Proceedings of the

11th International Workshop on the Implementation of Logics

November 23rd, 2015
Suva, Fiji

Boris Konev[1], Stephan Schulz[2] and Laurent Simon[3], editors

[1] University of Liverpool
[2] DHBW Stuttgart
[3] University of Bordeaux

Preface

This volumes contains the papers selected for presentation at the *11th International Workshop on the Implementations of Logic*, at the Laucla Bay Campus of the University of the South Pacific in Suva, Fiji.

The first *Workshop on Implementations of Logic* was held in November 2000 on Reunion Island as an invitation-only event, associated with the *7th International Conference on Logic for Programming and Automated Reasoning*. This successful event sparked the creation of a new workshop series, the *International Workshop on the Implementation of Logics* (IWIL), into which it retroactively was adopted. Since 2001, the IWIL workshop has followed LPAR around the world, thus boldly going where no workshop has gone before. This year's IWIL is the 11th instance of the workshop, and is associated with the *20th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*. To prevent hasty generalization: not all instances of the workshop were held on tropical islands.

The program committee received 12 submissions, each of which was submitted to three reviewers for peer review. As always, using EasyChair has made organisation of the review-process a largely pain-free experience. We were able to accept 11 of the submissions, dealing with topics as diverse as applications of automated reasoning to mathematics and logistics, improvements in propositional logic solvers, and new algorithms for and extensions to first-order and equational logics.

November 2015

Boris Konev
Stephan Schulz
Laurent Simon

Defining the meaning of TPTP formatted proofs

Roberto Blanco, Tomer Libal and Dale Miller

Inria & LIX/École polytechnique
{roberto.blanco,tomer.libal,dale.miller}@inria.fr

Abstract

The TPTP library is one of the leading problem libraries in the automated theorem proving community. Over time, support was added for problems beyond those in first-order clausal form. TPTP has also been augmented with support for various proof formats output by theorem provers. Such proofs can also be maintained in the TSTP proof library. In this paper we propose an extension of this framework to support the semantic specification of the inference rules used in proofs.

1 Introduction

A key element in optimizing the performance of systems is the ability to compare them on common benchmarks. In the automated theorem proving community, such benchmarks are available via the "Thousands of Problems for Theorem Provers" (TPTP) library [31]. A part of the library's success lies in its syntactic conventions, which are both intuitive to read and rich enough to encode many kinds of problems. Another advantage of its syntax is its simple structure that allows one to easily write parsers and other utilities for it. As part of the evolution of the library and its syntax, a support for proofs was added. In order to support the proof library, called "Thousands of Solutions from Theorem Provers" (TSTP), the syntax needed to be extended to support different types of proofs, in particular, directed acyclic graph proofs. This syntax allows for the description of proofs as a series of steps which are themselves encoded collections of inference rules and some additional annotations. One shortcoming of this format is its emphasis on syntax and its inability to describe precisely the semantics of the inferences used.

The increased complexity of today's automated theorem provers has brought with it a need for proof certification. Errors in proofs can result from several sources ranging from bugs in the code to inconsistencies in the object theory. In order to improve this situation, several tools for proof certification have been implemented that can improve our confidence in the proofs output from theorem provers. These tools can be classified into two groups. First it is possible to actually prove that a theorem prover is formally correct (see, for example, Ridge and Margetson [24]). The second group consists of tools for verifying, not the theorem provers themselves, but their output. This group can be further divided into two groups. The first group consist of systems for replaying proofs using external theorem provers for verifying specific steps. Among these, one can count the general tools Sledgehammer [22, 3], PRocH [13] and GDV [29] as well as more specific efforts such as the verification of E prover [27] proofs using Metis [22]. The second group contains tools having an encoding or a translation of the semantics of theorem provers, which is then used in the replaying process. This last group can be divided again into specific tools, such as Ivy [17] and the encodings of MESON [15] and Metis [12] in HOL Light and Isabelle, respectively, and general tools such as Dedukti [2] and ProofCert [19].

These various classes of tools represent different approaches to proof certification. While we can have a high level of trust in the correctness of the provers in the first group, their performance cannot be compared to that of the leading theorem provers like E [27] and Vampire

[23]. The remaining groups do not pose restrictions on the provers themselves but the generality and automation of those in the second group come with the cost of using an external theorem prover and translations, which might result in reduced confidence. The last two groups require an understanding of a theorem prover's semantics so that one can guarantee the soundness of proofs by their reconstruction in a low level formal logic. Working with an actual proof has several advantages as one can apply proof transformations and other procedures. The last group has additional advantages over the previous one: a single certifier can be written that should be able to check proofs from a range of different systems and the existence of a common language for proofs allows for the creation of proof libraries and marketplaces [19].

Those tools in the last group have, so far, only limited success in the general community. One reason for this is that understanding and specifying the semantics of proofs requires sophistication in the interplay between deduction and computation (whether via function-style rewriting or proof-search).

The difficulty in understanding the semantics of the object calculi lies in the gap between the implementers of theorem provers and the implementers of the proof certifiers. Currently, the normal process for certifying the output of a certain theorem prover is for a dedicated team on the certifier side to try to understand the semantics of each inference rule of the object calculus. This approach suffers many times from missing documentation, different names and versions of actual software, and insufficient information in the proofs themselves [5]. This gap is enlarged by the fact that teams of implementers and certifiers can reside in different locations or even work in different periods, thus making the communication between them difficult or even impossible.

One way to overcome this gap is to supply the implementers of theorem provers with an easy to use and well-known format in which to describe the semantics of their inference rules. This format should be general enough to allow specifications to range from precise (functional) definitions—translating a proof in the object calculus into a proof in another, trusted and well-known calculus—and informal definitions, with hints on the right way to understand the object calculus without needing to specify how to actually reconstruct a formal proof.

In this paper we aim at helping to reduce the gap mentioned above between those who produce proofs and those who must certify them. We propose to use a format which is well-known to the implementers of theorem provers—the TPTP format itself—for the purpose of describing not only problems and proofs but also the semantics of proofs. This will make a TPTP file an independent unit of information which can be used for certification as well.

An additional advantage of using the TPTP format to specify semantics is the same one mentioned above for building tools for the TPTP library. The predicate logic form of the problems, their solutions, and now, also their semantics, will allow proof certifiers to easily access the semantics and will further diminish the gap between the theorem provers and their certifiers. For example, the `checkers` proof certifier [5] (written using logic programming), will only require minor computations to be applied to the input files, if any. Such simplicity helps to improve the trust of the certification process.

The paper is organized as follows. In the next section we present and describe both the TPTP syntax and the notion of using predicates in order to define the semantics of logics. In section 3 we present and discuss the minor augmentations needed in the TPTP format in order to support the ability to use this format to denote semantics. Section 4 is devoted to the full description of four examples from four different theorem provers. The concluding section suggests some additional advantages of using this approach.

2 Preliminaries

2.1 Syntax in TPTP

A beneficial side effect of the TPTP library as a standard test suite for automated theorem provers is the standardization of a language to express logic problems and their solutions. It is no coincidence that this standardization on the language is credited as one of the keys to the success of the TPTP project [31].

The TPTP syntax is built upon a core language called THF0 [1]. This core can be restricted to support a number of interface languages: untyped first-order logic as first-order form (`fof`) and clause normal form (`cnf`), typed first-order form (`tff`), and typed higher-order (`thf`). Furthermore, all of these can be used in combination with a process instruction language (`tpi`) for the manipulation of formulas. The concrete syntax revolves around the concept of annotated formulas and is expressive enough to structure proofs and embed arbitrarily complex information as annotations.

Among the stated design goals of the format, both extensibility and readability (by machines and logicians) figure prominently. In addition, care has been taken to ensure that the grammar remains compatible with the logic programming paradigm, and TPTP documents are, in fact, valid Prolog programs.

For defining a formula using the TPTP format, one uses the following templates:

```
Language(Name, Role, Formula).
Language(Name, Role, Formula, Annotations).
```

where `Language` $\in$ {`cnf`,`fof`,`tff`,`thf`,`tpi`} (see above for the list of interface languages), `Role` describes the role of the formula —i.e. 'axiom' or 'type'—, `Formula` is the encoding of the formula in the specified language and `Annotations` contains optional additional information.

A template for defining structural derivations is the following:

```
Language(Name, Role, Formula, inference(Rule, Info, Parents)).
```

Here `Rule` denotes the name of the inference rules used, `Info` optionally specifies additional information, like the SZS output value of the inference and `Parents` also optionally refers to the names of the formulas which were used in the application of this rule. `Formula`, as before, is an encoding of the derived formula in the respective language. The SZS ontology [30] referred to above supplies a set of inference properties, such as theoremhood, satisfiability, etc., which give some semantical information. It should be noted that this information might suffice for proof replaying using an external prover [22, 13, 29], but does not fully help in understanding the semantics of the inference rules themselves.

Another useful feature of TPTP is the `include` directive, which performs a syntactical inclusion of one file into another. This directive may help reduce redundancies.

2.2 Denoting semantics as logic programs

As already mentioned, one way to formally describe the semantics of an inference rule is by translating an instance of this rule into a derivation in another, well-known, calculus. This translation can be *determinate* or *nondeterminate*: in the latter case, a logic programming implementation of the translation could allow for that nondeterminism to be explored using backtracking search. Nondeterminism in the specification of proof semantics has been considered in other systems as well. In particular, nondeterminism is allowed in the *Foundational Proof*

Certificates (FPC) framework [7, 19] where client-side inference rules (i.e., rules implemented in theorem provers) are translated into low-level rules of sequent calculus. The `checkers` proof certifier [5], based on the FPC framework, used the λProlog logic programming language [20] to provide for a backtracking search approach to exploring any nondeterminism in such translations. The basic idea is to program a set of predicates which will guide the search in the target calculus. By guiding the search for a derivation of an instance of an inference rule in a well-known calculus, this set of predicates can be considered as denoting the semantics of this inference.

Before describing how we plan to use the TPTP framework to specify the translation of inference rules, we need to present the underlying principles behind the idea.

First, and critically, semantic descriptions are not "one size fits all": there is an underlying trade-off between space (for storing a proof) and time (for checking a proof). More detailed semantic translations, insofar as the information provided is useful, produce more efficient verifications; conversely, high-level, conceptual descriptions may serve as guidance but cannot be used to generate a constructive decision procedure without additional information or search. At one extreme are fully determinate translations of inference rules and at the other extreme are minimal but sufficient hints to allow a possible reconstruction of a proof in an independent checker. In contrast, here we consider the full spectrum of implicit vs. explicit reconstruction.

For example, suppose we wish to obtain a proof of a formula $A \wedge B \wedge C$. It may simply be stated that to do this, separate proofs for A, B, and C are needed:

$$\frac{A \quad B \quad C}{A \wedge B \wedge C}$$

To understand the meaning of this inference rule, one can try to infer the conclusion from the hypotheses using a well-known calculus, the sequent calculus for example, which tells us that in order to derive the original goal, two proofs are needed, one for A and a second one for $B \wedge C$, and then divide in turn this second composite proof into sub-proofs of B and C:

$$\dfrac{A \quad \dfrac{B \quad C}{B \wedge C}}{A \wedge B \wedge C}$$

The question of whether we can trust the first inference relies on the fact that its semantics is defined by the second, in the sense that it constitutes a formal derivation of the intended meaning, namely, that one can obtain a proof of the conjunction of three goals from proofs of each of those goals. Trust in the calculus of choice extends to trust in the inference rules that it can justify. Contrariwise, consider an alternative candidate for an inference rule:

$$\frac{A \quad B \quad C}{A \vee B \vee C}$$

It can be proved that a reasonable calculus will be unable to derive an inference of this shape. In the absence of a trustworthy proof reconstruction of the postulated inference rule, its validity cannot be accepted.

Consider now the more realistic case of paramodulation [25], a concrete instance of which we study in section 4.1. In this case, an explicit functional translation of the formal definition is far from being trivial. Conversely, it may be stated, more informally, that paramodulation handles equality modulo reflexivity: that is to say, the transitivity and symmetry axioms can be used to simulate this rule in a logic without explicit handling of equality (note that reflexivity axioms must be given externally for the equality procedure to be complete).

By applying some additional effort, this approach was implemented successfully in `checkers` and is capable of guiding the proof search for arbitrary instances of the paramodulation rule. This implies, therefore, that supplying the two axioms provides enough information to assist the automatic certification of this inference rule.

3 Thousands of Semantically Annotated Solutions for Theorem Provers (TATP)

In order to have the cleanest and most declarative treatment of one logic (i.e, the logic of the client prover) within a second logic encoding inference rules and their associated proof search, we shall make use of the notion of *order* of THF0. In this setting, defining the semantics of logical formulas of order n employs a meta-level logic of order $n+1$. For example, if our client proofs are only propositional formulas (order 0) then the first-order fragment of THF0 suffices. However, if our client proofs are first-order formulas, then we employ directly the second-order subset of THF0. As seen in section 2, TPTP is equipped with the necessary syntax necessary to define formulas of an arbitrary finite order. It is largely for this reason that we will employ λProlog [20] to automate[1] the translation of inference rules, since the logic underlying λProlog is close to that underlying THF0 (both are closely related to Church's Simple Theory of Types [8]). For example, in order to define the provability (via the predicate pr) of a classical first-order quantifier, one can use the following λProlog clause:

$$pr(\forall x.Bx) \ \texttt{:-} \ \Pi x.\ pr(Bx).$$

where $\forall$ is the object-logic universal quantifier and Π is the meta-level universal quantifier. The implementation of λProlog deals directly with the many issues related to binding, substitutions, eigenvariables, and unification [10].

TPTP proofs are already annotated by the inference rules that are used in order to derive the formula. These annotations, however, lack a formal semantics and they cannot normally be understood by a person not familiar with the details of the system that outputs those annotations. We propose to use the TPTP `thf` syntax in order to allow the implementer of a theorem prover to include semantical information about their inference rules, thereby replacing imprecise and specialized annotations with more formal annotations. Note that by using the TPTP `include` directive, one does not need to include these definitions in every proof generated but just define them once.

In order to allow such a use, we can first define a new `role` for formula definitions. We therefore add the following directive to the TPTP syntax:

```
<formula_role> :== semantics
```

A TPTP semantics definition will have the following form:

```
thf(Name, semantics, Formula, Annotations).
```

where `Name`, by convention, should consist of the prover name, underscore and then the inference name which was used in the proof, for example "`e_pm`". `Formula` can contain an arbitrary higher-order typed formula denoting the semantics definitions. In the rest of the paper, though, we ignore typing information in order to focus on clarity. Note that the logical programming

[1] Both Teyjus [21] and ELPI [9] are implementations of λProlog.

language λProlog requires formulas to be in fragment of higher-order logic called *hereditary-Harrop formulas* [20]. As has been shown elsewhere (for example, [10] and [20, Chapter 9]), this fragment of logic is able to elegantly specify a variety of inference rules. Thus, if one can define the semantics of inference rules using these formulas, one could use, with minimal intervention, proof checking software like `checkers` to verify proofs. Lastly, `Annotations` can contain additional (informal) information which can help understanding the semantics. These annotations may include the name of a target logic which can be used for proof reconstruction or a reference to a paper which defines the target or object logics.

A question we still need to answer is how one can define the semantics of an inference rule and how we can make that task as simple as possible. The decision taken here is to allow the implementer of a theorem prover to use, in order to define the semantics of their own rules, the inference rules and the theory of any other calculus. In general, they can decide to specify the semantics using the inference rules of another theorem prover. However, it would be preferable to specify the semantics using the inference rules of a well known calculus, like the original resolution calculus by Robinson [26], for example. The approach we are discussing here (with examples in the next section) stands in contrast to what is being done in the ProofCert project [18]. In that project, the meanings of all inference rules are "compiled" into a low-level proof system (representing an "assembly language" for inference). We do not insist on employing that framework, opting instead for a less tedious and more high-level approach to providing some useful information about the inference rules used in a specific theorem prover.

It should be noted that according to the above conventions, one has full control over the amount of detail and choice of the target calculus. A large amount of detail might enable a precise proof reconstruction in a fine grained calculus, for example, in the sequent calculus (the ProofCert project does exactl1y this, for example). We want to stress here that since the aim of these definitions is to communicate information about the semantics, such detailed information is not necessary. There can be benefits for both the certification team and the implementers in specifying information about the semantics of their own rules using the highest level calculus known to the community. This will make the definition simpler and will also contribute to the modularity of a certification tool since the certifiers will only need to implement the semantics of the high level calculus, the semantics of which being widely known. For example, all superposition provers are using variants of the paramodulation inference rule [25]. Defining the semantics of these variants can be done by a number of individualized, detailed descriptions or be based on the known notion of paramodulation. It seems more intuitive and simple for implementers to choose the second option and let the certification team implement the general semantics of paramodulation. This is the approach taken in `checkers` and described below in the examples. A fine grained description of the semantics of paramodulation can be found, for example, in [6].

When defining proofs in the TPTP format, the information of which inference rule to use is supplied using the annotation directive. We will do the same and use this directive in order to supply the information of what calculus is being used to define the semantics. To this end, we will add the following directives to the TPTP syntax:

```
<source> ::= <calculus_info>
<calculus_info> ::= calculus(<calculus_name><optional_info>)
<calculus_name> ::= <atomic_word>
```

Using these directives, the user can specify the name of the calculus used and supply additional information, such as the name of a paper where this calculus is defined.

The last remaining task is to be able to bind the instances of the inference rules in the

proofs to their semantics definitions. We suggest the following convention: in order to specify an inference call in DAG form, i.e. the ones used in proofs, the user will employ the following predicate:

$$\texttt{<inference\_rule>}(f, f_1, \ldots, f_n)$$

where f is the derived formula and the remaining formulas are the inputs used in the derivation. Examples of this convention are given next.

4 Examples

We demonstrate the use of THF0-style annotations on four examples taken from inference rules used by four different theorem provers.

4.1 Paramodulation in E

The E prover [27] was among the first provers to output proofs using the TPTP format. A staple on the podium at the annual CASC competitions [32], E is used by many other first- and higher-order theorem provers. E is a superposition-based, saturating, automated theorem prover based on a purely equational paradigm. As such, it implements several variants of the paramodulation rule.

Definition 1 (Paramodulation [25])**.** *Given clauses A and $\alpha' = \beta' \vee B$ (or $\beta' = \alpha' \vee B$) having no variables in common and such that A contains a term δ, with δ and α' having a most general common instance α identical to $\alpha'[s_i/u_i]$ and to $\delta[t_j/w_j]$, form A' by replacing in $A[t_j/w_j]$ some single occurrence of α (resulting from an occurrence of δ) by $\beta'[s_i/u_i]$, and infer $A' \vee B[s_i/u_i]$.*

One concrete variant, for example, is given in the `pm` rule of E. This rule is applied in TPTP syntax using the following form:

```
cnf(ClauseId, Role, Formula, inference(pm, [status(thm)],
    [SourceId1, SourceId2, theory(equality)])).
```

where `SourceId1` and `SourceId2` are the two clauses to which the paramodulation rule is applied to obtain `ClauseId`, corresponding to the formula given by `Formula` and with role `Role`. The semantics of this rule is similar to the semantics of the paramodulation rule (from [25] and our definition), with the peculiarity that the tactic presents symmetry for both `ClauseId1` and `ClauseId2`.

To produce the full definition we proceed in two steps. First we present a TPTP formula that denotes the semantics of the `pm` rule.

```
thf(eprover_pm, semantics, Formula, calculus(paramodulation,
    [p.5 in  [25] ])).
```

where the semantics is documented by a suitable bibliographic reference. Second, we define `Formula` as the mapping between the specific variation of paramodulation defined by the E prover (namely, the `pm` tactic) and the canonical semantics derived from the definition:

```
∀ SourceId1, SourceId2, ClauseId:
pm(SourceId1, SourceId2, ClauseId)
    ⇐ paramodulation(ClauseId, SourceId1, SourceId2)
    ∨ paramodulation(ClauseId, SourceId2, SourceId1)
```

where, as usual, free variables are universally quantified; we have made this quantification explicit in the present formulation.

4.2 Binary resolution in Vampire

Vampire [23] is a theorem prover that implements the superposition calculus and is the regular winner of the first-order division in the CASC competition over the last decade [32]. Proofs proceed by saturation and rely on redundancy elimination and a wide range of advanced techniques to maximize performance, one of its original design goals. It features a rich collection of inference rules and supports the TPTP syntax, including various extensions. Here we inspect TSTP entries produced by Vampire 4.0 to infer program semantics.

Definition 2 (Binary resolution [26])**.** *Given two clauses* $A = a_1 \vee \ldots \vee a_m$ *and* $B = b_1 \vee \ldots \vee b_n$ *and a pair of complementary literals, one from each clause, i.e.,* $a_i = \neg b_j$ *or* $\neg a_i = b_j$*, the resolution rule derives a new clause with all the literals except the complementary pair:* $C = a_1 \vee \ldots \vee a_{i-1} \vee a_{i+1} \vee \ldots \vee a_m \vee b_1 \vee \ldots \vee b_{j-1} \vee b_{j+1} \vee \ldots \vee b_n$*.*

The binary resolution rule includes the possibility of applying a unification procedure to a pair of unifiable literals, and substituting the most general unifier in the resolvent C. Some categories of binary resolution can be defined. These are not necessarily mutually exclusive:

- Positive resolution, if one of the parent clauses is a positive clause, i.e., all its literals are positive.
- Negative resolution, if one of the parent clauses is a negative clause, i.e., all its literals are negative.
- Unit resolution, if one of the parent clauses is a unit clause, i.e., formed by exactly one literal.

Vampire outputs natively to TPTP in addition to its own internal format, closer to that of Prover9 that we treat in the next subsection. Now, we consider the TPTP output of the basic resolution rule.

```
fof(ClauseId, plain, Formula,
    inference(resolution, [], [SourceId1, SourceId2])).
```

The translation takes this to the higher-order formula and adjusts the annotation information in the inference name to point to the name of the logic program that implements the procedure.

```
thf(vampire_resolution, semantics, Formula, calculus(hol)).
```

where `Formula` is defined to be

```
∀ S1, S2, R1, R2:
resolution(S1, S2, R1 ∨ R2)
  ⇐ ∃ L: select(S1, L, R1) ∧ select(S2, ¬L, R2)
       ∨ select(S1, ¬L, R1) ∧ select(S2, L, R2).
```

Here the standard list selection predicate is used to pick a literal from a list-like clause and yield a copy of the clause without the chosen literal. We are still free to use concatenate clauses by way of a disjunction.

A clause produced by binary resolution is specified by the two premise clauses and (considering each of these in CNF form, and in turn a CNF form as an indexed list of disjuncts) by the disjunct from each clause that is involved. We also assume a predicate specifying, and therefore declaratively implementing, binary resolution, that acts on formulas and can check whether the specified application of resolution yields the target formula.

4.3 Hyperresolution in Prover9

Prover9 [16] is a theorem prover based around the techniques of resolution and paramodulation, and the successor of the Otter theorem prover. The last available version is 2009-11A, dated November 2009. While development has since ceased, the tool remains in use. Prover9 does not produce output in TPTP format, and therefore TSTP contains unparsed execution traces. However, the input and output formats of the prover are simple and well documented, and their semantics can be easily formalized. Interestingly, such a translation procedure offers the possibility of generating the native TSTP output that is missing from the problem library, together with its semantics.

In this subsection we consider hyperresolution [11], one of the primary tactics used by Prover9. An informal definition of the inference rule follows.

Definition 3 (Hyperresolution [11])**.** *Assume a nucleus clause A, nonpositive, with a number k of negative literals $\neg a_{i_1}, \ldots, \neg a_{i_k}$, and as many satellite clauses $B_1, \ldots, B_k$, each of which resolves on of those negative literals, i.e., $B_j = \ldots \vee a_{i_j} \vee \ldots$. The hyperresolution rule resolves all the negative literals in the nucleus, each with its satellite, producing a positive clause C.*

Hyperresolution can be seen as a sequence of applications of binary resolution. It is likewise possible to reverse polarities and speak of negative hyperresolution. A related concept is that of unit-resulting resolution, where the satellites are unit clauses and the nucleus is reduced down to a single literal, i.e., another unit clause.

Prover9 implements this as the `hyper` tactic. The output language divides files in several sections, one of which contains proofs presented as justifications: a sequence of clauses, each derived from the starting clauses or by previous derivations in the chain. Inferences in each step of the justification are themselves lists of tactics: exactly one primary tactic, possibly followed by a number of secondary tactics. Hyperresolution is one of the primary tactics, and for simplicity we will consider its treatment in isolation. It will become clear that sequences of secondary steps follow an analogous compositional pattern.

An example of hyperresolution step is `hyper(59, b, 47, a, c, 38, a)` where clauses are referenced by Arabic numerals and literals within a clause by letters: `a`, `b`, `c`... Though represented by a plain list, it is to be interpreted as the nucleus clause followed by a sequence of triples, each specifying a satellite clause and the literals that are involved to produce the next clause in the hyperresolution chain. Thus, in the example, 59 is the nucleus; applying binary resolution to its second literal and the first literal of clause 47 produces a new clause; and applying binary resolution again, this time between the third literal of the new clause and the first literal of 38, produces the final result.

Ignoring labels and secondary steps in Prover9 syntax, an instance of the hyperresolution rule is expressed as follows.

```
Clause Formula. [hyper(Nucleus,
                       First1,Satellite1,Second1,
                       ...,
                       FirstN,SatelliteN,SecondN)]
```

For the translation to our extension of TPTP, we provide the logic program that implements the procedure and define the mapping.

```
thf(prover9_hyperresolution, semantics, Formula, calculus(hol)).
```

Here `Formula` defines the logical semantics of hyperresolution recursively, in terms of the same generic (binary) resolution procedure that was used to model the tactic in Vampire.

```
∀ S1, S2, R:
hyperresolution([S1, S2], R)
  ⇐ resolution(S1, S2, R).
∀ S1, S2, Ss, R:
hyperresolution([S1, S2 | Ss], R)
  ⇐ ∃ R': resolution(S1, S2, R')
        ∧ hyperresolution([R' | Ss], R).
```

Insofar as the sequence of clauses and the expected final formula are known, we can ignore the triples passed as additional info and entrust the backtracking search mechanism to find an appropriate application of hyperresolution (assuming one exists). Consequently, the encoding drops the conjunct selection guidance given by Prover9 and represents a more general problem, solvable directly by the definition given here.

4.4 Object- to meta-level lifting of disjunction in LEO-II

As a final example, we consider a two-level logic tactic in the theorem prover LEO-II. In particular, we consider the `extcnf_or_pos` tactic, which is responsible for lifting a disjunction from the object level to the meta level of the logic [28]. The rule has the following definition:

$$\frac{C \vee [A \vee B]^{tt}}{C \vee [A]^{tt} \vee [B]^{tt}}$$

The tool expresses the application of this rule natively in TPTP syntax as follows.

```
thf(ClauseId, plain, Formula,
    inference(extcnf_or_pos, [status(thm)], [SourceId])).
```

It should be noted that atoms in LEO-II are labeled with either true or false using the TPTP notation `F = $true`. Once a substitution is applied, atoms can become more complex formulas. Concretely, this inference rule is used to translate the object-level disjunction into the clause-level one.

To provide the semantics of this rule, we use a higher-order logic formulation:

```
thf(leo2_extcnf_or_pos, semantics, Formula, calculus(hol)).
```

Here `Formula` supplies the following definition for the underlying semantics (using explicit quantifiers).

```
∀ ClauseId, SourceId:
extcnf_or_pos(ClauseId, SourceId)
  ⇐ (((∀ C: C ⇔ C = ⊤) ∧ SourceId) ⇒ ClauseId)
```

It is easily seen that using the additional axiom one can easily use any calculus for higher-order logic to prove this normalization rule.

5 Discussion and conclusion

Even when we restrict our attention to the community of resolution theorem provers, there are several different approaches to proof certification. Sutcliffe [29] proposed using the proof derivations in the TSTP library as a skeleton, which one can use to reconstruct a proof (possibly with the help of theorem provers). The Dedukti proof certifier [4] is a universal proof certifier which was successfully used to certify proofs of the iProver resolution theorem prover [14]. The proof certifier closest to the approach presented in this paper is that of the system `checkers` [5], which uses logic programming in order to encode inference rule semantics and to reconstruct proofs. `checkers` has been used to partially certify E's [27] proofs. While the first method is based on using theorem provers for filling in the missing semantics in TPTP proofs, the latter two systems are stem from a concrete effort to denote the semantics of different theorem provers using deterministic and non-deterministic approaches, respectively. This effort is normally made by a different team from that which implemented the theorem prover and which has the deepest knowledge about the actual semantics of its calculus.

The approach which was taken in this paper tries to make this effort easier and more accessible to the implementers of theorem provers. First, the language used to denote the semantics is well known to the implementers as it is already used to input problems and to output proofs. Second, unlike the last two systems mentioned, the implementers have a high degree of flexibility to define the semantics and are not restricted by external notions such as efficient or effective translations. This indeed put at risk the ability to mechanize these definitions into an actual certifier for the system but as mentioned in the paper, the parts which cannot be mechanized as given can, at least, be used to bring mechanization closer with some further help, for example, by the certification team.

The aim of this proposal is to convince the implementers of theorem provers that even semi-formal semantics, which can easily be defined using the approach presented, are useful for the purpose of full certification of their provers. The implementers can thus control the effort required of them in order to generate the semantics. The examples given in this paper range from the minimal effort of specifying a simple set of axioms to the greater effort of defining a full translation. While the second can be used efficiently by any of the two systems described at the beginning of this section, the first method requires only minimal additional effort in order to be used for proof reconstruction by a system like `checkers`.

In conclusion, TPTP can serve as a format for specifying the semantics of proofs for various degrees of concreteness. By using the same format for both problems, proofs and semantics, implementers are encouraged to consider the semantics as part of the implementation effort. This effort can both serve as documentation of the internal calculus and as an implementation of the semantics which can be later used for proof checking.

Acknowledgments. This work has been funded by the ERC Advanced Grant ProofCert.

References

[1] Christoph Benzmüller, Florian Rabe, and Geoff Sutcliffe. THF0–the core of the TPTP language for higher-order logic. In *Automated Reasoning*, pages 491–506. Springer, 2008.

[2] Mathieu Boespflug, Quentin Carbonneaux, and Olivier Hermant. The λΠ-calculus modulo as a universal proof language. In David Pichardie and Tjark Weber, editors, *Proceedings of PxTP2012: Proof Exchange for Theorem Proving*, pages 28–43, 2012.

[3] Sascha Böhme and Tobias Nipkow. Sledgehammer: judgement day. In *Automated Reasoning*, pages 107–121. Springer, 2010.

[4] Guillaume Burel. A shallow embedding of resolution and superposition proofs into the λΠ-calculus modulo. In J. C. Blanchette and J. Urban, editors, *Third International Workshop on Proof Exchange for Theorem Proving (PxTP 2013)*, volume 14 of *EPiC Series*, pages 43–57. EasyChair, 2013.

[5] Zakaria Chihani, Tomer Libal, and Giselle Reis. The proof certifier checkers. In Hans De Nivelle, editor, *Proceedings of the 24th Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX)*, number 9323 in LNCS, pages 201–210. Springer, 2015.

[6] Zakaria Chihani and Dale Miller. Proof certificates for equality reasoning. To appear in the Post-proceedings of LSFA 2015: 10th Workshop on Logical and Semantic Frameworks, with Applications. Natal, Brazil. Draft dated 28 October 2015.

[7] Zakaria Chihani, Dale Miller, and Fabien Renaud. Foundational proof certificates in first-order logic. In Maria Paola Bonacina, editor, *CADE 24: Conference on Automated Deduction 2013*, number 7898 in LNAI, pages 162–177, 2013.

[8] Alonzo Church. A formulation of the Simple Theory of Types. *J. of Symbolic Logic*, 5:56–68, 1940.

[9] Cvetan Dunchev, Ferruccio Guidi, Claudio Sacerdoti Coen, and Enrico Tassi. A fast interpreter for λProlog. In *LPAR-20: Logic Programming and Automated Reasoning, International Conference*, 2015. To appear.

[10] Amy Felty and Dale Miller. Specifying theorem provers in a higher-order logic programming language. In *Ninth International Conference on Automated Deduction*, number 310 in LNCS, pages 61–80, Argonne, IL, May 1988. Springer.

[11] Christian G Fermüller, Alexander Leitsch, Ullrich Hustadt, and Tanel Tammet. Resolution decision procedures. In *Handbook of automated reasoning*, pages 1791–1849. Elsevier Science Publishers BV, 2001.

[12] Joe Hurd. First-order proof tactics in higher-order logic theorem provers. *Design and Application of Strategies/Tactics in Higher Order Logics, number NASA/CP-2003-212448 in NASA Technical Reports*, pages 56–68, 2003.

[13] Cezary Kaliszyk and Josef Urban. Proch: Proof reconstruction for hol light. In *Automated Deduction–CADE-24*, pages 267–274. Springer, 2013.

[14] Konstantin Korovin. iprover–an instantiation-based theorem prover for first-order logic (system description). In *Automated Reasoning*, pages 292–298. Springer, 2008.

[15] Donald W Loveland. Mechanical theorem-proving by model elimination. *Journal of the ACM (JACM)*, 15(2):236–251, 1968.

[16] W. McCune. Prover9 and mace4. `http://www.cs.unm.edu/~mccune/prover9/`, 2005–2010.

[17] William McCune and Olga Shumsky. Ivy: A preprocessor and proof checker for first-order logic. In *Computer-Aided reasoning*, pages 265–281. Springer, 2000.

[18] Dale Miller. Proofcert: Broad spectrum proof certificates. An ERC Advanced Grant funded for the five years 2012-2016, February 2011.

[19] Dale Miller. A proposal for broad spectrum proof certificates. In J.-P. Jouannaud and Z. Shao, editors, *CPP: First International Conference on Certified Programs and Proofs*, volume 7086 of *LNCS*, pages 54–69, 2011.

[20] Dale Miller and Gopalan Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, June 2012.

[21] Gopalan Nadathur and Dustin J. Mitchell. System description: Teyjus — A compiler and abstract machine based implementation of λProlog. In H. Ganzinger, editor, *16th Conf. on Automated Deduction (CADE)*, number 1632 in LNAI, pages 287–291, Trento, 1999. Springer.

[22] Lawrence C Paulson and Kong Woei Susanto. Source-level proof reconstruction for interactive theorem proving. In *Theorem Proving in Higher Order Logics*, pages 232–245. Springer, 2007.

[23] Alexandre Riazanov and Andrei Voronkov. The design and implementation of VAMPIRE. *AI Communications*, 15(2-3):91–110, 2002.

[24] Tom Ridge and James Margetson. A mechanically verified, sound and complete theorem prover for first order logic. In *TPHOLs*, volume 3603, pages 294–309. Springer, 2005.

[25] G. Robinson and L. Wos. Paramodulation and theorem-proving in first-order theories with equality. In Jörg H. Siekmann and Graham Wrightson, editors, *Automation of Reasoning*, Symbolic Computation, pages 298–313. Springer Berlin Heidelberg, 1983.

[26] J. A. Robinson. A machine-oriented logic based on the resolution principle. *JACM*, 12:23–41, January 1965.

[27] Stephan Schulz. System description: E 1.8. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 735–743. Springer, 2013.

[28] Nik Sultana and Christoph Benzmüller. Understanding LEO-II's proofs. In *IWIL@ LPAR*, pages 33–52, 2012.

[29] Geoff Sutcliffe. Semantic derivation verification: Techniques and implementation. *International Journal on Artificial Intelligence Tools*, 15(06):1053–1070, 2006.

[30] Geoff Sutcliffe. The szs ontologies for automated reasoning software. In *LPAR Workshops*, volume 418, 2008.

[31] Geoff Sutcliffe. The tptp problem library and associated infrastructure. *Journal of Automated Reasoning*, 43(4):337–362, 2009.

[32] Geoff Sutcliffe and Christian Suttner. The state of casc. *AI Communications*, 19(1):35–48, 2006.

A Method to Simplify Expressions: Intuition and Preliminary Experimental Results

Baudouin Le Charlier[1] and Mêton Mêton Atindehou[2]

[1] Université catholique de Louvain,
Louvain-la-Neuve, Belgique
baudouin.lecharlier @uclouvain.be
[2] Université catholique de Louvain,
Louvain-la-Neuve, Belgique
meton.atindehou@uclouvain.be

Abstract

We present a method to simplify expressions in the context of a formal, axiomatically defined, theory. In this paper, equality axioms are typically used but the method is more generally applicable. The key idea of the method is to represent large, even infinite, sets of expressions[1] by means of a special data structure that allows us to apply axioms to the sets as a whole, not to single individual expressions. We then propose a bottom-up algorithm to finitely compute theories with a finite number of equivalence classes of equal terms. In that case, expressions can be simplified (i.e., minimized) in linear time by "folding" them on the computed representation of the theory. We demonstrate the method for boolean expressions with a small number of variables. Finally, we propose a "goal oriented" algorithm that computes only small parts of the underlying theory, in order to simplify a given particular expression. We show that the algorithm is able to simplify boolean expressions with many more variables but optimality cannot be certified anymore.

1 Introduction

Algorithms to simplify expressions often start by simplifying sub-expressions. Then they attempt to apply a number of simplification rules to the whole already partly simplified expression. Very often the simplification rules are restricted to rules that are guaranteed to produce a simpler (shorter) expression. This ensures that the simplification process is fast. However, in many situations it is necessary to first compute a more complicated expression in order to get a satisfactorily simplified one. For example, let us consider the boolean expression $a + ba$. Its sub-expressions are already simplified. To complete the simplification using basic axioms of the boolean calculus, we must write a sequence of equalities such as: $a + ba = 1a + ba = (1 + b)a = 1a = a$. At least the expression $1a + ba$ is more complicated than the initial one. And it is the key for the simplification.

In this paper, we propose an approach to simplification where we basically compute all expressions that are equivalent (i.e., equal with respect to a given theory) to an expression to be simplified. And we pick the simplest one (or possibly all simplest ones) at the end. The key idea for making that possible is to apply rules (i.e., axioms) to (large) sets of expressions instead of single ones. To do so we introduce a data structure that allows us to compactly represent such sets of terms and we show how it can be used to compute (representations of) some theories in such a way that a given expression can be mapped to its equivalence class in linear time. So simplifiying the expression amounts to pick a simplest expression in the equivalence class. The approach is not applicable to all theories but it can be adapted to theories that are not finitely

[1] We use the words "term" and "expression" as synonymous.

representable to derive a simplification algorithm that gives good results in some interesting situations.

The rest of this paper is organised as follows. In Section 2, we introduce the data structure that is used to represent sets of equal terms. This data structure can be related to the congruence closure method used in [8, 1] but it allows us to represent sets of terms in a much more compact way. In Section 3, we describe a bottom-up algorithm that computes (a representation of) the equivalence classes of terms that can be built from a set of equality axioms and a set of initially given terms. The algorithm (theoretically) terminates if and only if the number of these sets is finite. But the sets themselves can of course be infinite. We illustrate the functioning of the algorithm on a simple theory. In Section 4, we use the bottom-up algorithm together with a set of axioms for the boolean calculus to compute a complete representation of all boolean expressions using at most three variables. Based on this representation we show that any boolean expression can be simplified in linear time. It is also possible to use this representation to write down all minimal boolean expressions using three variables, according to various size notions. In Section 5, we propose a different but related algorithm to simplify expressions in the context of larger theories. Minimization cannot be guaranteed anymore. We show that the algorithm is able to simplify boolean expressions with many variables. In Section 6, we relate our work to the literature. Finally, Section 7 provides the conclusion and a list of extensions and improvements that we plan to make in the future.

All program runs presented in this paper are executed on a MacBook Pro 2.4GHz (Intel Core i5, 4Gb RAM) using Mac OS X 10.6.8. The programs are written in Java, and compiled and executed using the basic `javac` and `java` commands without any option. Timings are measured using the method `System.nanoTime()`.

2 Structures and Sets of Structures

To represent terms and sets of terms, we use "objects" that we simply call *structures*. A structure is of the form $f(i_1, i_2) : i$ where f is a function symbol, and i_1, i_2 and i are so-called *set of structures identifiers*. It is convenient to use natural numbers as identifiers and it is done so in the following and in our implementation but, from a "theoretical" standpoint, identifiers could be chosen from any infinite set I. We call $f(i_1, i_2)$, the *key* of the structure. The identifier i, is the identifier of the *set of structures* to which the structure belongs. Thus, at a given time, we consider a finite "collection" of structures that is partitioned in a finite number of sets of structures. For convenience, we only use binary keys and we "simulate" constant and unary function symbols by binary ones that are applied to the special identifier i_{null} which is the identifier of a conventional "dummy" set of structures. This can be related to the Currying and flattening method of [9] and to the transformation to directed graph of out-degree 2 of [4]. When we display structures, however, we use a simplified notation with constant and unary symbols.

The meaning of structures and sets of structures is defined as follows. Given n sets of structures $E_1, \ldots, E_n$, those sets denote together the smallest sets of terms $T_1, \ldots, T_n$ such that a term $f(t_1, t_2)$ belongs to T_i whenever E_i contains the structure $f(i_1, i_2) : i$ and t_1, t_2 belong to T_{i_1} and T_{i_2}, respectively.

As a simple example, let us consider the case of three structures partitioned into two sets of structures:

$$E_1 = \{f(1, 2) : 1, a : 1\} \qquad E_2 = \{b : 2\}$$

These two sets of structures denote the sets of terms:

$$T_1 = \{a, f(a, b), f(f(a, b), b), \ldots, f(\ldots f(a, b) \ldots, b), \ldots\} \qquad T_2 = \{b\}$$

We observe that the set T_1 is infinite. The sets of structures E_1 and E_2 constitue what is computed by our method when it is given the equality $a = f(a, b)$ or, maybe more intriguingly, the two equalities $f(f(f(a, b), b), b) = a$ and $f(f(f(f(f(a, b), b), b), b), b) = a$.

