
Calulemus 2009

The 16th Symposium on the Integration of
Symbolic Computation and Mechanised Reasoning



Calulemus 2009 Emerging Trends

Editors:

Lucas Dixon, Jacques Carette

6-7 July 2009, Joint with CICM'09, Ontario, Canada

Table of Contents

Reecting Data: Formally Correct Results for Efficient (and Dirty) Algorithms	3
<i>L. Dixon</i>	
The Naproche System	8
<i>D. Kühlwein, M. Cramer, P. Koepke, and B. Schröder</i>	
Reusing Proofs in a Mathematical Library	19
<i>I. Noyer, R. Rioboo</i>	
Computing Ranking and Unranking Functions for BDDs	31
<i>Paul Tarau, Brenda Luderman</i>	

Reflecting Data: Formally Correct Results for Efficient (and Dirty) Algorithms

Lucas Dixon

University of Edinburgh
Informatics Forum, 10 Crichton Street, Edinburgh EH8 9AB, UK.
l.dixon@ed.ac.uk
<http://dream.inf.ed.ac.uk/>

Abstract. We describe an approach to writing efficient algorithms in fully formal proof systems by reflecting the data the algorithm uses, but not the algorithm itself. We illustrate the approach with an efficient algorithm for normalisation of arithmetic terms. Our approach ensures the correctness of the algorithm's result without extending the trusted code of the proof system and without having to prove any properties about the algorithm itself, such as its termination. This approach allows correct results to be ensured even for 'dirty' algorithms - such as those which lack a proof of termination.

1 Introduction

This paper presents an approach to implementing efficient algorithms that manipulate mathematical objects within a fully formal system. A common view is that fully formal tactic-based systems are not sufficiently extensible to implement efficient computer-algebra algorithms. The aim of this paper is to show that, contrary to this common belief, it is possible to provide efficient implementations with results that have been proved to be correct. We believe this provides a strong argument that fully-formal proof systems can enjoy efficient and correct computer-algebra algorithms.

We consider fully-formal proof systems to be those with a fixed and trusted logical kernel of inference rules. This is the so called LCF-style, also referred to as tactic-based theorem proving. Examples include systems such as Isabelle [9] and HOL [4]. Extensions to the system are made by writing tactics which combine the primitive inference rules with previously derived theorems to derive new theorems. This ensures that extensions do not endanger the system's correctness. Following this methodology, our approach avoids introducing any additional code that needs to be trusted. The key trick we employ is to observe that, to verify the result, only the representations used by an algorithm need to be characterised formally. In contrast to approaches based on computational reflection or program extraction, such as those used by MetaPr1 [8], Coq [2], Isabelle [1], we avoid the need to represent the algorithm in the proof system. Thus we avoid needing to prove properties about the algorithm, such as its termination. Furthermore,

we avoid needing extend the trusted code by to include the code-extraction machinery. Instead, the algorithm can be written outside the logical kernel as a tactic that justifies its steps using a representation within the proof system’s object logic.

We note that while you do less, you also get less: within Coq and Isabelle, you also prove that your algorithm terminates and that it covers all cases. Our approach lets the algorithm behave badly, but ensures that the results are still right. In the rest of this paper, we introduce an example problem, that of term normalisation, outline other approaches, describe an efficient algorithm, and then describe our approach, illustrated by how it applies to term normalisation. Finally we conclude and remark on further work.

2 An Example Problem: Term Normalisation

This work was initially motivated by an example problem posed by Bruno Buchberger during a visit to Edinburgh in June 2005. He suggested that, without extending the logical kernel of an LCF-style proof system, an efficient normalisation algorithm for Peano arithmetic terms involving zero, addition and successor cannot be defined. In particular, given a term such as “ $a + (Suc (0 + b) + (c + (Suc d)))$ ”, an efficient normalisation algorithm can be written by traversing the term tree once. The idea is that the traversal builds a list of the non-zero elements and counts the number of successors. The result is a normalised representation of a term with the successors on the outside, i.e. “ $(2, [a, b, c, d]) \equiv Suc(Suc(a + b + c + d))$ ”. The efficiency of such an algorithm is linear with respect to the size of the term.

It is clear that the same level of efficiency is not achieved by a naive algorithm based on rewriting the term. The naive rewriting based approach would be to use the rewrite rules:

$$\begin{aligned} a + (Suc\ b) &\Rightarrow Suc(a + b) \\ (Suc\ a) + b &\Rightarrow Suc(a + b) \\ 0 + b &\Rightarrow b \\ b + 0 &\Rightarrow b \end{aligned}$$

and normalisation is achieved by exhaustive rewriting. However, such an algorithm will move successors out one step at a time. The overall runtime will be asymptotically equal to the number of successors multiplied by the size of the term, multiplied by the time taken to find the rule’s left hand side in the term. This is polynomial in the term size, significantly worse than the linear time for a single term traversal.

The question is how can we implement the efficient algorithm in a fully formal proof system? Before we present our approach, we briefly discuss other approaches.

2.1 Related Work

Verification of Oracles

For problems where the result can be verified quickly, but the computation of that result is expensive, it is possible to perform the computation outside the prover and simply verify the result within the prover [6]. Examples that use this technique include factorisation and integration. However, this approach is only useful for problems where the verification of the result is significantly quicker than the computation. For many problems, including our problem of normalising arithmetic terms, it is not clear how the correct result can be used as an oracle without re-performing the full computation. The approach we suggest applies more generally than the verification of oracles.

Formalised Algorithms and Computational Reflection

Perhaps the most obvious way to get an efficient computer algebra algorithm within a formal proof system is to write it within the logic of the proof system. Then algorithm can be applied by simply evaluating it within the proof system. Doing this naively will have a significant linear slowdown. However, efficient evaluation techniques has been demonstrated in systems such as ACL2 [5].

The main drawback of this approach is that formalising algorithms is a lot of work. The user is required to prove properties about the algorithm, such as its termination, even though they may only be interested in the correctness of the result. Furthermore, certain algorithms cannot even be defined as functions within the proof system. In particular, functions within a proof system cannot recurse over the system's abstract syntax.

The way to overcome the limits of recursion on the system's syntax is to 'reflect' the syntax as object inside the prover. This involves creating an equivalent representation of the system's syntax within the system itself. This can be a lot of work, but once this is done, the algorithm can again be expressed as a function on the reflected representation. However, to apply the algorithm within the proof system requires extracting the algorithm from the formalisation so that it can be applied to the system's original syntax. Such an approach has been developed for the MetaPrl system [8]. Other techniques that extract efficient functional programs from formalised algorithms have been demonstrated in Coq [2, 7] and Isabelle [1]. In terms of the trusted code, extraction and reflection-based approaches, require significantly more code trusted.

The main difference with our approach is that we avoid needing to formalise and prove properties of the algorithm, while preserving the correctness of its results. We also avoid the need to extend the trusted code base.

3 The Efficient Algorithm

Before we describe our suggested approach, we clarify the introduced example by detailing the efficient algorithm for normalisation of arithmetic terms. We will

present the efficient algorithm over an abstract syntax. For clarity and brevity we will hide the types. This leaves us with the following datatype for terms:

```
T = App of T * T
  | Abs of T
  | Bound of int
  | Const of string
```

Using this syntax, the term $a + (Suc\ b)$ is represented by `(App (App (Const 'a') (Const 'b')) (App (Const 'Suc') (Const '1')))`. However, because the abstract syntax is rather unreadable, we will write terms within \llbracket and \rrbracket , and let ourselves use a natural syntax with infix operators. With this notation, the main part of our efficient normalisation algorithm can be written as:

```
intern  $\llbracket x + y \rrbracket$  (s,l) = intern  $\llbracket y \rrbracket$  (intern  $\llbracket x \rrbracket$  (s,l))
intern  $\llbracket Suc\ x \rrbracket$  (s,l) = intern  $\llbracket x \rrbracket$  (Suc 1, l)
intern  $\llbracket 0 \rrbracket$  (s,l) = (s,l)
intern  $\llbracket n \rrbracket$  (s,l) = (s,n::l)
```

where the infix function `::` is the cons operator for lists. The last case is a catch-all for any other constants, variables or subterms that are outside the scope of the normalisation procedure. The `intern` function computes the number of successors in the term and creates a list of all non-normalisable, non-zero leaf nodes. Observe that this function cannot be expressed within the object logic of a proof-system. This is because the function is defined recursively on the object-logic's syntax. Furthermore, it evaluates to different results for the terms $\llbracket 0 \cdot a \rrbracket$, where \cdot represents multiplication, and $\llbracket 0 \rrbracket$, even though they are semantically the same within the proof system ¹.

Once the intermediate representation is calculated, we extract the normalised expression for it. This is done by evaluating the following function:

```
extract (a, l) = foldr (op  $\llbracket + \rrbracket$ )  $\llbracket a \rrbracket$  l
```

Unlike `intern`, this can be expressed within a fully formal proof system.

The full normalisation function is then simply the combination of the extraction with the computation of the intermediate representation:

```
normalise x = extract (intern x)
```

By using a list to represent the non-normalisable subterms of the expression, we normalise away addition's associative structure. If we wanted to keep the original structure of the term, then our intermediate representation would simply be a tree rather than a list.

¹ In general, this issue occurs whenever the representation function is not injective to the original representation; whenever the efficient representation removes symmetries in the original.

4 Reflecting Algorithmic Data

Our approach is to implement the efficient version of the algorithm as a tactic. Unlike functions in the object logic, a tactic can recurse on the structure of terms. To ensure correctness of the result, the tactic maintains a representation of the value being computed within the proof system. Steps that transform the representation require derived theorems to justify their correctness. However, the algorithm as a whole remains outside the proof system and thus avoids needing a proof of termination and totality. We illustrate the idea by implementing the efficient term normalisation.

Returning to the efficient normalisation algorithm, its intermediate internal representation is defined in the proof system. This is simply a pair of a natural number - the successor count - with a list of un-normalisable terms: the pair type $(\mathbf{nat} * (\mathbf{nat} \text{ list}))$. We will call this type **E**.

Given the intermediate representation within the proof system, we then prove its relationship to the original representation. For our example, we define the relationship between objects of type **E** and those of type **nat**. This is based on two functions, firstly, the function *fin* which creates an **E** from a **nat**:

$$\mathit{fin} \ 0 \ (s, l) = (s, l) \tag{1}$$

$$\mathit{fin} \ (\mathit{Suc} \ x) \ (s, l) = \mathit{fin} \ x \ (\mathit{Suc} \ s, l) \tag{2}$$

This function takes an extra accumulator argument. This is to characterise the behaviour of the **intern** function. If **intern** was not recursive then *fin* would not need the accumulator argument.

The second function we need is a fully formal version of the **extract** function, which we will call *fec*, which creates a **nat** from an **E**:

$$\mathit{fec} \ (a, l) = \mathit{foldr} \ (\mathit{op} \ +) \ a \ l \tag{3}$$

In the proof system we then derive theorems, based on these functions, to justify the cases that make up the main part of the efficient algorithm (the **intern** function):

$$\mathit{fec} \ (\mathit{fin} \ (x + y) \ (s, l)) = \mathit{fec} \ (\mathit{fin} \ y \ (\mathit{fin} \ x \ (s, l))) \tag{4}$$

$$\mathit{fec} \ (\mathit{fin} \ (\mathit{Suc} \ x) \ (s, l)) = \mathit{fec} \ (\mathit{fin} \ x \ (\mathit{Suc} \ s, l)) \tag{5}$$

$$\mathit{fec} \ (\mathit{fin} \ 0 \ (s, l)) = \mathit{fec} \ (s, l) \tag{6}$$

$$\mathit{fec} \ (\mathit{fin} \ x \ (s, l)) = \mathit{fec} \ (s, x :: l) \tag{7}$$

These exactly characterise the cases of **intern** under the **extract** function. We will use these to justify the steps taken by the efficient normalisation algorithm. Although all these theorems can be proved automatically by IsaPlanner [3], we anticipate that for more complex representations, it will be more difficult.

