0,1,2,4,5,9\}}, \neg P_2^{\{1,3,4,6\}} \rangle \\
3 : \langle P_2^{\{0,5,7\}}, P_3^{\{3,8\}}, P_0^{\{1,2,6,7\}}, \neg P_1^{\{4,5,6,8,9\}} \rangle \\
4 : \langle \neg P_4^{\{0,9\}}, P_3^{\{3,8\}}, P_0^{\{1,2,6,7\}}, \neg P_1^{\{4,5,6,8,9\}} \rangle \\
5 : \langle P_4^{\{7,8\}}, \neg P_0^{\{3\}}, \neg P_1^{\{4,5,6,8,9\}}, \neg P_3^{\{0,1,2,4,5,9\}} \rangle \\
6 : \langle \neg P_0^{\{3\}}, P_2^{\{0,5,7\}}, \neg P_1^{\{4,5,6,8,9\}}, \neg P_3^{\{0,1,2,4,5,9\}} \rangle
\end{array}
$$

[1] Although clauses, dual clauses and sets of quanta are treated as sets, we note them using [ ] and ⟨ ⟩ according to their class.

[2] It should be noted that, differently from the literal case, the mirror of a clause is not the negation of this clause.

where each quantum is represented in the form: ϕ^F, with F its set of coordinates. For legibility reasons, the clauses in the sets F are represented by their numbers.

3 Simply a Search

The basic idea of the proposed algorithm is, given a propositional theory $\mathcal{L}$ represented by a CNF W_c, calculate the set of all Φ that represent the dual clauses in the minimal DNF W_d. If $\mathcal{L}$ is unsatisfiable then this set will be empty.

This problem can be seen as a search in a state space where each state is represented by an incomplete set Φ, associated with an incomplete dual clause in the minimal DNF W_d, and successor states are generated by adding a new quantum to the set, i.e., a new literal in the dual clause. Each incomplete set Φ has an associated *gap*, defined as the set of clauses to which none of its associated literals belong: $G_\Phi = W_c - \cup_{i=1}^{k} F_i$.

Any quantum, associated with literals that belong to the clauses in G_Φ, is, in principle, a relevant quantum to be added to Φ in order to generate a successor.

A space state search should begin in one or more initial states. A possible choice for these initial states is to select all quanta associated with the literals that belong to one specific clause $C_i \in W_c$. The choice of this clause is a first heuristic decision to be taken, e.g., for random theories choosing the clause that contains the most frequent literal in W_c or the one that contains the literal whose negated form is the most frequent literal in W_c, or some combination of both, seems to be sensible options. Once an initial clause is adopted, the problem reduces to a set of independent search problems, one for each literal in this clause, because any path through W_c must pass through exactly one literal in clause C_i.

Finally, the final states are defined as those that satisfies the condition to be a dual clause, i.e., a path through W_c: $\cup_{i=1}^{k} F_i = W_c$. To calculate the minimal set W_d, a complete search should be done but, if the goal is only to determine the satisfiability of $\mathcal{L}$, then when the first final state is found, the search stops.

3.1 Avoiding Redundancy

To keep disjoint the searches associated with each literal in the chosen initial clause C_i, it is necessary to restrict the simultaneous presence of two literals of C_i in some L_Φ to dual clauses Φ that originate from an initial state associated with only one of them. This means that each state Φ must remember its origins, in the form of a list of *forbidden* quanta X_Φ.

Example 2. Consider the theory of example 1, a possible best clause according to the heuristic discussed above is: $4 : [\neg P_3^{\{0,1,2,4,5,9\}}, \neg P_1^{\{4,5,6,8,9\}}, \neg P_2^{\{1,3,4,6\}}]$, where the literals are already sorted according to some quality criterion. States that originate from the best initial state $\langle \neg P_3^{\{0,1,2,4,5,9\}} \rangle$ can be extended to states that contain either $\neg P_1^{\{4,5,6,8,9\}}$ or $\neg P_2^{\{1,3,4,6\}}$ or both, but states that originate from the second best initial state $\langle \neg P_1^{\{4,5,6,8,9\}} \rangle$ cannot be extended

to states that contain $\neg P_3^{\{0,1,2,4,5,9\}}$ and states that originate from $\langle \neg P_2^{\{1,3,4,6\}} \rangle$ can only be extended to states that do not contain neither $\neg P_3^{\{0,1,2,4,5,9\}}$ nor $\neg P_1^{\{4,5,6,8,9\}}$.

The same strategy can be used to avoid the generation of duplicated states in general. Usually, several quanta would qualify as possible extensions to some given dual clause. We propose to sort them according to the same quality criterion used to sort the quanta in the initial clause and to use the same method to restrict which quanta can be added to its successors. Given a dual clause Φ that can be extended by a set of different quanta, S_Φ, already sorted according to the adopted quality criterion, and two quanta, $\phi_i^{F_i}$ and $\phi_j^{F_j}$ in S_Φ, such that $\phi_i^{F_i}$ is better than $\phi_j^{F_j}$, we allow Φ to be extended by adding first $\phi_i^{F_i}$ and then $\phi_j^{F_j}$, or just by adding $\phi_j^{F_j}$. This implies adding new quanta to the forbidden list of each successor state, when it is generated.

The definition of the *quality criterion* used to sort the quanta is a second heuristic decision to be taken. Just to avoid duplicated states, any fixed arbitrary total order among the literals in L would be enough, because all possible combinations would be verified. But it is possible to find orders that are also complete, but avoid the generation of some combinations, that would, themselves or their successors, eventually be excluded by one of the pruning conditions (see Section 3.2). The information available to support the construction of such an order is: the gap of the dual clause, G_Φ, the coordinates of the quanta in S_Φ and the coordinates of their associated mirror quanta. Let $F_i^G = F_i \cap G_\Phi$ and $\overline{F}_i^G = \overline{F}_i \cap G_\Phi$ be the intersection of the quanta coordinates with the current gap, and $F_{ij} = F_i^G \cap F_j^G$, the intersection of the restricted coordinates of quanta i and j. A tentative set of rules that such an order would have to satisfy is:

- If $\mid F_i^G - F_{ij} \mid > \mid F_j^G - F_{ij} \mid$ then $\phi_i \succ \phi_j$ else $\phi_j \succ \phi_i$.
- If $\mid F_i^G - F_{ij} \mid = \mid F_j^G - F_{ij} \mid$ then, if $\mid \overline{F}_i^G - \overline{F}_{ij} \mid >$ $\mid \overline{F}_j^G - \overline{F}_{ij} \mid$ then $\phi_i \succ \phi_j$ else $\phi_j \succ \phi_i$.

The idea behind these rules is that a literal that covers alone more clauses in the current gap should be tried first and, in the case there are two that cover the same number of clauses, the one whose mirror literal covers the greater number of clauses should be preferred. This seems to be a sensible choice for random theories, but different or more elaborated conditions are surely possible.

The consequence of this redundancy avoiding mechanism is that each newly generated dual clause can be seen as the initial state of a new independent search, eliminating the necessity of backtracking.

3.2 Pruning the Search

Given a dual clause Φ, any new quantum to be included in it should satisfy the following *basic conditions*:

- The relevance condition: a new quantum ϕ_ϕ^F should only be included in Φ if $F_\phi \cap G_\Phi \neq \emptyset$. This condition restricts new quanta only to those that can decrease the gap associated with Φ.
- The non contradiction condition: if $\phi \in L_\Phi$ then $\neg\phi \notin L_\Phi$.
- The condensed condition: $\forall i \in \{1,\ldots,k\}, F_i^* = F_i - \cup_{j=1,i\neq j}^{k} F_j \neq \emptyset$. This condition restricts new quanta only to non-redundant ones. The clauses in the set F_i^* are called the *exclusive coordinates* associated with literal ϕ_i in dual clause Φ.

Example 3. Consider the theory of example 1 and a possible incomplete dual clause found during the search: $\Phi = \{\neg P_1^{\{4,5,6,8,9\}}, \neg P_2^{\{1,3,4,6\}}, \neg P_3^{\{0,1,2,4,5,9\}}\}$. The quantum $P_4^{\{7,8\}}$ qualify as a candidate to extend the dual clause Φ, because its coordinate set $F = \{7,8\}$ intersects the gap of the dual clause, $G_\Phi = \{7\}$. But the exclusive coordinates associated with the quanta in Φ are: $\Phi = \{\neg P_1^{\{8\}^*}, \neg P_2^{\{3\}^*}, \neg P_3^{\{0,2\}^*}\}$
and the inclusion of $P_4^{\{7,8\}}$ would make $\neg P_1$ redundant. Therefore, because of the condensed condition, the dual clause Φ can not be extended by the quantum $P_4^{\{7,8\}}$.

The fact that including one literal in L_Φ imply the impossibility of including its negation leads to restrictions with respect to the clauses in G_Φ, i.e., those clauses that are not yet covered by Φ. These are the *gap conditions*:

- If there is a clause $C \in G_\Phi$ such that $C \subseteq \overline{L_\Phi}$, where $\overline{L_\Phi}$ is the set of the mirror quanta of L_Φ, then Φ contradicts one of the clauses in G_Φ and cannot represent a minimal dual clause.
- If there is a clause $C \in G_\Phi$ such that $\mid C - \overline{L_\Phi} \mid = 1$, i.e., L_Φ contradicts all literals in C except one, then the set L_Φ must contain this remaining literal, otherwise clause C would not be represented in Φ. Therefore, if this remaining literal does not qualify as a valid successor of Φ, according to the preceding conditions, then Φ cannot be extended to represent a minimal dual clause.
- Analogous considerations applies to the case in which there is a clause $C \in G_\Phi$ such that $\mid C \mid > \mid C - \overline{L_\Phi} \mid > 1$. In this case, at least one of the remaining literals in $C - \overline{L_\Phi}$ must qualify as a successor of Φ, according to the preceding conditions, otherwise Φ cannot be extended to represent a minimal dual clause.

The gap conditions can be described in a more principled way. Consider the set: $R_\Phi = \{C - \overline{L_\Phi} \mid C \in G_\Phi \text{ and } C \cap \overline{L_\Phi} \neq \emptyset\}$.

This *set of restrictions* represents, in the form of a logical theory in CNF, the gap conditions of the incomplete dual clause Φ. If the first gap condition is verified, then the empty clause belongs to R_Φ, which is, therefore, contradictory, and Φ can not be extended to represent a minimal dual clause. In the case of the second and/or third gap conditions, R_Φ must be coherent with respect to L_Φ and internally coherent, i.e., R_Φ should not contain a pair of contradictory

unitary clauses. Some elements of R_Φ may also be redundant, i.e., if there are clauses $[\phi] \in R_\Phi$ and $C \in R_\Phi$, such that $\phi \in C$, then C is redundant. In order to better detect dual clauses that would become eventually contradictory, because of gap conditions, the minimal CNF of the theory R_Φ should be calculated for each newly generated dual clause Φ.

Example 4. Consider the search for dual clauses of the theory of example 1 at the moment in which the quantum $\neg P_0^{\{3\}}$ is considered as a possible extension of the incomplete dual clause $\Phi = \langle \neg P_1^{\{4,5,6,8,9\}} \rangle$, that has gap $\{0, 1, 2, 3, 7\}$ and list of forbidden quanta $\{\neg P_3^{\{0,1,2,4,5,9\}}\}$. The negation of the literals associated with the new set of quanta – $\langle \neg P_0^{\{3\}}, \neg P_1^{\{4,5,6,8,9\}} \rangle$ – appear in four clauses – 1, 2, 6 and 7 –, clause 6 is not in the gap, i.e., one or more of the literals in L_Φ occur in it. The remaining clauses are: $1 : [P_0, \neg P_2, \neg P_3]$, $2 : [P_0, \neg P_3, P_1]$ and $7 : [P_4, P_0, P_2]$.

In clause 1, $\neg P_0$ imply that the dual clause must include $\neg P_2$ or $\neg P_3$. In clause 2, $\neg P_0$ and $\neg P_1$ imply that the dual clause must include $\neg P_3$. In clause 7, $\neg P_0$ implies that the dual clause must include P_2 or P_4. The simplified clauses are: $1 : [\neg P_2, \neg P_3]$, $2 : [\neg P_3]$ and $7 : [P_4, P_2]$.

The new clause 1 is subsumed by the new clause 2. Therefore, the theory R_Φ is given by $R_\Phi = \langle [\neg P_3^{\{0,1,2,4,5,9\}}], [P_4^{\{7,8\}}, P_2^{\{0,5,7\}}] \rangle$.

The fact that $\neg P_3$ is in the forbidden list indicates that dual clause Φ can not be extended with $\neg P_0$, because of the gap conditions.

A third heuristic decision concerns the order in which these conditions should be tested. The order they are presented already proposes a possible priority, but the best order is clearly theory dependent. In the case of an implementation, the computational cost associated with testing each condition should also be taken into account.

3.3 Failure Propagation

The pruning conditions above are local conditions, in the sense that they depend only on information associated with one dual clause Φ. Quanta not satisfying the first two basic conditions can be easily avoided[3], but the third basic condition and the gap conditions are more complex. Failure that results from the condensed condition is a consequence of the specific composition of the set of quanta in the dual clause plus the new quantum and, according to section 3.1, this specific combination occurs only once.

This is not the case of the failures that result from the gap conditions. In this case, only a limited number of quanta (typically one or two) are responsible for the failure, and whenever this combination occurs it will cause a failure. In particular, all dual clauses that are generated from the same dual clause as the

[3] The non contradiction condition test can also be implemented using the list of forbidden quanta, X_Φ, it is only necessary to add to this list the mirror of each quantum included in the dual clause.

one in which the failure was detected share this fatal combination. It is possible to suitably update the forbidden list of this original dual clause to avoid testing these future failures in its successors.

A further way of propagating failure is the following: given a dual clause Φ and its set of possible extensions S_Φ, a set of new dual clauses is generated by including some of the quanta in S_Φ into Φ, call this set of used quanta S'_Φ. If the quanta in $S_\Phi - S'_\Phi$ were refused as successors of Φ, they will also be refused as successors of all its successors, therefore we can add them to the forbidden list of all successors of dual clauses of Φ.

3.4 The Algorithm

In the beginning of this section, the problem was defined as a search in a state space. This search is solved using a standard A* algorithm. To access the problem the search algorithm needs three interface functions: the initial state, the final state and the state successors. The initial state is defined as the best clause C_i in the theory $\mathcal{L}$, i.e., the clause whose literals coordinates cover the greatest number of clauses in W_c. To start the search, the literals in the best clause are sorted, i.e., the literals that represent more clauses in W_c are used first. A state Φ is final if $G_\Phi = \emptyset$, i.e., the coordinates of the associated literals cover the set W_c. The successor states are generated by the following algorithm:

Successors(Φ)
0. Initialize the successor list: $\Omega \leftarrow \emptyset$.
1. Determine the set of possible extensions:
   $\Theta \leftarrow \{\phi^F \mid \phi \in C \text{ and } C \in G_\Phi\} - X_\Phi$.
2. Sort Θ according to the *quality criterion* $\succ$ (see Section 3.1).
3. Verify the satisfiability of the clauses in the set of restrictions:
   if $\exists C \in R_\Phi, \Theta \cap C = \emptyset$ *then return* $\emptyset$.
4. Main loop: $\forall \phi^F \in \Theta$ do, let $\Phi^+ \leftarrow \Phi \cup \{\phi^F\}$
   if $\forall \phi_i^{F_i} \in \Phi, F_i^* \not\subset F$ — exclusive coordinates are compatible.
   and $\emptyset \notin R_{\Phi^+}$ — new restrictions are not contradictory.
   and $\forall C \in R_{\Phi^+}, C \not\subset X_\Phi$ — new restrictions are compatible with the forbidden list.
   then $\Omega \leftarrow \Omega \cup \{\Phi^+\}$ — create a new state.
5. *return* Ω.

Fig. 1. Successor Function

One important feature does not appear explicitly in the algorithm of figure 1: How the forbidden list to be associated with a new state – X_{Φ^+} – is generated. The process is presented in Section 3.3. The algorithm is trivially correct and complete, because it is an implementation of the dual transformation with minimization, which is correct and complete by definition.

4 Failure Communication

Further improvement can be obtained if, besides propagating failure to successors, states "communicate" failure to all other active states in the search to which this specific failure is relevant. To accomplish this, a communication channel is necessary. We propose to add a new attribute to the quantum that contains, at each moment, the set of incomplete dual clauses to which the quantum belongs. Using this information the active dual clauses in the search that share the relevant quanta with the current dual clause can be identified.

Given a quantum ϕ^{F_ϕ}, we define the attribute F_ϕ^d as the set of incomplete dual clauses, not still processed by the search algorithm, to which the quantum belongs. When the search is completed, only minimal dual clauses remain and the attribute F_ϕ^d becomes the dual coordinates of the quantum, i.e., the set of all dual clauses to which the quantum belongs.

We are yet studying the heuristic potential of the use of the attribute F_ϕ^d, but, initially, we propose to use this information in two cases: when a failure occurs because of the gap conditions and when a new minimal dual clause is generated.

When a gap condition failure occurs in dual clause Φ, all the incomplete dual clauses associated with its quanta are selected. Those dual clauses that share the restriction associated with the gap condition, receive a communication that, if the quanta that fire this specific gap condition appear as candidates to extend the dual clause, a failure should occur. More formally, let $C \in W_c$ be the clause that fired the gap condition, this means that $C - \overline{L_\Phi}$ is not allowed in Φ. The dual clauses selected to receive the communication are those that share this restriction. The contents of the communication is the set of literals $\overline{C} \cap L_\Phi$ and its effect is that, whenever the last of these literals is considered to extend the dual clause, the search branch fails.

When a new minimal dual clause is generated, it may be the case that it has neighbors, i.e., other minimal dual clauses that differ from the one found by only one quanta. The exclusive coordinates of the quanta in the dual clause can be used to search for these neighbor solutions. The idea is, given one quantum in the minimal dual clause, to find another quantum that does not belong to the dual clause, is compatible with it and whose coordinates cover the exclusive coordinates of the given quantum.

Once the set of new solutions is constructed, all incomplete dual clauses that share quanta with them receive a communication that these specific combination of quanta was already found and that any search should stop before reproducing it.

5 Results

In order to test the relevance of the ideas discussed above, an experimental system was developed in the programming language Common Lisp [28]. This system includes a function that implements the proposed algorithm as defined

in Section 3, without the failure communication mechanism (see Section 4). The algorithm was implemented with no optimization concerns, using Lisp structures and plain lists as data structures.

To give an idea of the absolute performance of the algorithm, we used the set of benchmark theories available at:
`http://www.intellektik.informatik.tu-darmstadt.de/SATLIB/`,
with 20, 50, 75 and 100 propositional symbols and 91, 218, 325 and 430 clauses, respectively.

The algorithm was tested both as decision procedure, i.e., halting at the first dual clause found, and as a generator of complete dual clause sets. Some of the obtained results are shown in tables 1, 2, 3 and 4, where *Time first* corresponds to the time until the first minimal dual clause is found, *Time all* the time needed to calculate the complete set of minimal dual clauses, *Calls first* and *Calls all* are the number of internal recursive calls for both the first and all clauses solutions and $\mid W_d \mid$ is the size of the minimal dual clause set. Each unit of time in these tables corresponds to 0.01 seconds. It is interesting to note that roughly theories with smaller dual clause sets are more difficult than theories with bigger ones, in the sense of it takes more time to find the first dual clause.

Problem	Time first	Calls first	Time all	Calls all	$\mid W_d \mid$
uf20-0110	9	327	31	1730	21
uf20-0111	8	301	19	819	5
uf20-0112	9	250	9	383	3
uf20-0113	7	260	8	330	1
uf20-0114	8	249	17	833	6
uf20-0115	8	229	16	693	4
uf20-0116	7	243	14	569	2
uf20-0117	7	218	7	255	1
uf20-0118	9	315	19	1342	14
uf20-0119	9	360	10	433	2

Table 1. Benchmark uf20-91

We also tested the program with some unsatisfiable theories of the benchmark uuf50-218. The results are shown in table 5.

The obtained results, although the limited range of parameters and low statistical significance, seems to indicate that the approach is promising. All the results were obtained with a compiled version of the system in the *CMU Common Lisp* [17] running on a *Celeron 500MHz, 256Mb*. The sources of the system as well as the data files will be made available on the Internet when appropriate.

6 Knowledge Representation

We intend to use propositional logic theories to represent world knowledge of autonomous robots. *Autonomous systems* should present some characteristics,

Problem	Time first	Calls first	Time all	Calls all	$\mid W_d \mid$
uf50-0110	231	1547	1433	17928	27
uf50-0111	218	1759	310	3889	1
uf50-0112	215	1521	338	5576	4
uf50-0113	234	1518	549	6503	9
uf50-0114	477	5307	5372	80159	244
uf50-0115	361	3503	1935	25558	40
uf50-0116	232	2248	747	8891	15
uf50-0117	213	1656	3560	55777	191
uf50-0118	254	2165	377	4778	7
uf50-0119	242	1732	1345	17953	24

Table 2. Benchmark uf50-218

Problem	Time first	Calls first	Time all	Calls all	$\mid W_d \mid$
uf75-010	1153	4141	122304	890578	1025
uf75-011	2377	10870	5940	39255	9
uf75-012	2111	16525	3029	28794	2
uf75-013	1915	8693	18028	113412	132
uf75-014	1154	6164	10131	96129	157
uf75-015	1673	12596	6301	49012	16
uf75-016	1648	7703	2596	20731	6
uf75-017	1032	3555	36273	256330	177
uf75-018	2154	12685	26104	258697	925
uf75-019	1712	10406	1714	10707	2

Table 3. Benchmark uf75-325

such as adaptation and self-organization, that allow them to deal with unexpected situations and to handle complex tasks, without human interference [7, 23]. To acquire and process information is one of the most important activities to support these characteristics. In recent works in the mobile robotics domain, it is possible to perceive a tendency towards hybrid approaches [1] that joins the best features of the planning [16] and sensor-based approaches [3]. The hybrid approach supplies an "intelligent" behavior to the robot, with human like capabilities, such as learning and adaptation. In this sense, Artificial Intelligence techniques have been used in mobile robotics to provide robots with intelligent behavior, mainly in navigation and map building problems [19, 29, 32]. There are also some applications in sensor fusion and control [30, 15].