2.1 Operations over Sets of Structures

There are two main operations over sets of structures: *toSet* and *unify*.

The operation *toSet* takes as input a term $f(t_1, t_2)$ and returns the identifier i of a set of structures to which the term belongs.[2] (More exactly the term belongs to the set of terms T_i denoted by E_i.) The operation first recursively computes the identifiers i_1 and i_2 corresponding to t_1 and t_2. Then there are two cases. Either a structure $f(i_1, i_2) : i$ already exists for some i. Then the identifier i is returned. Otherwise, a new set identifier i is chosen and a new set $E_i = \{f(i_1, i_2) : i\}$ is created. Finally, the identifier i is returned.

The operation *unify* puts two sets of structures together, assuming that the terms denoted by the two sets are all equal, and taking into account the fact that two terms that have equal corresponding subterms are equal as well (function congruence [8, 1]). It uses two sub-operations: *substitute* and *normalize*.

- The operation *substitute* takes as input two set of structures identifiers i and j. It removes, from all sets of structures, all structures that involve j (i.e., structures of one of the three forms $f(i_1, i_2) : j$ or $f(j, i_2) : i'$ or $f(i_1, j) : i'$, for some i_1, i_2, i') and it substitutes to them possibly new structures obtained by replacing j by i in the removed ones. The set E_j is then discarded.
- The operation *normalize* takes into account the fact that different structures with the same key can result from the operation *substitute*. Since these structures denote the same non empty set of terms, the sets of structures to which they belong must be recursively unified. Thus, the operation *normalize* collects all pairs of distinct structures with identical keys and apply the operation *unify* to the identifiers of the sets of structures to which they belong.
- The operation *unify* simply consists of executing *substitute* followed by *normalize*.

It is important to say that both operation *toSet* and *unify* can be implemented efficiently by means of adequate data structures: mainly, a hash table for keys and three doubly linked lists for the identifiers i_1, i_2, and i used by the structures. (In fact, there are three such lists for each set identifier.) However, we do not give the details of the implementation here.

Another important fact to note is that the operation *unify* always reduces the number of structures and the number of sets of structures that are "currently living", while, at the same time, it increases the set of all terms that are represented by the sets of structures. The more the sets of structures are reduced the more the sets of represented terms are increased. This is a key observation to understand the power of our method.

3 Bottom-up Algorithm

We now describe, mainly by showing how it works on examples, a bottom-up algorithm that aims at computing a set of sets of structures that describes exactly the equivalence classes of terms that can be built from a set of equality axioms and a set of initially given terms. Thus, the algorithm, at the same time, builds a representation of all terms that can be constructed from the initially given terms and the function symbols used by the axioms, and classifies them into the equivalence classes determined by the axioms. The algorithm terminates if and only

[2] This way of representing terms can be related to the term banks of [10].

if there are only finitely many such equivalence classes. (Of course, in practice, it may fail to terminate because of lack of memory or it may take too much time.)

To fix ideas, we present first an example of an execution of the algorithm. Consider the following simple set of axioms:

```
x.x = x ;          x.y = y.x ;          (x.y).z = x.(y.z) ;
```

This set of axioms states that the binary operation `.` is idempotent, commutative and associative. The letters `x`, `y`, `z` are thus universally quantified *variables*. We simply use the last letters of the alphabet as variables. Others characters such as `a`, `b`, `f`, `0`, `!`, `+`, ... stand for constant and function symbols. Assuming that these axioms are put in a file called `Simple.txt`, let us consider the following run of the algorithm.

```
java BottomUpV3 Simple '(ab)'
================================================
Number of sets of structures : 3
Total number of structures : 11
Intermediate time              : 9.32E-4 sec
Number of created sets of structures : 12
================================================
Total time                      : 0.001659 sec
Number of created sets of structures : 12
================================================
?t
----------------------------------------------------------------
Set of structures no 1 [id = 1] [size = 1]  [tcS = 2]
----------------------------------------------------------------
Minimal term : a
----------------------------------------------------------------
.(1, 1):1 [size = 3] [key = 433592]
a:1 [size = 1] [key = 65]
Number of structures : 2
----------------------------------------------------------------
Set of structures no 2 [id = 2] [size = 1]  [tcS = 3]
----------------------------------------------------------------
Minimal term : b
----------------------------------------------------------------
.(2, 2):2 [size = 3] [key = 867170]
b:2 [size = 1] [key = 66]
Number of structures : 2
----------------------------------------------------------------
Set of structures no 3 [id = 3] [size = 3]  [tcS = 4]
----------------------------------------------------------------
Minimal term : ba
----------------------------------------------------------------
.(3, 2):3 [size = 5] [key = 300630]
.(2, 3):3 [size = 5] [key = 867297]
.(1, 3):3 [size = 5] [key = 433846]
.(3, 3):3 [size = 7] [key = 300757]
.(3, 1):3 [size = 5] [key = 300503]
.(2, 1):3 [size = 3] [key = 867043]
.(1, 2):3 [size = 3] [key = 433719]
Number of structures : 7
```

```
================================================
Number of sets of structures : 3
Total number of structures : 11
?s
>aaaaaaaaababababbbbbbbbbabababbbbbbbbbbbbbbbbbab
Simplified expression : ab
Simplification time : 1.01E-4 sec
================================================
```

- The first line launches the algorithm with the file of axioms `Simple.txt` and the initial term $a.b$. The program stops after 0.001659 seconds and displays that 12 sets of structures have been created during the whole execution.
- Then, the command `t` allows us to display the sets of structures that are obtained as the final result of the bottom-up algorithm. We see that there are only three different sets of structures representing the terms that are respectively equals to a, b, and $a.b$. The first set contains the two structures $.(1,1) : 1$ and $a : 1$ because a term t is equal to a if and only if it is just a itself or if it is of the form $t_1.t_2$ where both t_1 and t_2 are equal to a. The third set contains seven different structures describing all possible forms of the terms that are equals to $a.b$. For instance, the structure $.(2,3) : 3$ stands for all terms $t_1.t_2$ such that t_1 is equal to b and t_2 is equal to $a.b$.
- Finally, the command `s` allows us to enter expressions to be simplified. We see that a rather long expression quickly simplifies to $a.b$. Note that the operation `.` is denoted by simple juxtaposition here. Moreover, we do not use parentheses since we assume that the operation associates to the left. The simplification is obtained by first applying the operation *toSet* to the introduced expression and by afterwards displaying a minimal term chosen among all those that are represented by the set of structures (whose identifier is) returned by *toSet*.

Let us now explain how the bottom-up algorithm works. Basically, it applies all axioms to all sets of structures by replacing variables by sets identifiers and unifying the sets corresponding to the left-hand and right-hand sides of the axioms. Axiom application uses a variant (generalization) of the *toSet* operation that takes as an additional argument a mapping from axiom variables to set of structures identifiers. To apply all axioms to all sets of structures, we simply generate all sequences of identifiers whose length is at most the maximum number of variables in an axiom, and, for each such sequence, we apply all axioms using the corresponding number of variables. Let us show how it works on the previous example. We display all generated sequences as well as all axiom applications.

```
java BottomUpV3 Simple '(a.b)' lx
1
[x<1>x<1>: 4  =  x<1>: 1] ==>
    aa = a  [BU: 1[a]]
1 1
[x<1>y<1>: 1  =  y<1>x<1>: 1] ==>
    aa = aa  [BU: 1[a]]
1 1 1
[x<1>y<1>z<1>: 1  =  x<1>(y<1>z<1>): 1] ==>
    aaa = a(aa)  [BU: 1[a]]
2
[x<2>x<2>: 4  =  x<2>: 2] ==>
    bb = b  [BU: 2[b]]
```

```
2 1
[x<2>y<1>: 4  =  y<1>x<2>: 3] ==>
    ba = ab  [BU: 3[ba]]
2 1 1
[x<2>y<1>z<1>: 4  =  x<2>(y<1>z<1>): 3] ==>
    baa = b(aa)  [BU: 3[ba]]
2 1 2
[x<2>y<1>z<2>: 4  =  x<2>(y<1>z<2>): 5] ==>
    bab = b(ab)  [BU: 4[bab]]
2 2
[x<2>y<2>: 2  =  y<2>x<2>: 2] ==>
    bb = bb  [BU: 2[b]]
2 2 2
[x<2>y<2>z<2>: 2  =  x<2>(y<2>z<2>): 2] ==>
    bbb = b(bb)  [BU: 2[b]]
3
[x<3>x<3>: 5  =  x<3>: 3] ==>
    (ba)(ba) = (ba)  [BU: 3[ba]]
3 1
[x<3>y<1>: 3  =  y<1>x<3>: 5] ==>
    (ba)a = a(ba)  [BU: 3[ba]]
3 1 1
[x<3>y<1>z<1>: 3  =  x<3>(y<1>z<1>): 3] ==>
    (ba)aa = (ba)(aa)  [BU: 3[ba]]
3 1 2
[x<3>y<1>z<2>: 4  =  x<3>(y<1>z<2>): 3] ==>
    (ba)ab = (ba)(ab)  [BU: 3[ba]]
3 1 3
[x<3>y<1>z<3>: 3  =  x<3>(y<1>z<3>): 3] ==>
    (ba)a(ba) = (ba)(a(ba))  [BU: 3[ba]]
3 2
[x<3>y<2>: 3  =  y<2>x<3>: 3] ==>
    (ba)b = b(ba)  [BU: 3[ba]]
3 2 2
[x<3>y<2>z<2>: 3  =  x<3>(y<2>z<2>): 3] ==>
    (ba)bb = (ba)(bb)  [BU: 3[ba]]
3 2 3
[x<3>y<2>z<3>: 3  =  x<3>(y<2>z<3>): 3] ==>
    (ba)b(ba) = (ba)(b(ba))  [BU: 3[ba]]
3 3
[x<3>y<3>: 3  =  y<3>x<3>: 3] ==>
    (ba)(ba) = (ba)(ba)  [BU: 3[ba]]
3 3 3
[x<3>y<3>z<3>: 3  =  x<3>(y<3>z<3>): 3] ==>
    (ba)(ba)(ba) = (ba)((ba)(ba))  [BU: 3[ba]]
================================================
Number of sets of structures : 3
Total number of structures : 11
Total time                      : 0.010333 sec
Number of created sets of structures : 12
================================================
```

We see that the first generated sequence simply is `1`. The axiom `x.x = x` is then applied to it.

The axiom is depicted as `[x<1>x<1>: 4 = x<1>: 1]` to indicate that the set of structures identifier `1` is substituted to `x`: the operation *toSet* is applied to the term `x.x` where `x` is replaced by `1`. It creates a new set of structures $E_4 = \{.(1,1) : 4\}$, which is then unified with $E_1 = \{a : 1\}$. After unification the set E_4 is discarded and we get $E_1 = \{.(1,1) : 1, a : 1\}$. The next line `aa = a [BU: 1[a]]` provides additional information on the effect of the axiom: the terms `a.a` and `a` are now considered equal and they belong to the set T_1 of all terms represented by E_1. A minimal term of T_1 is `a`.

The following lines show that the commutativity and associativity axioms are now trivially satisfied by the terms in T_1: no modification is made to the current sets of structure. The next three lines are similar but afterwards various axioms involving both `a` and `b` are applied. Their application progressively fills the set E_3 with the six new structures depicted in the previous execution trace of the program. We encourage the reader to find out at which axiom application each final structure is exactly created. Finally, we observe that not all sequences of numbers involving `1`, `2`, and `3` have been generated. This is because they are not all necessary due to the commutativity and associativity of the `.` operation.

This first example is particularly simple because no new set of structures created by application of the axioms remains after considering the three initial sets of structures. The program stops because the current sets of structures definitely verify the axioms. Applying them again would not create any new structure. Of course this is not true in general. Here is another run of the program in the slightly more complicated case of three constants `a`, `b`, `c`. We display a different kind of information and we show only a small part of the trace.

```
java BottomUpV3 Simple '(a.b.c)' fi
================================================
Number of sets of structures : 15
Total number of structures : 50
Number of created sets of structures : 51
================================================
Normalize .(5, 4): [5 |--> 15]  [size = 7] : we have
    bac = bacc
  because
    bac = (b(ac))c [in 5]
  and
    bacc = (b(ac))c [in 15]
Normalize .(5, 5): [5 |--> 16]  [size = 11] : we have
    bac = bacbac
  because
    bac = (b(ac))(b(ac)) [in 5]
  and
    bacbac = (b(ac))(b(ac)) [in 16]
This sequence is no longer valid : 11 2 5
================================================
Number of sets of structures : 9
Total number of structures : 61
Number of created sets of structures : 80
================================================
This sequence is no longer valid : 12 2 4
================================================
Number of sets of structures : 7
Total number of structures : 52
Total time                      : 0.009743 sec
```

```
Number of created sets of structures : 83
================================================
```

Looking at this trace we can make the following observations.

- This time, the algorithm does not stop after applying the axioms to all initial sets of structures. It iterates three times. The first iteration creates 10 new sets of structures to which –roughly speaking– the axioms are applied at the second iteration. The second iteration reduces the number of sets but it increases the number of structures and even creates a few new sets. The last iteration finishes the work.
- The trace shows two cases where the operation *normalize* is needed: at some point of the execution two structures with the same key exist, namely $.(5,4) : 5$ and $.(5,4) : 15$. Therefore, the sets E_5 and E_{15} are unified. An example of a term represented by the structure is added as a "comment": `(b(ac)).c`. Since this term is equal to `(ba).c` in T_5 and to `((ba)c).c` in T_{15}, the unification of the two sets proves –in particular– that `(ba).c` = `((ba)c).c`.
- The fact that sets of structures are removed by the operation *unify* (and thus also by *normalize*) complicates the generation of all sequences of set of structures identifiers: an identifier may "disappear" while we are generating sequences using it. Two examples of this situation are shown in the trace. In such a case, we must be careful to continue with the appropriate "next" sequence. But we omit the details here.

A last important remark is the following. When we unify two sets of structures E_i and E_j, we must choose to keep one identifier and to remove the other one. Experiments show that it is of utmost importance to keep the older one. Consider the previous example. It takes 0.01 seconds to be executed and it creates 83 sets of structures (of which most are discarded afterwards by the operation *unify*, of course). If we choose to "put the older set into the more recent one", the execution takes 0.25 seconds and it creates 627 sets of structure. (Of course, the final sets of structures are equivalent.) In less simple situations such as those of the next section, the wrong choice is simply not usable and the program runs out of memory. We explain the difference as follows: when a recent set of structures is unified with an old one, it is often the case that the old set has already been involved in many –if not all– axiom applications using its identifier; thus, removing the recent set of structures makes it disappear without using it in any axiom application. With the opposite policy however there is a high probability that all axioms will be applied later on to the more recent set of structures, which is useless since this has been implicitly already done by unifying it with the older one.

4 Simplifying Boolean Expressions

In this section, we show how the bottom-up algorithm of the previous section behaves for simplifying boolean expressions. We use the following set of axioms for the "boolean calculus".

```
  x 0 = 0 ;        !!x = x ;              x + y = y + x ;
x + 0 = x ;         xy = yx ;       (x + y) + z = x + (y + z) ;
x + 1 = 1 ;        xyz = x(yz) ;         x + yz = (x + y)(x + z) ;
 x !x = 0 ;                              !(xy) = !x + !y ;
```

Assuming that the file `Boole.txt` contains exactly this set of axioms, let us consider the following run of our program.

```
java BottomUpV3 Boole '(abc)'
```

```
================================================
Number of sets of structures : 144
Total number of structures : 307
Intermediate time              : 0.00691 sec
Number of created sets of structures : 308
================================================
Number of sets of structures : 859
Total number of structures : 4642
Intermediate time              : 0.207848 sec
Number of created sets of structures : 6804
================================================
Number of sets of structures : 783
Total number of structures : 110052
Intermediate time              : 1.776162 sec
Number of created sets of structures : 216105
================================================
Number of sets of structures : 281
Total number of structures : 134053
Intermediate time              : 3.732329 sec
Number of created sets of structures : 338710
================================================
Number of sets of structures : 256
Total number of structures : 131333
Total time                      : 3.73564 sec
Number of created sets of structures : 338728
================================================
?s
>abc + ab!c + a!bc + a!b!c + !abc + !ab!c + !a!bc + !a!b!c
Simplified expression : 1
Simplification time : 5.8E-5 sec
================================================
>abc  + a!b!c + !abc + !a!bc + !a!b!c + !a!b!c
Simplified expression : !(ac + b) + bc
Simplification time : 5.1E-5 sec
================================================
>!((b+!b)!b!(c!(cc!a!a+!b))+(!b+a)(!c+c+bca)+b+c+!((cc+!b)!(a+!b)(a+!!b)+!b+a+a))+!ccb+
(a!c(c+a!!b+c)a+(b+!a)a!(cb!bb)((b+c)!!b(c+a)+c!b+a)(ca+c!c))!b(a+c)+(a!c+c+!b+!acbc)c+
(ba+cc+!c+!a+!(c+c))(!!(c+b+b+a)+c+!b)!(a(a+!a)!(!a+!!(aa)))(c+(a+b+bc+a+!b+ca+c)!a(!c+
!c+a)(!a+a)+!!((!c+ba(!c+b))!c)+!b+b+!c)(b+b)(b+c+c+c+c(b+c)+b+!((a+a)a+!b)+!c+!b+!(c+a
+!a)+(!(ba+!a!c)+ca+!c+(ac+c)!(a!a)+!bc(a+!c))!(!c+!!b)(a+!a+b+c+b+!c))+!(!(bb(b+a)(ac+
a!ba)b(!c+!(!(cb)bb))!c(a+!!(bc)))+!(!((!a!c+!((b+!c)b)(bb+bb+c))b!(b!c)(!a+c+ba)c(b+a+
(b+c)cb)b+(b+b+b+b+c+c+a+c+!c+b+(a+a)!c+b)(b+aba(baa+a!a!c((c+a)!b+c+b)+(!c+ca(b+c))cb)
)+!(!(a!b((!b+a!c)a+a))(!!b+b!(b(!(babb)+(ca+c+!b+!(cc))cc)(!c+a+a)(c!a+!c)(b+!a!c))+!(
a!ab)+b+!c)+(!(b!a+!!a)+a+c)bab(!c+!a)(a+bb+!b!b)!a(c+c!b)+c+!(!a+ac+!b)))+(bc(a+b)(b+c
)!ab(!b+aa)c+!!b+b)!a!((!c+c!a+a+c!a)b)(ca(b+!b)+b)ac(a!c+a)))
Simplified expression : b + c
Simplification time : 4.77E-4 sec
================================================
```

We observe the following facts.

- The program stops after less than 4 seconds. Only five “iterations” are needed to get the final sets of structures.

- We get 256 sets of structures and 131333 structures in the end. This is what should be expected. Indeed, there are exactly $2^{2^3} = 256$ boolean functions with three arguments. Hence, the number of equivalence classes of boolean expressions with three letters a, b, c is 256. Moreover, any boolean expression of the form $t_1 + t_2$ must be represented by a structure $+(i_1, i_2) : i$. Since there are 256 possible values for i_1 and i_2, there must exist $256 \times 256 = 65536$ such structures. Similarly, there must exist 65536 structures of the form $.(i_1, i_2) : i$ and 256 structures of the form $!(i_1) : i$. Finally, there must be 5 structures corresponding to the five constants $a, b, c, 0, 1$. It gives us a total of 65536+65536+256+5 = 133133 structures.
- Having computed a complete representation of all boolean expressions not containing other letters than a, b, and c, we can use it to simplify any such expression. Three examples are shown in the trace. The simplified expressions are chosen by minimizing the size of a tree representation of the expression (or, equivalently, the number of characters of the expression written in polish notation). The timings show that the simplification is fast. They are also consistent with our claim that it is done in linear time.

5 Goal Oriented Algorithm

The bottom-up algorithm described in Section 3 is not applicable to "large theories" with many equivalent classes of equal terms. We now propose a different algorithm, which is not optimal anymore but which allows us to simplify expressions with more symbols. This algorithm is "goal-oriented" because it is "driven" by an initial expression to be simplified and, afterwards, by the intermediate simplifications of this initial one. Let us show a first run of this algorithm (for simplifying a boolean expression).

```
java GoalOriented
Enter an expression to be simplified : :
(b(e+f)+ca+!b+b+b+!b)(!a+d+!a)(dd+c)c!df
----------------------------------------------------------------------
Current reduced term : (b(e + f) + ac + !b + b + b + !b)(!a + d + !a)(c + d)c!df
----------------------------------------------------------------------
Current reduced term : 1(!a + d + !a)(c + d)c!df [size = 20]
----------------------------------------------------------------------
Current reduced term : (!ac + d)c!df [size = 13]
----------------------------------------------------------------------
Current reduced term : !d(!ac + d)f [size = 11]
----------------------------------------------------------------------
Current reduced term : f!(d + a)(c + d) [size = 10]
----------------------------------------------------------------------
Current reduced term : !(d + a)cf [size = 8]
======================================================================
Number of sets of structures : 10
Total number of structures : 156
Number of created sets of structures : 69808
Intermediate time : 2.298233 sec
======================================================================
```

The algorithm simplifies an expression of size 40 to an expression of size 8 in 2.3 seconds. The initial expression uses 6 different letters but the simplified one uses only four of them. Since it uses each letter and the operator `!` only once, we can guess that it is minimal.

The algorithm works as follows. It basically proceeds like the bottom-up algorithm by generating sequences of set of structures identifiers but it performs an axiom application only when the left hand-side of the axiom returns an identifier of a sub-expression of the current expression to be simplified. So it only keeps sets of structures that are "relevant" for simplifying the expression "at hand". It also maintains the minimal size of all expressions represented by the set of structures "containing" the current expression to be simplified. As soon as this minimal size decreases, all sets containing a structure corresponding to an expression of minimal size are marked. Then the execution of the algorithm resumes only considering identifiers of the marked sets. Using this strategy, the algorithm really concentrates on the current smallest expressions. The less interesting (non minimal) sets of structures are not immediately removed. They can become minimal later. But, of course, it may quickly happen that too much memory is used. In that case, the structures of the non minimal sets are freed first.

Here is another, more difficult example, with an expression of size 800 using 15 letters. We only show a small part of the trace.

```
java GoalOriented
Enter an expression to be simplified : :
(ei!b!!(a+!m)+lf+!!ialm+l)(!!(!((!k(l+c)!d+!a)(e+m!h)(!g+!j)(l+b)!hj)+k!i+!(fo)j(j+o+l)
(o+!i)+!(h+e!d)+!((!e(k+jj)+e+!i+!e+d(j+l))nh)e+((!g+h)o+(e+o!n+f+!e)!l((b+k)(!!g+!n)+
!h)(o+!d!a)+m+(l+!b)k+!l)g!n!km!!((a+b)(eg+f))+(!l!j+!(!jk+!h+!e))!(!bb(m+!b))(!b(!k+m
+b+!f)+!a))+!(!i!((f+eg)!!di+!d))!g(a!j+n)(!b+a+!d)!(!(!o+(!d+!f)(!!fk!a+h))+o)o(e!nbj
+f+fl)(!(ch+!f+g+!k!i)+!!n+b)(e+i)a!e(!(((!oi(!!h+k)+!!o)(hd+o!l+f)+n!l)!(!d+m+o)(!((a
h+(!l+!h)(g+n!f))dk)+f+!jd+!(k+a)+(h+!f)fe+i+!b))+(!i!b(n+i)+(d+i+ad)(g+(a+j)(c+c)))!(
(!m+!c+a+n!h(!c(j+o+h)+do(i+l)!!g!j!djj(j+e)))je(!(hbd)+!a!k+og)))+!ol!!nj(!(!c+ei(a!j
!k+!i+j))(!!!d+!l)i+!((b+hd)j))(!f+!!o(m+!l+!f)+a(o+g+o)+jc!m)+cl!d(!i+he+!(ja))(e+k)+
!((!(ejnk)+m+f)i!f!lgm(f+a+m)(!oeme+!k)!l(f(m+d)o+lh))+!(!a(f(!d+c!l)+(!b+ch)(f+m))+((
n+!f)!b+!!l+e+!c)d!k+n!j+a+!(!l+c)+!!e+(h+l)e!f)+k!f(!k+j+(h+!m)!l))
----------------------------------------------------------------------
Current reduced term : (ei!b!!(a + !m) + lf + !!ialm + l)(!!(!((!k(l + c)!d + !a)(e + m
!h)(!g + !j)(l + b)!hj) + k!i + !(fo)j(j + o + l)(o + !i) + !(h + e!d) + !((!e(k + jj)
+ e + !i + !e + d(j + l))nh)e + ((!g + h)o + (e + o!n + f + !e)!l((b + k)(!!g + !n) + !
h)(o + !d!a) + m + (l + !b)k + !l)g!n!km!!((b + a)(eg + f)) + (!l!j + !(!jk + !h + !e))
!(0(m + !b))(!b(!k + m + b + !f) + !a)) + !(!i!((f + eg)!!di + !d))!g(a!j + n)(!b + a +
 !d)!(!(!o + (!d + !f)(!!fk!a + h)) + o)o(e!nbj + f + fl)(!(ch + !f + g + !k!i) + !!n +
 b)(e + i)a!e(!(((!oi(!!h + k) + !!o)(hd + o + l + f) + n!l)!(!d + m + o)(!((ah + (!l +
 !h)(g + n!f))dk) + f + !jd + !(k + a) + (h + !f)fe + i + !b)) + (!i!b(n + i) + (d + i
+ ad)(g + (a + j)(c + c)))!((!m + !c + a + n!h(!c(j + o + h) + do(i + l)!!g!j!djj(j + e
)))je(!(hbd) + !a!k + og))) + !ol!!nj(!(!c + ei(a!j!k + !i + j))(!!!d + !l)i + !((b + h
d)j))(!f + !!o(m + !l + !f) + a(o + g + o) + jc!m) + cl!d(!i + he + !(ja))(e + k) + !((
!(ejnk) + m + f)i!f!lgm(f + a + m)(!oeme + !k)!l(f(m + d)o + lh)) + !(!a(f(!d + c!l) +
(!b + ch)(f + m)) + ((n + !f)!b + !!l + e + !c)d!k + n!j + a + !(!l + c) + !!e + (h + l
)e!f) + k!f(!k + j + (h + !m)!l)) [size = 797]
...
?st no
?run
...
size = 633 idList.size() = 318
size = 629 idList.size() = 316
======================================================================
Number of sets of structures : 315
Total number of structures : 6387
Number of created sets of structures : 2970747
```

```
Intermediate time : 37.291304 sec
================================================================
size = 605 idList.size() = 305
size = 604 idList.size() = 305
...
size = 346 idList.size() = 185
size = 21 idList.size() = 794
size = 13 idList.size() = 165
================================================================
Number of sets of structures : 260
Total number of structures : 6731
Number of created sets of structures : 10792179
Intermediate time : 85.492769 sec
================================================================
Iteration no : 2
?st nice
Current reduced term : l + (a + !m)i!be [size = 13]
?q
```

We see that the execution is not "fully automatic". The user may enter commands to guide it. The command `st no` tells the program not to display each intermediate simplified expression. The command `run` tells the program to apply the axioms without asking the user until all marked sets have been considered. At this time, a new iteration starts because some axioms that were not applicable at the first iteration may then become applicable. The algorithm may iterate many times and it may not terminate because, due to the finite amount of memory, some sets of structures can be removed and afterwards replaced by different ones so that some axioms remain applicable indefinitely. Hence, the more practical choice is to let the user decide whether the expression is simplified or not. In this example, the expression looks simple enough. Thus, we have stopped the program with the command `q`. Some other commands are available, notably to monitor memory usage, but we do not give more details here. Notice finally that the size of the expression has suddenly "jumped" from 346 to 13. Probably because a large sub-expression was found equal to 0 or to 1.

6 Related Work

The work presented in this paper can be related to the method of G .Nelson and D. C. Oppen proposed in [8] (see also [1], Chapter 9). They build upon the well-kown algorithm of M. J. Fischer and B. A. Galler [6] (see also [7], pages 353, 360–361, and [12]) to compute congruence classes of terms, as a tool to determine the satisfiability of a conjunction of literals. Our method, which has been designed independently, allows us to represent sets of terms in a much compact way because their method represents sets of equivalent terms by directed acyclic graphs (DAGs) while our sets of structures can be viewed as cyclic graphs. In particular, their method only permits them to handle finite sets of terms. Therefore, it is not possible, for example, to use their method to implement the bottom-up algorithm that we have presented in Section 3. On the contrary, our method can be used to solve the decision problem described in [8], and, in fact, more efficiently than they do. To support our claim, we have implemented an algorithm that "solves" an arbitrary number of equations between uninterpreted ground terms. The algorithm amounts to apply the operation *unify* to the list of pairs $i_1 = j_1, \ldots, i_n = j_n$ where the i_k and j_k are identifiers of sets of structures representing the terms in the equations. Since the operation *unify* does not maintain the initially given terms, we have to use an additional data

structure to map the initial terms to their corresponding sets of structures. This can be done with a simple array as in [6]. The resulting algorithm is similar to the algorithm presented in [4] and probably even more efficient since it works on more compact structures. It is easy to show that in the worst case its execution time is bounded by $O(m \log m)$ where m is the number of structures needed to represent the initially given terms. However, it seems to behave better in practice as shown by some experiments on which we report below.

First, we have "randomly" chosen a large term of 100000 symbols (i.e., 100000 symbols if the term is written in polish notation).[3] Then, we have represented this term as a collection of structures, using the operation *toSet* (see Subsection 2.1). This representation uses 36901 structures, each of them belonging to a different set of structures. Using the identifiers of these sets we have generated a sequence of 36901 pairs of identifiers to which the operation `unify` has been applied, one by one. In the first experiment, each identifier is used exactly twice in the equations and exactly once in the first 18450 equations (except one). The results of this experiment are depicted in the first table below. Column i provides the number of equations already "solved" at a given line. Column t provides the amount of time needed to solve these equations, in seconds. Columns $\sharp Set$ and $\sharp Struct$ respectively indicate the number of sets of structures and the number of structures existing at that stage. Columns U and N contain the numbers of sets of structures unification already done, and resulting directly from applying *unify* to a pair of sets of structures identifiers (U), or indirectly from using *normalize* (N). (See again Subsection 2.1.) Column T is just $U + N$. Column t/i gives the ratio of the time (measured in microseconds) by the number of equations already solved. Column dt/di is a kind of "derivative" of the time with respect to the number of already solved equations: we divide the difference between the current time and the previous one, by the number of equations executed between the current and the previous stage. Column T/t gives the number of set reductions by millisecond and dT/dt is the "derivative" of T with respect to t, computed similarly to dt/di. Finally, dS/dt is the "derivative" of the number of removed structures ($36901 - \sharp Struct$) with respect to the time.

i	t	$\sharp Set$	$\sharp Struct$	U	N	T	t/i	dt/di	T/t	dT/dt	dS/dt
0	0.0	36901	36901	0	0	0			0	0	0
2500	0.042	34336	36836	2500	65	2565	16.8	16.8	60	60	1
5000	0.065	31758	36758	5000	143	5143	13.1	9.3	78	109	3
7500	0.079	29166	36666	7500	235	7735	10.6	5.5	97	186	6
10000	0.096	26565	36565	10000	336	10336	9.6	6.7	107	153	5
12500	0.117	23925	36425	12500	476	12976	9.4	8.4	110	125	6
15000	0.13	21301	36301	15000	600	15600	8.6	5.1	119	202	9
17500	0.148	18615	36115	17500	786	18286	8.4	7.2	123	148	10
20000	0.162	15803	35802	19999	1099	21098	8.1	5.3	130	210	23
22500	0.482	5	48	21919	14977	36896	21.4	128.1	76	49	111
25000	0.482	2	30	21921	14978	36899	19.3	0.2	76	5	34
36901	0.483	2	30	21921	14978	36899	13.1	0.0	76	0	0

We can make the following observations: until $i = 20000$ (i.e. shortly after that set of structures identifiers have all been used in one call to *unify*), the algorithm behaves uniformly by reducing the number of sets while the number of structures remain almost stable. Also, the time complexity is better than linear. Then, suddenly, between $i = 20000$ and $i = 22500$, a lot of set reductions are made, mainly due to the operation *normalize*. This happens because most

[3] More information about these data can be found at
`https://www.dropbox.com/sh/3eo2flkb26767u9/AAALb6_msRndFTD7P3wJwFlta?dl=0`

sets of structures now contain more structures, increasing the probability of having different structures with the same key. Unifying the sets containing those structures creates new structures with the same key, which has a snowball effect. By looking to the third last column, we can see that the number of set unifications by unit of time is continuously increasing until the critical interval between $i = 20000$ and $i = 22500$ is met. Then, it decreases significantly but it is because most structures are discarded, (i.e., merged) at this stage. When $i > 22500$ almost nothing is left to do, so it is not useful to comment on the two last lines. Hence, our algorithm is very efficient to solve the satisfiability problem of [8]. We conclude by showing the results of a second experiment in which the list of equations $i_1 = j_1, \ldots, i_n = j_n$ is generated completely randomly, allowing every identifier to appear in the list an arbitrary number of times and in any position. We see that the results are similar but the "critical section" of the algorithm takes place earlier (which could have been anticipated). Also most set reductions are due to the operation *normalize* contrary to the first experiment. More equations are now trivially verified because the two identifiers involved in them at the beginning are mapped on the same one.

i	t	$\sharp Set$	$\sharp Struct$	U	N	T	t/i	dt/di	T/t	dT/dt	dS/dt
0	0.0	36901	36901	0	0	0			0	0	0
2500	0.032	34339	36839	2500	62	2562	13.0	13.0	78	78	1
5000	0.055	31740	36740	5000	161	5161	11.0	9.0	93	115	4
7500	0.074	29122	36622	7500	279	7779	9.9	7.9	103	132	5
10000	0.096	26463	36463	10000	438	10438	9.6	8.6	108	122	7
12500	0.116	23613	36113	12500	788	13288	9.3	7.8	114	144	17
15000	0.314	860	2128	14296	21745	36041	20.9	79.1	114	114	171
17500	0.316	422	1295	14386	22093	36479	18.0	0.8	115	205	391
20000	0.317	296	1041	14432	22173	36605	15.8	0.5	115	97	196
22500	0.319	163	678	14471	22267	36738	14.1	0.5	115	97	265
25000	0.32	73	384	14490	22338	36828	12.8	0.5	114	71	233
36901	0.323	12	140	14510	22379	36889	8.7	0.3	113	16	66

Finally, it can be stressed that the timings reported here are consistent with the timings reported for our bottom-up and goal oriented algorithms, which create much more structures and solve many more equations.

A lot of work has been devoted to the problem of simplifying boolean expressions, but most of the work has been done to simplify expressions written in disjunctive or in conjunctive normal form (see, e.g., [2]). The algorithms presented in this paper are not intended to compete with those methods but they are more general since they are applicable to many kinds of simplification problems. Basically, we used the boolean expression simplification problem mainly as a (significant) example of application. Very often, boolean expression simplification is used to better understand facts represented by the boolean expressions. The use of OBDDs to simplify boolean expressions is often advocated in that case (see, e.g. [3]). We have applied our method to analyze the so-called guards of a medical process model constructed by the authors of [3], with good results; but we have not compared our results with the use of OBDDs, yet.

Finally, parts of our work can be related to other areas such as rewriting systems, constraint programming or SAT-solving, to name only three, but we have not investigated those relations in great details yet. We will surely do so in the future, mainly to improve the efficiency and the applicability of our goal-oriented algorithm.

7 Conclusion and Future Work

We have presented a method to represent large sets of equivalent terms compactly, and we have shown how this representation can be used to solve interesting simplification problems. We have put the focus on boolean expression minimization and simplification but our method obviously is much more general. Therefore, we plan to use it to investigate other simplification problems such as the (very difficult) problem of simplifying regular expressions (see, e.g., [11]).

The algorithms that we have presented in Sections 3 and 5 can certainly be significantly improved, especially the goal oriented algorithm. We plan to improve them by using incremental techniques related to the Rete algorithm [5]. Several other avenues of research can be considered. Let us consider two of them. We can extend and improve our current syntax for writing axioms. Our simple language can be augmented with "meta predicates" to write more specific axioms, and to allow the writing of implications. A second (difficult) topic should be to specialize our main data structure (sets of structures) to take into account common properties of operations such as associativity and commutativity.

Acknowledgments

The authors wish to thank Charles Pecheur for useful discussions and for providing them with pointers to important related work. Comments from the reviewers are also greatly acknowledged.

References

[1] Aaron R. Bradley and Zohar Manna. *The calculus of computation - decision procedures with applications to verification*. Springer, 2007.

[2] Olivier Coudert. Two-level logic minimization: an overview. *Integration*, 17(2):97–140, 1994.

[3] Christophe Damas, Bernard Lambeau, and Axel van Lamsweerde. Generating process models in multi-view environments. In *Dependable Software Systems Engineering*, pages 105–127. 2015.

[4] Peter J. Downey, Ravi Sethi, and Robert Endre Tarjan. Variations on the common subexpression problem. *J. ACM*, 27(4):758–771, October 1980.

[5] Charles L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17–37, 1982.

[6] Bernard A. Galler and Michael J. Fischer. An improved equivalence algorithm. *Commun. ACM*, 7(5):301–303, 1964.

[7] Donald E. Knuth. *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997.

[8] Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *J. ACM*, 27(2):356–364, 1980.

[9] R. Nieuwenhuis and A. Oliveras. Proof-Producing Congruence Closure. In Giesl J, editor, *Proc. 16th International Conference on Rewriting Techniques and Applications (RTA-2004), Nara, Japan*, number 3467 in LNCS. Springer, 2005.

[10] Stephan Schulz. System Description: E 1.8. In Ken McMillan, Aart Middeldorp, and Andrei Voronkov, editors, *Proc. of the 19th LPAR, Stellenbosch*, volume 8312 of *LNCS*, pages 735–743. Springer, 2013.

[11] A. Stoughton. *Formal Language Theory: Integrating Experimentation and Proof*. Cambridge University Press.

[12] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, 1975.

On Reducing Clause Database in Glucose

Chu-Min LI[1,2], Fan Xiao[1] and Ruchu XU[1]

[1] Huazhong University of Science and Technology, China
[2] MIS, Universit de Picardie Jules Verne, France

Abstract

Modern CDCL SAT solvers generally save the variable value when backtracking. We present a measure called *nbSAT* based on the saved assignment to predict the usefulness of a learnt clause when reducing clause database in Glucose 3.0. Roughly speaking, the nbSAT value of a learnt clause is equal to the number of literals satisfied by the current partial assignment plus the number of other literals that would be satisfied by the saved assignment. The nbSAT measure is similar to a previous measure called *psm* which is not implemented in Glucose 3.0. We study the nbSAT measure by empirically showing that it may be more accurate than the LBD measure originally used in Glucose. Based on this study, we implement an improvement in Glucose 3.0 to remove half of learnt clauses with large nbSAT values instead of half of clauses with large LBD values. This improvement, together with a resolution method to keep the learnt clauses or resolvents produced using a learnt clause that subsume an original clause, makes Glucose 3.0 more effective for the application and hard combinatorial instances from the SAT 2014 competition.

1 Introduction

In propositional logic, a variable x_i may take values 0 (for false) or 1 (for true). A literal l_i is a variable x_i or its negation $\bar{x}_i$. A clause is a disjunction of literals, and a CNF formula ϕ is a conjunction of clauses. The size of a clause is the number of its literals. An assignment of truth values to the propositional variables satisfies a literal x_i if x_i takes the value 1 and satisfies a literal $\bar{x}_i$ if x_i takes the value 0, satisfies a clause if it satisfies at least one literal of the clause, and satisfies a CNF formula if it satisfies all the clauses of the formula. An empty clause represents a conflict, because it contains no literals and cannot be satisfied. A unit clause contains only one literal that should be satisfied by assigning the appropriate truth value to the variable. An assignment for a CNF formula ϕ is complete if all the variables occurring in ϕ have been assigned; otherwise, it is partial. The SAT problem for a CNF formula ϕ is the problem of finding an assignment of values to propositional variables that satisfies all clauses of ϕ.

Thanks to the progress made in developing CDCL (Conflicting-Driven Clause Learning) SAT solvers, reducing combinatorial problems to SAT becomes a powerful solving strategy. Roughly speaking, in order to solve a SAT problem, a CDCL solver repeatedly makes and propagates a decision, i.e. pick a variable using a heuristic, assign it a truth value and propagate all unit clauses implied by the decision until an empty clause is produced. Then the empty clause is analyzed and a new clause is learnt and added into the clause database. The learnt clause allows the solver to avoid the same conflict in the future and to determine the decision on which the solver should backtrack [5] [8]. If all variables are assigned a truth value without producing any empty clause, the problem is satisfiable and the complete assignment is output as a solution. Otherwise, the solver should learn an empty clause to prove the unsatisfiability of the problem.

So, the learnt clauses are essential for the performance of a CDCL SAT solver. However, the solver is slowed down when there is a large number of learnt clauses, because the solver has to check too many clauses to find unit clauses in this case during the propagation of a

decision. Moreover, these learnt clauses can also overflow the memory. In order to remedy these drawbacks, a common practice when designing a CDCL solver is to measure the quality of each learnt clause using a heuristic and to periodically remove half of learnt clauses judged less useful in the future.

In this paper, we study the policies to reduce the learnt clause database in the well-known CDCL solver Glucose [3], and present a measure called nbSAT to predict the usefulness of a learnt clause. The nbSAT measure is similar to a previous measure called *psm* proposed in [2] which is not implemented in Glucose. We study the nbSAT measure by empirically showing that it may be more accurate than the LBD (Literals Blocks Distance) measure originally used in Glucose. Then we combine nbSAT with LBD in Glucose to reduce the learnt clause database. Furthermore, we implement a resolution method that detects if a learnt clause or a resolvent produced using a learnt clause subsumes an original clause, and in this case, the subsuming learnt clause or resolvent is kept and never removed.