In addition to these properties, in order to let us introduce normalisation, we also derive:

$$fex (fn x (0, [])) = x \tag{8}$$

Using these we can now write the efficient normalisation procedure as a tactic. The tactic traverses a term of interest and applies (8) from right to left to initialise the normalisation procedure on a subterm. The subterm matching x is then traversed and the rules (4) - (7) are applied by the tactic using pattern matching. This mirrors exactly the **intern** part of the efficient algorithm. Once this is completed, the definition of fex is unfolded to apply the **extract** function, completing the normalisation. Because each step is justified by an equation, the final theorem has been derived fully formally.

It is then interesting to ask how this could have gone wrong. Bugs in the algorithm's implementation cannot yield a false result, however, a buggy algorithm may not terminate. Another way the algorithm can be buggy is that it does not complete the normalisation procedure. In these cases extra constants such as fex and fn will be left within the term.

5 Conclusion

The approach we have presented provides a fully formalised proof of the result of an algorithm without requiring any proofs about the algorithm's correctness. Unlike reflection techniques, properties such as totality and termination do not need to be proved and no extra trusted code is needed. In comparison with verified oracles, our approach is more general. Our approach still requires more work than just trusting novel code. In particular, we still require some proofs about the representation to justify the algorithms steps. However, for this extra work, we ensure the soundness of the results. Thus our approach falls somewhere between verification of oracles and the reflection of algorithms. We have shown that correct results can be ensured independently of the correctness of the algorithm used to derive them.

We suggest that whatever representation is used, it can be modelled in a formal proof system, and thus our approach can be used. For this example, datatypes suffice. Working with pointers is more complex, but also has an analogue within a proof system as a function from natural numbers (the pointer) to the value pointed to. Dealing with the details of more complex representations is left as further work.

To see if this provides a feasible level of efficiency for more complex algorithms is further work. However, given that this incurs only a linear slowdown we expect that it will be sufficient for many applications. Extending our approach appropriately for algorithms which use pointer-based data-structures is another area further work.

References

1. U. Berger, S. Berghofer, P. Letouzey, and H. Schwichtenberg. Program extraction from normalization proofs. *Studia Logica*, 82(1):25–49, 2006.
2. Y. Bertot and P. Casteran. *Interactive Theorem Proving and Program Development*. SpringerVerlag, 2004.
3. L. Dixon. *A Proof Planning Framework for Isabelle*. PhD thesis, University of Edinburgh, 2006.
4. M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.
5. D. A. Greve, M. Kaufmann, P. Manolios, J S. Moore, S. Ray, J. L. Ruiz-Reina, R. Sumners, D. Vroon, and M. Wilding. Efficient Execution in an Automated Reasoning Environment. *Journal of Functional Programming*, 18(1):15–46, January 2007.
6. J. Harrison and L. Théry. A skeptic’s approach to combining HOL and Maple. *Journal of Automated Reasoning*, 21:279–294, 1998.
7. D. Hendriks. *Metamathematics in Coq*. PhD thesis, Department of Philosophy, Utrecht University, 2003.
8. J. Hickey, A. Nogin, X. Yu, and A. Kopylov. Practical reflection for sequent logics. *Electron. Notes Theor. Comput. Sci.*, 174(5):79–94, 2007.
9. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

The Naproche System

D. Kühlwein, M. Cramer, P. Koepke, and B. Schröder

Mathematical Institute, University of Bonn
German Linguistics, University of Duisburg-Essen
{cramer, koepke, kuehlwei}@math.uni-bonn.de
bernhard.schroeder@uni-due.de
<http://www.naproche.net>

Abstract. The Naproche project (Natural language Proof Checking) was initiated by BERNHARD SCHRÖDER and PETER KOEPKE at the University of Bonn to focus on an interdisciplinary study of the semi-formal language of mathematics. A central goal of Naproche is to develop a controlled natural language (CNL) for mathematical texts and adapted proof checking software which checks texts written in the CNL for syntactical and mathematical correctness. The project is still at a prototypical stage, further information is available at www.naproche.net.

This paper describes the Naproche system, an implementation of the ideas developed by the Naproche project. The Naproche system accepts L^AT_EX-style texts, consisting of mathematical formulas imbedded in a controlled natural language. Texts written in the controlled natural language are parsed using techniques from computational linguistics and transformed into first-order formulas. The formulas are given to an automatic theorem prover which checks whether each formula of an argument is a logical consequence of the preceding formulas or axioms.

Key words: Controlled natural language, formal mathematics, discourse representation theory, automated theorem proving

1 Introduction

Considering that mathematics has a reputation of being an exact science, it is interesting to note that one of the main concepts of mathematics, the mathematical proof, is somewhat vaguely defined. What exactly is a mathematical proof? Firstly, one can distinguish between two kinds of mathematical proofs: Formal and informal proofs.

Formal proofs are finite derivations in a calculus. They are sequences of mathematical symbols and do not contain natural language elements. Given a calculus, there is a definite answer whether or not a text is a formal proof.

An informal proof is a mixture of natural language and mathematical symbols. Most proofs in mathematical textbooks and journals are informal. Contrary to formal proofs, there is no clear criteria whether a text constitutes an informal proof or not.

Ever since ALFRED WHITEHEAD's and BERTRAND RUSSELLS's work *Principia Mathematica* [16] most mathematicians agree that it would be possible, even if extremely tedious, to find a formal proof for every theorem. With this in mind, one could interpret informal proofs as abbreviations for formal proofs.

Facilitating methods from (computational) linguistics, computer mathematics and mathematics, the Naproche project (NATURAL language PROOF CHECKING), a joint initiative of PETER KOEPKE (Mathematics, University of Bonn) and BERNHARD SCHRÖDER (Linguistics, University of Duisburg-Essen), studies the interplay between formal and informal proofs.

1.1 The Naproche System

As part of the Naproche project, we develop the Naproche system, a program which aims to automatically translate informal proofs into their formal counterparts.

Of course, one first has to define what exactly a translation of an informal proof is. Our approach is to see an informal proof as a blueprint for a formal proof. The major stepstones in the derivation are given, and all that is left to do is to flesh out each single step. To do this, we first transform the informal proof into a sequence of first order logic statements. Once an input text is translated, we use automated theorem provers (ATPs) to fill out the gaps. Proofs created by ATPs are basically derivations in a calculus, and therefore this gives us a method of creating a formal from an informal proof.

Given that this translation from natural language into first order logic is sound, this procedure also gives a method of checking the correctness of informal proofs.

There are several challenges which one has to face when trying to implement such a program. Firstly, one has to teach the computer to automatically interpret statements in natural language correctly. The fact that natural language is often very ambiguous and that there are usually several different ways of stating the same fact only adds to the difficulty. Another problem is the definition of the translation algorithm from informal proofs into first order logic. And finally, one has to find a way to extend the first order translation of a text to a full formal proof.

In order to handle the ambiguity of natural language texts, we developed a controlled natural language (CNL) for mathematics, the Naproche CNL, which has a clearly defined translation from texts written in this language into first order formulas. We use an adapted version of Discourse Representation Structures (DRS), which we call Proof Representation Structures (PRS), to extract the first order representation of a Naproche CNL text. To fill the gaps in the proof, we implemented a checking algorithm for PRS that creates the appropriate proof obligations for the ATP.

We will present each part of the Naproche system in more detail.

1.2 Related Work

While, to the authors knowledge, there is no other group that focuses on the connection between informal and formal proofs, there are already several proof checking systems that also emphasize the readability of their respective input language.

A. TRYBULEC’s Mizar [7] is arguably the most prominent. It was started in 1973, and by today several non-trivial mathematical theorems have been proved (e.g GÖDEL’s completeness theorem [2]). An active community continues to formulate and prove theorems in Mizar. The results are published regularly in the journal *Formalized Mathematics*.

The Isabelle [8] team is working on Isar [15], a “human-readable structured proof language”. The System for Automated Deduction (SAD, [12]) checks texts that are written in its input language, ForThel [13], for correctness.

We are collaborating with the VeriMathDoc project [1], which includes the PLATO program [14]. Their goal is to develop a mathematical assistant system that naturally supports mathematicians in the development, authoring and publication of mathematical work.

2 The Architecture of the Naproche System

The Naproche system consists of three main modules: The User Interface, the Linguistics module and the Logic module. Figure 1 shows an overview of the architecture of the Naproche system.

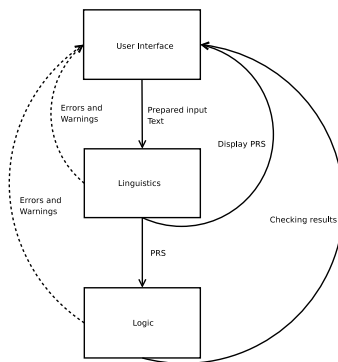


Fig. 1. The architecture of the Naproche system.

The User Interface is the standard communication gateway between the user and the Naproche system. The input text is given to the User Interface, and the other modules report back to the User Interface. The module transforms

the input text into a prolog list and hands it over to the Linguistics module. Currently, our User Interface is web-based and can be found on the Naproche website, *www.naproche.net*. We are also working on a plugin for the WYSIWYG editor $\text{\TeX}_{\text{MACS}}$ [11]. The input language is described in section 3.

The Linguistics module creates a PRS from the input text. Any errors or warnings which occur during the creation are reported back to the User Interface. The created PRS can either be given to the Logic module, or reported back to the User Interface. Section 4 explains PRSs in greater detail.

With the help of ATPs, the Logic module checks PRSs for logical correctness. As in the Linguistics module, errors or warnings which occur during the creation are reported back to the User Interface. The result of the check is sent to the User Interface. A brief overview of the checking algorithm is given in section 5.

3 The Naproche CNL

Natural language is usually full of ambiguities and heavily context dependent. This makes extracting the semantics of an arbitrary text written in common English a very hard, if not impossible, task. Controlled natural languages are subsets of natural languages, which usually try to reduce complexity and eliminate ambiguity while preserving the the relevant parts of the expressiveness of the original language. They give the user the means to 'easily' process and work with text written in such a language.

Attempto Controlled English (ACE) [3] showed that this concept can be very successful. Texts written in ACE read like normal English while having a unique first order representation which can be used for further tasks, e.g. reasoning or queries over the text.

The Naproche CNL aims to be the mathematical equivalent of ACE, i.e. to be as intuitive and expressive as the semi-formal language of mathematics as used in textbooks and journals, while defining an unambiguous translation into first order logic.

In the current version of the Naproche CNL, a text is structured by structure markers: *Axiom*, *Theorem*, *Lemma*, *Proof* and *Qed*. For example, a theorem is presented after the structure marker *Theorem*, and its proof follows between the structure markers *Proof* and *Qed*.

First order formulas can be combined with natural language expressions to form the usual mathematical constructs like statements, definitions, implications and assumptions.

Assumptions are always opened by an assumption trigger (e.g. *let*, *consider*, *assume that ...*), and closed by a sentence starting with *thus* or by a *Qed*. Definitions always start with *define*. Assertions are made by a first order formula or a naturally predicated term (e.g. *x is an ordinal*), optionally preceeded a statement trigger (e.g. *then*, *hence*, *therefore*, *so*, ...). Additionally, negation, conjunction, disjunction, quantification and implication may be expressed in natural language.

Since most mathematicians use L^AT_EX for writing papers, we tried to keep the Naproche CNL very close to original L^AT_EX code.