We intend to use propositional logic to represent the world knowledge necessary to implement intelligent behavior in mobile robots [34, 33] in such a way that they can be considered cognitive autonomous agents. These cognitive agents should be able to learn characteristics of the world, to generalize their knowledge and to draw inferences upon this knowledge in order to accomplish complex tasks.

Problem	Time first	Calls first	Time all	Calls all	$\mid W_d \mid$
uf100-0110	9866	48883	29554	153574	18
uf100-0111	2700	5903	43735	223867	140
uf100-0112	3388	10667	28655	125415	46
uf100-0113	13149	100660	15131	118886	3
uf100-0114	3948	14042	8782	56146	2
uf100-0115	3401	6458	46507	279033	424
uf100-0116	4132	8909	35260	165054	66
uf100-0117	9694	48348	29048	181519	47
uf100-0118	2788	5760	79995	403377	433
uf100-0119	3923	12283	244049	1484480	1835

Table 4. Benchmark uf100-430

Problem	Time	Calls
uuf50-0110	139	2342
uuf50-0111	190	4270
uuf50-0112	109	2176
uuf50-0113	86	1393
uuf50-0114	105	2365
uuf50-0115	112	1947
uuf50-0116	129	2872
uuf50-0117	69	1359
uuf50-0118	141	3006
uuf50-0119	203	3431

Table 5. Benchmark uuf50-218

The adopted model is derived from the generic model for a cognitive agent presented in [2]. This model is based on three hypothesis: (i) Cognition is an emergent property of a cyclic dynamic self-organizing process [20, 31] based on the interaction of a large number of functionally independent units of a few types [4]. (ii) Any model of the cognitive activity should be epistemologically [18] compatible with the Theory of Evolution. That applies not only to the "hardware" components of this activity but also to its "psychological" aspects [35]. (iii) Learning and cognitive activities are closely related and, therefore, the cognitive modeling process should strongly depend on the cognitive agent's particular history [24].

The agents based on this model have three levels: reactive, instinctive and cognitive. The cognitive level could be defined as a set of non-contradictory propositional theories that represent the agent's knowledge about the world. The *states of the world*, relevant to a given theory, are defined as the possible truth assignments to a set of *primitive* propositional symbols that occur in this theory. We suppose that the world drifts along the possible states (i.e., assignments), but changing only one primitive propositional symbol assignment at each moment. The primitive propositional symbols can be *controllable* or *uncontrollable*.

Roughly, uncontrollable symbols correspond to *perceptions*, controllable ones to *actions*.

The agent is embodied in a mobile robot that wanders around the world and *perceives* the *primitive* propositional symbols, through the reactive and instinctive levels. It is important to note that the agent should recognize the situations and abstract similar information, grouping them into concepts, represented by propositional theories. Each concept is associated with a set of dual clauses that are satisfied in the corresponding situations. Using the proposed algorithm we can obtain the CNF associated with this set of dual clauses. This CNF is a rule-based representation of the concept and can be used to control the agent behavior.

Each theory can itself be interpreted as an *abstract* propositional symbol, that may occur in other theories. The idea is that each theory represents some *concept*, just to have a single word to mean either an *object* or a *situation* in the world. From the agent point of view, these concepts are characterized by some patterns of truth assignments, represented by its propositional theories. Therefore, the agent could act in the world through the primitive propositional symbols that it can *control* and update its internal states when the *uncontrollable* primitive symbols propositional change.

The fact that each theory is represented by both, CNF and DNF, provides the agent with a "holographic" representation of the world, where possible future situations and relevant behavior rules are available simultaneously. The goal is to demonstrate that this syntactical representation, it is suitable to implement the necessary cognitive capabilities of a simple autonomous agent.

7 Conclusion

The paper has presented an algorithm to calculate the minimal dual form of a theory and some preliminary results on its application to the random 3SAT problem. The following characteristics of the proposed algorithm make it different from most of the algorithms in the Davis-Putnam [6] thread: (i) The use of an explicit representation of the relations between CNF and DNF. (ii) The use of a syntactic property of the input theory, its set of minimal dual clauses, to guide the search, instead of the possible semantic assignments. (iii) The use of a redundancy avoiding mechanism that eliminates the need of backtracking. (iv) The propagation of failure information from one search point to others search points, in order to avoid useless search effort. Although some results were presented, the main goal of the paper was to present what we believe to be a different approach to the satisfiability problem and to motivate its application in the autonomous agent knowledge representation task.

On going work includes the failure communication implementation and the computational complexity analysis of the algorithm and a new implementation in the C++ programming language, to allow experiments with larger theories and comparisons with other systems. In the future, we also intend to develop a concurrent implementation of the algorithm to explore the fact that each new

generated state of the search can be considered a new initial state of an independent search.

Future work also includes the extension of the algorithm to the first-order logic case and more investigation on the properties of the relation between the two minimal dual forms of a theory. Although the algorithm was defined to calculate the set of minimal dual clauses of a theory, it is absolutely symmetric and it is possible to obtain the minimal CNF, and its associated quanta, just executing the search in the other direction, beginning with the already calculated DNF. We believe that the explicit representation of these "holographic" relations, through the coordinates and exclusive coordinates of the quanta in both normal forms, has a high heuristic potential, specially in the first-order case.

References

1. R.C. Arkin. Towards the unification of navigational planning and reative control. In *AAAI Spring Symposium on Robot Navigation*, 1989.
2. G. Bittencourt. In the quest of the missing link. In *Proceedings of IJCAI 15, Nagoya, Japan, August 23-29*, pages 310–315. Morgan Kaufmann (ISBN 1-55860-480-4), 1997.
3. Rodney A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1):435–453, March 1986.
4. J.-P. Changeux. *L'Homme Neuronal*. Collection Pluriel, Librairie Arthème Fayard, 1983.
5. S.A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, ACM, New York*, pages 151–158, 1971.
6. Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the Association for Computing Machinery*, 7:201–215, 1960.
7. J.A. Fabro. Grupos neurais e sistemas fuzzy - aplicação à navegação autônoma. Master's thesis, UNICAMP - Universidade Estadual de Campinas, February 1996.
8. M. Fitting. *First-Order Logic and Automated Theorem Proving*. Springer Verlag, New York, 1990.
9. Georg Gottlob and Christian G. Fermüller. Removing redundancy from a clause. *Artificial Intelligence*, 61:263–289, 1993.
10. Brian Hayes. Can't can no satisfaction. *American Scientist*, 85(2):108–112, March-April 1997.
11. Tad Hogg, Bernardo A. Huberman, and Colin Williams (eds.). Frontiers in problem solving: Phase transitions and complexity. *A special issue of Artificial Intelligence*, 81(1-2), March 1996.
12. P. Jackson. Computing prime implicants. In *Proceedings of the 10^{th} International Conference on Automatic Deduction, Kaiserslautern, Germany, Springer Verlag LNAI No. 449*, pages 543–557, 1990.
13. A. Kean and G. Tsiknis. An incremental method for generating prime implicants/implicates. *Journal of Symbolic Computation*, 9:185–206, 1990.
14. Scott Kirkpatrick and Bart Selman. Critical behavior in the satisfiability of random boolean expressions. *Science*, 264:1297–1301, 1994.
15. B.J.A. Kröse and Eecen M. A self-organizing representation of sensor space for mobile robot navigation. In *Proceedings of the IEEE/RSJ/GI International Conference on Intelligent Robots and Systems IROS'94*, pages 9–14, 1994.

16. J.C. Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, Boston, MA, 1991.
17. R.A. MaClachlan. *CMU Common Lisp User's Manual*. Carnegie Mellon University, Pittsburgh,PA, 1992.
18. J. McCarthy and P.J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In D. Michie and B. Meltzer, editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, Edinburgh, GB, 1969.
19. J.R Millán. Reinforcement learning of goal-directed obstacles-avoiding reaction strategies in an autonomous mobile robot. *Robotics and Autonomous Systems*, 15:275–299, 1995.
20. E. Morin. *La Méthode 4, Les Idées*. Editions du Seuil, Paris, 1991.
21. M. Mzard, G. Parisi, and R. Zecchina. Analitic and algorithmic solution of random satisfiability problems. *Science*, 297:812–815, August 2002.
22. A. Newell and H.A. Simon. The logic theory machine. *IRE Transactions on Information Theory*, 3:61–79, September 1956.
23. D. Pagac, E.M. Nebot, and H. Durrant-Whyte. An evidential approach to probabilistic map-building. In *IEEE International Conference on Robotics and Automation*, pages 745–750, Minneapolis, Minnesota, April 1996.
24. J. Piaget. *The Origins of Intelligence in Children*. Norton, New York, 1963.
25. W.V.O. Quine. On cores and prime implicants of truth functions. *American Mathematics Monthly*, 66:755–760, 1959.
26. J.R. Slagle, C.L. Chang, and R.C.T. Lee. A new algorithm for generating prime implicants. *IEEE Transactions on Computing*, 19(4):304–310, 1970.
27. R. Socher. Optimizing the clausal normal form transformation. *Journal of Automated Reasoning*, 7(3):325–336, 1991.
28. G.L. Steele Jr. *Common LISP, the Language*. Digital Press, Burlington, 1984.
29. S. Thrun. An approach to learning mobile robot navigation. *Robotics and Autonomous Systems*, 15:301–319, 1995.
30. J.W.M. van Dam, B.J.A. Kröse, and D.C.A. Groen. Neural network applications in sensor fusion for an autonomous mobile robot. in Reasoning with Uncertainty in Robotics, (Dorst, L. and Lambalgen, M. van and Voorbraak, F., ed.), Springer, pp. 263-277, 1996.
31. F.J. Varela. *Autonomie et Connaissance: Essai sur le Vivant*. Editions du Seuil, Paris, 1989.
32. Jerusa M. VAZ and João FABRO. Snnap - sistema neural de navegação em ambientes pré-mapeados. In *IV Congresso Brasileiro de Redes Neurais (CBRN)*, São José dos Campos, SP, 19 a 22 de Julho 1999.
33. P.F.M.J. Verschure. Minds, brains and robots: Explorations in distributed adaptive control. In *Second Brazilian-International Conference on Cognitive Science*, pages 14–17, 1996.
34. P.F.M.J. Verschure, B.J.A. Kröse, and R. Pfeifer. Distributed adaptative control: The self-organization of strutured behavior. *Robotics and Autonomous Systems*, 9:181–196, 1992.
35. R. Wright. *The Moral Animal*. Vintage Books, New York, 1994.

Thoughts about the implementation of the Duration Calculus with Coq

Samuel Colin[1,2] Samuel.Colin@inrets.fr,
Vincent Poirriez[2] Vincent.Poirriez@univ-valenciennes.fr,
Georges Mariano[1] Georges.Mariano@inrets.fr

[1] INRETS*, 20, rue Elisée RECLUS, BP 317 F-59666 Villeneuve d'Ascq Cedex, France
[2] LAMIH**, Le Mont Houy, 59313 Valenciennes Cedex 9, France

Abstract. This work is a derivative of studies about the duration calculus , aiming at deciding whether it is sound to use it as an extension logic for a formal method (namely, the "B method"). Indeed, we wanted to know the feasability and the usability, of such a modal logic implemented in a proof assistant. In this paper, two complementary implementations are described, as well as problems inherited from both sides : the proof system for itself, and the tweaking of the proof assistant.

1 Introduction

We will present the reasons that drove us to the writing of Coq libraries for DC (duration calculus), and to that end we'll do a quick presentation of the B method.

The B method, a formal method, allows the development of safe software, from abstract, mathematical specifications, to computer code that is proved correct with regard to those specifications. The steps going from specifications to code are called refinements. The abstract specifications and the refinements have to be proved correct, through the proof of so-called proof obligations, that are formulas expressed with predicate calculus and set theory, generated from the specifications and the refinements.

While this method has convinced the industrial world, it still has limits, e.g. when dealing with problems having temporal constraints. Some examples of application of the B method to time-constrained problems exist (see for example [1, 2]), but the complexity of the generated proof obligations can easily become confusing for both the automatic theorem prover and the operator who must read the formulas having failed with this prover.

Methods involving the extension of the B method also exist ([3]), and we have chosen to study the extension of the logic used by B to Duration Calculus. To do so, we needed a proof tool able to handle both normal B logical formulas, and DC formulas. Coq having several set theory libraries at disposal, we chose it to build a library for DC.

In the section 2 we'll present the duration calculus, then in section 3 the Coq proof assistant. In section 4 we will highlight interesting points about the implementation of DC with Coq, and we'll conclude in sections 5 and 6.

* Institut National de REcherche sur les Transports et leur Sécurité
** Laboratoire d'Automatique, de Mécanique, et d'Informatique industrielles et Humaines

2 Duration Calculus

This section won't present an in-depth description of the Duration Calculus, we will rather focus on peculiar properties, which will be of interest in the other sections.

2.1 History

The Duration Calculus was first presented in [4], as a temporal logic based on IL (Interval Logic) [5]. Ever since, numerous extensions were proposed for DC ([6, 7]), allowing to express more and more complex properties of real-time systems. An in-depth survey of DC and its properties can be found in [8].

2.2 Syntax

Let X_i be a *propositional temporal letter* (interpreted as a boolean function over time intervals), P_i a state variable (interpreted as a boolean-valued function over time), $x, y, \ldots$ global variables (interpreted as real numbers), f_i functions and R_i relation symbols. Usually the functions are the standard arithmetic ones $(+, *)$ and the relations also are the usual ones $(=, \leq)$. The syntax of DC formulas is (functions and relations might be noted with prefix or infix notation, as syntax is not our main concern) :

formula ::= Atom | $\neg$ formula | formula $\vee$ formula | formula$\frown$formula | $\exists x$.formula
Atom ::= **true** | X | R(term,..., term)
term ::= x | ℓ | $\int$state | f(term,..., term)
state ::= 0 | 1 | P | state $\vee$ state | $\neg$ state

The additions of IL to predicate calculus are the special variable ℓ and the *chop* connector $\frown$. This connector chops a formula into two formulas representing the valid predicates on the first part of the time interval and the second part, respectively. The ℓ variable represents the length of the current time interval, i.e. the value of ℓ is influenced by the *chop* connector.

The additions of DC to IL are represented by the duration operator $\int$ and the state expressions. These have the expressive power of propositional logic, and the duration operator allows to express properties on these states and on logical relations between them.

2.3 Semantics

The most direct way to interpret DC formulas is to do so over time intervals. Let the following interpretations functions and definitions domains be:

- Time, usually represented by the real numbers $\mathbb{R}$
- TimeInterval = $\{(b,e)|b,e \in \text{Time} \wedge b \leq e\}$
- Val = term $\rightarrow$ TimeInterval $\rightarrow \mathbb{R}$
- ValState = state $\rightarrow$ Time $\rightarrow \{0,1\}$
- $\mathcal{I}$: ValState

- $\mathcal{V}$: Val
- I : formula $\rightarrow ((\text{Val} \times \text{ValState}) \times \text{TimeInterval}) \rightarrow \{true, false\}$

For readability reasons, in the description of I, $\mathcal{V}$ and $\mathcal{T}$ are implied. The same remark applies for the description of $\mathcal{V}$ and $\mathcal{T}$.

$$
\begin{array}{ll}
I(X)([b,e]) & = X_I([b,e]) \\
I(R(\theta_1,\ldots,\theta_n))([b,e]) & = R(c_1,\ldots,c_n) \text{ where } c_i = \mathcal{V}(\theta_i)([b,e]) \\
I(\neg\phi)([b,e]) & = \neg I(\phi)([b,e]) \\
I(\phi_1 \vee \phi_2)([b,e]) & = I(\phi_1)([b,e]) \vee I(\phi_2)([b,e]) \\
I(\exists x.\phi)([b,e]) & = I(\phi_{\mathcal{V}''})([b,e]) \text{ where } \mathcal{V}(y) = \mathcal{V}'(y) \text{ for } y \neq x \\
I(\phi_1{}^\frown\phi_2)([b,e]) & = \exists m.(I(\phi_1)([b,m]) \wedge I(\phi_2)([m,e])) \text{ for } m \in [b,e] \\
I(\textbf{true})([b,e]) & = \text{true}
\end{array}
$$

$$
\begin{array}{ll}
\mathcal{V}(x)([b,e]) & = x \\
\mathcal{V}(\ell)([b,e]) & = e - b \\
\mathcal{V}(f(\theta_1,\ldots,\theta_n))([b,e]) & = f(c_1,\ldots,c_n) \text{ where } c_i = \mathcal{V}(\theta_i)([b,e]) \\
\mathcal{V}(\int S)([b,e]) & = \int_b^e \mathcal{T}(S)(t)dt
\end{array}
$$

$$
\begin{array}{ll}
\mathcal{T}(0)(t) & = 0 \\
\mathcal{T}(1)(t) & = 1 \\
\mathcal{T}(S \vee T)(t) & = \begin{cases} 0 \text{ if } \mathcal{T}(S)(t) = 0 \text{ and } \mathcal{T}(S)(t) = 0 \\ 1 \text{ otherwise} \end{cases} \\
\mathcal{T}(\neg S)(t) & = 1 - \mathcal{T}(S)(t) \\
\mathcal{T}(P)(t) & = P_{\mathcal{T}}(t)
\end{array}
$$

A proviso is added for the state variables, which are interpreted as functions over time : for the functions to be integrable, they need to be *finitely variable* over the considered time interval. For example, the function :

$$P_{\mathcal{T}}(t) = \begin{cases} 0 \text{ if } t \text{ is irrational} \\ 1 \text{ otherwise} \end{cases}$$

2.4 Examples

Some examples are inspired from [8]:

1. Let the state variables *Gas* and $Flame$ be the expressions of the event "gas is produced" and "flame exists", respectively. Then this DC formula states that during the non-zero time interval, each time gas is produced, the flame must be present :

$$\int(Gas \Rightarrow Flame) = \ell \wedge \ell > 0$$

2. The formula $\ell = 10{}^\frown\ell = 5$ states that in the first part of the time interval is 10 time units long, and the second part 5 time units long.

3. **true**$^\frown(\phi^\frown$**true**$)$ states that the ϕ formula is valid in some sub-interval. This special construction is also noted $\Diamond\phi$, and is comparable with the $\Diamond$ one can find in other temporal logics.
4. Similarly, the formula $\neg\Diamond(\neg\phi)$ is noted $\Box\phi$, and is interpreted as : "for any time sub-interval, the ϕ formula is valid".

Now an example of the semantics of DC, over a given time interval $[b,e]$, with $e > b$:

Example 1. Let's suppose that *Gas* is a state whose value is 1 all over the interval, and $Flame$ a state whose value is 0 in the first half of the [e,b] time interval, 1 in the second. Intuitively, it means that the gas is leaking, before it is set on fire :

$$\begin{aligned} & I(\int(Gas \Rightarrow Flame) = \ell \wedge \ell > 0)([b,e]) \\ & \quad \equiv I(\int(Gas \Rightarrow Flame) = \ell)([b,e]) \wedge I(\ell > 0)([b,e]) \\ & \quad \equiv \mathcal{V}(\int(Gas \Rightarrow Flame))([b,e]) = \mathcal{V}(\ell)([b,e]) \wedge \mathcal{V}(\ell)([b,e]) > \mathcal{V}(0)([b,e]) \\ & \quad \equiv \int_b^e \mathcal{T}((Gas \Rightarrow Flame)(t)dt)([b,e]) = e-b \wedge e-b > 0 \\ & \quad \equiv \int_b^e \mathcal{T}((\neg Gas \vee Flame)(t)dt)([b,e]) = e-b \wedge \text{true} \\ & \quad \equiv \frac{e-b}{2} = e-b \end{aligned}$$

Because $\neg Gas$ and $Flame$ are both 0 in the first half of the interval. The obtained formula is false, as $e > b$. So the given states *Gas* and $Flame$ don't fulfil the requirement.

2.5 Proof system

We will only underline in this section some hard points of the proof system of [8], on which we have based the implementation described in section 4. Now for some definitions beforehand:

Definition 2. A DC formula is called *rigid* if it doesn't contain any state variable, propositional letter or ℓ symbol. It is otherwise called *flexible*

Definition 3. A DC formula is called *chop free* if the $^\frown$ doesn't occur in the formula

Definition 4. The term θ is *free for x in* ϕ if x doesn't occur freely in ϕ within the scope of the quantified variable *y*, *y* occurring in θ

This last definition is used later in a side-condition to address the problem of variable instanciation.

The axioms of the proof system are distributed between the ones coming from IL, and the ones coming from DC. For example:

Example 5. Some IL axioms:

$\ell \geq 0$	The length of a time interval can't be negative
$\phi^\frown\psi \Rightarrow \phi$ if ϕ is rigid	If a rigid formula is valid on a part of an interval, it is also valid on the whole interval, as it is not influenced by temporal variables or symbols

Example 6. Some DC axioms:

$\int 1 = \ell$	The "always true" state lasts the whole time interval
$\int S_1 = \int S_2$ if $S_1 \Leftrightarrow S_2$ holds in propositional logic	Equivalent states have the same duration

Some inference rules are added, and the ones inherited from predicate calculus are modified.

Two noticeable things about the proof system is that:

1. Side-conditions might require non-trivial analysis of the involved formulas
2. Inference rules doesn't hold hypotheses, as in sequent calculus, for example. Thus some of them won't be valid if coded "as is" in a prover (these problems have already been solved in [9], see section 4 for more information).

For example:

Example 7. Some DC inference rules (inherited from IL inference rules):

$$\frac{\forall x.\phi(x)}{\phi(\theta)} \text{ if } \theta \text{ is free for } x \text{ in } \phi(x) \text{ and } \begin{cases} \text{either } \theta \text{ is rigid} \\ \text{or } \phi(x) \text{ is chop free} \end{cases}$$

$$\frac{\phi \Rightarrow \psi}{(\phi^\frown \Phi) \Rightarrow (\psi^\frown \Phi)}$$

3 The Coq proof assistant

3.1 Presentation

Paraphrasing the Coq reference manual (see [10]), "Coq is a proof assistant for higher-order logic, allowing the development of computer programs consistent with their formal specification".

Coq's logical language is based on the *Calculus of Inductive Constructions*, a variety of type theory, which allows manipulations of higher order terms in a consistent framework, ensured by type-checking of formulas. Still citing the Coq reference manual, "It is possible to understand the Calculus of Inductive Constructions at a higher level, as a mixture of predicate calculus, inductive predicate definitions presented as typed PROLOG, and recursive function definitions close to the language ML".