This paper is organized as follows. Section 2 recalls the main features of Glucose 3.0 related to our work. Section 3 presents the nbSAT measure and studies its properties. Section 4 and Section 5 present different uses of the nbSAT measure in reducing clause database in Glucose 3.0. Section 6 presents the resolution method to keep the subsuming learnt clauses or resolvents. Section 7 presents experimental results. Section 8 concludes.

2 Main features of Glucose

Glucose is a very efficient CDCL-based complete SAT solver developed from Minisat [6]. It is always one of the awarded SAT solvers in SAT competitions since 2009. Our work is based on the following features of Glucose 3.0, the last sequential release of Glucose[1].

2.1 LBD: a measure of the usefulness of a learnt clause

In a CDCL SAT solver, each decision is followed by a unit propagation. The decision and all literals fixed (i.e. satisfied or falsified) during the unit propagation belong to the same level. In Glucose, these literals, as well as the decision, are said to form a "Block". When a clause C is learnt, all literals in C are falsified under the current partial assignment, and the number of different blocks in C is called Literals Blocks Distance (LBD) of C. This measure is static in most cases, i.e., it is recomputed and updated only in special cases. The learnt clauses with LBD=2 are called "Glue Clauses" and are attached a particular importance, because they only contain one variable in the last decision level and all other variables belong to another block. It is expected that these glue clauses are frequently involved in future unit propagations and conflicts, so the glue clauses are never removed in Glucose.

It is well-known that industrial SAT instances usually exhibit a clear community structure, i.e., variables in an industrial instance form communities. The variables inside a community have strong relations, but the relations among the variables in different communities are much weaker [1]. In [9], it is shown that LBD can be strongly related to the community structure of the initial formula, which explains why clauses with smaller LBD are more probably used in unit propagations.

[1] available at http://www.labri.fr/perso/lsimon/glucose/

2.2 Aggressive learnt clause database reduction

Since Glucose, aggressive clause database reduction policies are essential ingredients of CDCL solvers. Once the number of clauses learnt since the last database reduction reaches 2000 + 300*n, where n is the number of database reductions performed so far, the reducing process is fired: the learnt clauses are sorted in the decreasing order of their LBD, and the first half of learnt clauses are removed except binary clauses, glue clauses, and the clauses that are reasons of the current partial assignment. One consequence of this aggressive clause database reduction policy is that more than 93% of learnt clauses can be removed (see the Glucose webpage). The new measure of learnt clause usefulness we present in this paper is strongly related to the aggressive reduction policy.

2.3 Fast restart and phase saving

A CDCL based SAT solver usually uses the restart mechanism [7] to prevent heavy-tailed phenomena, every restart constructing a search tree from scratch. In Glucose 3.0, the restart policy is very aggressive. It does not depends on the learnt clause database reduction, nor on the number of conflicts reached since the last restart. It depends on the average LBD of the learnt clauses [4]. The consequence is that Glucose 3.0 is restarted only after few hundreds of conflicts in the average.

Together with the aggressive restart policy, Glucose implements phase saving [10] policy: when backtracking, Glucose saves the value of each fixed variable, and when a variable is picked up to make a decision, it is assigned the save value. In this way, Glucose stays in the same search space and benefits from the results of early restarts. The new measure of learnt clause usefulness we present in this paper is strongly related to the aggressive restart policy and the phase saving policy.

3 A Measure of Learnt Clause Usefulness: nbSAT

From a theoretical point of view, all learnt clauses are logical consequences of original clauses of a CNF formula and thus are redundant. From a practical point of view, a learnt clause is useful only if it is involved in at least one unit propagation and helps to reach a conflict. The intuition of the LBD measure in Glucose is based on the tight links between the variables in a block: once a variable is fixed, the other variables in the same block can probably be also fixed in a unit propagation because of the tight links among them, so that a clause with smaller LBD has more chance to become unit or empty. Nevertheless, when the links among the variables in a block are not so tight, other measures of the learnt clause usefulness might be more relevant.

Observe that in a CDCL solver with phase saving, a learnt clause has more chance to become unit, if all its literals would be falsified by the saved phases. For example, if the save phase of the three variables x_1, x_2 and x_3 is true, the clause $\neg x_1 \vee \neg x_2 \vee \neg x_3$ becomes unit after two of the three variables are picked as decision variables. More generally, in the search space characterized by the saved phases, a learnt clause has more chance to become unit during search if it is satisfied by fewer variables with the saved value.

Definition 1. *In a CDCL solver with phase saving, the nbSAT (short for number of satisfied literals) of a learnt clause C is the number of literals of C satisfied by the current partial assignment plus the number of literals not fixed but would be satisfied by the saved assignment.*

Proposition 1. *If a CDCL solver with phase saving makes every decision according to the saved assignment, at least one learnt clause with nbSAT=0 will become unit.*

The nbSAT measure is clearly dynamic, contrary to the LBD measure, because the saved assignment is changing upon backtracking. We have to frequently update the nbSAT value for each learnt clause. The definition of nbSAT is similar to the *psm* measure proposed in [2], except that the psm value of a learnt clause, as is described in [2], does not take the current partial assignment into account, and is the number of literals that would be satisfied by the saved assignment, no matter if these literals are actually satisfied or not by the current partial assignment. Note that the value of the variables in the current partial assignment may be different from the saved assignment if these variables are fixed by unit propagation instead of decisions in the current partial assignment. More importantly, the nbSAT measure will be exploited differently in this paper, as will be presented in Section 4.

We now empirically compare nbSAT and LBD in Glucose 3.0 (the psm measure is not compared because it is not implemented in Glucose 3.0). Following [3], we run Glucose 3.0 on the set of hard combinatorial and industrial benchmarks of the SAT competition 2014 on a computer with Intel Westmere Xeon E7-8837 of 2.66GHz and 10GB of memory under Linux with a cutoff time of 5000 seconds as in the competition. Each benchmark contains 300 instances. Before each learnt clause database reduction, the nbSAT value of each learnt clause is computed. For each nbSAT value k in {0, 1, 2, ..., 9, 10, 11+}, where 11+ represents all values equal to or bigger than 11, we measure the total number of times in which all learnt clauses with this nbSAT value are useful in unit propagation #up(k) and in conflict analysis #analyze(k), respectively. Note that the clauses learnt between the m^{th} and the $(m+1)^{th}$ clause database reductions are not counted in #up(k) and #analyze(k) before the $(m+1)^{th}$ reduction. Also note that the real nbSAT value of a learnt clause can be changed during search. However, we use its nbSAT value computed at the m^{th} clause database reduction to compute #up(k) and #analyze(k) between the m^{th} and the $(m+1)^{th}$ clause database reductions. In fact, we want to use this nbSAT value to predict its usefulness between the m^{th} and the $(m+1)^{th}$ clause database reductions.

We compute the cumulative distribution functions for the nbSAT values:

$$f_{nbSATup}(k) = \frac{\sum_{i=0}^{k} \#up(i)}{\sum_{i=0}^{11+} \#up(i)}$$

and

$$f_{nbSATanalyze}(k) = \frac{\sum_{i=0}^{k} \#analyze(i)}{\sum_{i=0}^{11+} \#analyze(i)}$$

Similarly, we compute the cumulative distribution functions for LBD values and clause sizes in {2, 3, ..., 9, 10, 11+}. Again note that clauses learnt between the m^{th} and the $(m+1)^{th}$ clause database reductions are not counted in the functions before the $(m+1)^{th}$ reduction. Generally the LBD value of a clause is not changed. It is changed only when the new LBD value is smaller. In this case, the new LBD value is taken into account when computing the cumulative distribution functions, favoring the set of learnt clauses with small LBD values. Observe that the set of learnt clauses with small LBD values is further favored in the cumulative distribution functions in Glucose 3.0, because the clauses with large LBD values tend to be removed, which is not the case for the nbSAT measure.

Figure 1 and Figure 2 compare the cumulative distribution functions of the nbSAT and LBD values and the clause sizes for the industrial benchmark and hard combinatorial benchmark, respectively. The two figures clearly show that clauses with small nbSAT values are significantly more frequently used than the clauses with small LBD values in unit propagation and conflict

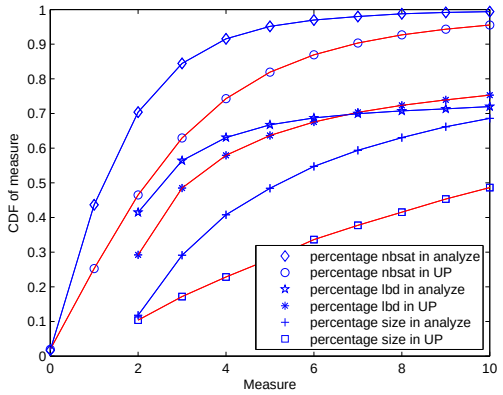

Figure 1. Cumulative distribution functions for nbSAT, LBD and clause size in glucose-3.0 on industrial benchmark of the SAT competition 2014. Names of curves are in the same ordering as the curves. Each point (x, y) represents the percentage y of learnt clauses used in unit propagation or conflict analyses with nbSAT, LBD or clause size$\leq x$.

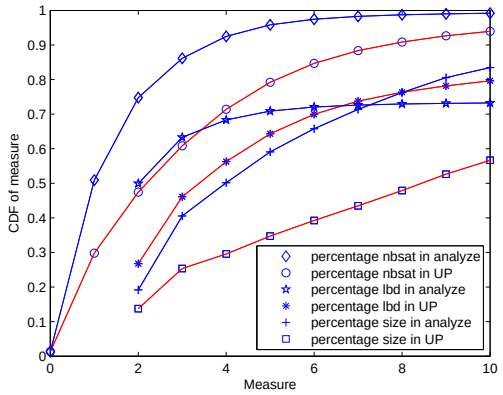

Figure 2. Cumulative distribution functions for nbSAT, LBD and clause size in glucose-3.0 on hard combinatorial benchmark of the SAT competition 2014. Names of curves are in the same ordering as the curves. Each point (x, y) represents the percentage y of learnt clauses used in unit propagation or conflict analyses with nbSAT, LBD or clause size$\leq x$.

analyses. In particular, more than 95% of learnt clauses used in unit propagation and conflict analyses have nbSAT $\leq$10, which is especially true for the learnt clauses used in conflict analyses.

Figure 3 and Figure 4 show the cumulative distribution functions of the number of learnt clauses with each nbSAT, LBD and clause size for the industrial benchmark and hard combinatorial benchmark, respectively (at each database reduction, after removing the half of learnt clauses with bigger LBD, the number of remaining learnt clauses with each nbSAT (LBD, clause size) value is counted. The total number of learnt clauses with each nbSAT (LBD, clause size) value is obtained by summing up the numbers for all database reductions). On the industrial benchmark, about 46% of learnt clauses are of nbSAT $\leq$2, and these clauses contribute about 71% in conflict analyses (i.e. 71% of the learnt clauses used in conflict analyses are of nbSAT$\leq$2)

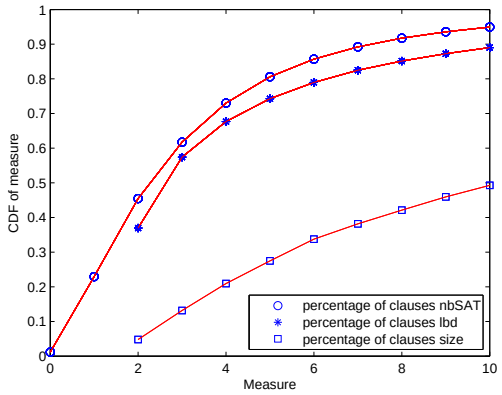

Figure 3. Cumulative distribution functions for number of learnt clauses with different nbSat, LBD and clause size in glucose-3.0 on industrial benchmark of the SAT competition 2014. Names of curves are in the same ordering as the curves. Each point (x, y) represents the percentage y of learnt clauses with nbSAT, LBD or clause size$\leq x$.

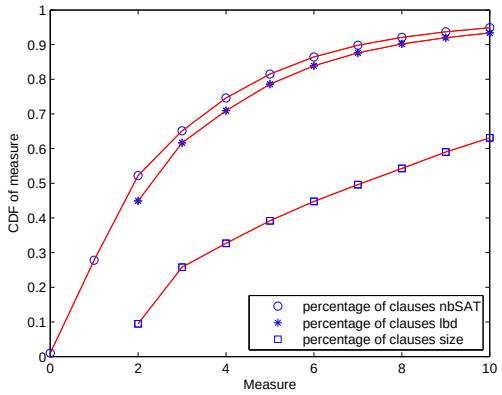

Figure 4. Cumulative distribution functions for number of learnt clauses with different nbSAT, LBD and clause size in glucose-3.0 on hard combinatorial benchmark of the SAT competition 2014. Names of curves are in the same ordering as the curves. Each point (x, y) represents the percentage y of learnt clauses with nbSAT, LBD or clause size$\leq x$.

and 47% in unit propagation; while about 58% of learnt clauses are of LBD$\leq$3, and these clauses contribute about 62% in conflict analyses and about 46% in unit propagation. These data mean that the clauses with small nbSAT are probably more useful for deriving conflicts and for unit propagation during search than the clauses with small LBD. Considering that Glucose 3.0 can remove some clauses with nbSAT$\leq$2 that could otherwise contribute in unit propagation and in conflict analyses, but always keeps the half of learnt clauses with smaller LBD in each database reduction, the effectiveness of the clauses with small nbSAT for deriving conflicts and for unit propagation might be still more important than as suggested in Figure 1. Similar observation can be made by analyzing Figure 2 and Figure 4 on hard combinatorial benchmark.

4 Using nbSAT in Clause Database Reduction

We implement two derived versions of Glucose 3.0 to use the nbSAT measure in the clause database reduction: Glucose_nbSAT+LBD and Glucose_LBD+nbSAT, in which everything is identical to Glucose 3.0, except that nbSAT is used to predict the usefulness of a learnt clause. Concretely, if the number of learnt clauses becomes bigger or equal to 2000+300*n since the last database reduction, we will remove half of the learnt clauses in Glucose_nbSAT+LBD as follows.

1. compute the $nbSAT$ value for each learnt clause;
2. Sort all learnt clauses in the decreasing order of their nbSAT value, breaking ties using the decreasing order of their LBD value. The remaining ties are broken using the clause activity value as in Glucose 3.0.
3. Remove the first half of learnt clauses (i.e. those with bigger nbSAT values), by keeping binary clauses, clauses whose LBD is 2, and the locked clauses (i.e. clauses that are reasons of the current partial assignment).

Glucose_LBD+nbSAT is identical to Glucose_nbSAT+LBD except that all learnt clauses are sorted in the decreasing order of their LBD value, breaking ties using the decreasing order of their nbSAT value.

Note that the saved assignment changes frequently during search. The measurement nbSAT works well only when the learnt clause database reduction is fired frequently, because otherwise, it does not reflect the current search state after many conflicts. This is not a problem with Glucose_nbSAT+LBD and Glucose_LBD+nbSAT, because the two solvers reduces the database frequently as Glucose 3.0, making it relevant to use the nbSAT measure in the database reduction.

Recall that the definition of nbSAT is similar to the psm measure proposed in [2]. Nonetheless, the psm measure is used in a different way to manage learnt clause database. At each database reduction, if the psm value of a learnt clause is bigger than a dynamically determined threshold, the clause is frozen (i.e., it is no longer used in unit propagation) but not definitely removed, otherwise it is activated (i.e., it can be used in unit propagation). A clause is definitely removed if it is not activated after k(=7) times, or if it is not involved in search for more than k times. The motivation of the approach can be stated as follows. The psm value of a learnt clause can change quickly. Freezing learnt clauses with big psm values instead of definitely removing them allows to keep those learnt clauses of which the psm values are big at a database reduction but will become small (thus useful for unit propagation and conflict analyses) in the near future. See [2] for more details. The utilization of the nbSAT measure appears simpler in our case: at every database reduction, the half of learnt clauses with bigger nbSAT values is simply and definitely removed without using any threshold.

5 Reducing Learnt Clause Database in the Root of the Search Tree

In Glucose, as well as in most CDCL-based solvers, a clause database reducing process can be fired inside a search tree. Two observations can be made about this strategy: (1) there are locked clauses, i.e. clauses that are reasons of the current partial assignment, that cannot be

removed, (2) the part of the tree before the reducing and the part of the tree after the reducing are constructed with very different clause database.

We can reduce the learnt clause database always at the beginning of each restart, i.e., at the root of the search tree that is going to be constructed, every time the number of learnt clauses is bigger or equal to 2000 + 300*n since the last database reducing. In this way, clauses satisfied by variables fixed at the root are simply removed, as well as the literals falsified in the remaining clauses. Note that no clause is locked at the root of a search tree. Moreover, since the reduction is not done inside the search tree, the search tree is constructed with the same incremental clause database.

This reduction policy is implemented in Glucose_nbSAT+LBD and Glucose_LBD+nbSAT, giving Glucose_nbSAT+LBD_root and Glucose_LBD+nbSAT_root, respectively. Compared with Glucose 3.0, the database reduction is delayed in Glucose_nbSAT+LBD_root and Glucose_LBD+nbSAT_root, because it is not fired as soon as the number of the newly learnt clauses reaches a limit, but should wait for the next restart. However the delay is not important, because the two solvers perform fast restart as Glucose 3.0. This reducing policy was also used in the parallel version of Glucose. Its effectiveness will be empirically studied in Section 7.

6 Keeping Subsuming Learnt Clauses

When a learnt clause is in the first half after all learnt clauses are sorted in the decreasing order of their nbSAT value, i.e., when it is going to be removed by the database reduction process, we check if it subsumes an original clause or if it can be resolved with an original clause to produce a resolvent that subsumes the original clause.

In the first case, the learnt clause replaces the original clause and will never be removed.

Example. Let $x_1 \vee x_2 \vee x_3 \vee x_4$ be an original clause, and $x_2 \vee x_3 \vee x_4$ be a learnt clause, then the shorter learnt clause is added as an original clause that is never removed, and the original clause $x_1 \vee x_2 \vee x_3 \vee x_4$ is removed.

In the second case, the produced resolvent replaces the original clause and will never be removed.

Example. Let $x_1 \vee x_2 \vee x_3 \vee x_4$ be an original clause, and $\bar{x}_2 \vee x_3 \vee x_4$ be a learnt clause, then the resolvent $x_1 \vee x_3 \vee x_4$ is added as an original clause that is never removed, and $x_1 \vee x_2 \vee x_3 \vee x_4$ is removed.

Glucose_nbSAT+LBD_root_rsltn and Glucose_LBD+nbSAT_root_rsltn are respectively Glucose_nbSAT+LBD_root and Glucose_LBD+nbSAT_root that keep the subsuming learnt clauses. This policy is similar to subsumption elimination (SE) and self-subsuming resolution (SSR) used in preprocessing in some SAT solvers. The difference is that in Glucose_nbSAT+LBD_root_rsltn and Glucose_LBD+nbSAT_root_rsltn, the policy is not limited in the preprocessing, but is dynamically applied in every clause database reduction, which requires a highly efficient implementation, because it is applied for an exponential number of times.

7 Experimental Results

We run experiments on the industrial benchmark and the hard combinatorial (crafted) benchmark of the SAT 2014 competition to compare the following solvers:

`Glucose_nbSAT+LBD_root_rsltn:` It is identical to Glucose 3.0, except that it sorts the learnt clauses in the decreasing order of their nbSAT value, breaking ties using the LBD values, when reducing clause database (See Section 4), that clause database reduction is always performed at the root (see Section 5), and that resolution is used to keep subsuming learnt clauses (see Section 6)

`Glucose_LBD+nbSAT_root_rsltn:` It is identical to Glucose_nbSAT+LBD_root_rsltn, except that the it sorts the learnt clauses in the decreasing order of their LBD value, breaking ties using the nbSAT values, when reducing clause database (See Section 4)

`Glucose_nbSAT+LBD_rsltn:` It is identical to Glucose_nbSAT+LBD_root_rsltn, except that the clause database reduction is fired as in Glucose 3.0, i.e., it is not fired in the root of a search tree, but fired as soon as the number of learnt clauses reaches a limit as in Glucose 3.0 (see Section 5)

`Glucose_LBD_root_rsltn:` It is identical to Glucose_LBD+nbSAT_root_rsltn, except that nbSAT is not used to break ties when sorting learnt clauses

`Glucose 3.0:` It is available at www.labri.fr/perso/lsimon/glucose/

All solvers are run on a computer with Intel Westmere Xeon E7-8837 of 2.66GHz and 10GB of memory under Linux. The cutoff time is 5000 seconds as in the competition. Since there can be other users on the machine, each solver is run three times for each instance and the best result is taken into account to minimize the possible perturbation. Table 1 compares the results of each solver.

solvers	Application #solved(sat+unsat)(time)	Hard combinatorial #solved(sat+unsat)(time)
Glucose_nbSAT+LBD_root_rsltn	210(103+107)(995.02s)	174(79+95)(1081.07s)
Glucose_LBD+nbSAT_root_rsltn	207(100+107)(960.02s)	166(78+88)(928.08s)
Glucose_nbSAT+LBD_rsltn	209(97+112)(1002.67s)	176(83+93)(1056.70s)
Glucose_LBD_root_rsltn	201(95+106)(949.60s)	171(79+92)(1064.69s)
Glucose 3.0	206(99+107)(962.35s)	164(77+87)(1087.17s)

Table 1. Number of instances solved within 5000 seconds of each solver, as well as the average runtime to solve an instance, in the set of 300 industrial or hard combinatorial instances of the 2014 SAT competition

The first observation that can be made from Table 1 is that the resolution to keep subsuming learnt clauses in Glucose_LBD_root_rsltn is not very effective for the industrial instances, because Glucose_LBD_root_rsltn solves 5 instances fewer than Glucose 3.0, but it is effective for the crafted instances, because Glucose_LBD_root_rsltn solves 7 instances more than Glucose 3.0. The explanation of this phenomenon is that the resolution is very costly for the industrial instances that contain a large number of clauses, even with a highly efficient implementation. However, The subsuming learnt clauses kept thanks to the resolution becomes effective for both industrial and hard combinatorial instances when the nbSAT measure is used to predict

the usefulness of a learnt clause. In fact, the three solvers with nbSAT and the resolution to keep subsuming learnt clauses solve more industrial and hard combinatorial instances, especially when nbSAT is used as the main measure as in Glucose_nbSAT+LBD_root_rsltn and Glucose_nbSAT+LBD_rsltn.

The second observation made from Table 1 is that it does not matter much to reduce clause database at the root of a search tree or inside the search tree, since Glucose_nbSAT+LBD_root_rsltn reduces the clause database only at the root of a search tree, and solves roughly the same number of instances in both industrial and hard combinatorial categories as Glucose_nbSAT+LBD_rsltn does.

The most important observation made from Table 1 is that the nbSAT measure appears to be more accurate than the LBD measure in predicting the usefulness of a learnt clause, especially for the hard combinatorial instances. In fact, Glucose_nbSAT+LBD_root_rsltn solves 10 hard combinatorial instances more than Glucose 3.0, and Glucose_nbSAT+LBD_rsltn solves 12 hard combinatorial instances more than Glucose 3.0. One possible explanation is that industrial instances exhibit a community structure that can be captured by LBD, while the community structure is not so clear in hard combinatorial instances. Therefore, the impact of nbSAT is more important for the hard combinatorial instances.

The solver Glucose_nbSAT+LBD_root_rsltn participated in the SAT Race'2015 under the name Glucose_nbSatRsltn and solves 4 instances more than Glucose in the set of 300 application instances (see http://baldur.iti.kit.edu/sat-race-2015/index.php?cat=results).

8 Conclusion

We have presented a measure called *nbSAT* to predict the usefulness of a learnt clause when reducing learnt clause database in Glucose 3.0. The nbSAT measure is similar to a previous measure called *psm*, but is exploited in a different way. It is closely associated with the aggressive clause database reduction and fast restart policies in Glucose 3.0. Experimental results on instances from the SAT 2014 competition suggest that the learnt clauses with small nbSAT values are more useful than the learnt clauses with small LBD values. We then implemented an improvement in Glucose 3.0 to remove half of learnt clauses with large nbSAT values, instead of half of clauses with large LBD clauses, when periodically reducing the learnt clause database. In addition, we implemented a resolution method in Glucose 3.0 to keep the learnt clauses or resolvents produced using a learnt clause that subsume an original clause of the instance to solve. The experimental results on benchmarks from the SAT 2014 competition show that Glucose 3.0 with these two improvements solve significantly more application and crafted instances than Glucose 3.0, no matter if the clause database reductions are fired at the root of a search tree or not.

Since the nbSAT value of a learnt clause can be changed frequently during search, one way to make the nbSAT measure more effective might be to use a yet more aggressive database reduction policy in Glucose 3.0, so that the nbSAT value can be more frequently re-computed to reflect its actual usefulness. We will investigate this research line in the future.

Acknowledgements We thank the anonymous reviewers for insightful comments. This work is supported by National Natural Science Foundation of China (NSFC, Grant No.61370184, Grant No.61070235 and Grant No.61272014) and by the MeCS platform of university of Picardie Jules Verne.

References

[1] Carlos Ansótegui, Jesús Giráldez-Cru, and Jordi Levy. The community structure of sat formulas. In *Theory and Applications of Satisfiability Testing–SAT'12*, pages 410–423. Springer, 2012.

[2] Gilles Audemard, Jean-Marie Lagniez, Bertrand Mazure, and Lakhdar Sais. On freezeing and reactivating learnt clauses. In *14th International Conference on Theory and Applications of Satisfiability Testing–SAT'11*, pages 188–200, 2011.

[3] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern sat solvers. *International Joint Conference on Artificial Intelligence Proceedings-IJCAI'09*, 38(4):399–404, 2009.

[4] Gilles Audemard and Laurent Simon. Refining restarts strategies for sat and unsat formulae. *in Proceedings of the 22nd International Conference on Principles and Practice of Constraint Programming–CP'12*, 2012.

[5] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. Conflict-driven clause learning sat solvers. *Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications*, pages 131–153, 2009.

[6] Niklas Eén and Niklas Sörensson. An extensible sat-solver. pages 502–518, 2004.

[7] Carla P. Gomes, Bart Selman, and Kautz Henry. Boosting combinatorial search through randomization. *in Proc. AAAI-98*, Madison, WI, July 1998.

[8] Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535. ACM, 2001.

[9] Zack Newsham, Vijay Ganesh, Sebastian Fischmeister, Gilles Audemard, and Laurent Simon. Impact of community structure on sat solver performance. In *Theory and Applications of Satisfiability Testing–SAT'14*, pages 252–268. Springer, 2014.

[10] Knot Pipatsrisawat and Adnan Darwiche. A lightweight component caching scheme for satisfiability solvers. In *Theory and Applications of Satisfiability Testing–SAT'07*, pages 294–299. Springer, 2007.

The Thousands of Models for Theorem Provers (TMTP) Model Library - First Steps

Geoff Sutcliffe[1] and Stephan Schulz[2]

[1] University of Miami, USA
[2] DHBW Stuttgart, Germany

Abstract

The TPTP World is a well established infrastructure that supports research, development, and deployment of Automated Theorem Proving (ATP) systems for classical logics. The TPTP world includes the TPTP problem library, the TSTP solution library, standards for writing ATP problems and reporting ATP solutions, and it provides tools and services for processing ATP problems and solutions. This work describes a new component of the TPTP world - the Thousands of Models for Theorem Provers (TMTP) Model Library. This is a library of models for axiomatizations built from axiom sets in the TPTP problem library, supported by functions for efficiently interpreting ground terms and closed formulae wrt interpretations, and tools for examining and processing interpretations. The TMTP supports the development of semantically guided theorem proving ATP systems, provide examples for developers of model finding ATP systems, and provides insights into the semantics of axiomatizations.

1 Introduction

Automated Theorem Proving (ATP) is concerned with the development of automatic techniques and computer programs for checking whether a conjecture is a theorem of some axioms, and for checking the consistency of a set of formulae. The TPTP World [21] is a well established infrastructure that supports research, development, and deployment of ATP systems for classical logics. The TPTP world includes the TPTP problem library [20], the TSTP solution library [21], standards for writing ATP problems and reporting ATP solutions [24, 19], tools and services for processing ATP problems and solutions [21], and it supports the CADE ATP System Competition (CASC) [22]. The TPTP world infrastructure has been deployed in a range of applications, in both academia and industry. The web page `http://www.tptp.org` provides access to all components.

The Thousands of Problems for Theorem Provers (TPTP) problem library is the original core component of the TPTP world, and is commonly referred to as "the TPTP". The TPTP problem library supplies the ATP community with a comprehensive library of the test problems that are available today, in order to provide an overview and a simple, unambiguous reference mechanism, to support the testing and evaluation of ATP systems, and to help ensure that performance results accurately reflect capabilities of the ATP systems being considered. The Thousands of Solutions from Theorem Provers (TSTP) solution library is the "flip side" of the TPTP – a corpus of ATP systems' solutions to TPTP problems. A major use of the TSTP is for ATP system developers to examine solutions to problems, and thus understand how they can be solved, leading to improvements to their own systems. The TPTP language is one of the keys to the success of the TPTP world. The language is used for writing both TPTP problems and TSTP solutions, which enables convenient communication between different systems and researchers. In conjunction with the TPTP language, the TPTP world uses the SZS[1] ontologies

[1] SZS is an acronym from the initials of the original authors' family names [25].

to record what is known or has been established about a TPTP problem or solution, and to describe sets of formulae. The TPTP world includes tools, programming libraries, and online services that are used to support the application and deployment of ATP systems. One of the most used services is SystemOnTPTP [17], which is an online service that allows an ATP problem or solution to be easily and quickly submitted in various ways to a range of ATP systems and tools. An important tool is GDV [18], which verifies TPTP format derivations. A very useful tool for human users is IDV [26], which provides an interactive interface for viewing TPTP format derivations.

This work describes a new component of the TPTP world - the Thousands of Models for Theorem Provers (TMTP) Model Library.[2] This is a library of models for axiomatizations built from axiom sets in the TPTP. The library is supported by functions for efficiently interpreting ground terms and closed formulae wrt interpretations, and tools for examining and processing interpretations. The components parallel those already in the TPTP world for ATP problems and solutions. Details of the TMTP's components are provided in this paper, but in summary ... The TMTP model library is similar to the TPTP problem library and TSTP solution library. The TPTP language is used for writing the models (they are one type of solution), and the SZS ontology is used to describe the various kinds of interpretations. The SystemOnTMTP service allows an interpretation to be submitted to various tools for evaluating formulae wrt the interpretation, and for examining and processing the interpretation. The IMV tool provides an interactive interface for viewing interpretations, and the GMV tool verifies that an interpretation is a model for a given set of formulae. The web page `http://www.tptp.org/TMTP` provides access to the TMTP.

The TMTP provides support for the development and execution of semantically guided ATP systems, in the style of SLM [6], SGLD [16], and SCOTT [15], which use one or more preselected interpretations to guide their search. (This is in contrast to ATP systems that use models that are computed or updated during their execution, e.g., iProver [10], Satallax [5], and CVC4 [3].) An implementation of semantic resolution [14] is planned. It is noteworthy that the dates of the publications cited here are rather old, and it is the first author's opinion that the potential for semantically guided ATP has not been fully exploited. This viewpoint was also expressed in [7]. The TMTP provides support for semantically guided techniques for axiom selection in large theories, in the style of SRASS [23]. The TMTP provides a basis for empirical research into the semantics of axiomatizations, hopefully leading to insights that benefit ATP research and development in general. Finally, while the TMTP does provide examples of solutions to satisfiable and countersatisfiable ATP problems[3], it is not intended to be a comprehensive repository of justifications for ATP systems' claims of satisfiability or countersatisfiability – that is the purpose of the TSTP solution library, which might provide justifications other than models, e.g., a claim of countersatisfiability can be justified by a proof showing that a conjecture is a countertheorem of the axioms.

The remainder of this paper is organized as follows: This section ends with definitions for the terminology used in this paper. Section 2 explains how the models in the TMTP are collected, and describes formats for writing various kinds of interpretations using the TPTP language. Section 3 describes some tools that have been developed for examining and processing interpretations. Section 4 concludes, and outlines plans for further development of the TMTP.

[2]Not all of these components have been completely implemented at the time of writing.

[3]The SZS ontology supplies the definitions of status values such as "theorem", "countertheorem", "satisfiable", "countersatisfiable".

Terminology: A (logical) *language* is defined in the usual way [2], with variables, functions, and predicates. An *interpretation* for a language $\mathcal{L}$ is a structure that has a domain $\mathbb{D}$, and can interpret all ground terms $\mathcal{T}$ in $\mathcal{L}$ as elements of $\mathbb{D}$, and all closed formula $\mathcal{F}$ in $\mathcal{L}$ as either $true$ or $false$. A *model* of a set $\mathcal{S}$ of closed formulae is an interpretation that interprets all the formulae in $\mathcal{S}$ as $true$. An interpretation is *complete* in the sense that it can interpret all ground terms and closed formulae in $\mathcal{L}$. A *partial interpretation* for $\mathcal{L}$ can interpret some (not necessarily all, but possibly all in which case it is complete) ground terms and closed formulae in $\mathcal{L}$. A *strictly partial interpretation* for $\mathcal{L}$ cannot interpret all ground terms and closed formulae in $\mathcal{L}$. A strictly partial interpretation can be complete for ground terms or closed formulae, but not for both. A strictly partial interpretation is *complete for a set* $\mathcal{S}$ of ground terms and closed formulae if it can interpret all the elements in the set. A *strictly partial model* of a set $\mathcal{S}$ of closed formulae is a strictly partial interpretation that is complete for $\mathcal{S}$, and interprets all the formulae in $\mathcal{S}$ as $true$.

2 Collecting Models

TMTP models (both complete and strictly partial) of a language $\mathcal{L}$ are expressed using the TPTP language, as a set of TPTP formulae. Four kinds of interpretations are currently defined for the TMTP: Herbrand interpretations, finite interpretations, integer interpretations, and real interpretations. These types have been categorized in the SZS dataform ontology, along with the notions of complete, partial, and strictly partial interpretations. The relevant section of the ontology is shown in Figure 1.[4] Examples of ontology values and their three-letter acronyms include "Interpretation (Int)" at the top of the hierarchy, "Herbrand Strictly partial Model (HSM)" in the right branch, and "Integer Partial Interpretation (IPI)" at bottom middle of the left branch. Full details of each possible value are given in Appendix A. The lines in the ontology can be followed up the hierarchy as "isa" links, e.g., an Integer Partial Interpretation (IPI) isa Domain Partial Interpretation (DPI) isa Partial Interpretation (PIn) isa Interpretation (Int). The classification of an interpretation into the ontology can be partially automated, e.g., if a finite domain is used then the interpretation is somewhere on the lefthand branch. More precise placement, e.g., specifying that the interpretation is complete, partial, or strictly partial, normally requires information from the ATP system that produced the interpretation. This information is important for some uses of interpretations, in particular for using deduction to interpret formulae wrt an interpretation (see Section 3.1).

2.1 Herbrand Interpretations

Herbrand interpretations are represented by sets of TPTP formulae that define the subset of the Herbrand base that is *true*, and a formula is evaluated wrt a Herbrand interpretation by trying to prove the formula is a theorem of the interpretation's formulae (see Section 3.1).

Figure 2 provides an example defining Herbrand model for `PUZ001-3`, produced by the iProver ATP system [10]. Examples of ATP systems that generate Herbrand models include iProver and Darwin [4].

Saturations are a special case. The existence of a saturation (with respect to a complete calculus) guarantees the existence of at least one Herbrand model. However, the model is not, in general, uniquely defined on the logical level alone. For superposition-based calculi,

[4] The figure conflates the analogous trees for Interpretations and Models, and similarly the analogous subtrees for the Partial and Strictly partial cases. It could be expanded into a taxonomic tree. Each path from the "Interpretation" root to a leaf of this figure is one branch of the taxonomic tree.

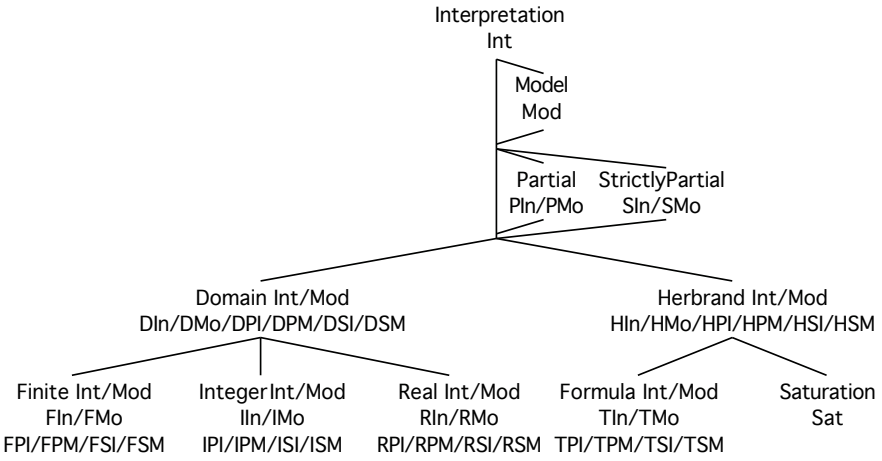

Figure 1: SZS Ontology for Interpretations

```
%------ Negative definition of lives
fof(lives_defn,axiom,(
    ! [X0] : ( ~ lives(X0) <=> $false ) )).

%------ Positive definition of killed
fof(killed_defn,axiom,(
    ! [X0,X1] :
      ( killed(X0,X1)
    <=> ( X0 = agatha & X1 = agatha ) ) )).

%------ Positive definition of richer
fof(richer_defn,axiom,(
    ! [X0,X1] :
      ( richer(X0,X1)
    <=> ( X0 = butler & X1 = agatha ) ) )).

%------ Negative definition of hates
fof(hates_defn,axiom,(
    ! [X0,X1] :
      ( ~ hates(X0,X1)
    <=> ( ( X0 = agatha & X1 = butler )
        | ( X0 = butler & X1 = butler )
        | X0 = charles
        | ( X1 = butler & X0 != butler ) ) ) )).
```

Figure 2: A Herbrand Model for PUZ001-3

the saturation defines a unique Herbrand model for a given literal selection strategy and term ordering. If the term ordering (and the induced ordering on literals and clauses) is not only terminating, but also has length-bounded descending chains (as in the case of the frequently used Knuth-Bendix ordering), Bachmair/Ganzinger style bottom-up model construction [1] can be used to interpret arbitrary ground atoms. However, the process is unlikely to be efficient in the general case, although it is plausible for, e.g., EPR. Using a saturation to interpret a formula by trying to prove the formula from the saturation (see Section 3.1) is possible, but cannot always succeed even in principle, since the model is not always uniquely defined. Moreover, it is necessary to use an ATP system that implements exactly the same ordering and redundancy criteria that were used to produce the saturation, i.e., in practice the ATP system that found the saturation in the first place.

Figure 3 provides an example saturation, for the TPTP problem `PUZ001-3`, produced by the E ATP system [13]. It is noteworthy that the saturation is expressed as a set of clauses in TPTP format, but it does not specify the ordering and redundancy criteria that were used. It will be necessary to extend the TPTP presentation to include this information. Examples of ATP systems that generate saturations include E, SPASS [27], and Vampire [11].

```
cnf(c_0_19,plain, ( hates(butler,butler) | killed(agatha,agatha) )).
cnf(c_0_20,plain, ( hates(butler,charles) | ~ killed(charles,agatha) )).
cnf(c_0_21,plain, ( hates(butler,X1) | richer(X1,agatha) | ~ lives(X1) )).
cnf(c_0_22,plain, ( hates(X1,X2) | ~ killed(X1,X2) )).
cnf(c_0_23,plain, ( ~ hates(butler,butler) | ~ hates(butler,charles) )).
cnf(c_0_24,plain, ( ~ hates(X1,agatha) | ~ hates(X1,butler) |
                    ~ hates(X1,charles) )).
cnf(c_0_25,plain, ( hates(butler,X1) | ~ hates(agatha,X1) )).
cnf(c_0_26,plain, ( ~ richer(X1,X2) | ~ killed(X1,X2) )).
cnf(c_0_27,plain, ( ~ hates(agatha,X1) | ~ hates(charles,X1) )).
cnf(c_0_28,plain, ( ~ hates(agatha,butler) )).
cnf(c_0_29,plain, ( ~ hates(charles,charles) )).
cnf(c_0_30,plain, ( hates(butler,agatha) )).
cnf(c_0_31,plain, ( hates(agatha,charles) )).
cnf(c_0_32,plain, ( hates(agatha,agatha) )).
cnf(c_0_33,plain, ( lives(charles) )).
cnf(c_0_34,plain, ( lives(butler) )).
cnf(c_0_35,plain, ( lives(agatha) )).
```

Figure 3: A Saturation for `PUZ001-3`

2.2 Finite Interpretations

Finite interpretations [24] are represented by FOF that specify the domain, define the interpretation of the functions, and define the interpretation of the predicates. The elements of the domain are specified in a formula of the form:

`fof(`*fi_name*`,fi_domain,`
`  ! [X] : ( X =` e_1 `| X =` e_2 `| ... | X =` e_n `) ).`

where the e_i are all "distinct object"s, or all distinct integers, or all distinct constants. The use of "distinct object"s or integers for a domain is preferred over constants, because they are predefined to be unequal. If the domain elements are constants then their inequality must be

explicitly stated in formulae of the form:

```
fof(e_i_not_e_j,fi_domain,
    e_i != e_j ).
```

The interpretation of functors is written in the form:

```
fof(fi_name,fi_functors,
    ( f(e_1,...,e_m) = e_r
    & f(e_1,...,e_p) = e_s
     ...  )  ).
```

specifying that, e.g., `f(`e_1`,...,`e_m`)` is interpreted as the domain element e_r. The interpretation of predicates is written in the form:

```
fof(fi_name,fi_predicates,
    ( p(e_1,...,e_m)
    & ~ p(e_1,...,e_p)
     ...  )  ).
```

specifying that, e.g., `p(`e_1`,...,`e_m`)` is interpreted as *true* and `p(`e_1`,...,`e_p`)` is interpreted as $false$. Equality is interpreted naturally by the domain, with the understanding that identical elements are equal. For the interpretation of functors and predicates, universal quantifications can be used if all domain elements can be used in an argument position, e.g.,

```
fof(fi_name,fi_functors,
    ( ! [X] : ( f(e_1,...,X) = e_r
    & f(e_1,...,e_p) = e_s
     ...  )  ).
```

specifies that the last argument position can be any domain element in the functions interpreted as e_r.