As an example, we present a short proof written in the Naproche CNL.

```
Define $Trans(x)$ if and only if $\forall u, v (u \in v \wedge v \in x) \implies u \in x$.
Define $Ord(x)$ if and only if $Trans(x) \wedge \forall y (y \in x \implies Trans(y))$.
```

Theorem.

For all x, y , if $x \in y$ and $Ord(y)$ then $Ord(x)$.

Proof.

Suppose $x \in y$ and $Ord(y)$. Then $Trans(x)$.

Assume that $u \in x$. Then $u \in y$.

Hence $Trans(u)$. Thus $Ord(x)$.

Qed.

Note that formal quantification and implication is used in the definitions, whereas the statement of the theorem is written using natural language quantification and implication.

Substituting `\and` with `\wedge` and `\implies` with `\rightarrow` gives L^AT_EX compatible text:

```
Define Trans(x) if and only if  $\forall u, v (u \in v \wedge v \in x) \rightarrow u \in x$ .
Define Ord(x) if and only if  $Trans(x) \wedge \forall y (y \in x \rightarrow Trans(y))$ .
```

Theorem.

For all x, y , if $x \in y$ and $Ord(y)$ then $Ord(x)$.

Proof.

Suppose $x \in y$ and $Ord(y)$. Then $Trans(x)$. Assume that $u \in x$. Then

$u \in y$. Hence $Trans(u)$. Thus $Ord(x)$.

Qed.

4 Proof Representation Structures

While extracting the semantics of a CNL is definitely easier than extracting the semantics of a text in unrestricted natural language, it is still no trivial task. The more sophisticated the CNL gets, the more advanced approaches need to be used.

Computational linguistics has developed several techniques to automatically extract the semantics of a natural language text. We adapted the Discourse Representation Structures (DRS, [4]) from Discourse Representation Theory, which are also used in the *Attempto* project [3], for our needs, and called the

result *Proof Representation Structures*¹ (PRSS, see [5], [6]). PRSS are Discourse Representation Structures that are enriched in such a way as to represent the distinguishing characteristics of the semi-formal language of mathematics.

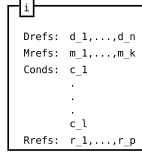


Fig. 2. A PRS with identification number i , discourse referents d_1, \dots, d_n , mathematical referents m_1, \dots, m_k , conditions c_1, \dots, c_l and textual referents r_1, \dots, r_p

A PRS has five constituents: An identification number, a list of discourse referents, a list of mathematical referents, a list of textual referents and an ordered list of conditions. Similar to DRSS, we can display PRSS as “boxes” (See fig. 2).

Mathematical referents are the terms and formulas which appear in the text. As in DRSS, discourse referents are used to identify objects in the domain of the discourse. However, the domain contains two kinds of objects: mathematical objects like numbers or sets, and the symbols and formulas which are used to refer to or make claims about mathematical objects. Discourse referents can identify objects of either kind.

PRSS have identification numbers, so that they can be referred to at some later point. The textual referents indicate the intratextual and intertextual references.

Just as in the case of DRSS, PRSS and PRS conditions have to be defined recursively: Let A, B be PRSS, f be a function symbol, X, X_1, \dots, X_n discourse referents, Y a mathematical referent, and let *Word* denote an English noun, adjective or verb. Then

- $holds(X)$ is a condition representing the claim that the formula referenced by X is true.
- $math_id(X, Y)$ is a condition which binds a discourse referent to a mathematical referent (a formula or a term).
- A is a condition.
- $\neg A$ is a condition, representing a negation.
- $A := B$ is a condition, representing a definition.
- $A \Rightarrow B$ is a condition, representing an implication or universal quantification.
- $A \Leftrightarrow B$ is a condition, representing an equivalence statement.

¹ Note that even though CLAUS ZINN [17] also uses the term *Proof Representation Structure*, the definitions of Naproche and ZINN are different.

- $A \leq B$ is a condition, representing a B if A statement.
- $A \vee B$ is a condition, representing a disjunction.
- $A \implies B$ is a condition, representing an assumption.
- $f :: A \Rightarrow B$ is a condition, representing a function definition.
- *contradiction* is a condition, representing a contradiction.
- $\text{predicate}(X, \text{Word})$ is a condition, representing a natural language statement with one argument.
- $\text{predicate}(X_1, X_2, \text{Word})$ is a condition, representing a natural language statement with two arguments.

Note that contrary to the case of DRSs, a bare PRS can be a direct condition of a PRS. This allows to represent the nested structure of mathematical texts in a PRS: The different building blocks of a text (axioms, definitions, lemmas, theorems, proofs), that are denoted by structure markers, correspond to sub-PRSs (See fig. 3 for an example). The hierarchical structure of assumptions is represented by nesting conditions of the form $A \implies B$: A contains an assumption, and B contains the representation of all claims made inside the scope of that assumption.

The algorithm creating PRSs from CNL proceeds sequentially: It starts with the empty PRS. Each sentence or structure marker in the discourse updates the PRS according to an algorithm similar to a standard DRS construction algorithm but taking the nesting of assumptions into account. [5]

The PRS constructed from the example proof is shown in figure 3.

5 Checking PRS

The checking algorithms keeps a list of first order formulas it considers to be true, called premises, which gets continuously updated during the checking process.

To check a PRS, the algorithms considers the conditions of the PRS. The conditions are checked sequentially and each condition is checked under the currently active premises. According to the kind of condition, the Naproche system creates obligations which have to be discharged by an automated theorem prover. If all obligations of a condition can be discharged, then the condition is proved to be valid. After a conditions is deemed valid, the active premises get updated, and the next condition gets checked. A PRS is accepted by Naproche if all its conditions are valid.

In our example PRS (Fig. 3), the first condition is a definition. As this is the first condition of the PRS, the sequence of currently active premises is empty. Since definitions do not have to be checked, all we have to do is store the first order representation of this definition as a premise for future use. So after this condition is processed, the sequence of currently active premises contains the formula $\forall x \text{Trans}(x) \leftrightarrow (\forall u, v(u \in v \wedge v \in x) \rightarrow u \in x)$.

The second conditions is also a definition and therefore gets treated the same way. The sequence of active premises after this condition is

$$\begin{aligned} &[\forall x \text{Trans}(x) \leftrightarrow (\forall u, v(u \in v \wedge v \in x) \rightarrow u \in x), \\ &\forall x \text{Ord}(x) \leftrightarrow (\text{Trans}(x) \wedge \forall y(y \in x \rightarrow \text{Trans}(y)))] \end{aligned}$$

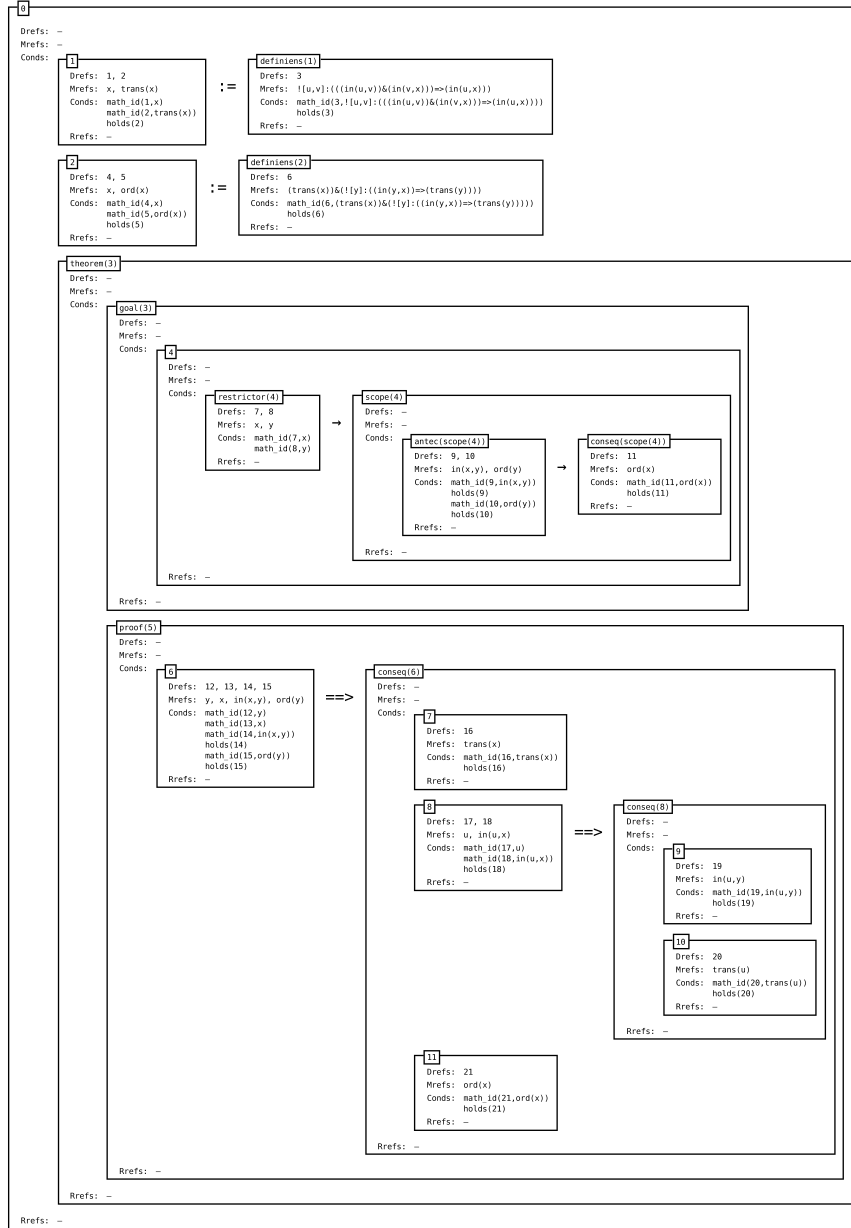


Fig. 3. The PRS for the proof of: For all x, y , if $x \in y$ and $\text{Ord}(y)$ then $\text{Ord}(x)$.

The next condition in our example PRS is a theorem condition. A theorem condition is a PRS with two sub-PRS, the goal PRS and the proof PRS. The statement of the theorem is stored in the goal PRS, the proof for the theorem is in the proof PRS. The algorithm first checks the proof PRS and then uses the updated premises to check the goal PRS.

The first condition in the proof PRS is an assumption. Again, all we have to do is to update the list of active premises. It now contains the following formulas:

$$\begin{aligned} & [\forall x \text{Trans}(x) \leftrightarrow (\forall u, v (u \in v \wedge v \in x) \rightarrow u \in x), \\ & \forall x \text{Ord}(x) \leftrightarrow (\text{Trans}(x) \wedge \forall y (y \in x \rightarrow \text{Trans}(y))), \\ & x \in y, \text{Ord}(y)] \end{aligned}$$

Next comes the statement $\text{Trans}(x)$. The algorithm tries to prove this statement from the active premises. For this, the premises as well as the statement are transformed into the TPTP first order format [9] and an ATP query is created:

```
fof(1, axiom, ![Vx]:((trans(Vx))<=>
  (![Vu,Vv]:(((in(Vu,Vv))&(in(Vv,Vx))=>(in(Vu,Vx)))))).
fof(2, axiom, ![Vx]:((ord(Vx))<=>
  ((trans(Vx))&(![Vy]:((in(Vy,Vx))=>(trans(Vy)))))).
fof(3, axiom, in(vx,vy)).
fof(4, axiom, ord(vy)).
fof(1, conjecture, trans(vx)).
```

GEOFF SUTCLIFFE's service tools for the TPTP library (e.g. SystemsOnTPTP [10]) are used to interact with the ATP. The result is given as an output to the user, $\text{Trans}(x)$ gets added to the active premises and the checking algorithm proceeds to the next condition.