One of the noticeable properties of Coq (even if not useful for the comprehension of next section) is its ability to extract programs from proofs, endorsing hereby the Curry-Howard isomorphism.

3.2 Description

Coq may be used interactively, in a console toplevel or in an editor with a dedicated mode (e.g. ProofGeneral for Emacs), or with the Coq compiler (producing a compiled file containing the proved theorems and the corresponding lambda-terms, to speed up the development of complex proofs requiring lots of lemmas).

Syntax As Coq is first meant to be used interactively, it provides a natural feeling for building proofs. The allowed terms of Coq can be subdivided into three categories:

1. The *vernacular* terms : these are the commands that allow one to add definitions, telling Coq we want to prove a theorem, or changing Coq's behaviour.

   Example 8. Some vernacular commands :
   - `Theorem ModusPonens:(A,B:Prop)(A /\ (A -> B)) -> B.` tells Coq we want to prove the formula $\forall A,B(A \wedge (A \Rightarrow B) \Rightarrow B)$.
   - `Definition excluded_middle := (A:Prop) A \/ ~A.` allows to associate the excluded middle formula to a variable named `excluded_middle`.
   - `Quit.` allows to quit Coq's toplevel

2. The tactics : these are the commands used *during* the proof of a theorem, to specify what kind of rule we want to use, e.g. introduction rules, elimination rules, apply a theorem,etc. There are also higher level tactics used to describe complex but repetitive proof commands.

   Example 9. Some tactics commands :
   - `Intros x P.` tells Coq to apply introduction rules to the current goal, and naming the obtained hypotheses x and P.
   - `Repeat Left.` allows to choose in a goal the leftmost innermost term. For example it would produce the goal P_1 if applied to the goal $((...(P_1 \vee P_2)... \vee P_{n-1}) \vee P_n)$.

3. The grammar redefinition language : it allows the user to define its own grammar for new terms or definitions he introduced, and even for complex tactics. There are example of it in section 4

3.3 Examples

Let's show a proof example with Coq :

Example 10. This example is a possible proof for the formula (A,B,C being propositions) : $\forall A,B,C(A \wedge B) \vee (A \wedge C) \Rightarrow A$

```
Theorem easyproof:(A,B,C:Prop)(A /\ B) \/ (A /\ C) -> A.
Intros.
Elim H.
Intros.
Elim H0.
Intros.
Assumption.
Intros.
Elim H0.
Intros.
Assumption.
Qed.
```

The proofs are done the top-down way. This corresponds to the following proof tree (where inference steps are annotated with the tactics commands) :

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{H,H_0,H_1:A,H_2:B\vdash A}\texttt{Assumption.}}{H,H_0\vdash A\Rightarrow B\Rightarrow A}\texttt{Intros.}}{H,H_0:A\wedge B\vdash A}\texttt{Elim H0.}}{H\vdash A\wedge B\Rightarrow A}\texttt{Intros.} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{H,H_0,H_1:A,H_2:C\vdash A}\texttt{Assumption.}}{H,H_0\vdash A\Rightarrow C\Rightarrow A}\texttt{Intros.}}{H,H_0:A\wedge C\vdash A}\texttt{Elim H0.}}{H\vdash A\wedge C\Rightarrow A}\texttt{Intros.}}{\dfrac{H:(A\wedge B)\vee(A\wedge C)\vdash A}{\vdash(A\wedge B)\vee(A\wedge C)\Rightarrow A}\texttt{Intros}}\texttt{Elim H.}$$

`(x:sort), /\, \/, ->, ~`	These are the symbols for universal quantification (for a variable x of sort `sort`), conjunction, disjunction, implication, negation respectively. More generally connectors of the usual logic are represented by visually similar ASCII symbol.
`Intros`	Put the premisses of the goal in the hypotheses
`Elim H`	Apply an elimination rule for the given formula H
`Assumption`	Attempts to solve the current goal by telling Coq the goal is also present in the current hypotheses
`Qed`	Ends the proof and saves the generated proof term.

An additional reason that made us choose Coq, was the disponibility of a library of definitions and theorems for real numbers, as DC can be used as well in the domain of integers as in the domain of real numbers.

4 Two paths towards an implementation of DC with Coq

The proof system of DC presented in [8] isn't a sequent-style system, and the ℓ variable is context-dependent w.r.t. the $\frown$ connector. Thus the inference rules and axioms of this proof system might raise incompatibilities with the ones already present in Coq (see section 4.2). That's why we have defined two approaches so as to implement DC's proof system in Coq.

Despite those different approaches, in each case grammar redefinitions had been done, in order to ease the proof process. For example, the *diamond* meaning "for some sub-interval" (see 2.4) has been given the ASCII symbol <>.

Side-conditions involving the analysis of the formula, also had been coded with inductive definitions or functions.

We will focus in the following, on the IL part of the DC implementations, as DC-specific axioms and inference rules didn't bring much problems.

Also notice that problems we'll speak about have been solved in [9], but Isabelle/HOL being a meta-logic, it's thus easier to build a full logic in it than coping peculiarities of already existent logic one can base the implementation on. In this point of view, the shallow-embedded Coq implementation of DC in section 4.1 is comparable to the Isabelle/HOL one.

4.1 Shallow-embedded implementation

In this implementation, we simply added the missing connectors of DC with their correct type, and defined the properties of these connectors through axioms and inference rules, as described in [8].

The development libraries have been divided by logical system and by functionality : there are an IL axioms library, an IL syntax definition and an IL theorems library, and based on that, a DC axioms library, a DC syntax definition library and a DC theorems library. Hence we can say that the shallow-embedded implementation is *modular*, as one can develop a DC extension (e.g. [11]) without having to know the internals of the DC library.

Example 11. Here are the definitions of the connectors, along with grammar and syntax redefinitions. `R` is the type of real numbers, `Prop` the type of propositions, and the quotes around the definition of `point` helps Coq's parser knowing that it requires the axioms and definitions of the real numbers library.

```
Parameter l:R.
Parameter chop:Prop->Prop->Prop.
Definition point:=``l == R0``.
Definition sometime:=[P:Prop](chop True (chop P True)).
Definition always:=[P:Prop]~(sometime ~P).

Grammar constr constr5:=
  chop [constr5($c1) "^^" constr5($c2)] -> [ (chop $c1 $c2) ].

Syntax constr level 5 :
  chop [ $t1 ^^ $t2 ] -> [ [<v 0> $t1:L "^^" $t2:L ] ].

Grammar constr constr2:=
  timepoint ["[[]]"] -> [ point ]
 |sometime ["<>" constr2($c)] -> [ (sometime $c) ]
 |always ["[]" constr2($c)] -> [ (always $c) ].


Syntax constr level 2 :
  timepoint [ point ] -> [ [<v 0> "[[]]"] ]
 |sometime [ (sometime $t) ] -> [ [<h 0> "<>" $t:L] ]
 |always [ (always $t) ] -> [ [<h 0> "[]" $t:L] ].
```

With these syntax redefinitions, we can write axioms two ways, for example:

```
Axiom chop_assoc:(p,q,r:Prop)(chop (chop p q) r) <-> (chop p (chop q r)).
```

which is equivalent to:

```
Axiom chop_assoc:(p,q,r:Prop)((p ^^ q) ^^ r) <-> (p ^^ (q ^^ r)).
```

As Coq inference rules are hard-coded in its core, the inference rules for DC are defined through axioms.

Example 12. For example, the necessitation rule of DC. After having defined the axioms, we also define tactics so the user has the impression to use an inference rule instead of a simple axiom.

```
Axiom necessitation_left:(p,q:Prop)p -> ~(~p ^^ q).
Axiom necessitation_right:(p,q:Prop)p -> ~(q ^^ ~p).

Tactic Definition NecessitationLeft:=Apply necessitation_left.
Tactic Definition NecessitationRight:=Apply necessitation_right.
```

The specific side-conditions of DC are coded by inductive definitions:

Example 13. The rigidity side-conditions is (not all the induction cases are represented):

```
Inductive rigid:Prop->Prop:=
 | rig_true : (rigid True)
 | rig_false : (rigid False)
 | rig_chop : (p,q:Prop) (rigid p)/\(rigid q) -> (rigid (chop p q))
 | rig_imp : (p,q:Prop) (rigid p)/\(rigid q) -> (rigid (p -> q))
 | rig_and : (p,q:Prop) (rigid p)/\(rigid q) -> (rigid (p /\ q))
...
```

The cases when a formula is not rigid do not belong to the inductive definition, so as when trying to prove the rigidity of a formula, the inference system will be blocked. Then axioms involving rigidity are written e.g. as follows:

```
Axiom rigid_chop_left:(p,q:Prop)(rigid p)->(p ^^ q)->p.
```

Then the theorems are proved the normal way in Coq.

Example 14. The Coq proof of $\Box(A \Rightarrow B) \Rightarrow (\Box A \Rightarrow \Box B)$

```
Theorem always_distr_implies:(A,B:Prop)[](A->B) -> ([]A -> []B).
Unfold always; Unfold sometime.
Intros A B alw_A_B alw_A.
Unfold not; Intros som_nB.
Apply alw_A_B.
Monotony; Intros nB__True.
Monotony; Intros nB.
Unfold not; Intros A_B.
Apply alw_A.
Monotony; Intros nB__True2.
Monotony; Intros nB2.
Tauto.
Qed.
```

Note that the proviso of term freedom for variables ("The term θ is *free for x in* ϕ" in the section 2.5) wasn't necessary to define, as this proviso actually prevents abusive scoping of newly introduced variables. Indeed Coq is aware of variables bindings at any level.

Unfortunately, problems that can be easily solved in [9] can't have an easy solution here : as we don't define the whole logical system "from scratch", we are forced to deal with the already present logical connectors and inference rules.

Let's take for example the problem of equality, described in [9, p.21]. Usually one term can be replaced by another if they are equal. But this is not true for Duration calculus, for example:

Example 15.

$$\overline{\ell = 3, \ell = 3 \Rightarrow \ell = 2^\frown \ell = 1 \vdash 3 = 3 \Rightarrow 3 = 2^\frown 3 = 1}$$

This problem is addressed by constraining the equality with an "always" operator, but this can't be done for the shallow-embedded DC in Coq : the equality is already defined (this is Leibniz's one), and redefining it would be contrary to a shallow-embedding approach.

Other similar problems involve the $\frown$ operator and ℓ (and, by extension, the duration operator, as one of the axioms states that $\int 1 = \ell$).

One way to solve such a problem, is to exploit the ability of Coq to allow one to define "plugins" so to redefine tactics and inference rules through the language Coq is written in, but this solution is a difficult one, and makes us lose the advantages deep-embedding could offer us (i.e. using already present logical connectors and inference rules without worrying).

4.2 Deep-embedded

In this implementation, all operators involved in the logic (even the ones coming from predicate calculus) are redefined. One could compare this approach to the Isabelle/HOL's one [9], as we here use Coq mostly as an inference engine. The main advantage of this approach is the absence of conflict between the implemented proof system and the proof system of the tool itself.

Example 16. Definition of a formula:

```
Inductive Formula:Type :=
 | FTrue : Formula
 | FLetter : Name -> Formula
 | FNot : Formula -> Formula
 | FOr : Formula -> Formula -> Formula
 | FExists : (DCTerm -> Formula) -> Formula
 | FChop : Formula -> Formula -> Formula
 | Flt : DCTerm -> DCTerm -> Formula
 | Feq : DCTerm -> DCTerm -> Formula
.
```

Then the validity of a formula, is stated by proving : `plvalid`*formula*. `plvalid` is an interpretation function defined by axioms. One could write an interpretation function using the original semantic of DC, i.e. interval numbers.

Then with the help of syntax redefinition allowed by Coq, one can write the formulas two ways:

Example 17. 1. `Axiom pos_interval:%``l>=0``%.`
2. `Axiom pos_interval:(plvalid (Fge length (RVal R0))).`

The special functions needed for the checking of some side-conditions are also coded inductively, but with functions this time:

Example 18. In this example, due to the nature of the existencial quantification, we do an instanciation so we can analyse further the formula.

```
Fixpoint chop_free [f:Formula]:Prop:=
Cases f of
| FTrue => True
| (FLetter _) => True
| (FNot g) => (chop_free g)
| (FOr g h) => (chop_free g) /\ (chop_free h)
| (FExists z) => (chop_free (z (RVal R0)))
| (FChop _ _) => False
| (Flt u v) => True
| (Feq u v) => True
end.
```

Then, when a proof requires to state the rigidity of a formula, one simply has to make a simplification to find out if the formula is rigid or not (`True` of `False` after the simplification, respectively).

So, even if the deep embedded approach is still at early stages, we already have positive results for this implementation:

1. The inductive definitions of formulas gives us an easier definition of side-conditions
2. The grammar and syntax redefinitions help us to have a readable system
3. The ability to interpret formulas over miscellaneous paradigms gives us the possibility to prove all the modifications of the proof system we could make for e.g. the deep-embedded implementation

But there are also drawbacks:

1. The system is not easy to extend : there are many flavours of DC out there (e.g. [6, 7, 11]), and having a static definition for the shape of formulas makes a slight modification having repercussions all over the system, grammar redefinitions and side-condition functions.
   Contrary to the shallow-embedded implementation, extending DC here requires adding the new connectors in the inductive definition above, adding axioms and

modifying interpretation functions possibly all over the library, making this deep-embedded implementation a much less modular one than the shallow-embedded one.
Note that this remark would also be true for any implementation "from scratch" (see [9]).

2. Having to define all axioms and inference rules for well-known logical operators from the beginning can be a source of bugs. Indeed, in a shallow-embedded implementation, we can make the decision to trust the definition of already present connectors. Moreover building such an implementation is time-consuming.

5 Perspectives

What made us stop our work, besides other developments, in each one of the implementation is:

- The conflicts caused by the temporal logical connector $\frown$ and the special variable ℓ with the inference rules of the proof system, for the shallow-embedded implementation
- The time-consuming task of defining the whole proof system from the beginning, for the deep-embedded implementation

In [9], the former is solved by the modification of the axioms and rules causing those conflicts : the equality is redefined, the DC-specific inference rules are modified to take in account that the inference system is a sequent-style one. Moreover, those modifications are not proved with pen and paper, but are proved with an earlier implementation of DC with PVS [12]. In short, modifications for the implementation of a proof system in a proof tool are proved with another proof tool.

This is where the deep-embedded implementation can help us : we can use it to prove modifications of the proof system that would solve the problems of the shallow-embedded implementation.

An interesting track to solve those problems, is to consider ℓ no more as a variable (because of its peculiar properties), but rather as predicates over values of the chosen numeric domain (real numbers usually) with the adequate axioms.

Example 19. E.g., with $interval\_le$ stating that the current time interval is lower than or equal to some value, we define the axioms relating this predicate with "normal" order relations:
$interval\_le(x) \wedge x \leq y \Rightarrow interval\_le(y)$

6 Conclusion

As explained in section 1, this work is an effort to have a proof tool for both normal logic and DC at disposal.

We built two implementations, a shallow-embedded and a deep-embedded one, having in mind different uses for them : the former would be used as a proof tool allowing

one to reason on formulas and specifications made with DC, and the latter would allow one to reason on the DC proof system itself, e.g. to prove the equivalence of two interpretations of DC, or find decidability results.

The shallow-embedded implementation has shown us problems already faced in [9] with solutions that are not easy to apply with Coq, and the deep-embedded implementation, whereas long to define, can help us modify the proof system so as to solve these problems. So the same proof tool is used both to implement a proof system and to prove properties of this proof system.

7 Thanks

I would like to thank Vincent Poirriez for his enlightening remarks, Georges Mariano, the team of the Coq project for making such a powerful proof tool.

References

1. Lano, K.: Specifying reactive systems in B AMN. LNCS **1212** (1997) 242–275
2. Treharne, H., Schneider, S.: Capturing timing requirements formally in AMN. Technical Report CSD-TR-99-06, Royal Holloway, Department of computer science, Egham, Surrey TW20 0EX, England (1999)
3. Hammad, A., Julliand, J., Mountassir, H., Okalas Mossami, D.: Expression en B et raffinement des systèmes réactifs temps réel. In: AFADL'2003. (2003) 211–225
4. Zhou, C., Hoare, C., Ravn, A.: A calculus of durations. In: Information Processing Letters. Volume 10(5). (1991) 269–276
5. Dutertre, B.: On first order interval temporal logic. Technical Report CSD-TR-94-3, University of London, Department of computer science, Egham, Surrey TW20 0EX, England (1995)
6. Zhou, C., Wang, J., Ravn, A.: A duration calculus with infinite intervals. In Reichel, H., ed.: Fundamentals of Computing Theory. Volume 965 of LNCS. Springer-Verlag, Lübeck, Germany (1995) 16–41
7. Zhou, C., Guelev, D., Naijun, Z.: A higher-order duration calculus. Technical Report 167, UNU/IIST, P.O.Box 3058, Macau (1999)
8. Hansen, M., Zhou, C.: Duration calculus, logical foundations. In: Formal Aspects of Computing. Volume 9. (1997) 283–330
9. Heilmann, S.T.: Proof Support for Duration Calculus. Phd-thesis, Department of Information Technology, Technical University of Denmark (1999)
10. : Coq (1989-2003) http://coq.inria.fr.
11. Guelev, D., Hung, D.: Completeness and decidability of a fragment of duration calculus with iteration. Technical Report 163, UNU/IIST, P.O. Box 3058, Macau (1999)
12. Skakkebæk, J.U.: A Verification Assistant for a Real-Time Logic. Phd-thesis, Department of Computer Science, Technical University of Denmark (1994) Also available as Technical Report ID-TR: 1994-150.

The Termination Prover AProVE

Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, Stephan Falke

LuFG Informatik II, RWTH Aachen, Ahornstr. 55, 52074 Aachen, Germany
{giesl—thiemann}@informatik.rwth-aachen.de
{nowonder—spf}@i2.informatik.rwth-aachen.de

Abstract. We describe the system AProVE, an automated prover to verify (innermost) termination of term rewrite systems (TRSs), functional programs, and logic programs. For this system, we have developed and implemented efficient algorithms based on classical simplification orders (recursive path orders with status, Knuth-Bendix orders, polynomial orders), dependency pairs, and the size-change principle. In AProVE, termination proofs can be performed with a user-friendly graphical interface and the system is currently among the most powerful termination provers available.

1 Introduction

The system AProVE (Automated Program Verification Environment) can be used for automated termination and innermost termination proofs of (conditional) term rewrite systems, as well as for termination proofs of functional and logic programs. AProVE offers a variety of techniques for automated termination proofs: First of all, it provides efficient implementations of classical simplification orders to prove termination "directly" (such as *recursive path orders* possibly with status [7, 19], *Knuth-Bendix orders* [20], and *polynomial orders* [22]), cf. Sect. 2. To increase the power of automated termination proofs, we implemented the *dependency pair* technique [2, 15] in AProVE which allows the application of classical simplification orders to many examples where automated termination analysis would fail otherwise (Sect. 3). In contrast to most other implementations of dependency pairs, we integrated refinements such as *narrowing*, *rewriting*, and *instantiation* of dependency pairs [14] and we improved the dependency pair technique further (e.g., by integrating the generation of *argument filterings* with the computation of *usable rules*) [16] to increase both the efficiency and the power of the approach. Due to these extensions and improvements, AProVE succeeds on many examples where all other automated termination provers fail. Thus, the principles used in the implementation of AProVE are also very helpful for other tools based on dependency pairs ([1], CiME [6], TTT [18]) and we conjecture that they can also be used in other recent approaches for termination of TRSs [5, 11] which have several aspects in common with dependency pairs. Apart from direct termination proofs and dependency pairs, as a third termination technique, AProVE offers the *size-change principle* [9, 23] and it is also possible to combine this principle with dependency pairs [27] (Sect. 4). The tool

is written in Java and proofs can be performed both in a fully automated or in an interactive mode via a graphical user interface (Sect. 6), as shown in Fig. 1.

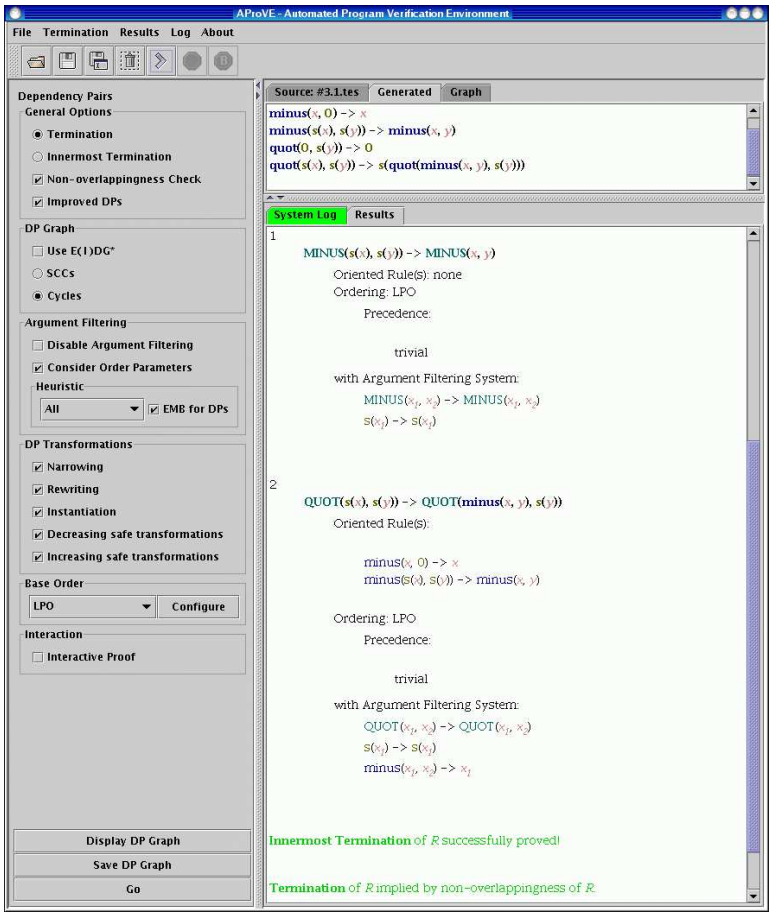

Fig. 1. Screenshot of the AProVE system

2 Direct Termination Proofs

In this section we describe the base orders available in AProVE which can be used for direct termination proofs, but also for proofs with preprocessing techniques like dependency pairs or the size-change principle.