Figure 4 provides an example set of formulae that has a finite model, and Figure 5 provides an example finite model for the formulae, produced by the ATP system Paradox [8]. In the model the interpretation of the predicate `p` can be specified in two ways, one using a universal quantifier, and the other explicitly using the domain elements. The difference is relevant when interpreting formulae wrt an interpretation, as discussed in Section 3.1. Examples of ATP systems that generate finite models include Paradox, Mace4 [12] and DarwinFM [4].

```
%----About the constants
fof(a_not_b,   axiom, a != b ).

%----About the functions
fof(s_not_X,   axiom, ! [X] : s(X) != X ).
fof(f_b_a,     axiom, f(b) = a ).
fof(f_ss_X,    axiom, ! [X] : f(s(s(X))) = X ).

%----About the predicates
fof(p_a,       axiom, p(a) ).
```

Figure 4: Example Axiomatization that has a Finite Model

2.3 Integer and Real Interpretations

The domain of an integer interpretation is the integers, as defined by the `$int` type of the TPTP's TFF language. The interpretations are then represented by three types of TFF formu-

```
%----Model domain
fof(domain,fi_domain, ! [X] : ( X = "d1" | X = "d2" | X = "d3" ) ).

%----Constants
fof(a,   fi_functors, a = "d1" ).
fof(b,   fi_functors, b = "d2" ).

%----Total functions
fof(f,   fi_functors, f("d1") = "d3" & f("d2") = "d1" & f("d3") = "d2" ).
fof(s,   fi_functors, s("d1") = "d3" & s("d2") = "d1" & s("d3") = "d2" ).

%----Total predicates - Universal quantification
%---- fof(p, fi_predicates, ! [X1] : p(X1) <=> $true ).

%----Total predicates - Listed
fof(p, fi_predicates, p("d1") & p("d2") & p("d3") ).
```

Figure 5: A Finite Model for Figure 4

lae: type declarations, formulae to define the interpretation of the functions, and formulae to define the interpretation of the predicates. The type declarations declare all constants to be of type `$int`, all functions to be from tuples of `$int` to `$int`, and all predicates to be from tuples of `$int` to `$o`. Thus every constant and function is interpreted as an element of the integer domain, the formulae that define the interpretation of the functions map tuples of domain elements to a domain element, and the formulae that define the interpretation of the predicates map tuples of domain elements to *true* or $false$.

Figure 6 provides an example set of formulae (modified from those in Figure 4) that does not have a finite model, and Figure 7 provides an example integer model for the formulae. Examples of ATP systems that generate integer models include CVC4 [3] and Z3 [9].

```
%----About the constants
fof(a_not_b,   axiom, a != b ).

%----About the functions
fof(bigger_s,  axiom, ! [X] : bigger(s(X),X) ).
fof(bigger_t,  axiom, ! [X,Y] : ( bigger(X,Y) => bigger(s(X),Y) ) ).
fof(s_not_X,   axiom, ! [X,Y] : ( bigger(X,Y) => X != Y ) ).
fof(f_b_a,     axiom, f(b) = a ).
fof(f_ss_X,    axiom, ! [X] : f(s(s(X))) = X ).

%----About the predicates
fof(p_a,       axiom, p(a) ).
```

Figure 6: Example Axiomatization that does not have a Finite Model

The domain of a real interpretation is the reals, as defined by the `$real` type of the TPTP's TFF language. The interpretations are then represented by three types of TFF formulae, analogous to integer interpretations, but using the `$real` type. Figure 8 provides an example real model for the formulae of Figure 6. Example of ATP systems that generate real models

```
%----Model types
tff(a_type,    type,  a: $int ).
tff(b_type,    type,  b: $int ).
tff(s_type,    type,  s: $int > $int ).
tff(f_type,    type,  f: $int > $int ).
tff(b_type,    type,  bigger: ( $int * $int ) > $o ).
tff(p_type,    type,  p: $int > $o ).

%----Constants
tff(a_is_1,    axiom, a = 1 ).
tff(b_is_1,    axiom, b = 4 ).

%----Total functions
tff(s,         axiom, ! [X: $int] : s(X) = $product(X,2) ).
tff(f_s,       axiom, ! [X: $int] : f(X) = $quotient_t(X,4) ).

%----Total predicates
tff(bigger,    axiom, ! [X: $int,Y: $int] : ( bigger(X,Y) <=> $greater(X,Y) ) ).
tff(p_natural, axiom, ! [X: $int] : ( p(X) <= $greatereq(X,1) )).
tff(not_p_more,axiom, ! [X: $int] : ( ~ p(X) <= $less(X,1) )).
```

Figure 7: An Integer Model for Figure 6

are CVC4 and Z3.

```
%----Model types
tff(a_type,    type,  a: $real ).
tff(b_type,    type,  b: $real ).
tff(s_type,    type,  s: $real > $real ).
tff(f_type,    type,  f: $real > $real ).
tff(b_type,    type,  bigger: ( $real * $real ) > $o ).
tff(p_type,    type,  p: $real > $o ).

%----Constants
tff(a_is_1,    axiom, a = 1.0 ).
tff(b_is_1_4,  axiom, b = 0.25 ).

%----Total functions
tff(s,         axiom, ! [X: $real] : s(X) = $quotient(X,2) ).
tff(f_s,       axiom, ! [X: $real] : f(X) = $product(X,4) ).

%----Total predicates
tff(bigger,    axiom, ! [X: $real,Y: $real] : ( bigger(X,Y) <=> $greater(X,Y) ) ).
tff(p_natural, axiom, ! [X: $real] : ( p(X) <= $greatereq(X,1) )).
tff(not_p_more,axiom, ! [X: $real] : ( ~ p(X) <= $less(X,1) )).
```

Figure 8: A Real Model for Figure 6

2.4 Building the TMTP

The TPTP problem library includes satisfiable *axiomatization problems* that consist of only `include` directives for TPTP axiom files (no problem-specific formulae), so that the axioms constitute an axiomatization of some recognized theory. For example, `PHI001^1` consists of

```
%----Axioms for Quantified Modal Logic KB.
include('Axioms/LCL016^0.ax').
include('Axioms/LCL016^1.ax').
%----Axioms about God
include('Axioms/PHI001^0.ax').
```

The TMTP is built by collecting solutions to axiomatization problems, i.e., their models from the TSTP solution library. As new (versions of) ATP systems are added to the TPTP world, they are run over all the problems in the TPTP, and their solutions are added to the TSTP. In the case of a new version of an ATP system, the old version's solutions are replaced by the new version's solutions in the TSTP. The new systems' models for axiomatization problems are new candidates for addition to the TMTP. Each such new model is checked against existing models in the TMTP, and if it is not a syntactic variant of an existing model it is copied into the TMTP. In this way multiple models of the TPTP axiomatization are collected. At the time of writing, many more potential axiomatization problems have been identified for addition to the TPTP, and subsequently their models will be added to the TMTP.

It is important to ensure that non-trivial models are produced and included in the TMTP. For example, the axiomatization of the natural numbers $\{int(zero), \forall X(int(X) \Rightarrow int(succ(X)))\}$ has a trivial interpretation with a single domain element $d1$, with $zero$ and $succ(d1)$ both mapping to $d1$, and $int(d1)$ mapping to $true$. The intended integer interpretation that should (also) be in the TMTP would have the normal interpretation of $succ$ as the successor function.

The TMTP has a naming scheme similar to that used for problems in the TPTP. The model naming scheme is *DDDNNNFV*.*MMM*-*SZS*. *DDD* is the TPTP domain acronym (`ALG`, `PUZ`, `SET`, etc.), *NNN* is the abstract problem number, *F* is the logical form (`^` for THF, `=` for TFF with arithmetic, `_` for TFF without arithmetic, `+` for FOF, and `-` for CNF), and *V* is the problem version number, so that *DDDNNNFV* is the name of a TPTP axiomatization problem. *MMM* is the model number for that axiomatization, and *SZS* is the SZS dataform (`FMo`, `Sat`, etc.). Thus an example TMTP model name is `KRS176+1.005-FMo`. An extension of `.m` is added to create the model file name.

Each model file consists of a header containing information about the ATP system that built the model, the computing resources used to build the model, statistics about the model, and user comments. The formulae that define the model follow below the header. Figure 9 shows an example TMTP model file (with the bulk for the defining formulae omitted).

3 Examining and Processing the TMTP Models

3.1 Interpreting Closed Formulae wrt an Interpretation

A key capability required for using interpretations (and thus also the TMTP models) is efficiently interpreting closed formulae wrt the interpretations. Direct interpreting formulae according to the semantic rules of the logic often provides this, but the feasibility depends on the nature of the interpretation. For finite interpretations directly interpreting formulae is relatively easy, by instantiating quantified variables with domain elements, using the function

```
%------------------------------------------------------------------------------
% File       : Paradox---4.0
% Problem    : KRS176+1 : TPTP v6.2.0. Released v4.0.0.
% Transform  : none
% Format     : tptp:short
% Command    : paradox --no-progress --time %d --tstp --model %s

% Computer   : n189.star.cs.uiowa.edu
% Model      : x86_64 x86_64
% CPU        : Intel(R) Xeon(R) CPU E5-2609 0 2.40GHz
% Memory     : 32286.75MB
% OS         : Linux 2.6.32-573.1.1.el6.x86_64
% CPULimit   : 300s
% DateTime   : Wed Aug  5 05:31:13 EDT 2015

% Result     : Satisfiable 0.01s
% Output     : FiniteModel 0.01s
% Verified   :
% Statistics : Number of formulae       :   63 (  63 expanded)
%              Number of leaves         :   63 (  63 expanded)
%              Depth                    :    0
%              Number of atoms          :  149 ( 149 expanded)
%              Number of equality atoms :  125 ( 125 expanded)
%              Maximal formula depth    :    9 (   2 average)
%              Maximal term depth       :    2 (   1 average)

% Comments   :
%------------------------------------------------------------------------------
% domain size is 2
fof(domain,fi_domain,(
    ! [X] : ( X = "1" | X = "2" ) )).

fof(cax,fi_functors,(
    cax = "2" )).

    ...

fof(model,fi_predicates,
    ( ( model("1","1") <=> $false )
    & ( model("1","2") <=> $false )
    & ( model("2","1") <=> $false )
    & ( model("2","2") <=> $false ) )).

%------------------------------------------------------------------------------
```

Figure 9: A TMTP Model File

definitions to interpret functors applied to domain elements, and using the predicate definitions to interpret predicates applied to domain elements. If universal quantification is used in the predicate definitions, e.g., as in the commented out definition in Figure 5, this can be used to short-circuit the interpretation process. For example, the atom `p(a)` could be immediately interpreted as *true* without having to first interpret `a` as `"d1"`. For Herbrand, integer, and real interpretations, direct evaluation of anything other than ground terms and atoms seems tricky (which makes a nice research problem).

An alternative to directly interpreting formulae is to rely on deduction, using the interpretation as axioms, and the formula to be interpreted (or its negation) as a conjecture. This works because, in this setting, logical consequence and directly interpreting formulae coincide in many cases. If $\mathcal{I}$ is a partial interpretation for $\mathcal{L}$, then if $\mathcal{F}$ (in $\mathcal{L}$) is a theorem of (the formulae that define) $\mathcal{I}$ then $\mathcal{I}$ interprets $\mathcal{F}$ as *true*. If $\mathcal{F}$ is a countertheorem of $\mathcal{I}$ (or equivalently, $\tilde{}\mathcal{F}$ is a theorem of $\mathcal{I}$), then $\mathcal{I}$ interprets $\mathcal{F}$ as $false$. Further, if $\mathcal{I}$ is an interpretation (i.e., is complete), and $\mathcal{F}$ is countersatisfiable wrt $\mathcal{I}$ (or equivalently $\tilde{}\mathcal{F}$ is satisfiable with $\mathcal{I}$), then $\mathcal{F}$ is necessarily a countertheorem of $\mathcal{I}$, and $\mathcal{I}$ interprets $\mathcal{F}$ as $false$. Finally, if $\mathcal{I}$ is an interpretation, and $\mathcal{F}$ is satisfiable wrt $\mathcal{I}$ (or equivalently $\tilde{}\mathcal{F}$ is countersatisfiable with $\mathcal{I}$), then $\tilde{}\mathcal{F}$ is necessarily a countertheorem of $\mathcal{I}$, and $\mathcal{F}$ is interpreted as *true*. Note that if $\mathcal{I}$ is a strictly partial interpretation for $\mathcal{L}$ and $\mathcal{F}$ is countersatisfiable wrt $\mathcal{I}$, then nothing can be concluded about the interpretation of $\mathcal{F}$. An example to illustrate this is as follows: Let $\mathcal{L} = [\{a/0, b/0, c/0\}, \{p/1\}]$, and let $\mathcal{I} = \{a \neq b, p(c)\}$. $\mathcal{I}$ is a strictly partial interpretation of $\mathcal{L}$, e.g., it does not interpret the closed formula $\mathcal{F} \equiv b \neq c$. $\mathcal{F}$ is countersatisfiable wrt $\mathcal{I}$, but $\mathcal{F}$ is not a countertheorem of $\mathcal{I}$. These observations provide a simple, but possibly inefficient, way of interpreting a formula wrt a TMTP format interpretation – use an ATP system to determine the status of the formula wrt the model (theorem, countersatisfiable, countertheorem, satisfiable), and hence determine the interpretation of the formula wrt the model (*true*, $false$, $false$, *true*, respectively). An efficient implementation of this idea would try to prove both $\mathcal{F}$ and $\tilde{}\mathcal{F}$ from the interpretation in parallel.

3.2 Viewing, Verifying, and Examining Models

The Interactive Model Viewer (IMV) tool provides an interactive interface for viewing interpretations. IMV aims to provide insights into the structures and features of models, and hence the semantics of axiomatizations. For example, it might be observed that certain domain elements are more or less often the range value of certain functions, or that some argument of a function or predicate does not affect the interpretation.

At the time of writing IMV was still in the design phase, considering different possible visualizations of the different types of interpretations. For finite interpretations for untyped first-order languages, one possible visualization is to provide a term tree for each function and predicate, for each resultant domain value and $true/false$. For example, for the function $f/2$ and the predicate $p/3$, and the finite domain $\mathbb{D} = \{d1, d2, d3\}$, Figure 10 shows the term tree for f resulting in $d1$, and the term tree for p resulting in `true`. Thus one can see that, e.g., $\forall D \in \mathbb{D}\ f(d1, D)$ and $f(d2, d3)$ map to $d1$. Similarly, $p(d1, d1, d1)$, $p(d1, d2, d2)$, $\forall D \in \mathbb{D}\ p(d3, D, d1)$, and $\forall D \in \mathbb{D}\ p(d3, D, d3)$ map to *true*. A possible insight is that the second argument of f does not affect the interpretation when the first argument is interpreted as $d1$. Extensions of this visualization are being considered for other types of interpretations (Herbrand, integer domains, etc.), and more expressive languages (typed, higher-order, etc.).

The GMV[5] tool verifies that an interpretation is a model for a given set of formulae. This

[5]That's "Geoff's Model Verifier".

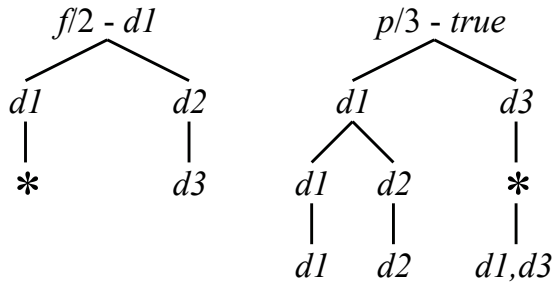

Figure 10: Example Term Trees

is done by checking that each formula in the set is interpreted as *true* in the model, using the techniques discussed in Section 3.1. If an ATP system is used to interpret the formulae wrt the model, it must be a trusted ATP system, and not the one that produced the model. A second approach, used for detecting that an interpretation is not a model of a set of formulae, is to conjoin the model with the set and check for unsatisfiability. If the conjoined set is unsatisfiable, then the interpretation is not a model of the set.

Another tool planned for examining interpretations is a *relationship tester*. This tool will check if two interpretations are syntactic variants of each other (as used when building the TMTP - see Section 2.4), compare the sizes of interpretations in terms of the number of $true/false$ Herbrand base elements, and check whether or not an interpretation makes a superset of Herbrand base elements $true/false$ compared to another interpretation (*interpretation subsumption*). Note that the syntactic-variant test is not the same as an equivalence test – two syntactically distinct interpretations can make the same Herbrand base elements *true*, e.g., they might be different types of interpretations, or might be the same type of interpretation but defined by different sets of formulae.

3.3 The TMTP Online

The TMTP has an online presence, starting at `http://www.tptp.org/TMTP`. The home page provides a linked hierarchy for browsing the TMTP models, and links to other relevant components, including the SystemOnTMTP web interface. SystemOnTMTP allows a model to be submitted to various tools, including parsers for models, evaluation of formulae wrt a model, and GMV. More tools, e.g., IMV, will be added as time goes by. Figure 11 shows the home page and the SystemOnTMTP page.

4 Conclusion

This paper has described the beginnings of the TMTP Model Library, a new component of the TPTP world. The TMTP includes standards for writing interpretations, a library of TPTP format models for TPTP axiomatization problems, and tools for examining and using interpretations.

Future work on the TMTP includes adding more axiomatization problems to the TPTP problem library so that their models are added to the TMTP, researching ways to efficiently interpret formulae wrt an interpretation, implementing the IMV model viewer, extending the GMV verifier, and implementing an interpretation relationship tester.

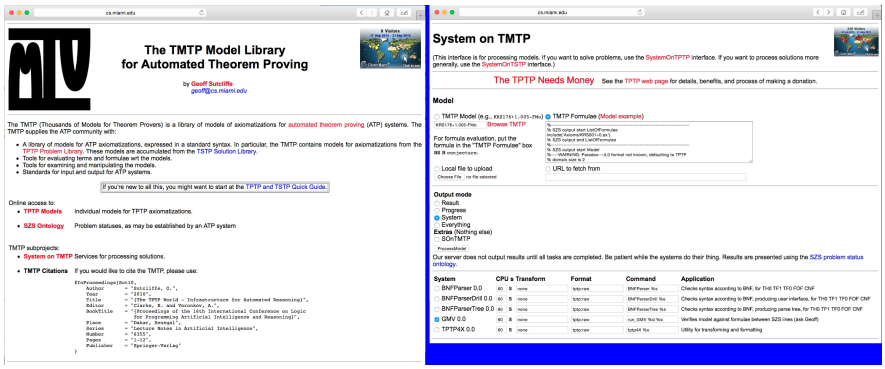

Figure 11: The TMTP and SystemOnTMTP Web pages

When the TMTP and associated tools are in place, they will be available as the basis for the development of semantically guided ATP systems, including the planned implementation of semantic resolution.

References

[1] L. Bachmair and H. Ganzinger. Rewrite-Based Equational Theorem Proving with Selection and Simplification. *Journal of Logic and Computation*, 4(3):217–247, 1994.

[2] L. Bachmair, H. Ganzinger, D. McAllester, and C. Lynch. Resolution Theorem Proving. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, pages 19–99. Elsevier Science, 2001.

[3] C. Barrett, C. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli. CVC4. In G. Gopalakrishnan and S. Qadeer, editors, *Proceedings of the 23rd International Conference on Computer Aided Verification*, number 6806 in Lecture Notes in Computer Science, pages 171–177. Springer-Verlag, 2011.

[4] P. Baumgartner, A. Fuchs, and C. Tinelli. Darwin - A Theorem Prover for the Model Evolution Calculus. In G. Sutcliffe, S. Schulz, and T. Tammet, editors, *Proceedings of the Workshop on Empirically Successful First Order Reasoning, 2nd International Joint Conference on Automated Reasoning*, 2004.

[5] C.E. Brown. Satallax: An Automated Higher-Order Prover (System Description). In B. Gramlich, D. Miller, and U. Sattler, editors, *Proceedings of the 6th International Joint Conference on Automated Reasoning*, number 7364 in Lecture Notes in Artificial Intelligence, pages 111–117, 2012.

[6] F.M. Brown. SLM. Technical Report Internal Memo #72, Department of Artificial Intelligence, University of Edinburgh, Edinburgh, United Kingdom, 1974.

[7] A. Bundy. Personal Correspondence with Geoff Sutcliffe. Available as PDF from Geoff Sutcliffe, 1987.

[8] K. Claessen and N. Sörensson. New Techniques that Improve MACE-style Finite Model Finding. In P. Baumgartner and C. Fermueller, editors, *Proceedings of the CADE-19 Workshop: Model Computation - Principles, Algorithms, Applications*, 2003.

[9] L. de Moura and N. Bjorner. Z3: An Efficient SMT Solver. In C. Ramakrishnan and J. Rehof, editors, *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, number 4963 in Lecture Notes in Artificial Intelligence, pages 337–340. Springer-Verlag, 2008.

[10] K. Korovin and C. Sticksel. iProver-Eq - An Instantiation-Based Theorem Prover with Equality. In J. Giesl and R. Haehnle, editors, *Proceedings of the 5th International Joint Conference on Automated Reasoning*, number 6173 in Lecture Notes in Artificial Intelligence, pages 196–202, 2010.

[11] L. Kovacs and A. Voronkov. First-Order Theorem Proving and Vampire. In N. Sharygina and H. Veith, editors, *Proceedings of the 25th International Conference on Computer Aided Verification*, number 8044 in Lecture Notes in Artificial Intelligence, pages 1–35. Springer-Verlag, 2013.

[12] W.W. McCune. Mace4 Reference Manual and Guide. Technical Report ANL/MCS-TM-264, Argonne National Laboratory, Argonne, USA, 2003.

[13] S. Schulz. System Description: E 1.8. In K. McMillan, A. Middeldorp, and A. Voronkov, editors, *Proceedings of the 19th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, number 8312 in Lecture Notes in Computer Science, pages 477–483. Springer-Verlag, 2013.

[14] J.R. Slagle. Automatic Theorem Proving with Renamable and Semantic Resolution. *Journal of the ACM*, 14(4):687–697, 1967.

[15] J.K. Slaney, E. Lusk, and W.W. McCune. SCOTT: Semantically Constrained Otter, System Description. In A. Bundy, editor, *Proceedings of the 12th International Conference on Automated Deduction*, number 814 in Lecture Notes in Artificial Intelligence, pages 764–768. Springer-Verlag, 1994.

[16] G. Sutcliffe. The Semantically Guided Linear Deduction System. In D. Kapur, editor, *Proceedings of the 11th International Conference on Automated Deduction*, number 607 in Lecture Notes in Artificial Intelligence, pages 677–680. Springer-Verlag, 1992.

[17] G. Sutcliffe. SystemOnTPTP. In D. McAllester, editor, *Proceedings of the 17th International Conference on Automated Deduction*, number 1831 in Lecture Notes in Artificial Intelligence, pages 406–410. Springer-Verlag, 2000.

[18] G. Sutcliffe. Semantic Derivation Verification. *International Journal on Artificial Intelligence Tools*, 15(6):1053–1070, 2006.

[19] G. Sutcliffe. The SZS Ontologies for Automated Reasoning Software. In G. Sutcliffe, P. Rudnicki, R. Schmidt, B. Konev, and S. Schulz, editors, *Proceedings of the LPAR Workshops: Knowledge Exchange: Automated Provers and Proof Assistants, and The 7th International Workshop on the Implementation of Logics*, number 418 in CEUR Workshop Proceedings, pages 38–49, 2008.

[20] G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure. The FOF and CNF Parts, v3.5.0. *Journal of Automated Reasoning*, 43(4):337–362, 2009.

[21] G. Sutcliffe. The TPTP World - Infrastructure for Automated Reasoning. In E. Clarke and A. Voronkov, editors, *Proceedings of the 16th International Conference on Logic for Programming Artificial Intelligence and Reasoning*, number 6355 in Lecture Notes in Artificial Intelligence, pages 1–12. Springer-Verlag, 2010.

[22] G. Sutcliffe. The CADE ATP System Competition - CASC. *AI Magazine*, page To appear, 2015.

[23] G. Sutcliffe and Y. Puzis. SRASS - a Semantic Relevance Axiom Selection System. In F. Pfenning, editor, *Proceedings of the 21st International Conference on Automated Deduction*, number 4603 in Lecture Notes in Artificial Intelligence, pages 295–310. Springer-Verlag, 2007.

[24] G. Sutcliffe, S. Schulz, K. Claessen, and A. Van Gelder. Using the TPTP Language for Writing Derivations and Finite Interpretations. In U. Furbach and N. Shankar, editors, *Proceedings of the 3rd International Joint Conference on Automated Reasoning*, number 4130 in Lecture Notes in Artificial Intelligence, pages 67–81, 2006.

[25] G. Sutcliffe, J. Zimmer, and S. Schulz. TSTP Data-Exchange Formats for Automated Theorem Proving Tools. In W. Zhang and V. Sorge, editors, *Distributed Constraint Problem Solving and Reasoning in Multi-Agent Systems*, number 112 in Frontiers in Artificial Intelligence and Applications, pages 201–215. IOS Press, 2004.

[26] S. Trac, Y. Puzis, and G. Sutcliffe. An Interactive Derivation Viewer. In S. Autexier and

C. Benzmüller, editors, *Proceedings of the 7th Workshop on User Interfaces for Theorem Provers, 3rd International Joint Conference on Automated Reasoning*, volume 174 of *Electronic Notes in Theoretical Computer Science*, pages 109–123, 2006.

[27] C. Weidenbach, A. Fietzke, R. Kumar, M. Suda, P. Wischnewski, and D. Dimova. SPASS Version 3.5. In R. Schmidt, editor, *Proceedings of the 22nd International Conference on Automated Deduction*, number 5663 in Lecture Notes in Artificial Intelligence, pages 140–145. Springer-Verlag, 2009.

A SZS Ontology for Interpretations

The list below provides details for each of the nodes in Figure 1. Each entry gives the full "OneWord" ontology values, its three-letter acronym, and a brief description.

- Interpretation (Int): An interpretation.
- Model (Mod): A model.
- PartialInterpretation (Pin): A partial interpretation.
- PartialModel (PMo): A partial model.
- StrictlyPartialInterpretation (SIn): A strictly partial interpretation.
- StrictlyPartialModel (SMo): A strictly partial model.
- DomainInterpretation (DIn): An interpretation whose domain is not the Herbrand universe.
- DomainModel (DMo): A model whose domain is not the Herbrand universe.
- DomainPartialInterpretation (DPI): A domain interpretation that is partial.
- DomainPartialModel (DPM): A domain model that is partial.
- DomainStrictlyPartialInterpretation (DSI): A domain interpretation that is strictly partial.
- DomainStrictlyPartialModel (DSM): A domain model that is strictly partial.
- FiniteInterpretation: A domain interpretation with a finite domain.
- FiniteModel (FMo): A domain model with a finite domain.
- FinitePartialInterpretation (FPI): A domain partial interpretation with a finite domain.
- FinitePartialModel (FPM): A domain partial model with a finite domain.
- FiniteStrictlyPartialInterpretation (FSI): A domain strictly partial interpretation with a finite domain.
- FiniteStrictlyPartialModel (FSM): A domain strictly partial model with a finite domain.
- IntegerInterpretation: An integer domain interpretation.
- IntegerModel (FMo): An integer domain model.
- IntegerPartialInterpretation (FPI): An integer domain partial interpretation.
- IntegerPartialModel (FPM): An integer domain partial model.
- IntegerStrictlyPartialInterpretation (FSI): An integer domain strictly partial interpretation.
- IntegerStrictlyPartialModel (FSM): An integer domain strictly partial model.
- RealInterpretation: A real domain interpretation.
- RealModel (FMo): A real domain model.
- RealPartialInterpretation (FPI): A real domain partial interpretation.
- RealPartialModel (FPM): A real domain partial model.
- RealStrictlyPartialInterpretation (FSI): A real domain strictly partial interpretation.
- RealStrictlyPartialModel (FSM): A real domain strictly partial model.
- HerbrandInterpretation (HIn): A Herbrand interpretation.
- HerbrandModel (HMo): A Herbrand model.
- FormulaInterpretation (TIn): A Herbrand interpretation defined by a set of TPTP formulae.
- FormulaModel (TMo): A Herbrand model defined by a set of TPTP formulae.
- FormulaPartialInterpretation (TPI): A Herbrand partial interpretation defined by a set of TPTP formulae.
- FormulaPartialModel (TMo): A Herbrand partial model defined by a set of TPTP formulae.
- FormulaStrictlyPartialInterpretation (TSI): A Herbrand strictly partial interpretation defined by a set of TPTP formulae.
- FormulaStrictlyPartialModel (TSM): A Herbrand strictly partial model defined by a set of TPTP formulae.
- Saturation (Sat): A Herbrand model expressed as a saturating set of formulae.

Clausal Proof Compression *

Marijn J.H. Heule[1] and Armin Biere[2]

[1] Department of Computer Science, The University of Texas at Austin, USA marijn@cs.utexas.edu
[2] Institute for Formal Models and Verification, JKU Linz, Austria biere@jku.at

Abstract

Although clausal propositional proofs are significantly smaller compared to resolution proofs, their size on disk is still too large for several applications. In this paper we present several methods to compress clausal proofs. These methods are based on a two phase approach. The first phase consists of a light-weight compression algorithm that can easily be added to satisfiability solvers that support the emission of clausal proofs. In the second phase, we propose to use a powerful off-the-shelf general-purpose compression tool, such as bzip2 and 7z. Sorting literals before compression facilitates a delta encoding, which combined with variable-byte encoding improves the quality of the compression. We show that clausal proofs can be compressed by one order of magnitude by applying both phases.

1 Introduction

Propositional proofs of unsatisfiability come in two flavors: resolution proofs [12] and clausal proofs [10]. An important drawback of using such proofs is their size on disk. Since resolution proofs can be up to two orders of magnitude larger compared to clausal proofs [5], the issue is much more severe for resolution proofs. Even clausal proofs are still too big for some applications, such as computing Van der Waerden number $W(2, 6)$ [8] and the optimal sorting network with ten wires [2]. This paper offers some compression techniques to make them more compact.

The compression techniques presented in this paper are inspired by the binary variant of the AIGER format [1], the input format of the hardware model checking competition. This binary format stores the gates of sequential circuits using a binary representation instead of ASCII characters. Additionally, delta encoding is applied to store the difference between successive numbers. Sorting literals in a clause does not influences validity of proof, but reduces these differences between successive literals — making it a useful pre-compression technique.

Proof compression has many applications. For instance, a restriction to 100GB disk space, the maximal local storage on cluster nodes in the SAT 2014 competition[1], prevented the validation of some proofs of the unsatisfiability tracks. This can be avoided by adding a light-weight compression algorithm to SAT solvers to reduce the size proof lines written to disk. Notice that the 100 GB space limit was per benchmark per solver. Storing all unsatisfiability proofs of the competition is unfeasible even after strong compression.

Clausal proof compression techniques are also useful to store proofs of hard combinatorial problems, such as the Erdős Discrepancy Conjecture (EDP) [7], for which a clausal proof is available. The initial proof was 13GB in size. Using symmetry-breaking methods, a proof of 2GB was produced [4]. In this paper, we show this proof can further be compressed to 128MB (less than 1% of the original proof). For other hard combinatorial problems, such as Van der Waerden numbers and minimal sorting networks, the expected size of (uncompressed) clausal proofs is many terabytes. Compression techniques will be crucial to deal with such proofs.

*This work was supported by the Austrian Science Fund (FWF) through the national research network RiSE (S11408-N23) and the National Science Foundation under grant number CCF-1526760.

[1] results of the certified unsatisfiability tracks at http://satcompetition.org/2014

2 Preliminaries

CNF Satisfiability. For a Boolean variable x, there are two *literals*, the positive literal x and the negative literal $\bar{x}$. A *clause* is a disjunction of literals and a CNF formula a conjunction of clauses. A truth assignment is a function τ that maps literals to $\{\mathbf{f}, \mathbf{t}\}$ under the assumption $\tau(x) = v$ if and only if $\tau(\bar{x}) = \neg v$. A clause C is satisfied by τ if $\tau(l) = \mathbf{t}$ for some literal $l \in C$. An assignment τ satisfies CNF formula F if it satisfies every clause in F.

Resolution and Extended Resolution. The resolution rule states that, given two clauses $C_1 = (x \vee a_1 \vee \ldots \vee a_n)$ and $C_2 = (\bar{x} \vee b_1 \vee \ldots \vee b_m)$, the clause $C = (a_1 \vee \ldots \vee a_n \vee b_1 \vee \ldots \vee b_m)$, can be inferred by resolving on variable x. We say C is the *resolvent* of C_1 and C_2. For a given CNF formula F, the *extension rule* [9] allows one to iteratively add definitions of the form $x := a \wedge b$ by adding the *extended resolution clauses* $(x \vee \bar{a} \vee \bar{b}) \wedge (\bar{x} \vee a) \wedge (\bar{x} \vee b)$ to F, where x is a new variable and a and b are literals in the current formula.

Unit Propagation. The process of *unit propagation* simplifies a CNF F based on unit clauses. It repeats the following until fixpoint: if there is a unit clause $(l) \in F$, remove all clauses containing literal l from set $F \backslash \{(l)\}$ and remove literal $\bar{l}$ from all clauses in F. If unit propagation on formula F produces complementary units (l) and $(\bar{l})$, we say that unit propagation *derives a conflict* and write $F \vdash_1 \epsilon$ with ϵ referring to the (unsatisfiable) empty clause.

Example Consider $F = (a) \wedge (\bar{a} \vee b) \wedge (\bar{b} \vee c) \wedge (\bar{b} \vee \bar{c})$. We have $(a) \in F$, so unit propagation removes $\bar{a}$, resulting in the new unit clause (b). After removal of $\bar{b}$, two complementary unit clauses (c) and $(\bar{c})$ are created. From these two units the empty clause can be derived: $F \vdash_1 \epsilon$.

Clause Redundancy. A clause C is called *redundant* with respect to a formula F iff $F \wedge \{C\}$ is satisfiability equivalent to F. *Asymmetric tautologies*, also known as *reverse unit propagation* clauses, are the most common redundant (learned) clauses in SAT solvers. Let $\overline{C}$ denote the conjunction of unit clauses that falsify all literals in C. A clause C is an asymmetric tautology with respect to a CNF formula F iff $F \wedge \overline{C} \vdash_1 \epsilon$. *Resolution asymmetric tautologies* (or RAT clauses) [6] are a generalization of both asymmetric tautologies and extended resolution clauses. A clause C has RAT on $l \in C$ (referred to as the *pivot* literal) with respect to a formula F if for all $D \in F$ with $\bar{l} \in D$, it holds that $F \wedge \overline{C} \wedge (\overline{D} \setminus \{(l)\}) \vdash_1 \epsilon$. Not only can RAT be computed in polynomial time, but all preprocessing, inprocessing, and solving techniques in state-of-the-art SAT solvers can be expressed in terms of addition and removal of RAT clauses [6].

Clausal Proofs. A *proof of unsatisfiability* (also called a *refutation*) is a sequence of redundant clauses containing the empty clause. It is important that the redundancy property of clauses can be checked in polynomial time. A *DRAT proof*, short for *Deletion Resolution Asymmetric Tautology*, is a sequence of addition and deletion steps of RAT clauses. A DRAT *refutation* is a DRAT proof that contains the empty clause. Figure 1 shows an example DRAT refutation.

Example Let $F = (a \vee b \vee \bar{c}) \wedge (\bar{a} \vee \bar{b} \vee c) \wedge (b \vee c \vee \bar{d}) \wedge (\bar{b} \vee \bar{c} \vee d) \wedge (a \vee c \vee d) \wedge (\bar{a} \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee b \vee d) \wedge (a \vee \bar{b} \vee \bar{d})$, shown in DIMACS format in Fig. 1 (left), where 1 represents a, 2 is b, 3 is c, 4 is d, and negative numbers represent negation. The first clause in the proof, $(\bar{a})$, is a RAT clause with respect to F because all possible resolvents are asymmetric tautologies:

$$F \wedge (a) \wedge (\bar{b}) \wedge (c) \vdash_1 \epsilon \quad \text{using} \quad (a \vee b \vee \bar{c})$$
$$F \wedge (a) \wedge (\bar{c}) \wedge (\bar{d}) \vdash_1 \epsilon \quad \text{using} \quad (a \vee c \vee d)$$
$$F \wedge (a) \wedge (b) \wedge (d) \vdash_1 \epsilon \quad \text{using} \quad (a \vee \bar{b} \vee \bar{d})$$

CNF formula

```
p cnf 4 8
 1  2 -3 0
-1 -2  3 0
 2  3 -4 0
-2 -3  4 0
-1 -3 -4 0
 1  3  4 0
-1  2  4 0
 1 -2 -4 0
```

DRAT proof

```
  -1  0
d -1 -2  3 0
d -1 -3 -4 0
d -1  2  4 0
   2  0
   0
```

Figure 1: Left, a formula in DIMACS CNF format, the conventional input for SAT solvers which starts with `p cnf` to denote the format, followed by the number of variables and the number of clauses. Right, a DRAT proof for that formula. Each line in the proof is either an addition step (no prefix) or a deletion step identified by the prefix "`d`". Spacing in both examples is used to improve readability. Each clause in the proof should be an asymmetric tautology or a RAT clause using the first literal as the pivot.

3 Proof Compression

We propose to compress clausal proofs using two phases. The first phase is a light-weight method which can easily be added to any SAT solver that can produce proofs of unsatisfiability. The second phase applies a strong off-the-shelf compression tool to the result of the first phase.

Byte Encoding The ASCII encoding of clausal proofs in Figure 1 is easy to read, but rather verbose. For example, consider the literal `-123456789`, which requires 11 bytes to express (one for each ASCII character and one for the separating space). This literal can also be represented by a signed integer (4 bytes). If all literals in a proof can be expressed using a signed integer, only 4 bytes are required to encode each literal. Such an encoding also facilitates omitting a byte to express the separation of literals. Consequently, one can easily compress a clausal ASCII proof with a factor of roughly 2.5 by using a binary encoding of literals.

In case the length of literals in the ASCII representation differs a lot, it may not be efficient to allocate a fixed number of bytes to express each literal. Alternatively, the *variable-byte encoding* [11] can be applied, which uses the most significant bit of each byte to denote whether a byte is the last byte required to express a given literal. The variable-byte encoding can express the literal `1234` (`10011010010` in binary notation) using only two bytes: **1**`1010010` **0**`0001001`. (in little-endian ordering, e.g., least-signifiant byte first).

Sorting Literals The order of literals in a clausal proof does not influence validness of the proof, nor does the order influences its size. However, the order of literals can influence the costs to validate a proof as it influences unit propagation and in turn determines which clauses will be marked in backward checking (the default validation algorithm used in clausal proof checkers). The order of literals in the proof produced by the SAT solver is typically not better or worse than any permutation. Experience shows that this is often not the case for SAT solving: the given order of literals in an encoding is generally superior compared to any permutation.

Sorting literals before compression has advantages in both phases. In the first phase, one can use delta encoding: store the difference between two successive literals. Clauses in a proof are typically long (dozens of literals) [5], resulting in a small difference between two successive sorted literals. Delta encoding is particularly useful in combination with variable-byte encoding.

In the second phase, off-the-shelf compression tools could exploit the sorted order of literals. Many clauses in proofs have multiple literals in common. SAT solvers tend to emit literals in a random order. This makes it hard for compression tools to detect overlapping literals between clauses. Sorting literals potentially increases the observability of overlap which in turn could increase the quality of the compression algorithm.

Table 1: Eight encodings of an example DRAT proof line. The first two encodings are shown as ASCII text using decimal numbers, while the last six are shown as hexadecimals using the MiniSAT encoding of literals. The prefix s denotes sorted, while the prefix ds denotes delta encoding after sorted. 4byte denotes that 4 bytes are used to represent each literal, while vbyte denotes that variable-byte encoding is used.

encoding	**example** (prefix pivot $\text{lit}_1 ... \text{lit}_{k-1}$ end)	**#bytes**
ascii	`d 6278 -3425 -42311 9173 22754 0\n`	33
sascii	`d 6278 -3425 9173 22754 -42311 0\n`	33
4byte	`64 0c 31 00 00 c3 1a 00 00 8f 4a 01 00 aa 47 00 00 c4 b1 00 00 00 00 00 00`	25
s4byte	`64 0c 31 00 00 c3 1a 00 00 aa 47 00 00 c4 b1 00 00 8f 4a 01 00 00 00 00 00`	25
ds4byte	`64 0c 31 00 00 c3 1a 00 00 e8 2c 00 00 1a 6a 00 00 cb 98 00 00 00 00 00 00`	25
vbyte	`64 8c 62 c3 35 8f 95 05 aa 8f 01 c4 e3 02 00`	15
svbyte	`64 8c 62 c3 35 aa 8f 01 c4 e3 02 8f 95 05 00`	15
dsvbyte	`64 8c 62 c3 35 e8 59 9a d4 01 cb b1 02 00`	14

Literal Encoding In most SAT solvers, literals are mapped to natural numbers. The default mapping function $map(l)$, introduced in MiniSAT [3] and also used in the AIGER format [1] converts signed DIMACS literals into unsigned integer numbers as follows:

$$map(l) = \begin{cases} 2l+1 & \text{if } l > 0 \\ -2l & \text{otherwise} \end{cases}$$

Table 1 shows a DRAT proof line in the conventional DIMACS and in several binary encodings. For all non-ASCII encodings, we will use $map(l)$ to represent literals. Notice that the first literal in the example is not sorted, because the proof checker needs to know the pivot literal (which is the first literal in each clause). The remaining literals are sorted based on their $map(l)$ value.