The remaining conditions are checked in a similar fashion. The detailed procedure as well as considerations about the completeness and the correctness of the algorithm can be found in [6].

6 Results

The Naproche system parses the Naproche CNL, creates the appropriate PRSs, and checks them for correctness using automated theorem provers. The BURALI-FORTI paradox, a well known mathematical theorem, as well as several basic statements in group theory, were formulated in the Naproche CNL, and checked for correctness with the Naproche system. A web-based interface to the Naproche system can be found on our homepage www.naproche.net.

7 Future Work

There are two major points that we would like to improve in upcoming versions of the Naproche system:

Firstly, we will extend the Naproche controlled natural language to include a rich mathematical formula language and many argumentative mathematical phrases and constructs. And secondly, we shall improve the Naproche-ATP interaction. We want to study which of the premises were actually used when discharging a proof obligation and whether the systems can be tailored to do inferences in the 'size' of human proofsteps.

Once the Naproche system is sufficiently extensive and powerful, proofs from various areas of mathematics can be reformulated and checked in a way understandable to men and machines.

References

1. VeriMathDoc, URL <http://www.ags.uni-sb.de/~afiedler/verimathdoc/>.
2. Patrick Braselmann and Peter Koepke. Gödel's Completeness Theorem. *Formalized Mathematics*, 13, 2005.
3. Norbert E. Fuchs, Stefan Höfler, Kaarel Kaljurand, Fabio Rinaldi, and Gerold Schneider. Attempto Controlled English: A Knowledge Representation Language Readable by Humans and Machines, 2005.
4. H. Kamp and U. Reyle. *From Discourse to Logic: Introduction to Model-theoretic Semantics of Natural Language*. Kluwer Academic Publisher, 1993.
5. Nickolay Kolev. Generating Proof Representation Structures for the Project NAPROCHE. Master's thesis, University of Bonn, 2008.
6. Daniel Kuehlwein. A Calculus for Proof Representation Structures. Master's thesis, University of Bonn, 2008.
7. Roman Matuszewski and Piotr Rudnicki. Mizar: the first 30 years. *Mechanized Mathematics and Its Applications*, 4:2005, 2005.
8. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
9. G. Sutcliffe and C.B. Suttner. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.
10. Geoff Sutcliffe. System Description: SystemOn TPTP. In *CADE*, pages 406–410, 2000.
11. J. van der Hoeven. GNU TeXmacs: A free, structured, wysiwyg and technical text editor. In Daniel Flipo, editor, *Le document au XXI-ième siècle*, volume 39–40, pages 39–50, 14–17 mai 2001. Actes du congrès GUTenberg.
12. Konstantin Verchinine, Alexander Lyaletski, and Andrei Paskevich. *System for Automated Deduction (SAD): a tool for proof verification*, volume 4603 of *Lecture Notes in Computer Science*, pages 398–403. Springer, July 2007.
13. Konstantin Vershinin and Andrey Paskevich. ForTheL - the language of formal theories. *International Journal of Information Theories and Applications*, 7(3):120–126, 2000.
14. M. Wagner, S. Autexier, and C. Benzmüller. PLATO: A Mediator between Text-Editors and Proof Assistance Systems. In S. Autexier and C. Benzmüller, editors, *7th Workshop on User Interfaces for Theorem Provers (UITP'06)*, volume 174(2) of *Electronic Notes on Theoretical Computer Science*, pages 87–107. Elsevier, August 2006.
15. Makarius Wenzel. *From Insight to Proof - Festschrift in Honour of Andrzej Trybulec*, chapter Isabelle/Isar - a generic framework for human-readable proof documents. 2007.

16. Alfred North Whitehead and Bertrand Russell. *Principia Mathematica*. Cambridge University Press, 1962.
17. Claus Zinn. *Understanding Informal Mathematical Discourse*. PhD thesis, Friedrich-Alexander-Universitt Erlangen Nürnberg, 2004.

Reusing Proofs in a Mathematical Library

Ivan Noyer¹ and Renaud Rioboo²

¹ Laboratoire d'Informatique de Paris 6
104 avenue du Président Kennedy
75016 Paris

`Ivan.Noyer@lip6.fr`

² École Nationale Supérieure d'Informatique pour l'Industrie et l'Entreprise
Centre de Recherche en Informatique du CNAM

1 place de la Résistance
91025 Evry

`Renaud.Rioboo@ensiie.fr`

Abstract. We present a framework for adapting existing proofs in a mathematical library of axioms and theorems to a proof under development. We give a heuristic which enables to reuse proofs for first order formulas of the FoCaL system. We first embed FoCaL concepts into a first order sequent calculus in which we characterize and build *quantitative proofs*. We then allow name conversion on functional and relational symbols which reduces the problem of proving a statement to finding existing proofs and related quantitative proofs. An informal algorithm is outlined on a FoCaL library example.

1 Context

Formally certifying programs is an activity which is known to be human time consuming. Indeed automatically verified algorithms need to be much more explicit than what one usually finds in mathematical books since those are intended for a human reader. As a consequence the number of lines dedicated to proving an algorithm is usually huge with respect to those devoted to its implementation.

This leaves the mathematical programmer with a puzzle: either she can find a statement together with a proof which suits her needs; either she must state and prove by herself. Proofs which are “left as an exercise for the reader” in textbooks must become explicit for an automated prover and the programmer must perform time consuming hand transformation of an existing proof to build a new one. For instance she will adapt a proof made for an additive law to a multiplicative law.

One solution to this problem is to abstract the proofs using supplementary parameters and to add premises which are the missing ingredients in the proof. Doing so we could state and prove some results for any commutative and associative composition law. However, this requires using higher order provers which are capable to manipulate functional values. During the design of the FoCaL model higher order was considered difficult to use by a mathematical programmer and the system only provides support for first order formulas.

We want in this work to provide tools adapted for this kind of formulas.

1.1 The FoCaL model

The FoCaL paradigm introduces layered concepts of *entities*, *collections* and *species*. Entities and collections are akin to elements and sets in mathematics.

Entities are the values manipulated by FoCaL programs, they belong to collections and are abstractly manipulated through *method calls*. In this paper we will view entities as terms of a Σ -algebra and methods as operations of the algebra.

Collections provide operations (*methods*) for manipulating entities and proved statements (*theorems*) relating methods and entities. That is we have first order formulas where entities may be quantified for logical statements and proofs of these formulas.

Collections are derived from species which describe and implement methods. Species follow an object oriented semantics with multiple inheritance where re-definition of methods and new proofs for statements are allowed. This makes collections and entities akin to objects and classes in object oriented programming languages as described in [9].

A species may contain *declarations* or *implementations*. At the declarative level the programmer may specify *operation signatures* (**signature**) or *logical statements* (**property**) expressing formulas. At the implementation level the programmer must provide code matching declared signatures (**let**) and proofs for logical statements (**proof of**).

Example 1. In the FoCaL library a *meet semi lattice* contains specifications for an equality and an infimum operation. That is we have a species Sp_a containing

```
signature equal : Self -> Self -> bool ;

signature inf : Self -> Self -> Self;

property inf_is_associative : all x y z in Self,
  !equal (!inf (!inf (x, y), z), !inf (x, !inf (y, z)));

property inf_left_substitution_rule : all x y z in Self,
  !equal (x, y) -> !equal (!inf (x, z), !inf (y, z));

property equal_symmetric : all x y in Self,
  !equal(x, y) -> !equal(y, x) ;

property equal_transitive : all x y z in Self,
  !equal(x, y) -> !equal(y, z) -> !equal(x, z) ;
```

We view signatures with target `bool` as predicate symbols and other signatures as function symbols. In order to simplify reading “`!equal(x, y)`” will be written “ $x = y$ ” and “`!inf(x, y)`” will be written “ $x \sqcap y$ ” in the text. We will furthermore use index on symbols denoting their origin.

We will use labels to name formulas and the above FoCaL properties will be

$$\forall d, \forall e, \forall f, e =_a f \Rightarrow (d \sqcap_a e) =_a (d \sqcap_a f) \quad (1)$$

$$\forall g, \forall h, \forall i, (g \sqcap_a (h \sqcap_a i)) =_a ((g \sqcap_a h) \sqcap_a i) \quad (2)$$

$$\forall l, \forall m, l =_a m \Rightarrow l =_a m \quad (3)$$

$$\forall n, \forall p, \forall q, n =_a p \Rightarrow p =_a q \Rightarrow n =_a p \quad (4)$$

The FoCaL programmer has access to formula 2 under the FoCaL name `inf_is_associative`.

Here we have renamed variable names in order to simplify the reading of further section which will use these specifications.

1.2 The Zenon prover

The FoCaL Proof Language allows declarative proof descriptions inspired by Leslie Lamport’s work (see [13]). The FoCaL compiler type-checks and translates user written proofs for the Zenon prover. Zenon (see [4]) is FoCaL’s dedicated automatic prover, it uses tableaux techniques to prove sequents (see [4] for details). One of the main features of Zenon is that it is able to produce proofs scripts or proof terms for the Coq proof assistant.

A FoCaL proof is translated into a set of sequents in an unsorted first order logic that Zenon is able to manage.

Example 2. Continuing meet semi lattices specifications of species \mathbf{Sp}_a , we also have a definition

```
let order_inf (x, y) = !equal (x, !inf (x, y));
```

We will denote it using an infix binary predicate symbol \leq_a which has FoCaL name `order_inf`. Furthermore the \leq_a symbol has a body which we denote

$$\forall j, \forall k, (j \leq_a k) \equiv (j =_a (j \sqcap_a k)) \quad (5)$$

Here by “ $N \equiv D$ ” we allow the prover to replace N with its definition D .

An important property for meet semi lattices is that “ \sqcap ” law defines an infimum for “ \leq ”. In FoCaL we have:

```
theorem order_inf_is_infimum : all x y z in Self,
  !order_inf (z, x) -> !order_inf (z, y) ->
  !order_inf (z, !inf (x, y))
proof =
  <1>1 assume x in Self,
    assume y in Self,
    assume z in Self,
    assume H1 : !order_inf (z, x),
    assume H2 : !order_inf (z, y),
    prove !order_inf (z, !inf (x, y))
  <2>1 prove !equal (z, !inf (!inf (z, x), y))
```

```

    by hypothesis H1, H2
      property inf_left_substitution_rule,
        equal_symmetric, equal_transitive
      definition of order_inf
<2>f qed by step <2>1
      property inf_is_associative, equal_transitive
      definition of order_inf
<1>2 conclude;

```

This script introduces the following steps:

- The name H1 standing for $z \leq_a x$
- The name <1>1 which stands for the statement

$$\forall x, \forall y, \forall z, (H1) \Rightarrow (H2) \Rightarrow z \leq (x \sqcap_a y)$$

- The global FoCaL theorem statement

$$\forall a, \forall b, \forall c, c \leq_a a \Rightarrow c \leq_a b \Rightarrow c \leq_a (a \sqcap_a b) \quad (6)$$

is then a direct consequence of statement (<1>1) and this is stated in the `conclude` directive. We will write that we have a sequent

$$(<1>1) \vdash (6)$$

- Similarly we have a sequent proving $z = ((z \sqcap_a x) \sqcap_a y)$ (ie <2>1)

$$\{(H1), (H2), (1), (3), (4), (5)\} \vdash <2>1$$

The `qed` directive proves <1>1 introducing statements (1) and (2).