In a direct termination proof of a TRS, the system tries to find a reduction order such that all rules of the TRS are decreasing. Currently, the following *path orders* are implemented in AProVE: the embedding order (`EMB`), the lexicographic path order (`LPO`, [19]), the `LPO` with status which compares subterms

lexicographically according to arbitrary permutations (`LPOS`), the recursive path order which compares subterms as multisets (`RPO`, [7]), and the `RPO` with status which is a combination of `LPOS` and `RPO` (`RPOS`).

Path orders may be parameterized by a precedence on function symbols and a status which determines how the arguments of function symbols are compared. To explore the search space for these parameters, the system leaves the precedence and the status as unspecified (or "*minimal*") as possible. The user can decide whether to perform a depth-first or a breadth-first search (where in the latter case, *all* possibilities for a minimal precedence and status are computed which satisfy the current constraints). Moreover, the user can configure the path orders by deciding whether different function symbols may be equivalent according to the precedence used in the path order ("`non-strict precedence`"). It is also possible to restrict potential equivalences to certain pairs of function symbols. When attempting termination proofs with path orders in AProVE, the precedence found by the system is displayed as a graph (in case of success) and in case of failure in the breadth-first search, the system indicates the problematic constraint.

In addition to the above path orders, AProVE offers *Knuth-Bendix orders* (`KBO`, [20]) using the polynomial-time algorithm of [21]. In this algorithm, one has to compute the degenerate subsystem of a system of homogeneous linear inequalities. This is done using the technique of [10].

The last class of orders available in AProVE are *polynomial orders* [22] where every function symbol is associated with a polynomial with natural coefficients. Here, the user can specify three parameters: the degree of the polynomials, the range of the coefficients, and the search algorithm that is used to find suitable coefficients in the given range. Apart from these global options, the user can also provide individual polynomials for some function symbols manually. To prove termination afterwards, AProVE generates a set of polynomial inequalities which state that left-hand sides of rules should be greater than the corresponding right-hand sides. Using the method of partial derivation [13, 22], these inequalities are transformed into a set of inequalities only containing coefficients, but no variables anymore. At that point, a search algorithm has to determine suitable coefficients that satisfy the resulting inequalities. The user can choose between four different search algorithms: we offer brute force search, greedy search, a genetic algorithm, and a constraint-based method based on interval arithmetic.

3 Termination Proofs With Dependency Pairs

The *dependency pair* approach [2, 14–16] increases the power of automated termination analysis significantly, since it permits the application of simplification orders for non-simply terminating TRSs. Instead of comparing left- and right-hand sides of rules, in this approach one compares the left-hand sides with those subterms of right-hand sides that correspond to recursive calls. More precisely, the root symbols of the left-hand sides are called *defined symbols* and for each defined symbol f we introduce a fresh *tuple symbol* F. For each rule

$f(s_1, \ldots, s_n) \to r$ and each subterm $g(t_1, \ldots, t_m)$ of r with defined root symbol g, we build a *dependency pair* $F(s_1, \ldots, s_n) \to G(t_1, \ldots, t_m)$. In order to prove termination one now has to find a weakly monotonic order $\succ$ such that $s \succ t$ for all dependency pairs $s \to t$ and $l \succsim r$ for all rules $l \to r$. When proving innermost termination, $l \succsim r$ is only required for the usable rules of the defined symbols occurring in the dependency pairs. Here, the *usable rules* for a symbol f are the f-rules together with the usable rules for all defined symbols occurring in right-hand sides of f-rules. In AProVE, the user can select whether to use the dependency pair approach for termination or for innermost termination proofs. The system can also check whether a TRS is non-overlapping (then innermost termination already implies termination).

To search for a suitable order $\succ$, the user can select any of the base orders from Sect. 2. However, while most of these orders are *strongly* monotonic, the dependency pair approach only requires *weak* monotonicity. (For polynomial orders, a weakly monotonic variant can be obtained immediately by permitting the coefficient 0 in polynomials. But lexicographic or recursive path orders as well as Knuth-Bendix orders are always strongly monotonic.) For that reason, before searching for a suitable order, some of the arguments of the function symbols in the constraints can be eliminated using an *argument filtering* π [2]. For example, a binary function symbol f can be turned into a unary symbol by eliminating the first argument of f. Then π replaces all terms $f(t_1, t_2)$ in the constraints by $f(t_2)$. Thus, we can obtain a weakly monotonic order $\succ_\pi$ out of a strongly monotonic order $\succ$ and an argument filtering π by defining $s \succ_\pi t$ iff $\pi(s) \succ \pi(t)$. For innermost termination proofs, we developed an improvement such that the argument filtering is also used for reducing the set of constraints [16, Thm. 11].

Since there are exponentially many argument filterings, a crucial problem for any implementation of dependency pairs is to explore this search space efficiently. In AProVE, we use a depth-first algorithm [16] to determine suitable argument filterings by treating the constraints one after another. We start with the set of argument filterings possibly satisfying the first constraint. Here we use the idea of [17] to keep argument filterings as "undefined" as possible. Then this set is reduced further to those filterings which can possibly satisfy the second constraint as well. This procedure is repeated until all constraints have been investigated. By inspecting the constraints in a suitable order, already after the first constraint the set of possible argument filterings is rather small and in this way, one only inspects a small subset of all potential argument filterings. Moreover, one can also combine the search for the argument filtering with the search for the base order by choosing the option "`consider order parameters`". If the user selects this option, then the system additionally stores for each possible argument filtering a minimal set of precedences and stati as described in Sect. 2. This option is only available for path orders.

(Innermost) termination proofs with dependency pairs can be performed in a modular way by constructing an estimated (innermost) *dependency graph* and by regarding its cycles separately [2, 15]. For each cycle, only one dependency pair must be strictly decreasing, whereas all others only have to be weakly decreasing.

As shown in [17], one should not compute all cycles, but only maximal cycles (*strongly connected components (SCCs)*). The reason is that the the chosen argument filtering and base order may make several dependency pairs in an SCC strictly decreasing. In that case, subcycles of the SCC containing such a strictly decreasing dependency pair do not have to be considered anymore. So after solving the constraints for the initial SCCs, all strictly decreasing dependency pairs are removed and one now builds SCCs from the remaining dependency pairs, etc. To inspect estimated (innermost) dependency graphs, they can be displayed in a special "`Graph`"-window. In order to benefit from all refinements on modularity of dependency pairs, we developed and implemented a technique which permits the combination of recent results on modularity of C_ε-terminating TRSs [28] with arbitrary estimations of dependency graphs, cf. [16].

To increase the power of the dependency pair technique, in [2, 14, 16] three different transformation techniques were suggested which transform a dependency pair into several new pairs: *narrowing*, *rewriting*, and *instantiation*. These transformations are often crucial for the success of the proof and in general, their application can never "harm": if the termination proof succeeds without transformations, then it also succeeds when performing transformations, but not vice versa. However, the problem is when to use these transformations, since in general, they may be applicable infinitely often. AProVE automatically performs these transformations in "safe" cases where their application is guaranteed to terminate. There are two kinds of "safe" cases: despite the fact that applying transformations can never prevent a termination proof that would have been possible without transformations, these transformations may increase runtime, since they can produce a large number of similar constraints. However, those transformations which delete dependency pairs or cycles do not have a negative impact on the efficiency and are called *decreasing*. The remaining transformations are called *increasing*. The system offers two switches where the user can enable or disable both kinds of transformation. If turned on, the decreasing transformations are applied before trying to solve the constraints for a cycle. Increasing transformations are only used a limited number of times whenever a proof attempt fails, and then the proof is re-attempted again.

In addition to the fully automated mode, (innermost) termination proofs with dependency pairs can also be performed in an interactive mode. Here, the user can specify which narrowing, rewriting, and instantiation steps should be performed and for any cycle or SCC, the user can determine (parts of) the argument filtering, the base order, and the dependency pair which should be strictly decreasing. Moreover, one can immediately see the constraints resulting from such selections, such that interactive termination proofs are supported in a very comfortable way. This mode is intended for the development of new heuristics as well as for the machine-assisted proof of particularly challenging examples.

4 Termination Proofs with the Size-Change Principle

A new *size-change principle* for termination of functional programs was presented in [23] and this principle was extended to TRSs in [27]. A similar prin-

ciple is also known for termination proofs of logic programs [9]. The main idea is to build a corresponding *size-change graph* from each dependency pair $F(s_1, ..., s_n) \to G(t_1, ..., t_m)$. This graph is bipartite where the nodes $1_F, \ldots, n_F$ on the left-hand side correspond to the arguments of the F-term and the nodes $1_G, \ldots, m_G$ on the right-hand side correspond to the arguments of the G-term. We draw an edge $i_F \stackrel{\succ}{\to} j_G$ iff $s_i \succ t_j$. Otherwise, there is an edge $i \stackrel{\succ}{\leadsto} j$ if at least $s_i \succsim t_j$ holds. Furthermore, we add a label $F \to G$ to the whole size-change graph.

We can concatenate two or more size-change graphs to a multigraph if the labels of each two consecutive size-change graphs are compatible (where the label $F \to G$ is *compatible* with $G \to H$ for all tuple symbols F and H). Each path from a left node of the first size-change graph to a right node of the last size-change graph leads to an edge in the multigraph. If there is at least one $\stackrel{\succ}{\to}$- edge on the path, then the resulting edge in the multigraph is labelled with $\succ$, otherwise it is labelled with $\succsim$. We call a multigraph G *maximal* iff the concatenation of G with G results in G again. The main theorem of the size-change principle states that termination can be concluded if each maximal multigraph contains an edge $i \stackrel{\succ}{\to} i$.

In AProVE, the technique of [27, Thm. 11] for size-change termination of TRSs is implemented, where we use the embedding order as underlying base order.[1] AProVE displays all size-change graphs as well as all maximal multigraphs (in case of success) or one critical maximal multigraph without a decreasing edge $i \stackrel{\succ}{\to} i$ (in case of failure).

AProVE also contains the new approach of [27] which combines the size-change principle with dependency pairs in order to prove (innermost) termination. This combined approach has the advantage that it often succeeds with much simpler argument filterings and base orders than the pure dependency pair approach. For each SCC $\mathcal{P}$ of the estimated (innermost) dependency graph, let $\mathcal{C}_\mathcal{P}$ be the constructors in $\mathcal{P}$ and let $\mathcal{D}_\mathcal{P}$ be a subset of the defined symbols in $\mathcal{P}$. Then the system builds the size-change graphs and the maximal multigraphs resulting from $\mathcal{P}$ using an argument filtering and the embedding order on $\mathcal{C}_\mathcal{P} \cup \mathcal{D}_\mathcal{P}$. Again, all these multigraphs must have an edge $i \stackrel{\succ}{\to} i$ and in case of success, the system displays them all. Next, the argument filtering must be extended such that all rules are weakly decreasing w.r.t. the selected base order. When proving innermost termination, instead it suffices if just the usable rules for the symbols $\mathcal{D}_\mathcal{P}$ are weakly decreasing. For reasons of efficiency, the user can impose a limit on the maximal size of $\mathcal{D}_\mathcal{P}$ and one can restrict the number of symbols in dependency pairs which may be argument-filtered.

In case of failure for some SCC, the dependency pairs are transformed by narrowing, rewriting, or instantiation and the proof attempt is re-started. If the user has selected the "`hybrid`" algorithm, then the pure dependency pair method

[1] As shown in [27], only very restricted base orders are sound in connection with the size-change principle. In addition to the results in [27], the full embedding order may be used, where $f(\ldots, x_i, \ldots) \succ x_i$ also holds for defined function symbols f.

is tried as soon as the limits for the transformations are reached. In this way, the combined dependency pair and size-change method can be used as a very fast technique which is checked first for every SCC. Only if this method fails, the ordinary dependency pair approach is used on this SCC.

5 Design of AProVE

The techniques of the previous two sections share one common property: they can be seen as *SCC processors* which transform one SCC into a set of new SCCs. The dependency pair technique generates a set of constraints for each SCC. If the constraints can be solved, then the SCC can be disregarded, while some new SCCs of subgraphs may have to be examined. The transformations "narrowing", "rewriting", and "instantiation" can also produce a set of new SCCs out of a given one. Finally, the combination of dependency pairs with the size-change principle processes the SCCs of the estimated (innermost) dependency graph one by one, too. Hence, all these termination proving algorithms work according to the following structure.

1. Compute the initial SCCs of the (estimated) innermost dependency graph.
2. While there are SCCs left and there is no failure:
   (a) Remove one SCC $\mathcal{P}$ from the set of SCCs.
   (b) Compute a new set of SCCs by processing $\mathcal{P}$ with an SCC processor.
   (c) Add the new set of SCCs to the remaining SCCs.

Thus, the termination proving techniques above are implemented in AProVE as modules which process one SCC and return a set of SCCs. Due to this modular structure, procedures for termination proofs which combine different termination techniques can easily be implemented within AProVE. One just has to configure the system by determining which SCC processors with which parameters should be used in Step 2(b). To obtain an efficient and powerful proof procedure, one should first try to use fast SCC processors which benefit from successful heuristics. In this way, SCCs that are easy to handle can be treated efficiently. Only for SCCs where these fast SCC processors fail, one should use slower but more powerful SCC processors afterwards. Examples for such termination procedures are the hybrid algorithm described in the last section or the "`meta combination`" algorithm of [16] that combines five different SCC processors. This algorithm is particularly useful if one does not want to get involved with the details of termination proving, but one wants to use AProVE in a "black box"-mode. Similar to the modular design of SCC processing, we have implemented the base orders and suitable heuristics in a modular way such that they can be combined freely. Up to now, to realize arbitrary combinations of several different SCC processors, base orders, and heuristics, one has to modify the code of the system, but we are working on a configuration language such that the user will be able to configure new termination proof procedures.

6 Running AProVE

The system AProVE accepts four different input languages: logic and (first-order) functional programs, conditional and unconditional TRSs. Functional programs are translated into conditional TRSs regarding the special semantics of the predefined conditional "if". Logic programs are translated into conditional TRSs, too, using the method of [4, 12]. Conditional TRSs are transformed further into unconditional TRSs according to the techniques of [14, 24] to prove their (innermost) quasi-decreasingness. For logic programs, these transformations correspond to the approach of the termination prover TALP [25].

When performing termination proofs, a "`system log`" can be inspected to examine all (possibly failed) proof attempts. The results of the termination proof are displayed in `html`-format and can be stored in `html`- or LaTeX-format. Any termination proof attempt may of course be interrupted by a stop-button. Instead of running the system on only one term rewrite system or program, it is also possible to run it on collections and directories of examples in a `batch mode`. In this case, apart from the information on the termination proofs of the separate examples, the "`result`" also contains statistics on the success and the runtime for the examples in the collection.

While the user can select between several different heuristics for performing termination proofs, we also provided the meta combination algorithm from the previous section which applies selected heuristics in a suitable way [16]. For example, when running the meta combination algorithm on the example collections of [3, 8, 26] (108 TRSs for termination, 151 TRSs for innermost termination), AProVE succeeded on 96.6 % of the innermost termination examples (including all of [3]) and on 93.5 % of the examples for termination. The automated proof for the whole collection took 80 seconds for innermost termination and 27 seconds for termination. These results also illustrate the power and efficiency of AProVE. For more details and to download the system, the reader is referred to the AProVE web-site `http://www-i2.informatik.rwth-aachen.de/AProVE`.

References

1. T. Arts. System description: The dependency pair method. In *Proc. 11th RTA*, LNCS 1833, pages 261–264, 2000.
2. T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236:133–178, 2000.
3. T. Arts and J. Giesl. A collection of examples for termination of term rewriting using dependency pairs. Technical Report AIB-2001-09, RWTH Aachen, Germany, 2001. Available from `http://aib.informatik.rwth-aachen.de`.
4. T. Arts and H. Zantema. Termination of logic programs using semantic unification. In *Proc. 5th LOPSTR*, LNCS 1048, pages 219–233, 1996.
5. C. Borralleras, M. Ferreira, and A. Rubio. Complete monotonic semantic path orderings. In *Proc. 17th CADE*, LNAI 1831, pages 346–364, 2000.
6. E. Contejean, C. Marché, B. Monate, and X. Urbain. CiME version 2, 2000. Available from `http://cime.lri.fr`.

7. N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3:69–116, 1987.
8. N. Dershowitz. 33 examples of termination. In *Proc. French Spring School of Theoretical Computer Science*, LNCS 909, pages 16–26, 1995.
9. N. Dershowitz, N. Lindenstrauss, Y. Sagiv, and A. Serebrenik. A general framework for automatic termination analysis of logic programs. *Applicable Algebra in Engineering, Communication and Computing*, 12(1,2):117–156, 2001.
10. J. Dick, J. Kalmus, and U. Martin. Automating the Knuth-Bendix ordering. *Acta Informatica*, 28:95–119, 1990.
11. O. Fissore, I. Gnaedig, and H. Kirchner. Cariboo: An induction based proof tool for termination with strategies. In *Proc. 4th PPDP*, pages 62–73. ACM, 2002.
12. H. Ganzinger and U. Waldmann. Termination proofs of well-moded logic programs via conditional rewrite systems. In *Proc. 3rd CTRS*, LNCS 656, pages 113–127, 1993.
13. J. Giesl. Generating polynomial orderings for termination proofs. In *Proc. 6th RTA*, LNCS 914, pages 426–431, 1995.
14. J. Giesl and T. Arts. Verification of Erlang processes by dependency pairs. *Applicable Algebra in Engineering, Communication and Computing*, 12(1,2):39–72, 2001.
15. J. Giesl, T. Arts, and E. Ohlebusch. Modular termination proofs for rewriting using dependency pairs. *Journal of Symbolic Computation*, 34(1):21–58, 2002.
16. J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Improving dependency pairs. In *Proc. 10th LPAR*, LNAI 2850, pages 165–179, 2003.
17. N. Hirokawa and A. Middeldorp. Automating the dependency pair method. In *Proc. 19th CADE*, LNAI 2741, 2003.
18. N. Hirokawa and A. Middeldorp. Tsukuba termination tool. In *Proc. 14th RTA*, LNCS 2706, pages 311–320, 2003.
19. S. Kamin and J. J. Lévy. Two generalizations of the recursive path ordering. Unpublished Manuscript, University of Illinois, IL, USA, 1980.
20. D. Knuth and P. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon, 1970.
21. K. Korovin and A. Voronkov. Verifying orientability of rewrite rules using the Knuth-Bendix order. In *Proc. 10th RTA*, LNCS 2051, pages 137–153, 2001.
22. D. Lankford. On proving term rewriting systems are Noetherian. Technical Report MTP-3, Louisiana Technical University, Ruston, LA, USA, 1979.
23. C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The size-change principle for program termination. In *Proc. POPL '01*, pages 81–92, 2001.
24. E. Ohlebusch. Termination of logic programs: Transformational approaches revisited. *Applicable Algebra in Engineering, Communication and Computing*, 12(1,2):73–116, 2001.
25. E. Ohlebusch, C. Claves, and C. Marché. TALP: A tool for the termination analysis of logic programs. In *Proc. 11th RTA*, LNCS 1833, pages 270–273, 2000.
26. J. Steinbach. Automatic termination proofs with transformation orderings. In *Proc. 6th RTA*, LNCS 914, pages 11–25, 1995. Full version appeared as Technical Report SR-92-23, Universität Kaiserslautern, Germany.
27. R. Thiemann and J. Giesl. Size-change termination for term rewriting. In *Proc. 14th RTA*, volume LNCS 2706, pages 264–278, 2003.
28. X. Urbain. Automated incremental termination proofs for hierarchically defined term rewriting systems. In *Proc. IJCAR 2001*, LNAI 2083, pages 485–498, 2001.

On the Implementation of a Rule-Based Programming System and Some of its Applications

Mircea Marin[1] and Temur Kutsia[2]

[1] Johann Radon Institute for Computational and Applied Mathematics
Austrian Academy of Sciences
A-4040 Linz, Austria
mircea.marin@oeaw.ac.at
[2] Research Institute for Symbolic Computation
Johannes Kepler University
A-4232 Hagenberg, Austria
tkutsia@risc.uni-linz.ac.at

Abstract. We describe a rule-based programming system where rules specify nondeterministic computations. The system is called FunLog and has constructs for defining elementary rules, and to build up complex rules from simpler ones via operations akin to the standard operations from abstract rewriting. The system has been implemented in Mathematica and is, in particular, useful to program procedures which can be encoded as sequences of rule applications which follow a certain reduction strategy. In particular, the procedures for unification with sequence variables in free, flat, and restricted flat theories can be specified via a set of inference rules which should be applied in accordance with a certain strategy. We illustrate how these unification procedures can be expressed in our framework.

1 Introduction

In this paper we describe a rule-based programming language and illustrate its usefulness for implementing unification procedures with sequence variables in free, flat and restricted flat theories. We have designed and implemented a rule-based system called FunLog, which provides programming constructs to define elementary rules and to build up complex rules from simpler ones. Our programming constructs include primitives to compose rules, to group them into a specification of a nondeterministic rule, to compute with their transitive closure, and to define various evaluation strategies.

Such a programming style is very suitable for complex computations steps which can be expressed as sequences of computation steps with the following characteristics:

1. Each computation step is driven by the application of a rule chosen (nondeterministically) from a finite set of alternative rules.

2. The sequence of steps must match a certain specification. In FunLog, such specifications can be built up via a number of operators which are similar to the operators of abstract term rewriting.

We used Funlog to implement unification procedures in free, flat, and restricted flat theories with sequence variables and flexible arity symbols. It was shown in [8, 9] that these procedures can be specified via a set of inference rules which should be applied in accordance with a certain strategy. Therefore, these procedures are a good example to be programmed in FunLog.

The rest of this paper is structured as follows. In Section 2 we give a brief description of the rule-based programming style supported by FunLog and describe the programming constructs of our system. Section 3 explains some details about the Mathematica implementation of FunLog. In Section 4 we describe our FunLog implementation of unification procedures with sequence variables. Section 5 concludes.

2 Programming with FunLog

FunLog is a rule-based system where

rule = specification of a partially defined, possibly nondeterministic computation.