4 Experiments

We implemented two tools: `ratz` (encode) and `ztar` (decode)[2]. We used `ratz` to transform DRAT proofs in the ASCII format to several alternative representations and applied off-the shelf compression tools to make the resulting files more compact. The compression tools used during the experiments are `gzip`, `bzip2`, and `7zip`. Due to space limitations the experiments focus on a single proof: a trimmed (i.e., removed redundant lines) DRAT proof [3] for Erdős Discrepancy Problem [7] based on symmetry-breaking [4]. We selected a trimmed proof, because in practice one wants to remove redundancy before compression.

Table 2 shows the results. The second column shows that delta encoding combined with sorting and variable-byte encoding (last row of the table) reduces proof size by already more than a factor of four in 25 seconds. This significant and efficient compression can easily be added to any SAT solver that can produce clausal proofs, thereby reducing the space burden. The results of the second phase, i.e., using off-the-shelf compression tools, are less clear. The smallest file is produced by sorting and variable-byte encoding followed by 7zip. Delta encoding, although it reduces the size in combination with variable-byte encoding, appears to be obstruct all the compression tools.

[2] available at `http://fmv.jku.at/ratz`
[3] available at `http://www.cs.utexas.edu/~marijn/sbp`

Table 2: Size of a trimmed DRAT proof (in bytes) and the conversion costs (wall-clock time in seconds) for Erdős Discrepancy Problem using different encodings and compression algorithms. A four core Intel Xeon E31280 @ 3.50GHz with 32GB memory was used for the experiments. The tool 7z used all cores, while the other programs used only a single core.

encoding	**first phase**	**gzip**	**bzip2**	**7z**
ascii	1,719,002,352 (——)	224,505,003 (58.68)	186,871,192 (183.63)	176,740,892 (173.44)
sascii	1,719,002,352 (48.75)	199,368,062 (51.43)	153,589,408 (204.46)	155,268,644 (162.71)
4byte	1,282,405,483 (19.46)	205,093,278 (47.75)	182,221,318 (98.61)	163,176,124 (114.07)
s4byte	1,282,405,483 (27.28)	179,853,433 (39.46)	144,742,387 (116.32)	141,086,084 (109.31)
ds4byte	1,282,405,483 (27.07)	210,994,395 (49.49)	168,958,717 (86.05)	157,274,204 (121.58)
vbyte	639,781,147 (12.76)	183,079,542 (24.70)	183,254,546 (58.39)	149,944,476 (66.89)
svbyte	639,781,147 (20.07)	158,535,823 (22.72)	146,177,432 (63.44)	128,300,756 (66.69)
dsvbyte	403,398,345 (17.97)	157,295,747 (16.15)	165,947,521 (45.96)	136,576,424 (40.42)

5 Conclusion

We proposed several compression techniques for clausal proofs. In particular the combination of delta and variable-byte encoding is very useful to make proofs more compact. Both techniques can easily be added to SAT solvers, which would hardly increase the costs to emit a clausal proof. Off-the-shelf compression tools can be used to further reduce the proof size. Combining both phases on a proof of Erdős Discrepancy Problem shows a compression of over 93%.

References

[1] Armin Biere. The AIGER and-inverter graph (AIG) format, version 20070427, 2007.

[2] Michael Codish, Luís Cruz-Filipe, Michael Frank, and Peter Schneider-Kamp. Twenty-five comparators is optimal when sorting nine inputs (and twenty-nine for ten). In *ICTAI 2014*, pages 186–193. IEEE Computer Society, 2014.

[3] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, volume 2919 of *LNCS*, pages 502–518. Springer, 2003.

[4] Marijn J. H. Heule, Jr. Hunt, Warren A., and Nathan Wetzler. Expressing symmetry breaking in DRAT proofs. In *CADE-25*, volume 9195 of *LNCS*, pages 591–606. Springer, 2015.

[5] Marijn J. H. Heule, Warren A. Hunt, Jr., and Nathan Wetzler. Trimming while checking clausal proofs. In *Formal Methods in Computer-Aided Design*, pages 181–188. IEEE, 2013.

[6] Matti Järvisalo, Marijn Heule, and Armin Biere. Inprocessing rules. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *IJCAR*, volume 7364 of *LNCS*, pages 355–370. Springer, 2012.

[7] Boris Konev and Alexei Lisitsa. A SAT attack on the Erdős Discrepancy Conjecture. In *SAT 2014*, volume 8561 of *LNCS*, pages 219–226. Springer, 2014.

[8] Michal Kouril and Jerome L. Paul. The van der Waerden number W(2, 6) is 1132. *Experimental Mathematics*, 17(1):53–61, 2008.

[9] Grigori S. Tseitin. On the complexity of derivation in propositional calculus. In *Automation of Reasoning 2*, pages 466–483. Springer, 1983.

[10] Allen Van Gelder. Verifying RUP proofs of propositional unsatisfiability. In *ISAIM*, 2008.

[11] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes (2Nd Ed.): Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers Inc., San Francisco, CA, 1999.

[12] Lintao Zhang and Sharad Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *DATE*, pages 10880–10885, 2003.

Implementing Polymorphism in Zenon *

Guillaume Bury, Raphaël Cauderlier and Pierre Halmagrand

Cedric/Cnam/Inria, Paris, France,
Guillaume.Bury@inria.fr Raphael.Cauderlier@inria.fr Pierre.Halmagrand@inria.fr

Abstract

Extending first-order logic with ML-style polymorphism allows to define generic axioms dealing with several sorts. Until recently, most automated theorem provers relied on preprocess encodings into mono/many-sorted logic to reason within such theories. In this paper, we discuss the implementation of polymorphism into the first-order tableau-based automated theorem prover Zenon. This implementation led us to modify some basic parts of the code, from the representation of expressions to the proof-search algorithm.

1 Introduction

Formal verification tends to be a common milestone in the development of software for safety-critical systems. Among the family of verification techniques, those using automated deductive tools raised confidence to a high level during last decades and are now used in a wide range of fields. These achievements were made possible thanks to the ability of such automated deductive tools to reason on specific theories, like set theory or arithmetic for instance, helped by an efficient decision procedure for each theory. When reasoning in several theories combining different sorts, it is often necessary to express some general axioms regarding all different sorts. One solution is to postulate one axiom per sort, leading to a multiplication of axioms. Another solution is to express axioms in a generic way. First-order logic extended with ML-style polymorphism (FOL-ML) is a good candidate to address this issue.

Until recently, most automated deductive tools, like automated theorem provers (ATP) or SMT solvers, were not handling polymorphism. Whenever someone wanted to use such an ATP or SMT solver to prove statements coming from a FOL-ML theory, he relied on a preprocessing phase to encode into a mono/many-sorted logic [1]. Such encodings generally modify theories by deconstructing the shape of formulas and adding some new axioms, leading to less efficient proof search. A solution to keep the original form of the input theory and the statement is to develop some deductive tools which natively understand polymorphism. As far as the authors know, implementation of polymorphism into an automated deductive tool began with the development of the SMT solver Alt-Ergo [3]. Two other projects have since been released, both based on superposition. The first one is a prototype based on the prover SPASS [8], and the other one is the new ATP Zipperposition [7].

Zenon [4] is a first-order monosorted tableau-based ATP. We present in this paper some insights about the implementation of polymorphism into Zenon. This extension has required to properly adapt a large part of the existing code, from the representation of expressions to the proof-search algorithm.

This paper is organized as follows: in Sec. 2, we describe the new syntax for typed expressions, in Sec. 3, we give the type-checking algorithm, and finally in Sec. 4, we discuss modifications in the proof-search algorithm and give the results of a benchmark.

*This work has received funding from the BWare project (ANR-12-INSE-0010) funded by the INS programme of the French National Research Agency (ANR).

Expressions

$a, b, \ldots$	$::=$	v	*(typed variable)*
	$\mid$	A	*(metavariable)*
	$\mid$	$\epsilon(e)$	*(Hilbert's operator)*
	$\mid$	$f(a_1, \ldots, a_n)$	*(function/predicate application)*
	$\mid$	$a_1 \to \ldots \to a_n \to b$	*(arrow type)*
	$\mid$	$\top \mid \bot$	*(true and false)*
	$\mid$	$a = b$	*(equality)*
	$\mid$	$\neg a \mid a \wedge b \mid a \vee b \mid a \Rightarrow b \mid a \Leftrightarrow b$	*(logical connectives)*
	$\mid$	$\forall v : a.\ b$	*(universal quantification on variables)*
	$\mid$	$\exists v : a.\ b$	*(existential quantification on variables)*

Figure 1: AST for types, terms and formulas

2 Syntax of typed expressions

In Zenon, both terms and formulas are represented using a single abstract syntax tree, presented in Figure 1. This decision follows from the use of Hilbert's operator to handle existentially quantified formulas, which introduces terms that depend on formulas. When we implemented typed expressions in Zenon, we chose to extend that single abstract syntax tree (AST) with the arrow type constructor, so that it could also represent types, rather than introduce another AST for types. This allowed us to minimize modifications to the code base, as well as reuse existing code and benefit from features already implemented such as hashconsing and substitutions.

In our implementation, each function/predicate and node of the AST is tagged with an optional type (itself built using the same AST). We then have four distinct classes of expressions built using the AST :

- A constant Type, with an empty tag
- Types are built using variables, meta-variables, and ϵ-terms with tag Type, universal quantification[1], the arrow type constructor, and application of type constructors, i.e functions whose type is of the form: Type $\to \ldots \to$ Type $\to$ Type. Types are tagged with Type.
- Terms are built using variables, meta-variables, ϵ-terms and application of functions. Terms are tagged with a type.
- Formulas are built using $\top$, $\bot$, equality of terms, application of predicates, logical connectives, universal quantification and existential quantification of formulas. Formulas are tagged with a type constant Prop.

We then have access to the type of expressions through the get_type function, which returns either Type, or a type.

3 Type checking

For efficiency reasons, the type-checking phase in Zenon occurs before the beginning of proof search so that expressions are checked once and for all. Since the equality relation uses implicit

[1] Universal quantification in types is used to represent the type of polymorphic functions and predicates.

$$\frac{(v:\tau)\in\Gamma}{\Sigma,\Gamma\vdash v\Rightarrow\tau}\qquad\frac{(f:\forall\alpha_1.\ \ldots\forall\alpha_n.\ \tau'_1\to\ldots\to\tau'_m\to\tau)\in\Sigma}{\Sigma,\Gamma\vdash f(\tau_1,\ldots,\tau_n,a_1,\ldots,a_m)\Rightarrow\tau\{\alpha_1:=\tau_1,\ldots,\alpha_n:=\tau_n\}}$$

$$\frac{(v:\tau)\in\Gamma}{\Sigma,\Gamma\vdash v\Leftarrow\tau}\qquad\frac{\Sigma,\Gamma\vdash a\Rightarrow\tau\qquad\Sigma,\Gamma\vdash a\Leftarrow\tau\qquad\Sigma,\Gamma\vdash b\Leftarrow\tau}{\Sigma,\Gamma\vdash a=b\Leftarrow\mathsf{Prop}}$$

$$\frac{c\in\{\top,\bot\}}{\Sigma,\Gamma\vdash c\Leftarrow\mathsf{Prop}}\qquad\frac{\Sigma,\Gamma\vdash a\Leftarrow\mathsf{Prop}}{\Sigma,\Gamma\vdash\neg a\Leftarrow\mathsf{Prop}}\qquad\frac{\square\in\{\wedge,\vee,\Rightarrow,\Leftrightarrow\}\ \Sigma,\Gamma\vdash a\Leftarrow\mathsf{Prop}\ \Sigma,\Gamma\vdash b\Leftarrow\mathsf{Prop}}{\Sigma,\Gamma\vdash a\square b\Leftarrow\mathsf{Prop}}$$

$$\frac{Q\in\{\forall,\exists\}\qquad\tau=\mathsf{Type}\text{ or }\Sigma,\Gamma\vdash\tau\Leftarrow\mathsf{Type}\qquad\Sigma,\Gamma,v:\tau\vdash a\Leftarrow\mathsf{Prop}}{\Sigma,\Gamma\vdash Qv:\tau.\ a\Leftarrow\mathsf{Prop}}$$

$$\frac{\Sigma,\Gamma,v:\mathsf{Type}\vdash a\Leftarrow\mathsf{Type}}{\Sigma,\Gamma\vdash\forall v:\mathsf{Type}.\ a\Leftarrow\mathsf{Type}}\qquad\frac{\Sigma,\Gamma\vdash\tau_i\Leftarrow\mathsf{Type}\qquad\text{for }i\in[0,n]}{\Sigma,\Gamma\vdash\tau_1\to\ldots\to\tau_n\to\tau_0\Leftarrow\mathsf{Type}}$$

$$\frac{\begin{array}{c}(f:\forall\alpha_1.\ \ldots\forall\alpha_n.\ \tau'_1\to\ldots\to\tau'_m\to\tau'_0)\in\Sigma\\ \sigma:=\{\alpha_1:=\tau_1,\ldots,\alpha_n:=\tau_n\}\\ \sigma(\tau'_0)=\tau_0\end{array}\qquad\begin{array}{c}\Sigma,\Gamma\vdash\tau_i\Leftarrow\mathsf{Type}\\ \Sigma,\Gamma\vdash a_i\Leftarrow\sigma(\tau'_i)\end{array}\qquad\text{for all }i\in[1,n]}{\Sigma,\Gamma\vdash f(\tau_1,\ldots,\tau_n,a_1,\ldots,a_m)\Leftarrow\tau_0}$$

Figure 2: Zenon's type-checking algorithm

polymorphic typing, we require each quantifier in the input problem to specify the type of the variables it binds (otherwise, formulas such as $\forall x.\ x = x$ would be ambiguous) and each function, type constructor, and predicate symbol to be declared with its type (this is required to type formulas such as $f(0) = f(1)$). Since equality is the only implicitly polymorphic symbol, we do not really need to infer types for all expressions but only for terms. We denote by $\Sigma,\Gamma\vdash t\Rightarrow\tau$ the functional relation mapping a term to its inferred type and by $\Sigma,\Gamma\vdash a\Leftarrow\tau$ the type-checking relation. These relations are defined in Figure 2 using a syntax-directed set of typing rules.

We do not need to give rules for expressions which are not present in the input problem such as meta variables and Hilbert's epsilons. Moreover, we only need to define inference for terms. Therefore it can be done by a single look-up in Σ or Γ. However, inference can return a type for an ill-typed term because it does not take subterms into account; this is the reason why in the rule for checking equality, we check back that a has the type returned by the inference machinery.

3.1 Typing substitutions

During proof search, the only way to generate ill-typed expressions is by applying substitutions, for example in the rules instantiating quantifiers and unfolding definitions. To avoid this issue, we check that substitutions are well-typed. However, there are at least two ways to define the notion of well-typed substitution:

- Strongly well-typed substitution:
  The substitution $\sigma = \{x_1 := t_1, \ldots, x_n := t_n\}$ is strongly well-typed if each x_i has the same type as t_i
- Weakly well-typed substitution:
  The substitution $\sigma = \{x_1 := t_1, \ldots, x_n := t_n\}$ is weakly well-typed if each $x_i\{x_1 := t_1, \ldots, x_{i-1} := t_{i-1}\}$ has the same type as $t_i\{x_1 := t_1, \ldots, x_{i-1} := t_{i-1}\}$

The function performing substitutions in Zenon does not preserve strong well-typedness in its recursive calls but only weak well-typedness which is, fortunately, enough to guarantee that applying the substitution on a well-typed expression will result in a well-typed expression.

Strong well-typedness is faster to check because we only need to traverse the list $[(x_1, t_1); \ldots; (x_n, t_n)]$ once, leading to linear complexity. Checking that two terms have the same type is performed in essentially constant time thanks to the type annotations of all expression nodes (hence obtaining the types is fast) and hashconsing (hence comparing expressions is most of the time as fast as comparing their hashes). On the other hand, checking weak well-typedness is of quadratic complexity.

As a compromise between safety and efficiency we distinguish two substitution functions: the old one, `substitute_unsafe`, preserving weak well-typedness in its recursive calls but not performing any typing check; and a wrapper function `substitute_safe` checking[2] that the substitution it gets as argument is strongly well-typed and then calling `substitute_unsafe`.

In new version of Zenon extended with typing, only `substitute_safe` is used during proof search and other parts of the new Zenon which need to substitute in well-typed expressions.

4 Proof Search

4.1 Dealing with Type Metavariables

Extension of Zenon to polymorphism slightly modifies the implementation of the proof-search algorithm. The main modification deals with universal quantification over type variables. When Zenon encounters a universally quantified formula φ, it generates a so-called metavariable linking to φ by applying the $\delta_{\forall M}$ rule [5]. The original formula φ is kept in the context for later instantiation.

For metavariables linked to quantified formulas over terms, the behavior of Zenon has not changed: such metavariables are only used as tricks to find, by unification, some possible instantiations for the original formulas that allow to close the local branches. After finding a relevant value, Zenon instantiates the original formula by applying the $\delta_{\forall_{\text{inst}}}$ rule and continues its proof search.

For type metavariables, we have to instantiate original formulas as soon as possible, when it makes possible the application of a further rule. Actually, only the relational rules of Zenon (those dealing with the equality symbol) are concerned, because the possibility to apply them depend on side conditions over the type of their parameters [5]. So, if we have some type metavariables in a formula and if there are some possible instantiations that allow to apply one of these rules, we instantiate the original formulas linked to the metavariables. In such a way, we ensure to capture all the possible instantiations needed for the proof search.

4.2 Experimental Results

To assess our extension of Zenon to polymorphism, we performed an experiment using a benchmark made of all the 337 problems with a theorem status coming from the TFF1 [2] category of the TPTP library, run on an Intel Xeon E5-2660 v2 2.20 GHz computer, with a timeout of 30 s and a memory limit of 1 GiB. We compare the new typed version of Zenon[3] presented in this paper with the previous monosorted one (using the encoding of types into pure first-order logic [1] implemented into the Why3 platform) and the other automated deductive tools dealing with

[2] these checks can also be disabled

[3] Available at: `https://www.rocq.inria.fr/deducteam/ZenonModulo/`.

337 Problems	Zenon Old	Zenon Typed	Zipperposition	Alt-Ergo
Proved	96	106	150	221
Mean Time (sec.)	1.9	0.95	3.3	0.64

Table 1: Experimental Results over the TPTP TFF1 Benchmark

polymorphism, Zipperposition v0.6.1 and Alt-Ergo v0.99.1, except the prototype based on SPASS which does not yet read TFF1 syntax. The results are summarized in Tab. 1 and, for each prover, they give the number of proved problems and the mean time needed to prove a problem. This experiment shows that the polymorphic version of Zenon proves 10 more problems than the monosorted one while being twice as fast. On the other side, the superposition-based ATP Zipperposition proves 44 more problems than Zenon with a larger mean time and the SMT solver Alt-Ergo proves 115 more problems with a lower mean time.

5 Conclusion

We have extended the automated theorem prover Zenon to polymorphism. Since we were adapting an existing code, we chose to minimize the impact of this extension to the original structure of Zenon. The experimental results, presented in this paper, show that the polymorphic version of Zenon is more efficient than the monosorted one on polymorphic problems.

Considering the significance of polymorphism in program verification, we hope that more provers will integrate expressive typing systems in the future, especially since it does not affect the generation of proof certificates [6] because proof checkers usually provide rich typing systems.

References

[1] J. C. Blanchette, S. Böhme, A. Popescu, and N. Smallbone. Encoding monomorphic and polymorphic types. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2013.

[2] J. C. Blanchette and A. Paskevich. TFF1: The TPTP Typed First-Order Form with Rank-1 Polymorphism. In *Conference on Automated Deduction (CADE)*. Springer, 2013.

[3] F. Bobot, S. Conchon, E. Contejean, and S. Lescuyer. Implementing Polymorphism in SMT solvers. In *SMT 2008: 6th International Workshop on Satisfiability Modulo*, 2008.

[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 (LPAR)*. Springer, 2007.

[5] G. Bury, D. Delahaye, D. Doligez, P. Halmagrand, and O. Hermant. Automated Deduction in the B Set Theory using Typed Proof Search and Deduction Modulo. In *Logic for Programming Artificial Intelligence and Reasoning (LPAR)*, 2015.

[6] R. Cauderlier and P. Halmagrand. Checking Zenon Modulo proofs in Dedukti. In *Proof Exchange for Theorem Proving (PxTP)*, EPTCS, Aug. 2015.

[7] S. Cruanes. *Extending Superposition with Integer Arithmetic, Structural Induction, and Beyond*. PhD thesis, Ecole Polytechnique, 2015.

[8] D. Wand. Polymorphic+Typeclass Superposition. In B. Konev, L. de Moura, and S. Schulz, editors, *4th Workshop on Practical Aspects of Automated Reasoning (PAAR 2014)*, Vienna, Austria, 2014.

Well-founded Functions and Extreme Predicates in Dafny: A Tutorial

K. Rustan M. Leino

Microsoft Research
leino@microsoft.com

Abstract

A recursive function is well defined if its every recursive call corresponds a decrease in some well-founded order. Such *well-founded functions* are useful for example in computer programs when computing a value from some input. A boolean function can also be defined as an extreme solution to a recurrence relation, that is, as a least or greatest fixpoint of some functor. Such *extreme predicates* are useful for example in logic when encoding a set of inductive or coinductive inference rules. The verification-aware programming language Dafny supports both well-founded functions and extreme predicates. This tutorial describes the difference in general terms, and then describes novel syntactic support in Dafny for defining and proving lemmas with extreme predicates. Various examples and considerations are given. Although Dafny's verifier has at its core a first-order SMT solver, Dafny's logical encoding makes it possible to reason about fixpoints in an automated way.

0. Introduction

Recursive functions are a core part of computer science and mathematics. Roughly speaking, when the definition of such a function spells out a terminating computation from given arguments, we may refer to it as a *well-founded function*. For example, the common factorial and Fibonacci functions are well-founded functions. There are also other ways to define functions. An important case regards the definition of a boolean function as an extreme solution (that is, a least or greatest solution) to some equation. For computer scientists with interests in logic or programming languages, these *extreme predicates* are important because they describe the judgments that can be justified by a given set of inference rules (see, e.g., [2, 17, 20, 24, 27]).

To benefit from machine-assisted reasoning, it is necessary not just to understand extreme predicates but also to have techniques for proving theorems about them. A foundation for this reasoning was developed by Paulin-Mohring [22] and is the basis of the constructive logic supported by Coq [0] as well as other proof assistants [1, 25]. Essentially, the idea is to represent the knowledge that an extreme predicate holds by the proof term by which this knowledge was derived. For a predicate defined as the least solution, such proof terms are values of an inductive datatype (that is, finite proof trees), and for the greatest solution, a coinductive datatype (that is, possibly infinite proof trees). This means that one can use induction and coinduction when reasoning about these proof trees. Therefore, these extreme predicates are known as, respectively, *inductive predicates* and *coinductive predicates* (or, *co-predicates* for short). Support for extreme predicates is also available in the proof assistants Isabelle [23] and HOL [5].

In this paper, I give my own tutorial account on the distinction between well-founded functions and extreme predicates. I also show how the verification-aware programming language Dafny [12] sets these up to obtain automation from an underlying first-order (that is, fixpoint and induction ignorant) SMT solver. The encoding for coinductive predicates in Dafny was described previously [15]. The present paper adds inductive predicates (which are duals of the coinductive ones), new syntactic shorthands (based on the experience of using inductive predicates in Dafny), and examples.

1. Function Definitions

To define a function $f : X \to Y$ in terms of itself, one can write an equation like

$$f \quad = \quad \mathcal{F}(f) \tag{0}$$

where $\mathcal{F}$ is a non-recursive function of type $(X \to Y) \to X \to Y$. Because it takes a function as an argument, $\mathcal{F}$ is referred to as a *functor* (or *functional*, but not to be confused by the category-theory notion of a functor). Throughout, I will assume that $\mathcal{F}(f)$ by itself is well defined, for example that it does not divide by zero. I will also assume that f occurs only in fully applied calls in $\mathcal{F}(f)$; eta expansion can be applied to ensure this. If f is a boolean function, that is, if Y is the type of booleans, then I call f a *predicate*.

For example, the common Fibonacci function over the natural numbers can be defined by the equation

$$\mathrm{fib} \quad = \quad \lambda n \bullet \text{ if } n < 2 \text{ then } n \text{ else } \mathrm{fib}(n-2) + \mathrm{fib}(n-1) \tag{1}$$

With the understanding that the argument n is universally quantified, we can write this equation equivalently as

$$\mathrm{fib}(n) \quad = \quad \text{if } n < 2 \text{ then } n \text{ else } \mathrm{fib}(n-2) + \mathrm{fib}(n-1) \tag{2}$$

The fact that the function being defined occurs on both sides of the equation causes concern that we might not be defining the function properly, leading to a logical inconsistency. In general, there could be many solutions to an equation like (0) or there could be none. Let's consider two ways to make sure we're defining the function uniquely.

1.0. Well-founded Functions

A standard way to ensure that equation (0) has a unique solution in f is to make sure the recursion is well-founded, which roughly means that the recursion terminates. This is done by introducing any well-founded relation $\ll$ on the domain of f and making sure that the argument to each recursive call goes down in this ordering. More precisely, if we formulate (0) as

$$f(x) \quad = \quad \mathcal{F}'(f) \tag{3}$$

then we want to check $E \ll x$ for each call $f(E)$ in $\mathcal{F}'(f)$. When a function definition satisfies this *decrement condition*, then the function is said to be *well-founded*.

For example, to check the decrement condition for fib in (2), we can pick $\ll$ to be the arithmetic less-than relation on natural numbers and check the following, for any n:

$$2 \le n \quad \Longrightarrow \quad n-2 \ll n \quad \wedge \quad n-1 \ll n \tag{4}$$

Note that we are entitled to using the antecedent $2 \le n$, because that is the condition under which the else branch in (2) is evaluated.

A well-founded function is often thought of as "terminating" in the sense that the recursive *depth* in evaluating f on any given argument is finite. That is, there are no infinite descending chains of recursive calls. However, the evaluation of f on a given argument may fail to terminate, because its *width* may be infinite. For example, let P be some predicate defined on the ordinals and let $\mathrm{PDownward}$ be a predicate on the ordinals defined by the following equation:

$$\mathrm{PDownward}(o) \quad = \quad P(o) \ \wedge \ \forall p \bullet \ p \ll o \implies \mathrm{PDownward}(p) \tag{5}$$

With $\ll$ as the usual ordering on ordinals, this equation satisfies the decrement condition, but evaluating $\mathrm{PDownward}(\omega)$ would require evaluating $\mathrm{PDownward}(n)$ for every natural number n. However, what we are concerned about here is to avoid mathematical inconsistencies, and that is indeed a consequence of the decrement condition.

1.0.0. Example with Well-founded Functions

So that we can later see how inductive proofs are done in Dafny, let's prove that for any n, $\mathrm{fib}(n)$ is even iff n is a multiple of 3. We split our task into two cases. If $n < 2$, then the property follows directly from the definition of fib. Otherwise, note that exactly one of the three numbers $n-2$, $n-1$, and n is a multiple of 3. If n is the multiple of 3, then by invoking the induction hypothesis on $n-2$ and $n-1$, we obtain that $\mathrm{fib}(n-2) + \mathrm{fib}(n-1)$ is the sum of two odd numbers, which is even. If $n-2$ or $n-1$ is a multiple of 3, then by invoking the induction hypothesis on $n-2$ and $n-1$, we obtain that $\mathrm{fib}(n-2) + \mathrm{fib}(n-1)$ is the sum of an even number and an odd number, which is odd. In this proof, we invoked the induction hypothesis on $n-2$ and on $n-1$. This is allowed, because both are smaller than n, and hence the invocations go down in the well-founded ordering on natural numbers.

1.1. Extreme Solutions

We don't need to exclude the possibility of equation (0) having multiple solutions—instead, we can just be clear about which one of them we want. Let's explore this, after a smidgen of lattice theory.

For any complete lattice $(Y, \leq)$ and any set X, we can by *pointwise extension* define a complete lattice $(X \to Y,\ \dot{\Rightarrow}\ )$, where for any $f, g\colon X \to Y$,

$$f\ \dot{\Rightarrow}\ g\ \ \equiv\ \ \forall x \bullet\ f(x) \leq g(x) \tag{6}$$

In particular, if Y is the set of booleans ordered by implication ($\mathrm{false} \leq \mathrm{true}$), then the set of predicates over any domain X forms a complete lattice. Tarski's Theorem [26] tells us that any monotonic function over a complete lattice has a least and a greatest fixpoint. In particular, this means that $\mathcal{F}$ has a least fixpoint and a greatest fixpoint, provided $\mathcal{F}$ is monotonic.

Speaking about the *set of solutions* in f to (0) is the same as speaking about the *set of fixpoints* of functor $\mathcal{F}$. In particular, the least and greatest solutions to (0) are the same as the least and greatest fixpoints of $\mathcal{F}$. In casual speak, it happens that we say "fixpoint of (0)", or more grotesquely, "fixpoint of f" when we really mean "fixpoint of $\mathcal{F}$".

In conclusion of our little excursion into lattice theory, we have that, under the proviso of $\mathcal{F}$ being monotonic, the set of solutions in f to (0) is nonempty, and among these solutions, there is in the $\dot{\Rightarrow}$ ordering a least solution (that is, a function that returns false more often than any other) and a greatest solution (that is, a function that returns true more often than any other).

When discussing extreme solutions, I will now restrict my attention to boolean functions (that is, with Y being the type of booleans). Functor $\mathcal{F}$ is monotonic if the calls to f in $\mathcal{F}'(f)$ are in *positive positions* (that is, under an even number of negations). Indeed, from now on, I will restrict my attention to such monotonic functors $\mathcal{F}$.

Let me introduce a running example. Consider the following equation, where x ranges over the integers:

$$g(x)\ \ =\ \ (x = 0\ \vee\ g(x-2)) \tag{7}$$

This equation has four solutions in g. With w ranging over the integers, they are:

$$\begin{array}{lcl} g(x) & \equiv & x \in \{w \mid 0 \leq w\ \wedge\ w \text{ even}\} \\ g(x) & \equiv & x \in \{w \mid w \text{ even}\} \\ g(x) & \equiv & x \in \{w \mid (0 \leq w\ \wedge\ w \text{ even})\ \vee\ w \text{ odd}\} \\ g(x) & \equiv & x \in \{w \mid \mathrm{true}\} \end{array} \tag{8}$$

The first of these is the least solution and the last is the greatest solution.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{g(0)}}{g(2)}}{g(4)}}{g(6)} \qquad\qquad \dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\;\;g(-5)\;\;}}{g(-3)}}{g(-1)}}{g(1)}$$

Figure 0. Left: a finite proof tree that uses the rules of (9) to establish $g(6)$. Right: an infinite proof tree that uses the rules of (10) to establish $g(1)$.

In the literature, the definition of an extreme predicate is often given as a set of *inference rules*. To designate the least solution, a single line separating the antecedent (on top) from conclusion (on bottom) is used:

$$\dfrac{}{g(0)} \qquad\qquad \dfrac{g(x-2)}{g(x)} \tag{9}$$

Through repeated applications of such rules, one can show that the predicate holds for a particular value. For example, the *derivation*, or *proof tree*, to the left in Figure 0 shows that $g(6)$ holds. (In this simple example, the derivation is a rather degenerate proof "tree".) The use of these inference rules gives rise to a least solution, because proof trees are accepted only if they are *finite*.

When inference rules are to designate the greatest solution, a double line is used:

$$\dfrac{}{\;\;g(0)\;\;} \qquad\qquad \dfrac{g(x-2)}{\;\;g(x)\;\;} \tag{10}$$

In this case, proof trees are allowed to be infinite. For example, the (partial depiction of the) infinite proof tree on the right in Figure 0 shows that $g(1)$ holds.

Note that derivations may not be unique. For example, in the case of the greatest solution for g, there are two proof trees that establish $g(0)$: one is the finite proof tree that uses the left-hand rule of (10) once, the other is the infinite proof tree that keeps on using the right-hand rule of (10).

1.2. Working with Extreme Predicates

In general, one cannot evaluate whether or not an extreme predicate holds for some input, because doing so may take an infinite number of steps. For example, following the recursive calls in the definition (7) to try to evaluate $g(7)$ would never terminate. However, there are useful ways to establish that an extreme predicate holds and there are ways to make use of one once it has been established.

For any $\mathcal{F}$ as in (0), I define two infinite series of well-founded functions, ${}^{\flat}f_k$ and ${}^{\sharp}f_k$ where k ranges over the natural numbers:

$${}^{\flat}f_k(x) \;=\; \begin{cases} \text{false} & \text{if } k = 0 \\ \mathcal{F}({}^{\flat}f_{k-1})(x) & \text{if } k > 0 \end{cases} \tag{11}$$

$${}^{\sharp}f_k(x) \;=\; \begin{cases} \text{true} & \text{if } k = 0 \\ \mathcal{F}({}^{\sharp}f_{k-1})(x) & \text{if } k > 0 \end{cases} \tag{12}$$

These functions are called the *iterates* of f, and I will also refer to them as the *prefix predicates* of f (or the *prefix predicate* of f, if we think of k as being a parameter). Alternatively, we can define ${}^{\flat}f_k$ and ${}^{\sharp}f_k$ without mentioning x: Let $\bot$ denote the function that always returns false, let $\top$ denote the function

that always returns true, and let a superscript on $\mathcal{F}$ denote exponentiation (for example, $\mathcal{F}^0(f) = f$ and $\mathcal{F}^2(f) = \mathcal{F}(\mathcal{F}(f))$). Then, (11) and (12) can be stated equivalently as ${}^\flat f_k = \mathcal{F}^k(\bot)$ and ${}^\sharp f_k = \mathcal{F}^k(\top)$.

For any solution f to equation (0), we have, for any k and ℓ such that $k \le \ell$:

$$ {}^\flat f_k \quad \Rightarrow \quad {}^\flat f_\ell \quad \Rightarrow \quad f \quad \Rightarrow \quad {}^\sharp f_\ell \quad \Rightarrow \quad {}^\sharp f_k \tag{13} $$

In other words, every ${}^\flat f_k$ is a *pre-fixpoint* of f and every ${}^\sharp f_k$ is a *post-fixpoint* of f. Next, I define two functions, $f^\downarrow$ and $f^\uparrow$, in terms of the prefix predicates:

$$ f^\downarrow(x) \quad = \quad \exists k \bullet {}^\flat f_k(x) \tag{14} $$

$$ f^\uparrow(x) \quad = \quad \forall k \bullet {}^\sharp f_k(x) \tag{15} $$

By (13), we also have that $f^\downarrow$ is a pre-fixpoint of $\mathcal{F}$ and $f^\uparrow$ is a post-fixpoint of $\mathcal{F}$. The marvelous thing is that, if $\mathcal{F}$ is *continuous*, then $f^\downarrow$ and $f^\uparrow$ are the least and greatest fixpoints of $\mathcal{F}$. These equations let us do proofs by induction when dealing with extreme predicates. I will explain in Section 2.2 how to check for continuity.

Let's consider two examples, both involving function g in (7). As it turns out, g's defining functor is continuous, and therefore I will write $g^\downarrow$ and $g^\uparrow$ to denote the least and greatest solutions for g in (7).

1.2.0. Example with Least Solution

The main technique for establishing that $g^\downarrow(x)$ holds for some x, that is, proving something of the form $Q \implies g^\downarrow(x)$, is to construct a proof tree like the one for $g(6)$ in Figure 0. For a proof in this direction, since we're just applying the defining equation, the fact that we're using a least solution for g never plays a role (as long as we limit ourselves to finite derivations).

The technique for going in the other direction, proving something *from* an established $g^\downarrow$ property, that is, showing something of the form $g^\downarrow(x) \implies R$, typically uses induction on the structure of the proof tree. When the antecedent of our proof obligation includes a predicate term $g^\downarrow(x)$, it is sound to imagine that we have been given a proof tree for $g^\downarrow(x)$. Such a proof tree would be a data structure—to be more precise, a term in an *inductive datatype*. For this reason, least solutions like $g^\downarrow$ have been given the name *inductive predicate*.

Let's prove $g^\downarrow(x) \implies 0 \le x \ \wedge \ x$ even. We split our task into two cases, corresponding to which of the two proof rules in (9) was the last one applied to establish $g^\downarrow(x)$. If it was the left-hand rule, then $x = 0$, which makes it easy to establish the conclusion of our proof goal. If it was the right-hand rule, then we unfold the proof tree one level and obtain $g^\downarrow(x-2)$. Since the proof tree for $g^\downarrow(x-2)$ is smaller than where we started, we invoke the *induction hypothesis* and obtain $0 \le (x-2) \ \wedge \ (x-2)$ even, from which it is easy to establish the conclusion of our proof goal.

Here's how we do the proof formally using (14). We massage the general form of our proof goal:

$$
\begin{array}{ll}
 & f^\uparrow(x) \implies R \\
= & \quad \{ \text{ (14) } \} \\
 & (\exists k \bullet {}^\flat f_k(x)) \implies R \\
= & \quad \{ \text{ distribute } \implies \text{ over } \exists \text{ to the left } \} \\
 & \forall k \bullet ({}^\flat f_k(x) \implies R)
\end{array}
$$

The last line can be proved by induction over k. So, in our case, we prove ${}^\flat g_k(x) \implies 0 \le x \ \wedge \ x$ even for every k. If $k = 0$, then ${}^\flat g_k(x)$ is false, so our goal holds trivially. If $k > 0$, then ${}^\flat g_k(x) = (x = 0 \ \vee \ {}^\flat g_{k-1}(x-2))$. Our goal holds easily for the first disjunct ($x = 0$). For the other disjunct, we apply the induction hypothesis (on the smaller $k-1$ and with $x-2$) and obtain $0 \le (x-2) \ \wedge \ (x-2)$ even, from which our proof goal follows.

1.2.1. Example with Greatest Solution

We can think of a given predicate $g^{\uparrow}(x)$ as being represented by a proof tree—in this case a term in a *coinductive datatype*, since the proof may be infinite. For this reason, greatest solutions like $g^{\uparrow}$ have been given the name *coinductive predicate*, or *co-predicate* for short. The main technique for proving something from a given proof tree, that is, to prove something of the form $g^{\uparrow}(x) \implies R$, is to destruct the proof. Since this is just unfolding the defining equation, the fact that we're using a greatest solution for g never plays a role (as long as we limit ourselves to a finite number of unfoldings).

To go in the other direction, to establish a predicate defined as a greatest solution, like $Q \implies g^{\uparrow}(x)$, we may need an infinite number of steps. For this purpose, we can use induction's dual, *coinduction*. Were it not for one little detail, coinduction is as simple as continuations in programming: the next part of the proof obligation is delegated to the *coinduction hypothesis*. The little detail is making sure that it is the "next" part we're passing on for the continuation, not the same part. This detail is called *productivity* and corresponds to the requirement in induction of making sure we're going down a well-founded relation when applying the induction hypothesis. There are many sources with more information, see for example the classic account by Jacobs and Rutten [8] or a new attempt by Kozen and Silva that aims to emphasize the simplicity, not the mystery, of coinduction [10].

Let's prove $\mathrm{true} \implies g^{\uparrow}(x)$. The intuitive coinductive proof goes like this: According to the right-hand rule of (10), $g^{\uparrow}(x)$ follows if we establish $g^{\uparrow}(x-2)$, and that's easy to do by invoking the coinduction hypothesis. The "little detail", productivity, is satisfied in this proof because we applied a rule in (10) before invoking the coinduction hypothesis.

For anyone who may have felt that the intuitive proof felt too easy, here is a formal proof using (15), which relies only on induction. We massage the general form of our proof goal:

$$
\begin{array}{ll}
 & Q \implies f^{\uparrow}(x) \\
= & \quad \{\ (15)\ \} \\
 & Q \implies \forall k \bullet {}^{\sharp}f_k(x) \\
= & \quad \{\ \text{distribute } \implies \text{ over } \forall \text{ to the right}\ \} \\
 & \forall k \bullet Q \implies {}^{\sharp}f_k(x)
\end{array}
$$

The last line can be proved by induction over k. So, in our case, we prove $\mathrm{true} \implies {}^{\sharp}g_k(x)$ for every k. If $k = 0$, then ${}^{\sharp}g_k(x)$ is true, so our goal holds trivially. If $k > 0$, then ${}^{\sharp}g_k(x) = (x = 0 \lor {}^{\sharp}g_{k-1}(x-2))$. We establish the second disjunct by applying the induction hypothesis (on the smaller $k-1$ and with $x-2$).

1.3. Other Techniques

Although in this paper I consider only well-founded functions and extreme predicates, it is worth mentioning that there are additional ways of making sure that the set of solutions to (0) is nonempty. For example, if all calls to f in $\mathcal{F}'(f)$ are *tail-recursive calls*, then (under the assumption that Y is nonempty) the set of solutions is nonempty. To see this, consider an attempted evaluation of $f(x)$ that fails to determine a definite result value because of an infinite chain of calls that applies f to each value of some subset X' of X. Then, apparently, the value of f for any one of the values in X' is not determined by the equation, but picking any particular result values for these makes for a consistent definition. This was pointed out by Manolios and Moore [18]. Functions can be underspecified in this way in the proof assistants ACL2 [9] and HOL [11].