In this paper we will not consider internal FoCaL names. We consider only that we have a proof for the sequent :

$$\{(1), (2), (3), (4), (5)\} \vdash (6)$$

1.3 Related work

There are many works about proof reuse, Roberto Di Cosmo's work [6] has led to numerous applications such as [1], [3], [5], [5] ... These works are based on type isomorphisms and for two isomorphic statement types τ and τ' and a proof term t for statement type τ one finds a proof term t' for statement type τ' . All these approaches are based on types and use the Curry-Howard-De Bruijn correspondence. We differ from these because we want to stay close to Zenon prover which uses first order unsorted logic.

Kolbe and Walther in [11] use generalization and second order unification which is implemented in the Plagiator (see [10]) system. Let us assume that we have a formula $P_1 = \forall x, f(x) > 0$, the symbol f is generalized giving the new formula $A = \forall f, \forall x, f(x) > 0$ and a formula $P_2 = \forall x, g(x) > 0$ is viewed as an

instance of A using second order unification and the proof of P_1 is adapted to P_2 using heuristics. This technique is also explored in [16] for Coq.

In [15] Melis and Shairer start by discovering analogies between two statements and similarities are used to transform a proof into another. They also use higher order unification to find similarities.

We differ from these approaches since we never need to use higher order unification. The section 2 is a simple overview of the concepts of terms, formulas, substitutions and unifiers. Our central result is the dependency property of section 3 which considers the order of quantifiers inside a prenex closed formula. Section 4 extends the substitutions to the symbols of functions and predicates. In section 5 we present our heuristic for reusing proofs.

2 Terms, formulas and unification

2.1 Expressions

Let us consider a set of (symbols of) *variables* \mathcal{X} , a set of (symbols of) *functions* \mathcal{F} , a set of (symbols of) *predicates* \mathcal{R} where each symbol has an arity. In this paper, formulas are always first order formulas built in a traditional way with symbols of \mathcal{X} , \mathcal{F} and \mathcal{R} respecting arities. A quantifier free formula will be called a *proposition*. We will write propositions with an indice “p” as in “ A_p ” in order to distinguish them from quantified formulas. An *expression* is either a term or a formula and we will call them $(\mathcal{X}, \mathcal{F})$ -terms, $(\mathcal{X}, \mathcal{F})$ -formulas or $(\mathcal{X}, \mathcal{F})$ -expressions.

For a formula A we will denote $U(A)$ (resp. $E(A)$) the set of universally (resp. existentially) quantified variables of A . For a term t we will denote by $\text{Var}_{\mathcal{X}}(t)$ the subset of \mathcal{X} made of variables appearing in t .

2.2 Substitutions

A variable substitution (or simply substitution) is a function σ from \mathcal{X} to $(\mathcal{X}, \mathcal{F})$ -terms which is written in postfix notation.

The domain of a substitution is the subset of \mathcal{X} made of those variables modified by σ and we only consider substitutions whose domain is finite. We will denote $\text{Dom}(\sigma)$ the domain of a substitution σ .

The substitution which gives to the variable x the image t is denoted $[x := t]$

A substitution σ is then extended to terms, formulas and sequents using standard induction rules.

Let t_1 and t_2 be two $(\mathcal{X}, \mathcal{F})$ -terms, an $(\mathcal{X}, \mathcal{F})$ -unifier (or simply unifier) of t_1 and t_2 is a substitution σ such that $t_1\sigma = t_2\sigma$. A *most general unifier* (an MGU for short) of t_1 and t_2 is an idempotent element minimal with respect to generality among unifiers of t_1 and t_2 . Herbrand’s finite unification algorithm enables to decide whenever two terms t_1 and t_2 are unifiable. Furthermore, the algorithm provides an MGU for t_1 and t_2 .

3 Quantitative proofs

In this section we restrict to prenex first order closed formulas. We will denote such formulas by $Q_{X_A}.A_p$, where A_p is a proposition and Q_{X_A} denotes the list of quantifiers binding the variables of A .

We will enumerate the variables of a prenex formula from the interior towards the outside of the formula. For a formula A we will denote by $\text{db}(x, A)$ the index of the quantified variable x of A . For instance in the following formula : $A = \forall x, \exists y, \forall z, P(x, y, z, t)$, we have $\text{db}(x, A) = 3$, $\text{db}(y, A) = 2$ and $\text{db}(z, A) = 1$.

A sequent $A \vdash B$ is *separated* if its hypothesis and conclusion are two prenex formulas and if bound variables of A (resp B) do not meet variables (either free or bound) of B (resp A). It is *closed* when both hypothesis and conclusion are closed. For instance $\forall x, P(x, u) \vdash \forall z, \exists y, Q(z, y)$, is a separated sequent but not $\forall x, P(x, z) \vdash \forall z, \exists y, Q(z, y)$ nor $\forall z, P(z, u) \vdash \forall z, \exists y, Q(z, y)$. A sequent $S = Q_{X_A}.A_p \vdash Q_{X_B}.B_p$, can always be transformed into an equivalent separated sequent S' by renaming its bounded variables. The sequent S' is then called a *separated renaming* of S .

Let $S = Q_{X_A}.A_p \vdash Q_{X_B}.B_p$ be a separated sequent. An *oriented unifier* of S is a unifier of the propositions A_p and B_p whose domain is contained in $U(Q_{X_A}.A_p) \cup E(Q_{X_B}.B_p)$. In short we only unify universal variables of the hypothesis and existential variables of the conclusion. Similarly we have *oriented MGUs* (OMGUs for short).

A *quantitative proof* of a separated sequent is a valid sequence of quantifier elimination inference rules terminated by an axiom. Such a proof is in particular a tree having only one leaf. We have identified a necessary and sufficient condition for a closed separated sequent to admit a quantitative proof. Our property, which we call *dependency property* (between quantifiers), can be expressed in several equivalent ways. Here we give one among them :

Proposition 1 (dependency property). *Let $S = Q_{X_A}.A_p \vdash Q_{X_B}.B_p$ be a separated closed sequent. The sequent S admits a quantitative proof if and only if there exists an OMGU σ such that for any $y \in E(Q_{X_A}.A_p)$, $x \in U(Q_{X_A}.A_p)$, $y' \in E(Q_{X_B}.B_p)$ and $x' \in U(Q_{X_B}.B_p)$ checking $\text{db}(y, Q_{X_A}.A_p) < \text{db}(x, Q_{X_A}.A_p)$ and $\text{db}(x', Q_{X_B}.B_p) < \text{db}(y', Q_{X_B}.B_p)$ we have :*

1. $y \notin \mathcal{V}(x\sigma)$
2. $x' \notin \mathcal{V}(y'\sigma)$
3. $y \notin \mathcal{V}(y'\sigma)$ or $x' \notin \mathcal{V}(x\sigma)$

Furthermore, if an OMGU σ for a separated closed sequent S satisfies the conditions 1, 2 and 3 above, then any other OMGU for S verifies 1, 2 and 3. The unifier σ is then called a *prover*. If a separated sequent S has a prover then any separated renaming of S has a prover. If a sequent S has a separated renaming which admits a prover we will also say that S has a prover.

As a corollary of the dependency property, one can notice that if all bounded variables of S are universally quantified, the existence of an OMGU is equivalent to the fact that S has a quantitative proof.

Example 3. Let us consider $Q_{X_A}.A_p$ which expresses continuity of the function f at a point:

$$\forall \epsilon, \forall x, \exists \alpha, \forall y, |y - x| < \alpha \Rightarrow |f(y) - f(x)| < \epsilon$$

Let $Q_{X_B}.B_p$ express uniform continuity of f :

$$\forall \epsilon', \exists \alpha', \forall x', \forall y', |y' - x'| < \alpha' \Rightarrow |f(y') - f(x')| < \epsilon'$$

– For $S_1 = Q_{X_A}.A_p \vdash Q_{X_B}.B_p$, we have the OMGU :

$$\sigma_1 = [\epsilon := \epsilon'] [x := x'] [y := y'] [\alpha' := \alpha]$$

The sequent S_1 has no quantitative proof because condition 3 of the dependency property is not true.

– For $S_2 = Q_{X_B}.B_p \vdash Q_{X_A}.A_p$, we have the OMGU :

$$\sigma_2 = [\epsilon' := \epsilon] [x' := x] [y' := y] [\alpha := \alpha']$$

The sequent S_2 has a quantitative proof because the three conditions of the dependency property are true.

– Finally let us consider this example:

$$S_3 = \forall x, \exists y, P(x, y) \vdash \forall x', \exists y', P(g(x', y'), y')$$

An OMGU fo S_3 is $\sigma_3 = [y' := y] [x := g(x', y)]$ The sequent S_3 has no quantitative proof because condition 1 of the dependency property is not verified.

4 Extended substitutions

Unification defined in section 2 does not allow terms such as $f(x, y)$ and $g(x, y)$ to match because no unification is possible on functional symbols. We will thus divide the set \mathcal{F} of functional symbols into a set \mathcal{F}_V of *abstract* (ie subject to unification) symbols and a set of \mathcal{F}_C of *concrete* (ie not subject to unification) symbols. We select ourselves which function symbols are considered abstract and which one have to be placed into \mathcal{F}_C . We proceed in a similar way splitting relational symbols of \mathcal{R} into $\mathcal{R}_C \cup \mathcal{R}_V$.

Example 4. Join semi lattices are very similar to meet semi lattices and we want to prove the theorem stating that the “ \sqcup_c ” operation induces a supremum for the “ \geq_c ” order relation. That is, we have a species \mathbf{Sp}_c containing the formula

$$\forall a', \forall b', \forall c', c' \geq_c a' \Rightarrow c' \geq_c b' \Rightarrow c' \geq_c (a' \sqcup_c b') \quad (7)$$

Which is very similar to formula (6) of species \mathbf{Sp}_a .

In order to work with both species \mathbf{Sp}_a and \mathbf{Sp}_c we view them as sets of properties, proofs and definitions over a common set of symbols. Symbols of \mathbf{Sp}_a will be considered abstract whereas symbols from \mathbf{Sp}_c will be considered concrete.

An *extended substitution* is a function σ from $\mathcal{X} \cup \mathcal{F}_V \cup \mathcal{R}_V$ to the union of the $(\mathcal{X}, \mathcal{F})$ -terms set, \mathcal{F} and \mathcal{R} which verifies the following conditions :

1. The set $\text{Dom}(\sigma) = \{e \in \mathcal{X} \cup \mathcal{F}_V \cup \mathcal{R}_V \mid \sigma(e) \neq e\}$ is finite. We will call this set the *domain* of σ and denote it by $\text{Dom}(\sigma)$.
2. For all f in \mathcal{F}_V , $f\sigma$ is in \mathcal{F} and has the same arity as f .
3. For all P in \mathcal{R}_V , $P\sigma$ is in \mathcal{R} and has the same arity as P .
4. For all x in \mathcal{X} , $x\sigma$ is a $(\mathcal{X}, \mathcal{F})$ -term.

As for variable substitution we denote $e\sigma$ for $\sigma(e)$. It is obvious that a variable substitution is a particular case of extended substitution which is constant over $\mathcal{F}_V \cup \mathcal{R}_V$.

Again, as for variable substitution, one can apply an extended substitution σ to the $(\mathcal{X}, \mathcal{F})$ -terms and $(\mathcal{X}, \mathcal{F}, \mathcal{R})$ -formulas (one has to be careful with variables captures).

For a set $\mathcal{V} \subset \mathcal{X} \sqcup \mathcal{F}_V \sqcup \mathcal{R}_V$ and an extended substitution σ we will denote $[\sigma_{\mathcal{V}}]$ the extended substitution such that:

- for all $e \in \mathcal{V}$, $e\sigma = e[\sigma_{\mathcal{V}}]$.
- $\text{Dom}([\sigma_{\mathcal{V}}]) \subset \mathcal{V}$

and we split σ into $[\sigma_{\mathcal{F}_V \cup \mathcal{R}_V}][\sigma_{\mathcal{X}}]$. In this way $[\sigma_{\mathcal{F}_V \cup \mathcal{R}_V}]$ can always be applied to any formula without capturing variables. If S is a general sequent having a proof in some first order sequent calculus, and if σ is an extended substitution, then $S[\sigma_{\mathcal{F}_V \cup \mathcal{R}_V}]$ also has a proof which is its image.