This paradigm makes our notion of rule very similar to the notion of strategy as defined in the rule-based system Elan [4, 7]. There are, however, some notable exceptions:

1. all rules can be nondeterministic. The nondeterministic application of basic rules stems from the fact that we allow patterns which can match in more than one way with a given expression.
2. the application of rules is not driven by a built-in leftmost-innermost rewriting strategy. Instead, rules are always applied at the root position of a term. Rewriting behaviors can be attained by associating a new rule to a given rule, and imposing a certain strategy to look up for the subterm to be rewritten.

We believe that these features make our rule-based system more flexible, mainly because we can control the positions where rules should be applied.

In FunLog, each rule is characterized by a name and a code which describes a partially defined, possibly nondeterministic, computation. Formally, a rule is an expression

$$lbl :: patt \rightarrow rhs \tag{1}$$

where *lbl* is the rule name, *patt* is a pattern expression and *rhs* specifies the computation of a result from the bindings of the variables which occur in *patt*. The expression $patt :> rhs$ is called the *code* of the rule *lbl*.

The main usage of rules is to act with them them on various expressions. The attempt to apply a rule of type (1) on an expression *expr* proceeds as follows:

1. `M` := enumeration of all matchers between *expr* and *patt*.
2. If `M` $= \emptyset$ then **fail** else goto 3.
3. θ := `first`(`M`), `M` := `rest`(M), `V` := enumeration of all values of $\theta(rhs)$.
4. If `V` $= \emptyset$ then goto 3 else **return** `first`(`V`).

We write $expr \not\rightarrow_{lbl}$ if the application of rule *lbl* to *expr* fails, and $expr \rightarrow_{lbl}$ if it succeeds.

There are two sources of non-determinism in FunLog: non-unique matches and non-unique ways to evaluate a partially defined computation. Clearly, the result of an application $expr \rightarrow_{lbl}$ depends on the enumerations of matchers (`M`) and values (`V`) which are built into a particular implementation. These enumeration strategies are relevant to the programmer and are described in the specification of the operational semantics of our implementation.

In the sequel we write $expr_1 \rightarrow_{lbl} expr_2$ if we can identify two enumerations, for `M` and `V`, which render the result $expr_2$ for the application $expr_1 \rightarrow_{lbl}$.

Rules can be combined with various combinators into more complex rules. The implementation of these combinators is compositional, i.e., the meaning of each combination of rules can be defined in terms of the meanings of the component rules.

2.1 Main Combinators

A rule with name *lbl* is applied to an expression $expr_1$ via the call

$$\texttt{ApplyRule}[expr_1, lbl] \tag{2}$$

which behaves as follows:

- If $expr_1 \not\rightarrow_{lbl}$ then **return** $expr_1$
- If $expr_1 \rightarrow_{lbl}$ then **return** the first $expr_2$ for which $expr_1 \rightarrow_{lbl} expr_2$.

The call

$$\texttt{ApplyRuleList}[expr_1, lbl] \tag{3}$$

returns the list of values $\{expr_2 \mid expr_1 \rightarrow_{lbl} expr_2\}$.

FunLog provides a number of useful constructs to build up rules. These constructs are described in the remainder of this section.

Basic rules. A basic rule is a rule named by a string, whose code is given explicitly as a Mathematica transformation rule. A basic rule $lbl :: patt :> rhs$ is declared by

$$\texttt{DeclareRule}[patt :> rhs, lbl] \tag{4}$$

We recall that $expr_1 \rightarrow_{lbl} expr_2$ iff there is a matcher θ between $expr_1$ and *patt* for which $\theta(rhs)$ evaluates to $expr_2$.

The enumeration strategy of basic rules depends only on the enumeration strategy of matches.

Example 1. The rule "split" introduced by the declaration

$$\texttt{DeclareRule}[\{x\_\_\_, y\_\_\_\}/; (\texttt{Length}[\{x\}] > \texttt{Length}[\{y\}]) :> \{x\}, \text{“split”}]$$

takes as input a list L of elements and yields a prefix sublist of L whose length is larger than half the length of L. The outcome of the call

$$\begin{array}{l}\texttt{ApplyRule}[\{a, b, c, d\}, \text{“split”}]\\ \{a, b, c\}\end{array}$$

yields the instance $\{a, b, c\} = \theta(\{x\})$ corresponding to the matcher $\theta = \{x \mapsto \ulcorner a, b, c \urcorner, y \mapsto \ulcorner d \urcorner\}$. Note that this is the first matcher found by the enumeration strategy of the MATHEMATICA interpreter, for which $\theta(\texttt{Length}[\{x\}] > \texttt{Length}[\{y\}])$ holds. □

Choice. $lbl_1 \mid \ldots \mid lbl_n$ denotes a rule whose applicative behavior is given by

$$expr_1 \rightarrow_{lbl_1|\ldots|lbl_n} expr_2 \text{ iff } expr_1 \rightarrow_{lbl_i} expr_2 \text{ for some } i \in \{1, \ldots, n\}. \quad (5)$$

The enumeration of the steps $expr_1 \rightarrow_{lbl_1|\ldots|lbl_n}$ starts with the enumeration of the steps $expr_1 \rightarrow_{lbl_1}$, followed by the enumeration of the steps $expr_1 \rightarrow_{lbl_2}$, and so on up to the enumeration of the steps $expr_1 \rightarrow_{lbl_n}$.

Example 2. Consider the declarations

$$\begin{array}{l}\texttt{DeclareRule}[\{x\_\_\_m\_, y\_\_\_, n\_, z\_\_\_\} :> \texttt{False}/; (m > n), \text{“test”}];\\ \texttt{DeclareRule}[\texttt{\_List} :> \texttt{True}, \text{“else”}];\end{array}$$

Here, `_List` is a pattern variable which matches any list structure. Then

$$\texttt{ApplyRule}[L, \text{“test”} \mid \text{“else”}]$$

yields `True` iff L is a list with elements in ascending order. This behavior is witnessed by the calls:

$$\begin{array}{l}\texttt{ApplyRule}[\{1, 2, 4, 5, 3\}, \text{“test”} \mid \text{“else”}]\\ \texttt{False}\\ \texttt{ApplyRule}[\{1, 2, 3, 4, 5\}, \text{“test”} \mid \text{“else”}]\\ \texttt{True}\end{array}$$

The first call yields `False` because $\{1, 2, 4, 5, 3\} \rightarrow_{\text{“test”}} \texttt{False}$ with the matcher $\theta = \{x \mapsto \ulcorner 1, 2 \urcorner, m \mapsto 5, y \mapsto \ulcorner \urcorner, n \mapsto 4, z \mapsto \ulcorner 3 \urcorner\}$. The second call yields `True` because $\{1, 2, 3, 4, 5\} \not\rightarrow_{\text{“test”}}$ and $\{1, 2, 3, 4, 5\} \rightarrow_{\text{“else”}} \texttt{True}$. □

Composition. $lbl_1 \circ lbl_2$ denotes a rule whose applicative behavior is given by

$$expr_1 \rightarrow_{lbl_1 \circ lbl_2} expr_2 \text{ iff } expr_1 \rightarrow_{lbl_1} expr \rightarrow_{lbl_2} expr_2 \text{ for some } expr. \quad (6)$$

The enumeration of values $expr_2$ for which the relation $expr_1 \rightarrow_{lbl_1 \circ lbl_2} expr_2$ holds, proceeds by enumerating all steps $expr \rightarrow_{lbl_2} expr_2$ during an enumeration of the steps $expr_1 \rightarrow_{lbl_1} expr$.

Example 3 (Oriented graphs). The following rule declarations

$$\begin{array}{l}\texttt{DeclareRule}[x\_ :> x, \text{“Id”}];\\ \texttt{DeclareRule}[a :> b, \text{“r1”}]; \texttt{DeclareRule}[a :> c, \text{“r2”}];\\ \texttt{DeclareRule}[c :> b, \text{“r3”}]; \texttt{DeclareRule}[b :> d, \text{“r4”}];\\ \texttt{DeclareRule}[b :> e, \text{“r5”}]; \texttt{DeclareRule}[c :> f, \text{“r6”}];\end{array}$$

define the edges of an oriented graph with nodes $\{a, b, c, d, e, f\}$. Then

$$a \rightarrow_{\texttt{Repeat}[\text{“r1”}\mid\text{“r2”}\mid\text{“r3”}\mid\text{“r4”}\mid\text{“r5”}\mid\text{“r6”},\text{“Id”}]} v$$

iff there exists a (possibly empty) path from a to v. To find such a v we can call

$$\texttt{ApplyRule}[a, \texttt{Repeat}[\text{“r1”} \mid \text{“r2”} \mid \text{“r3”} \mid \text{“r4”} \mid \text{“r5”} \mid \text{“r6”}, \text{“Id”}]]$$

and FunLog will yield the value d corresponding to the derivation

$$a \rightarrow_{\text{“r1”}} b \rightarrow_{\text{“r4”}} d \rightarrow_{\text{“Id”}} d.$$

The call

$$\texttt{ApplyRuleList}[c, \texttt{Repeat}[\text{“r1”} \mid \text{“r2”} \mid \text{“r3”} \mid \text{“r4”} \mid \text{“r5”} \mid \text{“r6”}, \text{“Id”}]]$$

will yield the list $\{d, e, b\}$ of all nodes reachable from b. □

Reflexive-transitive closures. If $lbl \in \{\texttt{Repeat}[lbl_1, lbl_2], \texttt{Until}[lbl_2, lbl_1]\}$ then

$$expr_1 \rightarrow_{lbl} expr_2 \text{ iff } expr_1 \rightarrow^*_{lbl_1} expr \rightarrow_{lbl_2} expr_2 \text{ for some } expr \qquad (7)$$

where $\rightarrow^*_{lbl_1}$ denotes the reflexive-transitive closure of $\rightarrow_{lbl_1}$. These two constructs differ only with respect to the enumeration strategy of the possible reduction steps.

The enumeration of $expr_1 \rightarrow_{\texttt{Repeat}[lbl_1, lbl_2]} expr_2$ proceeds by unfolding the recursive definition:

$$\texttt{Repeat}[lbl_1, lbl_2] = lbl_1 \circ \texttt{Repeat}[lbl_1, lbl_2] \mid lbl_2,$$

whereas the enumeration of $expr_1 \rightarrow_{\texttt{Until}[lbl_1, lbl_2]} expr_2$ proceeds by unfolding the recursive definition:

$$\texttt{Until}[lbl_2, lbl_1] = lbl_2 \mid lbl_1 \circ \texttt{Until}[lbl_2, lbl_1].$$

Example 4 (Sorting). Consider the declarations of basic rules

$$\begin{array}{l}\texttt{DeclareRule}[x\_ :> x, \text{“Id”}];\\ \texttt{DeclareRule}[\{x\_\_m\_, y\_\_\_, n\_, z\_\_\_\}/; (m > n) :> \{x, n, y, m, z\}, \text{“perm”}];\end{array}$$

Then the enumeration strategy of `Repeat`[“perm”, “Id”] ensures that the application of rule `Repeat`[“perm”, “Id”] to any list of integers yields the sorted version if that list. For example

$$\texttt{ApplyRule}[\{3,1,2\},\texttt{Repeat}[\text{“perm”},\text{“Id”}]]$$

yields $\{1,2,3\}$ via the following sequence of transformation steps

$$\{3,1,2\} \rightarrow_{\text{“perm”}} \{1,3,2\} \rightarrow_{\text{“perm”}} \{1,2,3\} \rightarrow_{\text{“Id”}} \{1,2,3\}.$$

□

Rewrite rules. FunLog provides the following mechanism to define a rule that rewrites with respect to a given rule:

$$\texttt{RWRule}[lbl_1, lbl, \texttt{Traversal} \rightarrow \ldots, \texttt{Prohibit} \rightarrow \ldots] \tag{8}$$

This call declares a new rule named *lbl* such that $expr_1 \rightarrow_{lbl} expr_2$ iff there exists a position p in $expr_1$ such that $expr_1|_p \rightarrow_{lbl_1} expr$ and $expr_2 = expr_1[expr]_p$. Here, $expr_1|_p$ is the subexpression of *expr* at position p, and $expr_1[expr]_p$ is the result of replacing the subexpression at position p by *expr* in $expr_1$. The option `Traversal` defines the enumeration ordering of rewrite steps (see below), whereas the option `Prohibit` restricts the set of positions allowed for rewriting. If the option `Prohibit` $\rightarrow \{f_1, \ldots, f_n\}$ is given, then rewriting is prohibited at positions below occurrences of symbols $f \in \{f_1, \ldots, f_n\}$. By default, the value of `Prohibit` is $\{\}$, i.e., rewriting can be performed everywhere.

The enumeration strategy of rewrite steps can be controlled via the option `Traversal` which has the default value "LeftmostIn". If the option `Traversal` $\rightarrow$ “LeftmostOut” is given, then the rewriting steps are enumerated by traversing the rewriting positions in leftmost-outermost order. If `Traversal` $\rightarrow$ “LeftmostIn” is given, then the rewriting steps are enumerated by traversing the rewriting positions in leftmost-innermost order.

Example 5 (Pure λ-calculus). In λ-calculus, a value is an expression which has no β-redexes outside λ-abstractions. We adopt the following syntax for λ-terms:

`term` ::=		*terms* :
	x	variable
|	`app`[`term`$_1$, `term`$_2$]	application
|	$\lambda[x, \texttt{term}]$	abstraction

β-redexes are eliminated by applications of the β-conversion rule, which can be encoded in FunLog as follows:

$$\texttt{DeclareRule}[\texttt{app}[\lambda[x\_, t1\_], t2\_] :> \texttt{repl}[t_1, \{x, t2\}], \text{“}\beta\text{”}]$$

where $\texttt{repl}[t_1, \{x, t2\}]$ replaces all free occurrences of x in $t1$ by $t2$. A straightforward implementation of `repl` in MATHEMATICA is shown below[3]:

$$\begin{array}{l}
\texttt{repl}[\lambda[x_, t_], \{x_, _\}] := \lambda[x, t];\\
\texttt{repl}[x_, \{x_, t_\}] := t;\\
\texttt{repl}[\lambda[x_, t_], \sigma_] := \lambda[x, \texttt{repl}[t, \sigma]];\\
\texttt{repl}[\texttt{app}[t1_, t2_], \sigma_] := \texttt{app}[\texttt{repl}[t1, \sigma], \texttt{repl}[t2, \sigma]];\\
\texttt{repl}[t_, _] := t;
\end{array}$$

The computation of a value of a λ-term proceeds by repeated reductions of the redexes which are not inside abstractions. In FUNLOG, the reduction of such a redex coincides with an application of the rewrite rule "β-elim" defined by

$$\texttt{RWRule}[\text{“}\beta\text{”}, \text{“}\beta\text{-elim”}, \texttt{Prohibit} \rightarrow \{\lambda\}]$$

The following calls illustrate the behavior of this rule:

$$\begin{array}{l}
t := \texttt{app}[z, \texttt{app}[\lambda[x, \texttt{app}[x, \lambda[y, \texttt{app}[x, y]]]], \lambda[z, z]]];\\
t1 := \texttt{ApplyRule}[t, \text{“}\beta\text{-elim”}]\\
\ \texttt{app}[z, \texttt{app}[\lambda[z, z], \lambda[y, \texttt{app}[\lambda[z, z], y]]]]\\
t2 := \texttt{ApplyRule}[t1, \text{“}\beta\text{-elim”}]\\
\ \texttt{app}[z, \lambda[y, \texttt{app}[\lambda[z, z], y]]]\\
t3 := \texttt{ApplyRule}[t2, \text{“}\beta\text{-elim”}]\\
\ \texttt{app}[z, \lambda[y, \texttt{app}[\lambda[z, z], y]]]
\end{array}$$

Thus, $t2$ is a value of t. To compute the value of t directly, we could call

$$\begin{array}{l}
\texttt{ApplyRule}[t, \texttt{Repeat}[\text{“}\beta\text{-elim”}, \text{“Id”}]]\\
\ \texttt{app}[z, \lambda[y, \texttt{app}[\lambda[z, z], y]]]
\end{array}$$

□

Normal forms. $\texttt{NF}[lbl]$ denotes a rule whose applicative behavior is given by

$$expr \rightarrow_{\texttt{NF}[lbl]} expr_2 \text{ iff } expr_1 \rightarrow_{lbl}^{*} expr_2 \text{ and } expr_2 \not\rightarrow_{lbl}. \tag{9}$$

The enumeration strategy of $\texttt{NF}[lbl]$ is obtained by unfolding the recursive definition

$$\texttt{NF}[lbl] = \texttt{NFQ}[lbl] \mid lbl \circ \texttt{NF}[lbl] \tag{10}$$

where $\texttt{NFQ}[lbl] :: x\_ :> x /; (x \not\rightarrow_{lbl})$.

Abstraction. The abstraction principle is provided in FUNLOG via the construct

$$\texttt{SetAlias}[expr, lbl] \tag{11}$$

[3] The MATHEMATICA definitions are tried top-down, and this guarantees a proper interpretation of the replacement operation.

where *lbl* is a fresh rule name (a string identifier) and *expr* is an expression built from names of rules already declared with the operators |, ∘, `Repeat` and `Until` described before. For example, the call

$$\texttt{SetAlias}[\texttt{Repeat}[\text{“perm”}, \text{“Id”}], \text{“sort”}]$$

declares a rule named “sort” whose applicative behavior coincides with that of the construct `Repeat`[“perm”, “Id”].

3 Notes on Implementation

We have implemented a MATHEMATICA package called FUNLOG which supports the programming style elaborated above. We decided to implement FUNLOG in MATHEMATICA because MATHEMATICA has very advanced pattern matching constructs for specifying transformation rules. Moreover, MATHEMATICA provides a powerful mechanism to control backtracking. More precisely, it allows to specify transformation rules of the form

$$patt :> \texttt{Block}[\{\texttt{result}, \ldots\}, \texttt{result}/; test\_with\_side\_effect]$$

which, when applied to an expression *expr* behave as follows:

1. If *patt* matches with *expr*, compute θ :=the first matcher and go to 2. Otherwise fail.
2. Evaluate the condition *test_with_side_effect*, in which we instantiate the pattern variables with the bindings of θ.
3. If the computation yields `True`, it also computes `result` as a side effect. This is possible because the `Block` construct makes the variable `result` visible inside the calls of *test_with_side_effect*.
4. If *test_with_side_effect* yields `False` then the interpreter of MATHEMATICA backtracks by computing θ := next matcher and goto 2. If no matchers are left, then fail.

We have employed this construct to implement the backtracking mechanism of FUNLOG. For instance, the code for $lbl_1 \circ lbl_2$ is computed as follows:

– assume $patt :> rhs$ is the code of lbl_1. Then the code of $lbl_1 \circ lbl_2$ will be of the form

$$patt :> \texttt{Block}[\{\texttt{result}, \ldots\}, \texttt{result}/; \texttt{CasesQ}[rhs, lbl_2]]$$

where `CasesQ`$[rhs, lbl_2]$ does the following:
- If $rhs \rightarrow_{lbl_2} expr_1$, then it binds `result` to $expr_1$ and yields `True`.
- If $rhs \not\rightarrow_{lbl_2}$ it yields `False`.

This decision step relies on the possibility to detect whether the code of rule lbl_2 accepts $expr_1$ as input. In our implementation, the code of lbl_2 is a MATHEMATICA transformation rule of the form $patt_2 :> rhs_2$. The piece of MATHEMATICA code

$$ok = \texttt{True};\ \texttt{result} = \texttt{Replace}[expr_1, \{patt_2 \rightarrow rhs_2, \_ \rightarrow (ok = \texttt{False})\}]$$

does the following:

- It sets the value of *ok* to `True`.
- It applies the rule $patt_2 \to rhs_2$ to $expr_1$ by assigning $expr_2$ to `result` if $expr_1 \to_{lbl_2} expr_2$.
- If $expr_1 \not\to_{lbl_2}$ then it applies the MATHEMATICA rule $\_ \to (ok = \texttt{False})$, which assigns `False` to *ok*.

In this way we can simultaneously check whether the code of lbl_2 is defined for $expr_1$, and assign $expr_2$ to `result` if $expr_1 \to_{lbl_2} expr_2$.

– variations of the same trick can be used for all the possible syntactic shapes of the code of lbl_1.

A similar trick can be used to implement the code for `Repeat`$[lbl_1, lbl_2]$.

In principle, our implementation of the combinators for rules is based on an agreed-upon standard on how to represent the code of non-basic rules, which enables to easily compute the code of the newly declared rule. We do not expose the details here because some of them require a deep understanding of the evaluation and backtracking principles of MATHEMATICA.

4 Applications – Implementing Unification Procedures with Sequence Variables

We have implemented a library of unification procedures for free, flat and restricted flat theories with sequence variables and flexible arity symbols [8] in FUNLOG. The common characteristic features of the procedures is that they are based on transformation rules, applied in a "don't know" non-deterministic manner. Each of the procedures is designed as a tree generation process.

A sequence variable is a variable that can be instantiated by an arbitrary finite sequence of terms. Below we use $\overline{x}, \overline{y}, \overline{z}$ and $\overline{w}$ to denote sequence variables, while x, y, z will denote individual variables. Sequence variables are used together with flexible arity function symbols. Terms and equalities are built in the usual way over individual and sequence variables, and fixed and flexible arity function symbols, with the following restriction: sequence variable can not be a direct argument of a fixed arity function, and sequence variable can not be a direct argument of equality. Substitutions map individual variables to single terms and sequence variables to finite, possible empty, sequences of terms. Application of a substitution is defined as usual. For example, applying the substitution $\{x \mapsto a, y \mapsto f(\overline{x}), \overline{x} \mapsto \ulcorner\urcorner, \overline{y} \mapsto \ulcorner a, f(\overline{x}), b\urcorner\}$ to the term $f(x, \overline{x}, g(y, y), \overline{y})$ gives $f(a, g(f(\overline{x}), f(\overline{x})), a, f(\overline{x}), b)$. The standard notions of unification theory [3] can be easily adapted for terms with sequence variables and flexible arity symbols.