2. Functions in Dafny

In this section, I explain with examples the support in Dafny[0] for well-founded functions, extreme predicates, and proofs regarding these.

2.0. Well-founded Functions in Dafny

Declarations of well-founded functions are unsurprising. For example, the Fibonacci function is declared as follows:

```
function fib(n: nat): nat
{
  if n < 2 then n else fib(n-2) + fib(n-1)
}
```

Dafny verifies that the body (given as an expression in curly braces) is well defined. This includes decrement checks for recursive (and mutually recursive) calls. Dafny predefines a well-founded relation on each type and extends it to lexicographic tuples of any (fixed) length. For example, the well-founded relation $x \ll y$ for integers is $x < y \ \wedge \ 0 \leq y$, the one for reals is $x \leq y - 1.0 \ \wedge \ 0.0 \leq y$ (this is the same ordering as for integers, if you read the integer relation as $x \leq y - 1 \ \wedge \ 0 \leq y$), the one for inductive datatypes is structural inclusion, and the one for coinductive datatypes is false.

Using a `decreases` clause, the programmer can specify the term in this predefined order. When a function definition omits a `decreases` clause, Dafny makes a simple guess. This guess (which can be inspected by hovering over the function name in the Dafny IDE) is very often correct, so users are rarely bothered to provide explicit `decreases` clauses.

If a function returns `bool`, one can drop the result type `: bool` and change the keyword `function` to `predicate`.

2.1. Proofs in Dafny

Dafny has `lemma` declarations. These are really just special cases of methods: they can have pre- and postcondition specifications and their body is a code block. Here is the lemma we stated and proved in Section 1.0.0:

```
lemma FibProperty(n: nat)
  ensures fib(n) % 2 == 0 <==> n % 3 == 0
{
  if n < 2 {
  } else {
    FibProperty(n-2); FibProperty(n-1);
  }
}
```

The postcondition of this lemma (keyword `ensures`) gives the proof goal. As in any program-correctness logic (e.g., [6]), the postcondition must be established on every control path through the lemma's body. For `FibProperty`, I give the proof by an `if` statement, hence introducing a case split. The then branch is empty, because Dafny can prove the postcondition automatically in this case. The else branch performs two recursive calls to the lemma. These are the invocations of the induction hypothesis and they follow the usual program-correctness rules, namely: the precondition must hold at the call site, the call must

[0] Dafny is open source at dafny.codeplex.com and can also be used online at rise4fun.com/dafny.

terminate, and then the caller gets to assume the postcondition upon return. The "proof glue" needed to complete the proof is done automatically by Dafny.

Dafny features an aggregate statement using which it is possible to make (possibly infinitely) many calls at once. For example, the induction hypothesis can be called at once on all values `n'` smaller than `n`:

```
forall n' | 0 <= n' < n {
  FibProperty(n');
}
```

For our purposes, this corresponds to *strong induction*. More generally, the `forall` statement has the form

```
forall k | P(k)
  ensures Q(k)
{ Statements; }
```

Logically, this statement corresponds to *universal introduction*: the body proves that `Q(k)` holds for an arbitrary `k` such that `P(k)`, and the conclusion of the `forall` statement is then $\forall k \bullet P(k) \implies Q(k)$. When the body of the `forall` statement is a single call (or `calc` statement), the `ensures` clause is inferred and can be omitted, like in our `FibProperty` example.

Lemma `FibProperty` is simple enough that its whole body can be replaced by the one `forall` statement above. In fact, Dafny goes one step further: it automatically inserts such a `forall` statement at the beginning of every lemma [13]. Thus, `FibProperty` can be declared and proved simply by:

```
lemma FibProperty(n: nat)
  ensures fib(n) % 2 == 0 <==> n % 3 == 0
{ }
```

Going in the other direction from universal introduction is existential elimination, also known as Skolemization. Dafny has a statement for this, too: for any variable `x` and boolean expression `Q`, the *assign such that* statement `x :| Q;` says to assign to `x` a value such that `Q` will hold. A proof obligation when using this statement is to show that there exists an `x` such that `Q` holds. For example, if the fact $\exists k \bullet 100 \leq \mathrm{fib}(k) < 200$ is known, then the statement `k :| 100 <= fib(k) < 200;` will assign to `k` some value (chosen arbitrarily) for which `fib(k)` falls in the given range.

2.2. Extreme Predicates in Dafny

In this previous subsection, I explained that a `predicate` declaration introduces a well-founded predicate. The declarations for introducing extreme predicates are `inductive predicate` and `copredicate`. Here is the definition of the least and greatest solutions of g from above, let's call them `g` and `G`:

```
inductive predicate g(x: int) { x == 0 || g(x-2) }
copredicate G(x: int) { x == 0 || G(x-2) }
```

When Dafny receives either of these definitions, it automatically declares the corresponding prefix predicates. Instead of the names ${}^{\flat}g_k$ and ${}^{\sharp}g_k$ that I used above, Dafny names the prefix predicates `g#[k]` and `G#[k]`, respectively, that is, the name of the extreme predicate appended with `#`, and the subscript is given as an argument in square brackets. The definition of the prefix predicate derives from the body of the extreme predicate and follows the form in (11) and (12). Using a faux-syntax for illustrative purposes, here are the prefix predicates that Dafny defines automatically from the extreme predicates `g` and `G`:

```
predicate g#[_k: nat](x: int) { _k != 0 && (x == 0 || g#[_k-1](x-2)) }
predicate G#[_k: nat](x: int) { _k != 0 ==> (x == 0 || G#[_k-1](x-2)) }
```

The Dafny verifier is aware of the connection between extreme predicates and their prefix predicates, (14) and (15).

Remember that to be well defined, the defining functor of an extreme predicate must be monotonic, and for (14) and (15) to hold, the functor must be continuous. Dafny enforces the former of these by checking that recursive calls of extreme predicates are in positive positions. The continuity requirement comes down to checking that they are also in *continuous positions*: that recursive calls to inductive predicates are not inside unbounded universal quantifiers and that recursive calls to co-predicates are not inside unbounded existential quantifiers [15, 19].

2.3. Proofs about Extreme Predicates

From what I have presented so far, we can do the formal proofs from Sections 1.2.0 and 1.2.1. Here is the former:

```
lemma EvenNat(x: int)
  requires g(x)
  ensures 0 <= x && x % 2 == 0
{
  var k: nat :| g#[k](x);
  EvenNatAux(k, x);
}
lemma EvenNatAux(k: nat, x: int)
  requires g#[k](x)
  ensures 0 <= x && x % 2 == 0
{
  if x == 0 { } else { EvenNatAux(k-1, x-2); }
}
```

Lemma `EvenNat` states the property we wish to prove. From its precondition (keyword `requires`) and (14), we know there is some `k` that will make the condition in the assign-such-that statement true. Such a value is then assigned to `k` and passed to the auxiliary lemma, which promises to establish the proof goal. Given the condition `g#[k](x)`, the definition of `g#` lets us conclude `k != 0` as well as the disjunction `x == 0 || g#[k-1](x-2)`. The then branch considers the case of the first disjunct, from which the proof goal follows automatically. The else branch can then assume `g#[k-1](x-2)` and calls the induction hypothesis with those parameters. The proof glue that shows the proof goal for `x` to follow from the proof goal with `x-2` is done automatically.

Because Dafny automatically inserts the statement

```
forall k', x' | 0 <= k' < k && g#[k'](x') {
  EvenNatAux(k', x');
}
```

at the beginning of the body of `EvenNatAux`, the body can be left empty and Dafny completes the proof automatically.

Here is the Dafny program that gives the proof from Section 1.2.1:

```
lemma Always(x: int)
  ensures G(x)
{ forall k: nat { AlwaysAux(k, x); } }
lemma AlwaysAux(k: nat, x: int)
  ensures G#[k](x)
```

```
{ }
```

While each of these proofs involves only basic proof rules, the setup feels a bit clumsy, even with the empty body of the auxiliary lemmas. Moreover, the proofs do not reflect the intuitive proofs I described in Section 1.2.0 and 1.2.1. These shortcoming are addressed in the next subsection.

2.4. Nicer Proofs of Extreme Predicates

The proofs we just saw follow standard forms: use Skolemization to convert the inductive predicate into a prefix predicate for some `k` and then do the proof inductively over `k`; respectively, by induction over `k`, prove the prefix predicate for every `k`, then use universal introduction to convert to the coinductive predicate. With the declarations `inductive lemma` and `colemma`, Dafny offers to set up the proofs in these standard forms. What is gained is not just fewer characters in the program text, but also a possible intuitive reading of the proofs. (Okay, to be fair, the reading is intuitive for simpler proofs; complicated proofs may or may not be intuitive.)

Somewhat analogous to the creation of prefix predicates from extreme predicates, Dafny automatically creates a *prefix lemma* `L#` from each "extreme lemma" `L`. The pre- and postconditions of a prefix lemma are copied from those of the extreme lemma, except for the following replacements: For an inductive lemma, Dafny looks in the precondition to find calls (in positive, continuous positions) to inductive predicates `P(x)` and replaces these with `P#[_k](x)`. For a co-lemma, Dafny looks in the postcondition to find calls (in positive, continuous positions) to co-predicates `P` (including equality among coinductive datatypes, which is a built-in co-predicate) and replaces these with `P#[_k](x)`. In each case, these predicates `P` are the lemma's *focal predicates*.

The body of the extreme lemma is moved to the prefix lemma, but with replacing each recursive call `L(x)` with `L#[_k-1](x)` and replacing each occurrence of a call to a focal predicate `P(x)` with `P#[_k-1](x)`. The bodies of the extreme lemmas are then replaced as shown in the previous subsection. By construction, this new body correctly leads to the extreme lemma's postcondition.

Let us see what effect these rewrites have on how one can write proofs. Here are the proofs of our running example:

```
inductive lemma EvenNat(x: int)
  requires g(x)
  ensures 0 <= x && x % 2 == 0
{ if x == 0 { } else { EvenNat(x-2); } }
colemma Always(x: int)
  ensures G(x)
{ Always(x-2); }
```

Both of these proofs follow the intuitive proofs given in Sections 1.2.0 and 1.2.1. Note that in these simple examples, the user is never bothered with either prefix predicates nor prefix lemmas—the proofs just look like "what you'd expect".

Since Dafny automatically inserts calls to the induction hypothesis at the beginning of each lemma, the bodies of the given extreme lemmas `EvenNat` and `Always` can be empty and Dafny still completes the proofs. Folks, it doesn't get any simpler than that!

3. Case Study: Modeling Semantics

Computer scientists in the programming language area like to model the semantics of languages in order to reason about their behavior. This activity is often aided by the use of inductive predicates [3, 20, 24,

27], and sometimes coinductive predicates [17]. Let me illustrate with a small excerpt from a semantics proof how Dafny's features can be used.

Chapter 7 of Nipkow and Klein's book *Concrete Semantics* [20] defines the big-step and small-step semantics for the rudimentary imperative language IMP [27]. The commands (statements) of the language are defined using an inductive datatype:

```
datatype com = SKIP | Assign(vname, aexp) | Seq(com, com)
             | If(bexp, com, com) | While(bexp, com)
```

and the big-step semantics is defined using an inductive predicate, of which I show the case for sequential composition (`Seq`) here:

```
inductive predicate big_step(c: com, s: state, t: state)
{ match c ...
  case Seq(c0, c1) =>
    ∃ s' :: big_step(c0, s, s') && big_step(c1, s', t)
}
```

This case corresponds to what in inference rules would be rendered as follows:

$$\frac{big\_step(c0, s, s') \qquad big\_step(c1, s', t)}{big\_step(Seq(c0, c1), s, t)} \tag{16}$$

Note how the s' above the line becomes existentially quantified in the definition of the inductive predicate in Dafny.

The small-step semantics of IMP is defined using two inductive predicates, one for a single step (omitted here) and one for the reflexive transitive closure thereof:

```
inductive predicate small_step_star(c: com, s: state, c': com, s': state)
{
  (c == c' && s == s') ||
  ∃ c", s" :: small_step(c, s, c", s") && small_step_star(c", s", c', s')
}
```

A usual theorem of interest is to prove a correspondence between the big-step and small-step semantics. Here, I show the statement of that theorem along with the proof case for `Seq`:

```
inductive lemma BigStep_implies_SmallStepStar(c: com, s: state, t: state)
  requires big_step(c, s, t)
  ensures small_step_star(c, s, SKIP, t)
{ match c ...
  case Seq(c0, c1) =>
    var s' :| big_step(c0, s, s') && big_step(c1, s', t);
    calc {
      true;
    ==>  // induction hypothesis
      small_step_star(c1, s', SKIP, t);
    ==>  // small-step semantics with SKIP as first argument to Seq
      small_step_star(Seq(SKIP, c1), s', SKIP, t);
    ==>  // induction hypothesis
      small_step_star(c0, s, SKIP, s') && small_step_star(Seq(SKIP, c1), s', SKIP, t);
    ==>  { lemma_7_13(c0, s, SKIP, s', c1); }
      small_step_star(Seq(c0, c1), s, Seq(SKIP, c1), s') &&
```

```
      small_step_star(Seq(SKIP, c1), s', SKIP, t);
    ==>  { star_transitive(Seq(c0, c1), s, Seq(SKIP, c1), s', SKIP, t); }
      small_step_star(c, s, SKIP, t);
    }
}
```

This proof case first Skolemizes the `s'` from the corresponding definition of `big_step` and then embarks on a proof calculation [16] that shows successive implications from `true` to the proof goal. The proof looks natural and never mentions any prefix predicate explicitly. Under the hood, the lemma's precondition is really `big_step#[_k](c, s, t)` and the right-hand side of the Skolemization is really `big_step#[_k-1](c0, s, s') && big_step#[_k-1](c1, s', t)`. The first and third steps of the calculation hold on behalf of these prefix predicates and the (automatically applied) induction hypothesis. Overall, these shorthands contribute to a short and readable proof.

One more example will illustrate a final point. Here is the statement and proof of "Lemma 7.13" that was used above:

```
inductive lemma lemma_7_13(c0: com, s0: state, c: com, t: state, c1: com)
  requires small_step_star(c0, s0, c, t)
  ensures small_step_star(Seq(c0, c1), s0, Seq(c, c1), t)
{
  if c0 == c && s0 == t {
  } else {
    var c', s' :| small_step(c0, s0, c', s') && small_step_star(c', s', c, t);
    lemma_7_13(c', s', c, t, c1);
  }
}
```

The else branch of this lemma, like the `Seq` case in the proof above, uses Skolemization to give names (`c'` and `s'`) to what is known to hold at this point. This is typical when the definition of the inductive predicate uses an existential quantifier. But what if the definition has further disjuncts with existential quantifiers? Then the `if` statement in the lemma must check for these, which I can illustrate with the same example by just reversing the order of the then and else branches:

```
if ∃c', s' :: small_step(c0, s0, c', s') && small_step_star(c', s', c, t) {
  var c', s' :| small_step(c0, s0, c', s') && small_step_star(c', s', c, t);
  lemma_7_13(c', s', c, t, c1);
}
```

(Here, I omitted the else branch in the usual way, since it is empty anyway.) Having to repeat the condition both in the `if` guard and the subsequent Skolemization is clumsy. Therefore, Dafny features `if` statements with binding guards, which allow the body of `lemma_7_13` to be simply:

```
if c', s' :| small_step(c0, s0, c', s') && small_step_star(c', s', c, t) {
  lemma_7_13(c', s', c, t, c1);
}
```

In short, the `if` alternative is taken if there exist values for `c'` and `s'` that make the condition hold, and then `c'` and `s'` remain bound in the then branch. Dafny also includes a symmetric `if` statement, like the **if** ... **fi** statement in Dijkstra's guarded command language [4]. It also supports the binding guards, as can be seen here:

```
if {
  case c0 == c && s0 == t =>
```

```
  case c', s' :| small_step(c0, s0, c', s') && small_step_star(c', s', c, t) =>
    lemma_7_13(c', s', c, t, c1);
}
```

This concludes my tutorial examples. More examples of coinductive definitions and proofs are found in previous papers [14, 15]. The full Dafny encoding of Nipkow and Klein's chapter 7 is found in the test suite of the Dafny open-source distribution, dafny.codeplex.com. Never in this encoding (other than in an example) are the prefix predicates or prefix lemmas mentioned explicitly, which gives support to the idea that the syntactic rewrites hit the spot. A user can inspect the rewrites by hovering over the calls to the extreme lemmas and predicates in the Dafny IDE.

4. Other Tools

Other tools, like Coq [22], Isabelle [21], HOL [5], Agda [1], VeriFast [7], and F* [25], have since long supported inductive predicates, typically via dependent types. In these languages, the notation for defining inductive predicates is inverted compared to Dafny, using a *clausal form* rather than a *casewise form* [5]. For example, here is the big-step definition in Coq, showing the case for `Seq`:

```
Inductive big_step : com -> state -> state -> Prop := ...
  | BS_Seq : forall c0 c1 s s' t,
      big_step c0 s s' -> big_step c1 s' t -> big_step (Seq c0 c1) s t
```

Sometimes, this direction of the definition is more intuitive. It would be nice to support in Dafny an alternative syntax for writing definitions this way.

In this paper, I have presented extreme predicates as being defined as extreme fixpoints of a functor $\mathcal{F}$. A well-founded predicate is also a fixpoint of the defining functor, but talking about it as a least or greatest fixpoint is not interesting, since the fixpoint of the defining functor is unique. From this perspective, it is curious that the keyword used in Coq to define a well-founded function is `Fixpoint`.

Another difference is that whereas tools like Coq and F* do the induction over the actual proof tree, Dafny's induction is essentially over the *height* of the proof tree (an upper bound of which is given by `_k`). With the syntactic rewriting shown in this paper, the Dafny proofs can read as if they were over the proof trees, rather than having to talk about the height explicitly.

When it comes to defining co-predicates, the continuity restriction is awkward. It means that the common existential quantifiers like in the inductive definition of `Seq` above cannot be used directly. The workaround is to move the existential quantifier outside the entire co-predicate, see [15]. It would be wonderful to have a different solution for this in Dafny.

The fact that the Dafny verifier uses an SMT solver provides useful automation for a lot of proof glue. Dafny's encoding of extreme predicates and its automatic insertion of the induction hypothesis extend this automation to more advanced proof steps. The verifying type checker for F* [25] also uses an SMT solver, but does not include the more advanced automation. The Why3 language provides a syntax for inductive predicates and its verifier backend supports several SMT solvers. However, the inductive predicates defined are treated as any arbitrary solution to (0), so the SMT solvers do not reason about least or greatest fixpoints [3].

In this paper, I've talked about Dafny as a tool to state and prove lemmas. More generally, Dafny is a programming language and the constructs I have shown for proofs are shared with the compiled fragment of the language. In particular, forms of the `forall` statement, the assign-such-that statement, and the binding `if` guards are also available for writing programs that compile and run.

5. Conclusions

In this paper, I have conveyed a way to understand well-founded functions and extreme predicates and have given a number of small but representative examples of their use. The Dafny language previously had well-founded functions, induction, co-predicates, and co-lemmas. New in this paper are the inductive predicates and inductive lemmas, which are simply the duals of the coinductive counterparts. Having both makes for a more balanced understanding of how these are used, and I tried in my presentation not to make the coinductive constructs seem any more mysterious than the inductive counterparts. Because the inductive constructs are used more often in practice, the additional experience with them has led to the further improvements presented in this paper, namely rewriting of focal predicates in extreme lemmas and the binding `if` guards. I hope that the automation facilitated by Dafny, as well as this tutorial itself, will give students and researchers less painful access to mechanized support around formalizations and proofs.

Acknowledgments Some of the crystalized thinking that went into the presentation in this paper came while preparing for a mini-course on Formal Semantics that I taught at Imperial College London in May 2015. I've always been jealous of the nice inductive predicates in Coq, sometimes philosophically pondering the question *Are they datatypes or functions?*, and I am glad to finally support a form of them (as functions) in Dafny. I thank the referees and Jonathan Protzenko for helpful comments on drafts of this paper and am grateful to Daan Leijen for assistance with type setting.

References

[0] Y. Bertot and P. Castéran. *Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Comp. Sci. Springer, 2004.

[1] A. Bove, P. Dybjer, and U. Norell. A brief overview of Agda — a functional language with dependent types. In *TPHOLs 2009*, volume 5674 of *LNCS*, pages 73–78. Springer, Aug. 2009.

[2] J. Camilleri and T. Melham. Reasoning with inductively defined relations in the HOL theorem prover. Technical Report 265, University of Cambridge Computer Laboratory, 1992.

[3] M. Clochard, J.-C. Filliâtre, C. Marché, and A. Paskevich. Formalizing semantics with an automatic program verifier. In *VSTTE 2014*, volume 8471 of *LNCS*, pages 37–51. Springer, July 2014.

[4] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, NJ, 1976.

[5] J. Harrison. Inductive definitions: Automation and application. In E. T. Schubert, P. J. Windley, and J. Alves-Foss, editors, *TPHOLs 1995*, volume 971 of *LNCS*, pages 200–213. Springer, 1995.

[6] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580,583, Oct. 1969.

[7] B. Jacobs and F. Piessens. The VeriFast program verifier. Technical Report CW-520, Dept. of Computer Science, Katholieke Universiteit Leuven, 2008.

[8] B. Jacobs and J. Rutten. An introduction to (co)algebra and (co)induction. In *Advanced Topics in Bisimulation and Coinduction*, number 52 in Cambridge Tracts in Theoretical Comp. Sci., pages 38–99. Cambridge Univ. Press, 2011.

[9] M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.

[10] D. Kozen and A. Silva. Practical coinduction. Technical Report http://hdl.handle.net/1813/30510, Comp. and Inf. Science, Cornell Univ., 2012.

[11] A. Krauss. *Automating Recursive Definitions and Termination Proofs in Higher-Order Logic*. PhD thesis, Technische Universität München, 2009.

[12] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR-16*, volume 6355 of *LNCS*, pages 348–370. Springer, 2010.

[13] K. R. M. Leino. Automating induction with an SMT solver. In *VMCAI 2012*, volume 7148 of *LNCS*, pages 315–331. Springer, Jan. 2012.

[14] K. R. M. Leino. Automating theorem proving with SMT. In *ITP 2013*, volume 7998 of *LNCS*, pages 2–16. Springer, July 2013.

[15] K. R. M. Leino and M. Moskal. Co-induction simply — automatic co-inductive proofs in a program verifier. In *FM 2014*, volume 8442 of *LNCS*, pages 382–398. Springer, May 2014.

[16] K. R. M. Leino and N. Polikarpova. Verified calculations. In *VSTTE 2013*, volume 8164 of *LNCS*, pages 170–190. Springer, 2014.

[17] X. Leroy and H. Grall. Coinductive big-step operational semantics. *Information and Computation*, 207(2):284–304, Feb. 2009.

[18] P. Manolios and J. S. Moore. Partial functions in ACL2. *Journal of Automated Reasoning*, 31(2):107–127, 2003.

[19] R. Milner. *A Calculus of Communicating Systems*. Springer, 1982.

[20] T. Nipkow and G. Klein. *Concrete Semantics with Isabelle/HOL*. Springer, 2014.

[21] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[22] C. Paulin-Mohring. Inductive definitions in the system Coq — rules and properties. In *TLCA '93*, volume 664 of *LNCS*, pages 328–345. Springer, 1993.

[23] L. C. Paulson. A fixedpoint approach to implementing (co)inductive definitions. In A. Bundy, editor, *CADE-12*, volume 814 of *LNCS*, pages 148–161. Springer, 1994.

[24] B. C. Pierce, C. Casinghino, M. Gaboardi, M. Greenberg, C. Hriţcu, V. Sjöberg, and B. Yorgey. *Software Foundations*. http://www.cis.upenn.edu/~bcpierce/sf, version 3.2 edition, Jan. 2015.

[25] N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan, and J. Yang. Secure distributed programming with value-dependent types. In *ICFP 2011*, pages 266–278. ACM, Sept. 2011.

[26] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[27] G. Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, 1993.

Improving Statistical Linguistic Algorithms for Parsing Mathematics

Cezary Kaliszyk
University of Innsbruck
cezary.kaliszyk@uibk.ac.at

Josef Urban
Czech Technical University in Prague
josef.urban@gmail.com

Jiří Vyskočil
Czech Technical University in Prague
jiri.vyskocil@gmail.com

Abstract

In this paper we describe our combined statistical/semantic parsing method based on the CYK chart-parsing algorithm augmented with limited internal typechecking and external ATP filtering. This method was previously evaluated on parsing ambiguous mathematical expressions over the informalized Flyspeck corpus of 20000 theorems. We first discuss the motivation and drawbacks of the first version of the CYK-based component of the algorithm, and then we propose and implement a more sophisticated approach based on better statistical model of mathematical data structures.

1 Introduction

Computer-understandable (formal) mathematics is today still quite far from taking over the mathematical mainstream. Despite the impressive formalizations such as Flyspeck [5], Feit-Thompson [4], seL4 [11], CompCert [13], and CCL [2], and the progress in general automation over such large formal corpora, formalizing proofs is still largely unappealing to mathematicians. While the research on AI and strong automation over large theories has taken off in the last decade and automation improvements are today coming from several directions, there has been so far very little progress in automating the understanding of informal LaTeX-written and ambiguous mathematical writings.

Recently, we have proposed to try to change this state of affairs by learning how to parse informal mathematics from aligned informal/formal corpora [10]. Such learning can be additionally combined with strong semantic filtering methods such as typechecking and large-theory ATP. Suitable aligned corpora are appearing today, the major example being Flyspeck and in particular its alignment (by Hales) with the detailed informal Blueprint for Formal Proofs [5]. Very recently [8] we have implemented the first version of a statistical/semantic parsing toolchain that learns parsing rules from many pairs of ambiguous/nonambiguous Flyspeck formulas, and combines statistical parsing of new ambiguous formulas with internal semantic pruning and external proving/disproving step. The resulting parsing/proving system trained on all of Flyspeck is available online[1] and can be used for experimenting with parsing ambiguous statements.

In this short paper we explain in more detail the particular statistical learning/parsing approach based on the CYK chart parsing algorithm [14] for probabilistic context-free grammars (PCFG) that we have been using (Sec. 2.1), and focus on some drawbacks of the context-free approach that can negatively influence the statistical learning and parsing performance (Sec. 2.2). We demonstrate this on a simple example, where the PCFG setting is not strong enough to eventually learn the correct parsing (Sec. 3.1). Then we propose and implement a modification of CYK that takes into account larger parsing subtrees and their probabilities (Sec. 3.2). This modification is motivated by an analogy with large-theory reasoning systems.

[1] http://colo12-c703.uibk.ac.at/hh/parse.html

There, the precision of the probabilistic selection of the right premises for a new conjecture can typically be significantly improved by considering the large number of term and formula subtrees of the data, rather than just characterizing formulas and their similarity by the bare symbols appearing in them. Finally, we report the first measurements done with the new implementation and discuss future work (Sec. 3.3).

2 Training Statistical Parsing on Aligned Corpora

2.1 PCFG

Given a large corpus of corresponding informal/formal (ambiguous/nonambiguous or LaTeX/HOL) formulas, how do we automatically train an AI system that will correctly parse the next informal formula into a formal one?

Our domain differs from the natural-language domains, where millions of examples of paired (e.g., English/German) sentences are available for training machine translation, the languages have many more words (concepts) than in mathematics, and the sentences to a large extent also lack the recursive structure that is frequently encountered in mathematics. Given that we currently have only thousands of the informal/formal examples, we have decided against using purely statistical alignment methods based on n-grams, and rather investigated methods that can learn how to compose larger parse trees from smaller ones based on those encountered in the limited number of examples that we have.

One well-known approach ensuring this kind of compositionality is the use of CFG (Context Free Grammar) parsers. This approach has been widely used, e.g., for word-sense disambiguation in natural languages, which is another linguistic area close to our informal/formal task. An advantage of this approach is the existence of a parsing algorithm that works in polynomial complexity wrt. the size of the parsed sentence and the input grammar. A well-known and frequently used example is the CYK (Cocke–Younger–Kasami) chart-parsing algorithm [14], using bottom-up parsing and dynamic programming. By default CYK requires the CFG to be in the Chomsky Normal Form (CNF), and the transformation to CNF can cause an exponential blow-up of the grammar. However, an improved version of CYK can be used that gets around this issue [12].

A CFG-based parser obviously needs for its work the *input grammar* – the set of all grammar rules that can be used for parsing. In linguistic applications such grammar is typically extracted from *grammar trees* which correspond to the correct parses of natural-language sentences. Great efforts have been made in the linguistic community to create large *treebanks* of such correct parses – typically taking years of manual annotation work. The grammar rules extracted from the treebanks are typically ambiguous: there are multiple possible parse trees for a particular sentence. This is why CFG is extended by adding a probability to each grammar rule, resulting in PCFG (Probabilistic CFG). During the PCFG parsing of an ambiguous sentence, each resulting parse tree is assigned its probability, which allare most probable wrt. the treebank of training examples. The grammar ows to focus on the few parses that rule probabilities can be trained e.g. by the inside-outside algorithm [1].

2.2 Using PCFG for learning informal/formal alignment

We have so far experimented with several ways how to set up the parsing grammar and its learning. The most low-level approach consists of using the simplest HOL Light lambda calculus internal term structure [6], where terms and types are annotated with only a few nonterminals

such as: `Comb` (application), `Abs` (abstraction), `Const` (higher-order constant), `Var` (variable), `Tyapp` (type application), and `Tyvar` (type variable). This has led to many possible parses in the context-free setting, because the top-level learned rules become very universal, e.g:

```
Comb -> Const Var.
Comb -> Const Const.
Comb -> Comb Comb.
```

So far, the type information does not help to constrain the applications, and the last rule allows a series of several constants to be given arbitrary application order, leading to uncontrolled explosion.

That is why we have first re-ordered and simplified the HOL Light parse trees to propagate the type information at appropriate places where the context-free rules have a chance of providing meaningful pruning information. For example, the raw HOL Light parse tree for theorem

```
REAL_NEGNEG: !x.  --(--x) = x
```

is as follows (see also the tree in Fig. 1):

```
(Comb (Const "!" (Tyapp "fun" (Tyapp "fun" (Tyapp "real") (Tyapp "bool")) (Tyapp "bool")))
(Abs "A0" (Tyapp "real") (Comb (Comb (Const "=" (Tyapp "fun" (Tyapp "real") (Tyapp "fun"
(Tyapp "real") (Tyapp "bool")))) (Comb (Const "real_neg" (Tyapp "fun" (Tyapp "real") (Tyapp
"real"))) (Comb (Const "real_neg" (Tyapp "fun" (Tyapp "real") (Tyapp "real"))) (Var "A0"
(Tyapp "real"))))) (Var "A0" (Tyapp "real")))))
```

Note that the CFG rules for this tree are often very general: e.g., the top-level node produces the rule `Comb -> Const Abs`, etc. After our re-ordering and simplification the parse tree used for grammar generation becomes (see also Fig. 2):

```
("(Type bool)" !  ("(Type (fun real bool))" (Abs ("(Type real)" (Var A0)) ("(Type bool)"
("(Type real)" real_neg ("(Type real)" real_neg ("(Type real)" (Var A0)))) = ("(Type real)"
(Var A0))))))
```

The CFG rules extracted from this transformed tree become quite a bit more meaningful, e.g., the two rules:

```
"(Type bool)" -> "(Type real)" = "(Type real)".
"(Type real)" -> real_neg "(Type real)".
```

say that equality of two reals has type `bool`, and negation applied to reals yields reals. Such "typing" rules restrict the number of possible parses much more than the general "application" rules extracted from the original HOL Light tree, while still having a non-trivial generalization (learning) effect that is needed for the compositional behavior of the information extracted from the trees. For example, once we learn that the variable "u" is mostly parsed as a real number, we will be able to apply `real_neg` to "u" even if the particular subterm `‘‘-- u’’` has never yet been seen in the training examples, and the probability of this parse will be relatively high. In other words, having the HOL types as "semantic categories" (corresponding e.g. to the word senses when using PCFG for word-sense disambiguation) seems to be quite a reasonable first choice for the experiments, even though one could probably come up with even more appealing semantic categories based on more involved statistical and semantic analysis of the data.

We should also note that ambiguous notation, such as `‘‘--’’`, is wrapped in the training trees in its disambiguated "semantic" nonterminal – in this case `$#real_neg`. While the type

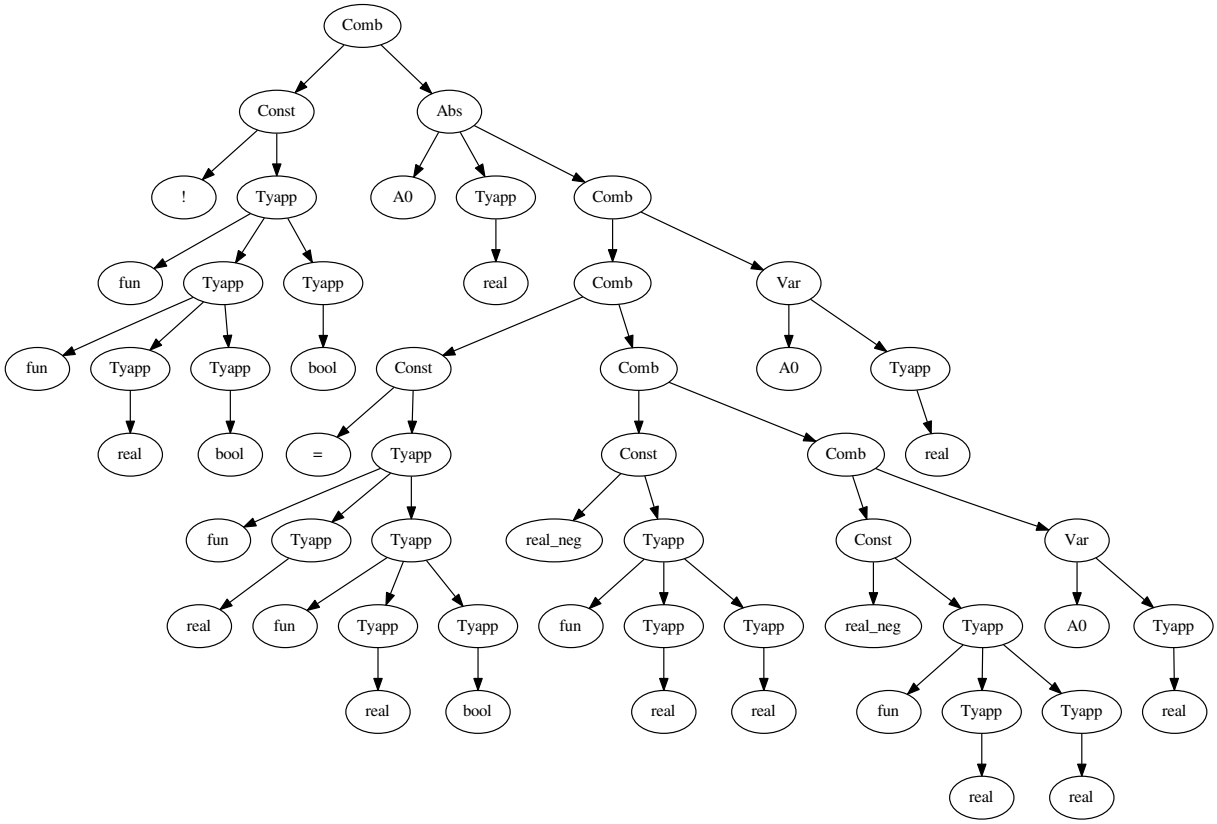

Figure 1: HOL Light parse tree

annotation might often be sufficient for disambiguation, such explicit disambiguation nonterminal is both more precise and allows easier extraction of the HOL semantics from the constructed parse trees. The actual tree used for training the grammar is thus as follows (see also Fig. 3):

```
("(Type bool)" !  ("(Type (fun real bool))" (Abs ("(Type real)" (Var A0)) ("(Type bool)"
("(Type real)" ($#real_neg --) ("(Type real)" ($#real_neg --) ("(Type real)" (Var A0))))
($#= =) ("(Type real)" (Var A0))))))
```

Once the PCFG is learned from such data, we use the CYK algorithm with additional internal lightweight semantic checks to parse ambiguous formulas. These semantic checks are performed to require compatibility of the types of free variables in parsed subtrees. The most probable parse trees are then given to HOL Light and typechecked there, which is followed by proof and disproof attempts by the HOL(y)Hammer system [7], using all the semantic knowledge available in the Flyspeck library (about 22k theorems). See Fig. 4 for the overall structure of the system. The first large-scale disambiguation experiment conducted over "ambiguated" Flyspeck in [8] showed that about 40% of the ambiguous sentences have their correct parses among the best 20 parse trees produced by the trained parser. This is encouraging, but certainly invites further research in improving the statistical/semantic parsing methods.

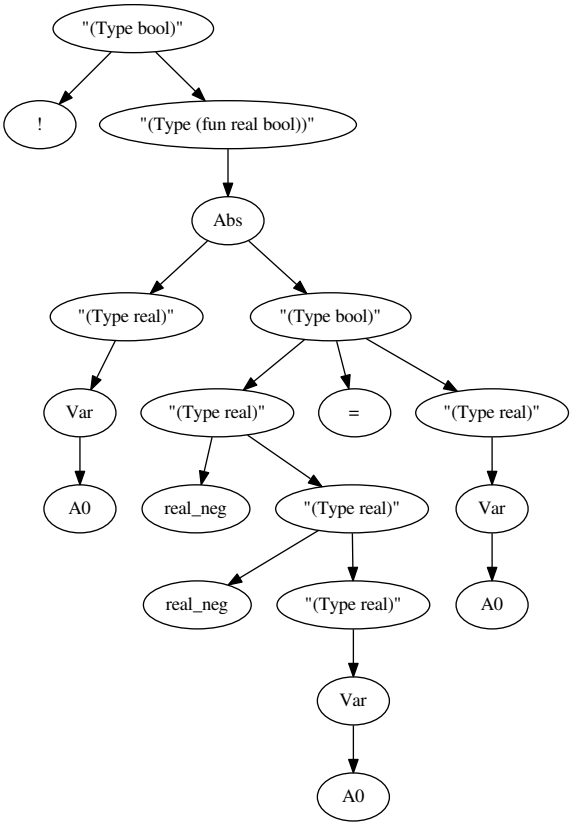

Figure 2: Transformed tree of `REAL_NEGNEG`

3 Adding Context

A major limiting issue when using PCFG-based parsing algorithms is the context-freeness of the grammar. In some cases, no matter how good are the training data, there is no way how to set up the parsing rules probabilities so that the required parse will have the largest probability.

3.1 Example

Consider the following term:

```
1 * x + 2 * x.
```

with the following simplified grammar tree (Fig. 5) as our training data (treebank):

```
(S (Num (Num (Num 1) * (Num x)) + (Num (Num 2) * (Num x))) .)
```

From the grammar tree we extract the following CFG:

```
S -> Num .
Num -> Num + Num
Num -> Num * Num
Num -> 1
Num -> 2
Num -> x
```

If we use this grammar for parsing the original (non-bracketed) sentence, we obtain the following five possible parse trees:

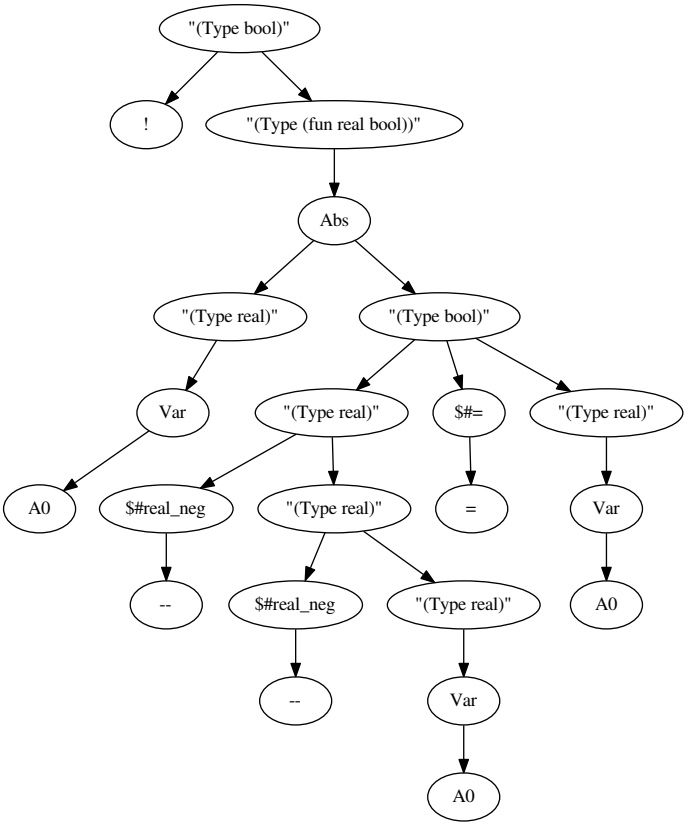

Figure 3: The tree of `REAL_NEGNEG` used for actual grammar training

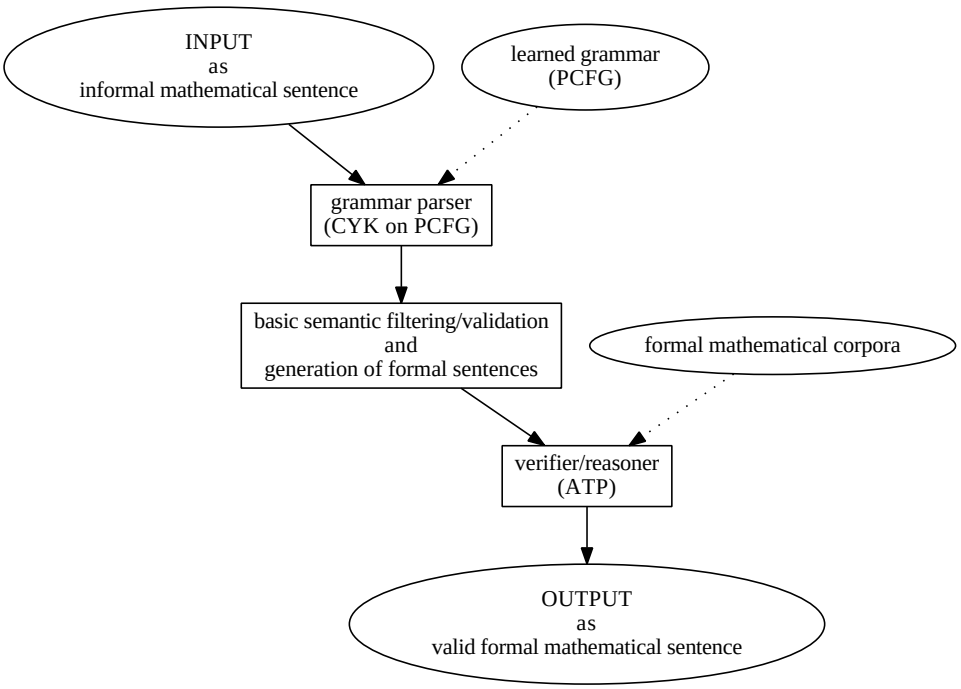

Figure 4: The statistical/semantic parsing toolchain.