Let e_1 and e_2 be two terms or two propositions, an *extended unifier* of e_1 and e_2 is an extended substitution σ such that $e_1\sigma = e_2\sigma$. An *extended MGU* (EMGU for short) is an idempotent extended substitution which is a maximal element for generality.

We have a restricted case of second order unification where we only allow name conversion on functional and relational symbols. This enables us to have a decision procedure for the existence of an EMGU. Such EMGUs are furthermore compatible with the dependency property.

Let $S = Q_{X_A}, A_p \vdash Q_{X_B}, B_p$ be a closed separated sequent with A_p and B_p being propositions. An *extended OMGU* (EOMGU for short) of S is an EMGU σ of A_p and B_p which is oriented.

If S has an EOMGU σ verifying conditions of the dependency property, then $S' = Q_{X_A}, A_p[\sigma_{\mathcal{F}_V \cup \mathcal{R}_V}] \vdash Q_{X_B}, B_p[\sigma_{\mathcal{F}_V \cup \mathcal{R}_V}]$ has a quantitative proof and $[\sigma_{\mathcal{X}}]$ is a prover of S' . We say that σ is a *preprover* of S and we will further use preprovers when reusing proofs. When a separated renaming of a sequent S admits a preprover we will also say that S admits a preprover.

FFF Finally, we have the theorem :

Proposition 2. *Let S be a separated closed sequent. The existence of extended substitution σ such that $S[\sigma_{\mathcal{F}_V \cup \mathcal{R}_V}]$ admits a quantitative proof is a decision procedure which is equivalent to find a preprover of S .*

Example 5. Let us consider the sequent S :

$$\forall x, \forall y, x +_a y = y +_a x \vdash \forall z, \exists t, z \times f(t) = f(t) \times z$$

where $+_a$ is abstract, and $\times, =$ and f are concrete. The sequent S as an EOMGU $\sigma = [\sigma_{\mathcal{F}_V \cup \mathcal{R}_V}] [\sigma_{\mathcal{X}}] = [+_a := \times] ([x := z] [y := f(t)])$. Conditions of the dependency property are satisfied, thus σ is a preprover of S . The sequent $S [\sigma_{\mathcal{F}_V \cup \mathcal{R}_V}]$ has thus a quantitative proof with prover $[\sigma_{\mathcal{X}}]$. Here $S [\sigma_{\mathcal{F}_V \cup \mathcal{R}_V}]$ is

$$\forall x, \forall y, x \times y = y \times x \vdash \forall z, \exists t, z \times f(t) = f(t) \times z$$

5 Proof reuse example

In this section we want to discover and adapt the proof given in example 2 for species \mathbf{Sp}_a to species \mathbf{Sp}_c . We thus need to find statements in species \mathbf{Sp}_c matching statements (1), (2), (3), (4) and (5) of species \mathbf{Sp}_a . Those are

$$\forall d', \forall e', \forall f', e' =_c f' \Rightarrow (d' \sqcup_c e') =_c (d' \sqcup_c f') \quad (8)$$

$$\forall g', \forall h', \forall i', (g' \sqcup_c (h' \sqcup_c i')) =_c ((g' \sqcup_c h') \sqcup_c i') \quad (9)$$

$$\forall l', \forall m', l' =_c m' \Rightarrow l' =_c m' \quad (10)$$

$$\forall n', \forall p', \forall q', n' =_c p' \Rightarrow p' =_c q' \Rightarrow n' =_c p' \quad (11)$$

$$\forall j', \forall k', (j' \geq_c k' \equiv j' =_c (j' \sqcup_c k')) \quad (12)$$

Since we renamed by hand variables in our examples any sequent we consider is separated, furthermore since our statements are closed any sequent is closed. No existential quantifier is involved for our lattice theory examples, thus conditions for dependency property are always met.

Species \mathbf{Sp}_a will be viewed as a pair $(\mathbf{Lp}_a, \mathbf{Th}_a)$ of its axioms and theorems which are lists of sequents. Here $\mathbf{Th}_a = \{\mathbf{S}_a\}$ and

$$\begin{aligned} \mathbf{Lp}_a &= \{(1), (2), (3), (4), (5)\} \\ \mathbf{S}_a &= (1), (2), (3), (4), (5) \vdash (6) \end{aligned}$$

Species \mathbf{Sp}_c has no theorem and is $(\mathbf{Lp}_c, \mathbf{Th}_c)$ with empty \mathbf{Th}_c and

$$\mathbf{Lp}_c = \{(8), (9), (10), (11), (12)\}$$

We want to prove concrete statement (7) using concrete statements of \mathbf{Lp}_c and we allow adapting proofs of \mathbf{Th}_a . That is a function $\mathbf{reuse}((7), \mathbf{Lp}_c, \mathbf{Th}_a)$ whose result is a list of pairs (Π, τ) where Π is a proof of the sequent $\mathbf{Lp}_b \vdash (7) [\tau_{\mathcal{F}_V \cup \mathcal{R}_V}]$. We perform the following steps:

- The formula (7) does not match any statement in \mathbf{Lp}_c but matches only with statement (6) which is the conclusion of a theorem in \mathbf{Th}_a .
- Thus the sequent (6) \vdash (7) has a preprover σ which is

$$[a := a'] [b := b'] [c := c'] [\leq_a := \geq_c] [\sqcap_a := \sqcup_c]$$

- We apply $[\sigma_{\mathcal{F}_V \cup \mathcal{R}_V}]$, which is equal to $[\leq_a := \geq_c] [\sqcap_a := \sqcup_c]$, to each hypothesis of \mathbf{S}_a .
- We compute $\mathbf{reuse}((i) [\sigma_{\mathcal{F}_V \cup \mathcal{R}_V}], \mathbf{Lp}_c, \emptyset)$ (since $\emptyset = \mathbf{Th}_a \setminus \{\mathbf{S}_a\}$) for each statement (i) in the premises of \mathbf{S}_a . We only describe $(1) [\sigma_{\mathcal{F}_V \cup \mathcal{R}_V}]$ which is

$$\forall d, \forall e, \forall f, e =_a f \Rightarrow \sqcup_c(d, e) =_a \sqcup_c(d, f)$$

- This formula only matches with the concrete formula (8) of \mathbf{Lp}_c .
- The sequent $(8) \vdash (1) [\sigma_{\mathcal{F}_V \cup \mathcal{R}_V}]$ admits the preprover $\sigma_{(1)}$ which is equal to $:[d' := d] [e' := e] [f' := f] [=_a := =_c]$.
- We have $(8) \left[\sigma_{(1)\mathcal{F}_V \cup \mathcal{R}_V} \right] = (8)$ since (8) is concrete. Thus, the sequent

$$(8) \vdash (1) [\sigma_{\mathcal{F}_V \cup \mathcal{R}_V}] \left[\sigma_{(1)\mathcal{F}_V \cup \mathcal{R}_V} \right]$$

has a (quantitative) proof.

- Finally, the formula $(1) [\sigma_{\mathcal{F}_V \cup \mathcal{R}_V}] \left[\sigma_{(1)\mathcal{F}_V \cup \mathcal{R}_V} \right]$ is provable under hypothesis (8) of \mathbf{Lp}_c .
- Similarly, for all other hypothesis (j) of the sequent \mathbf{S}_a , one find only a corresponding preprover $\sigma_{(j)}$. We thus obtained

$$\left[\sigma_{(1)\mathcal{F}_V \cup \mathcal{R}_V} \right], \left[\sigma_{(2)\mathcal{F}_V \cup \mathcal{R}_V} \right], \left[\sigma_{(3)\mathcal{F}_V \cup \mathcal{R}_V} \right], \left[\sigma_{(4)\mathcal{F}_V \cup \mathcal{R}_V} \right], \left[\sigma_{(5)\mathcal{F}_V \cup \mathcal{R}_V} \right]$$

which are all equal to $[_a := =_c]$ which is thus their generalization.
So, for each hypothesis (j) of \mathbf{S}_a ,

$$\mathbf{Lp}_c \vdash (j) [\sigma_{\mathcal{F}_V \cup \mathcal{R}_V}] [_a := =_c]$$

has proof π_j since we have a quantitative proof.

- The sequent $\mathbf{S}_a [\sigma_{\mathcal{F}_V \cup \mathcal{R}_V}] [_a := =_c]$ has proof Π_a image of the proof we have for \mathbf{S}_a .
- Thus, the sequent $\mathbf{Lp}_c \vdash (6) [\sigma_{\mathcal{F}_V \cup \mathcal{R}_V}] [_a := =_c]$ admits proof Π_0 made from $\Pi_1, \Pi_2, \Pi_3, \Pi_4, \Pi_5$ and Π_a .
- For formula (7) we have a quantitative proof of sequent

$$(6) [\sigma_{\mathcal{F}_V \cup \mathcal{R}_V}] [_a := =_c] \vdash (7) [\sigma_{\mathcal{F}_V \cup \mathcal{R}_V}] [_a := =_c]$$

Since (7) is concrete we have proof Π_7 of sequent

$$(6) [\sigma_{\mathcal{F}_V \cup \mathcal{R}_V}] [_a := =_c] \vdash (7)$$

- Finally we can build our result (Π, γ) from Π_0, Π_7 and $[\sigma [_a := =_c]]$

6 Conclusion and further work

We have given a framework where FoCaL species are embedded into lists of formulas, definitions and sequents over $(\mathcal{X}, \mathcal{F}, \mathcal{R})$ -formulas with abstract and

concrete symbols. We use abstract formulas as models for concrete formulas using substitutions of abstract symbols into concrete terms, functions or predicates. These substitutions preserve good properties of MGUs and we presented the notion of extended oriented MGUs.

In proposition 1 (dependency property) we gave necessary and sufficient conditions for a sequent to admit a proof using only quantifier elimination rules. As a consequence, if a concrete formula is obtained from an abstract one by only inverting quantifiers this is detected. We then have adapted the classical unification algorithm to reduce more complex matchings to the dependency property.

We presented a heuristic enabling to transform a proof of an abstract formula into a proof of a matching concrete formula. So far our heuristic works well on all the examples we have taken from the FoCaL library. For instance in the FoCaL library equality is a user level notion and every function needs to be compatible with equality giving one formula for each signature defined. Fortunately all their proof are similar and this is detected by our heuristic.

We think that our heuristic is sufficiently specified to start an implementation which we will do very soon. We hope that this will reduce the amount of work needed to achieve a fully certified FoCaL development.

Other work to do is to go deeper in the structure of a proof in the FoCaL Proof Language since we have so far shown that the overall proof was reusable but we have no guarantee on the different steps which are passed to Zenon.

On a more theoretical point of view we also need to better identify how our generalization of first order unification relates to other forms of higher order unification.