4.1 Free Unification with Sequence Variables and Flexible Arity Symbols

In [9] a minimal complete unification procedure for a general free unification problem with sequence variables and flexible arity symbols was described. The procedure consists of five types of rules: projection, success, failure, elimination

and split. They are given in Appendix. All three types of free unification with sequence variables (elementary, with constants and general) are decidable, but infinitary. We do not implement the decision procedure (it requires Makanin algorithm [10] and the combination method [2]). Instead, we put a bound on the unification tree depth and perform a depth-first search with backtracking. Optionally, if the user sets the tree depth bound to infinity, FunLog performs iterative deepening with the predefined depth (by default it is set to 20, but the user can change it), and reports the solutions as they are found.

Nodes in the unification tree are pairs (u, σ), where u is a unification problem together with its context and σ is the substitution computed so far. The successful nodes are labelled only with substitutions.

For instance, the third and fourth elimination rules can be encoded in FunLog as follows:

```
DeclareRule [{{({f_[x_?SVarQ, s1___], f_[t_, s2___]} |
              {f_[t_, s2___], f_[x_?SVarQ, s1___]})/; Not[SVarQ[t]],
              ctx_}, σ_}/; FreeQ[t, x] :>
              {{{f[s1], f[s2]}, ctx}/.x → t,
              ComposeSubst[σ, {x → t}], "Elim-svar-nonsvar-1"];


DeclareRule [{{({f_[x_?SVarQ, s1___], f_[t_, s2___]} |
             {f_[t_, s2___], f_[x_?SVarQ, s1___]})/; Not[SVarQ[t]],
             ctx_}, σ_}/; FreeQ[t, x] :>
             {{{f[x, Apply[Sequence, {s1}/.x → seq[t, x]]],
             f[s2]/.x → seq[t, x]}, ctx/.x → seq[t, x]},
             ComposeSubst[σ, {x → seq[t, x]}], "Elim-svar-nonsvar-2"];
```

"Elim-svar-nonsvar-1" rule corresponds to the cases with the substitution σ_1 of the third and fourth elimination rules in Fig. 1, and "Elim-svar-nonsvar-2" corresponds to the cases with σ_2 of the same rules. In this manner, we can declare a finite set of FunLog rules that cover all the situations shown in Fig. 1. The non-success rules can be grouped together via the FunLog construct

$$\texttt{SetAlias}[lbl_1 | \cdots | lbl_n, \text{“Non-success”}]$$

where $lbl_1, \ldots, lbl_n$ are labels of all non-success rules. Similarly, the success rules can be grouped into a rule

$$\texttt{SetAlias}[lbl_1 | \cdots | lbl_m, \text{“Success”}]$$

where $lbl_1, \ldots, lbl_m$ are the names of all success rules encoded with FunLog.

We encode computation of one unifier into the following FunLog rule:

$$\texttt{SetAlias}[\text{“Projection”} \circ \texttt{Repeat}[\text{“Non-success”}, \text{“Success”}], \text{“Unify”}].$$

After that, the computation of a unifier of a free unification problem Γ is achieved by the call

$$\texttt{ApplyRule}[\Gamma, \text{“Unify”}]$$

whereas the list of all unifiers is produced by the call

$$\texttt{ApplyRuleList}[\Gamma, \text{“Unify”}].$$

Example 6. For the free unification problem $f(x, b, \overline{y}, f(\overline{x})) \simeq^?_\emptyset f(a, \overline{x}, f(b, \overline{y}))$ the procedure computes the solution $\{\{x \mapsto a, \overline{x} \mapsto \ulcorner b, \overline{x}\urcorner, \overline{y} \mapsto \overline{x}\}, \{x \mapsto a, \overline{x} \mapsto b, \overline{y} \mapsto \ulcorner\urcorner\}\}$. □

Problems like word equations [1] or associative unification with unit element [14] can be encoded as a particular case of free unification with sequence variables and flexible arity symbols. Similarly, associative unification [13] can be translated into a particular case of free unification with sequence variables when projection rules are omitted. Thus, as a side effect, our implementation also provides unification procedures for those problems.

4.2 Flat Unification with Sequence Variables and Flexible Arity Symbols

Flat theory with sequence variables is axiomatized by the equality $f(\overline{x}, f(\overline{y}), \overline{z}) \simeq f(\overline{x}, \overline{y}, \overline{z})$. This theory gives a precise characterization of evaluation behavior of flat functions as it is implemented in the MATHEMATICA system. It was the main motivation to study the flat theory, but then it turned out to have some interesting properties. Namely, it was shown that both matching and unification are infinitary but decidable, and the unification procedure was designed. It should be noted that flat pattern matching of MATHEMATICA implements a restricted, finitary case of matching in the flat theory with sequence variables and flexible arity symbols.

The FUNLOG implementation of a general flat unification with sequence variables and flexible arity symbols goes along the procedure described in [8]. It combines the rules specific for the flat theory with those specific for the free theory. The implementation does not contain the decision procedure and uses the depth-first search with bounded depth. Since the procedure does not enumerate directly the minimal complete set of unifiers, after answer generation a certain minimization effort is required. As a result, the answer returned by the procedure represents a minimal subset of the complete set of solutions (and not a subset of minimal complete set of solutions).

Example 7. Let $f(\overline{x}) \simeq^?_F f(a)$ be a flat unification problem. It has infinitely many solutions. Our implementation computes the subset

$\{\{\overline{x} \mapsto a\}, \{\overline{x} \mapsto f(a)\}, \{\overline{x} \mapsto \ulcorner a, f()\urcorner\}, \{\overline{x} \mapsto \ulcorner f(a), f()\urcorner\}, \{\overline{x} \mapsto \ulcorner f(), a\urcorner\},$
$\{\overline{x} \mapsto \ulcorner f(), f(a)\urcorner\}, \{\overline{x} \mapsto \ulcorner f(), a, f()\urcorner\}, \{\overline{x} \mapsto \ulcorner f(), f(a), f()\urcorner\},$
$\{\overline{x} \mapsto \ulcorner f(), f(), a\urcorner\}, \{\overline{x} \mapsto \ulcorner f(), f(), f(a)\urcorner\}, \{\overline{x} \mapsto \ulcorner a, f(), f()\urcorner\},$
$\{\overline{x} \mapsto \ulcorner f(a), f(), f()\urcorner\}\}.$

of the complete set of unifiers of the problem, with the tree depth set to 4. □

Example 8. Let $f(\overline{x}, g(\overline{x})) \simeq^?_F f(a, b, g(a, f(), b))$ be a general flat unification problem, with f flat and g free. The the procedure computes the unique unifier $\{\overline{x} \mapsto \ulcorner a, f(), b\urcorner\}$. □

4.3 Restricted Flat Unification with Sequence Variables and Flexible Arity Symbols

The restricted flat theory with sequence variables is axiomatized by the equality $f(\overline{x}, f(\overline{y}, x, \overline{z}), \overline{w}) \simeq f(\overline{x}, \overline{y}, x, \overline{z}, \overline{w})$. In this theory only nested terms with at least one non-sequence variable argument can be flattened. Such a restriction makes matching finitary, while other properties of the flat theory are retained.

Example 9. The restricted flat unification problem $f(\overline{x}) \simeq^{?}_{RF} f(a)$ has two solutions: $\{\overline{x} \mapsto a\}$, $\{\overline{x} \mapsto f(a)\}$. □

We have implemented in FunLog the restricted flat unification procedure described in [8].

5 Conclusion and Future Work

FunLog is intended to be used in areas where problems can be specified conveniently as combinations of abstract rewrite rules. In particular, the package turned out to be useful in implementations of procedures for E-unification.

Obviously, the range of problems which can be tackled with FunLog is very large. We expect to identify more interesting problems which can be easily programmed with transformation rules. But we also expect that our future attempts to solve new problems will reveal new programming constructs which are desirable for making our programming style more expressive.

Currently, we investigate how these programming constructs can be employed to implement provers in the Theorema system [5, 6]. We also believe that our programming constructs could underlie a convenient tool to write reasoners by Theorema users and developers.

Another direction of future work is to introduce control mechanisms for pattern matching with sequence variables. The current implementation of FunLog relies entirely on the enumeration strategy of matchers which is built into the Mathematica interpreter. However, there are many situations when this enumeration strategy is not desirable. We have already addressed this problem in [11, 12] and implemented the package Sequentica with language extensions which can overwrite the default enumeration strategy of the Mathematica interpreter. The integration of those language extensions in FunLog will certainly increase the expressive power of our rule-based system. We are currently working on integrating Sequentica with FunLog.

The current implementation of FunLog can be downloaded from

`http://www.score.is.tsukuba.ac.jp/~mmarin/FunLog/`

Acknowledgements. Mircea Marin has been supported by the Austrian Academy of Sciences. Temur Kutsia has been supported by the Austrian Science Foundation (FWF) under Project SFB F1302.

References

1. H. Abdulrab and J.-P. Pécuchet. Solving word equations. *J. of Symbolic Computation*, 8(5):499–522, 1990.
2. F. Baader and K. U. Schulz. Unification in the union of disjoint equational theories: Combining decision procedures. *J. of Symbolic Computation*, 21(2):211–244, 1996.
3. F. Baader and W. Snyder. Unification theory. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 8, pages 445–532. Elsevier Science, 2001.
4. P. Borovanský, C. Kirchner, H. Kirchner, and Ch. Ringeissen. Rewriting with strategies in ELAN: a functional semantics. *International Journal of Foundations of Computer Science*, 12(1):69–98, 2001. Also available as Technical Report A01-R-388, LORIA, Nancy (France).
5. B. Buchberger, C. Dupré, T. Jebelean, F. Kriftner, K. Nakagawa, D. Vasaru, and W. Windsteiger. The Theorema project: A progress report. In M. Kerber and M. Kohlhase, editors, *Symbolic Computation and Automated Reasoning. Proc. of Calculemus'2000*, pages 98–113, St.Andrews, UK, 6–7 August 2000.
6. B. Buchberger, T. Jebelean, F. Kriftner, M. Marin, E. Tomuta, and D. Vasaru. A survey of the Theorema project. In W. Küchlin, editor, *Proceedings of the International Symposium on Symbolic and Algebraic Computation, ISSAC'97*, pages 384–391, Maui, Hawaii, US, 21–23 July 1997. ACM Press.
7. The PROTHEO Group. http://www.loria.fr/equipes/protheo/softwares/elan/.
8. T. Kutsia. *Solving and Proving in Equational Theories with Sequence Variables and Flexible Arity Symbols*. PhD thesis, Institute RISC-Linz, Johannes Kepler University, Hagenberg, Austria, June 2002.
9. T. Kutsia. Unification with Sequence Variables and Flexible Arity Symbols and its Extension with Pattern-Terms. In J. Calmet, B. Benhamou, O. Caprotti, L. Henocque, and V. Sorge, editors, editor, *Artificial Intelligence, Automated Reasoning and Symbolic Computation. Proceedings of Joint AICS'2002 - Calculemus'2002 Conference*, volume 2385 of *LNAI*, Marseille, France, 2002.
10. G. S. Makanin. The problem of solvability of equations on a free semigroup. *Math. USSR Sbornik*, 32(2), 1977.
11. M. Marin. Functional Programming with Sequence Variables: The Sequentica Package. In J. Levy, M. Kohlhase, J. Niehren, and M. Villaret, editors, *Proceedings of the 17th International Workshop on Unification (UNIF 2003)*, pages 65–78, Valencia, June 2003.
12. M. Marin and D. Ţepeneu. Programming with Sequence Variables: The Sequentica Package. In P. Mitic, P. Ramsden, and J. Carne, editors, *Challenging the Boundaries of Symbolic Computation. Proceedings of 5th International Mathematica Symposium (IMS 2003)*, pages 17–24, Imperial College, London, July 7–11 2003. Imperial College Press.
13. G. Plotkin. Building in equational theories. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 7, pages 73–90, Edinburgh, UK, 1972. Edinburgh University Press.
14. A. F. Tiu. A1-unification. Technical Report WV-01-08, Knowledge Representation and Reasoning Group, Department of Computer Science, Dresden University of Technology, 2001.

A Rules for Free Unification

Projection:	$s \simeq^?_\emptyset t \rightsquigarrow \langle\langle s\pi_1 \simeq^?_\emptyset t\pi_1,\, \pi_1\rangle, \ldots,$ $\langle s\pi_k \simeq^?_\emptyset t\pi_k,\, \pi_k\rangle\rangle$	where $\{\pi_1, \ldots, \pi_k\} = \Pi(s \simeq^?_\emptyset t)$.
Success:	$t \simeq^?_\emptyset t \rightsquigarrow \langle\langle\top,\, \varepsilon\rangle\rangle$.	
	$x \simeq^?_\emptyset t \rightsquigarrow \langle\langle\top,\, \{x \mapsto t\}\rangle\rangle$,	if $x \notin vars(t)$.
	$t \simeq^?_\emptyset x \rightsquigarrow \langle\langle\top,\, \{x \mapsto t\}\rangle\rangle$,	if $x \notin vars(t)$.
Failure:	$c_1 \simeq^?_\emptyset c_2 \rightsquigarrow \bot$,	if $c_1 \neq c_2$.
	$x \simeq^?_\emptyset t \rightsquigarrow \bot$,	if $t \neq x$ and $x \in vars(t)$.
	$t \simeq^?_\emptyset x \rightsquigarrow \bot$,	if $t \neq x$ and $x \in vars(t)$.
	$f_1(\tilde{t}) \simeq^?_\emptyset f_2(\tilde{s}) \rightsquigarrow \bot$,	if $f_1 \neq f_2$.
	$f() \simeq^?_\emptyset f(t_1, \tilde{t}) \rightsquigarrow \bot$.	
	$f(t_1, \tilde{t}) \simeq^?_\emptyset f() \rightsquigarrow \bot$.	
	$f(\overline{x}, \tilde{t}) \simeq^?_\emptyset f(s_1, \tilde{s}) \rightsquigarrow \bot$,	if $s_1 \neq \overline{x}$ and $\overline{x} \in svars(s_1)$.
	$f(s_1, \tilde{s}) \simeq^?_\emptyset f(\overline{x}, \tilde{t}) \rightsquigarrow \bot$,	if $s_1 \neq \overline{x}$ and $\overline{x} \in svars(s_1)$.
	$f(t_1, \tilde{t}) \simeq^?_\emptyset f(s_1, \tilde{s}) \rightsquigarrow \bot$,	if $t_1 \simeq^?_\emptyset s_1 \rightsquigarrow \bot$.
Eliminate:	$f(t_1, \tilde{t}) \simeq^?_\emptyset f(s_1, \tilde{s}) \rightsquigarrow \langle\langle g(\tilde{t}\sigma) \simeq^?_\emptyset g(\tilde{s}\sigma),\, \sigma\rangle\rangle$,	if $t_1 \simeq^?_\emptyset s_1 \rightsquigarrow \langle\langle\top,\, \sigma\rangle\rangle$.
	$f(\overline{x}, \tilde{t}) \simeq^?_\emptyset f(\overline{x}, \tilde{s}) \rightsquigarrow \langle\langle f(\tilde{t}) \simeq^?_\emptyset f(\tilde{s}),\, \varepsilon\rangle\rangle$.	
	$f(\overline{x}, \tilde{t}) \simeq^?_\emptyset f(s_1, \tilde{s}) \rightsquigarrow$ $\langle\langle f(\tilde{t}\sigma_1) \simeq^?_\emptyset f(\tilde{s}\sigma_1),\, \sigma_1\rangle,$ $\langle f(\overline{x}, \tilde{t}\sigma_2) \simeq^?_\emptyset f(\tilde{s}\sigma_2),\, \sigma_2\rangle\rangle$,	if $s_1 \notin \mathcal{V}_{\mathrm{Seq}}$ and $\overline{x} \notin svars(s_1)$, where $\sigma_1 = \{\overline{x} \mapsto s_1\}$, $\sigma_2 = \{\overline{x} \mapsto \ulcorner s_1, \overline{x}\urcorner\}$.
	$f(s_1, \tilde{s}) \simeq^?_\emptyset f(\overline{x}, \tilde{t}) \rightsquigarrow$ $\langle\langle f(\tilde{s}\sigma_1) \simeq^?_\emptyset f(\tilde{t}\sigma_1),\, \sigma_1\rangle,$ $\langle f(\tilde{s}\sigma_2) \simeq^?_\emptyset f(\overline{x}, \tilde{t}\sigma_2),\, \sigma_2\rangle\rangle$,	if $s_1 \notin \mathcal{V}_{\mathrm{Seq}}$ and $\overline{x} \notin svars(s_1)$, where $\sigma_1 = \{\overline{x} \mapsto s_1\}$, $\sigma_2 = \{\overline{x} \mapsto \ulcorner s_1, \overline{x}\urcorner\}$.
	$f(\overline{x}, \tilde{t}) \simeq^?_\emptyset f(\overline{y}, \tilde{s}) \rightsquigarrow$ $\langle\langle f(\tilde{t}\sigma_1) \simeq^?_\emptyset f(\tilde{s}\sigma_1),\, \sigma_1\rangle,$ $\langle f(\overline{x}, \tilde{t}\sigma_2) \simeq^?_\emptyset f(\tilde{s}\sigma_2),\, \sigma_2\rangle,$ $\langle f(\tilde{t}\sigma_3) \simeq^?_\emptyset f(\overline{y}, \tilde{s}\sigma_3),\, \sigma_3\rangle\rangle$,	where $\sigma_1 = \{\overline{x} \mapsto \overline{y}\}$, $\sigma_2 = \{\overline{x} \mapsto \ulcorner \overline{y}, \overline{x}\urcorner\}$, $\sigma_3 = \{\overline{y} \mapsto \ulcorner \overline{x}, \overline{y}\urcorner\}$.
Split:	$f(t_1, \tilde{t}) \simeq^?_\emptyset f(s_1, \tilde{s}) \rightsquigarrow$ $\langle\langle f(r_1, \tilde{t}\sigma_1) \simeq^?_\emptyset f(q_1, \tilde{s}\sigma_1),\, \sigma_1\rangle, \ldots,$ $\langle f(r_k, \tilde{t}\sigma_k) \simeq^?_\emptyset f(q_k, \tilde{s}\sigma_k),\, \sigma_k\rangle\rangle$	if $t_1, s_1 \notin \mathcal{V}_{\mathrm{Ind}} \cup \mathcal{V}_{\mathrm{Seq}}$ and $t_1 \simeq^?_\emptyset s_1 \rightsquigarrow \langle\langle r_1 \simeq^?_\emptyset q_1,\, \sigma_1\rangle,$ $\ldots, \langle r_k \simeq^?_\emptyset q_k,\, \sigma_k\rangle\rangle$.

Fig. 1. $\tilde{t}$ and $\tilde{s}$ are possibly empty sequences of terms; $\Pi(\Gamma)$ is the set of substitutions $\{\{\overline{x_1} \mapsto \ulcorner\urcorner, \ldots, \overline{x_n} \mapsto \ulcorner\urcorner\} \mid \{\overline{x_1}, \ldots, \overline{x_n}\} \subseteq svars(\Gamma)\}$; $svars(t)$ ($vars(t)$) is the set of all seq. variables (all variables) in t; f, f_1, f_2 are free (fixed or flexible) symbols; g is a new free flexible symbol, if in the same rule f is of the fixed arity, otherwise g is f.

Implementing the Clausal Normal Form Transformation with Proof Generation

Hans de Nivelle

Max Planck Institut für Informatik
Stuhlsatzenhausweg 85
66123 Saarbrücken, Germany
nivelle@mpi-sb.mpg.de

Abstract. We explain how we intend to implement the clausal normal form transformation with proof generation. We present a convenient data structure for sequent calculus proofs, which will be used for representing the generated proofs. The data structure allows easy proof checking and generation of proofs. In addition, it allows convenient implementation of proof normalization, which is necessary in order to keep the size of the generated proofs acceptable.

1 Introduction

In [2], a method for generating explicit proofs from the clausal normal form transformation was presented, which does not make use of choice axioms. It is our intention to implement this method. In this paper we introduce the data structure for the representation of proofs that we intend to use, and we give a general algorithm scheme, with which one can translate formulas and obtain correctness proofs at the same time.

In [2], natural deduction was used for showing that it is in principle possible to generate explicit proofs. It is however in practice better to use sequent calculus, because sequent calculus allows proof reductions that reduce the size of generated proofs. In order to be able to keep the sizes of the resulting proofs acceptable, it is necessary to normalize proofs in such a way that repeated building up of contexts is avoided.

In the preceeding paper [1], which was still proposing to use choice axioms, it was explained how to do this in type theory. An intermediate calculus was introduced, called the *replacement calculus*, which allows for proof normalization. After normalization, the resulting proof could be translated into type theory through a simple replacement schema. If one uses sequent calculus instead of natural deduction, the standard reductions of sequent calculus can do the proof normalization. It turns out that proof normalization in the replacement calculus corresponds to a restricted form of cut elimination in sequent calculus. Therefore, if one uses sequent calculus instead of natural deduction, the replacement calculus can be omitted alltogether.

In the next section we introduce sequent calculus. After that, we introduce the data structure that we will use for representing sequent calculus proofs. Then

we will give a general scheme for translating formulas and generating proofs at the same time. In the last section, we show that our sequent proof data structure is convenient for implementing the kind of proof reduction that we need.