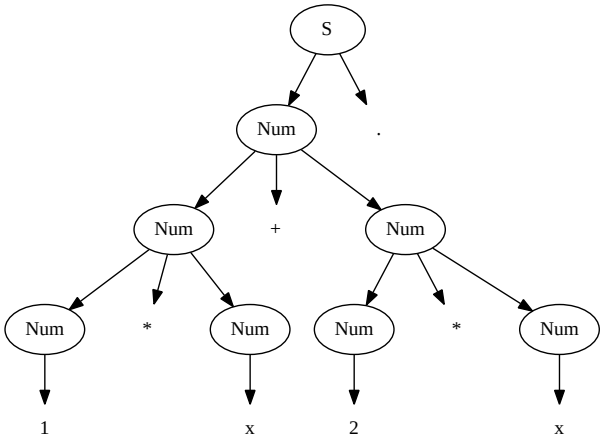

Figure 5: Example grammar tree

```
(S (Num (Num 1) * (Num (Num (Num x) + (Num 2)) * (Num x))) .)
(S (Num (Num 1) * (Num (Num x) + (Num (Num 2) * (Num x)))) .)
(S (Num (Num (Num 1) * (Num (Num x) + (Num 2))) * (Num x)) .)
(S (Num (Num (Num (Num 1) * (Num x)) + (Num 2)) * (Num x)) .)
(S (Num (Num (Num 1) * (Num x)) + (Num (Num 2) * (Num x))) .)
```

Only the last tree however corresponds to the training tree. The problem is that no matter what probabilities we add to the grammar rules, we cannot make the priority of `+` smaller than the priority of `*`: a context-free grammar forgets the context and cannot remember and apply complex mechanisms such as priorities. The probability of all parsed trees is in this case always the same:

$$p(\texttt{S -> Num .}) \times p(\texttt{Num -> Num + Num}) \times p(\texttt{Num -> Num * Num}) \times p(\texttt{Num -> Num * Num}) \times$$
$$p(\texttt{Num -> 1}) \times p(\texttt{Num -> 2}) \times p(\texttt{Num -> x}) \times p(\texttt{Num -> x})$$

While the example does not strictly imply the priorities as we know them, it is clear that we would like the grammar to prefer parse trees that are in some sense *more similar* to the training data. One method that is frequently used for dealing with similar problems in the NLP domain is *grammar lexicalization* [3] where additional terminal can be appended to nonterminals and propagated from the subtrees, thus creating many more possible (more precise) nonterminals. This approach however does not solve the particular problem with operator priorities. We also believe that considering probabilities of larger subtrees in the data as proposed below is conceptually cleaner.

Some of the problems with priorities could also be solved by generating different grammars in more complicated ways than just extracting them directly from the treebank. If we consider our example then such a different grammar can be the following:

```
S -> NumP2 .
NumP2 -> NumP2 + NumP2
NumP2 -> NumP1
NumP1 -> NumP1 * NumP1
NumP1 -> NumP0
NumP0 -> 1
NumP0 -> 2
NumP0 -> x
```

Here we do not investigate such more complicated ways, also because we want to allow parses that do not strictly obey the priorioties. We believe that informal mathematical texts do not strictly obey them either.

3.2 Considering Subtrees

The underlying idea is a simple analogy with the n-gram statistical machine-translation models, or with the large-theory premise selection systems where characterizing formulas by all subterms and subformulas typically considerably improves the performance of the algorithms [9]. While considering subtrees may initially seem computationally involved, we believe that by using good indexing datastructures it becomes feasible, solving some the PCFG problems mentioned above in a reasonably clean way.

In more detail, the idea is as follows. We will extract not just subtrees of depth 2 from the treebank (as is done by PCFG), but all subtrees of certain depth. So far we work with depth 3, but other depths and approaches (e.g., frequency-based rather than depth-based) are possible. During the CYK parsing we will adjust the probabilities of the parsed subtrees also according to the subtree statistics extracted from the treebank. The extracted subtrees will be technically treated as new "grammar rules" of the form:

```
root of the subtree -> list of the children of the subtree
```

We will learn the probabilities of these new grammar rules, formally treating the nonterminals on the left-hand side as different from the old nonterminals when counting the probabilities (this is the current technical solution, which can be modified in the future). Since the right-hand side of the new grammar rules contains whole subtrees, we will be able to compute the parsing probabilities using more context/structural information than in PCFG.

In our example, after the extraction of all subtrees of depth 3 followed by a suitable adjustment of their probabilities, we would get a new "extended PCFG" with the following additional rules:

```
S -> (Num Num + Num) .
Num -> (Num Num * Num) + (Num Num * Num)
Num -> (Num 1) * (Num x)
Num -> (Num 2) * (Num x)
```

This grammar could again parse all the five different parse trees as above, but they would have different probabilities, and the training tree would obviously be the most probable one. For example the probability of the original treebank parse would be:

$$p(\texttt{Num -> (Num 1) * (Num x)}) \times p(\texttt{Num -> (Num 2) * (Num x)}) \times$$
$$p(\texttt{Num -> (Num Num * Num) + (Num Num * Num)}) \times$$
$$p(\texttt{S -> Num .})$$

On the other hand, the probability of some of the parses (e.g., the first two when using the original algorithm) would remain unmodified, because in these parses there are no subtrees of depth 3 from the training tree.

3.3 Technical Implementation

We use a discrimination tree D to store the subtrees from the treebank and to quickly look them up during the chart parsing. When a particular cell in the chart is finished (we know all its parses), we go through all its parses and try to look up their subtree of depth 3 in the discrimination tree D. If we succeed, we recompute the probability according to the new "subtree grammar rule", compare the resulting probability with the old one, and keep the better one.

The subtree lookup is logarithmic, however the number of subtrees we may need to look up can grow quite a lot in the worst case. This is why we have kept the depth at 3 so far. We have not done an extensive evaluation yet, however preliminary experiments with depth 3 and limiting CYK to the best 10 parses show that the new implementation is actually a bit faster than the old one [8]. In particular, when training on all 21873 Flypeck trees and testing on 11911 of them, the new version is about 23% faster than the old one (10342.75s vs 13406.97s total time). The new version also fails to produce at least a single parse less often than the old version (631 vs 818).

This likely means that the subtrees help to promote the correct parse, which in the old version is considered at some point too improbable to make it into the top 10 parses and consequently thrown away by the greedy optimization. The correct (training) parse appears among the best 10 parses in 39% of the 11911 examples for the old algorithm (their average rank there being 2.68), and in 58% cases of the 11911 examples for the new algorithm (their average rank there being 1.97). While this is just a preliminary evaluation where we use the training data also for testing and do not run external typechecking and ATPs, this improvemnt in the parsing precision is very promising. Thorough experimental evaluation and further optimization of the set of subtrees used by the algorithm is future work.

References

[1] James K. Baker. Trainable grammars for speech recognition. In *Speech communication papers presented at the 97th Meeting of the Acoustical Society*, pages 547–550, 1979.

[2] Grzegorz Bancerek and Piotr Rudnicki. A Compendium of Continuous Lattices in MIZAR. *J. Autom. Reasoning*, 29(3-4):189–224, 2002.

[3] Michael Collins. Three generative, lexicalised models for statistical parsing. In Philip R. Cohen and Wolfgang Wahlster, editors, *35th Annual Meeting of the Association for Computational Linguistics and 8th Conference of the European Chapter of the Association for Computational Linguistics, Proceedings of the Conference, 7-12 July 1997, Universidad Nacional de Educación a Distancia (UNED), Madrid, Spain.*, pages 16–23. Morgan Kaufmann Publishers / ACL, 1997.

[4] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry. A machine-checked proof of the Odd Order Theorem. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *ITP*, volume 7998 of *LNCS*, pages 163–179. Springer, 2013.

[5] Thomas Hales. *Dense Sphere Packings: A Blueprint for Formal Proofs*, volume 400 of *London Mathematical Society Lecture Note Series*. Cambridge University Press, 2012.

[6] John Harrison. HOL Light: A tutorial introduction. In Mandayam K. Srivas and Albert John Camilleri, editors, *FMCAD*, volume 1166 of *LNCS*, pages 265–269. Springer, 1996.

[7] Cezary Kaliszyk and Josef Urban. Learning-assisted automated reasoning with Flyspeck. *J. Autom. Reasoning*, 53(2):173–213, 2014.

[8] Cezary Kaliszyk, Josef Urban, and Jiří Vyskočil. Learning to parse on aligned corpora (rough diamond). In Christian Urban and Xingyuan Zhang, editors, *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings*, volume 9236 of *Lecture Notes in Computer Science*, pages 227–233. Springer, 2015.

[9] Cezary Kaliszyk, Josef Urban, and Jiří Vyskočil. Efficient semantic features for automated reasoning over large theories. In Qiang Yang and Michael Wooldridge, editors, *IJCAI'15*, pages 3084–3090. AAAI Press, 2015.

[10] Cezary Kaliszyk, Josef Urban, Jiří Vyskočil, and Herman Geuvers. Developing corpus-based translation methods between informal and formal mathematics: Project description. In Stephen M. Watt, James H. Davenport, Alan P. Sexton, Petr Sojka, and Josef Urban, editors, *Intelligent Computer Mathematics - International Conference, CICM 2014, Coimbra, Portugal, July 7-11, 2014. Proceedings*, volume 8543 of *LNCS*, pages 435–439. Springer, 2014.

[11] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an operating-system kernel. *Commun. ACM*, 53(6):107–115, 2010.

[12] Martin Lange and Hans Leiß. To CNF or not to CNF? an efficient yet presentable version of the CYK algorithm. *Informatica Didactica*, 8, 2009.

[13] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.

[14] Daniel H. Younger. Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10(2):189–208, 1967.

Experiments with State-of-the-art Automated Provers on Problems in Tarskian Geometry (Position paper)

Josef Urban
Czech Technical University
josef.urban@gmail.com

Robert Veroff
University of New Mexico
veroff@cs.unm.edu

Abstract

We describe our initial experiments with several state-of-the-art automated theorem provers on the problems in Tarskian Geometry created by Beeson and Wos. In comparison to the manually-guided Otter proofs by Beeson and Wos, we can solve a large number of problems fully automatically, in particular thanks to the recent large-theory reasoning methods.

1 Introduction

In their 2014 paper [1] Beeson and Wos report on their project which uses OTTER to find proofs of theorems in Tarskian geometry proved in Part I of the book "Metamathematische Methoden in der Geometrie" by Schwabhäuser, Szmielew and Tarski. We have become interested in their work for a couple of reasons.

First, we have recently started to look at good ways how to apply Veroff's techniques such as *hints* and *proof sketches* [5] on large-theory problems coming from ITPs like Mizar [3], Isabelle [4], and HOL [2]. Under the hints strategy, a generated clause is given special consideration if it *matches* (subsumes) a user-supplied hint clause. A *proof sketch* for a theorem T is a sequence of clauses giving a set of conditions *sufficient* to prove T. Typically, the clauses of a proof sketch identify potentially notable milestones on the way to finding a proof. The hints mechanism provides a natural and effective way to take full advantage of proof sketches in the search for a proof. It can be very effective to include as proof sketches proofs of related theorems in the same area of study. The large number of related problems coming from the ITP libraries seem to be a natural target for such techniques.

This however turns out to be nontrivial. Hints currently work on the clause level, and consistency of their symbols (i.e., the same symbol always having the same meaning) across different problems is assumed. This works for the algebraic clausal problems that Prover9 is typically applied to. The large-theory problems are however formulated in FOF, and their skolemization typically produces many symbols that might not be consistently used among the different problems. This, and the large number of problems, formulas and symbols so far confuses the hints technique. The Tarski problems seem to be right in between the two kinds of problems: they are clausal and do not contain too many symbols, yet they at least partially also qualify as large-theory problems. The number of clauses is about 200, and the more advanced problems typically contain all the previously proved lemmas as axioms. Making the hints method work on the Tarski problems would be a useful step towards making hints work on the ITP-generated large-theory problems.

The large-theory aspect of the Tarski problems is another motivation. It turns out that some of the problems really need many axioms (e.g., 59 for and automatically found proof of Satz10.12a by the E.T system). Quite some work has been done recently to develop methods that select the most relevant lemmas for proving a particular conjecture, and to develop specialized

strategies for solving large-theory problems. Beeson and Wos do not use such methods. Instead, they use OTTER and often revert to what is quite close to interactive theorem proving: writing manually intermediate lemmas that guide the proof search. Looking at the results of recent CASC and its large-theory divisions, it would be quite surprising if the performance of unguided OTTER run in an automated mode on the problems would be anywhere near the state-of-the-art systems. That's why we first simply try to run some of those, and see how many problems we can today prove without any interactive guidance. Here we report the results and issues found so far.

2 Running Several ATPs on the Tarski Problems

We started by downloading 164 problems in the Otter format from the web site of the project, and re-proved 163 of them with Otter (using all the tricks used by Beeson and Wos). The remaining Satz11.15b.in – which was marked as work-in-progress – was eventually proved in 19000 s by Otter.

We first wrote a script that commented out the main sources of "guiding information" in the files: the special parameter settings, the hints and the demodulators. Then we ran Otter and Prover9 on such "blank" problems in auto mode for 300 s. Otter solved 72 problems and Prover9 102. In 20 s Prover9 solved only 92 problems . This indicated that using more CPU time and different strategies would likely help further.

To be able to use other ATPs, we have used the ladr_to_tptp translator, followed by some pre- and post-processing that gets around using some symbols both as functions and as predicates in the Otter problems. Since the problems are large and may profit from conjecture-oriented clause selection mechanisms in E, we also renamed the status of the conjecture-originated axioms to "negated_conjecture". Then we ran E 1.8 (using "–auto-schedule", i.e., its strategy scheduling mode) for 300 s solving 125 problems, and E.T 0.1 also for 300 s, solving 129 problems. E.T uses more strategies than E, and applies stronger axiom pruning. The strongest system is however Vampire 3.0, which solves 137 of the problems in 300 s. We have also tried Z3 (to get some more solutions), which solves 92 files in 300 s. All the systems together can prove 141 problems in 300 s at this point – about twice as many as unguided Otter. The manual coding of the proofs by providing intermediate steps to Otter could have been largely avoided with quite limited resources.

Finally, we did some experiments with higher time limits. Three more problems could be solved by running Vampire 3.0 in the CASC mode for 1 hour. Running Vampire for longer in this mode does not work – it seems that the CASC mode is limited to such shorter overall times. Vampire and E.T (run with even higher time limits) together add 6 more problems, making the final count so far at 147 fully automatically solved problems out of the 164. These results are however a bit questionable due to the issues described next.

3 Problems with the Problems

Already during the translation to TPTP and the initial runs with other ATP we have encountered several kinds of problems with the files. Some of them – e.g. problems with naming of clauses – prevent us so far from running stronger learning-based large-theory systems such as MaLARea on the problems. We list them here and note how they could be prevented. We have spent quite significant effort on cleaning up the files, however it is still not completely finished.

- In several cases the statement of a particular lemma or definition is different in different problems. This is really dangerous in a larger formalization and could lead to many issues. Sometimes this only involves naming of the variables, but sometimes the clauses really differ. This could not happen in an ITP-based development, where each lemma/definition is stated only once, and ATP problems are generated automatically by the corresponding hammer systems. Even pervasive use of simple ATP mechanisms such as includes should be enough to prevent this.

- Using manual skolemization and the clausal form has its dangers: some of the different statements of the same formula from the book are caused by bugs in manual Skolemization, or even just by introducing differently named skolem functions and constants. This could be prevented by switching to formulas instead of clauses.

- While the Otter/Prover9 format occasionally allows more mathematician-friendly notation than TPTP, unlike in TPTP the name of a clause is just an unchecked comment that can be anywhere and have typos in it. This already prevents simple duplication/consistency checking as done by the TPTP tools, and also more advanced checking of the proofs (and formulas uses in them) as done, e.g., by the GDV verifier.

- In some of the clauses there are clear typos, leading, e.g., to the appearance of singleton variables. If a Prolog-friendly language such as TPTP was used, they would be easily detected and reported when loading the formulas into Prolog.

We were quite surprised that relatively many of such issues have escaped not only the authors, but also the IJCAR'14 refereeing – at least there is no mention of the debugging processes in the paper. In the conferences focused on interactive theorem proving as well as when reviewing submissions to some large formal libraries, it is today a common practice to not only run and re-check the formal developments, but also to look at and point out various issues with definitions, obvious naming problems, dangerous use of additional axioms, etc. It would be good if IJCAR – and in general the ATP community – adopted similar refereeing practices in such cases.

4 Future Work

Our plan is to try to prove as many Tarski problems as possible by the standard (large-theory) techniques available in systems like E, Prover9 and Vampire, and then proceed to test Prover9 techniques that transfer hints between the problems. It is likely that in this almost-large-theory setting we will need to complement the hints method with methods that select only the most relevant hints for a particular problem. Such methods are similar to the recent methods for premise and lemma selection developed in the context of large theories. We hope that this way we will eventually (thanks to the Tarski problems) also arrive at good techniques for hint guidance in large theories.

Beeson and Wos have – independently of our feedback and later taking it into account – started to work on cleaning up the Tarski problems, and very recently produced a new version of their problems. A useful future extension would be to import and verify the whole development in a safe ITP such as HOL Light, using the existing TPTP proof importing capabilities.

5 Acknowledgments

Thanks to Mike Beeson for providing us with the problems at IJCAR'14 and discussing several issues.

References

[1] Michael Beeson and Larry Wos. OTTER proofs in Tarskian geometry. In Stéphane Demri, Deepak Kapur, and Christoph Weidenbach, editors, *Automated Reasoning - 7th International Joint Conference, IJCAR 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 19-22, 2014. Proceedings*, volume 8562 of *Lecture Notes in Computer Science*, pages 495–510. Springer, 2014.

[2] Cezary Kaliszyk and Josef Urban. Learning-assisted automated reasoning with Flyspeck. *J. Autom. Reasoning*, 53(2):173–213, 2014.

[3] Cezary Kaliszyk and Josef Urban. MizAR 40 for Mizar 40. *J. Autom. Reasoning*, 55(3):245–256, 2015.

[4] Daniel Kühlwein, Jasmin Christian Blanchette, Cezary Kaliszyk, and Josef Urban. MaSh: Machine learning for Sledgehammer. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *ITP 2013*, volume 7998 of *LNCS*, pages 35–50. Springer, 2013.

[5] Robert Veroff. Using hints to increase the effectiveness of an automated reasoning program: Case studies. *J. Autom. Reasoning*, 16(3):223–239, 1996.

Functional Pearl: the Proof Search Monad

Jonathan Protzenko

Microsoft Research
protz@microsoft.com

Abstract

We present the proof search monad, a set of combinators that allows one to write a proof search engine in a style that resembles the formal rules closely. The user calls functions such as `premise`, `prove` or `choice`; the library then takes care of generating a derivation tree. Proof search engines written in this style enjoy: first, a one-to-one correspondence between the implementation and the derivation rules, which makes manual inspection easier; second, proof witnesses "for free", which makes a verified, independent validation approach easier too.

1 Theory and practice

This paper attempts to present, in a tutorial-style, the design of an OCaml library. In order to facilitate the discussion, we focus on a very constrained logic; later (Section 5), we briefly discuss how to extend the library to cover more use-cases. The original motivation for the library was to serve as a core building block for the type-checker of Mezzo [12]. The nature of the core, minimal logic that we are about to present is, of course, inspired by typical nature of type-checking problems: it features equality, quantifiers, and positive literals; Section 5 mentions how to extend it with, among other things, function symbols and variance (positive/negative positions), as is typical for type-checking problems.

1.1 A minimal theory

We are concerned with proving the validity of logical formulas; that is, with writing a search procedure that determines whether a given goal is satisfiable. To get started, we consider a system made up of conjunctions of equalities, along with existential quantifiers. Any free variables are assumed to be universally quantified. For instance, one may want to prove the following formula:

$$\exists y.\ x = y \tag{1}$$

In order to show the validity of this judgement, one usually builds a proof derivation using rules from the logic. In our case, the rules are given in Figure 1, where $[x/y]P$ means "substitute x with y in P". For instance, proving Equation 1 requires applying ExistsE, then Refl.

$$\textsc{Refl}\ \frac{}{x = x} \qquad \textsc{And}\ \frac{P \qquad Q}{P \wedge Q} \qquad \textsc{ExistsE}\ \frac{[x/y]P}{\exists y.\ P}$$

Figure 1: A simple logic

$$\textsc{Refl}\ \frac{}{V, \sigma \vdash x = x \dashv \sigma} \qquad \textsc{Subst}\ \frac{V, \sigma \vdash \sigma P \dashv \sigma'}{V, \sigma \vdash P \dashv \sigma'} \qquad \textsc{Inst}\ \frac{x \in V \qquad y^? \in V \qquad y^? \notin \sigma \qquad V, \{y^? \mapsto x\} \circ \sigma \vdash P \dashv \sigma'}{V, \sigma \vdash P \dashv \sigma'}$$

$$\textsc{And}\ \frac{V, \sigma \vdash P \dashv \sigma' \qquad V, \sigma' \vdash Q \dashv \sigma''}{V, \sigma \vdash P \wedge Q \dashv \sigma''} \qquad \textsc{ExistsE}\ \frac{V \uplus y^?, \sigma \vdash P \dashv \sigma'}{V, \sigma \vdash \exists y.\ P \dashv \sigma'_{\restriction V}}$$

Figure 2: Algorithmic proof rules

These rules embody the Truth of our logic, i.e. an omniscient reader may use them to show with absolute certainty that a given formula is true. However, if one wants to algorithmically determine whether a given formula is true, ExistsE is useless. Indeed, unless the algorithm (solver) is equipped with superpowers, it cannot magically guess, out of the blue, a suitable x in ExistsE that will ensure the remainder of the derivation succeeds. To put it another way, x is a free variable (a parameter) of ExistsE; the whole point of writing a proof search algorithm is to 1) find that ExistsE is the right rule to apply, and 2) find that x is a suitable value for instantiating y, because it will make $y = x$ succeed.

Hence, in order to build a *search procedure* for that logic, one will use another set of *algorithmic* rules, which hopefully enjoy:

soundness : if the algorithmic rules succeed, then there exists a derivation in the logic that proves the validity of the original formula, and

completeness : if the algorithmic rules fail, then there exists no derivation in the logic that would prove the validity of the original formula.

For instance, in our logic of existentially-quantified conjunctions of equalities, one may want to use the rules from Figure 2. These rules differ from Figure 1 in that they are algorithmic; they take an input ($\vdash$) and return an output ($\dashv$).

In particular, in order to determine suitable values for the x parameter in ExistsE, the implementation reasons in terms of substitutions. V is a set of variables which may be substituted (recall that free variables are considered universally quantified, hence not eligible for substitution); variables that may be substituted are typeset as $y^?$. The algorithm has internal state, that is, it carries a substitution σ. Upon hitting an existential quantifier $y^?$, the algorithmic rules *open* $y^?$ and mark it as eligible for substitution (ExistsE). Later on (for instance, upon hitting $y^? = x$), the algorithm may pick a substitution for $y^?$ using Inst. A substitution may be applied at any time (Subst). The preconditions of Inst guarantee that the algorithm makes at most one choice for instantiating $y^?$.

In other words, the algorithmic rules *defer* the *instantiation* of the existential quantifier until some sub-goal, later on, gives us a *hint* as to what exactly this instantiation should be. This *implementation technique* is known as *flexible variables*.

The new algorithmic rules differ from the original logical rules significantly; first, there are five rules for the algorithmic system, compared to just three for the logical system. Second, these five rules do not map trivially to their counterparts in the logical system. Third, these rules are still very much abstract; the implementation that we are about to roll out uses an optimized representation for substitutions (union-find) that is not formalized in Figure 2. Phrased

differently, one not only needs to check that the algorithmic rules are faithful to the proof rules, but also that the implementation itself is faithful to the algorithmic rules.

This paper presents a library that allows one to write an implementation of the algorithmic rules while automatically generating a derivation. The library forces the client code to lay out premises, rule applications and instantiations. The level of detail of the resulting derivation is left up to the client code; the user may wish to record a proof derivation using the proof rules, or record a trace of the algorithm using the algorithmic rules. In any case, the derivation serves as a witness; in the case of a proof derivation, a validator may certify that the proof is valid, while in the case of an algorithmic trace, the user may verify the algorithm, or inspect the trace for debugging or feedback purposes.

The library has been used, in a preliminary form, to implement the core of the Mezzo type-checker [12]. This paper presents a cleaned-up, isolated version of this library that exposes a proper interface using monads and domain-specific combinators.

1.2 An implementation with flexible variables and union-find

The logic we present is a much simplified version of the logic (type system) of Mezzo [10]. In the present document, we only mention the right-exists quantifier. General systems such as Mezzo have all four possible combinations of left/right exists/forall. The right-elimination of existential quantifiers, or the left-elimination of universal quantifiers gives *flexible variables*, while the right-elimination of universal quantifiers, or the left-elimination of existential quantifiers gives universally-quantified variables, also called *rigid variables*.

In order to simplify the problem, we assume that all existential variables have been introduced as flexible variables already. That way, we won't be sidetracked, talking about binders and the respective merits of De Bruijn *vs.* locally nameless. Furthermore, we assume that all instantiations of flexible variables are legal. This is not true in general: for instance, if the goal is $\forall x, \exists y^?, \forall z.\ P$, picking $y^? = z$ makes no sense. Mezzo forbids this choice using *levels* [11]; in the present document, we skip this discussion altogether and assume that "all is well". Finally, although in a general setting, several rules may trigger for a given goal (this is the case in Mezzo), the algorithmic set of rules we use is syntax-driven: the syntactic shape of the goal determines which rule should be applied.

We thus restrict our formulas to conjunctions of equalities between variables. The plan is to write a solver that takes, as an input, a formula, and outputs a valid substitution, if any. That is, write an algorithm that abides by the rules from Figure 2. For instance, one may want to solve: $x = y^? \wedge z = z$. A solution exists: the solver outputs $\sigma = \{y^? \mapsto x\}$ as a valid substitution that solves the input problem. However, if one attempts to solve: $x = y^? \wedge y^? = z$, the solver fails to find a proper substitution, and returns nothing. Indeed, the first clause demands that $y^?$ substitutes to x, meaning that the second clause becomes $x = z$, which always evaluates to false (x and z are two distinct rigid variables).

Once the algorithm has run, we obtain an output substitution σ. One can, if they wish to do so, take the reflexive-transitive closure σ^*, and apply it to a flexible variable (say, $y^?$) to recover the parameter of ExistsE that should be used in the logical rules (here, x). This way of checking correctness is not satisfactory and does not scale if nested quantifiers appear in the goal; the point of the subsequent sections is to make sure the search algorithm produces a proper proof witness (derivation) that has a tree-like structure and does not leak implementation details (such as substitutions).

We implement proof search in OCaml [8] (Figure 3); we implement substitutions using a union-find data structure [1, 13]. The data type of formulas is self-explanatory. Variables

```
type formula =
| Equals of var * var
| And of formula * formula

and var = P.point
```

```
and descr =
  | Flexible
  | Rigid

and state = descr P.state
```

Figure 3: Formulas and state

```
module MOption = struct
  (* ... defines [return], [nothing] and [>>=] *)
end

let unify state v1 v2: state option =
  match P.find v1 state, P.find v2 state with
  | Flexible, Flexible
  | Flexible, Rigid ->
      return (P.union v1 v2 state)
  | Rigid, Flexible ->
      return (P.union v2 v1 state)
  | Rigid, Rigid ->
      if P.same v1 v2 state then
        return state
      else
        nothing
```

```
let rec solve state formula: state =
  match formula with
  | Equals (v1, v2) ->
      unify state v1 v2
  | And (f1, f2) ->
      solve state f1 >>= fun state ->
      solve state f2
```

Figure 4: Solver for the simplified problem

are implemented as equivalence classes in the *persistent* union-find data structure, which the module `P` implements. The V, σ parameters in our rules are embodied by the `state` type; just like the σ parameter is chained from one premise to another (AND), `state` is an input and an output to the solver. Just like the σ parameter in the rules, a `state` of the persistent union-find represents a set of equations between variables. The algorithmic rules mentioned a theoretical σ parameter; the `state` is our specific implementation choice.

The choice of a union-find (as opposed to explicit substitutions) is irrelevant. All that matters is that we pick a data structure that models substitutions, and that the structure be *persistent*.

Figure 4 implements a solver for our minimal problem; since we perform computations that either return a result of a failure, the code leads itself well to an implementation using monads [14, 15], in our case, the `MOption` monad. The `Some state` is for success, meaning a substitution has been found, while the `None` case means no solution exists. The solver is complete.

The solver uses `MOption.>>=` to sequence premises in the `And` case. It doesn't keep track of premises; it just ensures (thanks to `>>=`) that if the first premise evaluates to `nothing`, the second premise is not evaluated, since it is suspended behind a `fun` expression (OCaml is a strict language).

```
(* These two modules belong to the library. *)
module type LOGIC = sig
  type formula
  type rule_name
end

module Derivations.Make (L: LOGIC) = struct
  type derivation = L.formula * rule
  and rule = L.rule_name * premises
  and premises = Premises of derivation list
end

(* This is the client code using modules from the library. *)
module MyLogic = struct
  type formula = ... (* as before *)
  type rule_name = R_And | R_Refl | R_Inst
end
module MyDerivations = Derivations.Make(MyLogic)
```

Figure 5: The functor of proof trees (library and client code)

2 Building derivations

There are two shortcomings with this solver. First, the `unify` sub-routine conflates several rules together. Indeed, the `return (P.union ...)` expression hides a combination of Inst and Refl. Second, we have no way to replay the proof to verify it independently. One may argue that in this simplified example, the outputs substitution *is* the proof witness: one can just apply the substitution to the original formula and verify that all the clauses are of the form $x = x$, without the need for a proof tree. In the general case, however, the proof tree contains the ExistsE rule, and proof witnesses are attached to arbitrary nodes of the tree. We thus need to build a properly annotated proof tree in the general case.

2.1 Defining proof trees

One way to make the solver better is to make sure each step it performs corresponds in an obvious manner to the application of an admissible rule. To that effect, we define the data type of all three rules in our system, which we apply to the functor of *proof trees* (Figure 5).

We record applications of Inst, Refl and And. This produces a derivation tree (algorithm trace) that makes sure that the algorithm follows the algorithmic rules from Figure 2. Section 4 shows how to generate a different, more compact tree that matches the rules from Figure 1.

A `derivation` tree is a pair of a `formula` (the goal we wish the prove) and a `rule` (that we apply in order to prove the goal). A `rule` has a name and `premises`; the `premises` type is simply a `derivation list` (the `Premises` constructor is here to prevent a non-constructive type abbreviation). When using the library, the client is expected to make sure that each `rule_name` is paired with the proper number of premises (0 for Refl, 1 for Inst and 2 for And); this is not enforced by the type system.

In the (simplified) sketch from Figure 5, rule names are just constant constructors, since the rule parameters (such as x and $y^?$ in Inst) can be recovered from the `formula`. In the general case (Section 4), the various constructors of `rule_name` do have parameters that record how one specific rule was instantiated.

```
module WriterT (M: MONAD) (L: MONOID): sig
  type 'a m = (L.a * 'a) M.m
  val return: 'a -> 'a m
  val ( >>= ): 'a m -> ('a -> 'b m) -> 'b m

  val tell: L.a -> unit m
end = ...

module M = MOption
module MWriter = WriterT(M)(L)
```

```
module L = struct
  type a = Derivations.derivation list
  let empty = []
  let append = List.append
end
```

Figure 6: The writer monad transformer (library code)

2.2 Proof tree combinators

We previously used the `>>=` operator from the `MOption` monad in order to chain premises (Figure 4). We now need a new operator, that not only *binds* the result (i.e. stops evaluating premises after a failure, as before), but also *records* the premises in sequence, in order to build a proper derivation. The former is still faithfully implemented by the option monad; the latter is implemented by the writer monad [5].

Computations in the writer monad return a result (of type `'a`) along with a log of elements (of type `L.a`). The (usual) `>>=` and `return` combinators operate on the result part of the computation, while the (new) `tell` combinator operates on the logging part of the computation. The `tell` combinator appends a new element to the log; this is done by way of the `MONOID` module type, which essentially demands a value for the `empty` log, and a function to `append` new entries into the log.

In order to get a new `>>=` operator that combines the features of the option and writer monads, we apply the `WriterT` monad transformer to the `MOption` monad (Figure 6) and obtain `MWriter`.

The type of computations `'a MWriter.t` boils down to `(derivation list * state) option` after functor application. A computation in the monad represents a given point in the proof; the solver is focused on a rule; has proved a number of premises so far (the `derivation list`); has reached a certain `state` (threaded through the premises). The `option` type accounts for failure; in case a premise cannot be proved, the computation aborts and becomes `None`.

Once all the premises have been proven, one needs to draw a horizontal line and reach the conclusion of the proof. That is, take the final state and the list of premises, and generate a `derivation` that stands for the application of the entire rule.

Contrary to the first implementation (Figure 4), where the working state and the return value of `solve` both had type `state option`, we now distinguish between an `outcome` (the result of a call to `solve`) and a working state (a computation in the `MWriter` monad).

An `outcome` is the pair of a final `state` along with a `derivation` that justifies that we reached this state. The pair is wrapped in `M.t` (here, `option`): if the computation of premises is a failure, then the proof of the desired goal is a failure too.

The type `outcome` (Figure 7) is parametric: it works for any state that the client code uses. In other words, our library is generic with regards to the particular `state` type the client uses.

We now have a duality between the `outcome` type (the result of solving a goal) and the `m` type (a computation within the monad, i.e. a working state between two premises). Therefore, we introduce two high-level combinators: `premise` and `prove`. The former goes from `outcome` to `m`: it injects a new sub-goal as a premise of the rule we are currently trying to prove. The

```
(* This snippet is in the [MWriter(M)(L)] monad. Upon a first reading, think
   [module M = MOption]. *)
type 'a outcome = ('a * derivation) M.m

let premise (outcome: 'a outcome): 'a m =
  M.bind outcome (fun (state, derivation) ->
    tell [ derivation ] >>= fun () ->
    return state
  )

let prove (goal: goal) (x: ('a * rule_name) m): 'a outcome =
  M.(x >>= fun (premises, (state, rule)) ->
    return (state, (goal, (rule, Premises premises))))

let axiom (state: 'a) (goal: goal) (axiom: rule_name): 'a outcome =
  prove goal (return (state, axiom))

let qed r e =
  return (e, r)

let fail: 'a outcome =
  M.nothing
```

Figure 7: The high-level combinators for building proof derivations (library code)

latter goes from `m` to `outcome`: if all premises have been satisfied, it draws the horizontal line that builds a new node in the derivation tree.

- `premise` is the composition of `tell`, which records the derivation for this sub-goal, and `return`, which passes the state on to the next sub-goal.
- `prove` is a computation in the `M` monad (here, `MOption`). If all the premises have been satisfied, it bundles them as a new node of the derivation tree. If a premise failed, then `x` is `M.nothing`, and `prove` also returns a failed outcome.
- `axiom` is short-hand for a rule that requires no premises.
- `fail` is for situations where no rule applies: this is a failed outcome.
- `qed` is a convenience combinator that pairs the state with the name of the rule we want to conclude with; it makes the implementation of `solve` (Figure 8) more elegant.

2.3 A solver in the new style

Figure 8 demonstrates an implementation of `solve` in the new style. Compared to the previous implementation (Figure 4):

- `prove_equality` makes it explicit which rules are applied, and singles out two distinct rule applications in the flexible-rigid case;
- the premises of each rule are clearly identified;
- axioms and failure conditions are explicit,
- the `And` case is easy to review manually, to make sure that no premise was forgotten.

```
let rec prove_equality (state: state) (goal: formula) (v1: var) (v2: var) =
  let open MOption in
  match P.find v1 state, P.find v2 state with
  | Flexible, Flexible
  | Flexible, Rigid ->
      let state = P.union v1 v2 state in
      prove goal begin
        (* Recursive call reduces to the [axiom ... R_Refl] case below. *)
        premise (prove_equality state goal v1 v2) >>=
        qed R_Instantiate
      end
  (* ... *)
  | Rigid, Rigid ->
      if P.same v1 v2 state then
        axiom state goal R_Refl
      else
        fail

let rec solve (state: state) (goal: formula): state outcome =
  match goal with
  | Equals (v1, v2) ->
      prove_equality state goal v1 v2
  | And (g1, g2) ->
      prove goal begin
        premise (solve state g1) >>= fun state ->
        premise (solve state g2) >>=
        qed R_And
      end
```

Figure 8: A solver written using the high-level combinators (client code)

This is, as mentioned previously, a minimal example that showcases the usage of the library. In the implementation of Mezzo, switching the core of the type-checker to this style revealed several bugs where premises were not properly chained or simply forgotten.

3 Backtracking

3.1 Limitations of the option monad

We now extend our formulas with disjunctions (Figure 10). A consequence is that we now need our base monad `M` to offer a new operation; namely, one that, among several possible choices, picks the first one that is not a failure. We thus augment `MOption` with a search combinator (Figure 10), which in turn allows us to implement a high-level `choice` combinator for our library. The `choice` combinator attempts to prove a `goal` by trying a function `f` on several arguments of type `a`, each of which has a given `outcome`. We extend `solve` with an extra case, which attempts to prove a disjunction by first trying a left-elimination (Or-L, Figure 9), then a right-elimination (Or-R).

The solver can now solve problems of the form $x = z \vee y^? = z$. It fails, however, to solve problems of the form $(y^? = x \vee y^? = z) \wedge y^? = z$. The reason is, the option monad is not powerful enough: upon finding a suitable choice in the disjunction case, it commits to it and drops the

$$\textsc{Or-L}\ \frac{V \vdash P \dashv V'}{V \vdash P \vee Q \dashv V'} \qquad \textsc{Or-R}\ \frac{V \vdash Q \dashv V'}{V \vdash P \vee Q \dashv V'}$$

Figure 9: New proof rules for disjunction

```
(* We extend formulas with disjunctions. *)
type formula =
  (* ... *)
  | Or of formula * formula

(* The logic is also extended with two rules. *)
type rule_name =
  (* ... *)
  | R_OrL
  | R_OrR

module MOption = struct
  (* ... *)
  let rec search f = function
    | [] -> None
    | x :: xs ->
        match f x with
        | Some x -> Some x
        | None -> search f xs
end

(* Equipped with [search], we define the [choice] library combinator... *)
let choice (goal: goal) (args: 'a list) (f: 'a -> ('b * rule_name) m): 'b outcome =
  M.search (fun x -> prove goal (f x)) args

(* ...which one uses as follows: *)
let rec solve (state: state) (goal: formula): state outcome =
  match goal with
  (* ... *)
  | Or (g1, g2) ->
      choice goal [ R_OrL, g1; R_OrR, g2 ] (fun (r, g) ->
        premise (solve state g) >>=
        qed r
      )
```

Figure 10: The choice combinator (library and client code)

```
module LL = LazyList
module MExplore
  type 'a m = 'a LL.t
  let return = LL.one
  let ( >>= ) = LL.flattenl (LL.map f x)
  let nothing = LL.nil
  let search f l = LL.bind (LL.of_list l) f
end
```

Figure 11: The exploration monad

other one. In other words, when hitting the disjunction, `MOption` commits to $\sigma = \{y^? \mapsto x\}$, instead of keeping $\sigma = \{y^? \mapsto z\}$ as a backup solution. Phrased yet again differently, we need to replace `MOption` with the non-determinism monad that will implement *backtracking*.

3.2 The exploration monad

Conceptually, we want to change our way of thinking; instead of thinking of `solve` as a function that returns *a solution*, we now think of it as a function that returns *several possible solutions*. The state is now a set of states, each of which represent a path in the search tree of derivation trees.

The monad of non-determinism is implemented using lists; OCaml is a strict language, so we write the non-determinism monad (also known as the exploration or backtracking monad) using lazy lists (Figure 11).

The reader can now go back and replace `module M = MOption` with `module M = MExplore` in Figure 6. The rest of the library remains unchanged; the `solve` function (the client code) is also unchanged; and the combinators of the library now implement backtracking.

In particular, the earlier example of $(y^? = x \vee y^? = z) \wedge y^? = z$ is now successfully solved by the library. Thanks to laziness, no extra computations occur; further solutions down the lazy list are only evaluated if the first ones failed.