References

1. Gilles Barthe and Olivier Pons. Type isomorphisms and proof reuse in dependent type theory. *Lecture Notes in Computer Science*, 2030:57–71, 2001.
2. Jon Barwise. *Handbook of Mathematical Logic*. North-Holland Publishing Co, 1982.
3. Olivier Boite. Proof reuse with extended inductive types. In K Slind et al., editor, *TPHOLS*, 2004.
4. R. Bonichon, D. Delahaye, and D. Doligez. Zenon: An extensible automated theorem prover producing checkable proofs. In *Logic for Programming, Artificial Intelligence, and Reasoning, 14th International Conference, LPAR 2007*. Springer, 2007.
5. David Delahaye. Information Retrieval in a Coq Proof Library using Type Isomorphisms. In Thierry Coquand, Peter Dybjer, Bengt Nordström, and Jan Smith, editors, *TYPES*, volume 1956 of *Lecture Notes in Computer Science (LNCS)*, pages 131–147, Lökeberg (Sweden), jun 1999. Springer.
6. Roberto Di Cosmo. *Isomorphisms of types: from λ -calculus to information retrieval and language design*. Progress in Theoretical Computer Science. Birkhauser, 1995. ISBN-0-8176-3763-X.
7. C. Dubois, T. Hardin, and V. Donzeau-Gouge. Building certified components within FoCaL. In *Revised Selected Papers from the Fifth Symposium on Trends in Functional Programming, TFP 2004*, pages 33–48, 2006.

8. T Hardin, F Pessaux, P Weis, and D Doligez. *The FoCalize Reference Manual*, 2009. available from <http://focalize.inria.fr>.
9. Thérèse Hardin and Renaud Rioboo. Les objets des mathématiques. *RSTI - L'objet*, Octobre 2004.
10. Thomas Kolbe and Jürgen Brauburger. Plagiator – a learning prover. In *in 14th Conference on Automated Deduction*, pages 256–259. Springer Verlag, 1997.
11. Thomas Kolbe, Christoph Walther, Fachbereich Informatik, and Technische Hochschule Darmstadt. Adaptation of proofs for reuse. In *Working Notes of the 1995 AAAI Fall Symposium on Adaptation of Knowledge for Reuse*, 61–67, 1995.
12. René Lalement. *Logique, réduction, résolution*. Masson, 1990.
13. Leslie Lamport. How to write a proof. *The American Mathematical Monthly*, 102(7):600–608, 1995.
14. The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0.
15. Erica Melis and Axel Schairer. Similarities and reuse of proofs in formal software verification. In *EWCBR '98: Proceedings of the 4th European Workshop on Advances in Case-Based Reasoning*, pages 76–87, London, UK, 1998. Springer-Verlag.
16. Olivier Pons. Generalization in type theory based proof assistants. In *TYPES*, pages 217–232, December 2000.
17. Virgile Prevosto and Damien Doligez. Algorithms and proof inheritance in the FoC language. *Journal of Automated Reasoning*, 29(3-4):337–363, December 2002.
18. The FoCaL Projet. URL, 2003. <http://focal.inria.fr/>.
19. Wolfgang Reif, Kurt Stenzel, Abteilung Programmiermethodik Und Compilerbau, and Fakultat Fur Informatik. Reuse of proofs in software verification. In *Workshop on Formal Approaches to the Reuse of Plans, Proofs, and Programs*, pages 284–293. Springer, 1993.

Computing Ranking and Unranking Functions for BDDs

Paul Tarau¹ and Brenda Luderma²

¹ Department of Computer Science and Engineering
University of North Texas

tarau@cs.unt.edu

² ACES CAD

brenda.luderman@gmail.com

Abstract. We describe Haskell implementations of combinatorial generation algorithms with focus on boolean functions and logic circuit representations. Using *pairing* and *unpairing* functions on natural number representations of truth tables, we derive an encoding for Binary Decision Diagrams (BDDs) with the unique property that its boolean evaluation faithfully mimics its structural conversion to a natural number through recursive application of a matching pairing function. We then use this result to derive *ranking* and *unranking* functions for BDDs and reduced BDDs.

The paper is organized as a self-contained literate Haskell program, available at <http://logic.csci.unt.edu/tarau/research/2008/fBDD.zip>.

Keywords: *binary decision diagrams, encodings of boolean functions, pairing/unpairing functions, ranking/unranking functions for BDDs, computational mathematics in Haskell*

1 Introduction

This paper is an exploration with functional programming tools of *ranking* and *unranking* problems on Binary Decision Diagrams. The paper is part of a larger effort to cover in a declarative programming paradigm some fundamental combinatorial generation and boolean function manipulation algorithms along the lines of Knuth's recent work [2].

The paper is organized as follows:

Sections 2 overviews efficient evaluation of boolean formulae in Haskell using bitvectors represented as arbitrary length integers and BDDs. Section 3 introduces pairing/unpairing functions acting directly on bitlists. Section 4 introduces a novel BDD encoding (based on our unpairing functions) and discusses the surprising equivalence between boolean evaluation of BDDs and the inverse of our encoding. Section 5 describes *ranking* and *unranking* functions for BDDs and reduced BDDs. Sections 7 and 8 discuss related work, future work and conclusions.

2 Evaluation of Boolean Functions with Bitvector Operations

Evaluation of a boolean function can be performed one result bit at a time for each combination of inputs. Unfortunately this does not take advantage of the ability of modern hardware to perform such operations one word a time - with the instant benefit of a speed-up proportional to the word size. An alternate representation, adapted from [2] uses integer encodings of 2^n bits for each boolean variable x_0, \dots, x_{n-1} . Bitvector operations are used to evaluate all value combinations at once.

Proposition 1 *Let x_k be a variable for $0 \leq k < n$ where n is the number of distinct variables in a boolean expression. Then column k of the truth table represents, as a bitstring, the natural number:*

$$x_k = (2^{2^n} - 1) / (2^{2^{n-k-1}} + 1) \quad (1)$$

For instance, if $n = 2$, the formula computes $x_0 = 3 = [0, 0, 1, 1]$ and $x_1 = 5 = [0, 1, 0, 1]$.

The following functions, working with arbitrary length bitstrings are used to evaluate the $[0..n-1]$ variables x_k with formula 1 and map the constant 1 to the bitstring of length 2^n , $111\dots 1$:

```
var_n n k = var_mn (bigone n) n k
var_mn mask n k = mask 'div' (2^(2^(n-k-1))+1)
bigone nvars = 2^2^nvars - 1
```

Variables representing truth tables specified as bitstring representations of natural numbers can be combined with the usual bitwise integer operators, to obtain new bitstring truth tables, encoding all possible value combinations of their arguments. Note that the constant 0 is represented as 0 while the constant 1 is represented as $2^{2^n} - 1$, corresponding to a column in the truth table containing ones exclusively.

We have seen that Natural Numbers in $[0..2^{2^n} - 1]$ can be used as representations of truth tables defining n -variable boolean functions. A binary decision diagram (BDD) [1] is an ordered binary tree obtained from a boolean function, by assigning its variables, one at a time, to 0 (right branch) and 1 (left branch).

3 Pairing/Unpairing

Let Nat denote the set of natural numbers (0 included). A *pairing function* is an isomorphism $f : Nat \times Nat \rightarrow Nat$. Its inverse is called an *unpairing function*.

We introduce here an unusually simple pairing function (also mentioned in [4], p.142). The function `bitpair` works by splitting a number's big endian bitstring representation into odd and even bits, while its inverse `bitunpair` blends the odd and even bits back together.

```

type Nat = Integer
data Nat2 = P Nat Nat deriving (Eq,Ord,Read,Show)

```

```

bitpair :: Nat2 → Nat
bitpair (P i j) =
  set2nat ((evens i) ++ (odds j)) where
    evens x = map (2*) (nat2set x)
    odds y = map succ (evens y)

```

```

bitunpair :: Nat → Nat2
bitunpair n = P (f xs) (f ys) where
  (xs,ys) = partition even (nat2set n)
  f = set2nat . (map ('div' 2))

```

The functions `set2nat` and `nat2set` convert to/from natural numbers to lists of exponents of 2 representing positions of bits=1.

```

nat2set n | n ≥ 0 = nat2exps n 0 where
  nat2exps 0 _ = []
  nat2exps n x =
    if (even n) then xs else (x:xs) where
      xs=nat2exps (n 'div' 2) (succ x)

```

```

set2nat ns = sum (map (2^) ns)

```

The transformation of the bitlists is shown in the following example with bitstrings aligned:

```

*BDD> bitunpair 2008
(60,26)

-- 2008:[0, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1]
-- 60:[   0,  1,  1,  1,  1]
-- 26:[  0,  1,  0,  1,  1 ]

```

4 Pairing Functions and Encodings of Binary Decision Diagrams

We show in this section that a Binary Decision Diagram (*BDD*) representing the same logic function as an n -variable 2^n bit truth table can be obtained by applying `bitunpair` recursively to `tt`. More precisely, we show that applying `bitunpair` results in a complete binary tree of depth n representing a *BDD* that returns `tt` when evaluated applying its boolean operations.

The binary tree type `BT` has the constants `B0` and `B1` as leaves representing the boolean values 0 and 1. Internal nodes (that represent `if-then-else` decision points), are marked with the constructor `D`. We also add integers, representing logic variables, ordered identically in each branch, as first arguments of `D`. The two other arguments are subtrees that represent `THEN` and `ELSE` branches:

```
data BT a = B0 | B1 | D a (BT a) (BT a) deriving (Eq,Ord,Read,Show)
```

The constructor BDD wraps together a tree of type BT and the number of logic variables occurring in it.

```
data BDD a = BDD a (BT a) deriving (Eq,Ord,Read,Show)
```

4.1 Unfolding natural numbers to binary trees with bitunpair

The following functions apply `bitunpair` recursively, on a Natural Number `tt`, seen as an n -variable 2^n bit truth table, to build a complete binary tree of depth n , that we represent using the BDD data type.

```
unfold_bdd :: Nat2 -> BDD Nat
unfold_bdd (P n tt) = BDD n bt where
  bt = if tt < max then shf bitunpair n tt
       else error
           ("unfold_bdd: last arg " ++ (show tt) ++
            " should be < " ++ (show max))
       where max = 2^2^n

shf _ n 0 | n < 1 = B0
shf _ n 1 | n < 1 = B1
shf f n tt = D k (shf f k tt1) (shf f k tt2) where
  k = pred n
  P tt1 tt2 = f tt
```

The examples below show results returned by `unfold_bdd` for the 2^{2^n} truth tables associated to n variables, for $n = 2$:

```
BDD 2 (D 1 (D 0 B0 B0) (D 0 B0 B0))
BDD 2 (D 1 (D 0 B1 B0) (D 0 B0 B0))
...
BDD 2 (D 1 (D 0 B1 B1) (D 0 B1 B1))
```

Note that no boolean operations have been performed so far and that we still have to prove that such trees actually represent BDDs associated to truth tables.

4.2 Folding binary trees to natural numbers with bitpair

One can “evaluate back” the binary tree of data type BDD, by using the pairing function `bitpair`. The inverse of `unfold_bdd` is implemented as follows:

```
fold_bdd :: BDD Nat -> Nat2
fold_bdd (BDD n bt) = P n (rshf bitpair bt) where
  rshf rf B0 = 0
  rshf rf B1 = 1
  rshf rf (D _ l r) = rf (P (rshf rf l) (rshf rf r))
```

Note that this is a purely structural operation and that integers in first argument position of the constructor D are actually ignored.

The two bijections, inverses of each other, work as follows:

```

*BDD>unfold_bdd (P 3 42)
  BDD 3 (D 2
    (D 1 (D 0 B0 B0)
      (D 0 B0 B0))
    (D 1 (D 0 B1 B1)
      (D 0 B1 B0)))

*BDD>fold_bdd it
42

```

4.3 Boolean Evaluation of BDDs

Practical uses of BDDs involve reducing them by sharing nodes and eliminating identical branches [1]. Note that in this case `bdd2nat` might give a different result as it computes different pairing operations. Fortunately, we can try to fold the binary tree back to a natural number by evaluating it as a boolean function.