2 Sequent Calculus

Definition 1. *A* sequent *is an object of form $\Gamma \vdash \Delta$, where both Γ and Δ are multisets.*

We give the rules of sequent calculus. We assume that α-equivalent formulas are not distinguished. We also give equality rules, although equality plays no rule in the CNF-transformation.

$$\text{(axiom)}\ \frac{}{\Gamma, A \vdash \Delta, A}$$

$$\text{(cut)}\ \frac{\Gamma, A \vdash \Delta \qquad \Gamma \vdash \Delta, A}{\Gamma \vdash \Delta}$$

Structural Rules:

$$\text{(weakening left)}\ \frac{\Gamma \vdash \Delta}{\Gamma, A \vdash \Delta} \qquad \text{(weakening right)}\ \frac{\Gamma \vdash \Delta}{\Gamma \vdash \Delta, A}$$

$$\text{(contraction left)}\ \frac{\Gamma, A, A \vdash \Delta}{\Gamma, A \vdash \Delta} \qquad \text{(contraction right)}\ \frac{\Gamma \vdash \Delta, A, A}{\Gamma \vdash \Delta, A}$$

Rules for the truth constants:

$$(\top\text{-left})\ \frac{\Gamma \vdash \Delta}{\Gamma, \top \vdash \Delta} \qquad (\top\text{-right})\ \frac{}{\Gamma \vdash \Delta, \top}$$

$$(\bot\text{-left})\ \frac{}{\Gamma, \bot \vdash \Delta} \qquad (\bot\text{-right})\ \frac{\Gamma \vdash \Delta}{\Gamma \vdash \Delta, \bot}$$

Rules for $\neg$:

$$(\neg\text{-left})\ \frac{\Gamma \vdash \Delta, A}{\Gamma, \neg A \vdash \Delta} \qquad (\neg\text{-right})\ \frac{\Gamma, A \vdash \Delta}{\Gamma \vdash \Delta, \neg A}$$

Rules for $\wedge, \vee, \leftarrow, \leftrightarrow$:

$$(\wedge\text{-left})\ \frac{\Gamma, A, B \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta} \qquad (\wedge\text{-right})\ \frac{\Gamma \vdash \Delta, A \qquad \Gamma \vdash \Delta, B}{\Gamma \vdash \Delta, A \wedge B}$$

$$(\vee\text{-left})\ \frac{\Gamma,\ A \vdash \Delta \qquad \Gamma, B \vdash \Delta}{\Gamma, A \vee B \vdash \Delta} \qquad (\vee\text{-right})\ \frac{\Gamma \vdash \Delta, A, B}{\Gamma \vdash \Delta, A \vee B}$$

$$(\to\text{-left})\ \frac{\Gamma \vdash \Delta, A \quad \Gamma, B \vdash \Delta}{\Gamma, A \to B \vdash \Delta} \qquad (\to\text{-right})\ \frac{\Gamma, A \vdash \Delta, B}{\Gamma \vdash \Delta, A \to B}$$

$$(\leftrightarrow\text{-left})\ \frac{\Gamma, A \to B,\ B \to A \vdash \Delta}{\Gamma, A \leftrightarrow B \vdash \Delta}$$

$$(\leftrightarrow\text{-right})\ \frac{\Gamma \vdash \Delta, A \to B \quad \Gamma \vdash \Delta, B \to A}{\Gamma \vdash \Delta, A \leftrightarrow B}$$

Rules for the quantifiers:

$$(\forall\text{-left })\ \frac{\Gamma, P[x := t] \vdash \Delta}{\Gamma, \forall x\ P \vdash \Delta} \qquad (\forall\text{-right })\ \frac{\Gamma \vdash \Delta, P[x := y]}{\Gamma \vdash \Delta, \forall x\ P}$$

$$(\exists\text{-left })\ \frac{\Gamma, P[x := y] \vdash \Delta}{\Gamma, \exists x\ P \vdash \Delta} \qquad (\exists\text{-right })\ \frac{\Gamma \vdash \Delta, P[x := t]}{\Gamma \vdash \Delta, \exists x\ P}$$

The t is an arbitrary term. The y is a variable which is not free in Γ, Δ, P

Rules for equality:

$$(\text{refl})\ \frac{}{\Gamma \vdash \Delta,\ t \approx t}$$

$$(\text{repl-left })\ \frac{t_1 \approx t_2,\ \Gamma[t_1] \vdash \Delta}{t_1 \approx t_2,\ \Gamma[t_2] \vdash \Delta} \qquad (\text{repl-right })\ \frac{t_1 \approx t_2,\ \Gamma \vdash \Delta[t_1]}{t_1 \approx t_2,\ \Gamma \vdash \Delta[t_2]}.$$

The last rules mean: If $t_1 \approx t_2$ appears among the premisses, then an arbitrary occurrence of t_1 can be replaced by t_2. The replacement can take place either on the left or on the right. Only one replacement at the same time is possible.

3 Proof Trees

We introduce a concise sequent calculus format, which allows for easy proof checking and implementation of proof reductions. It is closely related to the embedding of sequent calculus in LF, which is introduced in [5].

We first prove a simple lemma that shows that one should avoid explicitly mentioning the formulas occurring in the proof:

Lemma 1. *Consider the sequents* $(\neg\neg)^n A \vdash A$, *for* $n \geq 0$.

If one has a proof representation method that explicitly mentions the formulas in a sequent, then the proofs have size $O(n^2)$.

Proof. Because one will have to represent all subformulas $A, \neg A, (\neg)^2 A, (\neg)^3 A, \ldots, (\neg\neg)^n A$.

Nevertheless, the proof has a length of only n steps. If one does not mention the formulas, one can obtain a representation of size n. In our representation, we avoid explicitly mentioning formulas by assigning labels to them. Whenever a new formula is constructed, it will be clear what the new formula is, from the way it is constructed, so that we will not have to mention it.

Definition 2. *We redefine a sequent as an object of form $\Gamma \vdash \Delta$, where both Γ and Δ are sets of labelled formulas. So we have $\Gamma = \{\alpha_1 : A_1, \ldots, \alpha_p : A_p\}$ and $\Delta = \{\beta_1 : B_1, \ldots, \beta_q : B_q\}$, where $\alpha_i = \alpha_j$ implies $i = j$ and $\beta_i = \beta_j$ implies $i = j$.*

In case there is no A', s.t. $\alpha : A' \in \Gamma$, the notation $\Gamma + \alpha : A$ denotes $\Gamma \cup \{\alpha : A\}$. Otherwise, $\Gamma + \alpha : A$ is undefined. (even when $A = A'$)

In case there is an A, s.t. $\alpha : A \in \Gamma$, the notation $\Gamma - \alpha$ denotes $\Gamma \backslash \{\alpha : A\}$. Otherwise $\Gamma - \alpha$ is not defined.

In case there is an A, s.t. $\alpha : A \in \Gamma$, the notation $\Gamma[\alpha]$ denotes A. Otherwise $\Gamma[\alpha]$ is not defined.

For Δ, we define $\Delta + \beta : B$, $\Delta - \beta$, $\Delta[\beta]$ in the same way as for Γ.

Proofs are checked *top-down*, i.e. from the goal sequent towards the axioms. For each node in the proof tree, the node states the label of the conslusion in the derived sequent, and what labels the premisses should receive in the child sequents. During checking, the conclusion is removed from the sequent (if it exists, and has the right form), and replaced by the children, after which proof checking continues.

Definition 3. *We recursively define* proof trees *and when a proof tree* accepts *a labelled sequent. In the following list, we implicitly assume that α, β are labels. We will omit the definedness conditions. So we will assume that $\Delta[\alpha] = \Delta[\beta]$ means: $F[\alpha]$ and $F[\beta]$ are both defined and $F[\alpha] = F[\beta]$.*

- $\mathrm{ax}(\alpha, \beta)$ *is a proof tree. It is a proof of $\Gamma \vdash \Delta$, if $\Gamma[\alpha]$ is an α-variant of $\Delta[\beta]$.*
- *If π_1, π_2 are proof trees, and A is a formula, then* $\mathrm{cut}(A, \pi_1, \alpha, \pi_2, \beta)$ *is also a proof tree. It is a proof of $\Gamma \vdash \Delta$ if π_1 is a proof of $\Gamma + \alpha : A \vdash \Delta$ and π_2 is a proof of $\Gamma \vdash \Delta + \beta : A$.*
- *If π is a proof tree, then* $\mathrm{weakenleft}(\alpha, \pi)$ *is also a proof tree. It is a proof of $\Gamma \vdash \Delta$ if π is a proof of $\Gamma - \alpha \vdash \Delta$.*
- *If π is a proof tree, then* $\mathrm{weakenright}(\beta, \pi)$ *is also a proof tree. It is a proof of $\Gamma \vdash \Delta$ if π is a proof of $\Gamma \vdash \Delta - \beta$.*
- *If π is a proof tree, then* $\mathrm{contrleft}(\alpha_1, \pi, \alpha_2)$ *is also a proof tree. It is a proof of $\Gamma \vdash \Delta$ if π is a proof of $\Gamma + \alpha_1 : A[\alpha_2] \vdash \Delta$.*
- *If π is a proof tree, then* $\mathrm{contrright}(\beta_1, \pi, \beta_2)$ *is also a proof tree. It is a proof of $\Gamma \vdash \Delta$ if π is a proof of $\Gamma \vdash \Delta + \beta_1 : \Delta[\beta_2]$.*
- *If π is a proof tree, then* $\mathrm{trueleft}(\alpha, \pi)$ *is also a proof tree. It is a proof of $\Gamma \vdash \Delta$ if $\Gamma[\alpha] = \top$, and π is a proof of $\Gamma - \alpha \vdash \Delta$.*

- $\mathrm{trueright}(\beta)$ *is a proof tree. It is a proof of* $\Gamma \vdash \Delta$ *if* $\Delta[\beta] = \top$.
- $\mathrm{falseleft}(\alpha)$ *is a proof tree. It is a proof of* $\Gamma \vdash \Delta$ *if* $\Gamma[\alpha] = \bot$.
- $\mathrm{falseright}(\beta, \pi)$ *is a proof tree. It is a proof of* $\Gamma \vdash \Delta$ *if* $\Delta[\beta] = \bot$ *and* π *is a proof of* $\Gamma \vdash \Delta - \beta$.
- *If* π *is a proof tree, then* $\mathrm{negleft}(\alpha, \pi, \beta)$ *is also a proof tree. It is a proof of* $\Gamma \vdash \Delta$ *if* $\Gamma[\alpha]$ *has form* $\neg A$, *and* π *is a proof of* $\Gamma \vdash \Delta + \beta{:}\, A$.
- *If* π *is a proof tree, then* $\mathrm{negright}(\beta, \pi, \alpha)$ *is also a proof tree. It is a proof of* $\Gamma \vdash \Delta$ *if* $\Delta[\beta]$ *has form* $\neg A$, *and* π *is a proof of* $\Gamma + \alpha{:}\, A \vdash \Delta$.
- *If* π *is a proof tree and* $\alpha_1 \neq \alpha_2$, *then* $\mathrm{andleft}(\alpha, \pi, \alpha_1, \alpha_2)$ *is also a proof tree. It is a proof of* $\Gamma \vdash \Delta$ *if* $\Gamma[\alpha]$ *has form* $A \wedge B$, *and*
  π *is a proof of* $(\Gamma - \alpha) + \alpha_1{:}\, A + \alpha_2{:}\, B \vdash \Delta$.
- *If* π_1, π_2 *are proof trees, then* $\mathrm{andright}(\beta, \pi_1, \beta_1, \pi_2, \beta_2)$ *is also a proof tree. It is a proof of* $\Gamma \vdash \Delta$ *if* $\Delta[\beta]$ *has form* $A \wedge B$,
  π_1 *is a proof of* $\Gamma \vdash (\Delta - \beta) + \beta_1{:}\, A$, *and*
  π_2 *is a proof of* $\Gamma \vdash (\Delta - \beta) + \beta_2{:}\, B$.
- *If* π_1, π_2 *are proof trees, then* $\mathrm{orleft}(\alpha, \pi_1, \alpha_1, \pi_2, \alpha_2)$ *is also a proof tree. It is a proof of* $\Gamma \vdash \Delta$ *if* $\Delta[\alpha]$ *has form* $A \vee B$,
  π_1 *is a proof of* $(\Gamma - \alpha) + \alpha_1{:}\, A \vdash \Delta$, *and*
  π_2 *is a proof of* $(\Gamma - \alpha) + \alpha_2{:}\, B \vdash \Delta$.
- *If* π *is a proof tree and* $\beta_1 \neq \beta_2$, *then* $\mathrm{orright}(\beta, \pi, \beta_1, \beta_2)$ *is also a proof tree. It is a proof of* $\Gamma \vdash \Delta$ *if* $\Delta[\beta]$ *has form* $A \vee B$, *and*
  π *is a proof of* $\Gamma \vdash (\Delta - \beta) + \beta_1{:}\, A + \beta_2{:}\, B$.
- *If* π *is a proof tree, then* $\mathrm{impliesleft}(\alpha, \pi_1, \beta_1, \pi_2, \alpha_2)$ *is also a proof tree. It is a proof of* $\Gamma \vdash \Delta$ *if* $\Gamma[\alpha]$ *has form* $A \to B$, *and*
  π_1 *is a proof of* $(\Gamma - \alpha) \vdash \Delta + \beta_1{:}\, A$, *and*
  π_2 *is a proof of* $(\Gamma - \alpha) + \alpha_2{:}\, B \vdash \Delta$.
- *If* π *is a proof tree and* $\alpha_1 \neq \beta_2$, *then* $\mathrm{impliesright}(\beta, \pi_1, \alpha_1, \pi_2, \beta_2)$ *is also a proof tree. It is a proof of* $\Gamma \vdash \Delta$ *if* $\Delta[\beta]$ *has form* $A \to B$,
  π_1 *is a proof of* $\Gamma + \alpha_1{:}\, A \vdash (\Delta - \beta) + \beta_2{:}\, B$.
- *If* π *is a proof tree and* $\alpha_1 \neq \alpha_2$, *then* $\mathrm{equivleft}(\alpha, \pi, \alpha_1, \alpha_2)$ *is also a proof tree. It is a proof of* $\Gamma \vdash \Delta$ *if* $\Gamma[\alpha]$ *has form* $A \leftrightarrow B$, *and*
  π *is a proof of* $(\Gamma - \alpha) + \alpha_1{:}\, (A \to B) + \alpha_2{:}\, (B \to A) \vdash \Delta$.
- *If* π_1, π_2 *are proof trees, then* $\mathrm{equivright}(\beta, \pi_1, \beta_1, \pi_2, \beta_2)$ *is also a proof tree. It is a proof of* $\Gamma \vdash \Delta$ *if* $\Delta[\beta]$ *has form* $A \leftrightarrow B$,
  π_1 *is a proof of* $\Gamma \vdash (\Delta - \beta) + \beta_1{:}\, (A \to B)$, *and*
  π_2 *is a proof of* $\Gamma \vdash (\Delta - \beta) + \beta_2{:}\, (B \to A)$.
- *If* π *is a proof tree and* t *is a term, then* $\mathrm{forallleft}(\alpha, \pi, \alpha_1, t)$ *is also a proof tree. It is a proof of* $\Gamma \vdash \Delta$ *if* $\Gamma[\alpha]$ *has form* $\forall x\ P$ *and* π *is a proof of* $(\Gamma - \alpha) + \alpha_1{:}\, P[x := t] \vdash \Delta$.
- *If* π *is a proof tree and* y *is a variable, then* $\mathrm{forallright}(\beta, \pi, \beta_1, y)$ *is also a proof tree. It is a proof of* $\Gamma \vdash \Delta$ *if* $\Delta[\beta]$ *has form* $\forall x\ P$,
  y *is not free in* Γ, Δ *or* P, *and*
  π *is a proof of* $\Gamma \vdash (\Delta - \beta) + \beta_1{:}\, P[x := y]$.
- *If* π *is a proof tree and* y *is a variable, then* $\mathrm{existsleft}(\alpha, \pi, \alpha_1, y)$ *is also a proof tree. It is a proof of* $\Gamma \vdash \Delta$ *if* $\Gamma[\alpha]$ *has form* $\exists x\ P$,
  y *is not free in* Γ, Δ *or* P, *and*
  π *is a proof of* $(\Gamma + \alpha) + \alpha_1{:}\, P[x := y] \vdash \Delta$

- *If π is a proof tree and t is a term, then* existsright(β, π, β_1, t) *is also a proof tree. It is a proof of $\Gamma \vdash \Delta$ if $\Delta[\beta]$ has form $\exists x\ P$ and*
  π is a proof of $\Gamma \vdash (\Delta - \beta) + \beta_1 : P[x := t]$.
- *If t is a term, then* eqrefl(β, t) *is a proof tree. It is a proof of $\Gamma \vdash \Delta$ if $\Delta[\beta] = (t \approx t)$.*
- *If π is a proof tree and ρ is a position, then* replleft$(\alpha_1, \alpha_2, \pi, \rho, \alpha_3)$ *is also a proof tree. It is a proof of $\Gamma \vdash \Delta$ if $\Gamma[\alpha_1]$ has form $t_1 \approx t_2$, if $\Gamma[\alpha_2]$ has form $A_\rho[t_2]$, and π is a proof of $(\Gamma - \alpha_2) + \alpha_3 : A_\rho[t_1] \vdash \Delta$.*
- *If π is a proof tree and ρ is a position, then* replright$(\alpha, \beta_1, \pi, \rho, \beta_2)$ *is also a proof tree. It is a proof of $\Gamma \vdash \Delta$ if $\Gamma[\alpha]$ has form $t_1 \approx t_2$, if $\Delta[\beta_1]$ has form $B_\rho[t_1]$, and π is a proof of $\Gamma \vdash (\Delta - \beta_1) + \beta_2 : B_\rho[t_2]$.*

As an example, consider the following proof:

$$\dfrac{\dfrac{\alpha_1 : A,\ \alpha_2 : B \vdash \beta_1 : B \qquad \alpha_1 : A,\ \alpha_2 : B \vdash \beta_2 : A}{\alpha_1 : A,\ \alpha_2 : B \vdash \beta : B \wedge A}}{\alpha : A \wedge B \vdash \beta : B \wedge A}$$

It can be represented by the following proof term:

$$\text{andleft}(\alpha, \text{andright}(\beta, \text{ax}(\alpha_2, \beta_1), \beta_1, \text{ax}(\alpha_1, \beta_2), \beta_2), \alpha_1, \alpha_2).$$

Following [5], we consider a rule as a binder that binds the labels that it introduces in the subproofs where the label is introduced. For example, andleft$(\alpha, \pi, \alpha_1, \alpha_2)$ introduces the labels α_1, α_2 in π. Therefore, it can be viewed as binding any occurrences of α_1, α_2 in π. Likewise, we consider forallright(β, π, β_1, y), as a binder that binds any occurrences of y in π.

Viewing the rules as binders makes it possible to define a notion of α-equivalence of proofs. This has the advantage that label conflicts can be resolved by renaming labels. Without α-equivalence, a rule introducing some formula with label α cannot be applied on a labelled sequent already containing α. However, if we use α-equivalence, we can rename α into a new label α' and continue proof checking. As an example, the proof tree given a few lines above would not be a proof of $\alpha : A \wedge B,\ \alpha_1 : A \vdash \beta : B \wedge A$. Using α-equivalence, we can replace α_1 in the proof tree by some α_1' and the sequent will be accepted. The main advantage of this is that proof checking becomes monotone:

Lemma 2. *If π is a proof tree, which is a proof of some labelled sequent $\Gamma \vdash \Delta$, and $\Gamma \subseteq \Gamma'$, $\Delta \subseteq \Delta'$, then π is also a proof of $\Gamma' \vdash \Delta'$.*

The following property is important for proof reductions. It is assumed that substitution is capture avoiding:

Lemma 3. *Let π a proof of labelled sequent $\Gamma \vdash \Delta$ containing a free variable x. Let t be some term. Then $\pi[x := t]$ is a proof of $(\Gamma \vdash \Delta)[x := t]$.*

The following property is important, because it makes it possible to use proof terms as schemata, i.e. as objects that can be instantiated.

Theorem 1. *Let π be a proof of a labelled sequent $\Gamma \vdash \Delta$. Let $A(x_1, \ldots, x_n)$ be an n-ary atom occurring in $\Gamma \vdash \Delta$, s.t. $x_1, \ldots, x_n$ are its free variables. Let $F(x_1, \ldots, x_n, y_1, \ldots, y_m)$ be a formula having at least free variables $x_1, \ldots, x_n$, and with possible other free variables $y_1, \ldots, y_m$. Assume that no occurrence of $A(x_1, \ldots, x_n)$ in $\Gamma \vdash \Delta$ is in the scope of a quantifier that binds one of the y_j and that no occurrence of $A(x_1, \ldots, x_n)$ in a cut formula occurring in π is in the scope of a quantifier that binds one of the y_j. Let π' be obtained from π by substituting $A(x_1, \ldots, x_n) := F(x_1, \ldots, x_n, y_1, \ldots, y_m)$ in every cut formula in π. Then π' is a proof of $(\Gamma \vdash \Delta)[A(x_1, \ldots, x_n) := F(x_1, \ldots, x_n, y_1, \ldots, y_m)]$.*

Note that when π is cut free, then $\pi' = \pi$. The reason that Theorem 1 holds, is the fact that the cut rule is the only rule that explicitly mentions formulas.

In case variables from $y_1, \ldots, y_m$ are caught, it is always possible to obtain an α-variant of $\Gamma \vdash \Delta$ and π, s.t. no capture takes place. The same holds for any cut formula in π.

As an example, consider the sequent $\forall x(A(x) \wedge B) \vdash (\forall x A(x)) \wedge B$, which clearly has a cut free proof. Lemma 1 allows to substitute $P(y, z)$ for B, (because y and z are not caught), but it does not allow to substitute $P(x, y, z)$ for B. The sequent can be renamed into $\forall x_1(A(x_1) \wedge B) \vdash (\forall x_1 A(x_1)) \wedge B$.

4 The Negation Normal Form Transformation

We describe in detail how we intend to implement the negation normal form transformation with proof generation.

Definition 4. *Formula F is in* negation normal form *(NNF) if* **(1)** *F does not contain $\rightarrow$ or $\leftrightarrow$,* **(2)** *negation is applied only on atoms in F,* **(3)** *if F contains $\top$ (or $\bot$,) then $F = \top$, (or $\bot$).*

A formula can be easily transformed into NNF by two rewrite systems. The first rewrite system removes $\rightarrow$ and $\leftrightarrow$, and it pushes the negations inwards. The second rewrite system moves $\bot$ and $\top$ upwards until they either disappear, or reach the top of the formula. The rewrite systems could be combined into one rewrite system, but that would be inefficient, because the two rewrite systems are more efficient with different rewrite systems. The first rewrite system is given by the following table:

$$
\begin{array}{ll}
A \rightarrow B & \Rightarrow \neg A \vee B \\
A \leftrightarrow B & \Rightarrow (\neg A \vee B) \wedge (A \vee \neg B)
\end{array}
$$

$$
\begin{array}{ll}
\neg\neg A & \Rightarrow A \\
\neg(A \vee B) & \Rightarrow \neg A \wedge \neg B \\
\neg(A \wedge B) & \Rightarrow \neg A \vee \neg B \\
\neg(\forall x\ P(x)) & \Rightarrow \exists x\ \neg P(x) \\
\neg(\exists x\ P(x)) & \Rightarrow \forall x\ \neg P(x)
\end{array}
$$

The following algorithm normalizes a formula under the set of rules.