4 Extension: quantifiers and proof trees

We mentioned earlier that the derivation we were building tracked the application of algorithmic rules; that is, we were building a *trace* of the algorithm. While the trace is useful to extract information for the user, one may also want to build a proper proof witness in order to certify the validity of the formula.

In order to make a proof tree relevant and not just provide the substitution as the proof witness, we introduce quantifiers to the language, and construct proof trees that apply the proof rules from Figure 1, Figure 9, Figure 12. The nodes of the proof tree are rules; each node is annotated, if applicable, by its implicit parameters. That is, Refl and ExistsE are annotated with their implicit x parameter.

The updates to the library required to implement quantifiers are minimal; the bulk of the work is essentially writing substitution and a proper treatment of binders on the client-side.

Figure 13 presents in an informal style the series of updates required.

i) We augment the data type of formulas with quantifiers; we replace the type of rules with the rules from the logic. Furthermore, we demand that the Refl and ExistsE rules be annotated with their argument.

$$\textsc{ForallE}\ \frac{P}{\forall y.\ P}$$

Figure 12: Extra rule for the universal quantifier

```
(* i) Update of the [MyLogic] module. *)
type formula =
  (* ... *)
  | Exists of string * formula
  | Forall of string * formula

type rule_name =
  | R_And
  | R_Refl of atom
  | R_OrL
  | R_OrR
  | R_ExistsE of atom
  | R_ForallE

(* ii) Update of the [Derivations] module. *)
type derivation =
  goal * rule

and goal =
  L.state * L.formula

and (* ... *)

(* iii) Update of the [Combinators] module. *)
let prove (goal: Logic.formula) (x: ('a * rule_name) m): 'a outcome =
  M.(x >>= fun (premises, (env, rule)) ->
    return (env, ((env, goal), (rule, Premises premises))))

(* iv) Update of the client. *)
let rec solve (state: state) (goal: formula): state outcome =
  (* ... *)
  | Exists (atom, g) ->
      let var, g, state = open_flexible state atom g in
      let var = assert_open var in
      prove goal begin
        premise (solve state g) >>= fun state ->
        qed (R_ExistsE (name var state)) state
      end
```

Figure 13: Dealing with quantifiers

Bound variables are globally-unique atoms (strings); open variables are equivalence classes of the union-find, as before (not shown here).

ii) We previously did not distinguish between a `goal` and a `formula`; this was only possible because we assumed all variables were initially open, meaning that we could deference an open variable in any `state`. Now, we open binders and substitute variables, through the allocation of new points in the union-find (the `state`). Therefore, a given `formula` only makes sense when paired with a specific `state`.

iii) We update the `prove` combinator to record the `state` upon creating a new node in the derivation tree. More precisely, the `prove` combinator records the `state` after the premises have been satisfied.

iv) Only a slight is needed on the client side to record a proper proof witness: in the `Exists` case, the solver prods the union-find state to discover the instantiation choice that *has been made* for the existentially-quantified variable, and records it in the proof tree.

The sample code in the library comes with a pretty-printer. Here is the output for a simple formula that combines all features from our formula language.

```
prove ∀x. ∀z. ∃y. (y = x \/ y = z) /\ y = z using [forall]
| prove ∀z. ∃y. (y = x \/ y = z) /\ y = z using [forall]
| | prove ∃y. (y = x \/ y = z) /\ y = z using [exists[z]]
| | | prove (z = x \/ z = z) /\ z = z using [/\]
| | | | prove z = x \/ z = z using [\/_r]
| | | | | prove z = z using axiom [refl[z]]
| | | | prove z = z using axiom [refl[z]]
```

5 Extending the library; extending the logic; limitations

The approach advocated in the present paper works well for sequent-style calculi; in the context of Mezzo, the logic is extended with the following extra features:

- function symbols, such as, but not limited to: ML arrow types ($\rightarrow$), type applications ($\mathsf{list}\ \alpha$) and constructor applications ($\mathsf{Cons}\ \{\mathsf{head} : \alpha;\ \mathsf{tail} : \mathsf{list}\ \alpha\}$)
- positive and negative positions (also known as "variance" in type-checking lingo; this applies to function symbols, such as arrows or type constructors)
- higher-order quantification ($\forall(p : \mathsf{predicate})\ldots$)
- affinity (where some hypothesis may be used at most once)
- framing, which bears some similarities with focusing.

This in turn requires the client code to keep track of more information, while also adopting more sophisticated data structures. In particular, the client code now carries, in addition to a substitution (*a.k.a.* union-find), a *set* of available hypotheses, which flows left-to-right in the algorithmic rules. Changing polarities changes the direction of the flow.

A limitation of this library is that it only works as long as every branch of the exploration terminates; contrary to Kiselyov *et al.*'s library [7], we do not implement fair interleaving. One could conceivably bound the depth of the search tree, but the exploration of the tree remains sequential, not concurrent.

The logic does not necessarily have to be decidable for this approach to work well; indeed, we conjecture that type-checking Mezzo programs is not decidable [12, p. 167]. What matters

is that exploration follows a deterministic set of rules; in Mezzo, the backtracking points are chosen and controlled [12, p. 165]. We, of course, explore a fine-tuned subset of the search space.

If one is willing to give up on modularity, stronger static guarantees can be attained by making the `rule_name` type more specific; namely, by encoding in each constructor the number of premises required. The drawback is that the library now has to be aware of the specific logic; in the current state of things, the library is completely agnostic with regards to the client code's particular logical system.

It is unclear how one could memoize the sub-computations, as they depend very much on the current state, which is likely to change at every step.

6 Source code

The library is available online at `https://github.com/msprotz/proof-search-monad`. The file `example01.ml` contains the full implementation of the primitive solver described in Section 1. The file `example02.ml` contains the backtracking solver written within the proof search monad, as described in Section 3. One can get the non-backtracking version, described in Section 2, by replacing `MExplore` with `MOption`. Finally, the file `example03.ml` contains the final algorithm described in Section 4. The representation of binders adopted in the last example is suboptimal; bound variables are represented using globally-unique atoms. One may want to use locally nameless, with De Bruijn indices for bound variables (as used in Mezzo). It allows keeps the boilerplate to a minimum, though.

7 Related work

An article titled "The Proof Monad" already exists [6]; in spite of the closely related title, the article is concerned with a slightly different problem, namely giving an operational semantics to tactic languages used in theorem provers. In that sense, the article is related to Mtac [16], which is also concerned with a proper monad for writing tactics in Coq.

Hedges [3] compares various explorations monads, notably using the continuation monad, the selection monad [2] and their respective monad transformers. The main focus of the article seems to be the relationship between backtracking and game theory.

Hinze [4] shows how to use the backtracking monad transformer, i.e. add backtracking to any existing monad. It would be interesting to determine whether our library can be re-implemented using a backtracking monad transformer, rather than the writer transformer applied to the monad of non-determinism. The (draft) version of the library used in Mezzo also builds failed derivations (as error messages) that list all attempted proofs, along with the first premise that failed; doing so would not be possible using exceptions.

The `choice` operator is related to polarization and focusing [9]. For instance, in the problem $y^? = x \vee y^? = z$, depending on which side of the disjunction the algorithm considers first, the outcome is going to be different. This is analogous to a synchronous phase (where the order of the rules matters, and where a particular choice may have consequences on the rest of the search). Similarly, one may swap premises chained by the `>>=` operator, as the order doesn't matter. This is analogous to a asynchronous phase.

8 Conclusion

We presented a support library for writing a proof search engine using backtracking. The library is parameterized by: the type of formulas; the type of rule applications; the internal state type of the client. This leaves complete freedom for the client to define their own logic. By merely using the combinators of the library, the client gets derivations built for free; this allows a separate verifier to independently check the steps required to prove the formula. By opting into the library, the client gets to rewrite their code in a new syntactic style that makes rule application explicits, forbids "bundled" applications of multiple rules at the same time and clearly lays out the premises required to prove a judgement. Since the code resembles the logical rules, mistakes are easier to spot.

The logic presented in this paper is as simple as it gets. It does, however, highlight the main concepts. A version of this library is used in the core of Mezzo's type-checker. The version of the library used in Mezzo also builds failed derivations; these failed derivations stop at the first failed premise or, in case of a choice, list all the failed attempts. We have not yet explained this last feature as a clean combination of monads and operators, but hope to do so in the near future.

Acknowledgments

I wish to thank Gabriel Scherer who helped me sketch the initial version of the library; François Pottier for motivating me enough that I would want to write about it, as well as providing the persistent union-find implementation; and Anonymous Review #2 for a very high quality review with numerous excellent suggestions.

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (Third Edition)*. MIT Press, 2009.

[2] Martin Escardó and Paulo Oliva. What sequential games, the tychonoff theorem and the double-negation shift have in common. In *Proceedings of the third ACM SIGPLAN workshop on Mathematically structured functional programming*, pages 21–32. ACM, 2010.

[3] Jules Hedges. Monad transformers for backtracking search. In *ACM SIGPLAN Workshop on Mathematically Structured Functional Programming (MSFP)*, 2014.

[4] Ralf Hinze. Deriving backtracking monad transformers. In *ACM SIGPLAN Notices*, volume 35(9), pages 186–197. ACM, 2000.

[5] Mark P Jones. Functional programming with overloading and higher-order polymorphism. In *Advanced Functional Programming*, pages 97–136. Springer, 1995.

[6] Florent Kirchner and César Muñoz. The proof monad. *The Journal of Logic and Algebraic Programming*, 79(3):264–277, 2010.

[7] Oleg Kiselyov, Chung-chieh Shan, Daniel P Friedman, and Amr Sabry. Backtracking, interleaving, and terminating monad transformers:(functional pearl). In *ACM SIGPLAN Notices*, volume 40(9), pages 192–203. ACM, 2005.

[8] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The OCaml system release 4.02. *Institut National de Recherche en Informatique et en Automatique*, 2014.

[9] Chuck Liang and Dale Miller. Focusing and polarization in intuitionistic logic. In *Computer Science Logic*, pages 451–465. Springer, 2007.

[10] François Pottier and Jonathan Protzenko. Programming with permissions in Mezzo. In *International Conference on Functional Programming (ICFP)*, pages 173–184, September 2013.

[11] François Pottier and Didier Rémy. The essence of ML type inference. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005.

[12] Jonathan Protzenko. *Mezzo: a typed language for safe effectful concurrent programs*. PhD thesis, Université Paris Diderot, September 2014.

[13] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, April 1975.

[14] Philip Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2:461–493, 1992.

[15] Philip Wadler. The essence of functional programming. In *Principles of Programming Languages (POPL)*, pages 1–14, 1992. Invited talk.

[16] Beta Ziliani, Derek Dreyer, Neelakantan R Krishnaswami, Aleksandar Nanevski, and Viktor Vafeiadis. Mtac: A monad for typed tactic programming in coq. In *ACM SIGPLAN Notices*, volume 48(9), pages 87–100. ACM, 2013.

Towards Formal Reliability Analysis of Logistics Service Supply Chains using Theorem Proving

Waqar Ahmed[1], Osman Hasan[1] and Sofiene Tahar[2]

[1] School of Electrical Engineering and Computer Science
National University of Sciences and Technology
Islamabad, Pakistan
{waqar.ahmad,osman.hasan}@seecs.nust.edu.pk
[2] Department of Electrical and Computer Engineering
Concordia University
Montreal, QC, Canada
tahar@ece.concordia.ca

Abstract

Logistics service supply chains (LSSCs) are composed of several nodes, with distinct behaviors, that ensure moving a product or service from a producer to consumer. Given the usage of LSSC in many safety-critical applications, such as hospitals, it is very important to ensure their reliable operation. For this purpose, many LSSC structures are modelled using Reliability Block Diagrams (RBDs) and their reliability is assessed using paper-and-pencil proofs or computer simulations. Due to their inherent incompleteness, these analysis techniques cannot ensure accurate reliability analysis results. In order to overcome this limitation, we propose to use higher-order-logic (HOL) theorem proving to conduct the RBD-based reliability analysis of LSSCs in this paper. In particular, we present the higher-order-logic formalizations of LSSC with different and same types of capacities. As an illustrative example, we also present the formal reliability analysis of a simple three-node corporation.

1 Introduction

Logistics service supply chain (LSSC) decisions are usually impossible to reverse, and their impact may span several decades. These decisions are very difficult to make given the involvement of several elements of uncertainty, such as changing demand patterns and weather conditions or failing components, associated with these decisions. On the other hand, the reliability of LSSCs, i.e., the ability to perform well when parts of the system fail, is very important as LSSCs are used in many safety-critical applications, such as medicine [14] and space logistics [18]. Moreover, ensuring that the inventory is delivered on time can be of great significance to many companies. Generally, the reliability of a LSSC can be increased by adding more redundancy in it but this choice eventually results in increasing the overall cost, which is also undesirable in many cases. Therefore, it is very important to judge the reliability of the LSSC and its associated cost before development [19]. This kind of reliability analysis is frequently based on Reliability Block Diagrams (RBDs) [23], which are graphical structures consisting of blocks and connectors (lines). The main idea is to represent the structure of the given LSSC in terms of an appropriate RBD [15]. Now, based on this RBD, the reliability characteristics of the overall system can be judged based on the failure rates of individual components, whereas the overall system failure happens if all the paths for successful execution fail.

Traditionally, the RBD-based analysis of LSSC has been done using paper-and-pencil proof methods and computer simulations. Due to the involvement of manual manipulation and simplification, paper-and-pencil proof methods are error-prone and the problem gets more severe

while analyzing large LSSCs. Moreover, it is possible, in fact a common occurrence, that many key assumptions required for the analytical proofs are in the mind of the mathematician and are not documented. These missing assumptions are thus not communicated to the supply chain designers and are ignored in the LSSC implementations, which may also lead to erroneous designs. RBD-based computer simulators, such as ReliaSoft [20] and ASENT [5], generate samples from the exponential and Weibull random variables to model the reliabilities of the sub-modules of the given LSSC. This data is then manipulated using computer arithmetic and numerical techniques to compute the reliability of the complete LSSC. These software are more scalable than the paper-and-pencil proof methods. However, they cannot ensure absolute correctness as well due to the involvement of pseudo-random numbers and numerical methods.

Formal methods [10], which are computer based mathematical reasoning techniques, has been used to overcome the inaccuracy limitations of the paper-and-pencil proof methods and simulation for communication networks. The main idea behind the formal analysis of a system is to first construct a mathematical model of the given system using a state-machine or an appropriate logic and then use logical reasoning and deduction methods to formally verify that this system exhibits the desired characteristics, which are also specified mathematically using an appropriate logic. For instance, Petri nets have been used for the RBD based analysis of a LSSC [15]. The technique has been used to automatically evaluate the reliability of a few node corporations, but the analysis is not scalable for large systems due to the state-space explosion problem [10]. Moreover, generic mathematical RBD relationships cannot be verified using such state-based petri nets techniques, which limits the scope of this approach. Similarly, a Colored Petri Nets (CPN) based tool has been used to model dynamic RBDs (DRBDs) [21], which are used to describe the dynamic reliability behavior of systems. The CPN verification tools, based on model checking principles, are then used to verify behavioral properties of the DRBDs models to identify design flaws [21]. However, due to the state-based model, only state related property verification, like deadlock checks, is supported by this approach and generic reliability relationships cannot be verified.

Higher-order logic [7] is a system of deduction with a precise semantics and can be used to formally model any system that can be described mathematically including recursive definitions, random variables, RBDs, and continuous components. Similarly, interactive theorem provers are computer based formal reasoning tools that allow us to verify higher-order-logic properties under user guidance. The foremost requirement for reasoning about reliability related properties of a LSSC in a theorem prover is the availability of the higher-order-logic formalization of probability theory. Hurd's formalization of measure and probability theories [13] is a pioneering work in this regard. Building upon this formalization, most of the commonly-used continuous random variables [9] and some reliability theory fundamentals [11][1] have been formalized using the HOL theorem prover [22]. However, the foundational formalization of probability theory [13] only supports the whole universe as the probability space. This feature limits its scope in many aspects [16] and one of the main limitations, related to RBD-based analysis, is the inability to reason about multiple continuous random variables [9][11]. Some recent probability theory formalizations [16][12] allow using any arbitrary probability space that is a subset of the universe and thus are more flexible than Hurd's formalization of probability theory. Particularly, Mhamdi's probability theory formalization [16], which is based on extended-real numbers (real numbers including $\pm\infty$), has been recently used to reason about the RBD-based reliability analysis of a series pipelines structure [4] and failure analysis of satellite solar arrays [3], which involves multiple exponential random variables.

In this paper, given the involvement of several elements of continuous and random nature in LSSCs, we propose to conduct the formal RBD-based reliability analysis of a LSSC within

the sound core of a higher-order-logic theorem prover [22]. For this purpose, we plan to build upon the recently proposed higher-order-logic formalization of series RBD, which has been used to conduct reliability analysis of simple oil and gas pipeline [4]. However, this foundational formalization of a series RBD [4] has limited scope and cannot be used to analyze the RBD model of a given LSSC due to the redundancies in these models. The main contribution of this paper is the extension of the series RBD formalization to series-parallel RBD configurations in order to model LSSC scenarios, including the cases when the capacities are different and of same types. For illustration purposes, the paper also presents the formal analysis of a simple LSSC that has been analysed using Petri Nets before [15]. Thanks to the sound reasoning process, the results obtained from the formal reliability analysis of the LSSC scenarios can help design engineers validating the reliability results that are generally obtained through traditional techniques. These accurately determined reliability results can bring many other benefits including trade-off studies for different LSSC designs in order to optimize reliability and cost.

The paper is organized as follows: Sections 2 and 3 present a brief description about the HOL theorem prover and the formalization of probability theory and random variables. Section 4 provides the RBD-based formalization of LSSC scenarios with different and same type of capacities in HOL. Section 5 presents the formal reliability analysis of a three node corporation LSSC by utilizing series and series-parallel RBD configurations. Finally, Section 6 concludes the paper.

2 HOL Theorem Prover

HOL is an interactive theorem prover developed at the University of Cambridge, UK, for conducting proofs in higher-order logic. It utilizes the simple type theory of Church [8] along with Hindley-Milner polymorphism [17] to implement higher-order logic. HOL has been successfully used as a verification framework for both software and hardware as well as a platform for the formalization of pure mathematics.

The HOL core consists of only 5 basic axioms and 8 primitive inference rules, which are implemented as ML functions. Soundness is assured as every new theorem must be verified by applying these basic axioms and primitive inference rules or any other previously verified theorems/inference rules.

Table 1 provides the mathematical interpretations of some frequently used HOL symbols and functions, which are inherited from existing HOL theories, in this paper.

3 Probability Theory and Random Variables

Based on the measure theoretic foundations, a probability space is defined as a triple (Ω, Σ, Pr), where Ω is a set, called the sample space, Σ represents a σ-algebra of subsets of Ω, where the subsets are usually referred to as measurable sets, and Pr is a measure with domain Σ and is 1 for the whole sample space. In the HOL probability theory formalization [16], given a probability space p, the functions `space` and `subsets` return the corresponding Ω and Σ, respectively. Based on this definition, all basic probability axioms have been verified. Now, a random variable is a measurable function between a probability space and a measurable space, which essentially is a pair $(S, \mathcal{A})$, where S denotes a set and $\mathcal{A}$ represents a nonempty collection of sub-sets of S. A random variable is termed as discrete if S is a set with finite elements and continuous otherwise.

HOL Symbol	Standard Symbol	Meaning
$\wedge$	*and*	Logical *and*
$\vee$	*or*	Logical *or*
$\neg$	*not*	Logical *negation*
::	*cons*	Adds a new element to a list
++	*append*	Joins two lists together
`HD L`	*head*	Head element of list L
`TL L`	*tail*	Tail of list L
`EL n L`	*element*	n^{th} element of list L
`MEM a L`	*member*	True if a is a member of list L
λ`x.t`	$\lambda x.t$	Function that maps x to $t(x)$
`SUC n`	$n+1$	Successor of a *num*
`lim(`λ`n.f(n))`	$\lim_{n\to\infty} f(n)$	Limit of a *real* sequence f

Table 1: HOL Symbols and Functions

The probability that a random variable X is less than or equal to some value x, $Pr(X \leq x)$ is called the cumulative distribution function (CDF) and it characterizes the distribution of both discrete and continuous random variables. The CDF has been formalized in HOL as follows [4]:

```
⊢ ∀ p X x.  CDF p X x = distribution p X {y | y ≤ Normal x}
```

where the variables p, X and x represent a probability space, a random variable and a *real* number, respectively. The function `Normal` takes a *real* number as its inputs and converts it to its corresponding value in the *extended-real* data-type, i.e, it is the *real* data-type with the inclusion of positive and negative infinity. The function `distribution` takes three parameters: a probability space p, a random variable X and a set of *extended-real* numbers and outputs the probability of a random variable X that acquires all values of the given set in probability space p.

Now, reliability $R(t)$ is stated as the probability of a system or component performing its desired task over a certain interval of time t.

$$R(t) = Pr(X > t) = 1 - Pr(X \leq t) = 1 - F_X(t) \tag{1}$$

where $F_X(t)$ is the CDF. The random variable X, in the above definition, models the time to failure of the system and is usually modeled by the exponential random variable with parameter λ, which corresponds to the failure rate of the system. Based on the HOL formalization of probability theory [16], Equation (1) has been formalized as follows [4]:

```
⊢ ∀ p X x.  Reliability p X x = 1 - CDF p X x
```

The series RBD, presented in [4], is based on the notion of mutual independence of random variables, which is one of the most essential prerequisites for reasoning about the mathematical expressions for all RBDs. If N reliability events L_i are mutually independent then

$$Pr(\bigcap_{i=1}^{N} L_i) = \prod_{i=1}^{N} Pr(L_i) \tag{2}$$

This concept has been formalized as follows [4]:

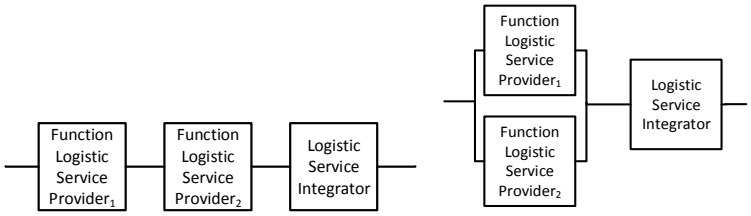

Figure 1: RBDs for the (a) Scenario with Different Types of Capacity (b) Scenario with the Same Type of Capacity

```
⊢ ∀ p L. mutual_indep p L = ∀ L1 n.  PERM L L1 ∧
1 ≤ n ∧ n ≤ LENGTH L ⇒
 prob p (inter_list p (TAKE n L1)) = list_prod (list_prob p (TAKE n L1))
```

The function `mutual_indep` accepts a list of events L and probability space p and returns $True$ if the events in the given list are mutually independent in the probability space p. The predicate `PERM` ensures that its two lists as its arguments form a permutation of one another. The function `LENGTH` returns the length of the given list. The function `TAKE` returns the first n elements of its argument list as a list. The function `inter_list` performs the intersection of all the sets in its argument list of sets and returns the probability space if the given list of sets is empty. The function `list_prob` takes a list of events and returns a list of probabilities associated with the events in the given list of events in the given probability space. Finally, the function `list_prod` recursively multiplies all the elements in the given list of real numbers. Using these functions, the function `mutual_indep` models the mutual independence condition such that for any 1 or more events n taken from any permutation of the given list L, the property $Pr(\bigcap_{i=1}^{N} L_i) = \prod_{i=1}^{N} Pr(L_i)$ holds.

4 Formalization of LSSC in HOL

A LSSC is essentially a service supply chain based on the ability logistics cooperation, which is generally required when the logistics service integrators face shortage in their capacity to deliver services to customers. At this stage, service integrators need to buy the logistics service capacity with functional logistics service providers. There could be a possible scenario where the type of capacity provided by the functional logistics service providers is of multiple (different) nature, such as transport and storage capacity. This scenario is modeled by using a series RBD configuration as shown in Figure 1(a) [15]. In case if the capacity type is the same then this scenario is modeled by using the series-parallel RBD configuration as depicted in Figure 1(b) [15].

In order to formalize the LSSC scenarios in HOL, we firstly present the formalization of series RBD and series-parallel RBD configurations, which are essentially utilized to conduct the reliability analysis of the LSSC, in HOL. If $A_i(t)$ is a mutually independent event that represents the reliable functioning of the i^{th} component of a serially connected system with N components at time t, then the overall reliability of the complete system is [6]:

$$R_{series}(t) = Pr(\bigcap_{i=1}^{N} A_i(t)) = \prod_{i=1}^{N} R_i(t) \tag{3}$$

The above equation can be utilized, by specifying $N = 3$, to evaluate the reliability of the LSSC for the first scenario by modeling it with a series RBD configuration consisting of three reliability blocks, as shown in Figure 1(a). Mathematically, it can be expressed as follows:

$$R_{LSSC\_fst\_scen} = R_{logis\_provdr1} * R_{logis\_provdr2} * R_{logis\_integr} \tag{4}$$

We formalized the corresponding LSSC first scenario series RBD configuration in HOL as:

Definition 1: `⊢ ∀ p logis_provdr1 logis_provdr2 logis_integr. LSSC_series_RBD p [logis_provdr1;logis_provdr2;logis_integr] = inter_list p [logis_provdr1;logis_provdr2;logis_integr]`

The function `LSSC_series_struct` takes a list of events corresponding to the failure of LSSC system components, i.e., *logis_provdr*1, *logis_provdr*2 and *logis_integr*, and the probability space p and returns the series structure event of the complete LSSC system. The function `inter_list` returns the intersection of all of the elements of the given list and the whole probability space, if the given list is empty.

We formally verified the reliability expression for the first scenario, given in Equation 4, representing different capacity types, shown in Figure 1(a), in HOL as follows:

Theorem 1:
```
⊢ ∀ p logis_provdr1 logis_provdr2 logis_integr.  prob_space p ∧
(∀ x'.  MEM x' [logis_provdr1;logis_provdr2;logis_integr] ⇒
x' ∈ events p) ∧
mutual_indep p [logis_provdr1;logis_provdr2;logis_integr] ⇒
 prob p (LSSC_series_struct p [logis_provdr1;logis_provdr2;logis_integr] =
 list_prod (list_prob p [logis_provdr1;logis_provdr2;logis_integr])
```

The first assumption ensures that p is a valid probability space based on the probability theory in HOL [16]. The next two assumptions guarantee that the list of events, representing the reliability of LSSC components, must be in the events space p and the reliability events are mutually independent. The conclusion of Theorem 1 models the series RBD configuration of LSSC first scenario with different capacity.

Similarly, in the series-parallel RBD configuration, if $A_{ij}(t)$ is the event corresponding to the reliability of the j^{th} component connected in a i^{th} subsystem at time t, then the reliability of the complete system can be expressed as follows:

$$R_{series-parallel}(t) = Pr(\bigcap_{i=1}^{N} \bigcup_{j=1}^{M} A_{ij}(t)) = \prod_{i=1}^{N} (1 - \prod_{j=1}^{M} (1 - R_{ij}(t))) \tag{5}$$

The above equation can be used to obtain the reliability of LSSC for the second scenario, which is modeled by a series-parallel RBD configuration, as shown in Figure 1(b). Mathematically, the reliability of this second scenario is as follows:

$$R_{LSSC\_snd\_scen} = (1 - (1 - R_{logis\_provdr1}) * (1 - R_{logis\_provdr2})) * (1 - (1 - R_{logis\_integr})) \quad (6)$$

The HOL formalization of Equation 6 is as follows:

Definition 2: `⊢ ∀ p logis_provdr1 logis_provdr2 logis_integr. LSSC_series_parallel_struct p [[logis_provdr1;logis_provdr2];logis_integr]= series_struct p (parallel_struct_list [[logis_provdr1;logis_provdr2];logis_integr])`

The function `LSSC_series_parallel_struct` accepts a two dimensional list, i.e., a list of lists, along with a probability space p and returns the corresponding reliability event of the system constituted from the series connection of the parallel stages. The function `series_struct` is used to model the series connection while the function `parallel_struct_list` is used to model the parallel stages. The function `parallel_struct_list` takes a two dimensional list of events along with probability space p and returns a single dimensional list of events by mapping the `inter_list` function, already explained in Definition 1, on each element of the given two dimensional event list.

Now, the reliability expression for the series-parallel RBD configuration of the LSSC, which corresponds to the second scenario with same capacity type, given in Equation 6, can be verified as the following HOL theorem:

Theorem 2:
```
⊢ ∀ p logis_provdr1 logis_provdr2 logis_integr.  (prob_space p) ∧
mutual_indep p FLAT([[logis_provdr1;logis_provdr2];logis_integr]) ∧
(∀x'.  MEM x' [logis_provdr1;logis_provdr2;logis_integr]) ⇒
x' ∈ events p) ⇒
 prob p
  (LSSC_series_parallel_struct p [[logis_provdr1;logis_provdr2];logis_integr] =
 list_prod (one_minus_list
  (list_compl_rel_list_prod p [[logis_provdr1;logis_provdr2];logis_integr]))
```

where *logis_provdr*1, *logis_provdr*2 and *logis_integr* are the reliability events associated with the logistic service providers and integrator, respectively. The function `list_compl_rel _list_prod` accepts a two-dimensional list of events, representing the time to failure of individual components connected in a series-parallel structure along with the probability space p and returns a list, which is the product of complement reliabilities of the components connected in parallel. The functions `list_prod`, `one_minus_list` and `list_prob` are used to model the product of reliabilities, complement of reliabilities, and the events corresponding to the component functioning reliably at the desired time, respectively. The assumptions of Theorem 2 are similar to the ones used in Theorem 1.

5 Case Study: A Three Node Corporation LSSC

In order to formally verify the reliability expression of a LSSC used in a typical three node corporation, we first need to formally model the reliability events that are associated with its logistic service providers and integrator. A reliability event list constructed from the list of random variables can be formalized in HOL is as follows:

Definition 3: $\vdash \forall$ p x. rel_event_list p [] x = [] $\wedge$
$\forall$ p x h t.rel_event_list p (h::t) x =
PREIMAGE h {y | Normal x < y} $\cap$ p_space p :: rel_event_list p t x

The function rel_event_list accepts a probability space p, a list of random variables, representing the failure time of individual components, and a real number x, which represents the time index at which the reliability is desired. It returns a list of events, representing the proper functioning of all individual components at time x.

Definition 4: $\vdash \forall$ p L x. List_rel_event_list p L x =
MAP (λa. rel_event_list p a x) L

The function List_rel_event_list accepts a probability space p, a list of random variables, representing the failure time of individual components, and a real number x, which represents the time index at which the reliability is desired. It returns a two dimensional list of events by mapping the function rel_event_list on every element of the given two dimensional list of random variables, which in turn models the proper functioning of all individual components at time x.

We consider that the reliability of each LSSC component connected in RBD configurations, as shown in Figure 1, is exponential distributed. The HOL formalization of the exponential distribution predicate, which models the failure behavior of LSSC components, is as follows:

Definition 5: $\vdash \forall$ p X l. exp_dist p X l =
$\forall$ x. (CDF p X x = if 0 $\leq$ x then 1 - exp (-l * x) else 0)

The function exp_dist guarantees that the CDF of the random variable X is that of an exponential random variable with a failure rate l in a probability space p. We classify a list of exponentially distributed random variables based on this definition as follows:

Definition 6: $\vdash \forall$ p L. list_exp p [] L = T $\wedge$
$\forall$ p h t L. list_exp p (h::t) L = exp_dist p (HD L) h $\wedge$ list_exp p t (TL L)

The function list_exp accepts a list of failure rates, a list of random variables L and a probability space p. It guarantees that all elements of the list L are exponentially distributed with the corresponding failure rates, given in the other list, within the probability space p. For this purpose, it utilizes the list functions HD and TL, which return the *head* and *tail* of a list, respectively. Next we model a two dimensional list of exponential distribution functions to model nodes connected in a series-parallel RBD as follows:

Definition 7: $\vdash$ ($\forall$ p L. list_list_exp p [] L = T) $\wedge$
$\forall$ h t p L. list_list_exp p (h::t) L =
list_exp p h (HD L) $\wedge$ list_list_exp p t (TL L)

The function list_list_exp accepts two lists, i.e., a two dimensional list of failure rates and random variables L, corresponding to the components at each stage of a series-parallel RBD. It calls the function list_exp recursively to ensure that all elements of the list L are exponentially distributed with the corresponding failure rates, given in the other list, within the probability space p.

The reliability of the first scenario of LSSC, modeled by a series RBD configuration and each component reliability is represented by exponential distribution, can be expressed as:

$$R_{LSSC\_fst\_scen}(t) = e^{(\lambda_{logis\_provdr1}+\lambda_{logis\_provdr2}+\lambda_{logis\_integr})t} \quad (7)$$

where the λ terms in the above equation represent the failure rates of logistic service providers and integrators.

Now, based on Equation (7), we carried out the formal reliability analysis of the first scenario of LSSC, given in Figure 1(a), in HOL and the resulting theorem is as follows:

Theorem 3: ⊢ ∀ X_logis_provdr1 X_logis_provdr2 X_logis_integr C_logis_provdr1 C_logis_provdr2 C_logis_integr p t.
0 ≤ t ∧ prob_space p ∧
(∀x'. MEM x'
rel_event_list p [X_logis_provdr1;X_logis_provdr2;X_logis_integr] t ⇒
x' ∈ events p) ∧
mutual_indep p
(rel_event_list p [X_logis_provdr1;X_logis_provdr2;X_logis_integr] t) ∧
list_exp p [C_logis_provdr1;C_logis_provdr2;C_logis_integr]
[X_logis_provdr1;X_logis_provdr2;X_logis_integr] ⇒
prob p (series_struct p
(rel_event_list p [X_logis_provdr1;X_logis_provdr2;X_logis_integr] t) =
exp (-list_sum [C_logis_provdr1;C_logis_provdr2;C_logis_integr]*t)

where the function list_sum returns the sum of all the elements of the given failure rate list. The first assumption ensures that the variable t models time as it can acquire positive integer values only. The next assumption ensures that p is a valid probability space based on the probability theory in HOL [16]. The next two assumptions ensure that the events corresponding to the failures modeled, by the random variables $X\_logis\_provdr1$, $X\_logis\_provdr2$ and X_logis_integr are valid events from the probability space p and they are mutually independent. Finally, the last assumption assigns the random variables $X\_logis\_provdr1$, $X\_logis\_provdr2$ and X_logis_integr, as exponential random variables with failure rates $C\_logis\_provdr1$, $C\_logis\_provdr2$ and C_logis_integr, respectively. The conclusion of Theorem 3 represents the desired reliability expression.

Similarly, the reliability of the second scenario of LSSC with exponential failure distribution, shown in Figure 1(b), can be expressed as:

$$R_{LSSC\_snd\_scen}(t) = (1-(1-e^{(\lambda_{logis\_provdr1}t)} * (1-e^{(\lambda_{logis\_provdr1}t)})) * (1-(1-e^{\lambda_{logis\_integr}t})) \quad (8)$$

We formally verified the above equation in HOL as follows:

Theorem 4: ⊢ ∀ X_logis_provdr1 X_logis_provdr2 X_logis_integr C_logis_provdr1 C_logis_provdr2 C_logis_integr p t.
(0 ≤ t) ∧ (prob_space p) ∧
mutual_indep p (FLAT
(List_rel_event_list p [[X_logis_provdr1;X_logis_provdr2];X_logis_integr] t)) ∧
list_list_exp p([[C_logis_provdr1;C_logis_provdr2];C_logis_integr])
([[X_logis_provdr1;X_logis_provdr2];X_logis_integr]) ⇒
prob p (LSSC_series_parallel_struct p
(list_rel_event_list p [[X_logis_provdr1;X_logis_provdr2];X_logis_integr] t)) =
list_prod (one_minus_list
(list_exp_func_list ([[C_logis_provdr1;C_logis_provdr2];C_logis_integr]) t)

where the functions `list_prod` and `list_exp_func_list` accept a two-dimensional list of failure rates and return a list with products of one minus exponentials of every sub-list. For example, `list_exp_func_list [[c1; c2; c3]; [c4; c5]; [c6; c7; c8] x = [1 - exp -(c1+c2+c3) x; 1 - exp -(c4+c5) x; 1 - exp -(c6+c7+c8) x]`. The assumptions of Theorem 4 are quite similar to the ones used in Theorem 3. The proofs of Theorems 3 and 4 involves Theorems 1 and 2 and some basic probability theory axioms and some properties of the exponential function `exp`. The reasoning process took about 2000 lines of HOL script [2] with dedicated probability-theoretic guidance. The first LSSC scenerio reliability analysis is mainly carried out by using the series RBD formalization, which is presented in [4]. However, the major part of the effort was put into the formalization of generic series-parallel RBD configurations. This formalization facilitated the formalization of second scenario of LSSC, considerably as the analysis only took about 650 of HOL code.

The distinguishing features of the formally verified Theorems 3 and 4, compared to the reliability analysis of the LSSC scenarios of Figure 1 using Petri Nets [15], includes its generic nature, i.e., all the variables are universally quantified and thus can be specialized to obtain the reliability of any number of logistic providers and integrators for any given failures rates. The guaranteed correctness of the theorems is due to the involvement of a sound theorem prover in their verification, which ensures that all the required assumptions for the validity of the result are accompanying the theorems. To the best of our knowledge, the above-mentioned benefits are not shared by any other computer based reliability analysis approach.

6 Conclusions

The accuracy of reliability analysis of LSSC is a dire need these days due to their extensive usage in safety-critical applications, where an incorrect reliability estimate may lead to disastrous situations including the loss of innocent lives. In this paper, we presented a higher-order-logic formalization of commonly used RBD configurations, i.e., series and series-parallel, to facilitate the formal reliability analysis of LSSC within a theorem prover. The commonly used LSSC RBDs are also formalized and we illustrated the usefulness of the proposed idea by considering a small application. In future, we plan to formally analyze the reliability of larger LSSC models.

Acknowledgments

This publication was made possible by NPRP grant # [5 - 813 - 1 134] from the Qatar National Research Fund (a member of Qatar Foundation). The statements made herein are solely the responsibility of the author[s].

References

[1] N. Abbasi, O. Hasan, and S. Tahar. An Approach for Lifetime Reliability Analysis using Theorem Proving. *Journal of Computer and System Sciences*, 80(2):323–345, 2014.

[2] W. Ahmad. Towards Formal Reliability Analysis of Logistics Service Supply Chains using Theorem Proving, Proof Script. http://save.seecs.nust.edu.pk/projects/rbd/LSSC, 2015.

[3] W. Ahmad and O.Hasan. Towards the Formal Fault Tree Analysis using Theorem Proving. In *Conferences on Intelligent Computer Mathematics*, LNAI, pages 39–54. Springer, 2015.

[4] W. Ahmed, O. Hasan, S. Tahar, and M. S. Hamdi. Towards the Formal Reliability Analysis of Oil and Gas Pipelines. In *Intelligent Computer Mathematics*, volume 8543 of *LNCS*, pages 30–44. Springer, 2014.

[5] ASENT. https://www.raytheoneagle.com/asent/rbd.htm, 2015.

[6] R. Bilintion and R.N. Allan. *Reliability Evaluation of Engineering System.* Springer, 1992.

[7] C.E. Brown. *Automated Reasoning in Higher-order Logic.* College Publications, 2007.

[8] A. Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[9] O. Hasan and S. Tahar. Formalization of the Continuous Probability Distributions. In *Automated Deduction*, volume 4603 of *LNAI*, pages 3–18. Springer, 2007.

[10] O. Hasan and S. Tahar. Formal Verification Methods. In *Encyclopedia of Information Science and Technology*, pages 7162–7170. IGI Global, 2014.

[11] O. Hasan, S. Tahar, and N. Abbasi. Formal Reliability Analysis using Theorem Proving. *IEEE Transactions on Computers*, 59(5):579–592, 2010.

[12] J. Holzl and A. Heller. Three Chapters of Measure Theory in Isabelle/HOL. In *Interactive Theorem Proving*, volume 6172 of *LNCS*, pages 135–151. Springer, 2011.

[13] J. Hurd. *Formal Verification of Probabilistic Algorithms.* PhD Thesis, University of Cambridge, UK, 2002.

[14] A.T. Kearny. Improving the Medical Supply Chain. `http://www.atkearneypas.com/knowledge/publications/2004/Medicines_Monograph_S.pdf`, 2004.

[15] Y. Li and H. Yi. Research on the Inherent Reliability and the Operational Reliability of the Supply Chain. *u- and e-Service, Science and Technology*, 7(1):104–112, 2014.

[16] T. Mhamdi, O. Hasan, and S. Tahar. On the Formalization of the Lebesgue Integration Theory in HOL. In *Interactive Theorem Proving*, volume 6172 of *LNCS*, pages 387–402. Springer, 2011.

[17] R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:348–375, 1977.

[18] MIT Strategic Engineering. Interplanetary Supply Chain Network for Space Exploration. http://strategic.mit.edu/spacelogistics/, 2015.

[19] D. Mu and Z. Du. Research on the Inherent Reliability and the Operational Reliability of the Supply Chain. *Logistics Technology*, 12(37), 2004.

[20] ReliaSoft. http://www.reliasoft.com/, 2015.

[21] R. Robidoux, H. Xu, L. Xing, and M. Zhou. Automated Modeling of Dynamic Reliability Block Diagrams Using Colored Petri Nets. *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans*, 40(2):337–351, 2010.

[22] K. Slind and M. Norrish. A Brief Overview of HOL4. In *Theorem Proving in Higher-order Logics*, volume 5170 of *LNCS*, pages 28–32. Springer, 2008.

[23] J. Soszynska. Reliability and Risk Evaluation of a Port Oil Pipeline Transportation System in Variable Operation conditions. *International Journal of Pressure Vessels and Piping*, 87(2-3):81–87, 2010.