The function `eval_bdd` describes the *BDD* evaluator:

```

eval_bdd (BDD n bt) = eval_with_mask (bigone n) n bt

eval_with_mask m _ B0 = 0
eval_with_mask m _ B1 = m
eval_with_mask m n (D x l r) =
  ite_ (var_mn m n x) (eval_with_mask m n l) (eval_with_mask m n r)

```

The *projection functions* `var_mn` defined in section 2 can be combined with the usual bitwise integer operators, to obtain new bitstring truth tables, encoding all possible value combinations of their arguments. Note that the constant 0 evaluates to 0 while the constant 1 is evaluated as $2^{2^n} - 1$ by the function `bigone`.

The function `ite_` used in `eval_with_mask` implements the boolean function `if x then t else e` using arbitrary length bitvector operations:

```

ite_ x t e = ((t 'xor' e) .&. x) 'xor' e

```

As the following example shows, it turns out that boolean evaluation `eval_bdd` faithfully emulates `fold_bdd`!

```

*BDD> unfold_bdd (P 3 42)
BDD 3 (D 2 (D 1 (D 0 B0 B0) (D 0 B0 B0))
  (D 1 (D 0 B1 B1) (D 0 B1 B0)))
*BDD> eval_bdd it
42

```

4.4 The Equivalence

We now state the surprising (and new!) result that boolean evaluation and structural transformation with repeated application of *pairing* produce the same result:

Proposition 2 *The complete binary tree of depth n , obtained by recursive applications of `bitunpair` on a truth table `tt` computes an (unreduced) BDD, that, when evaluated, returns the truth table, i.e.*

$$fold_bdd \circ unfold_bdd \equiv id \tag{2}$$

$$eval_bdd \circ unfold_bdd \equiv id \tag{3}$$

Proof. The function `unfold_bdd` builds a binary tree by splitting the bitstring $tt \in [0..2^n - 1]$ up to depth n . Observe that this corresponds to the Shannon expansion [7] of the formula associated to the truth table, using variable order $[n - 1, \dots, 0]$. Observe that the effect of `bitunpair` is the same as

- the effect of `var_mn m n (n-1)` acting as a mask selecting the left branch, and
- the effect of its complement, acting as a mask selecting the right branch.

Given that 2^n is the double of 2^{n-1} , the same invariant holds at each step, as the bitstring length of the truth table reduces to half.

We can thus assume from now on, that the BDD data type defined in section 4 actually represents BDDs mapped one-to-one to truth tables given as natural numbers.

5 Ranking and Unranking of BDDs

One more step is needed to extend the mapping between *BDDs* with n variables to a bijective mapping from/to *Nat*: we will have to “shift towards infinity” the starting point of each new block³ of BDDs in *Nat* as BDDs of larger and larger sizes are enumerated.

First, we need to know by how much - so we count the number of boolean functions with up to n variables.

```
bsum 0 = 0
bsum n | n>0 = bsum1 (n-1)
```

```
bsum1 0 = 2
bsum1 n | n>0 = bsum1 (n-1) + 2^2^n
```

The stream of all such sums can now be generated as usual⁴:

```
bsums = map bsum [0..]
```

```
*BDD> genericTake 7 bsums
[0,2,6,22,278,65814,4295033110]
```

³ defined by the same number of variables

⁴ `bsums` is sequence A060803 in The On-Line Encyclopedia of Integer Sequences, <http://www.research.att.com/~njas/sequences>

We can decompose n into the distance $n-m$ to the last `bsum` m smaller than n , and the index that generates the sum, k .

```
to_bsum n = (k,n-m) where
  k=pred (head [x|x←[0..],bsum x>n])
  m=bsum k
```

Unranking of an arbitrary BDD is now easy - the index k determines the number of variables and $n-m$ determines the rank. Together they select the right BDD with `unfold_bdd` and `bdd`.

```
nat2bdd n = unfold_bdd (P k n_m)
  where (k,n_m)=to_bsum n
```

Ranking of a BDD is even easier: we shift its rank within the set of BDDs with nv variables, by the value `(bsum nv)` that counts the ranks previously assigned.

```
bdd2nat bdd@(BDD nv _) = ((bsum nv)+tt) where
  P _ tt =fold_bdd bdd
```

As the following example shows, `bdd2nat` implements the inverse of `nat2bdd`.

```
*BDD> nat2bdd 42
BDD 3 (D 2 (D 1 (D 0 B0 B1) (D 0 B1 B0))
      (D 1 (D 0 B0 B0) (D 0 B0 B0)))
*BDD> bdd2nat it
42
```

We can now repeat the *ranking* function construction for `eval_bdd`:

```
ev_bdd2nat bdd@(BDD nv _) = (bsum nv)+(eval_bdd bdd)
```

We can confirm that `ev_bdd2nat` also acts as an inverse to `nat2bdd`:

```
*BDD> ev_bdd2nat (nat2bdd 2008)
2008
```

5.1 Reducing the *BDDs*

We sketch here a simplified reduction mechanism for BDDs eliminating identical branches. Note that the general mechanism involves DAGs and provides also node sharing [1].

The function `bdd_reduce` reduces a *BDD* by collapsing identical left and right subtrees, and the function `bdd` associates this reduced form to $n \in Nat$.

```
bdd_reduce (BDD n bt) = BDD n (reduce bt) where
  reduce B0 = B0
  reduce B1 = B1
  reduce (D _ l r) | l == r = reduce l
  reduce (D v l r) = D v (reduce l) (reduce r)
```

```
unfold_rbdd = bdd_reduce . unfold_bdd
```

The results returned by `unfold_rbdd` for $n=2$ are:

```

BDD 2 (C 0)
BDD 2 (D 1 (D 0 (C 1) (C 0)) (C 0))
BDD 2 (D 1 (C 0) (D 0 (C 1) (C 0)))
BDD 2 (D 0 (C 1) (C 0))
...
BDD 2 (C 1)

```

We can now define the *unranking* operation on reduced BDDs

```
nat2rbdd = bdd_reduce . nat2bdd
```

To be able to compare its space complexity with other representations we define a size operation on a BDD as follows:

```

bdd_size (BDD _ t) = 1+(size t) where
  size B0 = 1
  size B1 = 1
  size (D _ l r) = 1+(size l)+(size r)

```

6 Generalizing BDD ranking/unrankig functions

6.1 Encoding BDDs with Arbitrary Variable Order

While the encoding built around the equivalence described in Prop. 2 between bitwise pairing/unpairing operations and boolean decomposition is arguably as simple and elegant as possible, it is useful to parametrize BDD generation with respect to an arbitrary variable order. This is of particular importance when using BDDs for circuit minimization, as different variable orders can make circuit sizes flip from linear to exponential in the number of variables [1].

Given a permutation of n variables represented as natural numbers in $[0..n-1]$ and a truth table $tt \in [0..2^n - 1]$ we can define:

```

to_bdd vs tt | 0 ≤ tt && tt ≤ m =
  BDD n (to_bdd_mn vs tt m n) where
    n=genericLength vs
    m=bigone n
to_bdd _ tt = error
  ("bad arg in to_bdd⇒" ++ (show tt))

```

where the function `to_bdd_mn` recurses over the list of variables `vs` and applies Shannon expansion [7], expressed as bitvector operations. This computes the cofactor functions $f1$ and $f0$, to be used as `then` and `else` branches, when evaluating back the BDD to a truth table with if-the-else functions.

```

to_bdd_mn []      0 _ _ = B0
to_bdd_mn []      _ _ _ = B1
to_bdd_mn (v:vs) tt m n = D v l r where
  cond=var_mn m n v
  f0= (m 'xor' cond) .&. tt
  f1= cond .&. tt
  l=to_bdd_mn vs f1 m n
  r=to_bdd_mn vs f0 m n

```

Proposition 3 *The function `to_bdd` builds an (unreduced) BDD corresponding to a truth table `tt` for variable order `vs` that returns `tt`, when evaluated as a boolean function.*

We can reduce the resulting BDDs, and convert back from BDDs and reduced BDDs to truth tables with boolean evaluation:

```
to_rbdd vs tt = bdd_reduce (to_bdd vs tt)
from_bdd bdd = eval_bdd bdd
```

6.2 Generating Random BDDs

Random generation of BDDs has practical uses in testing and benchmarking of various electronic design automation tools and methodologies.

Deriving mechanisms for uniform generation of random instances is a classic application of ranking/unranking functions. Given a one-to-one mapping to *Nat* it reduces to the simpler problem of uniform generation of natural numbers in a given interval.

After customizing Haskell's library random generator

```
nrandom_nats smallest largest n seed =
  genericTake n
    (randomRs (smallest,largest) (mkStdGen seed))
```

one can define:

```
nrandom converter smallest largest n seed =
  map converter (nrandom_nats smallest largest n seed)
```

To generate 3 small instances of reduced BDD mapped to natural numbers from 10 to 20 one can write:

```
*BDD> nrandom nat2rbdd 10 20 3 77
[ BDD 2 (D 1 (D 0 B1 B0) B1),
  BDD 2 (D 1 (D 0 B0 B1) B1),
  BDD 2 (D 0 B0 B1)]
```

One can see the average size reduction from BDDs to reduced BDDs with something like:

```
*BDD> sum $ map bdd_size $ nrandom nat2bdd 1000 2000 10 7
320
*BDD> sum $ map bdd_size $ nrandom nat2rbdd 1000 2000 10 7
194
```

Needless to say, one might notice the compactness and elegance of a declarative language like Haskell for such ad-hoc tasks, recommending it as a powerful scripting language for electronic design automation tools.

7 Related work

Pairing functions have been used for work on decision problems as early as [5].

BDDs are the dominant boolean function representation in the field of circuit design automation [3]. BDDs have also been used in a Genetic Programming context [6] as a representation of evolving individuals subject to crossovers and mutations expressed as structural transformations.

8 Conclusion and Future Work

The surprising connection of bitstring based pairing/unpairing functions and to BDDs came out as the indirect result of implementation work on a number of practical applications. Our initial interest has been triggered by applications of the encodings to combinational circuit synthesis [8,9] and ongoing work on genetic programming algorithms.

Given the connection between BDDs to boolean and finite domain constraint solvers it would be interesting to explore in that context, efficient succinct data representations derived from our BDD encodings.

References

1. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.
2. D. Knuth. The Art of Computer Programming, Volume 4, draft, 2006. <http://www-cs-faculty.stanford.edu/~knuth/taocp.html>.
3. C. Meinel and T. Theobald. Ordered binary decision diagrams and their significance in computer-aided design of vlsi circuits. *Journal of Circuits, Systems, and Computers*, 9(3-4):181–198, 1999.
4. S. Pigeon. *Contributions à la compression de données*. Ph.d. thesis, Université de Montréal, Montréal, 2001.
5. J. Robinson. General recursive functions. *Proceedings of the American Mathematical Society*, 1(6):703–718, dec 1950.
6. H. Sakanashi, T. Higuchi, H. Iba, and Y. Kakazu. Evolution of binary decision diagrams for digital circuit design using genetic programming. In T. Higuchi, M. Iwata, and W. Liu, editors, *ICES*, volume 1259 of *Lecture Notes in Computer Science*, pages 470–481. Springer, 1996.
7. C. E. Shannon. *Claude Elwood Shannon: collected papers*. IEEE Press, Piscataway, NJ, USA, 1993.
8. P. Tarau and B. Luderman. A Logic Programming Framework for Combinational Circuit Synthesis. In *23rd International Conference on Logic Programming (ICLP), LNCS 4670*, pages 180–194, Porto, Portugal, Sept. 2007. Springer.
9. P. Tarau and B. Luderman. Exact combinational logic synthesis and non-standard circuit design. In *CF '08: Proceedings of the 2008 conference on Computing frontiers*, pages 179–188, New York, NY, USA, 2008. ACM.