Algorithm 1
formula nnf12(formula$F$)
begin
  while there are a rule $A \Rightarrow B$ *and a substitution* Θ, *s.t.*
  $A\Theta = F$ *do*
    $F := B\Theta$
  if F *is an atom,* $A = \bot$ *or* $A = \top$, *then return* F
  if F *has form* $\neg A$, *with* A *an atom,* $A = \bot$, *or* $A = \top$, *then return* $\neg A$.
  if F *has form* $A \wedge B$, *then return* $\mathrm{nnf12}(A) \wedge \mathrm{nnf12}(B)$
  if F *has form* $A \vee B$, *then return* $\mathrm{nnf12}(A) \vee \mathrm{nnf12}(B)$
  if F *has form* $\forall x\ P(x)$, *then return* $\forall x\ \mathrm{nnf12}(P(x))$
  if F *has form* $\exists x\ P(x)$, *then return* $\exists x\ \mathrm{nnf12}(P(x))$
end

The algorithm implements a particular rewrite strategy, namely *outside-inside* normalization. It assumes that the rewrite front starts at the outside and then moves inward. When the formula has been normalized at one point, then this point does not need to be reconsidered anymore. There second part of the rewrite system needs exactly the opposite strategy, *inside-outside* normalization. If one would combine the systems, one would have to look for possible rewrites everywhere in the formula, which is less efficient.

Definition 5. *A* justified sequent *is a pair of form* $(\alpha{:}\, A \vdash \beta{:}\, B,\ \pi)$, *s.t.* $\alpha \neq \beta$ *and* π *is a proof of* $\alpha{:}\, A \vdash \beta{:}\, B$. *A* justified rewrite rule *is a justified sequent* $(\alpha{:}\, A \vdash \beta{:}\, B,\ \pi)$, *s.t.* $A \Rightarrow B$ *is a rewrite rule.*

There is no formal distinction between a justified sequent and a justified rewrite rule, but we give them different names because their roles are different.

We now modify the rewrite algorithm, so that it will output a proof at the same time with its result. It will do this by returning a justified sequent.

Algorithm 2 *Function* $\mathrm{nnf12}(F, \alpha)$ *returns a justified sequent* $(\alpha{:}\, F \vdash \beta{:}\, F',\ \pi)$, *s.t.* $F' = \mathrm{nnf12}(F)$, *and* β *is some new label.*

justifiedsequent nnf12(formula F, label α)
begin
array of justifiedsequent Π;
Initialize Π *to the empty (zero length) array.*
while there are a justified rewrite rule $(\alpha' : A' \vdash \beta' : B',\ \pi')$ *and*
a substitution Θ, *s.t.* $A'\Theta = F$ *do*
begin
Let γ *be a new label, not occurring in* Π, *and distinct from* α.
Assign $\pi' := \pi'[\alpha' := \alpha,\ \beta' := \gamma]$. *(so that* π' *now proves* $\alpha : A' \vdash \gamma : B$*)*
Append $(\alpha : A'\Theta \vdash \gamma : B'\Theta,\ \pi')$ *to* Π. *(the length of* Π *is increased by* 1,
there is no need to modify π' *because of Theorem 1)*
Assign $F := B\Theta$.
Assign $\alpha := \gamma$.
end
If F *is an atom,* $F = \bot$ *or* $F = \top$, *then*
return applycut(Π).
If F *has form* $\neg A$, *where* A *is an atom,* $A = \bot$ *or* $A = \top$, *then*
return applycut(Π).
If F *has form* $A_1 \wedge A_2$, *then*
begin
Let $\alpha_1, \alpha_2, \beta$ *be new, distinct labels.*
Assign B_1, β_1, π_1 *from* $(\alpha_1 : A_1 \vdash \beta_1 : B_1,\ \pi_1) :=$ nnf12(A_1, α_1)
Assign B_2, β_2, π_2 *from* $(\alpha_2 : A_2 \vdash \beta_2 : B_2,\ \pi_2) :=$ nnf12(A_2, α_2)
Append $\alpha : A_1 \wedge A_2 \vdash \beta : B_1 \wedge B_2$,
andleft(α,
andright(β,
weakenleft(α_2, π_1), β_1,
weakenleft(α_1, π_2), β_2),
α_1, α_2)) *to* Π.
return applycut(Π)
end
If F *has form* $A_1 \vee A_2$, *then*
begin
Let $\alpha_1, \alpha_2, \beta$ *be new, distinct labels.*
Assign B_1, β_1, π_1 *from* $(\alpha_1 : A_1 \vdash \beta_1 : B_1,\ \pi_1) :=$ nnf12(A_1, α_1)
Assign B_2, β_2, π_2 *from* $(\alpha_2 : A_2 \vdash \beta_2 : B_2,\ \pi_2) :=$ nnf12(A_2, α_2)
Append $(\alpha : A_1 \vee A_2 \vdash \beta : B_1 \vee B_2$,
orright(β,
orleft(α,
weakenright(β_2, π_1), α_1,
weakenright(β_1, π_2), α_2),
β_1, β_2)) *to* Π.
return applycut(Π).
end

If F *has form* $\forall x\ P(x)$, *then*
begin
Let α_1 *and* β *be a new, distinct labels.*
Assign $Q(x), \beta_1, \pi_1$ *from* $(\alpha_1{:}P(x) \vdash \beta_1{:}Q(x),\ \pi_1) := \mathrm{nnf12}(P(x), \alpha_1)$
Append $(\alpha{:}\forall x\ P(x) \vdash \beta{:}\forall x\ Q(x),$
$\mathrm{forallright}(\beta, \mathrm{forallleft}(\alpha, \pi_1, \alpha_1, x), \beta_1, x)\ )$ *to* Π.
return $\mathrm{applycut}(\Pi)$
end

If F *has form* $\exists x\ P(x)$, *then*
begin
Let α_1 *and* β *be new, distinct labels.*
Assign $Q(x), \beta_1, \pi_1$ *from* $(\alpha_1{:}P(x) \vdash \beta_1{:}Q(x),\ \pi_1) := \mathrm{nnf12}(P(x), \alpha_1)$
Append $(\alpha{:}\exists x\ P(x) \vdash \beta{:}\exists x\ Q(x),$
$\mathrm{existsleft}(\alpha, \mathrm{existsright}(\beta, \pi_1, \beta_1, x), \alpha_1, x)\ )$ *to* Π.
return $\mathrm{applycut}(\Pi)$
end
end

Function $\mathrm{applycut}(\Pi)$ *combines the proofs* π_i *of* $\alpha_i{:}A_i \vdash \beta_i{:}B_i$ *into one proof by using the cut rule. It must be the case that* $\beta_{i+1} = \alpha_i$, *and* $B_{i+1} = A_i$, *for* $1 \leq i < |\Pi|$.

(justifiedsequent) applycut(array of justifiedsequent Π)
begin
Σ *is a variable of type labelled sequent.*
π *is a variable of type proof tree.*
Assign $(\Sigma, \pi) := \Pi_1$

for $i := 2$ *to* $|\Pi|$ *do*
begin
Assign $(\alpha{:}A \vdash \beta{:}B) := \Sigma$
Assign $(\beta{:}B \vdash \gamma{:}C,\ \rho) := \Pi_i$
Assign $\Sigma := \alpha{:}A \vdash \gamma{:}C$
Assign $\pi := \mathrm{cut}(B, \mathrm{weakenleft}(\alpha, \rho), \beta, \mathrm{weakenright}(\gamma, \pi), \beta)$
end
return $(\Sigma,\ \pi)$
end

We now come to the second part of the rewrite system that will ensure the third condition of Definition 4.

$$
\begin{array}{ll}
A \vee \bot \Rightarrow A & A \wedge \bot \Rightarrow \bot \\
A \vee \top \Rightarrow \top & A \wedge \top \Rightarrow A \\
\bot \vee A \Rightarrow A & \bot \wedge A \Rightarrow \bot \\
\top \vee A \Rightarrow \top & \top \wedge A \Rightarrow A
\end{array}
$$

$$
\begin{array}{ll}
\forall x\ \bot\ \Rightarrow \bot & \exists x\ \bot\ \Rightarrow \bot \\
\forall x\ \top\ \Rightarrow \top & \exists x\ \top\ \Rightarrow \top
\end{array}
$$

In order to obtain a normal form, Algorithm 1 cannot be used, because the outside-inside strategy does generally not result in a normal form. Instead, an inside-outside rewrite strategy has to be used:

Algorithm 3
formula nnf3 (formula F)
begin
if F *is an atom,* $A = \bot$ *or* $A = \top$, *then* $G := F$
if F *has form* $\neg A$, *with* A *an atom,* $A = \bot$, *or* $A = \top$, *then* $G := F$
if F *has form* $A \wedge B$, *then* $G := \mathrm{nnf3}(A) \wedge \mathrm{nnf3}(B)$
if F *has form* $A \vee B$, *then* $G := \mathrm{nnf3}(A) \vee \mathrm{nnf3}(B)$
if F *has form* $\forall x\ P(x)$, *then* $G := \forall x\ \mathrm{nnf3}(P(x))$
if F *has form* $\exists x\ P(x)$, *then* $G := \exists x\ \mathrm{nnf3}(P(x))$

while there are a rule $A \Rightarrow B$ *and a substitution* Θ, *s.t.* $A\Theta = G$ *do*
$G := B\Theta$
end

Algorithm 3 differs from Algorithm 1 only in the fact that rewriting on the current level is attempted only after the subterms have been normalized.

Algorithm 2 can be easily modified correspondingly, by moving the while-loop in the beginning towards the end. It can be also easily adopted to situations where more complicated rewrite strategies are needed.

5 Subformula Replacement

Some steps in the clausal normal form transformation can cause exponential blowup of the formula. The problematic steps are the replacement of $A \leftrightarrow B$ by $(\neg A \vee B) \wedge (A \vee \neg B)$, and the factoring of conjunctions over disjunctions performed by the following rules: $(A \wedge B) \vee C \Rightarrow (A \vee C) \wedge (B \vee C)$, $A \vee (B \wedge C) \Rightarrow (A \vee B) \wedge (A \vee C)$.
Expansion of $\leftrightarrow$ would cause exponential blowup on the following sequence of formulas

$$(a_1 \leftrightarrow (a_2 \leftrightarrow \cdots (a_{n-1} \leftrightarrow a_n))),\ n > 0.$$

Factoring would cause exponential blowup on the following sequence of formulas

$$(a_1 \wedge b_1) \vee \cdots \vee (a_n \wedge b_n),\ n > 0.$$

In order to avoid this, it is possible to use subformula replacement. For example, in the last formula, one can introduce new symbols $x_1, \ldots, x_n$, and replace it by the equisatisfiable set of formulas

$$x_1 \vee \cdots \vee x_n,\ x_1 \leftrightarrow (a_1 \wedge b_1), \ldots, x_n \leftrightarrow (a_n \wedge b_n).$$

Subformula replacement as such is not first-order, but it can be easily dealt with within first-order logic, by observing that the new names are abbreviations of certain formulas. During the CNF-transformation, we allow to add premisses of the following form to the set of premisses:

$$\forall x_1 \cdots x_n\ X(x_1, \ldots, x_n) \leftrightarrow F(x_1, \ldots, x_n).$$

X is a new symbol that does not yet occur in the premisses and also not in $F(x_1, \ldots, x_n)$. When the resolution prover succeeds, one obtains a proof π of a sequent $\Gamma, D_1, \ldots, D_k \vdash \bot$, in which Γ is the set of original first-order formulas, and $D_1, \ldots, D_k$ are the introduced premisses, which are all of form

$$\forall x_1 \cdots x_{n_j}\ X_j(x_1, \ldots, x_{n_j}) \leftrightarrow F_j(x_1, \ldots, x_{n_j}),\ \text{for } 1 \leq j \leq k.$$

A new symbol X_j can occur in $F_{j'}$ only when $j' > j$, and it cannot occur in Γ. By substituting the X_j away and applying Theorem 1, the proof π can be transformed into a proof π' of $\Gamma, E_1, \ldots, E_k \vdash \bot$ in which each E_j has form

$$\forall x_1 \cdots x_{n_j}\ F(x_1, \ldots, x_{n_j}) \leftrightarrow F(x_1, \ldots, x_{n_j}).$$

These are simple tautologies which can be proven and cut away.

6 Antiprenexing

The purpose of anti-prenexing (also called miniscoping) is to obtain smaller Skolem terms. In many formulas, not everything that is in the scope of a quantifier, does also depend on this quantifier. If one systematically factors such subformulas out of the scope of the quantifier, one can often reduce dependencies between quantifiers. For details, we refer to [4], here we give only a few examples:

Example 1. Without anti-prenexing, $\forall x\ \exists y[\ p(x) \wedge q(y)]$ skolemizes into $\forall x\ [p(x) \wedge q(f(x))]$. Antiprenexing reduces the formula to $(\ \forall x\ p(x)\ ) \wedge (\ \exists y\ q(y)\ )$, which Skolemizes into $(\ \forall x\ p(x)\ ) \wedge q(c)$.

Without anti-prenexing, $\forall x\ \exists y_1 y_2\ [p(y_1) \wedge q(x, y_2)]$ skolemizes into $\forall x\ [\ p(f_1(x)) \wedge q(x, f_2(x))\ ]$. Antiprenexing reduces the formula to $\forall x\ [\exists y_1\ p(y_1) \wedge \exists y_2\ q(x, y_2)]$, which Skolemizes into $\forall x\ [p(c_1) \wedge q(f_2(x))]$.

Without anti-prenexing, $\forall x\ \exists y\ [p(x) \wedge q(y) \wedge r(x)]$ skolemizes into $\forall x[p(x) \wedge q(f(x)) \wedge r(x)]$. Antiprenexing can reduce the formula to $\forall x\ [p(x) \wedge r(x) \wedge \exists y\ q(y)]$, which can be Skolemized into $\forall x\ [p(x) \wedge r(x) \wedge q(c)]$.

As far as we can see, all replacements can be handled by the following 'rewrite system':

$$
\begin{array}{ll}
A \vee B & \Rightarrow B \vee A \\
A \vee (B \vee C) & \Rightarrow A \vee B \vee C
\end{array}
\qquad
\begin{array}{ll}
A \wedge B & \Rightarrow B \wedge A \\
A \wedge (B \wedge C) & \Rightarrow A \wedge B \wedge C
\end{array}
$$

$$
\begin{array}{ll}
\forall x\ (P(x) \wedge Q) & \Rightarrow (\forall x\ P(x)) \wedge Q \\
\forall x\ (P \wedge Q(x)) & \Rightarrow P \wedge \forall x\ Q(x) \\
\forall x\ (P(x) \vee Q) & \Rightarrow (\forall x\ P(x)) \vee Q \\
\forall x\ (P \vee Q(x)) & \Rightarrow P \vee \forall x\ Q(x)
\end{array}
\qquad
\begin{array}{ll}
\exists x\ (P(x) \wedge Q) & \Rightarrow (\exists x\ P(x)) \wedge Q \\
\exists x\ (P \wedge Q(x)) & \Rightarrow P \wedge \exists x\ Q(x) \\
\exists x\ (P(x) \vee Q) & \Rightarrow (\exists x\ P(x)) \vee Q \\
\exists x\ (P \vee Q(x)) & \Rightarrow P \vee \exists x\ Q(x)
\end{array}
$$

$$
\begin{array}{ll}
\forall x\ P & \Rightarrow P \\
\forall x \forall y\ P(x,y) & \Rightarrow \forall y \forall x\ P(x,y)
\end{array}
\qquad
\begin{array}{ll}
\exists x\ P & \Rightarrow P \\
\exists x \exists y\ P(x,y) & \Rightarrow \exists y \exists x\ P(x,y)
\end{array}
$$

The system is not a rewrite system in the usual sense, because an additional strategy is needed for deciding when a certain rule should be applied. Straightforward normalization would not terminate due to the presence of permutation rules. If one would remove the permutation rules, one would often not obtain the best possible result. For example, in the last formula of the example, $(p(x) \wedge q(y)) \wedge r(x)$ first has to be permuted into $(p(x) \wedge r(x)) \wedge q(y)$, before the rule $\exists x(P \wedge Q(x)) \Rightarrow P \wedge \exists x\ Q(x)$ can be applied.

Despite the fact that the decision making is more complicated than was the case for the NNF, Algorithm 2 can be still modified for anti-prenexing, because the decision making plays no role in the proof generation. For the proof generation, only correctness of the rules matters, and all rules can be easily proven correct.

7 Proof Reductions

Proof reductions are important, because they make it possible to obtain modularity and flexibility. For a detailed motivation, we refer to [1]. There, a special calculus called *replacement calculus* was introduced which allows for certain reductions that remove repeated building up of the same context in a proof. In sequent calculus, the standard reductions of cut elimination correspond to the reductions of the replacement calculus, so there is no need anymore for the replacement calculus. For the purpose of proof simplification, one should implement all standard reductions of cut elimination (see [3]), except for the permutation of a cut with a contraction, because this permutation is the cause of increasement in proof length.

The proof reductions are needed in order to combine the repeated building up of contexts. Suppose that one has a big formula of form $F[A_1]$, that A_1 is first rewritten into A_2, and after that into A_3. Algorithm 2 lifts a proof of $A_1 \vdash A_2$ to a proof of $F[A_1] \vdash F[A_2]$. After that, it lifts a proof of $A_2 \vdash A_3$ to a proof of $F[A_2] \vdash F[A_3]$, which is then combined, using cut, into a proof of $F[A_1] \vdash F[A_3]$.

However, it would be more efficient to first apply cut on $A_1 \vdash A_2$ and $A_2 \vdash A_3$, resulting in $A_1 \vdash A_3$, and lift this proof to $F[A_1] \vdash F[A_3]$.

Combination of context lifting can be done only if one knows in advance the order in which the replacements will be made, and when they are near to each other. This was the case for the NNF-transformation, and Algorithm 2 makes use of this fact, both for the outside-inside strategy, and for the inside-outside strategy.

If one does not know the order of replacements in advance, then Algorithm 2 will not avoid repeated lifting into the same context. This would be the case for anti-prenexing. In that case, one has to rely on proof reductions. Using the standard reductions of cut elimination, the cut on the top level can be permuted with the rules that build up the context, until it either disappears, or reaches a contraction.

Using proof terms, the reductions can be easily implemented by a rewrite system on proof terms. We give a few examples of the reductions involved, and give the corresponding rewrite rules:

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash \Delta,\ \beta_1{:}A \qquad \Gamma \vdash \Delta,\ \beta_2{:}B}{\Gamma \vdash \Delta,\ \beta{:}A \wedge B} \qquad \dfrac{\Gamma,\ \alpha_1{:}A,\ \alpha_2{:}B \vdash \Delta}{\Gamma,\ \alpha{:}A \wedge B \vdash \Delta}}{\Gamma \vdash \Delta}}$$

is replaced by

$$\dfrac{\dfrac{\Gamma \vdash \Delta,\ \beta_2{:}B \qquad \dfrac{\Gamma \vdash \Delta,\ \beta_1{:}A \qquad \Gamma,\ \alpha_1{:}A,\ \alpha_2{:}B \vdash \Delta}{\Gamma,\ \alpha_2{:}B \vdash \Delta}}{\Gamma \vdash \Delta.}}$$

The corresponding rewrite rule is

$$\mathrm{cut}(A \wedge B, \mathrm{andleft}(\alpha, \pi, \alpha_1, \alpha_2), \alpha, \mathrm{andright}(\beta, \pi_1, \beta_1, \pi_2, \beta_2), \beta) \Rightarrow$$
$$\mathrm{cut}(B, \mathrm{cut}(A, \pi, \alpha_1, \pi_1, \beta_1), \alpha_2, \pi_2, \beta_2).$$

The following proof fragment

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash \Delta,\ \beta_1{:}P[x := y]}{\Gamma \vdash \Delta,\ \beta{:}\forall x\ P(x)} \qquad \dfrac{\Gamma,\ \alpha_1{:}P[x := t] \vdash \Delta}{\Gamma,\ \alpha{:}\forall x\ P(x) \vdash \Delta}}{\Gamma \vdash \Delta}}$$

reduces into

$$\frac{\Gamma \vdash \Delta,\ \beta_1{:}P[x := t] \qquad\qquad \Gamma,\ \alpha_1{:}P[x := t] \vdash \Delta}{\Gamma \vdash \Delta}$$

The corresponding rewrite rule is

$$\mathrm{cut}(\forall x\ P(x), \mathrm{forallleft}(\alpha, \pi_2, \alpha_1, t), \alpha, \mathrm{forallright}(\beta, \pi_1, \beta_1, y), \beta) \Rightarrow$$

$$\mathrm{cut}(P[x := t], \pi_2, \alpha_1, \pi_1[y := t], \beta_1).$$

8 Conclusions

We have shown that implementing the CNF-transformation with proof generation is possible. We have given a data structure (inspired by [5]) for the represention of sequent calculus proofs, which is concise, and which allows for implementation of proof reductions. We have given a general translation algorithm, based on rewriting, that covers nearly all of the transformations involved.

Proof generation will not be feasible for formulas that are propositionally complex. Such formulas will have exponentially large proofs, (because probably $\mathcal{NP} \neq \text{co-}\mathcal{NP}$.)

References

1. Hans de Nivelle. Extraction of proofs from the clausal normal form transformation. In Julian Bradfield, editor, *Proceedings of the 16th International Workshop on Computer Science Logic (CSL 2002)*, volume 2471 of *Lecture Notes in Artificial Intelligence*, pages 584–598, Edinburgh, Scotland, UK, September 2002. Springer.
2. Hans de Nivelle. Translation of resolution proofs into short first-order proofs without choice axioms. In Franz Baader, editor, *Proceedings of the 19th International Conference on Computer Aided Deduction (CADE 19)*, volume 2741 of *Lecture Notes in Artificial Intelligence*, pages 365–379, Miami, USA, July 2003. Springer Verlag.
3. Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1989.
4. Andreas Nonnengart and Christoph Weidenbach. Computing small clause normal forms. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 6, pages 335–367. Elsevier Science B.V., 2001.
5. Frank Pfenning. Structural cut elimination. In Dexter Kozen, editor, *Proceedings 10th Annual IEEE Symposion on Logic in Computer Science*, pages 156–166. IEEE Computer Society Press, 1995.

Author Index